

# Getting Started with uMonM and the Kinetis K6X MCU

Last update: 2/20/2015 12:51 a2/p2

## *Goal:*

The purpose of this document is to introduce MicroMonitor-M (MicroMonitor for MCUs) on Kinetis MCUs. The discussion is centered on uMonM on the [K64F \(Freedom Platform\)](#) and uumon on the K60 ([TWR-K60N512](#) with [TWR-SER](#)). While this document is specific to Kinetis targets; it should provide a basis for any Cortex-M3/4 platform that has enough memory and IO to justify the use of a flash-resident bootloader. As of this writing, uMonM has been ported to K64F, K60, STM32F, LM3S6918 and LPC1769. Actual sizes of images will vary from one port to the next due to library sizes and features configured into the build. The footprint sizes discussed in this document are taken from the TWR-K60N512 build.

Let's start by getting the naming straight... There are three different versions of MicroMonitor, all of which are available on the MicroMonitor homepage (<http://www.umonfw.com>)...

- **uMon** is a full-blown bootloader for use in systems that have a memory footprint that justifies the bells and whistles. It was/is typically used in systems that can afford to allocate 128K-256K of flash to the bootloader. If that's what you're looking for, then go to the [homepage](#), this document is not where you want to be.
- **uMonM** is a version of MicroMonitor slimmed down for use on Cortex-M3/4 MCUs with limited memory footprint. If you're familiar with uMon, then uMonM will look similar, just trimmed back wherever possible without eliminating the core feature set. It has TFS (tiny file system), supports the API hookup with the application, and a few demonstration applications (Websocket-ready HTTP server & Lua), to demonstrate that it is truly a "younger brother" to uMon. Its flash footprint is typically around 64K (very tunable).
- **uumon** is still a smaller version with only the bare essentials needed for a networked bootloader. While the base source tree is the same, the removal of TFS from the platform reduces the available feature set significantly; however, this may be just right for most MCU-based platforms that need a networked bootloader. Its flash footprint is typically around 32K.

## *Scope of this Document:*

This document is limited to uMonM and uumon. The source tree for this was created from the uMon tree, but features were snipped to cut the footprint down significantly. I opted to separate the two trees (uMon and uMonM) entirely to avoid sprinkling `#ifdef`'s all over the code; however the core design is the same, so if a feature from uMon is needed in uMonM it's not difficult to cut-n-paste it back in for custom ports. The uMonM tree covers both uMonM and uumon. Each port (under `umon_ports` directory of the source tree) supports the ability to build for both uMonM and uumon, as discussed later.

## *In the Nutshell:*

uMonM is a rework of uMon to deal with the slightly different architecture that comes with the smaller Cortex-M3/M4 MCUs available today...

- Far less memory
- Typically more flash than ram
- Different programming model for NOR flash

uMonM has essentially the same feature set as uMon so you can refer to the uMon homepage for details. Just realize that each feature has been trimmed back to reduce the footprint. A good introduction to MicroMonitor can be found here: [http://www.umonfw.com/docs/white\\_paper.pdf](http://www.umonfw.com/docs/white_paper.pdf); just be aware that uMonM is a subset of the functionality in uMon.

Uumon is smaller yet in both footprint and functionality; however it still supports the key features needed in a networked bootloader:

- **Flash:** write & erase-per-sector.

- **Network:** ping server & TFTP client.
- **Console:** simple command line interface.
- **Configurable:** flash based, non-volatile environment (shell variables) for storage of network parameters and autoboot mode.

This uses less than 32K of flash, with still room for further reduction. Extensions to this core set include...

- The ability to have the bootloader update itself in a running system.
- DHCP-based network configuration and bootfile download.
- TFTP server
- Memory peek & poke commands.
- UDP-based access to the console command line.

With all of these extensions turned on, the footprint remains under 40K. Obviously you can mix-n-match the setup that fits your needs using the configuration files that come with each port.

If you're familiar with uMon, the [uMon user's manual](#) now has a chapter that discusses the design issues associated with a transition from uMon to uMonM.

*Finally...*

This is a work in progress, so check with the [author](#) (see end of this document) to make sure you're using the latest firmware.

**Table of Contents:**

Getting Started with uMonM and the Kinetis K6X MCU..... 1

[FRDM-K64F Setup..... 4](#)

[Hardware..... 4](#)

[Software..... 4](#)

[Firmware..... 4](#)

[Initial uMonM Configuration..... 5](#)

[Console Connection..... 6](#)

[K64F-FRDM Running uMonM..... 8](#)

[Flash & TFS..... 8](#)

[flash:..... 9](#)

[tfs:..... 10](#)

[edit:..... 11](#)

[Ethernet Setup..... 12](#)

[Running with uMonM's Baby Brother Uumon..... 13](#)

[Building uumon..... 14](#)

[Booting uumon..... 14](#)

[Non-volatile Environment..... 15](#)

[Static Network Setup..... 15](#)

[Dynamic Network Setup \(DHCP\)..... 15](#)

[Running Without a Console..... 16](#)

[Commands Unique to uumon..... 16](#)

[call: Jump to address in memory..... 16](#)

[set: Set or display shell variable\(s\)..... 16](#)

[Installing an Application that Uses uMonM's API..... 17](#)

[Initial demo application:..... 17](#)

[Establish MONCOMPTR:..... 17](#)

[Build and Run the Demo..... 17](#)

[Http server application:..... 18](#)

["Lua" application:..... 20](#)

[Wrap Up..... 20](#)

# FRDM-K64F Setup

## Hardware

For the sake of this introduction, the FRDM-K64F runs out of the box (almost). For connectivity, you need a USB and Ethernet cable. The board is powered off the USB port, and also has a virtual comport connection through that link...

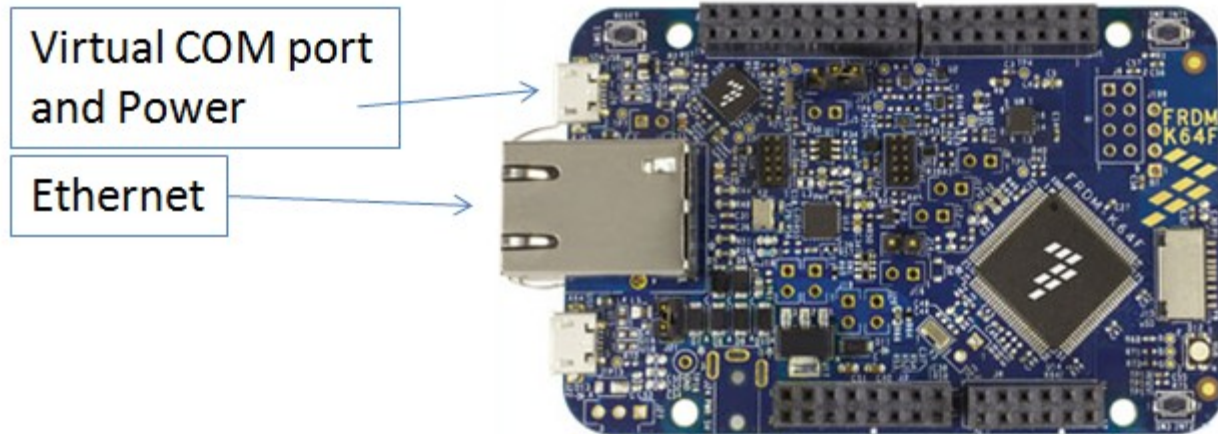


Figure 1: K64F-FRDM connected

The OpenSDA bootloader that comes with the board must be updated. Refer to instructions here: <http://www.segger.com/opensda.html> on how to do that.

## Software

There are several good options here; however, this walkthrough assumes only the basics:

- lite version [cross-compiler](#) from CodeSourcery
- the [J-Link tool](#) from Segger
- the [Windows serial port driver](#) from mbed.org
- Cygwin (<http://www.cygwin.com/>) environment (with bash shell) is on the PC.

There are also a few tools that normally come with the standard uMon distribution that you can grab here: <http://www.umonfw.com/releases/umontools.tgz>. Download this file and extract using "tar -xzvf umontools.tgz". Put these win-executables (tftp.exe, moncmd.exe, title.exe & elf.exe) in your PATH<sup>1</sup>.

The remainder of this document assumes that you've installed the above software.

