CodeWarrior®

MetroTRK Manual

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 101697 bpb

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation P.O. Box 334 Austin, TX 78766 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Mail order	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction

The Metrowerks Target Resident Kernel, or "MetroTRK" for short, is an on-target debug monitor for use with the CodeWarrior Debugger. This manual serves as a reference to MetroTRK, as well as a guide explaining how to customize it for use with your hardware configuration.

Read the Release Notes!

Before using MetroTRK, please take the time to read the release notes in the release notes folder. They contain important information about new features, bug fixes, and any late-breaking changes that may have occurred *after* the production of this manual.

About the MetroTRK Manual

This manual is organized into the following three chapters:

- Introduction
- Understanding MetroTRK
- Customizing MetroTRK
- MetroTRK Function Reference

You are reading the introduction now.

"Understanding MetroTRK" describes the various tasks MetroTRK performs and how these tasks are implemented. It gives a "big-picture" view of MetroTRK as well as enough detail to understand what you need to know to re-target MetroTRK to new board configurations.

"Customizing MetroTRK" goes into more detail in the specific areas where you customize or re-target MetroTRK to work with your hardware configuration.

"MetroTRK Function Reference" describes each function mentioned in the manual, as well as any other important MetroTRK utility functions.

What is MetroTRK?

MetroTRK allows you to use the CodeWarrior debugger, MW Debug, to debug programs running on an embedded system. The MetroTRK program lives on the embedded system (along with the target program) and communicates with MW Debug by way of a serial connection. Through this serial communication, MetroTRK acts as MW Debug's interface to the target hardware.

MetroTRK communicates with MW Debug to service requests (like the request for a register or memory value) as well as to notify the user of runtime events or exceptions as they occur on the target. In doing so, MetroTRK supplies all the target-side services necessary to provide various levels of debugging from within MW Debug.

Since MetroTRK manipulates all of the important target resources, its implementation inherently depends on the specifics of the hardware being targeted. For this reason, MetroTRK is distributed as source code so that you can modify portions of it for use with novel hardware configurations. The primary purpose of this manual is to point out the important concepts and sections of code that you will need to know in order to customize MetroTRK based on the details of your board configuration.

MetroTRK Compatibility

MetroTRK can be used with many board configurations based on the processor or family of processors supported by your CodeWarrior tools. There are, however, some minimum requirements for MetroTRK to be able to function properly.

Serial controller

Since MetroTRK must communicate with MW Debug running on the host computer, your hardware setup must be capable of conforming to the serial protocols used by MW Debug. In particular, the hardware must have a serial port that can communicate between 300 baud and 230.4k baud.

RAM

MetroTRK always takes up some RAM space on the target. If you are running MetroTRK from ROM, it takes up about 6KB for its global data. If you are running MetroTRK from RAM, it takes up an additional 26KB for its code section. In addition, MetroTRK needs 6KB for its stack. For more information on how MetroTRK uses RAM, see "Where MetroTRK Lives in Memory" on page 16.

Starting Points

This manual is intended as a general reference to the implementation of the MetroTRK system. It is meant for users who need to customize MetroTRK to work with a new board configuration. Details about how to actually load and use MetroTRK are not covered in this manual because they will vary from one set of CodeWarrior tools to another. Instead, they are covered in the Targeting manual for your target processor.

One or more MetroTRK implementations are included in your CodeWarrior distribution. These implementations are configured for specific reference boards. If you are using one of these reference boards, you should be able to use MetroTRK without any modifications whatsoever. If this is the case, you probably don't need to consult this manual, as most of the information talks about the *implementation* of MetroTRK, not how to debug using MetroTRK. Your Targeting manual explains which reference boards are supported.

If you already know how to use MetroTRK but need more specific details about the implementation in order to build your own customized version, you're in the right place! The chapter "Understanding MetroTRK" discusses the MetroTRK architecture in gener-

al, and the chapter "Customizing MetroTRK" spotlights which parts of MetroTRK will need to be customized for use with new board hardware…and how to do it.

Where to Learn More

This manual only covers the implementation of MetroTRK. There are many things not covered here which you may need to know in order to use MetroTRK productively. This section lists materials that might be useful.

Code Warrior Documentation

CodeWarrior IDE User Guide: The manual for using the CodeWarrior IDE (Integrated Development Environment.) The IDE is used for creating, organizing, and building development projects.

CodeWarrior Debugger Manual: The manual for using MWDebug, the CodeWarrior source debugger.

CodeWarrior Targeting (your target processor here): The "Targeting" manual will be extremely helpful in working with your embedded project. In fact, the targeting manual covers how to *use* MetroTRK with your target processor. In addition, it contains all other information about using CodeWarrior that is specific to your target processor.

2

Understanding MetroTRK

This chapter describes the basic functioning of the Metrowerks Target Resident Kernel (MetroTRK) and its various modules.

Understanding MetroTRK Overview

This chapter describes the internal functioning of MetroTRK, describing how the source code is organized into various distinct submodules. The sections in this chapter are:

- How Does MetroTRK Work?
- MetroTRK Architecture Overview
- Where MetroTRK Lives in Memory
- Low-Level Communication with Host Debugger
- MetroTRK Debug API

How Does MetroTRK Work?

MetroTRK is a debug monitor that runs on your embedded system, along with the target program that you are debugging. You as the developer indirectly manipulate MetroTRK by controlling the CodeWarrior debugger, MW Debug, on the host personal computer. MW Debug then engages in two-way communication with MetroTRK to perform the debugging services you have requested.

MetroTRK Architecture Overview

MetroTRK is organized around a core that serves as the central controller of its internal state. Around this core, MetroTRK has several

other modules which perform various tasks. This section discusses the following MetroTRK components:

- MetroTRK Core
- Communications Between MetroTRK and the Host Debugger
- Command and Request Handling
- Message Queues
- Putting it all Together

MetroTRK Core

MetroTRK has two normal operating modes, the *event-waiting* mode and the *request-handling* mode.

When the target program is running, MetroTRK is in *event-waiting* mode. While in this mode, MetroTRK remains inactive, waiting for an exception or interrupt that it's interested in. When one occurs, it stops the target program, and takes back control of the processor.

When MetroTRK has control of the processor, it is in *request-handling* mode. The target program remains stopped while MetroTRK is in request-handling mode. While in request-handling mode, MetroTRK is in a continuous loop, waiting for commands and requests from the host debugger. When it gets a command or request, MetroTRK passes it on to the appropriate handler function. This handling of commands and requests continues until MetroTRK receives a command to switch control back to the target program (a *continue* or *step* command). At this point, MetroTRK returns to its event-waiting mode. For more information on MetroTRK's state control, see "State diagram" on page 14.

We refer to this central state control and event-handler as the *MetroTRK core*. All other modules of MetroTRK are organized around this core.

The MetroTRK core is, generally speaking, independent of the target board configuration. Some of the handler functions which actually perform debugging requests, however, *are* board dependent.

Communications Between MetroTRK and the Host Debugger

MetroTRK is continuously engaged in two way communication with the host debugger (MW Debug.) We divide this communication into two distinct levels, the Transport level and the Debug API level.

The Transport level

The lower level of communication is the Transport level. This level transmits arbitrarily sized segments of data between the target board and the host debugger. These segments are transmitted over a standard serial connection. For more information on the segmented data level, see "Low-Level Communication with Host Debugger" on page 26.

The Debug API level

The higher level communication is the MetroTRK Debug API level. At this level, MetroTRK communicates with the debugger via a clearly defined messaging API. This API defines the various requests and notifications that MetroTRK and the host debugger understand. Each message or function in this API is structured to include relevant formal parameters and, if applicable, a return value. For more details, see "MetroTRK Debug API" on page 26.

Message Queues

When MetroTRK is in request-handling mode, it constantly monitors the serial line for incoming requests and commands. It stores each one in an incoming *message queue*. This message queue makes it possible for MetroTRK to handle requests at its own pace without losing new requests as they come in over the serial line.

Just as the serial line is monitored for *incoming* messages and requests, another queue is maintained for *outgoing* messages. Whenever MetroTRK needs to send a message to the host debugger, it simply puts the message in this queue and can go on processing. The message will be sent as soon as the serial line is free.

The message queues depend in no way on the target board configuration. They will not be discussed in greater detail in this manual.

Command and Request Handling

The handling of requests and commands from the host debugger is separated out from the MetroTRK core into a set of handler functions. These handler functions constitute another module of MetroTRK. Handler functions which are board-dependent are further separated out into another source file. For further details, see "Customizing Debug Services" on page 47.

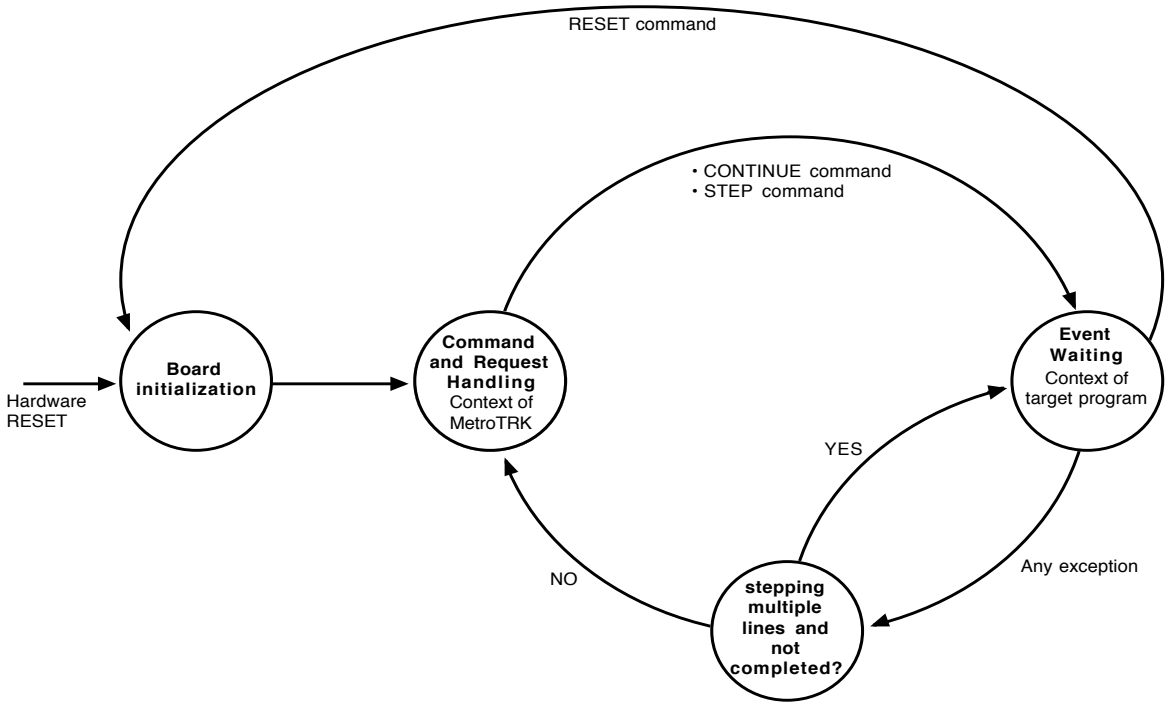
Putting it all Together

So now that we have an overview of how MetroTRK works and its various components, let's take a look at how everything works together. This section gives several diagrams illustrating of MetroTRK's architecture and functioning.

State diagram

The first illustration, Figure 2.1, shows the different states of MetroTRK. On a `Reset` command or a hardware reset, MetroTRK goes into its *Board Initialization* state. After board initializations are completed, MetroTRK goes immediately into its *Command and Request Handling* state. While in this state, MetroTRK continuously services commands and requests from the host debugger. The target program is not running at all while MetroTRK is in the Command and Request Handling state.

