

MicroMonitor: An Embedded System Boot Platform

Updated: Jan 6, 2009

Embedded systems today come in all shapes and sizes. Ranging from small 8-bit (sometimes 8-pin!) devices with as little as 1K of memory, on up to 64-bit multi-gigahertz CPUs, and even multi-core versions of the same. With that comes a similar range of diversity in the systems they control. As a result, there's just no single operating system solution appropriate for all platforms. Sometimes WinCE works, sometimes Embedded Linux works, sometimes neither are appropriate and a smaller, more lean and mean RTOS (real time operating system) is used as the basis for an application, and finally, in some cases there's no need for an operating system at all. This paper doesn't address the embedded operating system issue, there's plenty of text already out there for that. Rather this paper discusses an option for starting up the embedded system prior to running (or even choosing) the embedded OS. The purpose of this paper is to introduce release 1.0 of MicroMonitor (hereon referred to as uMon1.0 or just uMon) as an embedded system boot platform for booting anything from a standalone OS-less application on up to Embedded Linux.

What is MicroMonitor?

The uMon distribution (as of this writing, uMon1.16) is a package of open-source firmware and host-resident tools that build out-of-the box on Linux, Solaris, Windows (Cygwin) and MacOS using GNU cross-compilation tools. The cross-compiled uMon program installs on an embedded system and provides a "startup" (or boot) environment that runs on the target hardware prior to starting up the application. This block of firmware in an embedded system is commonly referred to as a boot monitor or boot loader. Like most boot monitors, it provides the ability to peek and poke memory, test memory ranges, transfer files to/from the target system's memory and turn over control to some other application resident on the target system. This is common for most boot monitors that have any sophistication at all. The uMon boot monitor attempts to raise the bar a bit. It provides all of the above, plus...

- Extensible built-in flash file system (TFS) mappable to NOR flash and/or RAM.
- Support for JFFS2 (NOR) and FAT (SD-Card)
- TFTP client/server for network file transfer
- Xmodem for serial file transfer
- On-board ASCII file creation (i.e. target resident file editor)
- File-based scripts with conditional branching
- ASCII-script-driven startup options
- Command line history and editing
- UDP and RS232 based command entry
- Versatile configuration management using files
- Symbols and shell variables
- Stack trace, runtime profiling and memory-based runtime trace
- Gdb server for application loads and post-mortem analysis
- Network host supporting ICMP and DHCP/BOOTP as a startup option
- Syslog client
- Zlib-based decompression

- Password-protected user levels
- Large API to hook the application to facilities provided by monitor
- Several demo applications including an LWIP-based HTTP server without any RTOS, just using the uMon API hooks.

One important aspect of a boot monitor is its “transparency” to the developer. In other words, it should never hinder the development process. With that in mind, uMon was designed to be very easy to port to new target systems. It runs without enabling interrupts, so aside from basic cpu & memory configuration done at reset, it can be installed on a target with a simple polled UART driver. Then a flash driver and ethernet driver (if applicable) can be installed and after that TFS and the network facilities will “just work”. Another aspect of “transparency” is that it provides several hooks (API) to allow the developer to use some of uMon’s facilities; however, it does not require that these hooks be used. Once uMon turns over control to the application, the application can choose to use or not use uMon’s API. In several cases, it turns out that uMon’s API is used early in the startup of an RTOS for trace and debug, then once the RTOS has completed initialization, RTOS-based facilities override the uMon-based hooks. The point is that the application can choose to use uMon’s API for the facilities it provides or it can ignore the fact that uMon is even in the system.

MicroMonitor’s Typical Usage Model

The following paragraphs document a typical usage scenario for an embedded system that uses uMon.

System Startup

The uMon executable resides within the instruction space that the CPU’s reset vector points to. The CPU/target system boots uMon first. The startup code in uMon then does some basic initialization of the memory (flash and ram), serial port and ethernet port (if applicable). Since uMon has a file system (Tiny File System: TFS) built in, the startup of a uMon based embedded system is very configurable because uMon uses files (see example listing below) in the file system to start up.

```
uMON> tfs ls
Name                Size   Location  Flags  Info
monrc                203   0x103ca64c e      envsetup
romfs.img           2216960 0x1008005c
startlinux          5041   0x103c923c e
zImage             1228056 0x1029d4bc

Total: 4 items listed (3450260 bytes).
uMON>
```

