

MICRO-MONITOR

An Embedded System Boot Platform

Introduction and Command Set

Revision date: November 11, 2002

MicroMonitor Command Set

Table of Contents

This document serves as an introduction to the monitor, covering some basics on what the monitor is trying to do, and how it works as a platform for an embedded system application. This also includes some of the capabilities of the monitor's command line as well as a full listing of the monitor's command set.

INTRODUCTION TO MICROMONITOR AND TFS.....	4
WHAT IS AN "EMBEDDED SYSTEM BOOT PLATFORM"?	4
FREQUENTLY ASKED QUESTIONS	5
WHAT IS THE PURPOSE OF THIS MICROMONITOR DISTRIBUTION?	5
I AM A NEW USER, WHAT DO I NEED TO KNOW?	5
THERE ARE A FEW .HTML PAGES & .C FILES REFERENCED, BUT NOT AVAILABLE. WHY?.....	5
HOW MUCH MEMORY SPACE IS TAKEN UP BY THE MONITOR ON THE TARGET?	5
HOW MANY FLASH DEVICES ARE NEEDED TO RUN MICROMONITOR WITH TFS?.....	5
CAN I USE THIS MONITOR IF I DON'T HAVE ANY FLASH IN MY SYSTEM?.....	6
CAN THE MONITOR PROCESS COMMANDS WHILE INTERACTING WITH REMOTE TFTP CLIENT?.....	6
THE MAKE FILES REFERENCE TOOLS THAT I CAN'T FIND. WHERE ARE THEY?.....	6
WHAT DO I NEED TO BE AWARE OF WHEN PUTTING AN APPLICATION ON TOP OF THE MONITOR?.....	6
WHERE CAN I GET A CROSS COMPILER?.....	7
WHAT ARE THE PLANS FOR FUTURE PORTS OF MICROMONITOR?	7
HOW MUCH DOES THE AUTHOR CHARGE FOR A NEW PORT OF THE MICROMONITOR?	7
CAN I CONTRIBUTE?.....	7
WHO WROTE THIS?	8
USING SHELL VARIABLES AND SYMBOLS.....	9
SHELL VARIABLE USAGE	9
SHELL VARIABLES VS. SYMBOLS	9
SHELL VARIABLES CREATED AND/OR USED BY MICROMONITOR	11
WRITING MICROMONITOR SCRIPTS.....	15
SCRIPT INVOCATION.....	15
SCRIPT-SPECIFIC COMMANDS	15
SCRIPT NESTING	15
EXAMPLE #1: CLEANUP	16
EXAMPLE #2: PING.....	16
EXAMPLE #3: NAMELIST	17
EXAMPLE # 4: NAMELIST USING "ITEM" COMMAND.....	17
EXAMPLE # 5: PROCESSING A VARIABLE NUMBER OF COMMAND LINE ARGUMENTS	18
EXAMPLE # 6: WHY WOULD YOU EVER WANT TO DO THIS??.....	18
EXAMPLE # 7: STARTUP SCRIPT USING SUBROUTINES, IF/ELSE AND FILE DECOMPRESSION	18
OVERRIDING THE DEFAULT COMMAND INTERPRETER USED BY THE SCRIPT RUNNER.....	20
USER LEVELS	21

DESCRIPTION	21
A USER LEVEL EXAMPLE	21
COMMAND LINE INTERFACE OUTPUT REDIRECTION	23
MICROMONITOR COMMAND SET:	24
ARGV	25
ARP	26
AT	27
CALL	29
CAST	30
CM	32
DHCP	33
DM	37
ECHO	39
EDIT	40
ETHER	43
EXIT	44
FLASH	45
FM	48
GO	49
GOSUB	50
GOTO	51
HEAP	52
HELP	55
HISTORY	56
ICMP	57
IDEV	58
IF	59
ITEM	61
MSTAT	62
MT	63
PIO	64
PM	65
READ	66
REG	67
RESET	68
RETURN	69
SET	70
SLEEP	71
SM	72
SS	73
STRACE	74
TFS	75
TFTP	79
ULVL	82
VERSION	83
XMODEM	84

Introduction to MicroMonitor and TFS...

The monitor is the firmware that the CPU executes immediately after a reset or power-up. It resides in the non-volatile flash memory of the target system. It is responsible for "starting up" or booting the CPU and getting it to a point where a user can access the target through some interface (typically either RS-232 or ethernet). The capabilities provided at this interface depend on what capabilities have been configured into the monitor when it was built (refer to *What comes with Micro Monitor?* for a brief list of capabilities).

Once the monitor does some initialization of the target, it presents itself as a command line interface to the user typically through an RS-232 port (also through UDP if an ethernet interface is included in the hardware). The command set includes basic capabilities like memory display and modification, parallel IO control and command line editing/history. In addition to this, the monitor configures part of the flash to be used as a file system (TFS: tiny file system). This basically means that code can access flash as name space instead of address space. Adding this basic capability opens up a whole new set of ideas applicable to the embedded target. Files can be set up with different attributes. Among other things, these attributes let the monitor know what to do with the files at startup. If a file is marked as "auto-bootable", then the monitor will automatically execute that file at startup. The executable file can be either a binary (COFF, ELF or AOUT) that will be loaded into DRAM and executed, or it can be an ascii script of commands that may, for example, configure the target to run DHCP or BOOTP, or may assign a fixed IP and MAC address to the target (refer to description of scripts within MicroMonitor). Multiple files can be configured for autoboot and the monitor will automatically run them in alphabetical order regardless of the order in which they actually exist in the flash. Additionally, TFS and the monitor share the concept of user levels. When the target first starts up, it is running at the highest user level (similar to UNIX super-user or Windows administrator). Once control is turned over to the individual TFS files, the user level can be adjusted so that certain portions of the system (both files and commands) are only accessible by certain user levels. The user level access is controlled through encrypted passwords that are stored in a TFS file. If somehow the passwords are lost or inaccessible, there is a "backdoor" password that is derived from the MAC address of the target.

Everything is a file, even the main application. It's just another file in TFS. When the application is running (as a result of the monitor loading it from TFS flash to DRAM), other files can be accessed by the active application. This makes it convenient to build one application binary that configures itself based on a few locally editable ascii files in TFS. Also, since the application executes out of DRAM (rather than directly out of the flash in TFS), a new application can be loaded into flash while the current application in DRAM is left running.

As you can see, the monitor is very heavily linked to TFS. As a result, many of the other commands within the monitor assume the existence of a file system. Xmodem and TFTP both interface to the file system so uploads and downloads can all be name based instead of address based. This makes life a lot easier, and it makes applications that reside on top of the monitor a lot easier to manage.

What is an "embedded system boot platform"?

In this context, an embedded system boot platform is a target-resident environment that provides the developer with a suite of capabilities that enhance the development process and potentially, the environment provided to the application being developed. In its simplest form, it is used to speed up some of the early stages of embedded system development by providing a simple file system, tftp/xmodem interfaces for application transfer and an array of additional commands and capabilities that provide a variety of enhancements to the development environment. As a part of a development strategy for an organization, it provides a common base platform; an integral part of the application itself, providing the system with a core set of features that are generic in nature, and should be useable by application code regardless of the operating system chosen.

Frequently Asked Questions

What is the purpose of this MicroMonitor distribution?

The reasons for this distribution are similar to those of other open source projects... The more the merrier. The maturity and extensibility of a software package is definitely related to how much it is used and the types of environments it is used in. The goal is to broaden the user base with the hope that those folks will generate bug reports (hey, and maybe even fix some!), generate new features (hey, and maybe even write the code for them!) and generate good constructive feedback. In the end, it benefits all.

I am a new user, what do I need to know?

There are a few different places you may have gotten this package from. If this is part of the CD-ROM on the book "Embedded Systems Firmware Demystified" or if you downloaded the package from Lucent's Research Software Distribution Web Site you have a "real" copy. There were a few other sites that this was available on early in its public release, but these are the places to get the most recent distribution. Read and be aware of the license. In summary, it is a very relaxed license that says "use it at your own risk". Aside from that, the only thing to do is spend some time reading through the text. Become familiar with some of the monitor's capabilities and limitations to decide if it fits your needs. The monitor has been used on several projects in and out of Lucent. It is a mature firmware package that provides an embedded system with a lot of useful and generic (OS & generally CPU independent) features. If you have comments or questions, contact the author. Finally, be aware that despite its maturity, this is an unsupported package, so no guarantees of functionality or support are made (refer to the license).

There are a few .html pages & .c files referenced, but not available. Why?

This is a package of descriptive .html pages and c/assembly code that Lucent has allowed me to make available to folks outside of the company. In doing this, some of the files in the package were determined to be inappropriate for inclusion in the package available outside of Lucent. Some of the files are not included because they are available through other sources, so contact the author for more information on this.

How much memory space is taken up by the monitor on the target?

Exact sizes are tough because it depends on what parts of the monitor you need/want in your system. If you look in one of the config.h files you will see a lot of INCLUDE_XXX macros. Each of those macros (described in inc_check.h) controls the inclusion of some facility or capability in the monitor package. At the low end it can conveniently be cut down to about 64K and the high end is about 256K right now. The low end is "conveniently" cut to 64K with these macros. If there is more trimming needed, then other chunks could also be removed, but they have not been separated out with INCLUDE_XXX macros because no one has needed the footprint to be that small.

On the RAM side, that too depends on what you want. It can range from about 16K to 64K of RAM typically. The two big users of memory here are ethernet and zlib; so, again, it depends on what you need. In the case of zlib, the memory needs are through malloc(), so the size of the heap would have to be adjusted.

How many flash devices are needed to run MicroMonitor with TFS?

Actually, a target system can run with zero flash devices and TFS can be configured to run out of RAM space (this assumes, of course, that the target boots out of EPROM or some other non-volatile memory). Typically however, the target needs only one flash device to run the monitor with flash file system. Additional devices can be used, but they are not needed. There is a page titled "Memory Layout" that talks about this.

Can I use this monitor if I don't have any flash in my system?

Yes. TFS (Tiny File System) typically expects files to be in flash; however, the flash drivers can be configured so that one of the TFS devices (there can be several configured) is a block of RAM. If the RAM is not battery-backed, then it is up to the monitor startup code to guarantee that the RAM space is properly initialized. Refer to the TFS-In-RAM appnote for configuration details.

Can the monitor process commands while interacting with remote TFTP client?

Yes, but there are limitations. When idle, the monitor is sitting in a function that waits for characters to come in from the console port. It is also polling its ethernet interface for incoming packets. If an incoming packet starts up some TFTP file transfer, then for each packet the console will be polled, so commands will be processed. Note that things are slowed down simply because there are essentially two processes going on, and there are some obvious limitations to this... If a tftp transfer is in progress, then be reasonable... If you try to kick off some other TFTP transaction you will cause confusion; if you try to write to the same memory that TFTP is using you will cause confusion; etc... As a rule, the monitor code is single threaded; however, some "multi-threaded-like" capabilities will work simply because of the way the interfaces are polled.

The make files reference tools that I can't find. Where are they?

First of all, there are a few different build environments depending on what you plan to build and where you plan to use what you build. Everything in the package that is cross-compiled uses gnu-cc with a BASH shell environment on a Win32 PC. The make files contain things like "rm", "cp" and "find" because they run within a bash shell and assume that there is a set of unix-like commands that are installed on the host system. You can hunt around on the web and find a lot of this stuff for free, or you can go to microcross.com and purchase a set of pre-built GNU cross-compilers, plus the BASH environment (for PC or UNIX) for a very reasonable price. Aside from the GNU stuff there are other tools, like moncmd, newmon, ttftp, defdate, elf, aout, coff etc... The Win32 .exe files for these tools are under tools/win95 directory. To build these tools for other environments, extract the source from the tar file called toolsrc.tar. Refer to the individual subdirectories. I have ported them to Sun Solaris, UNIXWARE (x86), LINUX and W95/98/NT. If you port to something else and make changes, please send them back for all to benefit.

What do I need to be aware of when putting an application on top of the monitor?

One of the most important features of MicroMonitor is that it was designed to be extremely independent of the underlying hardware (within reason of course) and the overlaying application with its RTOS. Despite this, there are still things that must be considered when putting an application on top of the monitor. The most important thing to be aware of regarding an application that is launched by the monitor is that each of these programs (the monitor and the application) are independently linked to exist within one shared block of memory space. If, after the monitor loads the application into RAM and turns over control to it, the application has no need to use any of the monitor's facilities, then it is safe to allow the application to use the RAM space that was previously used by the monitor. This; however, is not typically the case. Usually, the application comes up and it "connects" (see Application-to-Monitor hookup examples) to the monitor and uses the monitor's hooks to TFS and some other facilities. In this case, the application must be aware of the fact that the monitor is using some of the RAM/Flash space. The application must be built with a memory map that does not overlap with the monitor's memory map. In most cases, the monitor is built and a shell variable (\$APPRAMBASE) is loaded with the memory address that is the first location in RAM that can be used by the application; so, in many cases the application is built to reside starting at this point or some point higher.

This clash between monitor RAM space and application RAM space is usually avoided by simply adjusting the memory map of the application to be outside the range of space used by the monitor. In some cases (Linux, for example), the application assumes it will exist at a certain point in memory, so the monitor must be appropriately built to avoid the clash. In other cases (some versions of eCOS, for example), the adjustment of the memory map is done, but there are still some hard-coded assumptions in the application that certain areas of memory space can be used for RTOS-specific needs. This gets a bit

trickier to figure out because at the high level it looks like the maps do not clash (because the link-editor files are configured properly), but there is still some code within the application with a hard-coded address.

Where can I get a cross compiler?

All the source code in the world is no good if you can't compile it! In almost all cases, I use the GNU stuff. Never did any formal comparison of GNU vs other vendors, but you just can't beat the price. If you're really feeling energetic, there are several sites on the web that you can refer to for instructions on how to build the GNU cross-compilation environment from scratch, and for free. On the other hand, if you want a really nice set of pre-built GNU-based cross compilers, check out microcross.com for both PC and UNIX environments.

I've converted all of my monitor make files over to use this toolset. It includes cross-compilation tools for all of the GNU-GCC cross compilers, plus, for PCs, the Bash shell and full suite of unix-like tools are included. Check it out!

What are the plans for future ports of MicroMonitor?

Porting the monitor to a new platform is relatively straightforward (see "Porting to a New Target"). The ports made available through this package depend on a few things:

- Requests from users.
- Availability of an off-the-shelf platform.
- Other folks that contribute ports back to this package.

Note that I will respond to requests. The recent release of the FADS860 port was due to a large number of folks interested; however, I am hoping that folks are willing to do this on their own, and return the ported environment to me for distribution with the package.

How much does the author charge for a new port of the MicroMonitor?

In a perfect world, I would charge as much as I want, and you would be glad to pay me! In the real world, I do externally requested ports in my spare time for free, so when I commit to a new target port, realize that it is still potentially months away.

Can I Contribute?

NO. I want to do all the work by myself on weekends when it is really sunny outside and my son wants to have a catch! Chuckle! Of course! Please do! There are several ways you can contribute...

- Read this FAQ and the included documentation prior to emailing questions. Since you are reading this, then you are already a contributor. Thanks!
- Send me comments. Constructive feedback is very useful. Whether you are going to send "atta-boy" or "whatta-idiot" comments is up to you. If you have suggestions for change, then please make them clear. Don't just tell me that you don't like my coding style. Tell me why a different method is better.
- Bug reports are the primary reason for this distribution. Would be even better if you sent me back the fix!
- New ideas or enhancements are the secondary reason for this distribution. If you have ideas for improvement or just new stuff, I want to hear that too. Can't promise that it will be incorporated in, but that doesn't mean it wasn't a good idea.

Fame is the result! Yes, you too, could have your name added to the MicroMonitor "contributors" list. Boy, that's just gotta get you excited! *Yea, I know, I need a vacation.* Seriously though, the more folks that

contribute and are willing to have their name added to the contributors list, the more the load is distributed across other folks besides me. Yes, I will ask for your email address as a contributor so others can contact you to ask questions about something you have done.

