CodeWarrior®
MetroTRK Reference

Revised: 20020124 REV 0

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Table of Contents

1

Introduction

MetroTRK is a target-resident kernel that is an on-target debug monitor for the CodeWarrior debugger. This manual describes MetroTRK and explains how to customize it for use with your hardware configuration.

This chapter includes the following topics:

Read the Release Notes!

Please read the release notes. The release notes contain important information about new features, bug fixes, and incompatibilities that may not be included in the documentation due to release deadlines. The release notes reside on your CodeWarrior CD in the Release Notes folder.

What's in This Book

This manual provides information for users who must customize MetroTRK to work with a new target board configuration.

NOTE For information on loading and using MetroTRK (information which differs among sets of CodeWarrior tools), see the *Targeting* manual for your target processor.

Table 1.1 describes the contents of this manual, referenced by the name of each chapter and appendix.

NOTE In addition to the chapters and appendixes described in Table 1.1, this manual may contain one or more processor-specific appendixes, depending on the product you purchased. See the table of contents to determine whether your version of this manual contains any processor-specific appendixes.

Table 1.1 Contents of chapters and appendixes

Chapter or Appendix Name	Description
Introduction	This chapter, which provides an overview of the manual.
MetroTRK Concepts	Describes the various tasks MetroTRK performs and how these tasks are implemented. Provides an overview of MetroTRK.
MetroTRK Communications	Describes the MetroTRK communication protocol. This chapter is useful for debugging MetroTRK and for those developing software that communicates with MetroTRK.

Chapter or Appendix Name	Description
Customizing MetroTRK	Provides details about the specific areas where you can customize or re-target MetroTRK to work with your hardware configuration.
MetroTRK Porting Example	This chapter presents an example that describes how MetroTRK was ported to a 68K target board (M68328 ADS).
Debug Message Interface Reference	This chapter documents the MetroTRK debug message interface.
MetroTRK Function Reference	Describes MetroTRK functions that may be relevant if you are customizing MetroTRK to work with new target boards.

If this is your first attempt to customize the MetroTRK, read "MetroTRK Concepts" on page 15 and "MetroTRK Communications" on page 23 before reading "Customizing MetroTRK" on page 45. If you customized MetroTRK previously, you may prefer to go directly to "Customizing MetroTRK" on page 45 and "MetroTRK Porting Example" on page 63.

Where to Go from Here

This section lists other materials that may help you use or modify MetroTRK. All the CodeWarrior manuals mentioned in this section reside in the following directory on your CodeWarrior CD:

`CodeWarrior_dir\Documentation`

Other CodeWarrior Documentation:

- For information about the CodeWarrior integrated development environment and debugger, see the *IDE User Guide*.
- For information about using CodeWarrior and MetroTRK with a particular target processor, see the *Targeting* manual for your target processor.

Other documentation:

- You can find the RFC 1662 document, which describes the framing portion of the Point-to-Point Protocol, in the `Documentation` folder of the MetroTRK distribution or on the following web page:

  `http://andrew2.andrew.cmu.edu/rfc/rfc1662.html`

Manual Conventions

This manual uses the following typographical conventions:

- Text that appears in a `different` typeface (as the word `different` does in this sentence) indicates that the item is one of the following:
  - File name
  - Directory path
  - Source code
  - Keyboard input
- Variable items appear in a different font with italics applied as shown by *`variable1`* in this sentence. If an item is not variable, you type it exactly as it appears. For example, in the following directory path, *`MetroTRK_dir`* and *`processor_type`* are variable items; `Processor` and `Board` are literal items.

MetroTRK_dir\Processor*processor_type*\Board

2

MetroTRK Concepts

This chapter describes the architecture of MetroTRK and discusses how MetroTRK works. This chapter includes the following topics:

What is MetroTRK?

The MetroTRK program is a debug monitor that resides on the embedded system (along with the target application) and communicates with the debugger using a serial connection. Through this serial communication, MetroTRK acts as the interface between the debugger and the target board.

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

You can use MetroTRK with the CodeWarrior debugger to debug programs running on an embedded system. You indirectly manipulate MetroTRK by controlling the debugger on the host computer.

The implementation of MetroTRK differs depending on the target board for which you are developing. (The differences exist because MetroTRK manipulates target-board-specific resources.)

Your CodeWarrior distribution includes one or more MetroTRK implementations, configured for specific reference boards. If you

are using a supported reference board, you can use MetroTRK without modifications. For information on supported reference boards, see the *Targeting* manual for your target processor.

However, if you are using any board other than a supported reference board, you must modify the MetroTRK source code to support new target boards. (Your CodeWarrior distribution includes the MetroTRK source code.)

MetroTRK Architecture

This section discusses the following MetroTRK components:

- MetroTRK Core
- MetroTRK Execution States
- Request and Notification Handling
- Message Queues

MetroTRK Core

MetroTRK contains a core component called the *MetroTRK core* that controls its internal state and determines which function should handle a particular debugger request. Around this core, MetroTRK has several other modules that perform various tasks.

The MetroTRK core is independent of the target board configuration. However, some handler functions that perform debugging requests are board-dependent.

MetroTRK has two main operating states, the message-handling state and the event-waiting state.

MetroTRK Execution States

A Reset request or a hardware reset causes MetroTRK to enter its board initialization state. After the board initializations complete, MetroTRK enters its message-handling state. In this state, MetroTRK continuously services requests from the debugger.

MetroTRK is in a continuous loop, waiting for requests from the debugger. MetroTRK passes each request that it receives to an appropriate handler function.

NOTE The target application does not execute while MetroTRK is in the message-handling state.

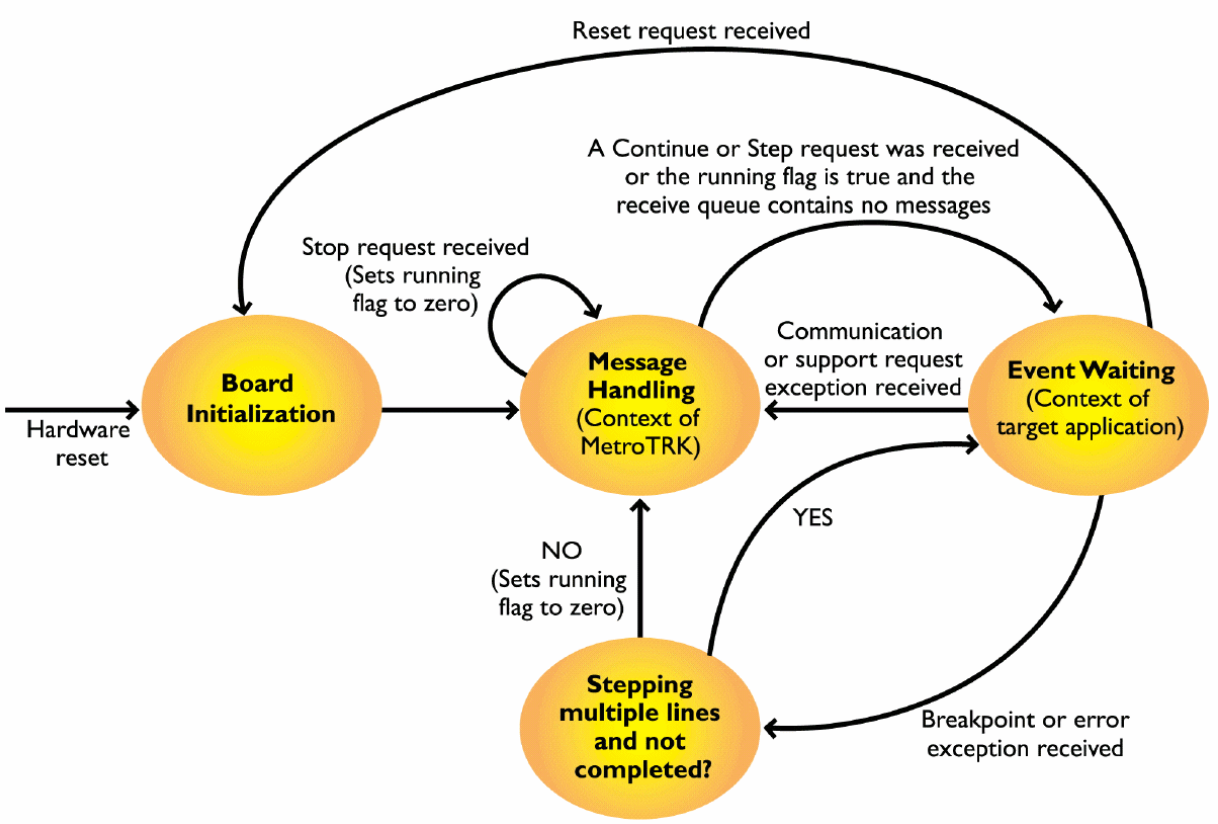
When the debugger sends a Continue or Step request, MetroTRK enters its event-waiting state. While in the event-waiting state, the target application executes rather than MetroTRK. MetroTRK remains inactive, waiting for a relevant exception. When one occurs, MetroTRK stops execution of the target application, resumes control of the processor, and reenters the message-handling state.

NOTE Usually, an exception causes MetroTRK to begin executing again (a context switch) and to enter the message-handling state. However, if MetroTRK currently is processing a multiple-line step command, the target application resumes control of the processor (resumes execution), and MetroTRK reenters the event-waiting state.

MetroTRK again remains in the message-handling state until the debugger sends a Continue or Step request. Then MetroTRK returns to the event-waiting state and the target application begins executing again.

Figure 2.1 shows the state diagram for MetroTRK.

