MetroTRK

Target Resident Debugging Kernel
for Embedded Systems

Freescale Semiconductor
Author, Rick Grehan

MetroTRK is the Freescale Debugger's ambassador to the growing array of embedded processor variants and their accompanying development boards. It is the "transparent eyeball" that allows the host debugger to debug the target application at full execution speed, with minimal effect on the target environment.

CONTENTS

Freescale's Target Resident Kernel, MetroTRK, is a software debugging monitor for embedded systems. MetroTRK runs on the target hardware, shepherding the application being debugged, and communicating with a host debugger that executes on Windows® or Unix® desktop systems. MetroTRK is controlled via the host debugger, which transmits commands to MetroTRK for execution on the target. MetroTRK sends back command responses as well as information describing any exceptional events occurring on the target.

MetroTRK was built for use with the highly-acclaimed Freescale Debugger, part of Freescale's multi-language/multi-target integrated development environment. MetroTRK brings the Freescale debugging environment to a wide variety of embedded development boards. Currently, MetroTRK has been ported to over 18 embedded boards, hosting processors that range from MIPS to PowerPC® to 68K. (And, by the time you read this, MetroTRK will likely be ported to even more development systems.)

New embedded processor variants are appearing almost daily, however, and this flood of new embedded CPUs has contributed to a widening gap in the most critical stage of the development process: debugging.

MetroTRK fills that gap.

MetroTRK is the gateway whereby the Freescale embedded toolset–compilers, linkers, and debuggers–brings its full strength to bear on a growing range of embedded development systems. Freescale compilers are already unparalleled in their support of languages, processors and operating systems, thus securing the front-end of the development process: project management, code writing and compiling.

MetroTRK seals the back-end of the development process: debugging.

1. MetroTRK Features

MetroTRK is far more than a simple debug monitor. MetroTRK was crafted with features that make it simultaneously portable and powerful.

Open Architecture. MetroTRK is available as well-written, well-documented source code. When you need to move it to new hardware, MetroTRK provides full access to its internals—no secrets.

Well-Factored. The structure of MetroTRK is such that its hardware-dependent routines are cleanly separate from its hardware-independent routines. This de-coupling makes adapting MetroTRK to new hardware easy; you focus your attention on only a minimal subset of the overall software. You don't have to modify the entire system. There is no "spaghetti" code to work through.

Extendible. The internal heartbeat of MetroTRK is driven by an event loop. All activities—commands from the host debugger, as well as exceptions originating on the target—pass through the event loop. Since command and response processing is centralized, it's easy to add new features.

Flexible Communication. The communication protocol used by MetroTRK is independent of the physical medium employed. MetroTRK can converse with the host debugger through a simple serial connection or a more responsive TCP/IP (UPD) communication link. Since source code is provided, an intrepid engineer can add other protocols (bi-directional parallel, for example).

Small Footprint. The lean design of MetroTRK is built on the notion that high-level debugging "intelligence" should reside in the host debugger, not in the target debug monitor. This keeps memory consumption on the target to a minimum without compromising overall functionality.

Non-Invasive. A problem with debugging any system is the effect the debugger has on the "debuggee". The presence of the debugger can often alter the runtime environment; the result being that the target code (or application) behaves differently while being debugged than it will behave when released. MetroTRK consumes CPU cycles only when it must (in response to a command from the host debugger or when alerting the host to an exceptional condition on the target). So, during normal execution the debug target is unaware of (and unaffected by) the presence of MetroTRK.

Hardware "Smart". MetroTRK is designed to be implemented on different hardware platforms. Therefore, it isn't simply hardware "neutral"; it is hardware "smart". The engineers who crafted MetroTRK have made it adaptable to virtually any embedded microprocessor. For example, MetroTRK resides with equal comfort on little-endian and big-endian processors. In addition, memory structures within MetroTRK are intentionally built so that they can reside on Princeton (von Neumann) architecture processors as well as Harvard architecture processors.

In short, the characteristics of MetroTRK make it both flexible and powerful. Though it supports a rich collection of debugging capabilities, these capabilities are not accompanied by unnecessary complexity that renders the source code opaque. Developers who want to extend the capabilities of MetroTRK by moving to new hardware or adding new commands will not find their task daunting.