Who wrote this?

This monitor package originated by me, Ed Sutter (esutter@lucent.com), and in many places both in this .html 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. On the other hand, if you don't like this package, or you just think it stinks, then I didn't write any of it!

Using Shell Variables and Symbols

The monitor supports the ability to assign data to or retrieve data from shell variables. The text below discusses the implementation of shell variables and symbols within the monitor as well as how they can be used and what variables are automatically created and/or used by the monitor itself.

Shell Variable Usage

Similar to most other shells, shell variable names can contain alphanumeric characters and the underscore ('_'). Once assigned, the value within the variable can be accessed by preceding the variable name with a dollar sign (\$). The `${}` syntax is also supported. Use of the curly braces tells the command line processor to force the start and end point of a shell variable name that may otherwise not be seen as a variable because it is embedded within a larger string of characters. In addition, the dollar sign can be preceded by a backslash to negate its meaning as a shell variable starting delimiter. The set command supports the ability to assign values to variable names and perform some basic operation on them.

The MicroMonitor command interpreter supports nested shell variable names. This means that the user can build a shell variable name based on other shell variables. For example, if `NAME_1` contains "Jane", `NAME_2` contains "John" and `IDX` contains "1" then the statement `echo ${NAME_${IDX}}` will output the string "Jane"; likewise if `IDX` contains "2" then that same statement will output the string "John". This mechanism of processing nested shell variables within the command line interface allows the user to conveniently use shell variables to represent n-dimensional arrays, and by default, only the space used by the individual members within each dimension take up memory space. For example, assume we have an application that is represented by 3 entries in a table and the entries may exist in any of the first 100 locations of the table. When the entries are created they use the index into the table as part of their name: `TBL3`, `TBL48`, `TBL99`. Then the user can "index" into the table with a second `IDX` variable... `${TBL${IDX}}` where `IDX` is set to 3, 48 or 99 will retrieve the content of each variable. Note that although the application appears to have a table that contains at least 100 entries, the shell actually only assigns those that are used. A multi-dimensional array is handled the same way... `PLOT[3][4][5]` can be represented with the shell variable `PLOT345 ... ${PLOT${X}${Y}${Z}}`, where `X=3`, `Y=4` and `Z=5`.

For examples of this usage refer to the "*namelist*" and "*why would you ever want to do this?*" script examples.

Shell Variables vs. Symbols

The command line processor makes token substitutions based on shell variable assignment and symbol table entries. As previously mentioned, shell variables are identified by preceding a string with a dollar sign (\$). Symbol table entries are identified by preceding a string with a percent sign (%). The use of shell variables and symbols is similar, but the way in which they are stored is different; hence, either may be appropriate depending on the situation.

A shell variable is created on the command line of the monitor using the set command or through the API using the `setenv()` function. Retrieval of the content on the command line is done by specifying the variable name preceded by a dollar sign (\$) or through the API using the `getenv()` function. Refer to the discussion on scripts for some examples of shell variable usage.

A symbol is not created, it exists or it doesn't exist based on the content of a symbol table file in TFS. Retrieval of the content of a symbol on the command line is done by specifying the symbol name preceded by a percent sign (%). There is no "getenv()-like" capability for symbols. The default name for the symbol table file is "symtbl". At compile time, this can be overridden in `config.h`, or at runtime if the shell variable `SYMFILE` is set, then the content of that shell variable is used instead of the default "symtbl". The format of the symbol table file is simple, each line contains one symbol with a value that the symbol represents (each of which are whitespace delimited). For example, if the following lines were in the file "symtbl"...

```
main      0x14004
```

func 0x14800

then if %main was on the command line, it would be replaced with 0x14004; likewise if %func was on the command line, it would be replaced with 0x14800. This symbol-table querying ability allows the stand-alone monitor to provide symbolic querying of the application without the need for a host-resident debugger; plus, when it is no longer needed, the symbol file can simply be removed from TFS. As an example of symbol table substitution, the following command line `dm -4d %symname` would be converted to something like `dm -4d 0x401008` if the symbol table file existed and there was an entry for `symname`. Likewise, the following command line `arp $HOSTIP` would be converted to `arp 135.3.94.39` if the shell variable `HOSTIP` was previously set to `135.3.94.39`.

The primary difference between shell variables and symbols is the way in which they are stored. Shell variables are allocated onto the heap so they use up RAM, symbol table entries are part of a file, so no RAM is involved. Also, future versions of MicroMonitor may support more complex symbol retrieval by allowing a symbol to have members (a structure).

NOTE1: The command interpreter can be told not to process the dollar sign by preceding it with the backslash (`\`). For example, assuming the shell variable `$ABC` is set to "hello", the command line `echo \ $ABC = $ABC` could be used to output `$ABC = hello`.

NOTE2: At the point in time when the command interpreter is making the substitution of the value of a variable in place of the variable name itself, if the variable does not exist, then no substitution is made. For example, if the shell variable `ABC` was previously set to "hello" then the command line `echo $ABC` would print out "hello"; but if the shell variable `ABC` was not previously set to anything, then the line `echo $ABC` would print out `$ABC`. For scripting then, a comparison can be made to determine if a shell variable exists in the current monitor environment: `if $ABC seq \ $ABC goto VARS_DOESNT_EXIST`.

NOTE3: The logic of notes 1 & 2 above applies to symbols as well as variables.

Shell Variables Created and/or Used by MicroMonitor

When the monitor starts up, it looks for the presence of certain shell variables to configure itself (see description of monrc file as an example); also, it sets up some shell variables to be used by scripts or application code for various purposes. Following is a list of the shell variables intrinsic to the monitor:

- **APPRAMBASE**

This shell variable is loaded with the starting point of the RAM space that is made available to the application. This typically starts on some modulo 0x100 boundary (or larger), just above the end of RAM space that is used by the monitor itself. Note that this variable is automatically loaded by the monitor at startup. Certain facilities within the monitor use this value as a pointer to memory that is assumed to be "accessible". This is important to be aware of, so here are the facilities that make this assumption: edit, tftp server, tfs cp, xmodem receive, simple version of tfs clean. If your application will run and use these facilities in runtime, then the application must be mapped somewhere above the APPRAMBASE address so that these other facilities will not overwrite the application space. The value of the shell variable can be modified, and the modified value will then be used by these facilities, but make sure you know what you're doing! In general, its intent is to provide a common means of accessing the address that has been loaded into it by the monitor.

- **ARGC**

Argument count. This variable is loaded by the argv command with the current argument count.

- **ARG'N'**

Argument content. The shell variables ARG0 thru 'N' are loaded with the argument list by the argv command.

- **APP_EXITONCLEANERROR**

This shell variable is used for testing TFS. It can be set to some value (anything, as long as it is set), and this will cause any error in defragmentation to result in a call to mon_appexit(0). Useful when testing for defragmentation faults. See also: SCR_EXITONCLEANERROR

- **ARPRETRYTUNE**

If this shell variable is set, then the ARP retry mechanism is reconfigured to the values specified. The format of the content of this variable is XX:YY:ZZ; where XX is the retransmit_delay value, YY is the giveup_count and ZZ is the retransmit_delay_max value. If the shell variable is not set, the defaults are 1:0:4. Refer to discussion of the tftp command for more details.

- **BOOTFILE**

This variable is loaded by the monitor's BOOTP/DHCP clients based on the content of the "file" member of the BOOTP/DHCP response. Refer to RFC 951 (bootp) or RFC 2131 (dhcp).

- **BOOTROMBASE**

This variable contains the address that the monitor sees as the starting point of the base flash device. This is typically used to allow a host-based script to be unaware of where the actual starting point of the flash is but still transfer data to it. Note that this variable is automatically loaded by the monitor at startup. It can be modified by the user, but its intent is to provide a common means of accessing the address that has been loaded into it by the monitor; so, assume it is read-only.

- **BOOTSrvR**

This variable is loaded by the monitor's BOOTP/DHCP clients based on the content of the "siaddr" member of the BOOTP/DHCP response. Refer to RFC 951 (bootp) or RFC 2131 (dhcp).

- **CONSOLEBAUD**

This variable is used to allow an application to run with the same baudrate that the monitor is currently running at. Each time the baudrate is set via mstat -b the new setting is stored in this variable.

- **DCLIPORT**

If this shell variable is present at startup, then MicroMonitor's DHCP client will use this as the client port number instead of the default of 68.

- **DHCPCLASSID**

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include this string as the DHCP Extension "Class-identifier" (refer to RFC 2132). The format of the content of this variable is a simple string.

- **DHCPCLIENTID**

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include this string as the DHCP Extension "Client-identifier" (refer to RFC 2132). The format of the content of this variable, since it represents a type:value pair, is #:HHHHHH...; where '#' is a decimal value less than 256, ':' is a delimiter, and 'HHHH...' is a string of ascii-coded pairs where each 'HH' is converted to one 8-bit binary value.

- **DHCPFLAGS**

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, or the BOOTP client issues a BOOTP_REQUEST, it will use the value stored here as the flags. If not present, then the flags are zero. Note that the only valid bit at this point is 0x8000 (enable broadcast reply), and it is only applicable to DHCP.

- **DHCPLEASETIME**

This shell variable serves two purposes for the DHCP client :

- If set in monrc, then it will be used as a minimum requirement lease time if the server sends the client a lease time (if the lease time from the server is less than the content of this shell variable, then ignore the server's offer);
- It will be loaded with the actual lease time (if any) specified by the server. The value is in hex, and will only be present if the server has sent the client some minimum lease time.

- **DHCPOFFRFLTR**

If this shell variable is present, then if the MicroMonitor's DHCP client receives a DHCP_OFFER, it will look to the content of DHCPOFFRFLTR to determine if the offer should be accepted or ignored. The format of the content of this variable is

DHCP_FIELD_ID,EXPECTED_STRING

where...

DHCP_FIELD_ID can be:

BFN:	bootfile name
SHN:	server-host name
VSO###	vendor-specific option number embedded within Vendor Specific Information (opt#43) (valid range: 0 < opt < 255)
SSO###	site-specific option number (valid range: 127 < opt < 255)

and...

EXPECTED_STRING is the ascii string that must be within the specified field. Note that "within" means that the string may be the entire

string returned by the server or a sub-string within the string returned by the server.

Refer to the logic in ValidDHCPOffer() for complete details.

- **DHCPRETRYTUNE**

If this shell variable is set, then the DHCP retry mechanism is reconfigured to the values specified. If the shell variable is not set, the defaults are 4:6:64. Refer to description of ARP_RETRYTUNE for syntax.

- **DHCPRQSTLIST**

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include the digits extracted from this string as the DHCP Parameter Request List (refer to RFC 2132 section 9). The format of the content of this variable is #:#:#:..; where each '#' is a decimal value representing one of possibly several parameter requests to be made.

- **DHCPVSA**

This shell variable will be loaded with an ascii-coded hex string that represents the vendor-specific area returned by the BOOTP or DHCP server. For BOOTP this array is fixed at 64 bytes of binary data, so

DHCPVSA will contain a 128-byte ascii-coded-hex copy of that array. For DHCP the size of the vendor-specific area can vary. Since the monitor scans for options itself, the function DHCPGetOption() can be re-used in application space after converting the content of DHCPVSA to binary.

- **DSRVPORT**

If this shell variable is present at startup, then MicroMonitor's DHCP client will use this as the server port number instead of the default of 67.

- **ENTRYPOINT**

This variable is set by the tfs ld command. The value corresponds to the entrypoint address of the application just loaded.

- **ETHERADD**

This variable is expected to contain the MAC address that is to be assigned to the target. The MAC address can be retrieved in one of several different ways. Refer to the discussion of MicroMonitor startup for details.

- **EXCEPTION_SCRIPT**

If set, then the content of this shell variable is assumed to contain a script name that is to be executed when an exception occurs.

- **GIPADD**

The Gateway IP address. This variable must be established for systems on a network. It is automatically loaded by BOOTP/DHCP from the vendor-specific options if applicable.

- **HEAPSIZE**

By default, the monitor has a fixed amount of allocated memory dedicated to its own heap. If this variable is set, then it tells the monitor how much additional memory space has been made available. NOTE that as of July 2, 1999, HEAPSIZE is no longer used. Its functionality is replaced by options in the heap command.

- **HEAPSPACE**

By default, the monitor has a fixed amount of allocated memory dedicated to its own heap. If this variable is set, then it tells the monitor that there is additional memory space that can be used by the monitor's allocator and it will start at the address specified in this variable. NOTE that as of July 2, 1999, HEAPSPACE is no longer used. Its functionality is replaced by options in the heap command.

- **IPADD**

This variable is expected to contain the IP address that is to be assigned to the target. The IP address can be retrieved in one of several different ways. Refer to the discussion of MicroMonitor startup for details.

- **MALLOC**

The heap -m command returns a block of memory from the monitor's heap. It also populates this shell variable with the address of the block.

- **MCMDPORT**

If this shell variable is present at startup, then MicroMonitor's MONCMD server will use this as its port number instead of the default of 777.

- **MONFLAGS**

If this shell variable is set at startup, the monitor will use it to setup some internal flags. The format of the content of this variable is xx:yy:zz...;

where "xx", "yy" and "zz" are ascii strings (flag names) that represent the setting of certain binary flags in the monitor...

flag name	flag meaning
nophdr	do not print initial header when monitor is reset
nopdf	quiet defragmentation when verbosity is off
noptftp	quiet TFTP when verbosity is off
nopmcmd	quiet MONCMD when verbosity is off
notftpovw	do not allow incoming TFTP write requests to overwrite an existing file
moncomv	enable verbosity for accesses to moncom through monConnect()

Note that this shell variable is only read at system startup. Changing it in runtime has no affect on the internal flag settings.

- **MONITORBUILT**

Automatically loaded with the date and time of the monitor build. The format is dd/mm/yy@hh:mm:ss and can be used to automatically determine if the monitor binary should be updated.

- **NETMASK**

The mask of the subnet on which the target resides. This variable must be established for systems on a network. It is automatically loaded by BOOTP/DHCP from the vendor-specific options if applicable.

- **NO_EXCEPTION_RESTART**

If this shell variable is present, then at the time of an exception, the monitor will not reset. The system will remain at the monitor level.

- **PROMPT**

This variable contains the string that the monitor uses as the user prompt. If not set, the default prompt is uMON: and this is loaded into the PROMPT shell variable. At any time this shell variable can be changed and the prompt used by the monitor will change to the content of PROMPT.

- **NETMASK**

The subnet mask. This variable must be established for systems on a network.

- **RLYAGNT**

The Relay Agent IP address. It is automatically loaded by BOOTP/DHCP from the giaddr field of the BOOTP/DHCP response if applicable.

- **SCR_EXITONCLEANERROR**

This shell variable is used for testing TFS. It can be set to some value (anything, as long as it is set), and this will cause any error in defragmentation to result in setting the internal flag that causes a script to terminate. Useful when testing for defragmentation faults. See also: APP_EXITONCLEANERROR

- **SCRIPTVERBOSE**

This variable, if present, is used to establish the verbosity level of a script running. Valid values are 0, 1 & 2. 0 is no verbosity, 1 means the command line is echoed, 2 means the command line is echoed before and after CLI processing.

- **SYMFILE**

If this shell variable is set, then the default file name of "symtbl" is overridden with the content of this variable. See discussion of symbols vs shell variables above.

- **TFTPGET**

If this shell variable is present after a tftp get command, it is indication that the file transfer succeeded; it will then contain the number of bytes of data that have been transferred through tftp.

- **TFTPPORT**

If this shell variable is present at startup, then MicroMonitor's TFTP client/server will use this as its port number instead of the default of 69.

- **TFTPRETRYTUNE**

If this shell variable is set, then the TFTP retry mechanism is reconfigured to the values specified. If the shell variable is not set, the defaults are 4:3:32. Refer to description of ARPRETRYTUNE for syntax.

Writing MicroMonitor Scripts

Since there is a file system and a command interpreter, there is an ability to create executables as simple ascii files that invoke the commands that are part of the monitor's command table. Note that it is not the intent of this page to go into full detail on each of the commands used within a script (refer to command listing for that detail); rather to provide an overall understanding of how to use/write scripts to be run under the monitor. The best way to learn how to take advantage of this capability is through example.