## Firmware

Download the uMonM tar ball from here: <http://www.umonfw.com/releases/umonm.tgz>; and place the umonm.tgz file under a new directory (i.e. C:/umonm) on your PC. Running with a bash shell (part of Cygwin), untar the file at this point with the command: "tar -xzvf umonm.tgz". This will install the source tree for uMonM. This includes the uMonM bootloader and a few demonstration applications (LWIP based http server using web sockets and a basic port of the [Lua programming language](#) using files in TFS as Lua programs. These

<sup>1</sup> If you prefer to build these tools from source, then you need to download the full uMon tar ball from <http://www.umonfw.com> and follow those instructions.

applications demonstrate the hookup between a network installable application and the underlying uMonM platform. At the top level there are three directories...

- **umon\_main:** this is the bulk of the monitor code most of which is core uMon but also some contributed functionality (i.e. zlib)
- **umon\_ports:** each directory below this contains code specific to one port. As of this writing, there are ports for FRDM (K64F), TWRK60N512, Simplecortex (LPC1769) and Discovery (STM32F4).
- **umon\_apps:** each directory below this is a demo application that can be installed on a system running uMonM. The apps that come with standard uMon should be easily adjusted to run on this platform.

## Initial uMonM Configuration

Under the `umon_ports/frdmk64f` directory is a file `config.h`. This file pulls in one of two “sub-config” files depending on a setting in the makefile. The “`umonm_config.h`” file builds a more complete (i.e. more flash) version of the bootloader (includes TFS and other bells & whistles); while “`uumon_config.h`” will build a basic networked bootloader<sup>2</sup>. These files are responsible for establishing what features are to be enabled within the uMonM build. Below is just an example of the configuration; refer to mentioned files for current default settings.

```
...
#define INCLUDE_MALLOC          1
#define INCLUDE_MEMCMDS        0
#define INCLUDE_DM              1
#define INCLUDE_PM              1
#define INCLUDE_SHELLVARS      1
#define INCLUDE_XMODEM         1
#define INCLUDE_EDIT           1
#define INCLUDE_UNZIP           0
#define INCLUDE_ETHERNET       1
#define INCLUDE_ICMP            INCLUDE_ETHERNET
#define INCLUDE_TFTP            INCLUDE_ETHERNET
#define INCLUDE_FLASH           1
#define INCLUDE_TFS             1
#define INCLUDE_TFSAPI          1
#define INCLUDE_TFSSCRIPT       1
#define INCLUDE_TFSCLI          INCLUDE_TFS
#define INCLUDE_LINEEDIT        1
#define INCLUDE_DHCPBOOT        INCLUDE_ETHERNET
#define INCLUDE_EE              1
#define INCLUDE_STOREMAC        0
#define INCLUDE_VERBOSEHELP     1
#define INCLUDE_HWTMR           1
#define INCLUDE_BLINKLED        1
...
#define INCLUDE_TFTPSVR         INCLUDE_ETHERNET
#define INCLUDE_ETHERVERBOSE    1
#define INCLUDE_MONCMD          INCLUDE_ETHERNET
#define INCLUDE_READ            1
...
```

### Listing 1: INCLUDE\_XXX macros in config.h

This is a relatively “loaded” configuration with most features enabled for the sake of demonstration; however in a real setup it is likely that several of these features will be disabled to reduce the size of the memory footprint. Later, when you get used to the environment, you can mix and match these features as you see fit.

---

<sup>2</sup> The `uumon` version is discussed in a later section. For now we assume we’re going with `umonm` (the bigger footprint version).

To build the monitor, change directory to `umon_ports/frdmk64f`. There is a `bashrc` file there that sets up some environment variables, so run `./bashrc`. Paths may need to be adjusted in the make file, so when you get them right you should be able to type "make rebuild" to build the monitor image destined for the board.

## Console Connection

*Note: At this point, it is assumed that you've installed the tools mentioned in section "Software" above. With that in place, connecting the USB cable to the virtual com port will provide both a com port and access to the OpenSDA debug interface (which we will use shortly to reprogram the K64FRDM's flash).*

With the USB cable plugged in, your system should have recognized the board as a USB-Serial cable (virtual com port); thus allowing you to run some console program. This documentation uses uCon ([http://www.umonfw.com/releases/ucon\\_install.exe](http://www.umonfw.com/releases/ucon_install.exe)) because it has additional tools built in that hook to uMon (TFTP client/server, DHCP/BOOTP server, UDP command interface, Xmodem, etc...). If you use something else, you'll have to adapt the instructions appropriately. The configuration is 115200 8-N-1. The actual COM port number you connect to will depend on your system...

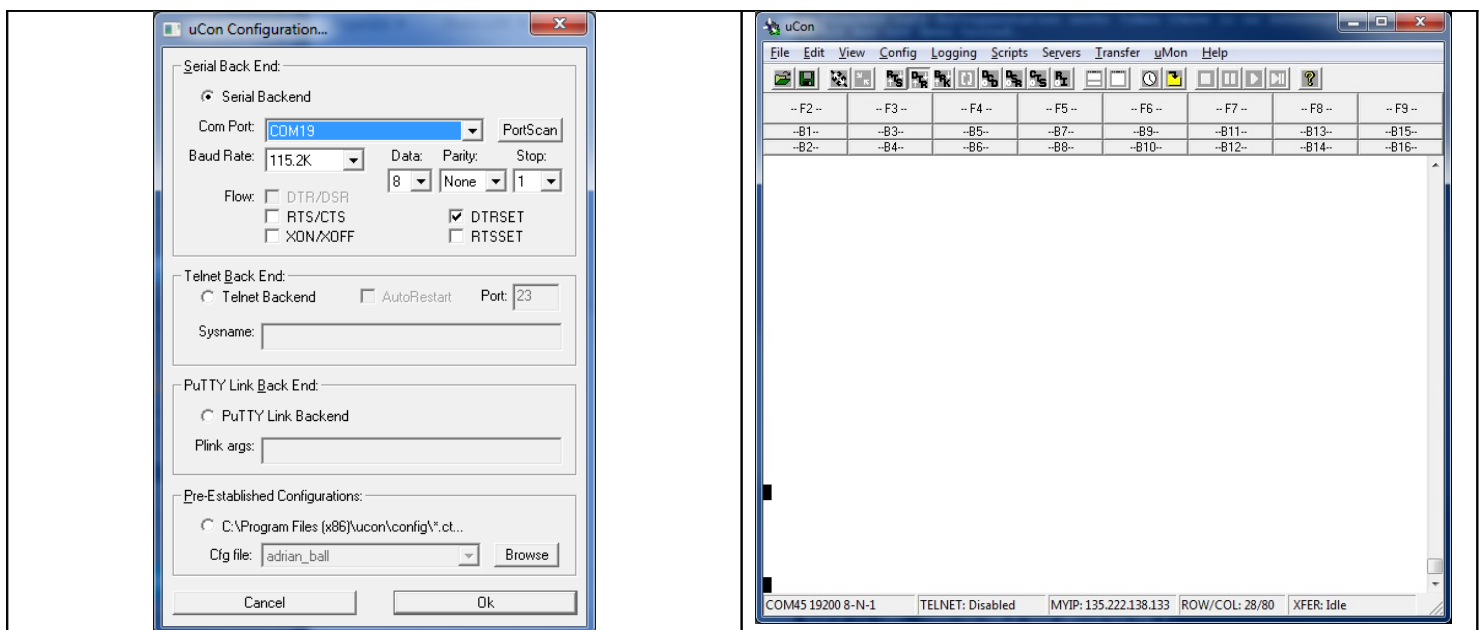


Figure 2: uCon Startup and Console

With the console application now running and connected to the correct COM port, you can now type "make flash" in the bash window under `umon_ports/frdmk64f` directory to use the J-link tool to install the image into the K64F's flash<sup>3</sup>. You may initially be asked to accept an agreement with JLINK, so just do that. You'll then see a small dialog box showing the progress of the programming and the output in your bash window (where you run make) will be something like this:

```
"C:/Program Files (x86)/SEGGER/JLinkARM_V484f/JLink.exe" jlink_flash.scr
SEGGER J-Link Commander V4.84f ('?' for help)
Compiled May  9 2014 20:05:50

Script file read successfully.
DLL version V4.84f, compiled May  9 2014 20:05:42
Firmware: J-Link OpenSDA 2 compiled Apr 24 2014 14:44:11
Hardware: V1.00
S/N: 621000000
VTarget = 3.300V
Info: Found SWD-DP with ID 0x2BA01477
```