2. Flexibility by Design

MetroTRK executes in cooperation with the host debugger. Acting as a remote agent on the target board, MetroTRK receives commands transmitted from the host and executes those commands on the host debugger's behalf. MetroTRK is the host's ambassador to the target and works as a sort of translation layer between the host debugger and the target hardware and software. For example, an engineer can debug a PowerPC embedded target (running MetroTRK) from an x86-based Windows host. That same engineer, using the same x86 host hardware, can (with MetroTRK) debug a MIPS target or a 68K target.

From that engineer's point of view, though the PowerPC target has changed from MIPS to 68K, the host tools have remained largely the same. The only changes the engineer will have experienced are nuances in the host debugging environment that necessarily differed for each target processor. A single host station can debug a wide range of targets.

MetroTRK is designed to be as non-invasive as possible, a "transparent eyeball" on the target that sees but does not disturb. Not only is MetroTRK memory-frugal, it is crafted so that the target application is essentially unaware of the debugger's presence. The target application executes at full speed; MetroTRK takes no CPU cycles from the target application during normal operation. As stated earlier, MetroTRK will consume CPU cycles only when it must.

Finally, not only is MetroTRK ideal as a "stand-alone" debugger, where MetroTRK and the target application are the only occupants on the target hardware, but it is also comfortable executing in conjunction with a real-time operating system (RTOS) or OS kernel. In such an environment, MetroTRK runs as a task or process and can use the debugging services provided by the operating system. MetroTRK can tap into the OS's knowledge of the target application being debugged, and has access to the internal state of OS objects, such as semaphores, mutexes, queues, etc, that are in use by the application. MetroTRK can pass this state information back to the host debugger, and thereby provide the developer with invaluable information about the target application's runtime environment. In addition, for those operating systems that are multitasking, MetroTRK can debug multiple tasks/processes and even multiple threads.

3. Under the Hood—In Action

The MetroTRK Core implements a service loop for managing incoming commands and outgoing replies. A pair of queues, the request and command queue (incoming) and the reply and notification queue (outgoing), carry the flow of information into and out of MetroTRK. Requests in the form of commands generated by both the host debugger and the target "debugee" arrive at the request and command queue, and are placed at its input end. When a request arrives, the MetroTRK Core removes the request packet from the queue, examines the packet's content and dispatches it to the appropriate handling code.

Similarly, responses that must be sent back to the host debugger pass through the MetroTRK Core, and are "packet-ized" into outgoing reply packets. The MetroTRK Core places these packets on the outgoing (reply and notification) queue for ultimate transmission back to the host.
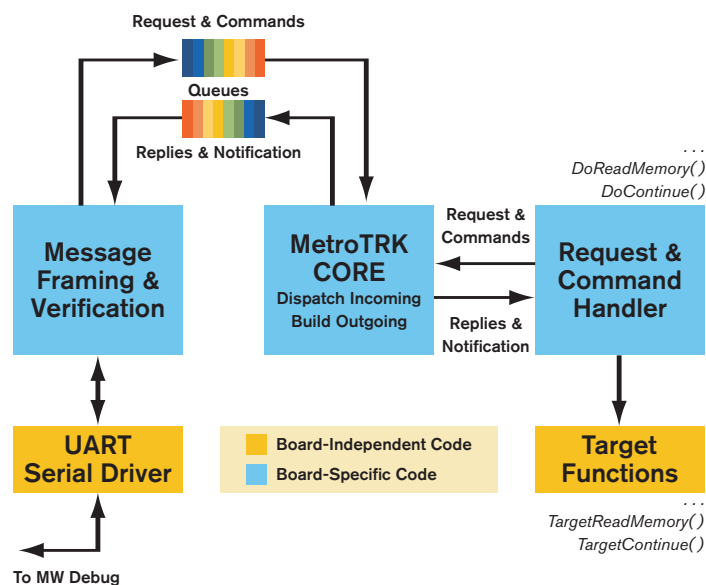
FIGURE 1. METROTRK INTERNAL ARCHITECTURE. NOTE THAT HARDWARE SPECIFIC COMPONENTS ARE WELL ISOLATED. IN ADDITION, DATA COMMUNICATION BETWEEN METROTRK AND THE HOST DEBUGGER (MW DEBUG) IS REGULATED BY A PAIR OF QUEUES.