Script Invocation

A file is recognized by TFS as a script if it has the executable (e) attribute set with no other file format attributes (E, C or A) set. Scripts can be started up in two different ways. Like Unix SH, the script name can be typed on the command line directly and the command interpreter will find it. This implies that a script should not have the same name as any of the built-in commands in the command table because the command interpreter will first look through its own list of built-ins for a match.

Alternatively, the `tfs run` command can be used. Running the script this way eliminates the concern of an executable having the same name as an internal built-in, but it also allows the user to specify that the script run in verbose mode. There are two levels of verbosity useful for script analysis:

- **level 1** (`tfs -v run scriptname`): each line is printed out with shell variables replaced by their actual value.
- **level 2** (`tfs -vv run scriptname`): same as level 1 plus, prior to the level 1 print, the original line is printed, preceded by a '+' (plus sign and space), and the shell variables are not replaced.

Either method (direct invocation through command interpreter, or using `tfs run`) has its purpose. The use of verbosity with `tfs run` makes the script somewhat run-time debuggable.

Script-Specific Commands

The following commands within the monitor's command line interface are applicable only within the context of a script: IF, GOTO, GOSUB, EXIT, RETURN, READ, ARGV, ECHO, ITEM and SLEEP.

Script Nesting

Script nesting (one script calling another script) is supported and is limited only by the amount of stack that has been allocated to the monitor. This works well because the script runner within TFS calls subscrips by simply furthering its depth into its own stack. If a script runs a binary executable (COFF, ELF or AOUT), things can get a bit more complicated, and just what happens depends on the way the executable was built. If the executable simply returns through the same point from which it was called (the entryptoint of that load image), then it will return to the script runner when complete and the calling script will regain control. If the executable starts up some multi-tasking operating system, then its only means of termination is through `mon_appexit()`. This means that the monitor is being reentered; hence, no stack frame to return through, so the calling context is lost and no further "unnesting" will be done. Refer to the example application startup code for a simple example that can be configured to use `mon_appexit()` or simply return to the monitor.

Note that the limitation described above is beyond the scope of this monitor. A typical system will boot up and run some diagnostic executable, followed by a script that configures the system and finally the actual application executable. All three of these files would be in TFS with the auto-boot flag set.

One final note regarding script nesting... All shell variables are global. If script A sets VARA, then calls script B which modifies VARA, script A will see that modification.

Example #1: cleanup

This script can be used to apply some conditions to the "tfs clean" command which is used to defragment the FLASH used by TFS. The script can be run at the command line with or without an argument passed to it, or (if the TFS attribute 'b' or 'B' is set) it will automatically run at system startup.

```
# Cleanup script:
# Used to call tfs clean if TFS is running out of flash space.
# If argument is present, then use it as the threshold; else
# use default of 1000000.
#
argv -v                                # Load cmd line args into shell vars.
set SIZE 1000000                        # Establish default size.
if $ARGC eq 1 goto CLEANUP             # If no args, jump to cleanup; else
set SIZE $ARG1                          # set SIZE to cmd line arg value.

# CLEANUP:
tfs -v freemem freemem                 # Determine amount of free memory left
                                        # in TFS and place that value in $freemem.
set -i freemem 0                       # Change to decimal value.
if $freemem gt $SIZE goto DONE         # If $freemem is less than size, run
tfs clean                               # tfs clean to defragment.

# DONE:
```

Notice the use of '#' as an indication that the remaining text on the line is a comment. Also, note the use of "goto tags" such as 'CLEANUP' and 'DONE'. When script execution is in progress, if a 'goto' statement is reached (either standalone or part of an 'if' statement) the argument to the 'goto' is assumed to be a tag that exists somewhere else in the script. A target of a goto tag is seen by the monitor as the first whitespace delimited block of text after a pound sign. The content of the argument to goto is compared to the same number of characters at the start of the tag. As a result, the line

```
# CLEANUP:
```

is the target of the line

```
if $ARGC eq 1 goto CLEANUP
```

even though the target has the colon at the end, the first 7 characters match.

Finally, note that multiple targets with the same text can cause undefined results. The following two lines are considered to be the same target...

```
# CLEANUP: this is the start of tfs defragmentation
# CLEANUP is the first line of tfs defragmentation
```

Example #2: ping

The following example is similar in syntax to the above script, but carries out a totally different task...

```
#
# ping script using icmp echo and argv:
# script syntax: ping IP_ADDRESS [optional ping count]
#
argv -v
if $ARGC eq 2 goto PING_1
```

```

if $ARGC eq 3 goto PING_N
echo $ARG0: requires IP address
exit

# PING_1:
icmp echo $ARG1
exit

# PING_N:
icmp -c $ARG2 echo $ARG1

```

Not much to say. Same kind of deal as cleanup above. Uses comments, goto tags and the argv command to build a fairly self-explanatory wrapper around the 'icmp' command in the monitor.

Example #3: namelist

The following example demonstrates a usage of the nested shell variable capability in the monitor's command interpreter...

```

# Build a name list:
set NAME_1 Jane
set NAME_2 John
set NAME_3 Peter
set NAME_4 Paul
set NAME_5 Tommy
set NAME_6 Adam
set NAME_7 Eric
set idx 1
set max 7

# Now print the name list using the idx shell variable as an index:
# TOP:
if $idx gt $max goto DONE
echo ${NAME_${idx}}
set -i idx
goto TOP
# DONE

```

The output of this script would be...

```

Jane
John
Peter
Paul
Tommy
Adam
Eric

```

Note that the 'idx' shell variable is used like an index into an array of names; where the array is called 'NAME_'.

Example # 4: namelist using "item" command

The following example demonstrates a usage of the "item" command in the monitor's command interpreter. The end result is similar to the above example, but implemented in a simpler way...

```

# Print each name in a list:
set idx 1
# Top_of_Loop:
item $idx NAME Jane John Peter Paul Tommy Adam Eric
if $NAME seq \ $NAME exit
echo $NAME

```

```
set -i idx
goto Top_of_Loop
```

The output of this script would be...

```
Jane
John
Peter
Paul
Tommy
Adam
Eric
```

Example # 5: processing a variable number of command line arguments

This example uses the argv command and the command interpreter's shell variable processing to handle a command line that has a variable number of arguments...

```
# This script processes a variable number of command line
# arguments using the argv command and nested shell variables...
argv -v
set idx 0
# TOP:
echo Arg $idx: ${ARG${idx}}
set -i idx
if $idx lt $ARGC goto TOP
```

If the script was invoked with "script aa bb cc dd", the output would be...

```
Arg 0: script
Arg 1: aa
Arg 2: bb
Arg 3: cc
Arg 4: dd
```

Example # 6: why would you ever want to do this??...

This example takes the shell variable usage to a bit of an extreme, but demonstrates its capability within a script yet again...

```
set abcX HELLO
set defY MOM
set var1 X
set var2 Y
set HELLO_MOM BINGO!
echo ${abc${var1}}_${def${var2}}
echo ${${abc${var1}}_${def${var2}}}
```

The output would be generated by the two final "echo" lines...

```
HELLO_MOM
BINGO!
```

Example # 7: startup script using subroutines, if/else and file decompression

This is a practical script example that could be used for a system that may (or may not) have their application compressed and their monitor has been configured in such a way that file decompression requires temporary monitor heap expansion. Note that the use of the 0x44000002 is PowerPC specific, but could be re-worked for other CPUs. See the heap man page for discussion on heap expansion.

```

#####
#
# run:
# Autobootable script used for starting up MicroMonitor
# based systems. This script allows the user to be
# unaware of the fact that the application program may be
# compressed. The heap expansion step is needed for decompression
# here because the monitor was intentionally built with less
# heap space allocated to it than is needed for decompression.
# If the monitor was allocated more memory at build time, this
# step could be eliminated.
# In addition, if the BREAKPOINT shell variable is set, then
# after the application has been loaded, but before execution
# starts, the value of 0x44000002 is placed in the address
# specified by $BREAKPOINT. This inserts a SC (system-call)
# instruction at that address and will cause an exception to
# occur. At that point, the monitor command "strace" can be
# used to dump the stack frame.
#
# Main:
argv -v
if $ARGC eq 1 gosub APPNAME_DEFAULT else gosub APPNAME_ARG1
gosub FILE_CHECK
if -t iscmp $APP gosub EXPANDLOAD else gosub NORMALLOAD
if $BREAKPOINT sne \ $BREAKPOINT gosub SETBREAKPOINT
call $ENTRYPOINT
reset

# end Main
#
#####
#
# begin Subroutines:

# SETBREAKPOINT:
pm -4 $BREAKPOINT 0x44000002
return

# EXPANDLOAD:
heap -X 0xf0200000,0x40000
gosub NORMALLOAD:
heap -x
return

# NORMALLOAD:
tfs ld $APP
return

# APPNAME_ARG1
set APP $ARG1
return

# APPNAME_DEFAULT:
if $PLATFORM seq MY_PLATFORM goto APP_MINE
if $PLATFORM seq YOUR_PLATFORM goto APP_YOURS
echo Invalid platform: $PLATFORM
exit

# APP_YOURS:
set APP you
return

# APP_MINE:
set APP me
return

```

MicroMonitor Introduction and Command Set

```

# FILE_CHECK:
set SIZE
tfs size $APP SIZE
if $SIZE sne \ $SIZE return
echo File error: $APP
exit

# USAGE:
echo Usage: $ARG0 [appname]
exit

```

Overriding the default command interpreter used by the script runner

By default, scripts are run in the monitor using the command interpreter that is part of the monitor. This allows all of the commands within the command table of the monitor to be accessible by a script. If an application takes over the system, it may have its own command interpreter, so if the application expects to be able to use the monitor's script runner, then two things must be done:

- The application must inform the monitor that it has a command interpreter and that the script runner in the monitor should use that command interpreter instead of the one in the monitor. This allows the monitor's script runner to access the commands that are in the application's command table. This is done with the monitor's API function `tfctrl()` and the `TFS_DOCOMMAND` request.
- The application's command interpreter must provide a hook so that if the command is not seen in the application's command table, the command is passed to the monitor's command interpreter. This can be done within the application's command interpreter by calling the monitor's API function `docommand()`.

If these two steps are taken, then an application will have the ability to use all of the commands in the monitor plus all of its additional commands in a script that is run when the application is active. Note that the monitor supports the case of duplicate commands existing in application and monitor space by ignoring an initial underscore if found on the command line. This means that if the application has a "help" command and the monitor also has a "help" command, then "_help" into the application's command interpreter will be processed as "help" by the monitor's command interpreter.

User Levels

The following discussion focuses on the monitor's concept of user levels. The current implementation of the monitor supports 4 user levels (0 thru 3). The files in TFS and the commands in the monitor shell can be configured with a user level.

Description

The monitor supports the concept of user levels. At any given time, there is an active user level. At startup, the user level is at its highest (3). If left untouched (via the `ulvl` command), the user level will remain at its max and all facilities within the monitor will be accessible through the command line interface. If, however, one of the auto-bootable files lowers the user level, then all accesses made to the hardware through the monitor's user interface will be limited to the facilities that are available to the new user level. Commands and files can be configured to be accessible only at a particular user level (or above). At system startup, all commands default to require that the system be at user level 0 or higher to execute (in other words, all commands can be executed). If commands are to be restricted to different user levels, then the `ulvl` command should be used in the `monrc` file to make all the necessary adjustments. For example, if the "flash" command is to be accessible only by user level 3, then in the `monrc` file (or some other autoboot-no-query script) the line `"ulvl -c flash,3"` will raise the required user level of the flash command and the line `"ulvl 0"` will lower the current user level of the system to 0; hence, to run the flash command, the user will need to know the user-level-3 password. Note that a default system startup does not have any password file installed, so access to the various user levels is approved with any password. The `ulvl -p` command must be used to create the three required passwords.

The usefulness of this feature depends on the needs of the application, but it basically provides the system with a mechanism to "protect" some portion of the system from unauthorized users. The user level can be raised, but only by a user that knows the password to get to that particular user level. The passwords are stored in a file in TFS that is automatically saved at the highest user level. The file is also unreadable by user levels lower than the max.

A User Level Example

Assume you have an embedded system project that is to be sold to a VAR (value-added-reseller). You want to use the same code base, but provide specific configuration parameters unique to each VAR you sell your box to. The VAR also wants to configure certain features based on the customer it is selling to, and finally, the end customer may have some administration parameters that it is able to configure on site. This situation calls for the ability to have certain parameters configurable by the manufacturer, but not modifiable (or possibly even readable) by the VAR or its end user. Similarly, certain parameters are configurable by the VAR and should not be accessible by its customer. Finally, certain parameters are configurable by the customer, but should not be accessible by "guests" of the customers system. Each of these parameters should only be modifiable by the entity that is able to configure them; hence, the need for file access and command execution priority levels.

This hierarchy of command and file access allows you to have certain portions of the system accessible only at user level 3 (those that you use to configure prior to shipment to the VAR). Then the VAR has certain configuration parameters set up at level 2, the customer has level 1 and, finally, a guest can log in at level 0. This allows a product to be shipped with a common code base that is configurable through files that are only accessible by the user level that needs to be able to access them.

Currently, all commands are "visible" at all levels, but, since each command has a user level attached to it; each command will only be executable if that user level (or higher) is currently active. Likewise, for files, a file can have a user level attached to it and only be modifiable at that user level (or higher). Also, there is a `"ulvl_unreadable"` flag that can be associated with each file. This flag tells TFS that for all user levels below the file's designated level, the file is not even visible to the user; hence, files that contain passwords or other information that may be sensitive, can be "hidden" to lower user levels.

Since the idea of user levels requires some kind of password to put the monitor at that level, there is also a backdoor mechanism set up so that we can gain access to a system by knowing its MAC address. This essentially eliminates the possibility of losing accessibility because of forgetting a password. The backdoor password is an encrypted string based on the MAC address and can be created from a tool (maccrypt) that uses the same code as the monitor but runs on SUN Solaris and/or PC Win95/98/NT.

Command Line Interface Output Redirection

The monitor's command line supports the ability to log the output of a command to a memory buffer and ultimately to a file. At build time, the INCLUDE_REDIRECT macro must be set to 1 in config.h to enable this. The syntax for doing this is similar, though not identical, to standard redirection on Unix. The difference is due to the fact that the redirection code within the monitor must be supplied with not only a file name, but also a buffer and buffer size. The idea here is that once supplied with a buffer, command output can be copied to this buffer and eventually the buffer will be transferred to a file. The syntax follows, with the redirection directives in red...

```
uMON> echo this is some text >buffer,buffer_size[,filename]
```

This is the syntax for the single right arrow. A one or two comma delimited string containing a buffer address followed by the size of the buffer and an optional file name. If the filename is specified, then the output of the command is copied to the buffer (truncated at buffer_size if necessary) and then transferred to TFS as "filename". The running buffer pointer is reset back to the base address of buffer. If filename is omitted, then the output of the command is copied to the buffer and the pointer into the buffer is left at the position just after the data copied (assuming buffer_size is not reached).

```
uMON> echo this is more text >>[filename]
```

This syntax is used to append the output of the command to the buffer that was created by the '>' syntax described above. If 'filename' is present, then the content of the buffer is transferred to TFS as "filename" and the running buffer pointer is reset back to the base address of the buffer. If "filename" is not specified, there is no transfer to a file, the running pointer is incremented to the position just after the data copied (once again assuming buffer_size is not reached).

In both cases above, the "filename" string can contain up to 2 additional commas. These would be used to tell TFS the flags and info field to apply to the file when it is created, so the filename string could be "filename,flags,info".

The memory space used is independent of the monitor's implementation of this capability; hence, it is the users responsibility to make sure that the buffer and size is memory space that can be used. There are a few different ways to determine a buffer area. Typically, the buffer would simply be \$APPRAMBASE, but if the application is running or for some reason \$APPRAMBASE is not appropriate, then the heap -m command can be used to allocate a buffer from the monitor's own heap space.