<sup>3</sup> I've noticed that the JLINK tool doesn't exit 0 on success, so this erroneously tells 'make' that an error occurred, so if the output of your 'make flash' step

```
Info: Found Cortex-M4 r0p1, Little endian.
Info: FPUUnit: 6 code (BP) slots and 2 literal slots
Info: TPIU fitted.
Info: ETM fitted.
Info: ETB present.
Info: CSTF present.
Found 1 JTAG device, Total IRLen = 4:
Cortex-M4 identified.
Target interface speed: 100 kHz
Processing script file...
```

```
Info: Device "MK64FN1M0XXX12" selected (1024 KB flash, 192 KB RAM).
Reconnecting to target...
Info: Found SWD-DP with ID 0x2BA01477
Info: Found SWD-DP with ID 0x2BA01477
Info: Found Cortex-M4 r0p1, Little endian.
Info: FPUUnit: 6 code (BP) slots and 2 literal slots
Info: TPIU fitted.
Info: ETM fitted.
Info: ETB present.
Info: CSTF present.
```

```
Target interface speed: 1429 kHz
```

```
Reset delay: 0 ms
Reset type NORMAL: Resets core & peripherals via SYSRESETREQ & VECTRESET bit.
```

```
PC = 00000414, CycleCnt = 00000000
R0 = 00000000, R1 = 00000000, R2 = 00000000, R3 = 00000000
R4 = 00000000, R5 = 00000000, R6 = 00000000, R7 = 00000000
R8 = 00000000, R9 = 00000000, R10= 00000000, R11= 00000000
R12= 00000000
SP(R13)= 20009BC0, MSP= 20009BC0, PSP= 00000000, R14(LR) = FFFFFFFF
XPSR = 01000000: APSR = nzcvcq, EPSR = 01000000, IPSR = 000 (NoException)
CFBP = 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00
```

```
FPS0 = 00000000, FPS1 = 00000000, FPS2 = 00000000, FPS3 = 00000000
FPS4 = 00000000, FPS5 = 00000000, FPS6 = 00000000, FPS7 = 00000000
FPS8 = 00000000, FPS9 = 00000000, FPS10= 00000000, FPS11= 00000000
FPS12= 00000000, FPS13= 00000000, FPS14= 00000000, FPS15= 00000000
FPS16= 00000000, FPS17= 00000000, FPS18= 00000000, FPS19= 00000000
FPS20= 00000000, FPS21= 00000000, FPS22= 00000000, FPS23= 00000000
FPS24= 00000000, FPS25= 00000000, FPS26= 00000000, FPS27= 00000000
FPS28= 00000000, FPS29= 00000000, FPS30= 00000000, FPS31= 00000000
FPSCR= 00000000
```

```
Downloading file... [build_K64F_FRDM/boot.bin]
Info: J-Link: Flash download: Flash download into internal flash skipped. Flash contents
already match
```

```
Script processing completed.
```

```
make: [flash] Error 1 (ignored)
```

Notice that 'make' may indicate an error. As far as I can tell, this can be ignored. Apparently the JLINK tool does not exit with a zero on success; hence make assumes this is an error. Anyway, this steps through an Erase/Program/Verify of the affected flash, and when that completes (about 10 seconds), uMonM will boot and look something like this at the console:

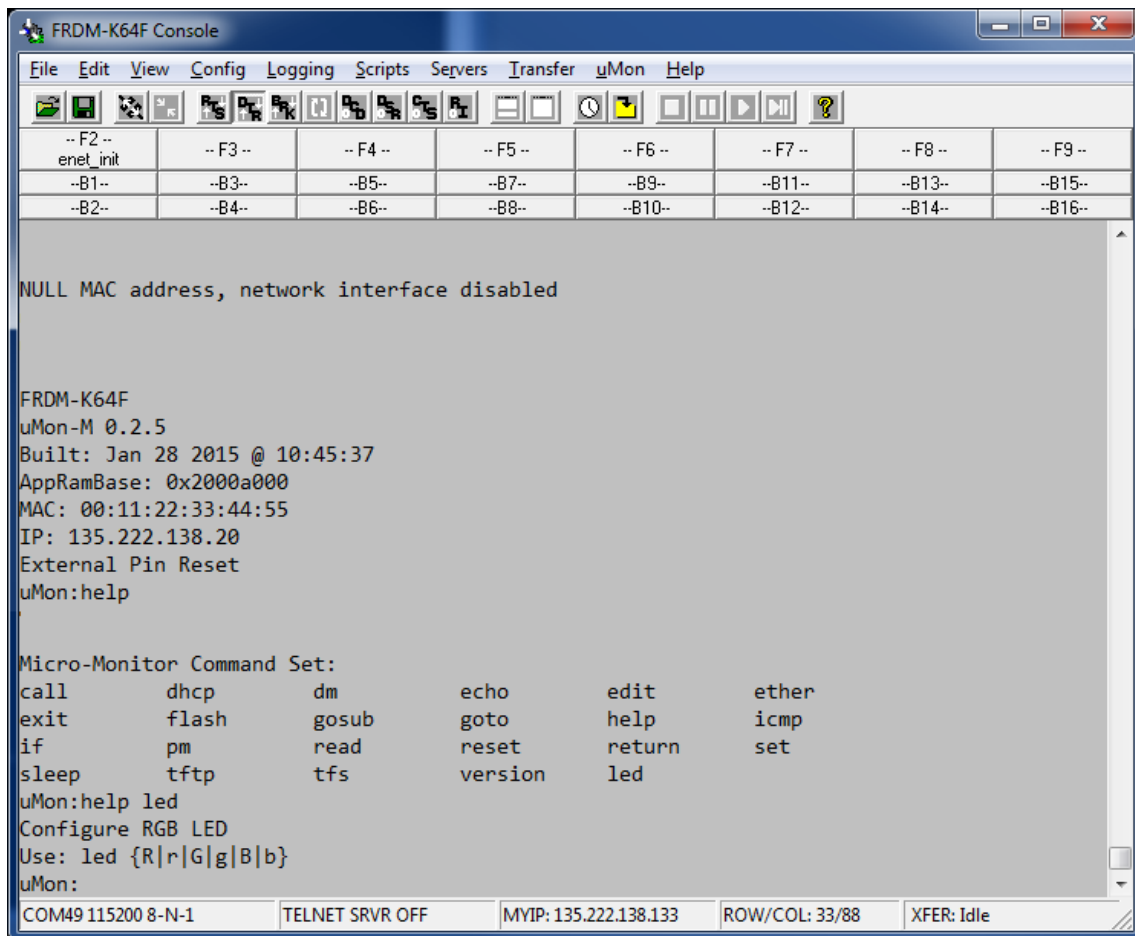


Figure 3: Startup message and some 'help' output from uMonM

If you don't see the initial boot-up header automatically, try pushing the 'reset' button on the edge of the board near the VCOM usb connector.

## K64F-FRDM Running uMonM

If everything is golden, then at this point, uMonM is installed on the board and you've seen output similar to that of Figure 3<sup>4</sup>. At the **uMon:** prompt type 'help' to see (assuming you're working with the default configuration) a few dozen commands.

Note the message: "NULL MAC address, network interface disabled". This is expected, because at this point the Ethernet interface has not been set up. There will be more discussion on that shortly.

Most of the commands are either a duplicate or a subset of the standard uMon command so it is beyond the scope of this documentation to go into all that detail. Refer to the uMon user manual ([http://www.umonfw.com/docs/umon\\_user\\_manual.pdf](http://www.umonfw.com/docs/umon_user_manual.pdf)) for the full set of manpages. Several of the chapters of that manual still apply; particularly chapters 3-7; just be aware that some features in uMon are not in uMonM<sup>5</sup>.

## Flash & TFS

The core feature of uMon is TFS (tiny file system) a basic but useful wrapper around NOR based flash that allows the user at the command line as well as the application (through the API) to access flash as named files rather than raw memory. The three commands that show this off best are: 'flash', 'tfs' and 'edit'.

<sup>4</sup> Note that the figure also shows output of a few commands issued after the reset.