Figure 2.1 MetroTRK State Diagram



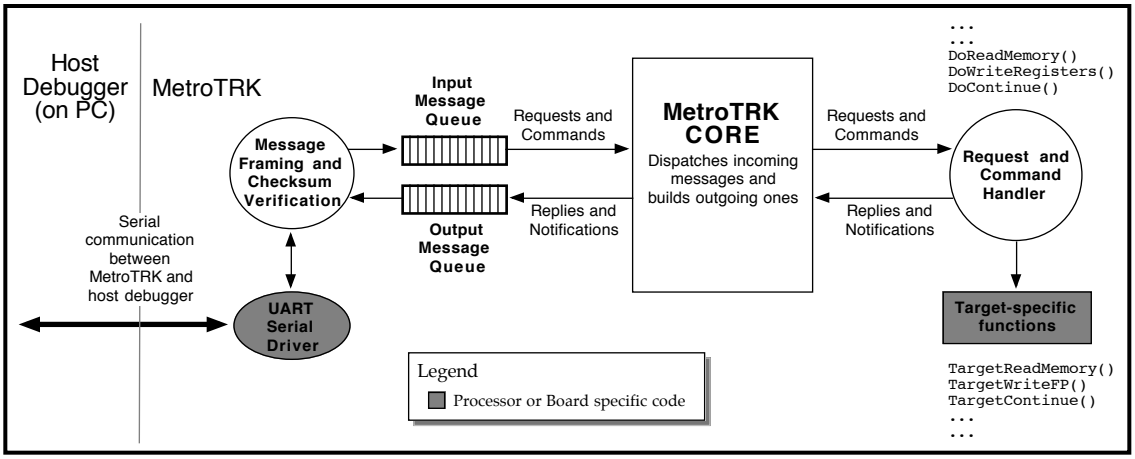
Whenever certain commands are given by the host debugger, namely the `CONTINUE` or `STEP` commands, MetroTRK goes from Command and Request Handling state into its *Event Waiting* state. While in Event Waiting state, MetroTRK is not running at all; the execution context is that of the target program.

The only way that MetroTRK gets out of Event Waiting state is when an exception occurs. In most cases, and exception causes a context switch (back to MetroTRK's context) and a move to the Command and Request Handling state. The only exception to this rule is when MetroTRK is in the middle of processing a multiple-line step command. If this is the case, control goes back to the target program and MetroTRK goes back into Event Waiting state.

Data-flow diagram

The next diagram (Figure 2.2) illustrates the different components of MetroTRK, and how data would flow through MetroTRK when it is in the Command and Request Handling state. When MetroTRK is in the Event Waiting state, all components are basically inactive as MetroTRK waits for the next exception to occur.

Figure 2.2 MetroTRK Data-flow Diagram



Where MetroTRK Lives in Memory

This section explains where the MetroTRK code and data lives in memory, as well as where your target application will live. Information in this section covers the default implementation of MetroTRK included with CodeWarrior. It is possible to customize MetroTRK's memory location to suit your needs by modifying variables in the linker preferences panel in your MetroTRK project. For information on these variables, see your CodeWarrior Targeting manual.

We discuss the following topics:

- MetroTRK's RAM Sections
- Locations of MetroTRK RAM Sections (on MIPS)
- Locations of MetroTRK RAM Sections (on PowerPC)
- Locations of Target Application RAM Sections

- TRK Memory Map (MIPS)
- TRK Memory Map (PowerPC)
- Loading MetroTRK onto Your Hardware

MetroTRK's RAM Sections

The number of different MetroTRK RAM sections depends on whether you are running a RAM-based version of MetroTRK or a ROM-based version.

ROM-based MetroTRK

If you are running MetroTRK from ROM, the following three memory sections will be set up in RAM:

- Data
- Exception Vectors
- The Stack

Data includes all read/write data included in the program. When running from a ROM-based version of MetroTRK, any initial values will be copied from ROM into their locations in RAM. MetroTRK uses 6KB of RAM for global data.

Exception Vectors are sections of code that are executed in the event of a processor exception; their location in RAM is set and determined by the processor. When running from a ROM-based version, MetroTRK will copy exception vectors from ROM into RAM upon initialization. For information on this initialization process, see "MetroTRK Initializations" on page 23.

The Stack uses no more than 6KB of RAM space. It is not recommended to give MetroTRK any less space for its stack.

RAM-based MetroTRK

If you are running MetroTRK from RAM, there will be an additional memory section for the MetroTRK's executable code. This section does not exist in the ROM-based version because the code is execut-

ed directly from ROM memory. The code section of the TRK takes up about 26KB.

Locations of MetroTRK RAM Sections (on MIPS)

In this topic, we discuss the locations of the various MetroTRK memory sections in RAM. In some cases, the default locations can be reconfigured to suit your needs, while in others the locations are fixed and should not be moved. For more information about the purpose of the different memory sections, see "MetroTRK's RAM Sections" on page 17.

Exception vectors (on MIPS)

The location of the exception vectors in RAM is a set characteristic of the processor. On MIPS, the exception vector must start at 0x80000000 (which is actually in low memory), and spans 4096 bytes to end at 0x80001000. In a ROM-based implementation, these vectors are copied from ROM in the MetroTRK initialization process.

The location of the exception vectors should not be changed, as the processor expects to find them at the set location.

Data and Code sections (on MIPS)

In the default implementation, the MetroTRK's data and code sections are placed together in high memory, right above the MetroTRK stack. In a ROM-based implementation of MetroTRK, there will be no code in RAM since it will execute directly from ROM.

For example, on the IDT79S381 board, the code and data sections are by default placed at the address 0x801f8000.

The location of the data and code sections can be set from within the linker preferences panel in the MetroTRK project. The linker preferences panel is available by going to the project's preferences in the CodeWarrior IDE. For information on linker preferences, see the Targeting manual for your target processor.

The stack (on MIPS)

In the default implementation, the MetroTRK's stack is placed in high memory and grows downward. MetroTRK needs no more than 6KB of stack space.

For example, on the IDT79S381 board, the stack is by default placed at the address 0x801f7ff0.

The location of the stack section can be changed by rebuilding the TRK project with new linker preferences. The linker preferences are set from within the linker preferences panel in the CodeWarrior IDE. For information on linker preferences, see the Targeting manual for your target processor.

Locations of MetroTRK RAM Sections (on PowerPC)

In this topic, we discuss the placement of the various MetroTRK memory sections in RAM. In some cases, the default locations can be reconfigured to suit your needs, while in others the locations are fixed and should not be moved. For more information about the purpose of the different memory sections, see "MetroTRK's RAM Sections" on page 17.

Exception vectors (on PowerPC)

The location of the exception vectors in RAM is a set characteristic of the processor. On the PowerPC, the exception vector must start at 0x000100 (which is in low memory), and spans 7936 bytes to end at 0x002000. In a ROM-based implementation, these vectors are copied from ROM in the MetroTRK initialization process.

The location of the exception vectors should not be changed, as the processor expects to find them at the set location.

Data and Code sections (on PowerPC)

In the default implementation, the MetroTRK's data and code sections are placed together in high memory, right above the stack. In a

ROM-based implementation of MetroTRK, there will be no code in RAM since it will execute directly from ROM.

For example, on the MPC821ADS board, the code and data sections are by default placed at the address 0x3f8000. On the MBX821 board, they are placed at 0x.3f0000

The location of the data and code sections can be set from within the linker preferences panel in the MetroTRK project. The linker preferences panel is available by going to the project's preferences in the CodeWarrior IDE. For information on linker preferences, see the Targeting manual for your target processor.

The stack (on PowerPC)

In the default implementation, the MetroTRK's stack is placed in high memory and grows downward. MetroTRK needs no more than 6KB of stack space.

For example, on the MPC821ADS board, the stack is by default placed at the address 0x3f7ff0. On the MBX821 board, it is placed at 0x.3efff0

The location of the stack section can be changed by rebuilding the TRK project with new linker preferences. The linker preferences are set from within the linker preferences panel in the CodeWarrior IDE. For information on linker preferences, see the Targeting manual for your target processor.

Locations of Target Application RAM Sections

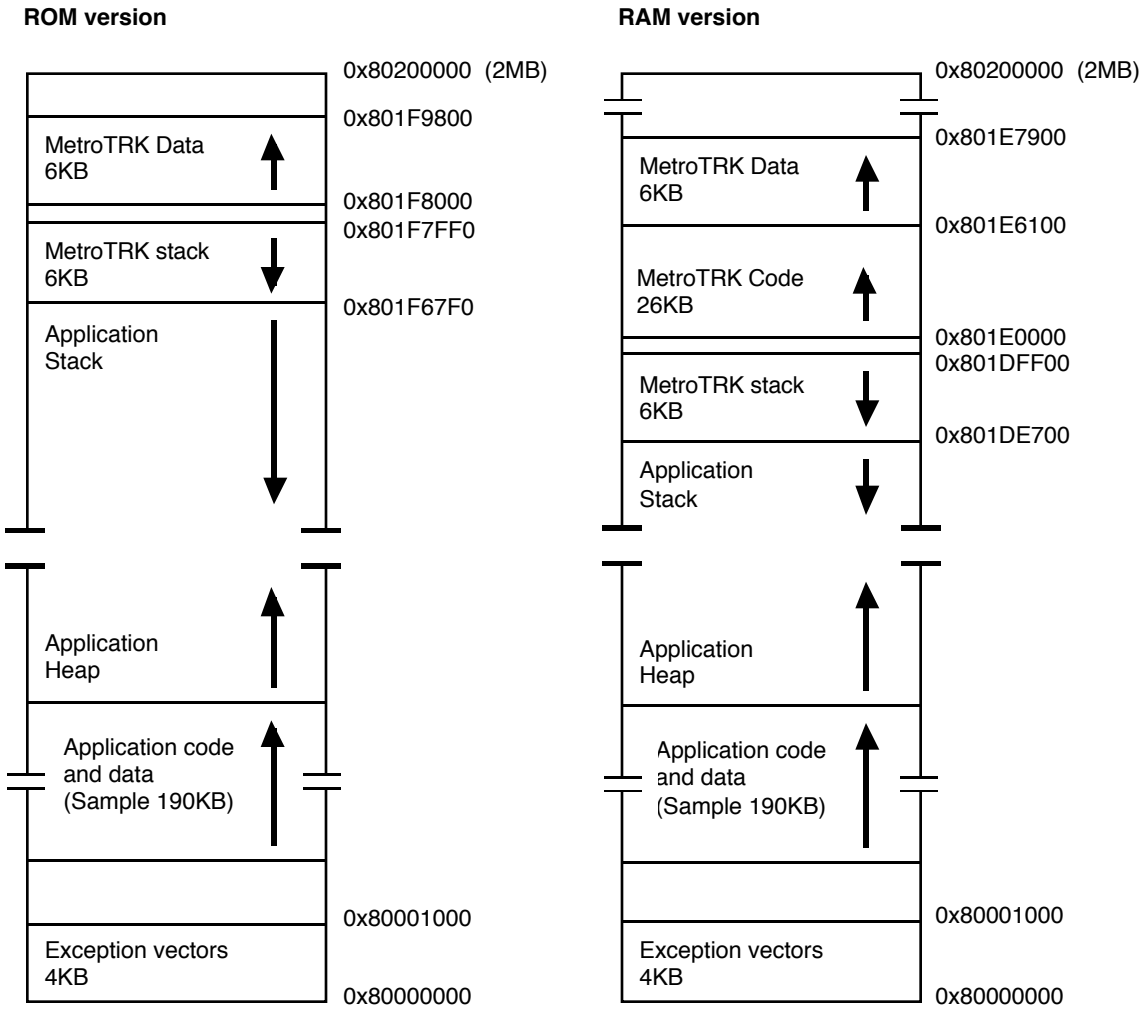
Target application memory sections can go anywhere where they don't interfere with the MetroTRK memory sections. These are described in "MetroTRK's RAM Sections" on page 17. Often, the best is to put the code and data sections below the MetroTRK's code and data sections in low memory, and the target application's stack below the stack of the MetroTRK. Make sure, in doing this, that you leave enough room for the MetroTRK's stack, since the stack grows down.