Figure 2.1 MetroTRK state diagram



Message Queues

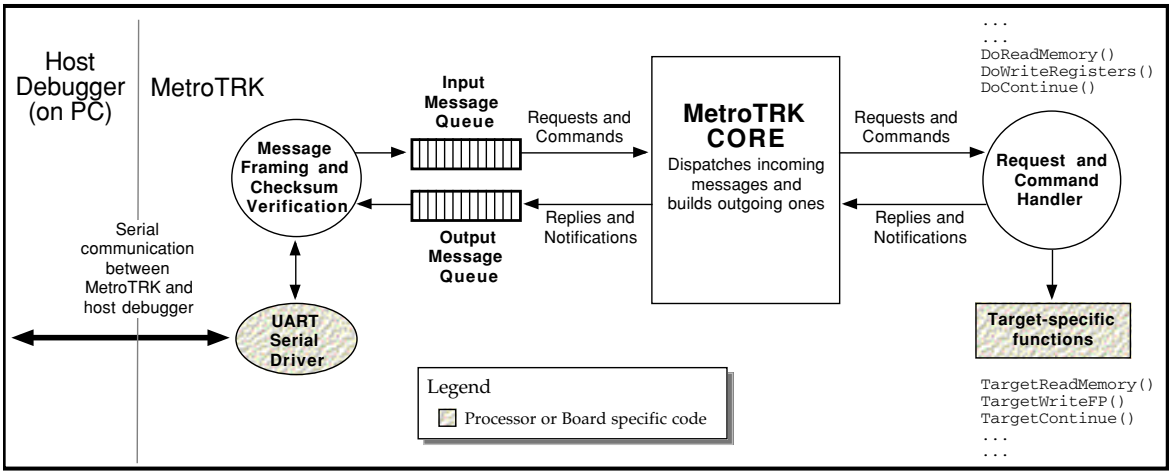
While in the message-handling state, MetroTRK constantly monitors the serial line for incoming requests and stores each one in an incoming message queue. MetroTRK also maintains another message queue for outgoing messages. To send a message to the debugger, MetroTRK places the message in the outgoing message queue and continues processing. The message is sent as soon as the serial line is free.

The message queues serve two important purposes. First, the queues are buffers between the debugger and MetroTRK, which run on different hardware platforms. These buffers keep faster hardware from outrunning slower hardware, which keeps requests and replies from being dropped (lost). Second, the message queues allow MetroTRK to use an event-driven design. The MetroTRK core

centrally dispatches all communication with the debugger, which passes first through the messages queues.

Figure 2.2 shows how data flows through MetroTRK when it is in the message-handling state, including the message queues discussed in this section. In the event-waiting state, MetroTRK is inactive while waiting for the next exception to occur.

Figure 2.2 MetroTRK data-flow diagram



The message queues are not dependent on the target board configuration.

Request and Notification Handling

A set of handler functions separate from the MetroTRK core comprises another module of MetroTRK that handles debugger requests and notifications. Board-specific information related to handling requests and notifications is, in most cases, encapsulated within special header files in the following directory:

MetroTRK_dir\Processor*processor_type*\Board

For more information, see "Customizing Debug Services" on page 55.

MetroTRK Memory Layout

This section contains the following topics:

MetroTRK RAM Sections

Table 2.1 shows the memory sections that exist in RAM when you run MetroTRK from ROM.

Table 2.1 MetroTRK RAM sections

RAM Section	Description
Data	The data section includes all read/write data in the program. When running from a ROM-based version of MetroTRK, MetroTRK copies any initial values from ROM to RAM. MetroTRK uses 6KB of RAM for global data.
Exception vectors	Exception vectors are sections of code executed in the event of a processor exception. The processor determines the location of exception vectors. For more information, see the processor-specific appendixes in this manual.
The stack	MetroTRK requires at least 8KB of RAM for its stack, which is also the maximum amount of RAM that the stack occupies.

Target Application RAM Sections

Specify target application memory sections so that they do not overwrite the MetroTRK memory sections. For more information, see "MetroTRK RAM Sections" on page 20.

One good way to specify the target application memory sections is to place the code and data sections in low memory below the code and data sections of the MetroTRK. You then can place the stack of the target application below the stack of MetroTRK. You must allow enough room for the MetroTRK stack to grow downward.

For more information, see "Customizing Memory Locations" on page 59.

MetroTRK Initializations

MetroTRK begins initializing when `__reset()` executes. The `__reset()` function calls:

- `__init_`*processor*

  The `__init_`*processor* assembly language code initializes processor-specific items. For example, `__init_`*processor* can perform such functions as clearing the cache and the TLB (translation lookaside buffer).

- `__init_board`

  The `__init_board` assembly language code initializes board-specific items. Examples include initializing the central control registers and disabling all interrupts.

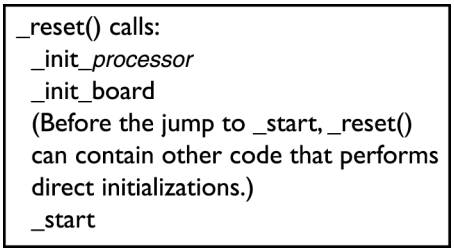
- `__start`

  The `__start` assembly language code starts the runtime code.

If needed, you can place code for other initializations in `__reset()` immediately before the jump to `__start`. For example, sometimes (depending on the reference board for which your default implementation of MetroTRK is targeted) `__init_board` moves ROM in memory. In this case, you can directly reset the program counter in `__reset()` following `__init_board`.

Figure 2.3 shows the preceding initialization sequence.

Figure 2.3 MetroTRK initialization sequence



NOTE The `__reset()` function either uses the initialization sequence discussed in this section or contains the initialization code directly. In either case, the processor-specific initializations precede the board-specific initializations. The board-specific initializations then precede any other direct initializations to perform before making the jump to `__start`.

Some reference boards use assembly language code that is labeled __reset rather than a C function. For information on the location of __reset() or __reset, see the processor-specific appendixes in this manual.

3

MetroTRK Communications

This chapter describes how MetroTRK communicates with the debugger. This chapter is useful for debugging when modifying MetroTRK and for developing debugging systems to communicate with MetroTRK.

MetroTRK continuously communicates with the debugger. This communication has the following levels:

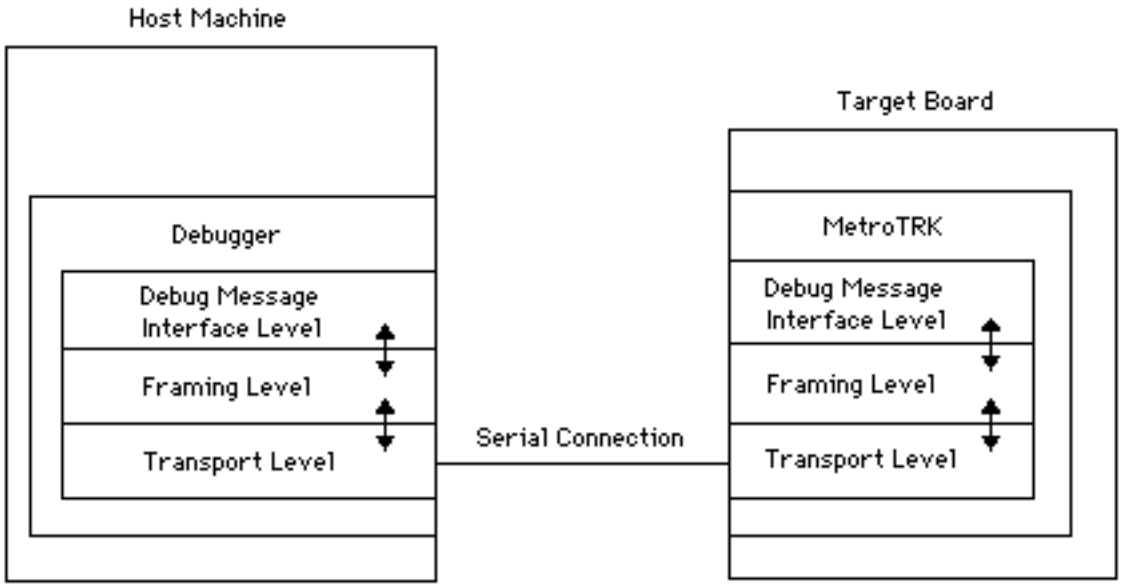
- Transport Level
- Framing Level
- Debug Message Interface Level

NOTE When discussing MetroTRK communications, this chapter uses the terms *sender* and *receiver*. The *sender* is the software that currently is sending a message. The *receiver* is the software that currently is receiving a message.

Both the debugger and MetroTRK can be either a sender or a receiver, depending on the action each is performing at a particular moment.

Figure 3.1 shows the interaction of the communication levels.

Figure 3.1 MetroTRK communication levels



Transport Level

The *transport level* is the level at which the host machine and the target board send physical signals to and receive physicals signals from each other over a serial cable connection.

For maximum portability, the low-level code that drives the serial controller is separated from the MetroTRK core. MetroTRK provides a simple interface that can work with different UART (Universal Asynchronous Receiver Transmitter) drivers if necessary. The default implementations work with the standard on-board serial ports. For more information, see "Modifying Serial Communication Functions" on page 47.

You must configure both MetroTRK and the debugger to use the correct data transmission rate (baud rate) for the serial connection for your target board. For more information, see "Changing the Data Transmission Rate" on page 53.

This section contains the following topics:

- Data Transmission Rate

Serial Communications Settings

Your target board must have a serial port so that MetroTRK can communicate with the debugger running on the host computer. For most target boards, the data transmission rate when communicating with MetroTRK is between 300 baud and 230.4k baud. (For more information, see the *Targeting* manual for your target board or the processor-specific appendixes in this manual.)

MetroTRK usually communicates using the following serial settings:

- 8 data bits
- no parity
- 1 stop bit (8N1)

However, if other supported settings work better for your target board, you can use those settings. For more information, see the *Targeting* manual for your target processor or the processor-specific appendixes of this manual. If you cannot find information about your board, contact technical support.

Data Transmission Rate

The default implementation of MetroTRK uses the highest data transmission rate (baud rate) that the target board can support. The data transmission rate varies depending on the target board or serial controller. For more information, see the *Targeting* manual for your target processor.

NOTE The maximum data transmission rate for the Solaris-hosted CodeWarrior debugger is 38.4 KB. Consequently, if you are using the Solaris-hosted debugger, you must set the data transmission rate in MetroTRK to 38.4 KB, even if the target board accepts a faster rate.