Here is an example with a buffer created and two additional command outputs appended to the buffer prior to it being transferred to a file (once again, the redirection directives are in red)...

```
echo this is some text >$APPRAMBASE,4000
echo this is another line of text >>
echo this is the final line >>logfile
```

Line 1 establishes the buffer to be 4000 bytes starting at APPRAMBASE. The command output is copied to the starting point of the buffer. Line 2 appends the output of the second command to the buffer (no file transfer). Line 3 appends to the buffer and then transfers the buffer to "logfile".

Note that at all times during the buffer writes, if the running pointer reaches the end of the buffer (as specified by the buffer_size), then the logging is stopped. When, eventually, the directive is specified to write the file, it will have truncated to the size of the buffer.

MicroMonitor Command Set:

ARGV

NAME:

argv

SYNOPSIS:

Create or read in an argument list.

USAGE:

argv -[cilv] [arg0] [arg1] ... [argN]

DESCRIPTION:

This command allows the monitor to build an argument list that is accessible by the application through the monilb function `mon_getargv()` (for compiled executables) or by the `argv` command itself (for scripts). For compiled executables, this requires that the application be built with the `mon_getargv()` call ran prior to `main()`; such that, `mon_getargv()` will build the "int argc, char *argv[]" argument list parameters accessed by the application. For scripts, the argument list is automatically built when the script is called on the command line and then the script uses the `-c` and `-v` options to access the argument list as shell variables (see below).

OPTIONS:

- `-c`
load the shell variable `$ARGC` with the argument count;
- `-i`
initialize (clear) the internal argv list;
- `-l`
list the current argv array strings;
- `-v`
load `$ARGC` with argument count and `$ARG'N'` with each argument passed into a script;

EXAMPLES:

- `argv appname PCM 0x1400`
Builds `argv[0]` as "appname", `argv[1]` as "PCM" and `argv[2]` as "0x1400"; the argument count would be 3
- `argv -l`
Displays the current "char *argv[]" array configured
- `argv -c`
Loads the shell variable `$ARGC` with the argument count currently stored by argv.
- `argv -v`
Loads `$ARGC` with argument count and `$ARG'N'` with each command line argument.

NOTES:

- A host-based debugger can use this command when an application is downloaded into RAM from the host.
- When an application under TFS is started, the arguments can be on the command line and `argv` is not needed.
- When the executable is a TFS script, the `-v` and `-c` options become useful for retrieving the argument count and argument list.
- The minimum argument count passed to a script would be 1, and the first argument of the list is `ARG0`.

ARP

NAME:

arp

SYNOPSIS:

Arp cache maintenance..

USAGE:

arp -[fp] [IP]

DESCRIPTION:

ARP (address resolution protocol) is used to allow a network element to query the network for the ethernet address assigned to a known IP address. This command is a user interface hooked into the monitor's basic ability to query the network for ethernet addresses and maintain a cache of the most recently used addresses.

OPTIONS:

- -f
flush the current arp cache;
- -p
do not use the gateway if the IP is on a different subnet;

EXAMPLES:

- arp 135.3.94.136
Returns the ethernet address for the network device that is assigned IP address 135.3.94.136.
- arp -f
Flushes all entries in the arp cache.

NOTES:

- If the IP address specified is not on the same subnet as the target making the query, then the actual ethernet address that is loaded into the arp cache is that of the gateway.
- This command requires some network knowledge. In particular, it assumes that the shell variable NETMASK is loaded with the correct sub-net mask setting for the network the target is residing on. If it is determined that the IP request is for a device not on the local sub-net, then GIPADD must be loaded with the IP address of the gateway and the arp request is destined for the gateway. If NETMASK is not set, then the IP address is assumed to not exist on the same subnet. If GIPADD is not set, then the ARP request is sent to the IP address specified (may be picked up by the gateway, if it supports proxy arp). Note that the GIPADD is only needed if '-p' is not specified.

AT

NAME:

at

SYNOPSIS:

Complex breakpoint facility.

USAGE:

at `-[Dd:f:lilrs:v:] {address} [if condition] [action]`

DESCRIPTION:

This command provides a state-machine-like capability for breakpoints. It uses the processors breakpoint facility as a way of entering the monitor from active application. Upon entry into the monitor, a few different things can happen:

- an action can be taken and then application is resumed;
- a condition is checked and if the condition is met, an action is taken;

It is basically an elaboration of the counting breakpoint (break at address HHHH the Nth time). The counting breakpoint uses the break trap to temporarily enter the monitor, increment the break count and if the count has reached the desired value, then break; else resume application. In this case the condition is met because the break count was reached, and the action taken is stoppage of the application. The "at" command simply takes this idea a bit further by providing several different conditions and actions.

OPTIONS:

- -D
delete all at-points;
- -d {#}
delete at-point number #;
- -f {xx}
set at-flag to xx
- -l
initialize 'at' structures;
- -i
install at-points;
- -l
list current at-point state;
- -r
remove at-points;
- -s
{x,y} swap at x with at y

CONDITIONS:

- %VV==##
at-var VV equals ##
- PCNT==##
pass count == ##
- 0xFFFF()==##
function at 0xFFFF returns
- FEQUXX
at-flag == XX

- FALLXX
at-flag OR XX == XX
- FANYXX
at-flag AND XX is non-zero
- %VV[.s]==*YY
at-var VVÆ (size 's') equals *YY

EXAMPLES:

- at 0x400000 %01++
If location 0x400000 is hit, then increment at variable æ01Æ.
- at 0x408010 if %01==12 BREAK
If location 0x408010 is hit and at variable 01 equals 12, then break to the monitor.
- at -l
Lists the current at point conditions, variables and flag.

NOTES:

- There are 8 AT variables (%VV in comments above) whose names are specified as %00 through %07.
- By default, the actual traps are installed by the `go` command and they are removed when a break point occurs. The traps can be inserted or removed forcefully with the `-i` and `-r` options.
- Since the % sign is used within the syntax of the at command, it may be necessary to avoid symbol table lookup by preceding it with a backslash (\).
- This command is extremely CPU dependent, so before assuming this works on your target verify that it has been ported!

CALL

NAME:

call

SYNOPSIS:

Execute an embedded function somewhere in the memory image.

USAGE:

call `-[aqv:] {address} [arg1] ... [argN]`

DESCRIPTION:

This command allows the monitor to call a function at a raw memory address. A maximum of 7 arguments can be specified. By default the function is called with the arguments converted to hex. For example `call 0x12345 45 99` would be used to interface to a function whose prototype is `func(int val1, int val2)`. The `-a` option allows call to work with functions whose prototype is `(int argc, char **argv)`. Refer to symbolic debugging for information on how to pass the function name as an argument to call instead of the hex address.

OPTIONS:

- `-a`
Pass parameters to embedded function as `(int argc, char **argv)`
- `-q`
Do not print the "Returned XX" string at the end of the command
- `-v {varname}`
Place the hex return value of the function call into the shell variable "varname".

EXAMPLES:

- `call 0xa0041000 0x100 45 0x999`
Pass control to a function located in memory at address `0xa0041000` and it will receive `(0x100,45,0x999)` as its parameter list.

NOTES:

- There is no attempt to verify that the address specified has valid code. It is up to the user to make sure the address is correct and the function resides there.

CAST

NAME:

cast

SYNOPSIS:

Overlay a structure definition onto a block of memory and display it.

USAGE:

cast [-al:n:pt:] {structure name} {address in memory}

DESCRIPTION:

This command allows the user of the monitor to display a block of memory as a structure, table of structures or linked list of structures. The definition of the structure is assumed to be in the TFS file called "structfile", and any number of structure definitions can be in that file. The format of the structure definition is similar in syntax to that of standard 'C'.

OPTIONS:

- -a
display the address of each member of each structure;
- -l {linkname}
treat the specified address as the base of a linked list and get the next link from the member of the structure that matches the string 'linkname';
- -n {structure name}
specify the name of the structure to be displayed (not needed, but provides a nicer display output);
- -p
display any padding that is part of the structure definition;
- -t {table name}
treat the specified address as the base of a table and query the user for permission to display each successive entry in the table of structures.

SYNTAX OF "structfile":

To cast a structure over a block of memory, there must be some structure definition. The definition used by the cast command is assumed to be in the file "structfile". In general, the format of this file is similar to that of standard 'C' structure definitions, but with some limitations. The types "char", "short" and "long" are supported and they will be displayed as a 1,2 or 4-byte decimal integer respectively. To support the ability to display in hex, the types "char.x", "short.x" and "long.x" are supported, and, if a character is to be displayed as a character (hex 0x31 printed as '1'), the "char.c" type can be applied. For example, in the following structure definition...

```
struct abc {  
    long i;  
    long.x j;  
    char.c c;  
    char.x d;  
    char e;  
}
```

The member 'i' would be displayed in decimal format and 'j' would be displayed in hex. The member 'c' would be displayed as a character, 'd' would be displayed in hex, and 'e' would be displayed as a 1-byte decimal integer.

If a structure has an array in it, then the user must define that as an array of one of the fundamental types described above with the appropriate size. This cast command does not display arrays within structures

simply because of the complexity of the output generated, so it is treated like padding with only the name and array size displayed.

Here is an example of a structure definition file that demonstrates all of the functionality of the cast command. Note that the '#' sign signifies a comment..

```
# Structures abc and def:
struct abc {
    long i;
    char.x c1;
    pad[3];          # Invisible unless -p specified.
    struct def d;
}
struct def {
    short s1;
    long ltbl[32];  # Array of 32 integers (not dumped)
    short s2;
}
```

Notice the embedded structures, use of the '.x' suffix and the pad[] format. Also, be aware that this cast command is totally unaware of compiler-specific padding and cpu-specific alignment requirements. If the structure definition puts a long on an odd boundary and the CPU does not support that, then cast will generate an exception itself. The user must add the appropriate padding to deal with this. As a result, the "pad[]" descriptor above is used for cpu/compiler-specific padding.

If the member is of type "char.c *" or "char.c []", then cast will display the ascii string (if you don't want it to be dereferenced, use "char.x *"). Finally, do not "test" this command's parsing ability. Put one member on each line of the file and keep it syntactically simple; otherwise, you will break it!

EXAMPLES:

- cast struct_x 0x10010000
Display memory at location 0x10010000 as if it was the structure "struct_x" as defined in the TFS file "structfile"
- cast -a struct_y 0x4000
Display memory at location 0x4000 as if it was the structure "struct_y" as defined in the TFS file "structfile". Prefix the display of each member with the address at which that member is being retrieved from memory.
- cast -t tasktbl tstruct 0x4100100
Starting at location 0x4100100, overlay the structure "tstruct" onto memory. Repeat the display process at what would be considered the next table entry (based on the size of the tstruct structure).
- cast -l next list 0x1000
Assume a structure of type "list" exists at location 0x1000, then use the content of the "next" member to determine the location of the next structure to display (a linked list).

CM

NAME:

cm

SYNOPSIS:

Copy memory from one target address to another.

USAGE:

cm `[-24fv]` {source address} {destination address} {count}

DESCRIPTION:

Copy memory from one location in memory space to another. The size of the copy is specified by the count, which is always considered a byte-count.

OPTIONS:

- `-2`
assume the width of the memory accesses is x2;
- `-4`
assume the width of the memory accesses is x4;
- `-f`
assume the destination address is a FIFO;
- `-v`
verify that 'count' bytes of data starting at the source address match those at the destination address.

EXAMPLES:

- `cm -2 0x400000 0x402000 0x400`
Copy 1024 bytes of data from 0x400000 to 0x402000 using a memory access width of 2 bytes.

NOTES:

- The command assumes that the memory writes are simple. All memory written to is RAM. Attempts to write to flash or other non-volatile memory will fail. Be aware that this is true for all memory access commands except the "flash" command.

DHCP

NAME:

dhcp

SYNOPSIS:

Dynamic Host Configuration Protocol discover request.

USAGE:

dhcp -[bvV]

DESCRIPTION:

The monitor implements a subset of the DHCP client protocol. Based on RFC2131 spec, the "automatic allocation" mode (DHCP server assigns a permanent IP address, or infinite lease, to a client) is the only mode supported. The DHCP handshake can be started in one of three different ways:

- set the IPADD shell variable to "DHCP" in the monrc file;
- issue this dhcp command from some other auto-bootable script;
- if no monrc exists and the DEFAULT_IPADD entry in config.h was set to DHCP when the monitor was built;

All dhcp startup mechanisms run the same client code; the only difference is that when DHCP is started through this command, there is no automatic retry and no TFTP server interaction; it is left to the script to handle that. This is done to allow the user to override the automatic sequence of events discussed below with some other startup method. Aside from that difference, all other aspects of the handshake are identical (refer to discussion below).

Sequence of events for this limited implementation of DHCP...

- Client issues a DISCOVER: broadcast the fact that it is looking for a DHCP server.
- Server responds with an OFFER: server has received a DISCOVER request from the client. The offer may contain all the information that the DHCP client needs to boot-up, but this is dependent on the configuration of the server.
- Client issues a REQUEST: the client accepts the offer from the server and asks the server to send back the information it needs.
- Server responds with an ACK: reply from the server with the information requested.

Refer to discussion below for a more detailed discussion on monitor startup.

OPTIONS:

- -v
limited verbosity enabled;
- -V
extreme verbosity enabled;
- -b
run BOOTP;

NOTES:

- The use of various shell variables mentioned in this discussion provides an application with a lot of flexibility. It is beyond the purpose of this client to support all possible DHCP configurations, so some application-specific coding may be necessary. The monitor's DHCP code (dhcpboot.c & dhcp_00.c) is set up to allow application-specific code to be added without touching the generic DHCP client code.

MicroMonitor
Introduction and Command Set

- The shell variable IPADD can also be set to DHCPV or DHCPv to enable different levels of verbosity during DHCP transactions... 'V' is full DHCP verbosity and 'v' only prints the DhcpSetEnv() calls.
- All of the above discussion applies to both BOOTP and DHCP. For BOOTP autostart, set the IPADD shell variable to BOOTP instead of DHCP (v & V extensions apply).

DHCP/BOOTP Details:

If configured to do so, the target can issue a BOOTP/DHCP broadcast. Shell variables are used to configure the outgoing request and to store some of the returned parameters from the server. Upon reception of the BOOTP/DHCP handshake with some server, the target expects the *yiaddr* field to be loaded with the target's IP address and it will be stored in the IPADD shell variable. In addition, it will store the content of the *siaddr* field into the BOOTSRVR shell variable, the *file* field into the BOOTFILE shell variable, the *giaddr* field into the RLYAGNT shell variable. Certain standard options are also transferred to shell variables...the router (option code 3) into the GIPADD shell variable and the subnet-mask (option code 1) will be stored into the NETMASK shell variable. Note that each of these additional fields will be loaded into their respective shell variable only if found to be non-zero. Note also that if after the DHCP/BOOTP transaction has completed, RLYAGNT is loaded, but GIPADD is not, then the content of RLYAGNT is copied to GIPADD. Finally, to satisfy other non-standard scenarios, the entire content of BOOTP vendor-specific-area or the DHCP options area will be copied (in ascii-coded-hex) to the DHCPVSA shell variable.

Following are the DHCP and BOOTP header formats including what entries in the received header are transferred to shell variables in the monitor...