You can change the placements of your program's memory sections by changing its linker preferences in the CodeWarrior IDE. For information on linker preferences, see the Targeting manual for your target processor.

TRK Memory Map (MIPS)

Figure 2.3 shows a sample map of RAM memory sections as configured when running MetroTRK along with a sample target program on a IDT MIPS 79S381 board.

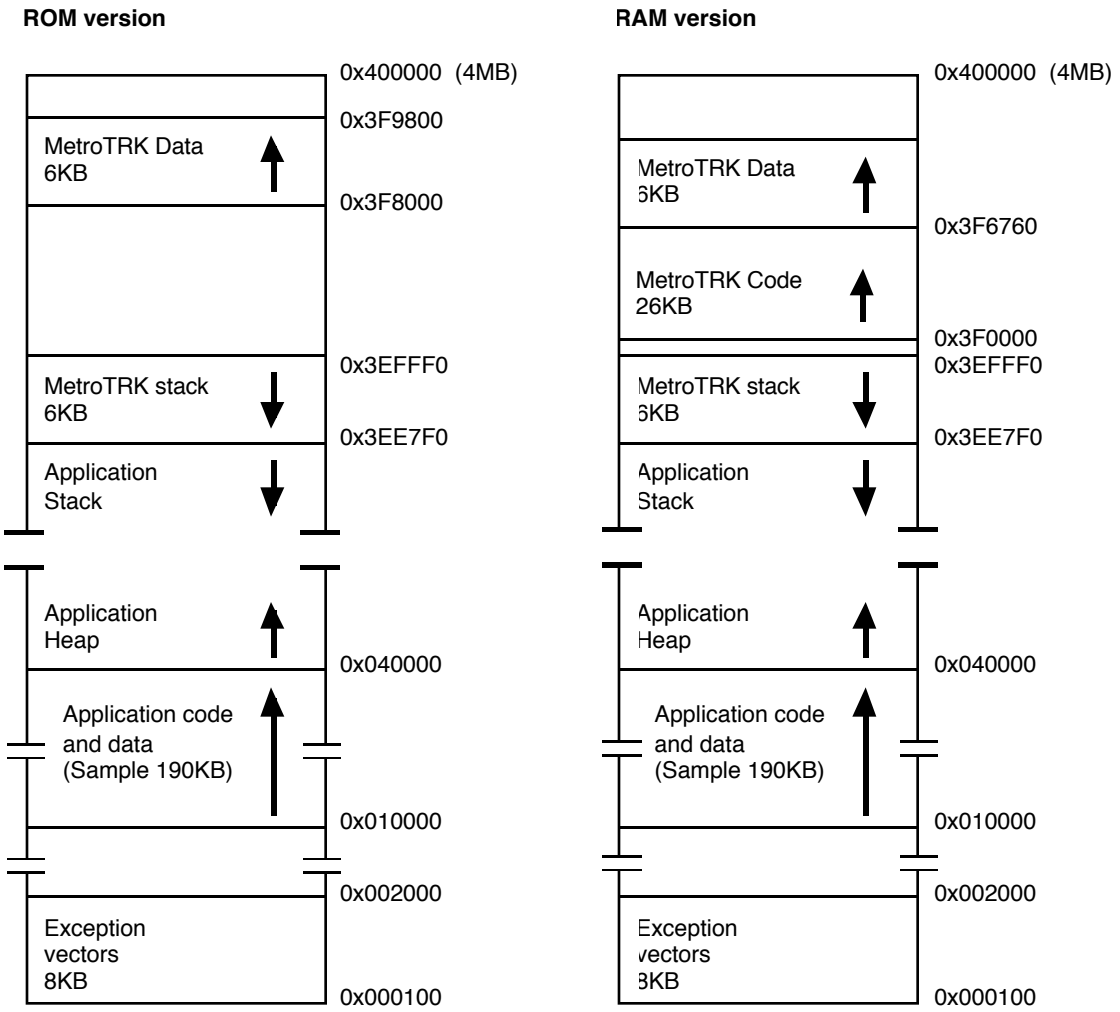
Figure 2.3 TRK RAM Map (MIPS)



TRK Memory Map (PowerPC)

Figure 2.4 shows a sample map of RAM memory sections as configured when running MetroTRK along with a sample target program on an MPC821 MBX board.

Figure 2.4 TRK RAM Map (PowerPC)



Loading MetroTRK onto Your Hardware

For specific information on how to load MetroTRK onto your target hardware, see the Targeting guide for your target processor.

MetroTRK Initializations

Initializations in MetroTRK follow a specific step-by-step sequence that is built into the MetroTRK core. Several of these steps are specific *only* to the processor being targeted, while others may be specific to other aspects of the board hardware being targeted.

Initialization occurs in the following order:

1. Initialization of the target processor's registers
2. Board Initializations that don't Access MetroTRK Memory
3. Exception Vector Initializations
4. Initialization of all TRK RAM sections
5. Additional initializations

Initialization of the target processor's registers

This step initializes the processor's standard EABI (Embedded Application Binary Interface) registers for use. Among those initialized are the stack and small data registers. This step does not touch memory nor is it dependent on any hardware configuration other than the type of processor being used. This step occurs within the `__init_registers()` function which, in turn, is called by `__start()`.

For a full specification of functions listed here, see the "MetroTRK Function Reference Overview" on page 55.

Board Initializations that don't Access MetroTRK Memory

This step initializes board hardware which does *not* rely on MetroTRK's memory sections being initialized. This can include a whole range of items, such as cache initializations, clock initializations, or other memory system initializations. These initializations occur within the `__init_hardware()` function which gets called by `__start()`.

This function is intended *specifically* for hardware initializations that are going to differ between one board configuration and another.

For this reason, you are going to want to pay special attention to this function when customizing MetroTRK.

For a full specification of functions listed here, see the "MetroTRK Function Reference Overview" on page 55.

Exception Vector Initializations

This step initializes system exception vectors by copying them to the appropriate place in system memory (RAM). This location is specific to the target processor. MetroTRK's default exception handling strategy is to act on certain exceptions that it is interested in, and notify the host debugger of all other exceptions.

If you want your application to handle certain exceptions, you must take care that you are not interfering with MetroTRK operation.

The PowerPC version of MetroTRK needs to receive all `Program Error` and `Trace` exceptions. Your application should not overwrite either of these exceptions. Other than that, your target program may overwrite whichever exceptions it needs to handle.

WARNING! On PowerPC, you must make sure that MetroTRK's exception-handling code gets called for the exception "`Software Emulation`." This exception is used to track breakpoints, and breakpoints will not work properly if the exception is not handled by MetroTRK.

The MIPS version of MetroTRK needs to receive all `Breakpoint` exceptions. If your application needs to handle either of these exceptions, it needs to call into MetroTRK's handling code at some point (you need to *share* the exception with MetroTRK.) Other than that, your target program may overwrite whichever exceptions it needs to handle. See "MetroTRK's RAM Sections" on page 17 for information on where MetroTRK's exception vectors are located.

Since the exception vectors locations in memory are specified by the target processor, this vector-copying process is not board-depen-

dent. Therefore, you should not need to be modify it for differing board configurations.

The exception vector copy takes place within the `__copy_vectors()` function called from within `__init_hardware()`. `__copy_vectors()` must be called in order for MetroTRK to function properly.

For a full specification of functions listed here, see the "MetroTRK Function Reference Overview" on page 55.

Initialization of all TRK RAM sections

This step basically makes sure that all the RAM used by MetroTRK is well initialized. This may mean different things for different processors, but generally includes copying some initial data values from ROM into RAM (in ROM-based TRK implementations), and making sure all uninitialized sections are cleared with zeros.

All such memory initializations take place within the `__init_data()` function which is called by `__start()`. If you need to modify the locations of RAM sections, you should not need to modify anything in `__init_data()`. Instead, you can change the location from within the "Linker Preferences" panel in your project. See the Targeting manual for your target processor for details on setting linker preferences.

For a full specification of functions listed here, see the "MetroTRK Function Reference Overview" on page 55.

Additional initializations

This step provides an opportunity to perform special hardware or software initializations not fitting into one of the previous steps. You perform these additional initializations in the `__init_user()` function. This function is called by `__start()`, and it is the last code called before MetroTRK goes into its core event-waiting loop. At the time this function is called, all other initializations have already been performed. You can rely on the assumption that all processor, memory, and other hardware initializations have been completed.

You may use `__init_user()` to customize MetroTRK for different hardware configurations, if there are certain initializations that rely on other initializations being complete.

For a full specification of functions listed here, see the "MetroTRK Function Reference Overview" on page 55.

Low-Level Communication with Host Debugger

At the lowest levels, communication between MetroTRK and the host debugger takes place over a standard serial connection. For maximum portability, the low-level code which drives the actual serial controller is completely factored away from the MetroTRK core. MetroTRK provides a simple interface which can be plugged into different UART (Universal Asynchronous Receiver Transmitter) drivers if necessary. The default implementations plug into the standard on-board serial ports.

The transmission rate (or baud-rate) of this serial communication is configurable depending on the target board's capabilities. Possible transmission rates range from 300 baud to 230.4k baud. On the MetroTRK side, transmission rate is set by changing a compile-time constant. For more information, see "Customizing Low-Level Communications" on page 46. On the host debugger side, it is set by changing a field in the preferences dialog. For more information on setting host debugger settings, see the Targeting manual for your target processor.

On top of this serial communication, a simple messaging protocol is used where each discrete message is framed verified by employing a checksum. This messaging protocol allows MetroTRK and the host debugger to send each other messages as defined by the MetroTRK debug API, as described in "MetroTRK Debug API" on page 26.

MetroTRK Debug API

Abstracting away from the actual serial communication between MetroTRK and the host debugger, the two sides communicate via a

well-defined debug API. This API defines a set of messages, each of which is either sent from the host debugger to MetroTRK, or vice-versa.

Some of the messages in the debug API return a value via a *reply message*. Since they have a return value, we refer to these messages as *functional* messages, or simply *functions*. Messages which do *not* return with a reply message are referred to as *commands*. In reality, all messages return a reply message, but a reply message of a simple command contains nothing more than an acknowledgment and an error code. Replies to functional messages contain one or more values in addition to the standard acknowledgment and error code.

Debug commands and functions are grouped into two categories, the *primary command set* (level 1) and the *extended command set* (level 2). The primary command set is absolutely necessary for the debugger to function properly. The extended command set, in contrast, adds functionality to the debugger but is not necessary.

MetroTRK uses commands and functions in two fundamental ways. The first is to receive and service requests from the host debugger. The second is to notify the host debugger that a special event has occurred on the target. We will discuss the specific commands and functions serving these two purposes in the following sections:

- Servicing Requests from the Host Debugger
- Sending Notifications to the Host Debugger

Servicing Requests from the Host Debugger

Most of the functions and commands in the MetroTRK debug API are *requests* from the host debugger to MetroTRK. These can take the form of requests for information (like asking for the value of a register) or requests for action (like asking MetroTRK to step across a line of code.)

When the MetroTRK core receives a request, it sends the request to the corresponding handler function for processing. In the request specifications that follow, the request's handler function is specified in the "Handler Function" field.