<sup>5</sup> Generally speaking, anything in uMon could be pulled into uMonM. The trees were separated to avoid an abundance of #ifdefs throughout the uMon code.



## flash:

For uMonM the two main features of this command are the ability to display the status of the entire flash block (flash info) and to erase sectors as needed (flash erase {range}). The output of "flash info" is as follows:

```
uMon:flash info
Current flash bank: 0
Device = K64F-Flash
Sctr TFS? Begin End Size SWProt? Erased?
  0 0x00000000 0x00000fff 0x001000 yes no
  1 0x00001000 0x00001fff 0x001000 yes no
  2 0x00002000 0x00002fff 0x001000 yes no
  3 0x00003000 0x00003fff 0x001000 yes no
  4 0x00004000 0x00004fff 0x001000 yes no
  5 0x00005000 0x00005fff 0x001000 yes no
  6 0x00006000 0x00006fff 0x001000 yes no
  7 0x00007000 0x00007fff 0x001000 yes no
  8 0x00008000 0x00008fff 0x001000 yes no
  9 0x00009000 0x00009fff 0x001000 yes no
 10 0x0000a000 0x0000afff 0x001000 yes no
 11 0x0000b000 0x0000bfff 0x001000 yes no
 12 0x0000c000 0x0000cfff 0x001000 yes no
 13 0x0000d000 0x0000dfff 0x001000 yes no
 14 0x0000e000 0x0000efff 0x001000 yes no
 15 0x0000f000 0x0000ffff 0x001000 yes no
 16 0x00010000 0x00010fff 0x001000 yes no
 17 0x00011000 0x00011fff 0x001000 yes no
 18 0x00012000 0x00012fff 0x001000 yes no
 19 0x00013000 0x00013fff 0x001000 yes no
 20 0x00014000 0x00014fff 0x001000 yes no
 21 0x00015000 0x00015fff 0x001000 yes no
 22 0x00016000 0x00016fff 0x001000 yes no
 23 0x00017000 0x00017fff 0x001000 yes no
 24 0x00018000 0x00018fff 0x001000 yes no
 25 0x00019000 0x00019fff 0x001000 yes no
 26 0x0001a000 0x0001afff 0x001000 yes no
 27 0x0001b000 0x0001bfff 0x001000 yes no
 28 0x0001c000 0x0001cfff 0x001000 yes no
 29 0x0001d000 0x0001dfff 0x001000 yes no
 30 0x0001e000 0x0001efff 0x001000 yes no
 31 0x0001f000 0x0001ffff 0x001000 yes no
 32 0x00020000 0x00020fff 0x001000 yes no
 33 0x00021000 0x00021fff 0x001000 yes no
 34 0x00022000 0x00022fff 0x001000 yes no
 35 0x00023000 0x00023fff 0x001000 yes no
 36 0x00024000 0x00024fff 0x001000 yes no
 37 0x00025000 0x00025fff 0x001000 yes no
 38 0x00026000 0x00026fff 0x001000 yes no
 39 0x00027000 0x00027fff 0x001000 yes no
 40 0x00028000 0x00028fff 0x001000 yes no
 41 0x00029000 0x00029fff 0x001000 yes yes
 42 0x0002a000 0x0002afff 0x001000 yes yes
 43 0x0002b000 0x0002bfff 0x001000 yes yes
 44 0x0002c000 0x0002cfff 0x001000 yes yes
...
223 0x000df000 0x000dffff 0x001000 no yes
224 0x000e0000 0x000e0fff 0x001000 no yes
225 0x000e1000 0x000e1fff 0x001000 no yes
226 0x000e2000 0x000e2fff 0x001000 no yes
227 0x000e3000 0x000e3fff 0x001000 no yes
228 0x000e4000 0x000e4fff 0x001000 no yes
```

229		0x000e5000	0x000e5fff	0x001000	no	yes
230		0x000e6000	0x000e6fff	0x001000	no	yes
231		0x000e7000	0x000e7fff	0x001000	no	yes
232		0x000e8000	0x000e8fff	0x001000	no	yes
233		0x000e9000	0x000e9fff	0x001000	no	yes
234		0x000ea000	0x000eafff	0x001000	no	yes
235		0x000eb000	0x000ebfff	0x001000	no	yes
236		0x000ec000	0x000ecfff	0x001000	no	yes
237		0x000ed000	0x000edfff	0x001000	no	yes
238		0x000ee000	0x000eefff	0x001000	no	yes
239		0x000ef000	0x000effff	0x001000	no	yes
240	*	0x000f0000	0x000f0fff	0x001000	no	yes
241	*	0x000f1000	0x000f1fff	0x001000	no	yes
242	*	0x000f2000	0x000f2fff	0x001000	no	yes
243	*	0x000f3000	0x000f3fff	0x001000	no	yes
244	*	0x000f4000	0x000f4fff	0x001000	no	yes
245	*	0x000f5000	0x000f5fff	0x001000	no	yes
246	*	0x000f6000	0x000f6fff	0x001000	no	yes
247	*	0x000f7000	0x000f7fff	0x001000	no	yes
248	*	0x000f8000	0x000f8fff	0x001000	no	yes
249	*	0x000f9000	0x000f9fff	0x001000	no	yes
250	*	0x000fa000	0x000fafff	0x001000	no	yes
251	*	0x000fb000	0x000fbfff	0x001000	no	yes
252	*	0x000fc000	0x000fcfff	0x001000	no	yes
253	*	0x000fd000	0x000fdfff	0x001000	no	yes
254	*	0x000fe000	0x000fefff	0x001000	no	yes
255	*	0x000ff000	0x000fffff	0x001000	no	yes

uMon:

## Listing 2: Summarized output for 'flash info' on K64FFRDM

This listing above (with repetitive sectors snipped out) shows a typical uMonM flash layout for the K64F. Each sector is 4K, and there are 256 of them. If your board is "clean" (i.e. empty, as it should be initially), then sectors 40 and above will be erased. Refer to the config.h file to see how the sectors are allocated. There are three main chunks: the uMonM image itself, space reserved for the application, and space reserved for TFS file storage (notice the asterisk in those rows). This allocation can be adjusted as needed with parameters in the config.h file. For now, to make sure all non-uMonM sectors are erased, at the command line type: "flash erase 48-255".

## tfs:

TFS is basically a wrapper around the NOR flash to allow the user to store named files rather than binary images at fixed locations. In addition to that basic capability, TFS supports scripting and auto booting so that a system can be configured based on the content of one or two ASCII files. The "startup" script (similar to autoexec.bat or .bashrc or .profile on other systems) is called "monrc" (monitor run-control). If present, then it is automatically run as a script; so it can be used to initially configure the personality of the board (typically this is the IP network information). uMon uses environment variables to do this, so for example IPADD is the shell variable that uMon looks to for the IP address of the board. GIPADD & NETMASK are the gateway IP and network mask respectively. So, a typical monrc file may contain the following lines:

```
set IPADD 192.168.1.200
set NETMASK 255.255.255.0
set GIPADD 192.168.1.1
set ETHERADD 00:11:22:33:44:55
```

## Listing 3: Example monrc file

With this being the content of the monrc file, the system would automatically boot up with that configuration. Some of the common uses of the "tfs" command itself would be to list all the files "tfs ls", remove a file "tfs rm {filename}" or defragment the flash "tfs clean" to eliminate "dead" space in the flash.

There are basically four ways to add files to TFS: Xmodem, TFTP, the 'edit' command and the "tfs add" command. How you do it will depend on what features you have installed in your configuration. For this text, we assume all features are installed so over the course of this text, we'll walk through a few examples. The simplest method (and the method that requires the least amount of overhead) is to simply build a string up in RAM and then add that as a file to TFS. For example, let's create a file that has two lines:

```
echo hello
echo there
```

To do this, we need to first put that text into ram....

```
uMon:pm -s $APPRAMBASE "echo hello\nnecho there\n"
```

Verify that we put it there:

```
uMon:dm $APPRAMBASE 22
```

Transfer it to a file in TFS, and verify that it is there:

```
uMon:tfs -fe add test $APPRAMBASE 22
uMon:tfs ls
Name                Size   Location  Flags  Info
test                22    0x000f0050 e
Total: 1 item listed (22 bytes).
uMon:
```

The output (above) shows that the file 'test' exists in TFS and has the 'e' flag, which means that it is an executable script. Now if you simply type "test" at the uMon: prompt you'll see the output..

```
uMon:test
hello
there
uMon:
```

You've just installed and executed a basic script on your board!