The queues serve two important purposes. First, they act as buffers between the host debugger and MetroTRK, each of which is likely to be running on different hardware platforms. The buffers keep faster hardware from outrunning the slower hardware so requests and replies won't get "dropped". Second, they allow MetroTRK to be built along "event driven" lines. All communication with the host passes through the queues and is processed by a "central dispatcher" within the MetroTRK Core. Note that MetroTRK handling requests is independent of the physical communications medium* as well as the processor. As you can see in Figure 1, the architecture of MetroTRK keeps those components that manage the physical transmission of commands and responses cleanly separated from the "generic" portions of MetroTRK. Those components that carry out the commands ("Target Functions" in Figure 1), and which must deal with the processor hardware, are separated from the MetroTRK generic code.

(*Communication between the host and target uses a simple packet-based serial protocol with checksum. This protocol is independent of the physical layer (i.e., RS-232 serial or TCP/IP). Transmission rates for serial transfer can range from 9600 bps to 230.4 Kbps, depending on the capabilities of the hardware.)

4. Under the Hood–Structure

MetroTRK was designed with flexibility in mind. Not only is it flexible in regard to where it resides on the target, but also how much target resource it consumes. MetroTRK is ROMable. It can be downloaded into Flash, or burned into ROM for immediate access from the target hardware. MetroTRK occupies only 26K of code space. (An additional 6K of RAM is needed for each stack and variable storage.) MetroTRK can run just as easily from RAM, consuming a mere 38K of memory.
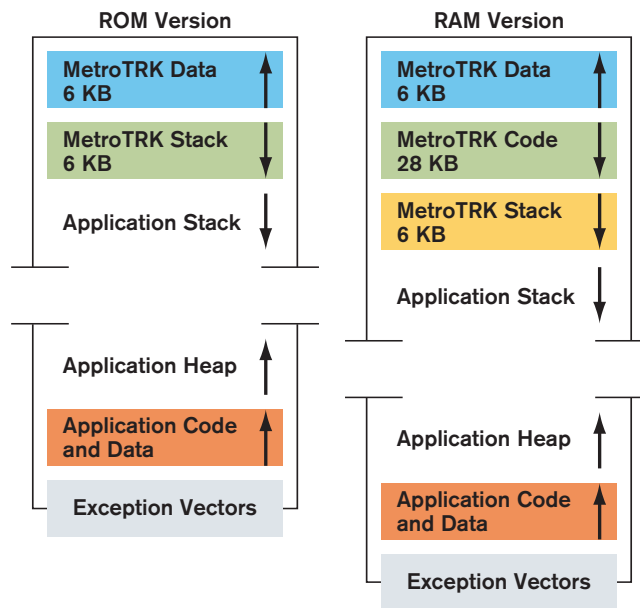
FIGURE 2. A MEMORY MAP OF METROTRK, RUNNING IN ROM (LEFT) AND RAM (RIGHT). THE EXACT LOCATION OF THE "PIECES" OF METROTRK VARIES FROM TARGET BOARD TO TARGET BOARD. TYPICALLY, METROTRK IS PLACED AT THE HIGH END OF THE PROCESSOR'S MEMORY MAP. THIS LEAVES A LARGE, CONTIGUOUS MEMORY SPACE AVAILABLE AT THE LOW END OF THE PROCESSOR'S MEMORY FOR THE APPLICATION BEING DEBUGGED.

The memory architecture of an example target being debugged is shown in Figure 2. Typically, MetroTRK is loaded at the high end of the processor memory. This is because exception vectors (ISR vectors on some processors) usually reside at the low end of a processor's memory map, and developers usually locate application code just above the exception vector space. With this arrangement, it's often safest to place MetroTRK at the high end of the processor's memory map. Of course, the precise location of MetroTRK is entirely optional and is usually dictated by the memory topology of the target development board.