The following is a list of each of the requests the host debugger can send. Detailed specifications for each request can be found in the section "Host Debugger to MetroTRK Requests" on page 28, or on the page number listed below:

Sending Notifications to the Host Debugger

In addition to receiving requests, MetroTRK can also *send* messages, called *notifications*, to alert the host debugger of important events.

The following is a list of each of the notifications that MetroTRK can send. Detailed specifications for each notification can be found in the section "MetroTRK to Host Debugger Notifications" on page 40, or on the page number listed below:

Host Debugger to MetroTRK Requests

This section lists the requests that the host debugger can send to MetroTRK. Each listing gives the following attributes:

- Command Set: Either "Primary Command Set" or "Extended Command Set". See "MetroTRK Debug API" on page 26 for more information.
- Description: A high-level description of the request
- Parameters: An explanation of each formal parameter, if any
- Handler function: The name of the MetroTRK function which handles the request
- Return: An explanation of the value returned by the request, if any (acknowledgments and error codes are omitted, since they are always returned.)
- Remarks: Implementational or other notes about the request
- See Also: References to related information

NOTE: Each of these messages is also outlined in the source file `msgcmd.h`. This file also defines all `MessageCommandID` values and message-specific constants.

The following requests are described in this section:

- Connect
- Reset
- GetVersions
- SupportMask
- ReadMemory
- WriteMemory
- ReadRegisters
- WriteRegisters
- Continue
- Step

Connect

Command Set Primary Command Set (level 1)

Description Begins debug session

Parameters None

Handler Function `DoConnect()`

Return None

Remarks This request should be sent by the host debugger once at the beginning of each debug session.

See Also "DoConnect()" on page 61

Reset

Command Set Extended Command Set (level 2)

Description Tells MetroTRK to reset the target board

Parameters None

Handler Function `DoReset()`

Return None

Remarks Restarts MetroTRK and performs all hardware initializations, just as if the board were being manually reset.

See Also "DoReset()" on page 66

GetVersions

Command Set Primary Command Set (level 1)

Description Returns MetroTRK version information

Parameters None

Handler Function `DoVersions()`

Return Return values for this function are:

`kernelMajor`	`ui8`	The "major change" part of the MetroTRK version number (in version 1.2, the kernelMajor would be 1.)
`kernelMinor`	`ui8`	The "minor change" part of the MetroTRK version number (in version 1.2, the kernelMinor would be 2.)
`protocolMajor`	`ui8`	The "major change" part of the messaging protocol version number (in version 1.2, the protocolMajor would be 1.)
`protocolMinor`	`ui8`	The "major change" part of the messaging protocol version number (in version 1.2, the protocolMinor would be 2.)

See Also "DoVersions()" on page 69

SupportMask

Command Set Primary Command Set (level 1)

Description Returns a list of which messages are supported by the MetroTRK

Parameters None

Handler Function `DoSupportMask()`

Return Return values for this function are:

`mask`	`ui8[32]`	A bit-array of 32 bytes, where each bit corresponds to the message (type `MessageCommandID`) with an ID matching the position of the bit in the array. If the bit value is 1, it signifies that the message is available. If the value is 0, it signifies that the message is *not* available. As an example, if `kDSReset` were available, then the 4th bit of mask would be 1 since `kDSReset` is the 4th message (its value is actually 3, but we start counting from 0.) See the documentation in `msgcmd.h` for details. Also see how the default values are set in `target.h`

Remarks None

See Also "DoSupportMask()" on page 68,

`msgcmd.h`

ReadMemory

Command Set Primary Command Set (level 1)

Description Reads an arbitrary section of memory off target board

Parameters

If the `options` parameter is `DS_MSG_MEMORY_EXTENDED`, the parameters are as follows

`options`	`ui8`	One of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_EXTENDED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` (See `msgcmd.h` for details on these)
`length`	`ui16`	Length of memory section, in bytes (1 to 2048)
`start High`	`ui32`	Start address of memory section (upper word)
`startLow`	`ui32`	Start address of memory section (lower word)

If the `options` parameter is anything else, the parameters are:

`options`	`ui8`	One of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_EXTENDED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` (See `msgcmd.h` for details on these)
`length`	`ui16`	Length of memory section, in bytes (1 to 2048)
`start`	`ui32`	Start address of memory section

Handler Function

`DoReadMemory()`

Return

Return values for this function are:

`length`	`ui16`	Length of data read
`data`	`ui8[]`	Data (up to 2048 bytes)

Remarks MetroTRK will attempt to catch and handle any memory access exceptions that occur while reading the data.

See Also "DoReadMemory()" on page 64,

`msgcmd.h`

WriteMemory

Command Set Primary Command Set (level 1)

Description Writes data to an arbitrary position in memory

Parameters If the `options` parameter is `DS_MSG_MEMORY_EXTENDED`, the parameters are as follows

`options`	`ui8`	One of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_EXTENDED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` (See `msgcmd.h` for details on these)
`length`	`ui16`	Length of data, in bytes (1 to 2048)
`start High`	`ui32`	Start address of destination (upper word)
`startLow`	`ui32`	Start address of destination (lower word)
`data`	`ui8[]`	Data (up to 2048 bytes)

If the `options` parameter is anything else, the parameters are:

`options`	`ui8`	One of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_EXTENDED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` (See `msgcmd.h` for details on these)
`length`	`ui16`	Length of data, in bytes (1 to 2048)
`start`	`ui32`	Start address of destination
`data`	`ui8[]`	Data (up to 2048 bytes)

Handler Function `DoWriteMemory()`

Return Return values for this function are:

`length`	`ui16`	Amount of memory actually written

Remarks MetroTRK will attempt to catch and handle any memory access exceptions that occur while writing the data.

See Also "DoWriteMemory()" on page 70,

`msgcmd.h`

ReadRegisters

Command Set Primary Command Set (level 1)

Description Reads a sequence of registers from the target processor

Parameters Parameters are as follows:

`options`	`ui8`	One of the following value: • `kDSRegistersDefault` • `kDSRegistersFP` • `kDSRegistersExtended1` • `kDSRegistersExtended2` (See `msgcmd.h` for details)
`first Register`	`ui16`	Number of the first register in the sequence
`last Register`	`ui16`	Number of the last register in the sequence

Handler Function `DoReadRegisters()`

Return Return values for this function are:

`register Data`	`void*`	An array of register values. The size of each element will depend on the size of the registers themselves. If the registers are 2 bytes wide, then a new value will start every 2 bytes. If the registers are 4 bytes wide, a new value will start every 4 bytes. The maximum length of this array is 2048 bytes.

Remarks MetroTRK will attempt to catch and handle any access exceptions that occur while reading.

See Also "DoReadRegisters()" on page 65,

PowerPC `m8xxreg.h`,

MIPS `mips.reg.h`,

`msgcmd.h`

WriteRegisters

Command Set Primary Command Set (level 1)

Description Writes data to a sequence of registers

Parameters Parameters are as follows:

`options`	`ui8`	One of the following value: • `kDSRegistersDefault` • `kDSRegistersFP` • `kDSRegistersExtended1` • `kDSRegistersExtended2` (See `msgcmd.h` for details)
`first Register`	`ui16`	Number of the first register in the sequence
`last Register`	`ui16`	Number of the last register in the sequence
`register Data`	`ui32[]`	An array of register values. The size of each element will depend on the size of the registers themselves. If the registers are 2 bytes wide, then a new value will start every 2 bytes. If the registers are 4 bytes wide, a new value will start every 4 bytes. The maximum length of this array is 2048 bytes.

Handler Function `DoWriteRegisters()`

Return None

Remarks MetroTRK will attempt to catch and handle any access exceptions that occur while writing.

See Also "DoWriteRegisters()" on page 71,

PowerPC `m8xxreg.h`,

MIPS `mips.reg.h`,

`msgcmd.h`

Continue

Command Set Primary Command Set (level 1)

Description Starts target program running

Parameters None

Handler Function `DoContinue()`

Return None

Remarks This request is sent by the host debugger to tell MetroTRK to let the target program continue execution. This will put MetroTRK back into its *event-waiting* mode, where it will let the program run until something interesting happens. For more information on the MetroTRK's event-waiting mode, see "MetroTRK Core" on page 12.

See Also "DoContinue()" on page 62

Step

Command Set Extended Command Set (level 2)

Description Request for MetroTRK to let the target program run an arbitrary number of instructions or, alternatively, until the PC is outside a given range of values.

Parameters

If the `options` parameter is `kDSStepSingle`, the parameters are as follows

`options`	`ui8`	One of the following values: • `kDSStepSingle` • `kDSStepOutOfRange` (See `msgcmd.h` for details on these)
`count`	`ui8`	Number of instructions to step over

If the `options` parameter is `kDSStepOutOfRange`, the parameters are:

`options`	`ui8`	One of the following values: • `kDSStepSingle` • `kDSStepOutOfRange` (See `msgcmd.h` for details on these)
`range-Start`	`ui32`	Start address of memory range
`rangeEnd`	`ui32`	End address of memory range

Handler Function

`DoStep()`

Return

None

Remarks

if the `options` parameter is `kDSStepSingle`, MetroTRK will step over `count` instructions in the target program, and then return control to the host. If the `options` parameter is `kDSStepOutOfRange`, MetroTRK will continue running the program until it hits an instruction whose address is outside the range specified by `range-Start` and `rangeEnd`. It will then return control to the host.

MetroTRK alerts the host that the end condition has been reached by sending a `NotifyStopped` notification. For more information on this notifications, see "NotifyStopped" on page 40.

See Also "DoStep()" on page 67,

"NotifyStopped" on page 40,

`msgcmd.h`

MetroTRK to Host Debugger Notifications

This section lists the notifications that MetroTRK can send to the host debugger. Each listing gives the following attributes:

- Command Set: Either "Primary Command Set" or "Extended Command Set". See "MetroTRK Debug API" on page 26 for more information.
- Description: A high-level description of the notification
- Parameters: An explanation of each formal parameter, if any
- Return: An explanation of the value returned by the notification, if any (acknowledgments and error codes are omitted, since they are always returned.)
- Remarks: Implementational or other notes about the notification
- See Also: References to related information

The following notifications are described in this section:

NotifyStopped

Command Set Primary Command Set (level 1)

Description Used to notify debugger that a breakpoint has been hit or a step command has completed.

Parameters The parameters are as follows

`target defined info`	`target specific`	This variable basically gives state information about the target. It is different for each target processor. It generally contains information like the PC and the instruction at the PC. See `TargetAddStopInfo()` for details.

Return None

Remarks None

See Also "TargetInterrupt()" on page 80,

"TargetAddStopInfo()" on page 78

NotifyException

Command Set Primary Command Set (level 1)

Description Used to notify debugger that an exception has occurred on the target processor.

Parameters If the `options` parameter is `kDSStepSingle`, the parameters are as follows

`target defined info`	`target specific`	This variable basically gives state information about the target. It may be different for each target processor, but generally contains information like the PC, the instruction at the PC, and the exception ID. See `TargetAddExceptionInfo()` for details.

Return None

Remarks None

See Also **PowerPC** `m8xxreg.h`,

MIPS `mips.reg.h`,

"TargetInterrupt()" on page 80,

"TargetAddExceptionInfo()" on page 78

Fputs

Command Set Primary Command Set (level 1)

Description Sends a string to the host debugger to display in a console window.

Parameters The parameters are as follows:

`file Options`	`ui8`	One of the following values: • `kDSStdout` • `kDSStderr` (See `msgcmd.h` for details on these)
`string Data`	`ui8[]`	A zero-terminated string to be sent to the host debugger.

Return None

Remarks None.

See Also "DoFputs()" on page 63,

`msgcmd.h`

3

Customizing MetroTRK

This chapter describes how you can customize MetroTRK to work with new board configurations.