## **edit:**

The 'edit' command is a very basic (but quite handy) file editor built right into the monitor. It "feels" like the old "ed" command for those of you familiar with that. For the sake of this demo we'll create the above monrc file using edit. The following instructions are a bit condensed; for more details refer to the uMon user manual:

At the uMon: prompt type "edit -fe monrc" and hit -ENTER-. The monitor will respond with something like (the buffer address will vary depending on the build):

```
Edit buffer = 0x2000b000
Type '?' for help
```

There are several options here, but just type the letter 'i' (for 'insert') and hit -ENTER-. At this point you are in "insert" mode so that whatever you type will end up in the file. For this example, type each of the lines as shown in Listing 3 above. After typing the last line followed by -ENTER-, then as the first character on the next line type '.' (dot) and hit -ENTER- and again as the first character on the next line type 'q' and hit -ENTER- again. This will return you to the uMon: prompt and you should not be able to type "tfs ls" to see that the monrc file has been created. You can also type "tfs cat monrc" to see the content of the file. Now, if you reset your board, you'll see that it comes up with the configuration as specified in the monrc file, and you also see "hello world!"...

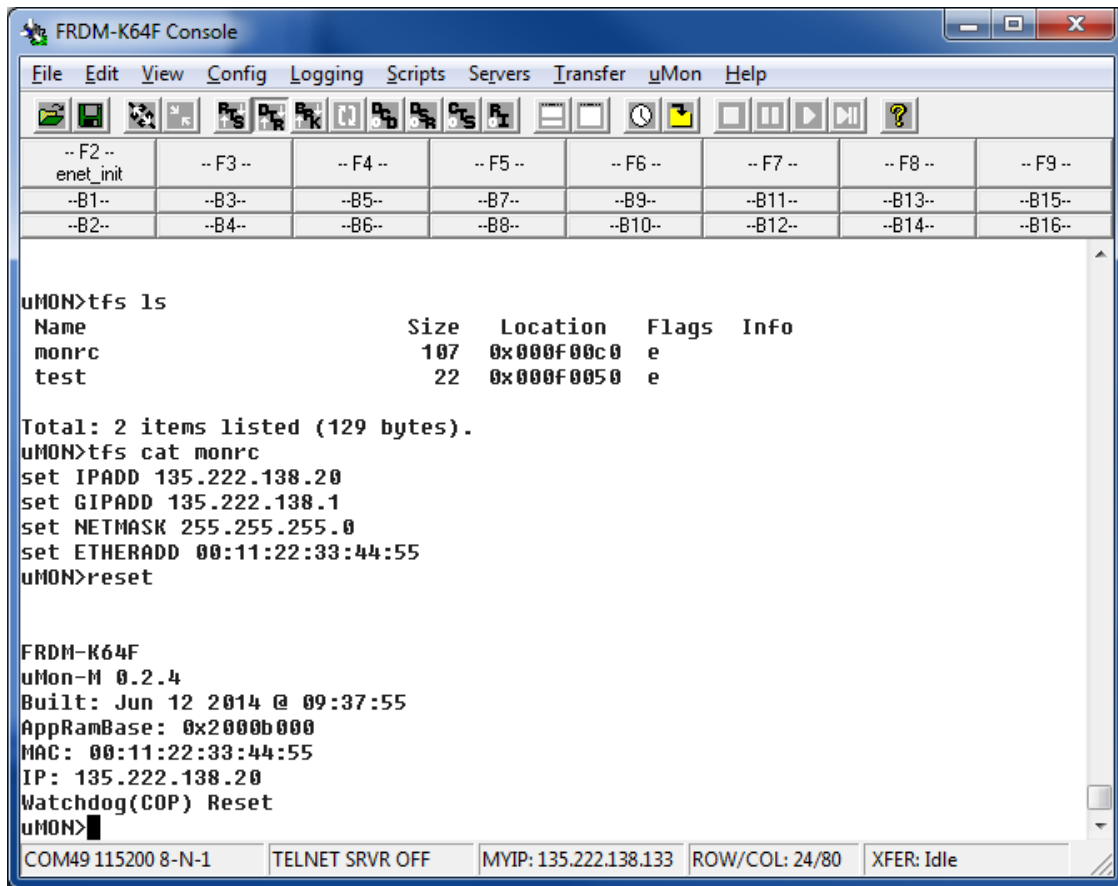


Figure 4: Console output: tfs ls & tfs cat monrc

At this point you have a system that is ready to connect to the network (assuming of course that you entered valid IP information in the monrc file). There is a lot more information on "auto-bootable" files, so refer to chapter 5 "Application Startup" of the uMon user manual.

## Ethernet Setup

Notice that the board now comes up with the MAC and IP addresses as specified in the monrc file. You're good to go at this point, but if the monrc file was not present, then the defaults from the make file would still be used. You can also override that by simply setting the IPADD shell variable and restarting the Ethernet driver. So at the uMon: prompt (within the console program) you could type<sup>6</sup>:

```
uMon:set ETHERADD 00:11:22:33:44:55
uMon:set IPADD 192.168.1.200
uMon:ether on
```

to restart the board's Ethernet interface using the specified IP address. Once this completes you should be able to ping that IP address from your PC and get a valid response.

As yet another alternative, the 'dhcp' command could be used to kick off a DHCP request; but that assumes you have a server configured; so we'll skip that for now (keep in mind that uCon can run a DHCP server, but that's too much detail for this text).

At this point you have a network-ready device that you can communicate with by ARP/ICMP (ping), TFTP and UDP (using uMon's moncmd<sup>7</sup> facility). Any TFTP client can be used to talk to the TFTP server that runs in uMonM; or you can use the TFTP client on uMonM to download files to the target.

<sup>6</sup> Pick a MAC and IP address suitable to your network of course.

<sup>7</sup> For more information on moncmd, refer to uMon manual section 18.17 'moncmd'.

Let's try issuing a command to the board over Ethernet using the moncmd.exe tool that came with the umontools.tgz file you downloaded earlier. Assuming the moncmd.exe tool is in your path, you can simply type: <moncmd 192.168.1.200 "tfs ls"><sup>8</sup> to send the string "tfs ls" to your board over UDP and get the response. You'll see the response on your host computer and also an echo of the response will be shown on the uMonM console...

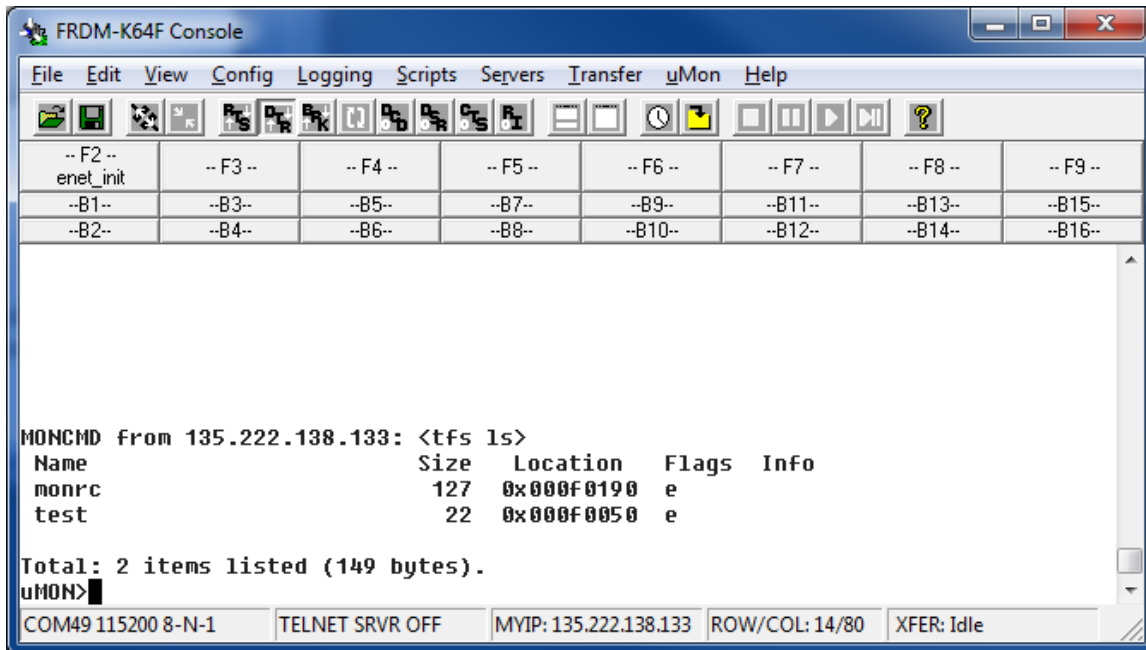


Figure 5: Console output from moncmd

Notice that the console shows the IP address of the host system that issued the moncmd. The same output (without the MONCMD tag) is seen at the host.