This is conceptually similar to the .bashrc or .profile files used to configure the startup of a user’s environment on Unix, or the autoexec.bat file used to configure the startup of a DOS based machine. The idea is that this startup file, called monrc (**mon**itor **run** **con**trol file), allows the user to establish basic configuration of the system. This typically includes the network host information like IP, NetMask, Gateway IP, etc.. Initially, this file can be created on board with uMon’s built in ASCII file editor, or it can be transferred to the target via Xmodem or TFTP. The content of the executable monrc script should be kept simple, basically used to set up a few shell variables...

```
uMON> tfs cat monrc
```

```
set ETHERADD 00:23:31:36:00:01
set IPADD 192.168.1.110
set NETMASK 255.255.255.0
set GIPADD 192.168.1.1
```

Once the monrc file has been executed (during uMon's internal startup), uMon then configures the serial and ethernet ports based on the content of a few specific shell variables (CONSOLEBAUD, ETHERADD, IPADD, etc...) that are assumed to have been set up as a result of the monrc script execution. For security purposes, this automatic execution of the monrc file is usually non-interruptible; hence, it guarantees some basic startup configuration will be invoked.

Now that uMon has initialized itself through the monrc file, it has several different potential paths, all of which depend on files in TFS. If there are no additional "auto-boot" files in TFS, then uMon simply sits at the console/network ports waiting for input from the user (either from RS-232, ICMP, UDP, GDB or TFTP). A typical command list dump (output of the 'help' command) at the uMon console is shown below.

```
uMON>help
```

```
Micro-Monitor Command Set:
```

```
arp          call          cast          cm            dhcp          dis
dm           echo           edit          ether         exit          flash
fm           gdb            gosub         goto          heap          help
?           history        icmp          if            item          mt
mtrace      pm             read          reg           reset         return
set         sleep         sm            strace        syslog        ulvl
tftp        tfs           unzip         xmodem        version       ldatags
```

```
uMON>
```

If on the other hand, there are additional "auto-boot" files in TFS, then uMon will execute them in alphabetical order. Typically, only one "auto-boot" after monrc is run (the application); however, uMon allows the user to configure this as needed. For example, it may be appropriate for an auto-boot script to first query the network for a server, then if found, download and run some application, and if not found, just run some on-board default application (see below). There are all kinds of script-configurable options.

```
1: icmp -v PING_RESULTecho 135.222.140.142
2: if $PING_RESULT sne ALIVE goto LOCAL_BOOT
3: echo Attempt network boot...
4: tftp -Fnet_app -fe 135.222.140.142 get net_app
5: if $TFTPGET seq \ $TFTPGET goto LOCAL_BOOT
6: net_app
7: goto DONE
8:
9: # LOCAL_BOOT:
10: echo Run local copy of application...
11: local_app
12:
13: # DONE:
14: echo Finished!
```

Referring to the listing above, line #1 is uMon's equivalent of a ping command (icmp echo). The command populates the shell variable PING_RESULT with the string "ALIVE" if the icmp echo succeeds. Line #2 tests to see if the ping succeeded and if not, it causes the script to branch to the LOCAL_BOOT tag (line #9). This simply runs a locally stored copy of the application called "local_app". If the test at line #2 finds that the icmp echo succeeded (i.e. \$PING_RESULT == "ALIVE") then a TFTP request is sent to a server at 135.222.140.142. A second test is made (line #5) to see if the TFTP transfer succeeded, if not, the script once again branches to LOCAL_BOOT. If yes, then it is assumed that the net_app application was transferred to the board via tftp and can be run.

The above script is just an example of the versatility that can be scripted into a uMon based target startup.

Application Runtime

So, based on the previous section, the application "somehow" has started up. The application has the option to totally ignore the fact that uMon is installed, or it can choose to connect itself to uMon's API and take advantage of uMon's console access, TFS, uMon's heap, environment variable access and some of the debugging facilities like memory based runtime-trace. Plus, depending on the CPU, it may be quite convenient to just allow uMon's exception handlers to remain installed so that any erroneous exceptions will be caught and will be traceable (via stack trace) when the exception returns control to uMon.

Note that this API discussion assumes that the operating system allows execution of these functions as they exist in the instruction space that uMon was built for. This means that some MMU-based OSes (i.e. Linux) may not be able to access this functionality simply because the memory space occupied by uMon is not mapped for execution once the MMU is turned on.

The **console API** (mon_putchar(), mon_getchar(), mon_getline() and mon_printf()) allows the application to hook to the functions in uMon that support raw and formatted console IO.

The **command line API** (mon_getline(), mon_docommand(), mon_addcommand()) allows the application to take advantage of uMon's entire command line interpreter including the command line editing and history. This allows the application to insert commands into the uMon command table at runtime, present this modified set of commands to the user and execute any of the commands in the command table as needed.