Customizing MetroTRK Overview

Certain parts of MetroTRK rely on specific details about the board configuration you are targeting. Each CodeWarrior embedded product comes with at least one complete implementation of MetroTRK which targets a specific reference board. If you are using this supported reference board or boards, you don't have to modify MetroTRK. If you are using a board *other* than the supported reference board(s), you may need to modify some parts of MetroTRK to customize it for your specific board configuration. MetroTRK code is factored with this in mind, so there are relatively few functions that are board-dependent and need customization.

NOTE: For information on supported reference boards and MetroTRK implementations for each, see the Targeting manual for your CodeWarrior product.

This chapter guides you through the parts of MetroTRK which may need customization. Each section explains customizations in a specific area of MetroTRK. The sections in this chapter are:

- Customizing MetroTRK Initializations
- Customizing Low-Level Communications
- Customizing Debug Services

Customizing MetroTRK Initializations

As discussed in "MetroTRK Initializations" on page 23, there are five steps to the initialization process. Out of these, two are not dependent on board-level specifics, and you should never need to modify them to customize for your board. They are:

- Initialization of the target processor's registers
- Initialization of all MetroTRK RAM sections

The remaining three steps may need to be customized and are discussed in the topics which follow:

- Initialization of hardware which doesn't access MetroTRK memory
- Exception vector initializations
- Additional initializations

Customizing Hardware Initializations

You can customize MetroTRK hardware initializations in one of two places. Initializations which do *not* depend on MetroTRK's RAM sections being set up should occur within the `__init_hardware()` function. Those that may need to access part of MetroTRK's RAM should be within the `__init_user()` function.

Customizing __init_hardware()

`__init_hardware()` is called immediately after initializing the processor's register set but before any of MetroTRK's RAM sections are initialized. For this reason, memory access is not safe at the start of `__init_hardware()`. In fact, one of the responsibilities of `__init_hardware()` is to make memory access safe. By the end of the function, all memory sub-systems should be initialized.

The `__init_hardware()` function also performs another step of MetroTRK initialization sequence. It calls into the `__copy_vectors()` function, which copies the exception vectors from ROM into the appropriate RAM locations for exception handling. Since `__copy_vectors()` must be called *after* memory ac-

cess has been made safe (because it must be able to write into system RAM), you must perform all memory initializations *before* the call to `__copy_vectors()`. See the topic on "Customizing Exception Vector Initializations" on page 45 for details about the `__copy_vectors()` function.

For more explanation of the `__init_hardware()` function, see "Board Initializations that don't Access MetroTRK Memory" on page 23, and its entry in the function reference, "__init_hardware()" on page 58.

Customizing __init_user()

If you need to perform a hardware initialization that depends on MetroTRK memory being initialized, use the `__init_user()` function. This function is basically the last thing to be executed in the MetroTRK initialization process.

WARNING! Remember that you can not perform initializations which access MetroTRK memory from the `__init_hardware()` function, because MetroTRK memory is not yet set up at that point. You should put such initializations in `__init_user()`.

For more explanation of the __init_user() function, see "Additional initializations" on page 25, and its entry in the function reference, "__init_user()" on page 59.

Customizing Exception Vector Initializations

You should not need to customize exception vector initialization. However, you may want to override the default exception-handling code to handle certain exceptions. If you want your application to handle certain exceptions, you must take care that you are not interfering with MetroTRK operation.

The PowerPC version of MetroTRK needs to receive all `Program Error` and `Trace` exceptions. Your application should not overwrite either of these exceptions. Other than that, your target program may overwrite whichever exceptions it needs to handle.

WARNING! On PowerPC, you must make sure that MetroTRK's exception-handling code gets called for the exception "`Software Emulation`." This exception is used to track breakpoints, and breakpoints will not work properly if the exception is not handled by MetroTRK.

The MIPS version of MetroTRK needs to receive all `Breakpoint` exceptions. If your application needs to handle either of these exceptions, it needs to call into MetroTRK's handling code at some point (you need to *share* the exception with MetroTRK.) Other than that, your target program may overwrite whichever exceptions it needs to handle. See "MetroTRK's RAM Sections" on page 17 for information on where MetroTRK's exception vectors are located.

For more information on the `__copy_vectors()` function, see "Exception Vector Initializations" on page 24, and its entry in the function reference, "__copy_vectors()" on page 56.

Customizing Additional Initializations

You can perform any additional initializations within the `__init_user()` function. This function is called after everything else in MetroTRK has been initialized, immediately before starting the actual program to be debugged.

For more explanation of the __init_user() function, see "Additional initializations" on page 25, and its entry in the function reference, "__init_user()" on page 59.

Customizing Low-Level Communications

Low-level communications between MetroTRK and the host debugger take place over a standard serial connection. The data transmission rate of this serial connection is set for MetroTRK at compile-time and for the host-debugger at debug time.

To make customization easy, the implementation of this serial messaging is completely factored from the rest of MetroTRK. The source

file `UART.h` declares a set of nine abstract functions which MetroTRK uses to send and receive serial messages. These functions are abstracted away from the main MetroTRK code so that MetroTRK can function with new serial drivers easily. To support any new serial hardware, these nine functions need to be re-implemented to properly control the hardware.

Of the nine UART functions, only five need to be re-implemented because the other four are derived from the first five. They can all be found in the function reference at the locations listed below:

- "InitializeUART()" on page 72
- "TerminateUART()" on page 94
- "ReadUARTPoll()" on page 73
- "ReadUART1()" on page 73
- "WriteUART1()" on page 96

Data Transmission Rate

MetroTRK can communicate with the host debugger at transmission rates between 300 baud and 230.4k baud. The transmission rate is set at compile time by setting the constant `TRK_BAUD_RATE` to a value of the enumerated type `UARTBaudRate` (`UARTBaudRate` is defined in `UART.h`.) In the default implementation, `TRK_BAUD_RATE` is defined in the file `target.h`.

In order to work properly, the host debugger must be set to communicate at the same transmission rate. This attribute can be set in the preference dialog box in MW Debug. For more information on debugger settings, see the Targeting manual for your target processor.

Customizing Debug Services

Debug services are provided by MetroTRK via a messaging API outlined in "MetroTRK Debug API" on page 26. Several of these messages may need to be customized for different board configurations. This section goes through each message in the API, discussing whether or not it is dependent on the configuration of the target

board. If it *is* board-dependent, it describes how to go about customizing it for your board.

Parts of code that are board-specific are generally factored out of the main message handling code so that they may be modified easily without having to maneuver through large sections of code which are not board-specific.

In this section, the following topics are covered:

Customizing Debug Request Handling

This section discusses customization of each of the debugger request messages.

Customizing Connect

Upon receiving a `Connect` request, MetroTRK simply sends a debug message to the console. It is not board-specific in any way and should not need any customization.

For more information on the `Connect` request, see "Connect" on page 29.

Customizing Reset

Upon receiving a `Reset` request, MetroTRK calls into its own reset code. This is not board-specific in any way and should not need any customization.

For more information on the `Reset` request, see "Reset" on page 30.

Customizing GetVersions

Upon receiving a `GetVersions` request, MetroTRK looks up version numbers and returns them to the host debugger. This function is not board-specific in any way and should not need any customization.

For more information on the `GetVersions` request, see "GetVersions" on page 30.

Customizing SupportMask

Upon receiving a `SupportMask` request, MetroTRK looks up which debug API messages are supported by calling the `TargetSupportMask()` function. This function itself is not board-specific and should not need to be changed, but it does rely on a set of board-specific variables that you may need to change in customizing MetroTRK. These compile-time variables, defined in the file `target.h`, specify exactly which debug API messages are supported by your custom version of MetroTRK.

Each variable is 8 bits wide, and there are 32 such variables. Each is a bit-vector where each bit represents one message in the MetroTRK debug API. The first variable, `DS_SUPPORT_MASK_00_07`, represents the first 8 messages, those with numbers 0x00 through 0x7. The second variable, `DS_SUPPORT_MASK_08_0F`, represents the next 8 messages and so on until you get to `DS_SUPPORT_MASK_F8_FF`, which represents messages 248 through 255.

WARNING! In the original implementation of MetroTRK, the support mask variables were mislabled. Each that should have been labeled `DS_SUPPORT_MASK_?0_?7` was mislabled `DS_SUPPORT_MASK_?0_?8`, and each that should have been labeled `DS_SUPPORT_MASK_?8_?F` was mislabled `DS_SUPPORT_MASK_?9_?F`. If you are using an older version of MetroTRK, be careful not to interpret values by these variables' incorrect naming.

By changing the values of these variables, you can remove support for any messages that you haven't implemented in your version of MetroTRK. Likewise, if you add support for any optional messages not supported by the default MetroTRK, you need to change these variables accordingly.

For more information, see "TargetSupportMask()" on page 87 and for more information on the `SupportMask` request, see "SupportMask" on page 31.

Customizing ReadMemory

Upon receiving a `ReadMemory` request, MetroTRK reads the specified section of memory and returns the result. To carry out this task, MetroTRK calls the function `TargetReadMemory()` to read memory from the board. This function itself is not board-specific, but it calls into another function that is: `ValidMemory32()`.

`ValidMemory32()` is the only part of the `ReadMemory` message which is board-specific. The job of this function is to check whether the addresses to be read are valid on the target board configuration.

On the MIPS version of MetroTRK, `ValidMemory32()` is not itself board-specific. It instead relies on a global, `gMemMap`, to make inferences about which ranges are valid and which are invalid. All you should need to do to customize memory checks is to redefine `gMemMap` in the file `memmap.h`.

On the PowerPC version of MetroTRK, ValidMemory32() is a board-specific function, in other words it relies on the specific memory layout of the board you are using. You will need to customize this function to properly represent the memory configuration of your board.

For more information on `TargetReadMemory()`, see "TargetReadMemory()" on page 85. For more information on `ValidMemory32()`, see "ValidMemory32()" on page 95. For more information on the `ReadMemory` request, see "ReadMemory" on page 32.

Customizing WriteMemory

Upon receiving a `WriteMemory` request, MetroTRK writes the specified data into memory at the specified address. To carry out this task, MetroTRK calls the function `TargetWriteMemory()` to write to memory on the board. This function itself is not board-specific, but it calls into another function that is, `ValidMemory32()`.

`ValidMemory32()` is the only part of the `WriteMemory` message which is board-specific. The job of this function is to check whether the addresses to be written to are valid on the target board configuration.

On the MIPS version of MetroTRK, `ValidMemory32()` is not board-specific. It instead relies on a global, `gMemMap`, to make inferences about which ranges are valid and which are invalid. All you should need to do to customize memory checks is to redefine `gMemMap` in the file `memmap.h`.

On the PowerPC version of MetroTRK, ValidMemory32() is a board-specific function, in other words it relies on the specific memory layout of the board you are using. You will need to customize this function to properly represent the memory configuration of your board.

For more information on `TargetWriteMemory()`, see "TargetWriteMemory()" on page 93. For more information on `ValidMemory32()`, see "ValidMemory32()" on page 95. For more information on the `WriteMemory` request, see "WriteMemory" on page 34.

Customizing ReadRegisters

Upon receiving a `ReadRegisters` request, MetroTRK reads the specified sequence of registers from the processor, returning the resulting values to the host debugger. Reading registers is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `ReadRegisters` request, see "ReadRegisters" on page 35.

Customizing WriteRegisters

Upon receiving a `WriteRegisters` request, MetroTRK writes the specified data into the specified register sequence. Writing to registers is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `WriteRegisters` request, see "WriteRegisters" on page 37.

Customizing Continue

Upon receiving a `Continue` request, MetroTRK swaps in the context of the target program and begins it executing. This is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `Continue` request, see "Continue" on page 38.