## Running with uMonM's Baby Brother Uumon...

Generally speaking, uMonM is a small version of regular uMon, so the operating philosophy can be derived from the main MicroMonitor documentation. This is not the case for uumon. This is a much smaller environment, there is no TFS, and with that removed, several other commands that only make sense with TFS are either removed or trimmed back to give you a very basic, but quite functional bootloader. With around 32K of flash you still get basic network host functionality (DHCP/TFTP), plus serial console and configurable boot modes are still supported; but obviously with a smaller feature set. It still supports non-volatile shell variables that allow you to configure network parameters and set up a custom auto-boot mode; all of this just uses a sector of the internal NOR flash. The same variables used in uMonM are applicable to uumon for network configuration (IPADD, NETMASK, GIPADD and ETHERADD). There's one other "special" variable is "BOOT". If set, this is used as the command that will automatically run at boot time. This would typically be loaded with a jump to some entrypoint; but could be whatever is appropriate.

You can add to this with serial flash, as it is already integrated into the current environment (minus device specific drivers). In the nutshell, the board running uumon boots up, establishes its environment and based on that environment it either boots independently, loads from the network or sits at the monitor level. The BOOT variable is acted upon first, but it can be configured to alternatively (or in addition to) use a custom startup routine that is derived from content found in the serial flash.

So, lets just walk through an example build and bootup of a target using uumon...

<sup>8</sup> The command string (in this case "tfs ls") must be in double quotes.

## Building uumon

Both uMonM and uumon use the same umon\_port/xxx directory. The primary difference is the use of uumon\_config.h vs. umonm\_config.h. This is controlled by the BUILDTYPE variable in the makefile. There should be two entries:

```
BUILDTYPE    = -D BUILD_FOR_UUMON=1      # Use this for building uumon
#BUILDTYPE   = -D BUILD_FOR_UMONM=1     # Use this for building uMonM
```

Only one of which should be uncommented. This will pull in the appropriate xxx\_config.h file for establishing the build configuration<sup>9</sup>. With that in mind, you can configure your build by adjusting the 1/0 state of the INCLUDE\_XXX macros in uumon\_config.h. Then run “make rebuild”. The image to be burned into flash will be build\_<TARGET>/boot.bin.

After this, it is a site-dependent task to get the image installed into the boot sectors of the device. In some of the target port makefiles, I have a “flash” rule that allows you to just run “make flash” to invoke some other tool that is connected to the target to burn in the boot loader. Some boards have OpenSDA-compliant microcontroller built in, so it may just be a matter of connecting to that.

## Booting uumon

With the image now installed for the first time (and assuming the remainder of the flash is erased), the console comes up with something similar to that of uMonM...

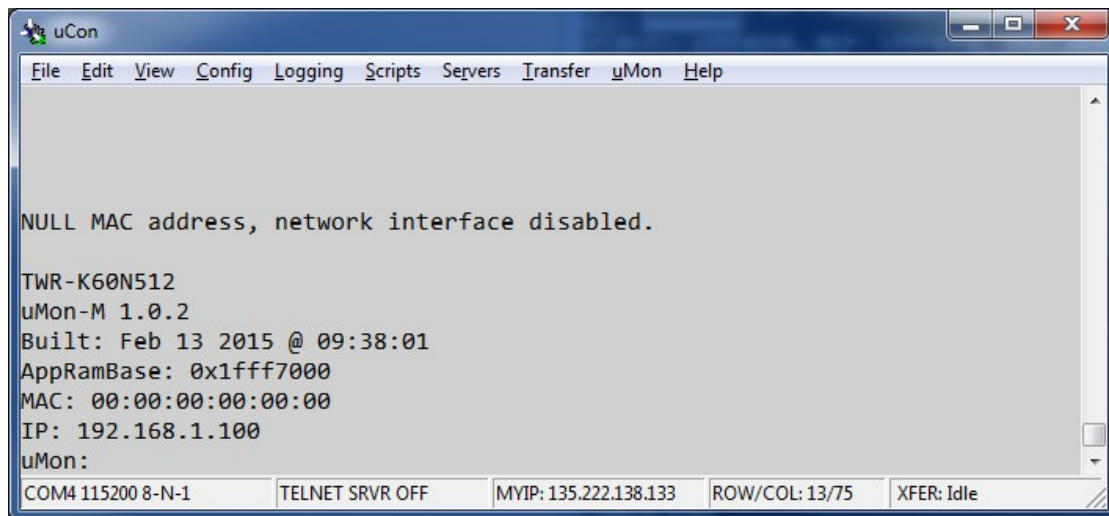


Figure 6: Bootup header for uumon

Note that at this point there is no environment established; hence, it does not have a MAC address assigned (unless it was established in config.h). The command set will depend on what you’ve got configured. For this example, we’ll assume a network bootable system including DHCP and TFTP. The output of the ‘help’ command dumps the same opening header along with the command set. As shown (Figure 7), built for the K60, this system uses less than 35K.

<sup>9</sup> The use of two very similar header files (one for each configuration) is to make it relatively painless to transition between a uMonM build and a uumon build.

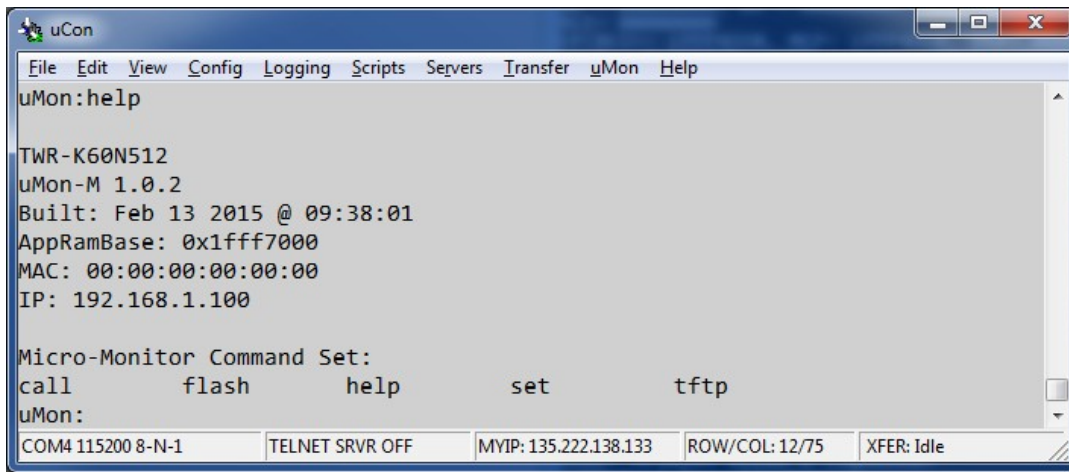
The image shows a terminal window titled 'uCon' with a menu bar containing 'File', 'Edit', 'View', 'Config', 'Logging', 'Scripts', 'Servers', 'Transfer', 'uMon', and 'Help'. The main text area displays the output of the 'uMon:help' command. The output includes: 'TWR-K60N512', 'uMon-M 1.0.2', 'Built: Feb 13 2015 @ 09:38:01', 'AppRamBase: 0x1fff7000', 'MAC: 00:00:00:00:00:00', and 'IP: 192.168.1.100'. Below this is a section titled 'Micro-Monitor Command Set:' with a list of commands: 'call', 'flash', 'help', 'set', and 'tftp'. The status bar at the bottom shows 'COM4 115200 8-N-1', 'TELNET SRVR OFF', 'MYIP: 135.222.138.133', 'ROW/COL: 12/75', and 'XFER: Idle'.