You can set the data transmission rate so that it is appropriate for your hardware. For more information, see "Changing the Data Transmission Rate" on page 53.

Framing Level

The framing level:

- is responsible for reliably transporting messages between MetroTRK and the debugger
- transmits the messages as arbitrarily sized segments of data
- places each data segment in a MetroTRK data frame before transmitting the data
- verifies the contents of each data frame using a checksum

The transmitted data consists of messages defined by the debug message interface. Each MetroTRK data frame contains one message. (For more information, see "Debug Message Interface Level" on page 36.)

NOTE To the framing level, a MetroTRK message is a string of data of a particular length; the framing level ignores the internal structure of the message.

This section contains the following topics:

MetroTRK Data Frames

When communicating over a serial connection, MetroTRK and the debugger transmit all messages in a MetroTRK data frame. A MetroTRK *data frame* is a data segment of arbitrary length delimited at its beginning and end by a special framing character. The MetroTRK data frame also contains checksum information used to verify the integrity of the data received.

Before sending a message, MetroTRK and the debugger place the message in a MetroTRK data frame as follows:

1. Arrange the message in big-endian byte order (most significant byte first).

NOTE Step 1 executes at the debug message interface level rather than the framing level, as shown in Figure 3.2 on page 28.

2. Calculate a one-byte checksum value (by default) and place the checksum value in a byte at the end of the message.
3. Apply an escape sequence to any reserved byte values contained in the message.
4. Add the *start-frame/end-frame flag* (a byte containing the value `0x7e` that allows the receiver to distinguish one frame from another) to the beginning and end of the MetroTRK data frame.

Figure 3.2 shows how MetroTRK and the debugger create a MetroTRK data frame.

Figure 3.2 Creating a MetroTRK data frame

Message: ReadRegisters (MessageID = 0x12)
Arguments:
options = 0x00
firstRegister = 101 (0x0065)
lastRegister = 126 (0x007e)

Original Debug Message Structure

0x12 Message ID	0x00 Options	0x0065 firstRegister	0x007e lastRegister

Building a MetroTRK Data Frame

1. Arrange the message in big-endian byte order.

0x12 Message Byte #1	0x00 Message Byte #2	0x00 Message Byte #3	0x65 Message Byte #4	0x00 Message Byte #5	0x7e Message Byte #6

Debug Message Interface Level

Framing Level

2. Calculate a one-byte checksum value as follows and add the value to the end of the message:

0x12 + 0x00 + 0x00 + 0x65 + 0x00 + 0x7e = 0xf5; the complement of 0xf5 = 0x0a

0x12 Message Byte #1	0x00 Message Byte #2	0x00 Message Byte #3	0x65 Message Byte #4	0x00 Message Byte #5	0x7e Message Byte #6	0x0a 1-byte checksum

3. Apply an escape sequence to any reserved byte values contained in the message.

0x12 Message Byte #1	0x00 Message Byte #2	0x00 Message Byte #3	0x65 Message Byte #4	0x00 Message Byte #5	0x7d Escape char	0x5e Message Byte #6 XOR 0x20	0x0a 1-byte checksum

4. Add the start-frame/end-frame flag to the beginning and end of the MetroTRK data frame.

0x7e Start flag	0x12 Message Byte #1	0x00 Message Byte #2	0x00 Message Byte #3	0x65 Message Byte #4	0x00 Message Byte #5	0x7d Escape char	0x5e Message Byte #6 XOR 0x20	0x0a 1-byte checksum	0x7e End flag

Note: Step 4 completes the MetroTRK data frame.

Checksum Values

The sender places a one-, two-, or four-byte checksum value immediately after the debug message as part of the framing process. The receiver uses the checksum value to verify the integrity of the data.

NOTE The sender calculates checksum values for a debug message *before* creating any escape sequences for the message. In addition, the sender must create an escape sequence for any byte in a checksum value that contains a reserved byte value. For more information, see "Escape Sequences" on page 33.

By default, MetroTRK uses a single-byte checksum value. However, you can customize MetroTRK to use a two- or four-byte checksum value. For more information, see "Customizing Checksum Values" on page 60.

The sender calculates a checksum value serially, by starting at the beginning of the message with an initial value and updating it for each successive byte of message data.

This section contains the following sections, which describe how MetroTRK and the debugger calculate one-, two-, and four-byte checksum values:

- Encoding single-byte checksum values
- Verifying single-byte checksum values
- Using multi-byte checksum values

NOTE MetroTRK contains implementations for computing checksum values. If you are developing your own debugger to communicate with MetroTRK, you may want to borrow your implementation directly from MetroTRK.

For the MetroTRK implementation, see the following files:

`MetroTRK_dir\Export\serframe.h`

`MetroTRK_dir\Portable\serpoll.c`

Encoding single-byte checksum values

The sender encodes a message using a one-byte checksum value as follows:

1. Specify an initial value of `0x00`.
2. Add the value of each character (byte) in the debug message to the running checksum total.
3. Complement the final value.
4. Place the checksum value immediately after the debug message in the MetroTRK data frame.

Listing 3.1 shows a C function that sends a debug message and calculates and sends a one-byte checksum value for the message.

Listing 3.1 Calculating a one-byte checksum value

```
void
SendMessage(ui8* messageData, unsigned int length)
{
    ui8 FCS;
    int onCharacter;

    /* Send message one byte at a time, calculating an FCS
       along the way */
    FCS = 0x00;
    for (onCharacter = 0; onCharacter < length; onCharacter++)
    {
        ui8 currentByte = messageData[onCharacter];

        /* First update the FCS count */
        FCS += currentByte;

        /* Now send the byte (escapes if necessary) */
        TransmitOneByte(currentByte);
    }

    /* Now complement and send the FCS value (escape if
       necessary) */
    TransmitOneByte(FCS ^ 0xFF);
}
```

Verifying single-byte checksum values

The receiver verifies a message encoded with a one-byte checksum value as follows:

1. Specify an initial value of `0x00`.
2. Until the end-frame flag arrives, add the value of each received byte to the current checksum value, in the order received.
3. Check whether the final calculated value is `0xFF`.

   If it is, the message arrived correctly. Otherwise, an error occurred in transmission. (However, this algorithm does not find all transmission errors.)

When the end-frame flag arrives, the receiver omits its value from the checksum value. Consequently, the last calculation adds the received checksum value to the checksum value the receiver was calculating. The sender complemented the received checksum value before sending it.

Adding any number to its complement yields the value `0xFF`. This fact allows the algorithm to determine whether the data arrived correctly.

Listing 3.2 shows a C function that demonstrates how to verify a message encoded with its checksum value.

Listing 3.2 Verifying a message using a one-byte checksum value

```
typedef unsigned char Boolean;
#define TRUE 1
#define FALSE 0

Boolean
VerifyMessageIntegrity()
{
    ui8 currentChar, FCS;

    /* Loop through characters until we hit the end flag */
    FCS = 0x00;
    while !EndFlag(currentChar = GetNextChar())
    {
        FCS += currentChar;
```

```
    }

    /* We have just passed over the encoded complement of the
       original FCS.  If this message matched the original, the
       FCS value should now be 0xFF                              */
    return (FCS == 0xFF);
}
```

Using multi-byte checksum values

For multi-byte checksum values, MetroTRK uses an algorithm called FCS (Frame Check Sequence). Similarly to the one-byte checksum value algorithm, the FCS algorithms for two- and four-byte checksum values calculate a single value over the length of the message data. However, the two- and four-byte algorithms, while more likely to catch communication errors, are computationally more expensive than the single-byte algorithm.

The FCS implementations used by MetroTRK are based on the RFC 1662 standard (the framing portion of the Point-to-Point Protocol). The RFC 1662 standard is based on the original Fast CRC (Cyclic Redundancy Check) algorithm.

The RFC 1662 document and the MetroTRK source code in the following file provide details on calculating multi-byte FCS values:

MetroTRK_dir\Export\serframe.h

The sender always adds multi-byte FCS values to the data frame using little-endian byte order. For example, to send a 32-bit FCS flag with the value `0x01234567`, MetroTRK or the debugger sends the following bytes in the order shown:

- `0x67`
- `0x45`
- `0x23`
- `0x01`

Escape Sequences

The MetroTRK communications protocol has the following reserved byte values:

- `0x7e` (the start-frame/end-frame flag)
- `0x7d` (the *escape character*, which indicates the beginning of an escape sequence)

A debug message or its checksum value can contain bytes equal to these reserved values. In this case, the sender must create an escape sequence for each such byte before sending the message.

An *escape sequence* is a two-character sequence composed of a special escape character (`0x7d`) followed by a transformation of the original byte value. To transform the original character, the sender `XORs` the character with the value `0x20`, as in the following line of C code:

```
escapedChar = originalChar ^ 0x20;
```

NOTE To send a byte with the value of the escape character, the sender must first send the escape character, followed by the transformation of the original byte.

A receiver must determine when a message contains an escape sequence. After encountering an escape character, the receiver performs the same transformation on the byte following the escape character to get the original value of the byte:

```
originalChar = escapedChar ^ 0x20;
```

NOTE The sender calculates checksum values for a debug message *before* creating any escape sequences for the message. In addition, the sender must create an escape sequence for any byte in a checksum value that contains a reserved byte value.

Reliable Message Delivery

A receiver sends a reply message in response to every request or notification message from a sender. Two kinds of reply messages exist: *ACK* (acknowledgment) messages and *NAK* (no acknowledgment) messages.

NOTE Although reply messages are defined on the debug message interface level, the framing level also uses reply messages to ensure reliable message transmission.

ACK messages confirm that the receiver correctly received the preceding message. NAK messages indicate that the receiver did not correctly receive the preceding message.

NOTE For more information, see "Reply Messages" on page 38.

To ensure reliable transmission of messages, the receiver must respond correctly to transmission failures. Two indications of a failed transmission exist:

- The receiver sends a NAK reply to the sender.
- The receiver does not send a reply to the sender.