Customizing Step

Upon receiving a `Step` request, MetroTRK steps through a particular number of instructions. This is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `Step` request, see "Step" on page 38.

Customizing Debug Notifications

This section discusses customization of each of the MetroTRK notification messages.

Customizing NotifyStopped

MetroTRK notifies the host debugger that the target program has been stopped by sending a `NotifyStopped` notification. Building this notification is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `NotifyStopped` request, see "NotifyStopped" on page 40.

Customizing NotifyException

MetroTRK notifies the host debugger that an exception has occurred on the target processor by sending a `NotifyException` notifica-

tion. Building this notification is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `NotifyException` request, see "NotifyException" on page 41.

Customizing Fputs

MetroTRK outputs a string on the host debugger side by sending a `Fputs` notification. Building this notification is not a board-specific task, so you should not need to customize this function for new board configurations.

For more information on the `Fputs` request, see "Fputs" on page 42.

A

MetroTRK Function Reference

This is a reference for all MetroTRK functions mentioned in the text of this manual.

MetroTRK Function Reference Overview

This appendix discusses every function appearing in the text of this manual, along with other functions you might encounter in the MetroTRK source code. The discussion of each function includes the following attributes:

- Description: A high-level description of the function
- Source File: The name of the file in which the function appears
- Prototype: The entire C prototype for the function
- Parameters: An explanation of each formal parameter, if any
- Return: An explanation of the value returned by the function, if any
- Remarks: Implementational or other notes about the function
- Board-specific: Specifies whether or not the function relies on specifics about the board configuration. If this attribute is "yes", the function may need modifications for new board configurations.
- See Also: References to related functions

The functions described in this chapter are:

__copy_vectors()

Description Copies exception vectors into the appropriate spot in RAM.

Source File **PowerPC** `__ppc_eabi_init.c`
MIPS `__mips_eabi_init.c`

Prototype
```
void __copy_vectors ( void );
```

Parameters None.

Return None.

Remarks This function initializes the processor exception vectors in their correct locations in RAM. This location is not configurable since it is specified by the target processor. Vectors are copied from MetroTRK's ROM or RAM data section.

`__copy_vectors()` depends on safe memory access, so the memory system must be initialized before calling `__copy_vectors()`.

This function is called only once, by the `__init_hardware()` function.

Board-specific No.

See Also "__init_hardware()" on page 58

__init_data()

Description Initializes MetroTRK data sections.

Source File `__start.c`

Prototype
```
static void __init_data ( void );
```

Parameters None.

Return None.

Remarks This function performs two kinds of initializations. First, it zero-initializes all data sections. Second, in the case where MetroTRK is run-

ning from ROM memory, it copies all read/write data sections into the correct section of RAM.

This function is called only once, by the `__start()` function.

Board-specific No.

See Also "__start()" on page 60

__init_hardware()

Description Performs hardware initializations.

Source File **PowerPC** `__ppc_eabi_init.c`
MIPS `__mips_eabi_init.c`

Prototype

```
void __init_hardware ( void );
```

Parameters None.

Return None.

Remarks This function performs all hardware initializations which do *not* depend on accessing TRK (data) memory sections. TRK data sections are set up after `__init_hardware()` is called.

`__init_hardware()` *must* perform the following two tasks, in order:

1. Set up the memory system so that memory access is safe.
2. Call into __copy_vectors(), which copies exception vectors into their proper place in memory. This must be performed *after* step 1 because it accesses memory.

This function is called only once, by the `__start()` function.

Board-specific Yes.

See Also "__copy_vectors()" on page 56,

"__start()" on page 60

__init_registers()

Description Initializes important processor registers.

Source File `__start.c`

Prototype
```
static void __init_registers ( void );
```

Parameters None.

Return None.

Remarks This function initializes the EABI (Embedded Application Binary Inteface) registers.

On the PowerPC processor, this would include the "r1", or stack pointer register, and the small data area pointers, "r2" and "r13".

On a MIPS processor, this would include the "sp", or stack pointer register, and the "gp", or global pointer register.

This function is called only once, by the `__start()` function.

Board-specific No.

See Also "__start()" on page 60

__init_user()

Description Performs any last-minute initializations.

Source File **PowerPC** `__ppc_eabi_init.c`
MIPS `__mips_eabi_init.c`

Prototype
```
void __init_user ( void );
```

Parameters None.

Return None.

Remarks In the default implementation of MetroTRK, this function does nothing. When customizing for new board configurations, you can use this function to do any "last-minute" initializations that depend on the rest of MetroTRK being initialized.

This function is called only once, by the `__start()` function.

Board-specific Yes.

See Also "__start()" on page 60

__reset

Description Resets the boards and re-starts MetroTRK.

Source File `__reset.s`

Prototype Not a function, this is simply labeled assembly code.

Remarks This code may have processor-specific or board-specific aspects to reset the board.

The final action of this code is to re-start MetroTRK by calling into the `__start()` function.

Board-specific Could be.

See Also "__start()" on page 60

__start()

Description Initializes important processor registers.

Source File `__start.c`

Prototype `void __start ( void );`

Parameters None.

Return None.

Remarks This function initializes the EABI runtime environment as well as MetroTRK data. It performs the following tasks, in order:

1. Register initializations: `__init_registers()`
2. Hardware initializations (this step sets up memory system and makes memory access safe): `__init_hardware()`
3. MetroTRK data initializations: `__init_data()`
4. Additional customized initializations: `__init_user()`

This function is the very first code executed upon board reset.

Board-specific No.

See Also "__init_data()" on page 57,

"__init_hardware()" on page 58,

"__init_registers()" on page 59,

"__init_user()" on page 59

DoConnect()

Description Responds to the `Connect` request from the host debugger.

Source File `msghndlr.c`

Prototype `DSError DoConnect ( MessageBuffer* b );`

Parameters Parameters for this function are:

b	`Message-Buffer*`	The message buffer which contains the original request (input) and the reply (output.) This message contains no input arguments.

Return Returns a `DSError` error code.

Remarks All this procedure does is send an acknowledgment back to the host debugger.

Board-specific No.

DoContinue()

Description Responds to the `Continue` request from the host debugger.

Source File `msghndlr.c`

Prototype

```
DSError DoContinue ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	`Message-Buffer*`	The message buffer which contains the original request (input) and the reply (output.) This message contains no input arguments.

Return Returns a `DSError` error code.

Remarks This procedure is responsible for swapping in the context of the target program and then starting it running again. Since it is processor-specific, most of the work is done in the board-level function `TargetContinue()`.

Board-specific No.

See Also "TargetContinue()" on page 79

DoFputs()

Description Creates and sends a `Fputs` message to the host debugger. This message tells the host debugger to output a string.

Source File `support.c`

Prototype

```
DSError DoFputs ( DSFputsFileOptions options,
char* s, bool needACK );
```

Parameters Parameters for this function are:

`options`	`DSFputsFileOptions`	One of the following values: · `kDSStdout` · `kDSStderr` (See `msgcmd.h` for details on these)
`s`	`char*`	The zero-terminated character string to be output by the host debugger.
`needACK`	`bool`	If TRUE, the procedure doesn't return until it gets acknowledgment from the host debugger (or it gets an error.)

Return Returns a `DSError` error code.

Remarks None.

Board-specific No.

See Also `msgcmd.h`

DoNotifyStopped()

Description MetroTRK uses this function to send notification to the host debugger that the target program has been stopped.

Source File `notify.c`

Prototype

```
DSError DoNotifyStopped ( MessageCommandID command
);
```

Parameters Parameters for this function are:

`command`	`Message-Command-ID`	The type of message to be sent to the debugger. This can be one of the following values: • `kDSNotifyStopped` • `kDSNotifyException` See `msgcmd.h` for more information about these messages.

Return Returns a `DSError` error code.

Remarks To build the actual notification message, this function calls `TargetAddStopInfo()` or `TargetAddExceptionInfo()`, depending on which kind of notification is being sent.

Board-specific No.

See Also "TargetAddStopInfo()" on page 78,

"TargetAddExceptionInfo()" on page 78,

`msgcmd.h`

DoReadMemory()

Description Responds to the `ReadMemory` request from the host debugger. Reads a section of memory from the target board.

Source File `msghndlr.c`

Prototype

```
DSError DoReadMemory ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	Message-Buffer*	The message buffer which contains the original request (input) and the reply (output.) For information on the arguments contained in this message, see "ReadMemory" on page 32.

Return Returns a `DSError` error code.

Remarks This procedure reads a section of memory from the target board. The first thing it does is to check that the memory addresses are within the 32-bit range (extended memory is not yet supported). It then reads from memory, checking first to make sure the range of addresses is valid. Most of the work is actually done in the processor-specific function `TargetReadMemory()`.

Board-specific No.

See Also "TargetReadMemory()" on page 85

DoReadRegisters()

Description Responds to the `ReadRegisters` request from the host debugger. Reads a sequence of registers from the target board.

Source File `msghndlr.c`

Prototype

```
DSError DoReadRegisters ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	Message-Buffer*	The message buffer which contains the original request (input) and the reply (output.) For information on the arguments contained in this message, see "ReadRegisters" on page 35.

Return Returns a `DSError` error code.

Remarks This procedure reads a sequence of registers from the target board. The first thing it does is to check that the input sequence is valid (first index smaller than last.) To actually read the register values, it first checks which *type* of registers are desired. Depending on the type of registers it is dealing with, it calls one of the following functions:

`kDSRegistersDefault`: Calls `TargetReadDefault()`

`kDSRegistersFP`: Calls `TargetReadFP()`

`kDSRegistersExtended1`: Calls `TargetReadExtended1()`

`kDSRegistersExtended2`: Calls `TargetReadExtended2()`

These "type" register constants are defined in `msgcmd.h`.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

See Also "TargetReadDefault()" on page 81,

"TargetReadFP()" on page 84,

"TargetReadExtended1()" on page 82,

"TargetReadExtended2()" on page 83,

`msgcmd.h`

DoReset()

Description Responds to the `Reset` request from the host debugger. This procedure re-initializes MetroTRK and resets board hardware.

Source File `msghndlr.c`

Prototype
```
DSError DoReset ( MessageBuffer* b );
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which contains the original request (input) and the reply (output.) This message contains no input arguments.

Return Doesn't really return anything since it never actually returns.

Remarks Calls the `__reset` code segment, which is the starting point for MetroTRK initialization. Sends an acknowledgment to the host debugger before resetting, since control won't be returned once `__reset` is called.

Board-specific No.

See Also "__reset" on page 60,

DoStep()

Description Responds to the `Step` request from the host debugger. This procedure steps through an arbitrary number of instructions in the host program or, alternatively, until the PC (Program Counter) is outside a given range of values.

Source File `msghndlr.c`

Prototype
```
DSError DoStep ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	`Message-Buffer*`	The message buffer which contains the original request (input) and the reply (output.) For information on the arguments contained in this message, see "Step" on page 38.

Return Returns a `DSError` error code.

Remarks Checks the `options` input argument, which can have one of the following values: `kDSStepSingle` or `kDSStepOutOfRange`. if the value is `kDSStepSingle`, it calls the processor-specific function `TargetSingleStep()`. This function will step the number of steps specified in the message. If the value is `kDSStepOutOfRange`, it calls the processor-specific function `TargetStepOutOfRange()`. This function will run the code until the PC is outside the range of values specified in the message.

Board-specific No.

See Also "TargetSingleStep()" on page 86,

"TargetStepOutOfRange()" on page 86

DoSupportMask()

Description Responds to the `SupportMask` request from the host debugger. This function basically replies to the message by sending a vector representing the messages that we support.

Source File `msghndlr.c`

Prototype

```
DSError DoSupportMask ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	`Message-Buffer*`	The message buffer which contains the original request (input) and the reply (output.) This message contains no input arguments.