Figure 7: uumon help output

At this point the board is just sitting idle. To configure the network parameters, you need to establish some basic environment.

## Non-volatile Environment

One sector of NOR flash is typically dedicated to storage of the environment variables. This interface is exposed to the user with the “-f” option of the “set” command. The basic philosophy is that at boot time, uumon will initialize its volatile (in ram) environment from the flash-based variables. Additional volatile variables can be created at runtime, but only when the ‘-f’ option is used will it be stored to flash (making it non-volatile). This is where the BOOT and network variables (ETHERADD, IPADD, GIPADD & NETMASK) would typically be stored so that the system can automatically boot up based on some pre-established environment.

The BOOT shell variable is used as the default autoboot mechanism. Whatever is in there is what is used to start the application. In almost all cases, this will be some kind of “call” command. IPADD and ETHERADD are used for network startup. *Note: if the ETHERADD variable is set to 00:00:00:00:00:00, then the boot up will not even include Ethernet device initialization.* In this case, the other network variables are ignored. If we assume ETHERADD is set legally, then IPADD should be set to either a valid IP address or the string “DHCP”.

## Static Network Setup

If you’re in a static environment, then set the ETHERADD, IPADD, GIPADD and NETMASK variables using “set -f” ( to store them to flash). They will be loaded at startup and the Ethernet driver will configure with them appropriately. In this case the BOOT variable will contain the command that will start the application. Thanks to the query-on-boot feature, you have a few seconds to abort the autoboot if you need to do some kind of upgrade.

## Dynamic Network Setup (DHCP)

You can use DHCP just to acquire an IP address, or you may want to use it to load a new image and boot using it. This is dependent on how you configure your DHCP server, as well as configuration of a few macros in the uumon\_config.h file...

- **INCLUDE\_DHCPBOOT:** this is the main macro required for any subset of DHCP capability.
- **INCLUDE\_BOOTP:** enable this if you need BOOTP compatibility.
- **INCLUDE\_DHCPCMD:** enable this if you want to have the dhcp command on the console.
- **DHCP\_FERASE\_FIRST:** first sector to be erased prior to TFTP’ing down a new boot image.
- **DHCP\_FERASE\_LAST:** last sector to erase prior to TFTP’ing down a new boot image.
- **DHCPBOOT\_LOADADDR:** base address at which the new image is written.
- **POSTDHCP\_ACTION:** function to be called when DHCP completes.

If you want to use DHCP for both network IP information and downloading a boot image, then you need to set up DHCP\_FERASE\_FIRST, DHCP\_FERASE\_LAST and DHCPBOOT\_LOADADDR. It is then up to the

DHCP server to feed your system the bootfile and IP address of the server to get it from. In this mode, the DHCP transaction will set the target up with the network addresses (IPADD, GIPADD & NETMASK) plus a bootfile name and tftp server IP address. The target will automatically use TFTP to pull down the bootfile. It does this by initially erasing the sectors specified in the range of DHCP\_FERASE\_FIRST thru DHCP\_FERASE\_LAST, then it downloads the bootfile directly into flash starting at DHCPBOOT\_LOADADDR.

When the DHCP transaction completes, if POSTDHCP\_ACTION is configured, then that will be executed. If it returns (or if POSTDHCP\_ACTION is not defined) uumon will look to the BOOT variable to do the final step. Usually the BOOT variable has some type of call command to jump directly into the application.

## Running Without a Console

In many cases there's no need for a serial port. If that's the case, then enable the INCLUDE\_MONCMD macro and you'll have the ability to communicate with the board over UDP.

## Commands Unique to uumon

Generally speaking the commands in uMonM are the same (probably with many options removed) as the bigger uMon, so refer to uMon documentation ( [http://www.umonfw.com/docs/umon\\_user\\_manual.pdf](http://www.umonfw.com/docs/umon_user_manual.pdf) ) for that. For the most part, commands in uumon follow the same rule; however there are a few commands different enough (due to the uniqueness of the platform) that they are worth mentioning here.

### call: Jump to address in memory

Usage:

```
call [prefix]{ADDRESS}[args]
```

Description:

This command allows the user to execute a function at a raw memory address.

There are two optional prefixes that can tell the command to treat the address a bit differently:

- An asterisk '\*' tells 'call' to dereference the address. In other words, jump to the location that 'ADDRESS' is pointing to.
- The letter 'v' tells 'call' to treat the address as the base of a Cortex M3/M4 vector table. Since the base of the vector table is a 4-byte stack pointer followed by a 4-byte program counter, the 'ADDRESS' is used to load the stack pointer and jump to the program counter.

The optional argument list does not apply if either of the prefixes are specified.

Examples:

```
call 0xc000          # jump to address 0xc000
call *0xc004         # jump to the address stored at location 0xc004
call v0xc000        # treat 0xc000 as the base of a vector table and simulate a reset
```

### set: Set or display shell variable(s)

Usage:

```
set [-cf] [varname] [value]
```

Description:

This command provides access to the volatile and non-volatile environment. With no arguments specified, this just lists the currently active variables in ram. If 'varname' 'value' is specified then it creates (or overwrites) a variable (varname) with a new value. The -c option is used to clear variables and the -f option tells the command to apply the operation to the flash-based variables.

The distinction between flash and ram based variables is that variables that are not stored in flash will not survive a power cycle.

Examples:



```
set                # Display all variables as they currently exist in ram.
set -f            # Display variables stored in flash.
set IPADD 1.2.3.4 # Set variable 'IPADD' to '1.2.3.4' (volatile)
set -f IPADD 1.2.3.4 # Set variable 'IPADD' to '1.2.3.4' (non volatile)
```

## Installing an Application that Uses uMonM's API

Note, these examples assume a uMonM version of the bootmonitor. These applications take advantage of the "uMon" API; allowing many interfaces already used by the monitor to be reused (without code duplication) by the application. As of this writing; there are two example applications that come with the uMonM distribution. One is a simple "hello world" and the other is a complete LWIP-based HTTP server that uses files in TFS as the basis for the pages displayed to the browser.

### *Initial demo application:*

This application is part of the `umon_ports/frdmk64f` directory. It demonstrates how an application can start up on the K64F and hook to the API that is already in flash as part of the uMon environment.

### **Establish MONCOMPTR:**

Before building it, you need to make sure you sync up your runtime API with the downloaded application that will use the API. This is done with something called **MONCOMPTR** (monitor communications pointer). This is just a well-known address (known to uMonM and to the application) that tells the application where to find the API that it can hook to in the uMonM image. In the nutshell, you need to build the application with **MONCOMPTR** defined to the value that "help" displays on the target; so type "help" and notice the line that looks like "**Moncmomp`tr`: 0xNNNNNNNNN**", take the value of 0xNNNNNNNNN and put it in the makefile where MONCOMPTR is set up<sup>10</sup>.

### **Build and Run the Demo**

Now type "make demo" at the bash shell under the `umon_ports/frdmk64f` directory. This basic demo application consists of three files: `demo crt0.S`, `demo_main.c` and `demo_monlib.c`. This is a bare-bones basic example of an application that is installed on top of uMonM in flash.

The file to be installed to flash is "demo.bin", and it needs to be placed at address 0x38000; so the tftp client must push demo.bin to a destination file that is simply named 0x38000. Once this is built, and assuming you've got all the tools in place, you can type "make TARGET\_IP=N.N.N.N demodld" or you can manually install it using tftp in binary mode and specifying the source file as demo.bin and the destination file as 0x38000. This MUST be done in binary mode (tftp uses that mode by default). This assumes that sector 56 is erased (automatically done if "make demodld" is used).

Notice that this executable is NOT installed within TFS space. That's because it will run as an "execute-in-place" image; hence, it is pushed into flash at the exact address from which it will run in memory. Once installed, you can simply type "call -A 0x38000 one two three" to see it run...