Responding to a NAK reply message

The sender can receive a NAK reply for the following reasons:

- The receiver did not receive the MetroTRK data frame correctly.
- The format of the MetroTRK data frame was incorrect.

In the first instance, the sender resends the original message. In the second instance, the sender must correct any errors in the format of the data frame before resending the message. For a list of possible errors, see Table 3.3 on page 42.

Examine the code that creates and sends the data frame. If needed, change that code to correct the MetroTRK data frame before resending it.

Responding when no reply message is received

If the sender of a message receives no reply in response, an error occurred. Possible reasons for the error include a crucial part of the message (such as the start-frame or end-frame flag) being transmitted incorrectly or the receiver crashing while servicing the request.

If the sender does not receive a reply message within a reasonable amount of time, the sender must resend the original message. What is a reasonable amount of time to wait before resending?

The amount of time is a sum of the following items:

- Amount of time for the original message to traverse the physical link.
- Amount of time for the reply message to traverse the physical link.
- Amount of time for the receiver to process the request.

The first two items depend on setup of the serial connection.

The third item is the amount of time for the receiver (MetroTRK or the debugger) to send an ACK or NAK reply. This amount varies on a message-by-message basis because different requests require more or less time to complete before sending an ACK reply. However, no request requires an amount of time that is noticeable to a human.

NOTE For requests that require a substantial amount of time to process (such as the Continue and Step commands), MetroTRK sends an ACK reply before performing the request.

One-third of a second usually works well as an amount of time to wait before resending a message when using the MetroTRK. However, this amount may sometimes require adjustment.

Preventing transmission failure

To help to prevent transmission failure, a receiver can ignore any start-frame/end-frame flags that immediately follow each other. A sequence of two start-frame/endframe flags often indicates that a message was sent and badly corrupted (the start-frame or end-frame flag was lost) and a copy of the original message was resent.

Ignoring the second start-frame/end-frame flag encountered allows communications to continue in most cases. Conversely, interpreting the second flag can disrupt the current communications session irreparably.

Debug Message Interface Level

The debug message interface level defines the following messages that MetroTRK and the debugger exchange:

- Requests
- Notifications
- Reply messages

A request asks the receiving software to perform a task. A notification merely sends information to the receiving software.

For example, MetroTRK can request that the debugger read information from a file and return the information to MetroTRK by sending a ReadFile request to the debugger. MetroTRK also can send a NotifyException notification to the debugger to inform the debugger that an exception occurred on the target board.

The debugger also can send requests to MetroTRK. After receiving a request from the debugger, the MetroTRK core examines the fields of the message to determine which handler function to call. The MetroTRK core then sends the request to the corresponding handler function, which extracts any needed values from the message and executes the request. (Some messages contain values that MetroTRK passes to its handler functions as parameters. For more information, see "Debug Message Interface Reference" on page 69 and "MetroTRK Function Reference" on page 103.)

MetroTRK and the debugger send reply messages in response to each received request or notification message. Some reply messages contain only an acknowledgement and an error code; others contain additional return values. (For more information, see "Reply Messages" on page 38 and "Debug Message Interface Reference" on page 69.)

This section contains the following topics:

Request and Notification Messages

The debug message interface specifies the format of each request and notification message in terms of the fields included in a message and the arrangement of the fields. In general, a message starts with an identifier byte that identifies the message type. Zero or more arguments follow the identifier byte, depending on the message type. (For more information on the structure of each request and notification message, see "Debug Message Interface Reference" on page 69.)

This section contains the following topics:

Alignment

Message fields contain no padding for alignment purposes. When one message field ends, the next field begins on the next byte.

Byte order

Multi-byte data in debug messages uses big-endian byte order (most significant byte first). Table 3.1 shows examples of data arranged in big-endian byte order.

Table 3.1 Data in big-endian byte order

Type	Hex Value	Big-Endian Byte Stream
ui8	0x12	[0x12]
ui16	0x1234	[0x12][0x34]
ui32	0x12345678	[0x12][0x34][0x56][0x78]
ui8[]	{0x12,0x34}	[0x12][0x34]
ui16[]	{0x1234,0x5678}	[0x12][0x34][0x56][0x78]
ui32[]	{0x12345678,0x9ABCDEF0}	[0x12][0x34][0x56][0x78] [0x9A][0xBC][0xDE][0xF0]

Message length

The maximum length of a debug message is 2176 bytes. This length includes 2048 bytes for the data block when reading and writing from memory or registers (before adding escape sequences) and 128 bytes for any additional items in the message.

Reply Messages

Usually, a reply message responds to each debug message sent. Two kinds of reply messages exist: *ACK* (acknowledgment) messages and *NAK* (no acknowledgment) messages.

ACK messages confirm that the receiver correctly received the preceding message. NAK messages indicate that the receiver did not correctly receive the preceding message.

This section contains the following topics:

- ACK messages
- NAK messages

ACK messages

An ACK reply message:

- Confirms that the original message was successfully received.
- Contains an error code that specifies whether the receiver handled the original request successfully. If the receiver did not handle the request successfully, this error code specifies the problem.
- Contains any return values associated with the original request, such as register values for a ReadRegisters request.

The first byte of an ACK message is the message-type identifier, in this case the value `0x80` (`kDSReplyACK` as defined in the MetroTRK header file `msgcmd.h`).

The second byte of an ACK message is an error code specifying whether the receiver handled the request or notification correctly. If the receiver handled the original message successfully, the value of the error code byte is `0x00` (the MetroTRK constant `kDSReplyNoError`). If the error code byte contains any other value, an error occurred.

Table 3.2 lists all possible values of the error code byte in an ACK message, which are defined in `msgcmd.h`.

Table 3.2 Possible error codes in an ACK reply message

Value	Error Code Name	Description
`0x00`	`kDSReplyNoError`	The request was handled successfully.
`0x02`	`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`0x10`	`kDSReplyUnsupportedCommandError`	The request was of an invalid type. You can query MetroTRK to determine which requests are supported. For more information, "SupportMask" on page 88.

Value	Error Code Name	Description
0x11	kDSReplyParameterError	The values of one or more fields in the message were incorrect. The following messages can return this error: • ReadMemory • WriteMemory • Step
0x12	kDSReplyUnsupportedOptionError	Some requests include field values that set certain options. This error indicates that the sender passed in an unsupported option value. The following messages can return this error: • ReadRegisters • WriteRegisters • ReadMemory • WriteMemory • Step
0x13	kDSReplyInvalidMemoryRange	The specified memory range is invalid. The ReadMemory and WriteMemory messages can return this error.
0x14	kDSReplyInvalidRegisterRange	The specified register range is invalid. The ReadRegisters and WriteRegisters messages can return this error.

Value	Error Code Name	Description
0x15	kDSReplyCWDSException	An exception was generated while processing the request. The following messages can return this error: • ReadRegisters • WriteRegisters • ReadMemory • WriteMemory
0x16	kDSReplyNotStopped	Some requests are valid only when the target application is stopped. If the target application is running, the following requests reply with the error code: • Continue • FlushCache • ReadMemory • ReadRegisters • Step • WriteMemory • WriteRegisters This applies only if MetroTRK and the debugger are using interrupt-driven communication. Otherwise, MetroTRK cannot receive any messages while the target application is running.
0x03	kDSReplyCWDSError	An unknown error occurred while processing the request.

Some ACK replies contain only two bytes, the message-type identifier and the error code. Replies to requests that expect return values, however, contain additional data following the second byte. The returned data values and their format differs for each message. For more information on return values, see the descriptions of individual debug messages in "Debug Message Interface Reference" on page 69.

NAK messages

NAK messages indicate that the receiver did not correctly receive the preceding message. In most cases, the sender resends the message after receiving a NAK message. (For more information, see "Reliable Message Delivery" on page 34.)

The identifier byte (the first byte) of a NAK message is the value `0xFF` (`kDSReplyNAK` as defined in the MetroTRK header file `msgcmd.h`). The second byte of the message is an error code.

Table 3.3 lists all possible values of the error code byte in a NAK message, which are defined in `msgcmd.h`.

Table 3.3 Possible error codes in a NAK reply message

Value	Error Code Name	Description
`0x04`	`kDSReplyEscapeError`	An escape character was immediately followed by a start-frame/end-frame flag. For more information, see "Escape Sequences" on page 33.
`0x02`	`kDSReplyPacketSizeError`	The length of the received message was zero.
`0x05`	`kDSReplyBadFCS`	The contents of the MetroTRK data frame did not match the FCS checksum value. For more information, "Checksum Values" on page 29.

Value	Error Code Name	Description
0x06	kDSReplyOverflow	The message exceeded the maximum length of the buffer. By default, the maximum length of a debug message is 2176 bytes. This length includes 2048 bytes for the data block when reading and writing from memory or registers (before adding escape sequences) and 128 bytes for any additional items in the message. You can change the maximum length by changing the value of the variable kMessageBufferSize in the file msgbuf.h and recompiling MetroTRK.
0x01	kDSReplyError	Unknown problem in transmission.

4

Customizing MetroTRK

This chapter discusses customizing MetroTRK to work with new target boards.

This chapter contains the following topics:

Table 4.1 lists the customization sections in this chapter and indicates whether each section discusses a required customization. (Table 4.1 marks customizations that you must always do or must always examine and consider as required customizations.)

Table 4.1 Required MetroTRK customizations

Customization Section	Customization Required?
"Customizing MetroTRK Initializations" on page 47	Yes
"Modifying Serial Communication Functions" on page 47	Yes
"Modifying Existing UART Drivers" on page 49	Yes
"Changing the Data Transmission Rate" on page 53	Yes
"Customizing MetroTRK to Be Interrupt-Driven" on page 54	No
"Customizing the CPU Speed" on page 55	Yes
"Changing ReadMemory-Related Code" on page 56	Yes
"Changing WriteMemory-Related Code" on page 56	Yes
"Changing SupportMask-Related Code" on page 57	No
"Changing Versions-Related Code" on page 58	No
"Changing the Maximum Message Length" on page 59	No
"Customizing Memory Locations" on page 59	No
"Customizing Exception Handling" on page 60	No
"Customizing Checksum Values" on page 60	No
"Customizing the Target Board Name" on page 61	No
"Customizing usr_put_config.h for Debugging" on page 61	No

NOTE Supported reference boards work with MetroTRK without modification. For information on supported reference boards and MetroTRK implementations for each, see the *Targeting* manual for your CodeWarrior product.