Return Returns a `DSError` error code.

Remarks This procedure calls into the processor/board-specific function `TargetSupportMask()`, which returns a 256-bit bit-vector describing which of the messaging API calls are supported. It then puts this vector into a reply message that gets sent to the host debugger. If you are customizing MetroTRK, you may need to change this function if you are adding support for any new messages or removing support for any existing messages.

Within the bit-vector returned, each bit corresponds to the message (type `MessageCommandID`) with an ID matching the position of the bit in the array. If the bit value is 1, it signifies that the message is available. If the value is 0, it signifies that the message is *not* available.

As an example, if `kDSReset` were available, then the 4th bit of mask would be 1 since `kDSReset` is the 4th message (its value is actually 3, but we start counting from 0.)

Board-specific No.

See Also "TargetSupportMask()" on page 87

DoVersions()

Description Responds to the `GetVersions` request from the host debugger. Replies with a set of four version numbers.

Source File `msghndlr.c`

Prototype

```
DSError DoVersions ( MessageBuffer* b );
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which contains the original request (input) and the reply (output.) This message contains no input arguments.

Return Returns a `DSError` error code.

Remarks This function replies to the host debugger with a set of four version numbers. These represent two attributes, *kernel* and *protocol*, and each attribute has a *major* and a *minor* version number.

The kernel attribute represents the version of the MetroTRK build. It should change anytime any part of MetroTRK is changed.

The protocol attribute represents the version of the messaging API and low-level serial protocols used by MetroTRK. This attribute should change whenever one of these protocols is altered.

Most of the actual work of this function is done in the function `TargetVersions()`.

Board-specific No.

See Also "TargetVersions()" on page 88

DoWriteMemory()

Description Responds to the `WriteMemory` request from the host debugger. Writes values to a segment of memory on the target board.

Source File `msghndlr.c`

Prototype

```
DSError DoWriteMemory ( MessageBuffer* b );
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which contains the original request (input) and the reply (output.) For information on the arguments contained in this message, see "WriteMemory" on page 34.

Return Returns a `DSError` error code.

Remarks This procedure writes values to a segment of memory on the target board. The first thing it does is to check that the memory addresses are within the 32-bit range (extended memory is not yet supported). It then writes to memory, checking first to make sure the range of addresses is valid on the target hardware. Most of the work is actually done in the processor-specific function `TargetWriteMemory()`.

Board-specific No.

See Also "TargetWriteMemory()" on page 93

DoWriteRegisters()

Description Responds to the `WriteRegisters` request from the host debugger. Writes values to a sequence of registers on the target board.

Source File `msghndlr.c`

Prototype

```
DSError DoWriteRegisters ( MessageBuffer* b );
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which contains the original request (input) and the reply (output.) For information on the arguments contained in this message, see "WriteRegisters" on page 37.

Return Returns a `DSError` error code.

Remarks This procedure writes values to a sequence of registers on the target board. The first thing it does is to check that the input sequence is valid (first index smaller than last.) To actually write the register values, it first checks which *type* of registers are being written to. Depending on the type of registers it is dealing with, it calls one of the following functions:

`kDSRegistersDefault`: Calls `TargetWriteDefault()`

`kDSRegistersFP`: Calls `TargetWriteFP()`

`kDSRegistersExtended1`: Calls `TargetWriteExtended1()`

`kDSRegistersExtended2`: Calls `TargetWriteExtended2()`

These type register constants are defined in `msgcmd.h`.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

See Also "TargetWriteDefault()" on page 89,

"TargetWriteFP()" on page 92,

"TargetWriteExtended1()" on page 90,

"TargetWriteExtended2()" on page 91,

`msgcmd.h`

InitializeUART()

Description Initializes the serial hardware on the target board.

Source File `uart.c`

Prototype

```
UARTError InitializeUART ( UARTBaudRate baudRate);
```

Parameters Parameters for this function are:

`baudRate`	`UARTBaudRate`	The rate at which MetroTRK communicates with the host debugger. The type `UARTBaudRate` is defined in `UART.h`.

Return Returns a `UARTError` error code.

Remarks None.

Board-specific Yes.

ReadUARTPoll()

Description Polls the serial device to see if there is a character to be read. If there is, it reads it; otherwise, it returns an error.

Source File `uart.c`

Prototype

```
UARTError ReadUARTPoll ( char* c );
```

Parameters Parameters for this function are:

`c`	`char*`	Pointer to the output variable for the character read.

Return Returns a `UARTError` error code. If there was no character ready for reading, it returns `kUARTNoData`, otherwise it returns `kUARTNoError`.

Remarks None.

Board-specific Yes.

ReadUART1()

Description Reads one byte from the serial device.

Source File `uart.c`

Prototype

```
UARTError ReadUART1 ( char* c );
```

Parameters Parameters for this function are:

`c`	`char*`	Pointer to the output variable for the character read.

Return Returns a `UARTError` error code.

Remarks Will wait until a character is available (or an error occurs.)

Board-specific Yes.

ReadUARTN()

Description Reads `N` bytes from the serial device.

Source File `uart.c`

Prototype

```
UARTError ReadUARTN ( void* bytes, unsigned long limit );
```

Parameters Parameters for this function are:

`bytes`	`void*`	Pointer to the output buffer for the data read.
`limit`	`unsigned long`	Number of bytes to read and size of output buffer.

Return Returns a `UARTError` error code.

Remarks Will not return until the specified number of bytes have been read (or an error occurs.)

This function is wholly derived from `ReadUART1()`.

Board-specific No.

See Also "ReadUART1()" on page 73

ReadUARTString()

Description Reads a terminated string from the serial device.

Source File `uart.c`

Prototype

```
UARTError ReadUARTString ( char* s, unsigned long
limit, char termChar );
```

Parameters Parameters for this function are:

`s`	`char*`	Pointer to the output buffer for the string read.
`limit`	`unsigned long`	Size of output buffer.
`termChar`	`char`	Character that signals the end of the string (in the input stream.)

Return Returns a `UARTError` error code.

Remarks This function always terminates the string (in the output buffer) with a null (`\0`) character. This means that the buffer must be one byte longer than the actual length of the string.

This function will not return until a terminating character is read from the input or the buffer is overflowed. It will not time-out if the input stream stops.

This function is wholly derived from `ReadUART1()`.

Board-specific No.

See Also "ReadUART1()" on page 73

TargetAccessMemory()

Description Used to read from or write to memory. Read or write operation is selected with a boolean input parameter. The function performs a

check to make sure specified memory addresses are within valid range for target board.

Source File `targimpl.c`

Prototype

```
DSError TargetAccessMemory ( void* data, void*
virtualAddress, size_t* memorySize,
MemoryAccessOptions accessOptions, bool read );
```

Parameters Parameters for this function are:

`data`	`void*`	In the case of a *read* operation, this is where the output of the read goes. In the case of a *write* operation, this is a pointer to the data to write.
`virtual-Address`	`void*`	This is the starting address in memory for the read or write operation.
`memory-Size`	`size_t*`	In the case of a *read* operation this is, on input, the size of the area to be read and, on output, the size of the area *actually* read. In the case of a *write* operation this is, on input, the amount of data to be written and, on output, the amount of data *actually* read.

`access-Options`	`Memory-Access-Options`	One of the following values: • `kUserMemory`: When reading code with debugging breakpoints, hide breakpoints. • `kDebuggerMemory`: When reading code with debugging breakpoints, show breakpoints. This enumerated type is defined in `targimpl.h`. It directly corresponds to the `DS_MSG_MEMORY_USERVIEW` message option as defined in `msgcmd.h` (used with the `ReadMemory` and `WriteMemory` messages.)
`read`	`bool`	This argument selects whether to do a read operation or a write operation. If the value is `TRUE`, a read is performed, else a write is performed.

Return Returns a `DSError` error code.

Remarks This procedure does not depend on specifics of the board memory configuration. It checks board-level specifics by calling into `ValidMemory32()`, which *is* board-specific and checks the validity of the target addresses based on the board's memory configuration.

Both `DoReadMemory()` and `DoWriteMemory()` funnel into this function to do the actual memory accessing.

Board-specific No.

See Also "DoReadMemory()" on page 64,

"DoWriteMemory()" on page 70,

"ValidMemory32()" on page 95,

`msgcmd.h`,

`targimpl.h`

TargetAddExceptionInfo()

Description Used to build a `NotifyException` message when notifying the host debugger that an exception has occurred on the board.

Source File `targimpl.c`

Prototype

```
DSError TargetAddExceptionInfo (MessageBuffer* b);
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which will be the `NotifyException` notification. For information on the arguments contained in this message, see "NotifyException" on page 41.

Return Returns a `DSError` error code.

Remarks The information written into the message may differ from processor to processor, but generally contains information like the PC (Program Counter) at the time the exception was generated, the instruction at that PC, and the exception ID. See the actual source code to find out the specifics for the processor you are targeting.

PowerPC For PowerPC, see the file `m8xxxcpt.h` in the `export\` directory.

MIPS For MIPS, see the file mips_`except.h` in the `export\` directory.

Board-specific No.

TargetAddStopInfo()

Description Used to build a `NotifyStopped` message when notifying the host debugger that the target program has been stopped.

Source File `targimpl.c`

Prototype

```
DSError TargetAddStopInfo ( MessageBuffer* b );
```

Parameters Parameters for this function are:

`b`	`MessageBuffer*`	The message buffer which will be the `NotifyStopped` notification. For information on the arguments contained in this message, see "NotifyStopped" on page 40.

Return Returns a `DSError` error code.

Remarks The information written into the message may differ from processor to processor, but generally contains information like the PC (Program Counter) at the time the exception was generated, the instruction at that PC, and the exception ID. See the actual source code to find out the specifics for the processor you are targeting.

PowerPC For PowerPC, see the file `m8xxxcpt.h` in the `export\` directory.

MIPS For MIPS, see the file mips_`except.h` in the `export\` directory.

Board-specific No.

TargetContinue()

Description Responds to the Continue request from the host debugger. This function basically starts the target program running and then blocks until MetroTRK gets control back (because a relevant exception occurred.)

Source File `targimpl.c`

Prototype

```
DSError TargetContinue ( MessageBuffer* b );
```

Parameters Parameters for this function are:

b	MessageBuffer*	The message buffer which contains the original request (input.) This message buffer is actually not used at all since there are no input parameters and no reply message.

Return Returns a `DSError` error code.

Remarks This function starts the program running by calling the `SwapAndGo()` function. When MetroTRK regains control (because of an unhandled exception or a breakpoint), control will fall out of `TargetContinue()` and back into the MetroTRK core, where the exception will be handled properly.

Board-specific No.

TargetInterrupt()

Description Handles an exception by notifying the host debugger.

Source File `targimpl.c`

Prototype

```
DSError TargetContinue ( NubEvent* event);
```

Parameters Parameters for this function are:

event	NubEvent*	This is the original event triggered by the exception or breakpoint.

Return Returns a `DSError` error code.

Remarks This function is called when an exception or breakpoint occurs. It basically just calls `DoNotifyStopped()` to notify the host debugger.

Board-specific No.

See Also "DoNotifyStopped()" on page 63

TargetReadDefault()

Description Reads a sequence of registers from the default register block.

Source File `targimpl.c`

Prototype

```
DSError TargetReadDefault ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to read.
`last-Register`	`unsigned int`	The number of the last register in the sequence to read.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually read (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while reading the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetReadExtended1()

Description Reads a sequence of registers from the `extended1` register block.

Source File `targimpl.c`

Prototype

```
DSError TargetReadExtended1 ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to read.
`last-Register`	`unsigned int`	The number of the last register in the sequence to read.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually read (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while reading the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetReadExtended2()

Description Reads a sequence of registers from the `extended2` register block.