Format of a BOOTP Message:

```

{PRIVATE}op      htype (1)      hlen (1)      hops (1)
(1)

                xid (4)
secs (2)        unused (2)
                ciaddr (4)
                yiaddr (4)
                Copied to IPADD shell variable
                siaddr (4)
                Copied to BOOTSRVR shell variable
                giaddr (4)
                Copied to RLYAGNT shell variable
                chaddr (16)
                sname (64)
                file (128)
                Copied to BOOTFILE shell variable
                vend (64)
                Option #1 copied to NETMASK
                Option #3 copied to GIPADD
                Entire field copied to DHCPVSA

```

Format of a DHCP Message:

```

{PRIVATE}op      htype (1)      hlen (1)      hops (1)
(1)

                xid (4)
secs (2)        flags (2)
                ciaddr (4)

                MicroMonitor
                Introduction and Command Set

```

yiaddr (4)
 Copied to IPADD shell variable
siaddr (4)
 Copied to BOOTSRVR shell variable
giaddr (4)
 Copied to RLYAGNT shell variable
chaddr (16)
sname (64)
file (128)
 Copied to BOOTFILE shell variable
options (312)
 Option #1 copied to NETMASK
 Option #3 copied to GIPADD
 Option #51 copied to DHCPLEASETIME
 Entire field copied to DHCPVSA

Upon completion of the BOOTP/DHCP handshake, if both the BOOTFILE and BOOTSRVR variables have been loaded, the target will attempt a TFTP transfer of the specified file from the specified server. Since the file (destined to reside in TFS) can be one of several different types, the target will look for a comma extension on the filename to be used as the file's TFS flags. For example, if the BOOTFILE variable contains abcde,eB, then the file will be stored into TFS with the name "abcde" and flags "eB" (refer to discussion regarding TFTP destination syntax specification). This filename extension will be processed as TFS flags only if each of the characters in that extension are legitimate TFS flags (or attributes). As a final step in the BOOTP/DHCP startup, if the file is successfully transferred from the TFTP server, the target will then execute it as a standard executable file under TFS. Note that execution is attempted, but will only be carried out if the flags have been properly set.

Note1: the incoming file is transferred to TFS using the API function tfsadd(). This means that if the file already exists in TFS and is 100% identical, there is no flash-write overhead.

Note2: if there are no specified flags for the file that is downloaded, then it is assumed to be 'e', meaning that the file is treated as an executable script.

Warning: this BOOTP (or DHCP) handshake is done in the background while the monitor is booting up; hence, any autobootable scripts can run while the handshake is in progress. The user must be careful here... Potentially, an autobootable script and the handshake will be occurring at the same time, so the script must be aware of this. The important thing to note is that if the file that is downloaded into TFS is autobootable, then it will run while the BOOTP/DHCP handshake is potentially loading in a new version of the same file. This will definitely confuse things; so ideally, if the file loaded in by BOOTP/DHCP is executable, then there should be no autobootable files in TFS. On the other hand, if the file loaded in is not executable, then some other autobootable file can be set up to be waiting for this to be loaded.

DHCP Specifically...

For DHCP, the monitor supports the 'automatic allocation' portion of the specification. There is no lease expiration (see note below), the IP address assigned at the time of the initial handshake is assumed to be owned by the target until it is reset. Basically this means that the DHCP supported by the monitor is an extension of BOOTP, providing a bit more flexibility with regard to the parameters that can be retrieved from the DHCP server and the way in which the client and server agree to handshake. The target issues a DHCP_DISCOVER broadcast. In that message, the flags, class identifier, client identifier and DHCP parameter-request-list may be loaded from the shell variables DHCPFLAGS, DHCPCLASSID, DHCPCLIENTID and DHCPREQUESTLIST respectively. The DHCP server may be equipped to expect that and based on its contents, the server should respond to the broadcast with a DHCP_OFFER. By default, the offer is accepted, but if the DHCPOFFRFLTR shell variable is set, some filtering is done to determine whether or not the client should proceed or just ignore the offer. If the client proceeds, then the target issues the DHCP_REQUEST and the server replies with a DHCP_ACK. The payload accompanying this final acknowledgement is what contains all of the target-specific parameters from DHCP. Similar to BOOTP

above, upon completion of the DHCP handshake, if both BOOTFILE and BOOTSRVR are set, the file specified in BOOTFILE is retrieved from the server specified by BOOTSRVR and executed (if the appropriate flags are set). See warning above.

As mentioned above, the monitor does not support lease expiration; however, it does provide hooks to allow an overlaying OS to handle lease renewal. By default, the incoming lease time option is ignored and the client accepts the server's offer and assumes infinite lease time. If, in the monrc file, the shell variable DHCPLEASETIME is set, then the content of this variable is used as a minimum that the incoming lease time from the server is compared to. If the incoming lease time is greater than or equal to this value, then the offer is accepted and the DHCPLEASETIME shell variable is reloaded with the incoming lease time. If there is no incoming lease time specified by the server, then the DHCPLEASETIME shell variable is cleared. The OS can now look at this shell variable (if present) to see when to issue a lease renewal (if at all).

DM

NAME:

dm

SYNOPSIS:

Display memory.

USAGE:

dm `[-[24bdefmsv:]] {address} {count}`

DESCRIPTION:

There are several different types of memory that can reside on a target.

- Standard memory: used by the CPU for instruction and/or data storage. This is addressed sequentially with the options to specify the width of the memory. The width specification effects both the display format as well as the access type (byte, short, and long) used to retrieve the data.
- FIFO memory: can be read/written to empty/fill a FIFO. This type of access is not sequential, the same address is used for each access and the FIFO itself handles the queuing or de-queuing of the data. This is also width dependent.
- Memory mapped peripherals: have their own unique "qualities" that put restrictions on how they can be accessed.

This command attempts to support all modes of memory access through different options. Width can be specified for 8/16/32 bit access, the exact number of accesses can also be specified; there is no modulo of the user-specified count. This is actually quite important for FIFO and/or memory mapped peripherals because you want to read the EXACT number of units of memory specified.

OPTIONS:

- -2
Retrieve data as 16-bit (2 bytes) wide units.;
- -4
Retrieve data as 32-bit (4 bytes) wide units;
- -b
Dump memory in binary. This is typically used by a host-resident debugger that is connected to the target for faster data retrieval. Width can be specified for 8/16/32 bits.
- -d
Display memory in decimal units instead of the default hex. Width can be specified for 8/16/32 bits.
- -e
Display 2 and 4-byte memory units endian reversed.
- -f
Display memory as if it were a FIFO. The address is not incremented, and width can be specified for 8/16/32 bits.
- -m
Interactively provide a "more?" query to the user to support user-controlled throttling of the output.
- -s
Display the data as a null-terminated ASCII string. Display continues through memory until a NULL is detected. The only width supported here is 8-bit.
- -v {varname}
Load a single data element at location 'address' into the shell variable 'varname'. If -s option is also present, then the variable is loaded with the address after the NULL termination of the string just printed.

EXAMPLES:

- `dm -2 0xa3000000`
Display in short (16-bit) format, a block of memory starting at location 0xa3000000.
- `dm -2f 0xa0100000 32`
Pull 32 units out of a 16-bit FIFO residing at location 0xa0100000 in the memory map.
- `dm -s 0xa0041000 32`
Display memory starting at address 0xa0041000 as if it were an ASCII string. If a null is detected prior to the 32nd byte, terminate; else terminate at the 32nd byte regardless.
- To print a list of concatenated, NULL-terminated strings...
 `set NEXT 0x100000`
 `dm -v NEXT -s $NEXT`
 `dm -v NEXT -s $NEXT`
 `dm -v NEXT -s $NEXT`
 etc...
Each 'dm' command reloads the value of NEXT with the address after the string just printed.

NOTES:

- All command arguments in the monitor are assumed to be of the syntax requiring that hex data be entered with a leading 0x, octal with a leading 0 else assume decimal.

ECHO

NAME:

echo

SYNOPSIS:

Print a string to local terminal.

USAGE:

echo [arg1] [arg2] ...

DESCRIPTION:

This command prints its arguments (separated by blanks and terminated by newline) to the local terminal connection. If there are no arguments, a blank line is printed. A few backslash characters are accepted:

\b	backspace
\c	no newline
\n	newline
\r	carriage return
\x##	ASCII-coded hex
\\	backslash

EDIT

NAME:

edit

SYNOPSIS:

Line-mode file editor for use with TFS.

USAGE:

edit `[-[b:c:f:i:m:rs:]] {filename}`

DESCRIPTION:

This command allows the user to edit ASCII files that are stored in TFS. It is a simple line-based file editor that supports line insertion, deletion, display and search. If the file already exists in TFS, then the content of that file is copied to a buffer in ram space and all interaction and modification with the content of the file is done in the buffer. It is not until the 'q' (quit) command is issued, that the file is written to flash. If at any point during the edit session, the 'x' (exit) command is issued, then there is no change to the original file.

OPTIONS:

- `-b {addr}`
specify the buffer address that edit is to use for temporary storage of the file while being edited; if not specified, then edit assumes it owns all RAM on the board and the buffer starts at the address specified in the APPRAMBASE shell variable.;
- `-c {cmd}`
in-line command executed prior to entering interactive mode;
- `-f {flags}`
flags that are applied to the newly created file (see TFS for flag description);
- `-i {info}`
information field applied to the newly created file (see TFS);
- `-m {size}`
use monitor's malloc to allocate buffer space
- `-r`
edit defaults to automatically remove carriage returns from the file it is editing, this shuts off that automatic removal;
- `-s {size}`
size of buffer to use for temporary storage.

NOTES:

- A typical usage of the edit command will put the user in an interactive mode that supports a basic set of editing commands. At the startup of the interactive mode, edit will display the address of the buffer it will be using for temporary storage, followed by the message "*type ? for help*". Following is a list of commands supported for interactive mode...

`{PRIVATE}d{L`delete line specified by "LRNG" (see below)

`RNG}`

`e#` edit line # (uses the same line editor as is used by the command line editor)

`i` begin "insert" mode (use '.' to exit insert mode)

`a` begin "append" mode (use '.' to exit append mode)

`P[LRNG]` print entire buffer with line numbers prepended

`p` print entire buffer

`q[fname]` quit edit, write file (with no fname specified) it writes the to file originally

MicroMonitor
Introduction and Command Set

	opened
s[srchstr]	go to next line that contains "srchstr"
x	exit edit, do not write file
#	go to line # (use '\$' to go to last line)
+/-#	go to line relative to current position

Where...

represents a decimal number;

LRNG represents a line number or inclusive line range (# or #-#);

Let's work through a simple example...

Assume we have the file 'monrc'. We want to change a line in monrc. We will simply delete the line and re-enter

the new line (there are other ways, but lets keep the example simple). For the sake of this example, the response to each command will be in red, comments will be in blue and commands entered in bold.

The output of "tfs cat monrc" is as follows...

```
{PRIVATE}uMON tfs cat monrc
```

Display the current content of the file.

```
set ETHERSPEED 10
set PROMPT "maint "
set MONFLAGS nophdr
set GIPADD 135.17.115.1
set IPADD 135.17.115.215
set NETMASK 255.255.255.0
uMON edit monrc
```

Start the editing process.

```
Edit buffer = 0xf000c000.
Type '?' for help
```

Print entire file with line numbers prepended.

```
P
1: set ETHERSPEED 10
2: set PROMPT "maint "
3: set MONFLAGS nophdr
4: set GIPADD 135.17.115.1
5: set IPADD 135.17.115.215
6: set NETMASK 255.255.255.0
```

Delete line number 4

Print and notice the removal of the old line 4...

```
d4
P
1: set ETHERSPEED 10
2: set PROMPT "maint "
3: set MONFLAGS nophdr
4: set IPADD 135.17.115.215
5: set NETMASK 255.255.255.0
```

Move "current position" pointer to line 3

Print and notice the pointer moved to line 3...

```
3
P
1: set ETHERSPEED 10
2: set PROMPT "maint "
3: set MONFLAGS nophdr
4: set IPADD 135.17.115.215
5: set NETMASK 255.255.255.0
```

Go into append mode (after current line pointer)

<-- This is the text being added

Terminate append mode

Print and notice new line 4 and pointer at line 5...

```
a
set GIPADD 135.17.115.2
.
P
1: set ETHERSPEED 10
2: set PROMPT "maint "
3: set MONFLAGS nophdr
4: set GIPADD 135.17.115.2
5: set IPADD 135.17.115.215
6: set NETMASK 255.255.255.0
```

Write the new file to TFS and quit the editor

Display the new file with the modification...

```
q
uMON tfs cat monrc
```

MicroMonitor

Introduction and Command Set

```
set ETHERSPEED 10
set PROMPT "maint "
set MONFLAGS nophdr
set GIPADD 135.17.115.2
set IPADD 135.17.115.215
set NETMASK 255.255.255.0
uMON
```

All done!

ETHER

NAME:

ether

SYNOPSIS:

Ethernet driver interface

USAGE:

ether -[pv:V] {on | off | stat}

DESCRIPTION:

When the monitor starts up, the ethernet interface is enabled. This command allows the user to display current statistics (errors, packet counts, etc) and optionally turn the interface off and/or on.

OPTIONS:

- -p
accept all packets (promiscuous mode);
- -v {vflags}
enable specific verbosity:
 - i incoming packets (minus broadcast)
 - I incoming packets (including broadcast)
 - o outgoing packets
 - d DHCP packets
 - t TFTP packets
 - x hex dump (requires i, I or o);
- -V
enable all verbosity (same as -vlodx)

EXAMPLES:

- ether -vio on
Enable the ethernet interface with verbosity on for incoming and outgoing packets.

EXIT

NAME:

exit

SYNOPSIS:

Terminate a script from any point within that script.

USAGE:

exit [-r]

DESCRIPTION:

This provides a clean and simple way to terminate a script from anywhere within the script. If the -r option is specified, then the script will automatically be deleted after exit is complete and the file has been closed under TFS. This is useful for a script that needs to be run only once, it can be placed in TFS as an autobootable script, then after a reset it will do its thing and delete.

OPTIONS:

- -r
After exiting from the script, the file is automatically removed from TFS.

EXAMPLE:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
if CNT lt 10 goto NEXT
exit

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

FLASH

NAME:

flash

SYNOPSIS:

Flash memory operations.

USAGE:

flash {operation} [args]

DESCRIPTION:

Flash is writeable non-volatile memory. This command supports the handshaking necessary to erase and write to the flash devices on the target. The code that makes up the flash command in the monitor supports multiple banks of flash, and also allows each bank to be a different bus width. This, of course, is dependent on the underlying hardware.

The flash commands support the idea of sector protection. This is different than the sector protection supported by the flash device. The protection provided by the flash command is simply a 1-command window that is opened with the "flash opw" command. This allows the next command to execute operations on the flash ignoring the fact that the particular sector may be protected. Typically, the monitor configures the sectors that are used by the monitor's text and data space as protected; all other sectors are left unprotected.

There are several operations supported...

- "bank [#]"
If no argument is specified, this command returns the currently active bank; if an argument is provided, then that value becomes the new default bank. Note that the number of banks supported is hardware-platform dependent.
- "init"
Initializes all internal flash operation data structures (monitor development only).
- "info"
Queries each bank configured in the hardware and displays the state that the monitor is assuming valid for the flash on board.
- "opw"
Enables a flag that allows the very next command to the monitor to be a flash operation that will modify the protected sectors of the flash. By default the monitor knows what parts of the flash must be protected from random writes. This command provides the user with a "1-command-window" to override that protection. Must be in user-level 3 mode to do this.
- "erase {sector # | all}"
Allows the user to erase a specified sector of the flash or all of the non-protected sectors (all). The erasure brings all bits to 1; hence the data would be read as 0xffs.
- "ewrite {dest} {src} {bytecnt}"
This command allows the user (typically only the owner of the monitor code) to re-write the protected sectors with a new code. This essentially allows the monitor to re-write itself. The arguments are a destination address where the data is to be written in flash, the source address of the data to be placed in flash and the number of bytes to be written.
- "write {dest} {src} {bytecnt}"
This command allows the user write to the flash. Like the ewrite command, it assumes that the data to be written is at the location specified by "src" and the number of bytes is specified by "bytecnt".

Some Discussion on Flash Protection:

The monitor does not require any hardware-implemented flash protection; yet it still provides a reasonably secure means of protecting the flash from accidental overwrite or erasure. The flash operations, as implemented by the manufacturer, make it extremely unlikely that random code would do any damage to flash; however, if you have flash on your system, it is likely that somewhere in your instruction space there is a function that is written to do these modifications. This means that a corrupt function pointer, could potentially call the block of code that will do damage (corrupt data) to the device. Obviously, the monitor has these functions. But with the functions comes a requirement that a previous function be called to modify a flag that must be set for the "dangerous" function to do the damage. This means that two things must "accidentally" happen in the proper sequence:

- ? The flag must be modified
- ? The function that does the flash operation must be called

The likelihood of this occurring greatly decreases the chances of messing up your flash.