Customizing MetroTRK Initializations

You can customize the MetroTRK initialization sequence for new target boards as follows:

1. **Examine the existing initialization sequence for your default implementation of MetroTRK.**
2. **If differences exist between the reference board and your target board, add to or change the contents of `__reset()`, `__init_processor`, and `__init_board` as needed.**

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

For more information, see "MetroTRK Initializations" on page 21. For an example of the type of customization that may be needed, see "Customizing Board Initialization" on page 64.

Customizing Serial Communications

Low-level communications between MetroTRK and the debugger occur over a standard serial connection.

This section contains the following topics:

Modifying Serial Communication Functions

The `UART.h` file declares a set of nine abstract functions that MetroTRK uses to send and receive serial messages. These functions are separated from the main MetroTRK code so that MetroTRK can function with new serial drivers easily.

MetroTRK provides configurable driver code for the following UARTs:

- TI TL16C552a (works with most 16552-compatible UARTs)
- Philips SCN2681/SCC2691 (works with Motorola 68681 and other xx681 UARTs)
- NEC UPD71051
- Zilog Z8530/Z85C30

In addition, many processors include an on-chip UART. When a supported reference board includes a connection for such a UART, MetroTRK usually provides applicable driver code.

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

Many UARTs are compatible with one of the preceding drivers. However, if your UART is not compatible with the supplied driver code, you must implement your own driver.

NOTE If your UART is compatible with one of the preceding drivers, see "Modifying Existing UART Drivers" on page 49.

If you are using the UART library only to support MetroTRK, you must change the following UART functions for new target boards:

- "InitializeUART()" on page 123
- "ReadUARTPoll()" on page 125
- "WriteUART1()" on page 154

However, if you are using the UART library to allow the MSL library to send output to the console, you also must change the following functions:

- "ReadUART1()" on page 126
- "TerminateUART()" on page 151

The following file prototypes the UART functions discussed in this section:

`Export/UART.h`

NOTE For information on the MSL library to use with your target board, see the *Targeting* manual for your target processor.

Modifying Existing UART Drivers

MetroTRK provides configurable driver code for the following UARTs:

- TI TL16C552a (works with most 16552-compatible UARTs)
- Philips SCN2681/SCC2691 (works with Motorola 68681 and other xx681 UARTs)
- NEC MPD71051
- Zilog Z8530/Z85C30

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

Many UARTs are compatible with one of the preceding drivers. However, if your UART is not compatible with the supplied driver code, you must implement your own driver.

NOTE If you need to implement your own driver, see "Modifying Serial Communication Functions" on page 47.

This section, which discusses how to modify and build two of the existing drivers to work with a target board, contains the following topics:

Building the TI TL16C552a UART driver

This driver is located in the following directory:

`Transport/uart/tl16c552a`

To build the driver:

1. **Include the relevant driver files in a separate library project that builds a library. (You will include the resulting library in your MetroTRK project.)**

   Include the following driver files:

   - `tl16c552a.c` (main driver code)
   - One of the following two files:
     - `tl16c552a_A.c` (for channel A of dual-channel UARTs)
     - `tl16c552a_B.c` (for channel B of dual-channel UARTs)
   - `uart.c`
   - `board_stub.c`

2. **Copy the driver configuration file to your local project directory.**

   Copy `tl16c552a_config_sample.h` to your local project directory.

3. **Rename the driver configuration file.**

   Rename `tl16c552a_config_sample.h` as follows:

   `tl16c552a_config.h`

4. **Include the renamed driver configuration file in your library project.**

   Include `tl16c552a_config.h` in your library project.

5. **Modify the constant values in the driver configuration file.**

   Change the constant values in `tl16c552a_config.h` as needed for your target board.

NOTE You can use `tl16c552a_config.h` to define items such as the base addresses of the two serial ports, the speed of the external UART clock, and the spacing (in bytes) between UART registers.

Examine `tl16c552a_config.h` to determine which, if any, changes to make for your target board.

6. **Ensure that the UART header file is in your include path.**

   Ensure that the following file is in your include path:

   ```
   Export/UART.h
   ```

7. **Build your library project.**

   After you build your library project, you must add the library to your MetroTRK project before building it.

NOTE For information on building projects, see the *IDE User Guide*.

Building the Philips SCN2681/SCC2691 UART driver

This driver is located in the following directory:

```
Transport/uart/scx26x1
```

To build the driver:

1. **Include the relevant driver files in a separate library project that builds a library. (You will include the resulting library in your MetroTRK project.)**

   Include the following driver files:

   - `SCx26x1.c` (main driver code)
   - One of the following two files:
     - `SCN2681_A.c` (for channel A of dual-channel UARTs)
     - `SCN2681_B.c` (for channel B of dual-channel UARTs)
   - `uart.c`
   - `board_stub.c`

2. **Copy the driver configuration file to your local project directory.**

   Copy `SCx26x1_config_sample.h` to your local project directory.

3. **Rename the driver configuration file.**

   Rename `SCx26x1_config_sample.h` as follows:

   `SCx26x1_config.h`

4. **Include the renamed driver configuration file in your library project.**

   Include `SCx26x1_config.h` in your library project.

5. **Modify the constant values in the driver configuration file.**

   Change the constant values in `SCx26x1_config.h` as needed for your target board.

NOTE You can use `SCx26x1_config.h` to define items such as the base address of the UART in memory, whether input and output are interrupt-driven, and the spacing (in bytes) between UART registers. Examine `SCx26x1_config.h` to determine which, if any, changes to make for your target board.

6. **Ensure that the UART header file is in your include path.**

   Ensure that the following file is in your include path:

   `Export/UART.h`

7. **Build your library project.**

   After you build your library project, you must add the library to your MetroTRK project before building it.

NOTE For information on building projects, see the *IDE User Guide*.

Changing the Data Transmission Rate

MetroTRK can communicate with the debugger at transmission rates between 300 baud and 230.4k baud. To change the data transmission rate (baud rate), set the transmission rate at compile time by setting the constant `TRK_BAUD_RATE` to a value of the enumerated type `UARTBaudRate`. Then rebuild MetroTRK.

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

Set the data transmission rate to the fastest speed that your UART can use without losing characters. If your board and UART driver support hardware or software flow control, set `TRK_BAUD_RATE` to the maximum data transmission rate for the UART. However, if you experience problems while using MetroTRK, try lowering the data transmission rate.

NOTE `UARTBaudRate` is defined in `UART.h`. In the default implementation, `TRK_BAUD_RATE` is defined in the file `target.h`.

You also must set the debugger to communicate at the same data transmission rate as MetroTRK. For more information, see the *Targeting* manual for your target processor and the processor-specific appendixes in this manual.

NOTE The maximum data transmission rate for the Solaris-hosted CodeWarrior debugger is 38.4 KB. Consequently, if you are using the Solaris-hosted debugger, you must set the data transmission rate in MetroTRK to 38.4 KB, even if the target board accepts a faster rate.

Customizing MetroTRK to Be Interrupt-Driven

Depending on the target board, MetroTRK uses either serial polling or interrupt-driven communication to respond to messages sent by the debugger. Interrupt-driven communication is the default communication method for target boards for which MetroTRK currently supports that communication method.

When using serial polling, MetroTRK does not respond to messages from the debugger while the target application is running. However, when using interrupt-driven communication, MetroTRK responds as follows to an interrupt received from the debugger:

1. Stops the target application from running.
2. Places the data from the serial line in a message buffer.
3. Checks whether the received message is a request or a notification.

NOTE Rather than accessing the message buffer directly, MetroTRK calls `UARTInterruptHandler()` when a UART interrupt occurs and `ReadUARTPoll()` when ready to receive input.

4. Resumes running the target application.
5. For a request, executes the request received from the debugger unless the request cannot execute while a target application executes. In this case, MetroTRK returns an error, and the debugger must stop the target application before resending the request.

If MetroTRK currently does not support interrupt-driven communication for a particular target board, you can customize MetroTRK to do so.

For any target board for which MetroTRK uses interrupt-driven communication, set the value of `TRK_TRANSPORT_INT_DRIVEN` to 1. In addition, depending on the target board, you may need to define a transport interrupt key so that MetroTRK can identify the interrupt that corresponds to the communication transport.

Ensure that the serial driver that you are using supports interrupt-driven serial input. If you wrote your own driver code, you may have to modify it. To modify your driver code, you must create your

own implementation of the following functions for your target board:

- "InitializeIntDrivenUART()" on page 122
- "UARTInterruptHandler()" on page 152

You can refer to the driver code provided with some of the supported reference boards as examples. (For more information and, for some processors, tips on implementing interrupt-driven communication, see the processor-specific appendixes in this manual.)

Customizing the CPU Speed

The value of the `CPU_SPEED` constant in `target.h` indicates the CPU speed of the target board. Set the value of the `CPU_SPEED` constant to the appropriate speed for your board. If you do not know the appropriate speed or the speed is variable, a value greater than the maximum speed is acceptable. For more information, see "Changing the CPU Speed" on page 66.

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

Customizing Debug Services

MetroTRK provides debug services using a debug message interface that consists of debug requests and debug notifications. (For more information, see "Debug Message Interface Reference" on page 69.)

Some debug messages require related code changes in MetroTRK so that MetroTRK can work with your new target board. You also can perform some optional customizations.

This section discusses several customizations that you can perform, which are related to the debug message interface:

- Changing ReadMemory-Related Code
- Changing WriteMemory-Related Code

- Changing SupportMask-Related Code
- Changing Versions-Related Code
- Changing the Maximum Message Length

NOTE If you are customizing MetroTRK, ensure that you implement all messages in the primary command set. For more information, see "Command Sets" on page 69.