The **environment API** (mon_getenv() and mon_putenv()) allows the application to retrieve variables that were established prior to the application starting up. This gives the application the ability to retrieve a variety of different things. For example, the target's network host information (IP, NetMask and Gateway IP addresses). Also, different portions of the application may have need to run in different runtime configurable modes, with the most obvious one being "DEBUG" mode. This mode could be made runtime-settable by simply establishing the DEBUG shell variable in monrc at startup, then when the application runs, it can detect the presence of this variable and enable its own internal debug flag.

The **heap access API** (mon_malloc(), mon_free(), mon_realloc()) allows the application to use uMon's heap. While it isn't usually a good idea to use malloc/free in an embedded system, there are times when you just gotta have it. The "heap" command at the monitor's command line interface (CLI) also allows the user to display the state and content of the heap. This allows

the user to catch the high-water mark of the allocated space, plus it can be used to debug and track down corruption and memory leaks.

The **file system API** (too many to list) provides the application with easy access to the files in TFS so that the application can read/write/modify/create files in several different ways/modes.

The **flash API** (`mon_flashwrite()`, `mon_flasherase()`, `mon_flashinfo()`) allows the user to modify raw flash through a standard API. uMon's flash space need not be dedicated entirely to TFS. It can be configured with TFS owning only a portion of the overall flash space; thus allowing the application to do whatever it wants to do with some block of flash. With this configuration, there is a use in having application-accessible API to the raw flash.

The **runtime trace API** (`mon_memtrace()`) allows a user to insert "printf-like" calls into portions of the code that may not be friendly to `printf()` (i.e. interrupt handlers, etc..). The formatted string is placed in a pre-configured buffer and can be dumped to the console later by using the "mtrace" command at the uMon command line.

The **runtime profiler** in uMon allows the application to periodically record the instruction pointer of the application and then, some time later, uMon will organize the information logged into a statistical, per-function representation of the runtime execution of the application. This allows developers to concentrate on the "heavy hitter" functions in the application.

Application Post Mortem Analysis

uMon supports some post-mortem debug. For example, assume the application terminates either legally (via `mon_appexit()`) or illegally (via some exception). If the application allowed uMon to catch the exception, then uMon's stack trace facility can be used to determine the function nesting at the time of the exception. All of the core of the application is still in the memory space accessible by uMon; hence, that space can be analyzed with gdb or the symbolic capabilities built into uMon. The basic gdb server built into uMon's network interface supports the "load" and "c" commands in gdb, plus variables and memory can be dumped using gdb's symbolic access commands like "print". The monitor supports symbolic access of variables in the application. This allows the user to display memory with the command: "dm %variable_name" instead of needing to specify the hard address of the variable. This is done through the CLI's ability to process symbols by looking them up in a "symtbl" file assumed to be installed in TFS, plus it doesn't require any external debugger.

RTOS & CPU Independence

Generally speaking, uMon doesn't care what RTOS (if any) you incorporate into the application. It has been used with Linux, VxWorks, Nucleus, CMX, eCOS, uC/OS-II, pSOS, WinCE, RTEMS and standalone (no OS at all). Obviously the memory map of the application must not conflict with that of uMon; however if it does, then just adjust uMon's memory map. As far as CPUs, well it's *almost* CPU independent. It has ports that run on ARM, Xscale, MIPS, PowerPC, Virtex-PPC, ColdFire/68K, Blackfin, SH, MicroBlaze and NIOS. Obviously there are limits regarding which CPU uMon can run on. Generally speaking, uMon wants a CPU that supports linear address space (no bank switching) and a target that has flash/ram that can support the uMon footprint and usually one or the other (or both) of RS-232 and/or Ethernet port.

For More Information...

As each release of uMon is made available to users, with that comes an update to the user's manual (see pointer below). The first section of the manual provides the user with a summary of the changes between one release and the next.

The MicroMonitor distribution comes as a compressed tar ball (.tgz) file. It includes the source for the host-based tools used, plus the common and port-specific code for a variety of targets. In addition, there is a template port directory that can be used as the starting point for a new port (assuming one of the ports already available isn't more appropriate). Finally, the distribution comes with a few general examples of applications that can be built to hook up to and run on top of a uMon based target.

If the environment described sounds interesting, then refer to the website <http://www.microcross.com/html/micromonitor.html> for a 300+ page user manual and the above mentioned tar ball. For general questions, refer to the documentation or contact the primary author, Ed Sutter, at <mailto:esutter@alcatel-lucent.com>.

Ed Sutter is a distinguished member of the technical staff at Alcatel-Lucent Technologies. He has written articles for Embedded Systems Programming magazine and Circuit Cellar Online, and has authored the book "Embedded Systems Firmware Demystified" published by CMP in February 2002. He is a graduate of the New Jersey Institute of Technology has been working with embedded system firmware for over 25 years.