There is the option of using some kind of hardware protection, and there are several different ways this can be implemented. A lot of the variation is caused by the fact that different devices offer different types of protection and mechanisms to temporarily "unprotect" sectors. The following list is in order based on the amount of the flash device that actually needs the hardware protection and does not consider the temporary "unprotect" mode that some newer devices now have...

Protect the entire device

This is obviously the safest, but it also defeats the whole purpose behind using flash. The device would be programmed in the factory with a monitor and the files to be in TFS space. The monitor and TFS files would be unchangeable. If there is no intent to ever change things, then do this.

Protect the monitor space and the first few sectors of TFS space

Similar to the above case, the monitor and some of the space allocated to TFS (based on a sector boundary) can be protected. Perhaps a backup application and the monrc file would be in the protected TFS space, then the remaining space would be open for update. This locks in some base application that could run if all else failed.

Protect the monitor space only

This would allow the application to be upgradeable in the field, without fear of ever losing total control of a system because worst case scenario the monitor would be able take over and allow the system to be re-established regardless of the level of corruption that has occurred in the non-protected sectors of the flash. The only disadvantage to this (as with the one above this) is that the monitor itself cannot be updated.

Protect a boot-sector

With this model, the monitor is built as two separate pieces: a boot-sector startup and the monitor itself. Here the only piece of code that is hardware protected is a small boot-sector that would contain a crc32 checker and xmodem (or a not-yet-implemented tftp) downloader... It would boot the hardware and immediately do a check to see if the monitor is sane... If it is, then jump into it; if it isn't, then prompt the user for a new monitor and wait there forever. Once a new monitor is downloaded into RAM it can be used to rebuild the corrupted flash. This allows the monitor to be safely upgraded in the field without fear of losing the system if the upgrade fails for some reason. It also allows the flash to be programmed and protected in bulk without losing the ability to upgrade the monitor and all the applications in TFS.

No hardware protection at all

This model works fine, but it is a little risky. There is the chance that the boot code can be corrupted (and if it was, the system would need to come back to the factory or lab); but, as discussed above, it is extremely unlikely and depending on the project, the above options may not be worth the added overhead of sector-protecting some of the device in the factory.

FM

NAME:

fm

SYNOPSIS:

Fill a memory range with a specified value.

USAGE:

fm -[c24] {source address} {destination address | count} {value}

DESCRIPTION:

Fill a block of memory with a specified value.

OPTIONS:

- -2
Access data as 16-bit (2 bytes) wide units.;
- -4
Access data as 32-bit (4 bytes) wide units;
- -c
Arg2 is count, in bytes.

EXAMPLES:

- fm -2 0x400000 0x402000 0x400
Fill memory range from 0x400000 through 0x402000 with 0x400. Access as 16-bit memory.
- fm -c 0x400000 0x2000 0x99
Fill 0x2000 bytes of memory starting at 0x400000 with 0x99.

NOTES:

- All command arguments in the monitor are assumed to be of the syntax requiring that hex data be entered with a leading 0x, octal with a leading 0 else assume decimal.

GO

NAME:

go

SYNOPSIS:

Turn over control to a resident (downloaded or TFS) application.

USAGE:

go -[p:Rr:s:u:] [address]

DESCRIPTION:

This command allows the user to turn over control to an application (client) with its own stack space, and a copy of the register set maintained across exceptions. If no arguments are specified, then the context currently stored in the accessible register set is loaded prior to control transfer.

OPTIONS:

- -p {PC}
specify PC to use upon entry into application;
- -R
dump registers prior to control transfer;
- -r {SR}
specify status register value to be used upon entry into application;
- -s {SSP}
specify supervisor stack pointer;
- -u {USP}
specify user stack pointer;

EXAMPLES:

- go 0x405000
Turn over control to an application whose starting point is 0x405000.

GOSUB

NAME:

gosub

SYNOPSIS:

Call a subroutine within the current script.

USAGE:

gosub {tag}

DESCRIPTION:

As an extension to the goto capability within TFS scripts, this command branches to the named tag, and assumes that at some point the code branched to will execute a return. At that point, the script will continue execution on the line after the gosub line.

EXAMPLE:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
goto NEXT

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

GOTO

NAME:

goto

SYNOPSIS:

Branch to a tagged line in a TFS script.

USAGE:

goto {tag}

DESCRIPTION:

Since TFS supports executable files that can be a set of monitor commands, this command enhances that capability by allowing the script to branch to specific tags within the script. The tag is simply a line starting with a # (pound sign), one blank and a tag. For example, # TAG on a line by itself is a target that could be branched to by the command *goto TAG*.

EXAMPLES:

- Within a script file:

```
# TOPofScript
set CNT 0

# ECHO_CNT
echo $CNT
set -i CNT
sleep 1
goto ECHO_CNT
```

HEAP

NAME:

heap

SYNOPSIS:

Display/modify current heap statistics.

USAGE:

heap -[cf:m:vX:x]

DESCRIPTION:

This command allows the user to peek into the internal structures allocated by the monitors memory allocator. With no options specified, a summary of the heap statistics is printed; add -v to dump all structures within the heap.

OPTIONS:

- -c
clear high-water level and malloc/free totals
- -f {pointer}
call "free" to release the block of memory pointed to by "pointer";
- -m {size}
call "malloc" to allocate a block of memory of size "size";
This will load the shell variable MALLOC with the result.
- -v
verbose mode
- -X {start,size}
extend the monitor's heap using "size" bytes beginning at "start";
- -x
disable the monitor's extended heap (if not in use)

EXAMPLES:

- heap
Dump heap summary, i.e...

```
Heap summary:
Malloc/free calls: 106/103 (delta=3)
Malloc/free totals: 270096/270040
High-water level: 46876
Malloc failures: 0
Bytes overhead: 1020
Bytes currently allocated: 644
Bytes free on current heap: 50856
Bytes left in allocation pool: 479960
```

- heap -v
Verbose listing of heap statistics, i.e...

```
   addr size free? mptr  nxt  prv  ascii@addr
0: 0x0000288c 12 n 0x00002878 0x00002898 0x00000000 .....(.....
1: 0x000028ac 36 n 0x00002898 0x000028d0 0x00002878 ENTRYPOINT.....
2: 0x000028e4 12 n 0x000028d0 0x000028f0 0x00002898 ..)...)<
3: 0x00002904 8 n 0x000028f0 0x0000290c 0x000028d0 PROMPT..
4: 0x00002920 8 n 0x0000290c 0x00002928 0x000028f0 uMON...
5: 0x0000293c 12 n 0x00002928 0x00002948 0x0000290c ..)|..)|..)
```

```

6: 0x0000295c 12 n 0x00002948 0x00002968 0x00002928 APPRAMBASE..
7: 0x0000297c 8 n 0x00002968 0x00002984 0x00002948 0xf000..
8: 0x00002998 12 n 0x00002984 0x000029a4 0x00002968 .....)...).
9: 0x000029b8 12 n 0x000029a4 0x000029c4 0x00002984 BOCTROMBASE.
10: 0x000029d8 12 n 0x000029c4 0x000029e4 0x000029a4 0x80000000..
11: 0x000029f8 12 n 0x000029e4 0x00002a04 0x000029c4 ..*8..*...*T
12: 0x00002a18 12 n 0x00002a04 0x00002a24 0x000029e4 CONSOLEBAUD.
13: 0x00002a38 8 n 0x00002a24 0x00002a40 0x00002a04 19200...
14: 0x00002a54 12 n 0x00002a40 0x00002a60 0x00002a24 ..*...*t...*.
15: 0x00002a74 12 n 0x00002a60 0x00002a80 0x00002a40 ETHERADD....
16: 0x00002a94 20 n 0x00002a80 0x00002aa8 0x00002a60 00:60:1D:02:0b:f
17: 0x00002abc 12 n 0x00002aa8 0x00002ac8 0x00002a80 ..*...*...+.
18: 0x00002adc 8 n 0x00002ac8 0x00002ae4 0x00002aa8 IPADD...
19: 0x00002af8 16 n 0x00002ae4 0x00002b08 0x00002ac8 135.3.94.136....
20: 0x00002b1c 12 n 0x00002b08 0x00002b28 0x00002ae4 ..+X..+<..+x
21: 0x00002b3c 8 n 0x00002b28 0x00002b44 0x00002b08 HIPADD..
22: 0x00002b58 12 n 0x00002b44 0x00002b64 0x00002b28 135.3.94.76.
23: 0x00002b78 12 n 0x00002b64 0x00002b84 0x00002b44 ..+...+...+.
24: 0x00002b98 8 n 0x00002b84 0x00002ba0 0x00002b64 GIPADD..
25: 0x00002bb4 12 n 0x00002ba0 0x00002bc0 0x00002b84 135.3.94.1..
26: 0x00002bd4 12 n 0x00002bc0 0x00002be0 0x00002ba0 .....+...4
27: 0x00002bf4 8 n 0x00002be0 0x00002bfc 0x00002bc0 NETMASK.
28: 0x00002c10 16 n 0x00002bfc 0x00002c20 0x00002be0 255.255.255.0...
29: 0x00002c34 12 n 0x00002c20 0x00002c40 0x00002bfc ...t...T...
30: 0x00002c54 12 n 0x00002c40 0x00002c60 0x00002c20 DNS_SERVER..
31: 0x00002c74 12 n 0x00002c60 0x00002c80 0x00002c40 135.3.29.32.
32: 0x00002c94 12 n 0x00002c80 0x00002ca0 0x00002c60 .....
33: 0x00002cb4 12 n 0x00002ca0 0x00002cc0 0x00002c80 DNS_DOMAIN..
34: 0x00002cd4 12 n 0x00002cc0 0x00002ce0 0x00002ca0 lucent.com..
35: 0x00002cf4 12 n 0x00002ce0 0x00002d00 0x00002cc0 ..-4...-L
36: 0x00002d14 12 n 0x00002d00 0x00002d20 0x00002ce0 TIME_GMTD...
37: 0x00002d34 4 n 0x00002d20 0x00002d38 0x00002d00 -5..
38: 0x00002d4c 12 n 0x00002d38 0x00002d58 0x00002d20 ..-...-1...-
39: 0x00002d6c 12 n 0x00002d58 0x00002d78 0x00002d38 TIME_PORT...
40: 0x00002d8c 4 n 0x00002d78 0x00002d90 0x00002d58 37..
41: 0x00002da4 12 n 0x00002d90 0x00002db0 0x00002d78 ...-...-.....
42: 0x00002dc4 12 n 0x00002db0 0x00002dd0 0x00002d90 TIME_SVR...
43: 0x00002de4 12 n 0x00002dd0 0x00002df0 0x00002db0 135.3.94.36.
44: 0x00002e04 12 n 0x00002df0 0x00002e10 0x00002dd0 ...H...S...(.
45: 0x00002e24 16 n 0x00002e10 0x00002e34 0x00002df0 BOOT_LINE_ADRS..
46: 0x00002e48 8 n 0x00002e34 0x00002e50 0x00002e10 0x17000.
47: 0x00002e64 80 n 0x00002e50 0x00002eb4 0x00002e34 ..D...=P.E...F.
48: 0x00002ec8 8 n 0x00002eb4 0x00002ed0 0x00002e50 0x18000.
49: 0x00002ee4 6548 y 0x00002ed0 0x00200000 0x00002eb4 .....8...9x....
50: 0x00200014 44308 y 0x00200000 0x00000000 0x00002ed0 .....
Malloc/free calls: 106/103 (delta=3)
Malloc/free totals: 270096/270040
High-water level: 46876
Malloc failures: 0
Bytes overhead: 1020
Bytes currently allocated: 644
Bytes free on current heap: 50856
Bytes left in allocation pool: 479960

```

- heap -X 0x200000,0x80000
Extend the monitor's heap by 0x80000 bytes using memory at location 0x200000.

NOTE:

The functionality of the -h and -H options was originally handled by the mstat command. This new command was created to support additional heap control functionality, so the options were removed from mstat. The functionality of -h is replaced with "heap" and -H is replaced with "heap -v". Originally there were two shell variables used to extend the monitor's heap. This has also been replaced with the -X option where "start" is what used to be stored in HEAPSPACE and size is what used to be stored in HEAPSIZ.

After an extension is made to the amount of memory available for malloc (using -x); that extension can be "undone" with -d ONLY if there are no active allocations within the extension space.

HELP

NAME:

help (or "?")

SYNOPSIS:

Display command information.

USAGE:

help `[-d]` `[command name]`

DESCRIPTION:

Displays help text for a specific command (requires command name to be specified) or display a tabular listing of all commands available. Note that the string "help" or the character "?" will be seen by the monitor's command interpreter as the same command.

OPTIONS:

- `-d`
when displaying all commands available, a per-command description is included.

HISTORY

NAME:

history

SYNOPSIS:

Display history of most recent commands issued at monitor command line.

USAGE:

history

DESCRIPTION:

The monitor supports command line editing capabilities. Along with that, a history is maintained of some number of the most recently executed commands. The command line editor/history facility supports a subset of "vi-like" command sequences used by KSH:

{PRIVA	step back through command history
TE}esc	
[k-]	
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
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

ICMP

NAME:

icmp

SYNOPSIS:

ICMP operation support.

USAGE:

icmp -[c:d:f:rv:] {operation} [args]

DESCRIPTION:

Allows the user to run specific ICMP-based queries on the local LAN. Currently, the ICMP TIMESTAMP and ECHO are the only supported operations.

OPTIONS:

- -c {##}
repetition count for echo;
- -d {##}
delta (in hours) relative to GMT;
- -f {x|d}
hex or decimal output (default is ASCII);
- -r
check server resolution;
- v {varname}
load result into shell var "varname".

OPERATIONS:

- time
Issue ICMP TIMESTAMP request.
- echo
Basic ping capability.

EXAMPLES:

- icmp time 135.3.29.35
Issue an ICMP time request to server on 135.3.29.35.
- icmp echo 135.3.29.39
Similar to ping 135.3.29.39.

NOTES:

- The *d*, *f*, *r* & *v* options are for the time operation and *c* is for echo.

IDEV

NAME:

idev

SYNOPSIS:

Device interface.

USAGE:

idev [-d:] [command] [command-specific arg(s)]

DESCRIPTION:

This command is used on target systems that have a set of devices that are to be managed. Typically, this is used for systems that have multiple serial ports and currently, that is its only use; however, it is not confined to that. The command allows the user interface to a device and reconfigure various parameters applicable to that device. ***Note that this command is not applicable to all target systems.***

OPTIONS:

- -d {device id}
by default, the command is applied to the device that interfaces to the Console port, but this option allows that to be overridden.

SUB-COMMANDS:

- console
this command allows the user to configure a different device to be the Console device. Since the default action is taken on the console device, using this sub-command without the -d option is of no use.
- baud {baudrate}
specify a device to run at a newly set baudrate.

IF

NAME:

if

SYNOPSIS:

Conditional test with branching.

USAGE:

if -[t:] [{arg1} {compare} {arg2}] {action} else {action}

DESCRIPTION:

To support scripting capability through TFS files, this commands allows the user to build conditional tests that result it branches to different points within a executable script. The most common use of this is to make a string or numerical comparison between two different arguments. If the -t option is specified, then that overrides the basic comparison and the argument to -t is considered to be the test.

Numerical comparisons:

- gt
true if arg1 is greater than arg2
- lt
true if arg1 is less than arg2
- le
true if arg1 is less than or equal to arg2
- ge
true if arg1 is greater than or equal to arg2
- eq
true if arg1 is equal to arg2
- ne
true if arg1 is not equal to arg2

Logical comparisons:

- and
true if arg1 AND arg2 is non-zero
- or
true if arg1 OR arg2 is non-zero

String comparisons:

- seq
true if the string of arg1 is identical to the string of arg2
- sne
true if the string of arg1 is not identical to the string of arg2

Actions:

- goto tag
Jump to the location in the script specified by "tag".
- gosub tag
Call the subroutine specified by "tag"
- return
Return from the currently active subroutine.
- exit
Terminate the currently active script.