Changing ReadMemory-Related Code

After receiving a ReadMemory request, MetroTRK reads the specified section of memory and returns the result. To perform this task, MetroTRK calls `TargetAccessMemory()` to read memory from the board.

The `TargetAccessMemory()` function calls another function, `ValidMemory32()`, that checks whether the addresses to read are valid for the target board. The `ValidMemory32()` function uses a global variable, `gMemMap`, to determine which memory ranges are valid. To customize memory checks, redefine `gMemMap` in the `memmap.h` file.

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

For more information, see "ReadMemory" on page 77, "TargetAccessMemory()" on page 132, and "ValidMemory32()" on page 153.

Changing WriteMemory-Related Code

After receiving a WriteMemory request, MetroTRK writes the specified data in memory at the specified address. To perform this task, MetroTRK calls the function `TargetAccessMemory()` to write to memory.

The `TargetAccessMemory()` function calls another function, `ValidMemory32()`, that checks whether the addresses to write to are valid for the target board. The `ValidMemory32()` function

uses a global variable, `gMemMap`, to determine which memory ranges are valid. To customize memory checks, redefine `gMemMap` in the `memmap.h` file.

NOTE This is a required customization. For more information, see Table 4.1 on page 46.

For more information, see "WriteMemory" on page 92, "TargetAccessMemory()" on page 132, and "ValidMemory32()" on page 153.

Changing SupportMask-Related Code

After receiving a SupportMask request, MetroTRK calls the `TargetSupportMask()` function. The `TargetSupportMask()` function uses a set of board-specific variables defined in the following file to determine which debug messages your customized version of MetroTRK supports:

`Portable\default_supp_mask.h`

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

Thirty-two compile-time variables exist; each variable is 8 bits wide. Each variable is a bit-vector where each bit represents one message in the debug message interface. The first variable, `DS_SUPPORT_MASK_00_07`, represents the first eight messages, those with numbers 0x00 through 0x7. The second variable, `DS_SUPPORT_MASK_08_0F`, represents the next eight messages and so on through `DS_SUPPORT_MASK_F8_FF`, which represents messages 248 through 255.

You can remove support for debug messages that your implementation of MetroTRK does not support by changing the value of the variables. (Changing the value of a variable changes the value of the individual bits that correspond to the various debug messages.)

To customize the value of the variables, cut and paste the variable definitions from `default_supp_mask.h` to `target.h` and change the definitions as needed.

NOTE You also can add support for additional messages by changing this set of board-specific variables. This ability is useful only if you are implementing your own debugger.

For more information, see "SupportMask" on page 88, "DoSupportMask()" on page 117, and "TargetSupportMask()" on page 149.

Changing Versions-Related Code

The Versions request causes MetroTRK to return the major and minor version numbers for MetroTRK and for the messaging protocol.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

The default implementation of MetroTRK (through the `TargetVersions()` function) uses compile-time constants to specify the version numbers. To customize MetroTRK, modify the constants.

The following constants, which specify the version numbers of the kernel, reside in a processor-specific file:

- `DS_KERNEL_MAJOR_VERSION`
- `DS_KERNEL_MINOR_VERSION`

NOTE For more information, see the processor-specific appendixes in this manual.

The following constants, which specify the version numbers of the protocol, reside in `msgcmd.h`:

- `DS_PROTOCOL_MAJOR_VERSION`
- `DS_PROTOCOL_MINOR_VERSION`

NOTE If you are using the CodeWarrior debugger, do not change the protocol version numbers.

For more information, see "Versions" on page 90, "DoVersions()" on page 118, and "TargetVersions()" on page 150.

Changing the Maximum Message Length

By default, the maximum length of a debug message is 2176 bytes. This length includes 2048 bytes for the data block when reading and writing from memory or registers (before adding escape sequences) and 128 bytes for any additional items in the message.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

You can change the maximum length by changing the value of the variable `kMessageBufferSize` in the file `msgbuf.h` and recompiling MetroTRK.

NOTE If you are using the CodeWarrior debugger, do not change the maximum message length.

Customizing Memory Locations

You can customize the memory locations of both MetroTRK and of your target application.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

Depending on your target processor, you can customize the memory locations used by MetroTRK by:

- Modifying variables in the Linker target settings panel in your MetroTRK project
- Modifying the linker command file in your MetroTRK project

For more information, see the processor-specific appendixes in this manual and the *Targeting* manual for your target processor.

To change the location of target application memory sections, modify your linker command file. The linker command file is the file in your project with the extension `.lcf`. For more information, see the *Targeting* manual for your target processor.

Customizing Exception Handling

You can customize exception handling by overriding the default exception-handling code so that your application handles some exceptions. MetroTRK must handle certain exceptions; in those cases, your application must accommodate MetroTRK if the application also must handle those particular exceptions. For more information, see the processor-specific appendixes in this manual.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

Customizing Checksum Values

By default, MetroTRK uses a one-byte checksum value for error-checking when it frames messages. (For more information, see "Checksum Values" on page 29.)

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

However, you can specify that the MetroTRK use a two- or four-byte checksum value to increase the probability of finding

transmission errors. To do so, change the value of the `FCSBITSIZE` variable in the following file and recompile the MetroTRK:

`{MetroTRK directory}\Export\serframe.h`

NOTE If you are using the CodeWarrior debugger, do not change the length of the checksum value.

Using a two- or four-byte checksum value requires:

- More computation time when creating and verifying the checksum values.
- Global data space for a lookup table. (A two-byte checksum value requires 512 bytes; a four-byte checksum value requires 1024 bytes.)

Customizing the Target Board Name

The name of the target board displays in the startup welcome message for MetroTRK. To customize the target board name, define the constant `DS_TARGET_NAME` in `target.h` as a string value that reflects the name of your target board. For more information, see "Changing the Target Board Name" on page 65.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

Customizing usr_put_config.h for Debugging

The `usr_put_config.h` file defines values that are useful when debugging MetroTRK.

NOTE This customization is not required. For more information, see Table 4.1 on page 46.

You can customize the following values:

- `DEBUGIO_SERIAL`

When you define `DEBUGIO_SERIAL`, MetroTRK writes the debug information to a serial port. MetroTRK uses the same serial port as the MetroTRK requests and notifications. Consequently, this option works best when debugging manually.

NOTE Usually, you define only `DEBUGIO_SERIAL` when debugging the MetroTRK.

- `DEBUGIO_RAM`

  When you define `DEBUGIO_RAM`, MetroTRK writes the debug information to a RAM buffer on the target board. If you define `DEBUGIO_RAM`, you also must define the following values, which also reside in `usr_put_config.h`:

  - `DB_START`

    Defines the start of the RAM buffer.

  - `DB_END`

    Defines the end of the RAM buffer.

- `DB_RAM_CONSOLE_DUMP`

  If you define `DEBUGIO_RAM`, you can periodically dump the contents of the RAM buffer to the console (`stdout`) by defining `DB_RAM_CONSOLE_DUMP`. This ability is useful when using the host plug-in to drive MetroTRK because the debugging output displays in the console window of the debugger.

NOTE MetroTRK currently performs a console dump only when the debugger instructs MetroTRK to step or execute the target application. In between, MetroTRK stores debugging output until the next opportunity to dump it.

5

MetroTRK Porting Example

This chapter presents an example that ports MetroTRK to a 68K target board (M68328 ADS). The following topics describe the porting process for this target board:

NOTE This chapter presents one example of how to customize MetroTRK for a particular target board. For other target boards, the steps required may differ. For more information, see "Customizing MetroTRK" on page 45.

Copying an Existing MetroTRK Configuration

Copy an existing MetroTRK configuration for a particular board. For this example, the following directory was copied to a new name:

```
Processor/M68K/Board/motorola/m68328_ads
```

NOTE This chapter refers to files in the preceding directories according to their original locations. However, create copies and modify the copies rather than the original files.

Customizing Board Initialization

For the 68K example target board, changes were made to the following file:

`Processor/M68K/Board/motorola/m68328_ads/Reset.s`

The following changes were made:

- The value of `ROMADDR` (the first word in the boot ROM) was changed to the address of the boot ROM after initialization completes.
- The value of `MON_STACKTOP` (the second word in the boot ROM) was changed to a value that did not overlap with interrupt vectors or other important data.

  MetroTRK uses the value set for `MON_STACKTOP` in `Reset.s` only during the initialization phase. The compiler runtime code assigns a new stack pointer after initialization completes.

Each board has its own set of peripherals and its own memory map. To boot from ROM, MetroTRK must initialize the memory controller to access RAM and prevent other peripherals on the board from causing spurious interrupts.

For the example target board, the initialization code entry point is labeled as follows:

```
__reset
```

NOTE For more information, see "MetroTRK Initializations" on page 21 and "Customizing MetroTRK Initializations" on page 47.

The initialization code completes and then jumps to the following entry point of the runtime code to start the MetroTRK:

```
__start
```

For the example target board, the first two words in the boot ROM must specify the initial PC (the address of the initialization code entry point) and an initial stack pointer. The address of the boot ROM is typically 0 at reset time, but the address can change after the

initialization code is run (for example, $400000 for the 68328 ADS board).

You must link MetroTRK so that absolute jumps work. If you use a symbol (such as `__reset`) to specify the initial PC value, its address at reset time is offset from the address provided by the linker.

Also, to ensure that these two words are the first items in the ROM image, the file that contains them was placed first in the link order of the project. For information on specifying the link order for a project, see the *IDE User Guide*.

Customizing MetroTRK Version Numbers

The MetroTRK version numbers display in the startup welcome message for MetroTRK. For the example target board, `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` were changed in the following file:

`Processor\M68k\Generic\m68k_version.h`

Define these constants to your preferred values.

For more information, see "Changing Versions-Related Code" on page 58.

Changing the Target Board Name

The name of the target board displays in the startup welcome message for MetroTRK. For the example target board, these values were changed in the `target.h` file by setting the value of the `DS_TARGET_NAME` constant as follows:

```
#define DS_TARGET_NAME    "M68328 ADS"
```

For more information, see "Customizing the Target Board Name" on page 61.