Source File `targimpl.c`

Prototype

```
DSError TargetReadExtended2 ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to read.
`last-Register`	`unsigned int`	The number of the last register in the sequence to read.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually read (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while reading the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetReadFP()

Description Reads a sequence of registers from the floating point register block.

Source File `targimpl.c`

Prototype

```
DSError TargetReadFP ( unsigned int firstRegister,
unsigned int lastRegister, MessageBuffer* b,
size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`firstRegister`	`unsigned int`	The number of the first register in the sequence to read.
`lastRegister`	`unsigned int`	The number of the last register in the sequence to read.
`b`	`MessageBuffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`registerStorageSize`	`size_t*`	On output, the number of bytes actually read (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while reading the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetReadMemory()

Description Handles the `ReadMemory` request from host debugger. This function reads a contiguous section of memory from the target board. It performs a validity check on the desired addresses before attempting the read.

Source File `targimpl.c`

Prototype

```
DSError TargetReadMemory ( void* data, void*
virtualAddress, size_t* memorySize,
MemoryAccessOptions accessOptions );
```

Parameters Parameters for this function are:

`data`	`void*`	The data read from memory gets copied into this buffer.
`virtual-Address`	`void*`	This is the starting address in memory for the read operation.
`memory-Size`	`size_t*`	This is, on input, the size of the area to be read and, on output, the size of the area *actually* read.
`access-Options`	`Memory-Access-Options`	One of the following values: • `kUserMemory`: When reading code with debugging breakpoints, hide breakpoints. • `kDebuggerMemory`: When reading code with debugging breakpoints, show breakpoints. This enumerated type is defined in `targimpl.h`. It directly corresponds to the `DS_MSG_MEMORY_USERVIEW` message option as defined in `msgc-md.h` (used with the ReadMemory and `WriteMemory` messages.)

Return Returns a `DSError` error code.

Remarks This function actually doesn't do any real work. It just calls into the function `TargetAccessMemory()`.

Board-specific No.

See Also "TargetAccessMemory()" on page 75,

`msgcmd.h`,

`targimpl.h`

TargetSingleStep()

Description Steps through a specified number of instructions.

Source File `targimpl.c`

Prototype

```
DSError TargetSingleStep ( unsigned count);
```

Parameters Parameters for this function are:

`count`	`unsigned int`	The number of lines to step across.

Return Returns a `DSError` error code.

Remarks This function works by setting up the trace exception, and then checking after each instruction whether it has completed the desired number of steps.

Board-specific No.

TargetStepOutOfRange()

Description Runs the target program until the PC (Program Counter) is outside a given range of values.

Source File `targimpl.c`

Prototype
```
DSError TargetStepOutOfRange ( ui32 start, ui32
end );
```

Parameters Parameters for this function are:

`start`	`ui32`	The starting address of the range.
`end`	`ui32`	The ending address of the range.

Return Returns a `DSError` error code.

Remarks This function works by setting up the trace exception, and then checking after each instruction whether the PC is outside the given range of values.

Board-specific No.

TargetSupportMask()

Description Returns a mask which indicates which debug messages the current MetroTRK supports.

Source File `targimpl.c`

Prototype
```
DSError TargetSupportMask ( DSSupportMask *mask )
```

Parameters Parameters for this function are:

Name	Type	Description
`mask`	`DSSupportMask`	A bit-array of 32 bytes, where each bit corresponds to the message (type `MessageCommandID`) with an ID matching the position of the bit in the array. If the bit value is 1, it signifies that the message is available. If the value is 0, it signifies that the message is *not* available. As an example, if `kDSReset` were available, then the 4th bit of mask would be 1 since `kDSReset` is the 4th message (its value is actually 3, but we start counting from 0.) See the documentation in `msgcmd.h` for details. Also see how the default values are set in `target.h`

Return None.

Remarks None.

Board-specific Yes.

See Also `msgcmd.h`

TargetVersions()

Description Returns a set of four version numbers for the running MetroTRK build.

Source File `targimpl.c`

Prototype
```
DSError TargetVersions ( DSVersions* versions);
```

Parameters Parameters for this function are:

`versions`	`DSVer-sions*`	Output variable containing version information for the running MetroTRK build.

Return Returns a `DSError` error code (always returns `kNoError`.)

Remarks This function returns in the output variable `versions` a set of four version numbers. These represent two attributes, *kernel* and *protocol*, and each attribute has a *major* and a *minor* version number.

The kernel attribute represents the version of the MetroTRK build. It should change anytime any part of MetroTRK is changed.

The protocol attribute represents the version of the messaging API and low-level serial protocols used by MetroTRK. This attribute should change whenever one of these protocols is altered.

The default implementation of this function uses compile-time constants which, by default, are defined in the board-specific file `target.h`. In order to customize MetroTRK, you shouldn't need to modify the function `TargetVersions()`, only these four constants. They are:

- `DS_KERNEL_MAJOR_VERSION`
- `DS_KERNEL_MINOR_VERSION`
- `DS_PROTOCOL_MAJOR_VERSION`
- `DS_PROTOCOL_MINOR_VERSION`

Board-specific Yes (Indirectly in that you may want to change the version numbers if you modify MetroTRK for your board configuration.)

See Also `target.h`

TargetWriteDefault()

Description Writes data to a sequence of registers in the default register block.

Source File `targimpl.c`

Prototype

```
DSError TargetWriteDefault ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to write to.
`last-Register`	`unsigned int`	The number of the last register in the sequence to write to.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually written (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while writing to the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetWriteExtended1()

Description Writes data to a sequence of registers in the `extended1` register block.

Source File `targimpl.c`

Prototype

```
DSError TargetWriteExtended1 ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`firstRegister`	`unsigned int`	The number of the first register in the sequence to write to.
`lastRegister`	`unsigned int`	The number of the last register in the sequence to write to.
`b`	`MessageBuffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`registerStorageSize`	`size_t*`	On output, the number of bytes actually written (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while writing to the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetWriteExtended2()

Description Writes data to a sequence of registers in the `extended2` register block.

Source File `targimpl.c`

Prototype

```
DSError TargetWriteExtended2 ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to write to.
`last-Register`	`unsigned int`	The number of the last register in the sequence to write to.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually written (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while writing to the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetWriteFP()

Description Writes data to a sequence of registers in the floating point register block.

Source File `targimpl.c`

Prototype

```
DSError TargetWriteFP ( unsigned int
firstRegister, unsigned int lastRegister,
MessageBuffer* b, size_t* registerStorageSize);
```

Parameters Parameters for this function are:

`first-Register`	`unsigned int`	The number of the first register in the sequence to write to.
`last-Register`	`unsigned int`	The number of the last register in the sequence to write to.
`b`	`Message-Buffer*`	The message buffer which contains the original request (input), as well as the reply message (output.)
`register Storage Size`	`size_t*`	On output, the number of bytes actually written (maximum of 2048 bytes.)

Return Returns a `DSError` error code.

Remarks This function checks the validity of the range of registers before actually trying to access them. It also tries to catch exceptions that occur while writing to the registers.

PowerPC All registers with mnemonic names are defined in the file `m8xxreg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

MIPS All registers with mnemonic names are defined in the file `mips_reg.h`. This file also defines the sizes (bit-lengths) of registers in the different register blocks.

Board-specific No.

TargetWriteMemory()

Description Handles the `WriteMemory` request from host debugger. This function writes data to a contiguous section of memory on the target board. It performs a validity check on the desired addresses before attempting the write.

Source File `targimpl.c`

Prototype

```
DSError TargetWriteMemory ( void* data, void*
virtualAddress, size_t* memorySize,
MemoryAccessOptions accessOptions );
```

Parameters Parameters for this function are:

`data`	`void*`	A pointer to the data to be written.
`virtual-Address`	`void*`	This is the destination starting address (on the target board) for the write operation.
`memory-Size`	`size_t*`	This is, on input, the size of the data to be written and, on output, the amount *actually* written.
`access-Options`	`Memory-Access-Options`	Has no effect on this function. It has an effect on the *read* version.

Return Returns a `DSError` error code.

Remarks This function actually doesn't do any real work. It just calls into the function `TargetAccessMemory()`.

Board-specific No.

See Also "TargetAccessMemory()" on page 75,

`msgcmd.h`,

`targimpl.h`

TerminateUART()

Description Deactivate the serial device, as necessary.

Source File `uart.c`

Prototype `UARTError TerminateUART ( void );`

Parameters None.

Return Returns a `UARTError` error code.

Remarks This function often does nothing. In fact, in the current MetroTRK implementation, it isn't even called.

Board-specific Yes.

ValidMemory32()

Description Called when MetroTRK reads or writes to memory. This function checks whether the range of addresses is valid on the target board.

Source File `targimpl.c`

Prototype

```
DSError ValidMemory32 ( const void* addr, size_t
length, ValidMemoryOptions readWriteable);
```

Parameters Parameters for this function are:

`addr`	`const void*`	The starting address of the memory segment.
`length`	`size_t`	The length of the memory segment.
`read Write-able`	`Valid Memory` Options	One of the following values: · `kValidMemoryReadable` · `kValidMemoryWriteable`

Return Returns a `DSError` error code. If the memory segment is valid, returns `kNoError`, else returns `kInvalidMemory`.

Remarks On the PowerPC version of MetroTRK, this function is board-specific, relying on information about the memory layout of the target board. To customize for a new board configuration, you need to change the values within this function.

On the MIPS version of MetroTRK, this function is not board-specific. It instead relies on a board-specific variable, `gMemMap`, to get information about the memory layout of the target board. To customize for a new board configuration, you need to change the value of `gMemMap`, which is defined in the file `memmap.h`.

Board-specific **PowerPC** `Yes.`
MIPS `No.`

See Also `memmap.h`

WriteUART1()

Description Writes one byte to the serial device.

Source File `uart.c`

Prototype
```
UARTError WriteUART1 ( char c );
```

Parameters Parameters for this function are:

`c`	`char`	The character to be written.

Return Returns a `UARTError` error code.

Remarks None.

Board-specific Yes.

WriteUARTN()

Description Writes `N` bytes to the serial device.

Source File `uart.c`

Prototype
```
UARTError WriteUARTN ( const void *bytes, unsigned
long length );
```

Parameters Parameters for this function are:

`bytes`	`const void*`	Pointer to the input data.
`length`	`unsigned long`	The number of bytes to be written.

Return Returns a `UARTError` error code.

Remarks This function is wholly derived from `WriteUART1()`.

Board-specific No.

WriteUARTString()

Description Writes a character string to the serial device.

Source File `uart.c`

Prototype

```
UARTError WriteUARTString ( const char* string );
```

Parameters Parameters for this function are:

`string`	`const char*`	Pointer to the input data.

Return Returns a `UARTError` error code.

Remarks The input string must have a null termination character (`\0`), but this terminating null character is *not* written to the serial device.

This function is wholly derived from `WriteUART1()`.

Board-specific No.

See Also "WriteUART1()" on page 96

Index

L

M

N

P

Q

R

S

T

U

V

W

CodeWarrior

MetroTRK Manual

Credits

writing lead: Benjamin Berman

other writers: Jim Trudeau

engineering: Steve Moore, Lawrence You, Khurram Qureshi

frontline warriors: L. Frank Turovich, Todd McDaniel, Jeanna Stavas, and CodeWarrior users everywhere!

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C and C++ compilers
Pascal Compilers Reference	Information on the Pascal and Object Pascal compilers
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler manual
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks standard C library
MSL C++ Reference	Function reference for the Metrowerks standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
The PowerPlant Book	Guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting the Java VM	How to use CodeWarrior to program for the Java virtual machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting Win32	How to use CodeWarrior to program for Windows 95/NT