OPTIONS:

- -t {testtype}
override the default "arg1 compare arg2" comparison;
testtypes:

{PRIVATE}gc	"gotchar"... a character is present on the UART interface
ngc	"not-gotchar"... a character is not present on the UART interface
iscmp {filename}	"is-compressed"... specified file is compressed

EXAMPLES:

- if \$VAR1 seq \$STR1 goto MATCH
if the string contained within \$VAR1 is the same as the string contained in \$STR1 then branch to the line containing the tag 'MATCH'.
- if -t gc goto GOT_ONE
if there is a character pending on the UART interface, branch to the line containing the tag 'GOT_ONE'.
- if -t iscmp app_file gosub COMPRESSED
if the file "app_file" is compressed, then call the subroutine 'COMPRESSED'.
- if \$VAR seq \ \$VAR goto VAR_NOT_SET
If the shell variable 'VAR' does not exist, branch to the subroutine 'VAR_NOT_SET'

ITEM

NAME:

item

SYNOPSIS:

Process a list of strings.

USAGE:

item {idx} {stor_var} [item1] [item2] [item3]

DESCRIPTION:

This command, in conjunction with "if" and "goto", allows the user to build scripts that conveniently process a list of strings (or items). Consider the 3rd through Nth arguments (itemN above) to be a list or table of items. The first argument (idx) is used as the index into this list (starting with 1) and the second argument is the name of the shell variable into which the item is to be placed. If the index is out of range, the variable is cleared.

EXAMPLES:

- item 2 letter a b c d e f
This would place 'b' in the shell variable "letter"
- item \$idx letter a b c d e f g
If the content of \$idx was between 1 and 7, then the shell variable "letter" would contain one of the letters a through g respectively. For any other value in \$idx, the shell variable "letter" would be empty.

MSTAT

NAME:

mstat

SYNOPSIS:

Display/modify monitor status.

USAGE:

mstat -[b:cs:]

DESCRIPTION:

Modify or display the current monitor status. If no options are entered, some monitor specific information is dumped to the user interface.

OPTIONS:

- -b {baud}
re-define the current baudrate of the UART interface;
- -c
display chip-select configuration
- -s {val}
set monitor state to value specified (monitor development use only).

EXAMPLES:

- mstat -b 115200
Set baudrate on console to 115200.

MT

NAME:

mt

SYNOPSIS:

Run memory diagnostics

USAGE:

mt -[qv] {address} {size}

DESCRIPTION:

Runs walking ones and address-on-address test across the memory range specified.

OPTIONS:

- -q
quit at error;
- -v
verbose mode;

PIO

NAME:

pio

SYNOPSIS:

Interface with the processor's parallel io ports.

USAGE:

pio `-[iogdw] {A|B} {bit #} [0|1]`

DESCRIPTION:

Allows the user to manually override the basic configuration of the processor's parallel input-output (PIO) pins. Options allow the user to reconfigure for input, output, dedicated and also support the ability to wait until a specified bit is a specified value. Note that this command is quite specific to the underlying hardware and 68EN302 CPU.

OPTIONS:

- `-i`
configure the specified bit as input ;
- `-o`
configure the specified bit as output;
- `-g`
configure the specified bit as general purpose io;
- `-d`
configure the specified bit as dedicated;
- `-w`
wait for the specified bit to be the specified value;

EXAMPLES:

- `pio -gi A1`
Configure pio bit 1 of port A as general purpose IO, input

PM

NAME:

pm

SYNOPSIS:

Patch memory local to the target.

USAGE:

pm -[24efSs] {address} {value | string} [value]

DESCRIPTION:

Refer to the dm command description for a discussion on the different types of memory that can reside on a target. This command attempts to support all modes of memory access through different options. Width can be specified for 8/16/32 bit access, the exact number of accesses can also be specified; there is no modulo of the user-specified count. This is actually quite important for FIFO and/or memory mapped peripherals because you want to read the EXACT number of units of memory specified.

OPTIONS:

- -2
assume the width of the memory accesses is x2;
- -4
assume the width of the memory accesses is x4;
- -e
for data written as 2 and 4-byte units, endian-swap prior to placing into memory location;
- -f
assume the destination address is a FIFO;
- -s
assume the data is an ASCII string and process `\t\r\n\b` appropriately
- -S
similar to -s except that the string is concatenated to the end of the string specified by the start address.

EXAMPLES:

- pm -2 0xa3000000 0x4411
Place in short (16-bit) format, 0x4411 at location 0xa3000000
- pm -f 0xa010000 0x32 0x33 0x34 0x35
Assume a FIFO is memory mapped at address 0xa010000 and push those four bytes onto it.
- pm -s 0x401000 "hello world"
Put the string "hello world" into memory at location 0x401000 and terminate it with a 0x00.
- pm -S 0x401000 " again"
Concatenate the string that starts at address 0x401000 with the string " again".

READ

NAME:

read

SYNOPSIS:

Interactive shell variable entry.

USAGE:

read -[ct:T:] {varname1} [varname2]

DESCRIPTION:

This command provides the monitors ability to run a script from TFS with the ability to interact with the user by requesting that the user input data that will be loaded into a specified shell variable.

OPTIONS:

- -c
special case that simply waits for any COM-port input (like hit-any-key) to continue;
- -t {###}
wait for input, but timeout after ### milliseconds, timeout is only applied to initial wait prior to receiving first character.
- -T{###}
wait for input, but timeout after ### milliseconds, timeout is restarted after each character is received.

EXAMPLES:

- read name
Wait forever for input from the user to load into the shell variable *name*
- read -t3000 name
Wait for up to 3 seconds for the user to enter data into the shell variable *name*.
- read -c
Wait for any input on the user interface.

REG

NAME:

reg

SYNOPSIS:

Display or modify a value in the current monitor register cache.

USAGE:

reg [regname] [value]

DESCRIPTION:

Some ports of the monitor support debugging of an application. Upon hitting a breakpoint, the exception handler copies the CPU's context (registers) to a monitor-maintained array (or register cache). Then while the monitor is active the user has the ability to display and/or modify the value of any of these registers. It is important to note that this command does NOT affect the current value of the register, it is the register cache that is affected.

RESET

NAME:

reset

SYNOPSIS:

Soft reset the monitor firmware.

USAGE:

reset `-[t:x]`

DESCRIPTION:

Request that the monitor firmware re-enter at various points in its own context. With no options, the reset is as close to a hard reset as can be supported by the firmware.

OPTIONS:

- `-x`
restart the monitor firmware just as it would be restarted if the application had exited;
- `-t {##}`
restart the monitor with the type specified (monitor development)

RETURN

NAME:

return

SYNOPSIS:

Return from subroutine.

USAGE:

return

DESCRIPTION:

Return to the point at which the currently active subroutine was called via gosub..

EXAMPLES:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
if CNT lt 10 goto NEXT
exit

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

SET

NAME:

set

SYNOPSIS:

Set, clear or adjust a shell variable within the monitor.

USAGE:

set -[adF:iox] [varname] [value]

DESCRIPTION:

The monitor allows the user to set up shell variables that can then be used by other commands with the monitor or by the client application through the *getenv()* shared library call. This capability provides a clean interface between monitor scripts and applications; thus, allowing the script to define operating characteristics that are activated by the application through the applications use of *getenv()*.

Keep in mind that the shell variables are generic in nature, so they can be used for general-purpose command line substitution as well.

If no arguments are present, *set* will simply dump the current set of environment variables.

OPTIONS:

- -a
logically AND the content of the shell variable with the value specified;
- -i
increment the content of the shell variable by the value specified (or 1 if value is not specified);
- -d
decrement the content of the shell variable by the value specified (or 1 if value is not specified);
- -F {filename}
build an executable script file that could be used to re-create the set of shell variables;
- -o
logically OR the content of the shell variable with the value specified;
- -x
output the result of the -i or -d operation in hex (else decimal).

SLEEP

NAME:

sleep

SYNOPSIS:

Delay for a specified number of seconds (or milliseconds).

USAGE:

sleep `[-[clmv:] {time (seconds/milliseconds) | new loops-per-second count}`

DESCRIPTION:

Used within a TFS script, this command simply puts the processor in a delay loop for a specified amount of time. Note that the monitor does not use any CPU facilities; hence, it is not truly aware of time because there is no dedicated interrupt handler that has a known frequency. To support the monitors ability to have a relatively close approximation of a second of delay, the monitor is built with a default loop size. If the sleep time does not appear to be close to the specified number of seconds (or milliseconds), it can be calibrated with the `-c` option.

If no options or arguments are specified, sleep will return the current loops-per-second (LPS) count.

OPTIONS:

- `-c`
calibrate with a new loop value that will be used in subsequent calls;
- `-l`
store the count specified as the new loops-per-second count;
- `-m`
use milliseconds instead of seconds;
- `-v {varname}`
load the specified shell variable with the current LPS count;

EXAMPLES:

- `sleep`
Display the current LPS count value.
- `sleep 3`
Delay for approximately 3 seconds.
- `sleep -m 300`
Delay for approximately 300 milliseconds
- `sleep -l 100000`
Redefine the loop count used for the delay to 100000 (non-interactive)
- `sleep -c 100000`
Display the old loop count, print out 10 dots each with a delay that corresponds to the new loop count, load 100000 as the new loop count.

NOTES:

- The calibration simply outputs 10 dots (.) at a rate specified by the new loopvalue (argument to the `-c` option). If the rate of the dot output is not 1 per second, adjust the value accordingly. The `-l` option is simply a non-interactive version of `-c`. Both `-l` and `-c` will modify the loops-per-second count used in subsequent calls. There are a few other internal timeouts that also use the loops-per-second value used by sleep; so adjustment of this duration will adjust other modules' timeout accuracy as well.

SM

NAME:

sm

SYNOPSIS:

Search memory.

USAGE:

sm `[-[24sx]` {start address} {finish address} {search_value}

DESCRIPTION:

Search through memory for a specified value. This value can be a byte, short, long value or a ASCII-coded hex string or straight ASCII string.

OPTIONS:

- -2
assume the width of the memory accesses is x2
- -4
assume the width of the memory accesses is x4
- -s
search_value is a string and the access is byte-wide
- -x
search-value is a ASCII-coded hex string and the access is byte-wide.

SS

NAME:

ss

SYNOPSIS:

Single step.

USAGE:

ss `[-oq]` `[count]`

DESCRIPTION:

Single-step through the application code one assembler line at a time. Optionally, a count of lines can be specified to step over several lines of assembly in one command. After each line is executed, the next instruction that is to be called is displayed (if `-q` is not specified).

OPTIONS:

- `-o`
step over the code that will be jumped into as a subroutine;
- `-q`
step quietly, do not display the next line of assembly to be executed;

NOTES:

- The availability of this command depends on the target port. As with all the debug commands (`dr`, `pr`, `at`, & `ss`), check with the port before assuming this command is functional on your target system.

STRACE

NAME:

strace

SYNOPSIS:

Stack trace (used after exception or breakpoint).

USAGE:

strace

DESCRIPTION:

In most cases, the user of the monitor is running some higher level package that has a complete debugging environment included. For those cases, this command is likely to be of little value. On the other hand, if you are using a system that does not trap exceptions, or allows the user to redefine some of the exception handlers, then this is useful for determining just how an exception occurred.

For best results, this command looks for a symbol table file to symbolically dump a function name instead of an address within a function. In some cases, the stack trace actually requires that the symbol table file be there to work.

EXAMPLES:

- strace
Dumps the current function nesting, showing the function name and the hex offset into the function.

NOTES:

- Because of the cpu-specific nature of this command, it may not be ported to all versions of the monitor. As of this writing, strace is available for power-pc and SH2/4 platforms. Additional ports will be done as the need arises.
- The monitor allows small applications to be run directly out of the monitor's own stack space. Be aware that strace will not work for those cases simply because the monitor will be reusing the same stack that the application had been using. Bottom line... if you want to use strace, then establish a stack frame for the application.

TFS

NAME:

tfs

SYNOPSIS:

Tiny file system command line interface.

USAGE:

tfs -[d:f:i:mv] {operation} [args]

DESCRIPTION:

TFS provides a small "file-system-like" capability on a target that contains FLASH memory. It provides a core set of commands that allow the user to add, delete, list, display, execute, load, and copy files within flash memory. In addition to the user interface command "TFS" there is also a set of application entry points (or functions) that the application can access through a shared vector table seen by both the monitor and the application. This allows the monitor to store files that can be accessed by the application. It provides a clean way to interface to non-volatile memory and also a means to allow a given application to configure itself based on files in the file system instead of through rebuilds of source code.

File Operations:

- **add** {name} {src_address} {size}
Create the file named 'name' to contain the data starting at location "src_address" of size "size". Options -f and -i can be used to specify the flags and information field associated with the newly created file.
- **cat** {name}
Print the specified file. Assumes the file is ascii.
- **check** [varname]
Check the sanity of the files stored in TFS by running various tests (like a crc32 on the data and header). If the -d option is specified, then only check files in the specified device. If a variable name is specified, then that shell variable will be loaded with the string "PASS" or "FAIL" based on the result of the FS check.
- **clean**[r]
Cleanup (defragment) the file system to free-up flash space. If 'r' is appended then the system will automatically restart after the cleanup. If the -d option is present, then only cleanup the device specified.
- **cp** {name} {newfilename | hex address}
Copy the named file to the new named file. If the destination begins with '0x', then it is assumed to be a hex address pointing to RAM. The source file is first copied to memory pointed to by APPRAMBASE, then the new file is created from the data in RAM.
- **freemem** [varname]
Return (or store in 'varname') the amount of flash memory that is still available for use by TFS. If the -d option is specified, then list the memory that is available for that device only. Note that since there is per-file overhead, the value returned here is the amount of data space available if ONE more file is stored in TFS; if additional files are to be stored, then the user must take into account the TFS overhead.
- **info** {filename} {varname}
Load the shell variable specified by 'varname' with the information field stored with the file specified by 'filename'.
- **init**
Initialize the file system (remove all files and erase flash).

- **ld**
Load the executable COFF, ELF or AOUT file from flash to RAM. If the file is compressed, ld will automatically decompress. Refer to decompression discussion on TFS.
- **ldv**
Verify that the executable COFF, ELF or AOUT file in flash space matches what is in RAM space.
- **log** [{on|off} {message}]
Turn on or off (or determine the current state of) the change-log facility.
- **ls** [filter] [filter...]
List the current set of files in the file system. There are 4 different levels of verbosity for ls:
 - lvl 0 No verbosity, short list of all active files.
 - lvl 1 Display "hidden" files (beginning with '.') in short format.
 - lvl 2 Display active files in long format.
 - lvl 3 Display active and deleted files in long format.

Specifying the filter can limit the number of files listed...

**filter* indicates a suffix match

*filter** indicates a prefix match

filter indicates a full filename match

See notes below for some additional details.

- **rm** {filter} [filter ...]
Remove the specified file(s) (see note below). See discussion on "ls" above for details on the filter.
- **rms** {size} [except_file1] [except_file2] ...
Remove space. Remove files until "size" bytes have been removed. Do not remove any of the "except" files. This is primarily used for testing TFS with scripts.
- **run** {name}
Run the specified file based on the creation attributes (see below).
- **size** {filename} {varname}
Load the shell variable specified by 'varname' with the size of the file 'filename'.
- **stat**
Dump the current state of TFS. This includes information regarding the location of multiple non-contiguous directories (if any), the amount of space taken up by file overhead, amount of space that would be released if a defragmentation was done, etc...
- **trace** [trace-level]
Establish a runtime trace of the TFS system calls. If no trace level is specified, then the current level is displayed. This tracing is useful for debugging application code that uses TFS system calls...
 - lvl 0 No runtime trace
 - lvl 1 Trace for all TFS system calls except tfswrite(), tfsread() and tfsgetline().
 - lvl 2 Trace for all TFS system calls including tfswrite(), tfsread() and tfsgetline().
 - lvl 3 Trace for all above plus the flash operations.
- **uname** {prefix} {varname}
Load the shell variable specified by 'varname' with a filename (starting with the specified prefix) that is not currently being used for file storage in TFS.