Customizing the Data Transmission Rate

The value of the `TRK_BAUD_RATE` constant in `target.h` determines the data transmission rate (baud rate) used by MetroTRK. For the example target board, the value was changed to kBaud38400.

Set the data transmission rate to the fastest speed that your UART can use without losing characters. For more information, see "Changing the Data Transmission Rate" on page 53.

Changing the CPU Speed

The value of the `CPU_SPEED` constant in `target.h` indicates the CPU speed of the target board. For the example target board, `CPU_SPEED` was changed as follows:

```
#define CPU_SPEED        33 * 1000000
```

For more information, see "Customizing the CPU Speed" on page 55.

Customizing MetroTRK Memory Locations

The memory locations for MetroTRK were customized in the following linker command file:

`Processor/M68K/Board/motorola/m68328_ads/trk68k_rom.lcf`

The following changes were made:

- The ROM address was set to the start address of the boot ROM as follows:

```
$segment ROM 0x00400000 LENGTH 0x00040000
{
  *(.text)
}
```

- The RAM address was set to a location that does not overlap with interrupt vectors or programs that you are debugging as follows:

```
$segment RAM 0x00070000 R
{
  *(.data)
  *(.sdata)

}
```

  Usually, the RAM address is set to the following:

```
<end_of_RAM> - 0x6000
```

  Setting the address this way places MetroTRK in RAM so that it does not overlap with your target application and allows enough space for the MetroTRK data and stack.

For more information, see "Customizing Memory Locations" on page 59.

Customizing UART Drivers

For the example board, the Philips SCN2681/SCC2691 driver was built as described in "Modifying Existing UART Drivers" on page 49. However, this particular board did not require changes to `SCx26x1_config.h`.

A

Debug Message Interface Reference

This appendix describes the debug message interface, that is, the set of debug messages that the debugger and MetroTRK use to communicate.

This appendix contains the following topics:

Command Sets

Each message described in this appendix belongs to either the *primary command set* (level 1) or the *extended command set* (level 2), which the description of the command indicates. To function properly, the debugger requires all messages in the primary command set. However, messages in the extended command set, while useful, are optional.

NOTE If you are customizing MetroTRK, ensure that you implement all messages in the primary command set.

Messages Sent by the Debugger

This section describes the messages that the debugger can send to MetroTRK, which are all requests. The debugger can send the following messages:

- Connect
- Continue
- CPUType
- FlushCache
- Reset
- ReadMemory
- ReadRegisters
- Step
- Stop
- SupportMask
- Versions
- WriteMemory
- WriteRegisters

NOTE See the following file for more information about these messages, definitions of all `MessageCommandID` values, and message-specific constants:

`Export\msgcmd.h`

The message descriptions include such information as fields sent in the original message, the name of the handler function for the message, and any return values.

NOTE MetroTRK and the debugger place return values for a request in a separate reply message. For more information, see "Reply Messages" on page 38.

Connect

Requests that MetroTRK begin a debug session.

Command Set Primary command set (level 1).

Fields This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSConnect` (defined in the `msgcmd.h` file).

Return Values None.

Error Codes None.

Remarks The debugger sends this request once at the beginning of each debug session.

Handler Function `DoConnect()`

See Also "DoConnect()" on page 106

Continue

Requests that MetroTRK start running the target application.

Command Set — Primary command set (level 1).

Fields — This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSContinue` (defined in the `msgcmd.h` file).

Return Values — None.

Error Codes — MetroTRK can return the following error code:

Error Code	Description
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.

Remarks — The debugger sends this request to tell MetroTRK to resume executing the target application. After receiving a Continue request, MetroTRK returns to the event-waiting state, swaps in the context of the target application, and resumes executing the target application. The target application runs until a relevant exception occurs. For more information, see "MetroTRK Execution States" on page 16.

Handler Function — `DoContinue()`

See Also — "DoContinue()" on page 107

CPUType

Requests that MetroTRK return CPU-related information for the target board.

Command Set — Extended command set (level 2).

Fields — This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSCPUType` (defined in the `msgcmd.h` file).

Return Values — This message causes MetroTRK to return the following values:

Return Value Field	Size	Description
`cpuMajor`	`ui8`	The major CPU type, which indicates the processor family of the target board.
`cpuMinor`	`ui8`	The minor CPU type, which indicates the particular processor within the processor family.
`bigEndian`	`ui8`	A value of 1 indicates that the board uses the big-endian byte order; a value of 0 indicates that the board uses the little-endian byte order.
`defaultTypeSize`	`ui8`	The size of the registers in the default register block.
`fpTypeSize`	`ui8`	The size of the registers in the floating-point register block. If there are no floating-point registers, this return value is 0.

Return Value Field	Size	Description
`extended1TypeSize`	`ui8`	The size of the registers in the first block of extended registers. If there are no extended registers, this return value is 0.
`extended2TypeSize`	`ui8`	The size of the registers in the second block of extended registers. If there is no second block of extended registers, this return value is 0.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`kDSReplyCWDSError`	An unknown error occurred while processing the request.

Handler Function

`DoCPUType ( )`

See Also

"DoCPUType()" on page 108

FlushCache

Requests that MetroTRK flush all cache entries corresponding to the specified memory range and possibly others, depending on the particular target board. (For more information, see your default implementation of MetroTRK.)

Command Set

Secondary command set (level 2).

Fields

This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSFlushCache` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain the following values, which specify the type of cache to flush: • `DS_MSG_CACHE_TYPE_INSTRUCTION` • `DS_MSG_CACHE_TYPE_DATA` • `DS_MSG_CACHE_TYPE_SECONDARY` For more information, see the `msgcmd.h` file.
`start`	`ui32`	The starting address of the specified memory section in the cache.
`end`	`ui32`	The end address of the specified memory section in the cache.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyPacketSizeError`	The length of the message does not equal the minimum for a message of that type.
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyInvalidMemoryRange`	The specified memory range is invalid.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.
`kDSReplyCWDSError`	An unknown error occurred while processing the request.

Remarks

To flush more than one type of cache, the debugger can OR multiple values before adding the `options` field to the message.

Handler Function

`DoFlushCache()`

See Also

"DoFlushCache" on page 109
`msgcmd.h`

ReadMemory

Requests that MetroTRK read a specified section of memory on the target board.

Command Set — Primary command set (level 1).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSReadMemory` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` For more information, see the `msgcmd.h` file.
`length`	`ui16`	The length of the memory section (a maximum of 2048 bytes).
`start`	`ui32`	The starting address of the memory section.

Return Values — This message causes MetroTRK to return the following values:

Return Value Field	Size	Description
`length`	`ui16`	The length of the data read (a maximum of 2048 bytes).
`data`	`ui8[]`	The data read (a maximum of 2048 bytes).

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyCWDSException`	An exception was thrown while processing the request.
`kDSReplyInvalidMemoryRange`	The specified memory range is invalid.
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`kDSReplyParameterError`	The value of the `length` field is greater than 2048 or the value of the `length` field is not equal to the size of the data field.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.

Remarks

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

Handler Function

`DoReadMemory()`

See Also

"DoReadMemory()" on page 111
`msgcmd.h`

ReadRegisters

Requests that MetroTRK read a specified sequence of registers on the target board.

Command Set

Primary command set (level 1).

Fields

This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSReadRegisters` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `kDSRegistersDefault` • `kDSRegistersFP` • `kDSRegistersExtended1` • `kDSRegistersExtended2` For more information, see the `msgcmd.h` file.
`firstRegister`	`ui16`	The number of the first register in the sequence.
`lastRegister`	`ui16`	The number of the last register in the sequence.

Return Values

This message causes MetroTRK to return the following value:

Return Value Field	Size	Description
`registerData`	`void*`	An array of register values. The size of each element depends on the size of the registers themselves. If the registers are 2 bytes wide, then a new value starts every 2 bytes. If the registers are 4 bytes wide, a new value starts every 4 bytes. The maximum length of this array is 2048 bytes.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyCWDSException`	An exception was thrown while processing the request.
`kDSReplyInvalidRegisterRange`	The specified register range is invalid.
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.

Remarks

After receiving a ReadRegisters request, MetroTRK reads the specified sequence of registers from the processor, returning the resulting values to the debugger. MetroTRK attempts to catch and handle any access exceptions that occur while reading.

NOTE For information on registers, see the processor-specific appendixes in this manual.

Handler Function

`DoReadRegisters()`

See Also

"DoReadRegisters()" on page 112
`msgcmd.h`

Reset

Requests that MetroTRK reset the target board.

Command Set — Extended command set (level 2).

Fields — This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSReset` (defined in the `msgcmd.h` file).

Return Values — None.

Error Codes — None.

Remarks — After receiving a Reset request, MetroTRK calls its own reset code. MetroTRK restarts and performs all hardware initializations as if the board were being manually reset.

Handler Function — `DoReset()`

See Also — "DoReset()" on page 114

Step

Requests that MetroTRK let the target application run a specified number of instructions or, alternatively, until the PC (program counter) is outside a specified range of values.

Command Set

Extended command set (level 2).

Fields

The fields in this message differ depending on the value of the `options` field. If the value of the `options` field is `kDSStepIntoCount` or `kDSStepOverCount`, the message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSStep` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `kDSStepIntoCount` • `kDSStepOverCount` For more information, see the `msgcmd.h` file.
`count`	`ui8`	The number of instructions to step over.

If the value of the `options` field is `kDSStepIntoRange` or `kDSStepOverRange`, the message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSStep` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `kDSStepIntoRange` • `kDSStepOverRange` For more information, see the `msgcmd.h` file.
`rangeStart`	`ui32`	The starting address of the specified memory range.
`rangeEnd`	`ui32`	The end address of the specified memory range.

Return Values

None.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.

Error Code	Description
`kDSReplyParameterError`	If the debugger is single stepping, this error code indicates that the value of the `count` field is less than one. (The debugger must step over one or more instructions.) If the debugger is stepping out of range, this error code indicates that the PC (program counter) is already outside the range specified by the rangeStart and rangeEnd fields.
`kDSReplyUnsupported OptionError`	The specified value of the `options` field is unsupported.