---

<sup>10</sup> This synchronization must be done for all applications built to run on your board, and note that the value of MONCOMPTR may change if you reconfigure uMon.

```
FRDM-K64F Console
File Edit View Config Logging Scripts Servers Transfer uMon Help
-- F2 -- -- F3 -- -- F4 -- -- F5 -- -- F6 -- -- F7 -- -- F8 -- -- F9 --
enet_init --B1-- --B3-- --B5-- --B7-- --B9-- --B11-- --B13-- --B15--
--B2-- --B4-- --B6-- --B8-- --B10-- --B12-- --B14-- --B16--

call -A 0x38000 one two three
argv[0]: '0x38000'
argv[1]: 'one'
argv[2]: 'two'
argv[3]: 'three'
Hello world! 0x400 (@0x2000b000)
Modified: 0x410 (@0x2000b000)

FRDM-K64F
uMon-M 0.2.4
Built: Jun 12 2014 @ 09:37:55
AppRamBase: 0x2000b000
MAC: 00:11:22:33:44:55
IP: 135.222.138.20
Watchdog(COP) Reset
uMON>
COM49 115200 8-N-1 TELNET SRVR OFF MYIP: 135.222.138.133 ROW/COL: 20/80 XFER: Idle
```

Figure 8: Console Output of “hello world” demo application

If this doesn't run, it may be because the memory map has changed since this writing. Refer to the demo\_map.ld file and make sure that the rom base is 0x38000 and the ram base is the same value as the “AppRamBase value shown in the uMon boot header (above, in this case 0x2000b000).

### ***Http server application:***

This application is found under umon\_apps/lwip. In the makefile, verify that the BOARD variable is set to FRDMK64F, then you should be able to run the following steps when in that directory (refer to the makefile for up-to-date instructions):

1. type 'make' to build the application (refer to section Establish MONCOMPTR:)
2. type 'make TARGET\_IP=N.N.N.N html' to install the html files
3. type 'make TARGET\_IP=N.N.N.N dld' to install the application

The output at the console will show the reception of each of the web files over TFTP and the final step that burns the application executable into flash at 0x38000. At this point everything is in place to start up the application, so you can type "call -A 0x38000 srvs" at the console and you should see this accumulated output:

```
FRDM-K64F Console
File Edit View Config Logging Scripts Servers Transfer uMon Help
--F2-- --F3-- --F4-- --F5-- --F6-- --F7-- --F8-- --F9--
enet_init --B1-- --B3-- --B5-- --B7-- --B9-- --B11-- --B13-- --B15--
--B2-- --B4-- --B6-- --B8-- --B10-- --B12-- --B14-- --B16--
Watchdog(COP) Reset
uMON>TFTP rcvd WRQ: file <web/index.shtml>
TFTP rcvd WRQ: file <web/umon_pot.ico>
TFTP rcvd WRQ: file <web/umonheader.jpg>
TFTP rcvd WRQ: file <srvr,e>
MONCMD from 135.222.138.133: <reset -x>

FRDM-K64F
uMon-M 0.2.4
Built: Jun 12 2014 @ 09:37:55
AppRamBase: 0x2000b000
MAC: 00:11:22:33:44:55
IP: 135.222.138.20
Watchdog(COP) Reset
uMON>MONCMD from 135.222.138.133: <flash erase 56-70>
15 sectors erased
uMON>TFTP rcvd WRQ: file <0x00038000>
srvr
call -A 0x00038000 srvs
Running HTTP & MONCMD servers...
Network interface configured...
IP: 135.222.138.20
GW: 135.222.138.1
NM: 255.255.255.0
APP:
COM49 115200 8-N-1 TELNET SRVR OFF MYIP: 135.222.138.133 ROW/COL: 26/80 XFER: Idle
```

**Figure 9: Console output of lwip application**

The HTTP server application is running, so using a browser you can attach to the target by simply using the target's IP address as the URL. The browser will show a simple example that allows you to enter uMonM commands and get the response (Figure 10). You can also type at the console and thanks to the WebSockets interface, that output will be dumped to the web page as well.

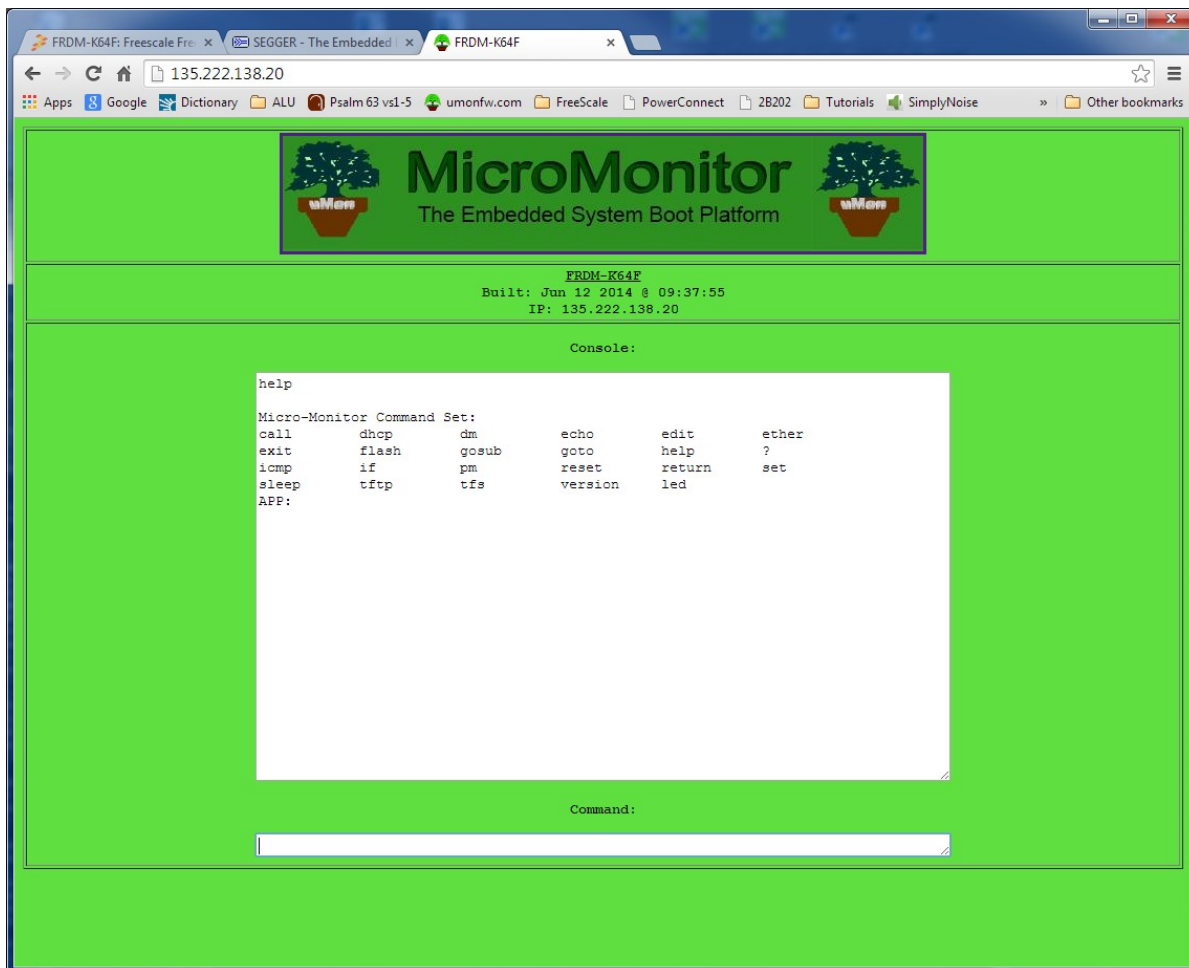


Figure 10: Web interface to lwip application

### ***"Lua" application:***

This application demonstrates that uMonM can support reasonably sophisticated applications. This demo shows Lua running on the K64F accessing files in TFS as Lua programs. As of this writing, the Lua application is "almost" ready. Check the tarball, it may be there. Follow instructions in the makefile.

### **Wrap Up...**

MicroMonitor (uMon) was originally written for bigger memory footprint systems. uMonM is an attempt to bring a subset of the uMon functionality to MCUs. This project is very much a work in progress. I'm still not even sure if I think this kind of bootloader is applicable for these MCUs; so it's as much an experiment as it is a project. Having said that, obviously there are no guarantees of any kind; just try it out and let me know if it's useful or just impractical for these targets.

If you're here and you have questions/comments/bug-fixes/etc..., I'd like to hear other folks' thoughts so shoot me an email: [ed.sutter@alcatel-lucent.com](mailto:ed.sutter@alcatel-lucent.com).