OPTIONS:

- -d {device-prefix}
apply the command to the specified TFS device
- -f {flags}
flags (see below) applied to the file

- -i {info}
information field included with the file created
- -m
enable 'more' when displaying a file or list of files
- -v
enable verbosity level (-v=lv1, -vv=lv2, -vvv=lv3).
- -x
set the script exit flag if there is an error with the tfs command.

EXAMPLES:

- tfs ls
Displays the current set of files and their attributes in alphabetical order.
- tfs -v list
Displays the current set of standard and hidden files (see notes below for more details on the output).
- tfs -vv list
Displays the current set of files and their data structure verbosely. Note that this list is displayed in the order in which they exist in the file system, not alphabetical.
- tfs -i test -f e add myfile 0xa0100000 459
Add the file "myfile" to the file system. It will contain a information field of "test" and have attributes "e" (executable script). The actual file will contain 459 bytes of data starting at location 0xa0100000.

NOTES:

The user must be aware of the fact that the file system is in flash and that there are certain limitations imposed by the underlying technology and microprocessor to memory interface.

Refer to the tfs page for more detailed information on the file system.

The "tfs ls" output can be formatted with varying levels of additional verbosity (see -v option above). In the simplest case, "tfs ls", the output would look like this:

```

Name                               Size  Location  Flags  Info
//BBRAM/
cfg/
html/
monrc                               177  0xfc0dc99c  e
r2.5e/
dirnam                              6   0xfc274abc
sfile101                           17608 0xfc27059c  BeE
sfile101R                           17608 0xfc00004c  BeE

```

```
Total: 9 items listed (35399 bytes).
```

Notice that although there is no directory hierarchy in TFS, filenames can include a slash and the output of this listing will attempt to treat that as if it was a directory. This allows the user to organize filenames in groups. Also, the above case demonstrates the situation where TFS spans across multiple, non-contiguous devices. //BBRAM/ is a battery backed ram and all other files are in flash. To see all files regardless of the slash, use "tfs ls **", or "tfs ls cfg/*" for a subset. The size of each file shown is the size of the data space within the flash that is used by the file; this does not include the overhead space needed for each file header and the space needed at the end of TFS flash area for defrag (refer to description of tfs overhead for details on this). The total item count displays the total number of entries listed, some of which may be what the "ls" is displaying as if it was a directory. The total bytes shown at the bottom of this listing also, does not include the overhead or files not displayed because of the slash.

The "tfs -vv" ls output displays more information about each file. This includes the information field, file attributes and other header-related items. The output is not in alphabetical order as it is for the lower level of verbosity and the default filter ignores the slash within the filename. The output in this level is in the order the files are stored in flash. One additional note for this level of verbosity, the total size

displayed includes the overhead of the file (header and defrag space needed). Following is an example of the verbose output...

```
Name: 'monrc'  
Info: ''  
Flags: executable,  
Addr: 0xfc0dc99c (hdr @ 0xfc0dc950, nnextptr = 0xfc0dca50)  
Size: 177 bytes  
Name: 'dirnam'  
Info: ''  
Flags:  
Addr: 0xfc274abc (hdr @ 0xfc274a70, nnextptr = 0xfc274ad0)  
Size: 6 bytes  
Name: 'sfile101R'  
Info: ''  
Flags: gry_run_at_boot, executable, elf,  
Addr: 0xfc00004c (hdr @ 0xfc000000, nnextptr = 0xfc004520)  
Size: 17608 bytes  
Name: 'sfile101'  
Info: ''  
Flags: gry_run_at_boot, executable, elf,  
Addr: 0xfc27059c (hdr @ 0xfc270550, nnextptr = 0xfc274a70)  
Size: 17608 bytes
```

Total: 4 accessible files (36619 bytes).

TFTP

NAME:

tftp

SYNOPSIS:

Trivial File Transfer Protocol client/server.

USAGE:

tftp -[aF;f:i:vV] {on|off|IP} {get fname [addr] }

TFTP CLIENT DESCRIPTION:

The tftp command is primarily for use by the client side of the TFTP implementation of MicroMonitor. One exception to this is that the arguments *on* and *off* are used to enable and disable (or shutdown) the tftp server (refer to the server discussion below for more details). This client allows the user to transfer a file from some remote TFTP server into either RAM/DRAM space or into a file in TFS. Specification of the -F option is what tells the client that the file (after being transferred to RAM) is to be transferred into TFS. By default, the file is transferred to the address specified by the \$APPRAMBASE shell variable, but addition of the addr field overrides that default. Upon successful completion of the tftp get command, the shell variable \$TFTPGET will be loaded with the amount of data transferred.

Packet Retransmission and Timeout Process...

Both the client and server will, under certain circumstances, have to re-transmit a packet that may have been lost. The retry mechanism is based on RFC2131 (DHCP, section 4.1). The initial retry value (retransmit_delay) is doubled until some maximum value (retransmit_delay_max) is reached. At that point the delay is no longer doubled, but some final number of retries (giveup_count) are performed at the rate of that last delay. These three values (retransmit_delay, retransmit_delay_max and giveup_count) have defaults but can be tuned if the shell variable \$TFTP_RETRYTUNE is set. Note that this timeout/retry mechanism is used by DHCP, TFTP and ARP.

OPTIONS:

- -a
transfer from host in netascii mode (default is "octet");
- -F {file}
name of TFS file to copy to
- -f {flags}
flags assigned to TFS file after copy;
- -i {info}
file info assigned to TFS file after copy
- -v
low verbosity, show TFTP opcodes
- -V
high verbosity, show TFTP opcodes, plus complete IP/UDP packet header

EXAMPLES:

- tftp -F appfile -f eC -i 11/18/1999 135.3.130.1 get dir1/hostfile
Retrieve a file from a TFTP server running on a system whose IP address is 135.3.130.1. The file on the host (relative to the top-level directory of the TFTP server) is called *dir1/hostfile* and it is transferred to the file *appfile* on the target with flags eC (executable COFF) and an info field of 11/18/1999.
- tftp 135.3.130.1 get dir2/another_file
Retrieve the file *dir2/another_file* from a host TFTP server at 135.3.130.1 and place it in the location stored in the shell variable \$APPRAMBASE.

- `tftp 135.3.130.1 get dir3/yanotherfile 0x10000000`
Retrieve the file `dir3/yanotherfile` from a host TFTP server at 135.3.130.1 and store it at location 0x10000000.

NOTES:

As a client, `tftp put` is not supported (use the server, and a host based client).

TFTP SERVER DESCRIPTION:

First of all...

For the sake of this discussion, realize that we are referring to a situation where the server is the MicroMonitor-based target and the client is some PC or workstation; hence, all of the "tftp" commands mentioned in this section are done on a remote PC or workstation. For transfers from target to host (*tftp get*), the source file is on the target and the destination file is on the host; likewise, for transfers from host to target (*tftp put*), the source file is on the host and the destination file is on the target.

Also...

When the monitor first starts up, if the ethernet interface is active, it automatically starts up a TFTP server and the monitor will respond to incoming TFTP requests by default. The TFTP server in the monitor looks much like any other TFTP server. Differences are due to the fact that the server allows the client to transfer files several different ways.

From host file to target file:

The client specifies a destination that consists of the destination file name, the flags that are to be associated with the file and the information field that is part of the TFS header for keeping a description of the file. The syntax of this destination is comma delimited:

`filename,flags,information_field`

All text up to the first comma is considered the filename; text between the first comma and second comma is used as the file flags or attributes, and all text after the second comma is taken as the information field for the TFS file.

Notes:

- the flags field can be empty (2 commas back-to-back) and the information_field need not be specified; or both the flags and information_field can be omitted. Here are a few variations of the above syntax:
`filename,,information_field`
`filename,flags`
`filename`
- when this transfer is actually in progress, the file is first transferred to the location stored in the \$APPRAMBASE shell variable; after data transfer completes, it is then copied to TFS flash space.
- if the file already exists in TFS, then it will be deleted and if necessary, a defragmentation will be done.
- if the file is ascii, then the TFTP server will properly handle "netascii" mode for transferring to the target; the default mode is "octet"

From host file to target RAM:

If the destination file name starts with "0x", then a file on the host can be transferred to the hex address that follows. This memory space is assumed to be standard writeable RAM or DRAM.

From host file to destination assumed to be in a shell variable:

If the destination file name starts with a dollar sign '\$' and the string following the '\$' is a valid shell variable in the monitor context, then the destination will be taken as whatever is the content of the shell variable. If the shell variable does not exist, then the destination will be that string (a filename that starts with a '\$').

From target file to host file:

This is a standard transfer. The file must exist in TFS, and it will be transferred to the host; otherwise, a TFTP error message is sent to the client.

From target memory to host file:

This satisfies the case where a block of memory not necessarily part of TFS needs to be transferred up to a host. The source file specification is of the following syntax:

hex_address,length

This will cause the server to transfer 'length' bytes of data starting at 'hex_address' to the host. The server detects this by seeing "0x" as the first two characters in the source filename, and a comma somewhere later in the string.

Command line examples when using a host-based client to talk to the MicroMonitor TFTP server:

- tftp 135.3.94.136 put srcfile tfsfile,eC,Mar_30,1999@11:18
Send the file "srcfile" to a target at IP address 135.3.94.136. The destination filename on TFS will be "tfsfile" with the flags eC indicating executable COFF, and the information field will contain the string Mar_30,1999@11:18.
- tftp 135.3.94.136 put srcfile 0x10400000
Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is an address that must be in RAM space of the target.
- tftp 135.3.94.136 put srcfile \\${APPRAMBASE}
Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is the content of the shell variable \$APPRAMBASE. Note that the '\$' is preceded by a backslash. This is because we want the host shell to ignore the '\$' and pass it to the target as is. Obviously this, then, depends on the shell running on the host at the time.
- tftp 135.3.94.136 put srcfile
Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is the same filename in TFS flash space.
- tftp 135.3.94.136 get srcfile
Retrieve the file "srcfile" from the target at IP address 135.3.94.135. Once on the host it will have the same name.
- tftp 135.3.94.136 get 0x10800400,900 bdata
Retrieve 900 bytes starting at location 0x10800400 and place them in the file bdata.

ULVL

NAME:

ulvl

SYNOPSIS:

User level.

USAGE:

ulvl `[-[phc:] [usrlvl | min | max] [password]`

DESCRIPTION:

This command is used to set or configure the monitor's user level. The user level of the monitor determines what commands it can execute and what files are accessible. This command is hard coded to require only user-level 0 to execute. All other commands are configurable through the config.h file. For a thorough description of the user level functionality in the monitor refer to the user-level description.

OPTIONS:

- `-c {cmd,lvl}`
modify the user level of command "cmd" to level "lvl". The value of lvl must be between 0 and 3 and if the level is being lowered, then the current user level that the monitor is running at must be at least as high as the command that is being adjusted.
- `-p`
go into an interactive mode to build the password storage file. This file will contain 3 lines, each of which will be the password for levels 1 through 3. This file is stored with flags "u3" for highest security.
- `-h`
since the backdoor entry into the monitor requires that the user know the MAC address of the system, this option simply dumps the header that contains that information.

EXAMPLES:

- `ulvl -c version,2`
change the user level of the "version" command to 2.
- `ulvl -c version,2 -c help,1 -c dm,4`
change user level of version to 2, help to 1, and dm to 4.

NOTES:

- The user level of the "ulvl" command cannot be adjusted. It must be able to run at user level zero.

VERSION

NAME:

version

SYNOPSIS:

Display the build date of the monitor (and application) executables.

USAGE:

version [application build info]

DESCRIPTION:

With no arguments, version simply displays the date/time at which the monitor was built. If previously, version was executed with application build information, then that string will be printed also.

XMODEM

NAME:

xmodem

SYNOPSIS:

Initiate an XMODEM (or YMODEM) data transfer.

USAGE:

xmodem -[a:BcdF:f:i:ks:t:uvy]

DESCRIPTION:

XMODEM is a very simple protocol that allows files to be transferred between 2 machines that understand the XMODEM protocol. This monitor supports XMODEM because it is small, and universal for probably all terminal emulation packages on the PC. Options allow the user to upload and download, and if enabled, transfer the downloaded file to the file system (see TFS). When downloading to the target, the data is initially placed in system RAM. Upon completion of the download, that downloaded data may be transferred to a file in TFS. By default, the RAM address used is the value established by the monitor at boot time known as APPRAMBASE (stored in the APPRAMBASE shell variable and also displayed on the console at reset).

OPTIONS:

- -a {addr}
address used to override the default APPRAMBASE download destination address.
- -B
new boot monitor load (see notes below);
- -c
use CRC instead of checksum;
- -d
download a block of data. This, along with other options, allows the downloaded file to be placed in RAM or the file system. Note that even if the downloaded file is destined for the file system, it is initially downloaded to RAM and then copied to the file system, so the address specified must be writeable RAM space;
- -f {flags}
for downloading to a file in TFS, this specifies the flags that will be assigned to the file after the download has completed
- -F {filename}
for upload or download, specifies the name of the file to transfer.
- -i {info}
for downloading to a file in TFS, this specifies the information field that will be assigned to the file after the download has completed;
- -k
use 1K block size (instead of default 128-byte);
- -s {size}
since XMODEM transfers in fixed block sizes of 128 bytes, the computed download size of a file is likely to be incorrect. This option allows the user to override the compute size with the value specified by "size".
- -t {address}
enable tracing to a buffer specified by "address". This is primarily used for debugging xmodem itself.
- -u
upload a file or block of data. If the upload is a file then the -F option must also be specified. If upload of raw data, then address and size must be specified on the command line;

- -v
verify only
- -y
support the YMODEM extensions to XMODEM.

EXAMPLES:

- xmodem -u -F filename
Upload the file "filename" from TFS to the host machine.
- xmodem -u -a 0x200400 -s 500
Upload 500 bytes of data starting at location 0x200400 to the host.
- xmodem -d -a 0x6000
Download a file from the host to location 0x6000 in memory.
- xmodem -d
Download a file from the host to the start of application RAM space.
- xmodem -d -F AppFile -f eCB -i 02_24_97 -a 0x6000 -s 23456
Download a file from host to TFS file 'AppFile'. When the file is created in TFS assign flags 'eCB' (see TFS man page) and information field '02_24_97'. The file will be downloaded into ram starting at location 0x6000 and the final file size used to place the file in TFS will be 23456 bytes.

NOTES:

- The command requires that either -u or -d be specified (-B implies -d).
- The -v (verify) option allows the user to download a block of data to some address, then invoke the same download with the -v option to verify that the data was transferred correctly.
- The basic XMODEM protocol forces all transfers to be some multiple of 128 bytes. This means the files may have junk at the end of them. For a download to TFS, this can be overridden by supplying the final size with the -s option. Then after the download is completed, instead of using the download size computed by XMODEM, it uses the value specified on the command line.
- The -B option makes it more convenient to rebuild the monitor onboard. **Be careful with this because it will take the binary file transferred and use it to rebuild the boot flash; hence, if the binary file is incorrect, the boot will be corrupted.** When rebuilding the monitor, the sequence of events (without using -B) would be:

1. uMON> xmodem -d {ram_address}
2. Hyperterm: xmodem {monitor-binary}
3. uMON> flash opw
4. uMON> flash ewrite {boot_address} {ram_address} {size_of_monitor-binary}

This is somewhat error prone, so the -B option automates these steps because it knows the ram_address, boot_address and the size of the space allocated to the monitor in flash. The above steps are replaced with...

1. uMON> xmodem -B
2. Hyperterm: xmodem {monitor-binary}

After the download completes, xmodem -B will then query the user for approval, at which time a carriage return or y approves, and all other characters will abort. Once you give approval, allow time for the bootflash to be reprogrammed. This can take several seconds, depending on the speed of the flash device interface and the size of the monitor binary being programmed.