5. Under the Hood—MetroTRK Source-Code Routines

A look at the source code internals of MetroTRK reveals a collection of routines that have been grouped according to the roles they play in the overall MetroTRK application.

This grouping serves several purposes:

Hardware-dependent and hardware-independent routines are separated. This means that a developer porting MetroTRK to a new platform must modify a minimum number of routines. Code in the hardware independent portion of MetroTRK suffers no side effects from the porting process.

Coupling between routines is kept to a minimum. This means that, should a developer have to modify MetroTRK to add or remove a capability, which would require alterations to the hardware-dependent portion of the code, changes are isolated to as few routines as is possible.

The structure of the MetroTRK source is designed for easy programmer navigation; it is easy to "reverse-engineer" MetroTRK. Any programmer who has ever been faced with modifying code not his (or her) own understands the trauma of learning the ins and outs of a foreign corpus of source code. A lot of time is spent simply learning the "lay of the land". It is not unlike exploring a foreign city without a map. During the building of the MetroTRK source code, its engineers kept this inevitable learning curve in mind. MetroTRK was written not only to be portable or easily modified, but to be readable. Partitioning of functionality helps immensely in its readability. There is no spaghetti code here. Independent activities within MetroTRK are isolated to the minimum number of routines.

5.1 Routines

Each routine in the MetroTRK source code falls into one of five categories. This categorization of routines is another instance of the MetroTRK easy reverse-engineering design. If you need to discover which routine covers a particular activity in MetroTRK, you need only ask yourself what operation the routine is responsible for, and that will naturally lead you to one of the five categories. The routine of interest will most likely be within that category.

The categories are:

Initialization Routines. This category consists of the "start-up" routines, whose job it is to initialize registers, verify memory, copy exception vectors from ROM to RAM and so on.

Communication Routines. There are two groups of routines within this category. Routines in the first group manage communication between the debugger on the host and MetroTRK (on the target). These include routines to initialize the physical communication hardware (typically a UART), read and write bytes, poll the communication hardware, etc. Routines in the second group provide the means for the target application to access the host filesystem. For example, the target application can open a data file on the host and download its contents for processing (an ideal way for providing test-vector data to your embedded application).

Execution and Control Routines. As the name implies, routines in this category manage the execution of the target being debugged—adding breakpoints, executing a single-step command, stopping the application, etc.

Memory/Register Routines. Routines in this category perform the actual interaction with target hardware. They provides MetroTRK with its ability to read and write memory and read and write processor registers (including registers within the floating-point coprocessor).

Maintenance Routines. This final category includes miscellaneous routines that allow the host debugger and MetroTRK to exchange version information. Routines in this category also allow the host debugger to query the currently executing MetroTRK and determine what commands it supports.

6. MetroTRK Command Repertoire

MetroTRK is already equipped with a sizable repertoire of commands. The commands cover the full range of activities needed to manage a remote debug session, including:

> Downloading the target application

> Controlling the application as it executes (e.g., single-stepping)

> Monitoring the target environment (processor registers, memory, etc.)

> Responding to error conditions on the target (e.g., processor exceptions)

MetroTRK also includes commands that allow the target application to access the host's filesystem.

Commands. Currently, MetroTRK recognizes nearly 50 commands. As with the MetroTRK source code routines, the commands are organized into three categories ("command sets").

Primary Command Set. Commands in this category provide basic functionality to MetroTRK. These commands deal with reading and writing processor memory and registers, connecting to the target, starting the target executable and querying the MetroTRK version. The commands are:

> read memory write registers

> write memory go (continue)

> read registers connect

> versions

Extended Command Set. Commands in this set provide the "next level" of MetroTRK capabilities. They include commands for stopping and resetting the system, for performing single-step operations and setting breakpoints and watchpoints, for determining the CPU type, filling and copying memory, clearing the processor cache and downloading the image to be debugged. The commands are:

> reset ping

> step get CPU type

> stop system fill memory

Application Level Command Set. Commands in this category provide process and thread control, breakpoint control (within threads) and file I/O. In addition, commands in this set give the host debugger control of OS-specific items (such as semaphores, queues, etc.) The commands are:

> set breakpoint

> copy memory

> clear breakpoint

> flush cache

> set watchpoint

> download image

> clear watchpoint

> create (launch) thread get processes

> create (launch) process get process state

> create (launch) DLL get threads

> create OS-specific items get thread state

> delete (kill) thread get DLLs

> delete (kill) process get DLL state

> delete (unload) DLL get OS-specific info

> delete OS-specific items open file

> stop thread read file

> stop process write file

> set thread breakpoint close file

> clear thread breakpoint set file position

> set thread watchpoint

> clear thread watchpoint

Finally, it is important to point out that the host debugger is not the sole initiator of the flow of information in and out of MetroTRK. Events on the target will cause MetroTRK to asynchronously send a "notification packet" to the host debugger. These events are usually triggered by some activity on the target, such as a thread being created, an exception being thrown, and so on. MetroTRK also provides a means by which the target application can access the host filesystem. When the target application makes such a filesystem access call, the call is fielded by MetroTRK, which sends the appropriate notification packet to the host and manages the flow of data back to the target.

The **"MetroTRK-Generated" commands** include:

> thread created (launched) thread stopped

> process created (launched) MetroTRK exception

> DLL created (loaded) MetroTRK internal error

> OS-specific item created

> open host file

> thread deleted (killed)

> read host file

> process deleted (killed)

> write host file

> DLL deleted (killed)

> close host file

> OS-specific items deleted

> set host file position

7. Systems Supported

As stated earlier, MetroTRK operates in conjunction with the Freescale Debugger. The Freescale Debugger runs on a host development system; currently, Freescale supports hosts for Windows NT, Windows 95, and Windows 98, with Unix systems available by the time you read this. (Freescale does provide development tools for the Macintosh, but host development of embedded systems using MetroTRK is not supported on the Macintosh.)

MetroTRK is already available on a wide array of development boards and executes on the most popular 32-bit embedded processors. The growing list is shown in Table 1.

TABLE 1. METROTRK CURRENT PROCESSOR AND BOARD SUPPORT

PROCESSOR	BOARD
PowerPC	Cogent CMA10 (with 603e or 740)
	Freescale ADS 821/860 Freescale EVB 505/509
	Freescale MBX 821/860
	Phytec PowerPC 505
NEC V8xx	Midas RTE-V830-PC
	Midas RTE-V821-PC
	NEC RT-V831
	NEC RT-V853
	Midas RTE-V831-PC* Midas RTE-V853-PC
	Midas RTE-V850E/MS1-PC
MCore	Powerstrike CMB/EVB 1200*
MIPS	IDT 79S381 Midas RTE-V4100-PC Midas
	RTE-VR4300-PC Midas RTE-VR5000-PC LSI BDMR4101*
M68K	Freescale M68328ADS
	Freescale M68EZ328ADS*

(*Support for these boards was pending at the time this whitepaper was being written.)

8. MetroTRK On Board

It is well known that the embedded market is one of the fastest growing regions in the entire computer landscape. And within the embedded market, 32-bit processors comprise one of the fastest growing areas of development. Manufacturers are pitching new processor variants faster than developers can catch them. And each variant possesses its own mixture of on-chip features and peripherals, each aimed at a particular niche in the embedded space.

For the embedded developer, this growing smorgasbord of embedded processors is certainly a welcome sight. But, there's a hidden catch: the availability of tools. A new processor, even if announced in conjunction with development hardware, is useless without software tools.

The Freescale IDE—its compilers, linkers, and project management—already provides the cross-platform target software generation capabilities developers need. But a piece has been missing: a way to debug the target software actually running the target application on its intended processor, controlled by a debugger executing on the host development system.

MetroTRK is the missing piece. It is portable, and so becomes the Freescale Debugger's ambassador to the growing array of processor variants, their accompanying development boards, and the operating systems that run on them. It is the "transparent eyeball" that allows the host debugger to debug the target application at full execution speed, with minimal effect on the target environment—critical in many embedded real-time systems. Best of all, it is available in compact, well-factored, well-documented source code. As new embedded processors arrive, MetroTRK will be ready to jump onboard.

freescale
semiconductor