Remarks

After receiving a Step request, MetroTRK steps through one or more instructions.

This message specifies

- whether MetroTRK steps through a specified number of instructions or through all remaining instructions within a specified memory range
- whether MetroTRK steps over or into function calls

If the value of the `options` parameter is `kDSStepIntoCount` or `kDSStepOverCount`, MetroTRK steps through `count` instructions in the target application and then returns control to the host. If the value of the `options` parameter is `kDSStepIntoRange` or `kDSStepOverRange`, MetroTRK continues running the program until it encounters an instruction whose address is outside the range specified by `rangeStart` and `rangeEnd`. MetroTRK then returns control to the host.

MetroTRK notifies the debugger that the end condition was reached by sending a NotifyStopped notification. For more information, see "NotifyStopped" on page 98.

Using `kDSStepOverCount` and `kDSStepOverRange` causes function calls to be counted as a single instruction. In other words,

MetroTRK does not evaluate instructions executed within a called function for the end condition of the step. Omitting the evaluation is called *stepping over* a function.

The following example shows some sample code:

```
i = 10;
i++;
DoSomeProcessing(i);
i--;
```

Assume the execution process is at the first line of the preceding code (`i = 10;`) and that each line corresponds to a single machine instruction. In that case, a request to step over four instructions causes MetroTRK to step past the final line of the preceding code (`i--;`). The number of lines executed in the `DoSomeProcessing`() function does not affect how many lines MetroTRK steps through in the main flow of execution.

NOTE One line of code in a high-level language such as C or C++ sometimes corresponds to more than one machine instruction.

When the debugger specifies `kDSStepIntoCount` or `kDSStepIntoRange`, MetroTRK does evaluate instructions within a called function for the end condition of the step. Evaluating the instructions within the function is called *stepping into* the function.

Handler Function `DoStep()`

See Also "DoStep()" on page 115
"NotifyStopped" on page 98
`msgcmd.h`

Stop

Requests that MetroTRK stop running the target application.

Command Set Extended command set (level 2).

Fields This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSStop` (defined in the `msgcmd.h` file).

Return Values None.

Error Codes MetroTRK can return the following error code:

Error Code	Description
`kDSReplyError`	Unknown problem in transmission.

Remarks None.

Handler Function `DoStop()`

See Also "DoStop()" on page 116

SupportMask

Requests that MetroTRK return a list of supported messages.

Command Set — Primary command set (level 1).

Fields — This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSSupportMask` (defined in the `msgcmd.h` file).

Return Values — This message causes MetroTRK to return the following values:

Return Value Field	Size	Description
`mask`	`ui8[32]`	A bit-array of 32 bytes, where each bit corresponds to the message (which is of type `MessageCommandID`) with an ID matching the position of the bit in the array.
`protocolLevel`	`ui8`	The protocol level supported by MetroTRK. For more information, see "Command Sets" on page 69.

Error Codes — MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.

Remarks

If the value of a bit in the `mask` return value field is 1, the message is available; if the value of the bit is 0, the message is not available. For example, if `kDSReset` is available, the value of the fourth bit is 1 because `kDSReset` is the fourth message.

For more information, see `msgcmd.h`. Also, for information on how the default values are set, see `target_supp_mask.h` and "Changing SupportMask-Related Code" on page 57.

Handler Function

`DoSupportMask()`

See Also

"DoSupportMask()" on page 117
`msgcmd.h`

Versions

Requests that MetroTRK return version information.

Command Set — Primary command set (level 1).

Fields — This message contains the following field:

Field	Size	Description
`command`	`ui8`	`kDSVersions` (defined in the `msgcmd.h` file).

Return Values — This message causes MetroTRK to return the following values:

Return Value Field	Size	Description
`kernelMajor`	`ui8`	The major version number for MetroTRK. (In version 1.2, the `kernelMajor` is 1.)
`kernelMinor`	`ui8`	The minor version number for MetroTRK. (In version 1.2, the `kernelMinor` is 2.)
`protocolMajor`	`ui8`	The major version number for the messaging protocol. (In version 1.2, the `protocolMajor` is 1.)
`protocolMinor`	`ui8`	The minor version number for the messaging protocol. (In version 1.2, the `protocolMinor` is 2.)

Error Codes — MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.

Handler Function `DoVersions()`

See Also "DoVersions()" on page 118

WriteMemory

Requests that MetroTRK write data to a specified memory location.

Command Set — Primary command set (level 1).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSWriteMemory` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `DS_MSG_MEMORY_SEGMENTED` • `DS_MSG_MEMORY_PROTECTED` • `DS_MSG_MEMORY_USERVIEW` For more information, see the `msgcmd.h` file.
`length`	`ui16`	The length of the data (a maximum of 2048 bytes).
`start`	`ui32`	The starting address of the destination in memory.
`data`	`ui8[]`	The data to write (a maximum of 2048 bytes).

Return Values — This message causes MetroTRK to return the following value:

Return Value Field	Size	Description
`length`	`ui16`	The amount of memory written.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyCWDSException`	An exception was thrown while processing the request.
`kDSReplyInvalidMemoryRange`	The specified memory range is invalid.
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`kDSReplyParameterError`	The value of the `length` field is greater than 2048 or the value of the `length` field is not equal to the size of the data field.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.

Remarks

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

Handler Function

`DoWriteMemory()`

See Also

"DoWriteMemory()" on page 119
`msgcmd.h`

WriteRegisters

Requests that MetroTRK write data to a specified sequence of registers.

Command Set — Primary command set (level 1).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSWriteRegisters` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of the following values: • `kDSRegistersDefault` • `kDSRegistersFP` • `kDSRegistersExtended1` • `kDSRegistersExtended2` For more information, see the `msgcmd.h` file.
`firstRegister`	`ui16`	The number of the first register in the sequence.
`lastRegister`	`ui16`	The number of the last register in the sequence.
`registerData`	`ui32[]`	An array of register values. The size of each element depends on the size of the registers. If the registers are 2 bytes wide, then a new value starts every 2 bytes. If the registers are 4 bytes wide, a new value starts every 4 bytes. The maximum length of this array is 2048 bytes.

Return Values — None.

Error Codes

MetroTRK can return the following error codes:

Error Code	Description
`kDSReplyCWDSError`	An unknown error occurred while processing the request.
`kDSReplyCWDSException`	An exception was thrown while processing the request.
`kDSReplyInvalidRegisterRange`	The specified register range is invalid.
`kDSReplyNotStopped`	The target application is running and must be stopped before the debugger issues this request.
`kDSReplyPacketSizeError`	The length of the message is less than the minimum for a message of that type.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.

Remarks

After receiving a WriteRegisters request, MetroTRK writes the specified data into the specified register sequence. MetroTRK attempts to catch and handle any access exceptions that occur while writing.

NOTE For information on registers, see the processor-specific appendixes in this manual.

Handler Function

`DoWriteRegisters()`

See Also

"DoWriteRegisters()" on page 120
`msgcmd.h`

Messages Sent by MetroTRK

This section describes the messages that MetroTRK can send to the debugger. Some messages are notifications; others are requests. MetroTRK can send the following messages:

- NotifyException
- NotifyStopped
- ReadFile
- WriteFile

The message descriptions include such information as fields sent in the original message and return values.

NOTE The message descriptions omit acknowledgements and error codes because the debugger always returns them.

NotifyException

Notifies the debugger that an exception occurred on the target board.

Command Set — Primary command set (level 1).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSNotifyException` (defined in the `msgcmd.h` file).
`target-defined info`	`target-specific`	The value of this field, which provides state information about the target board, differs for each target processor. Usually, this field contains information such as the value of the PC (program counter), the corresponding instruction, and the exception ID. For more information, see "TargetAddExceptionInfo()" on page 134.

Return Values — None.

Remarks — None.

See Also
"DoNotifyStopped()" on page 110
"TargetAddExceptionInfo()" on page 134
"TargetAddStopInfo()" on page 135
"TargetInterrupt()" on page 138

NotifyStopped

Notifies the debugger that MetroTRK reached a breakpoint or completed a step command.

Command Set — Primary command set (level 1).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSNotifyStopped` (defined in the `msgcmd.h` file).
target-defined info	target-specific	The value of this field, which provides state information about the target board, differs for each target processor. Usually, this field contains information such as the value of the PC (program counter) and the corresponding instruction. For more information, see "TargetAddStopInfo()" on page 135.

Return Values — None.

Remarks — None.

See Also — "TargetInterrupt()" on page 138
"TargetAddStopInfo()" on page 135

ReadFile

Requests that the debugger read data from a file (for the target application). If the file is `stdin`, the data is input from a console window.

Command Set — Extended command set (level 2).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSReadFile` (defined in the `msgcmd.h` file).
`file_handle`	`ui32`	The handle of the file to read. (`stdin` has a predefined handle.) For more information, see the `DSFileHandle` definition in `msgcmd.h`.
`length`	`ui16`	The length of the data to read from the file (a maximum of 2048 bytes).

Return Values — This message causes the debugger to return the following values:

Return Value Field	Size	Description
`io_result`	`ui8`	Standard I/O result returned by the debugger (`kDSIONoError`, `kDSIOError`, or `kDSIOEOF`).
`length`	`ui16`	The amount of data read.
`file_data`	`ui8[]`	The data (a maximum of 2048 bytes).

Remarks — The debugger can return less data than requested (but not more). For example, a console read request usually returns as soon as the user presses Enter. After receiving the requested data from the debugger, MetroTRK passes the data to the target application.

See Also `mslsupp.c`
`targsupp.h`
“SuppAccessFile()” on page 129

WriteFile

Requests that the debugger write data from the target application to a file. If the file is `stdout` or `stderr`, a console window displays the data.

Command Set — Extended command set (level 2).

Fields — This message contains the following fields:

Field	Size	Description
`command`	`ui8`	`kDSWriteFile` (defined in the `msgcmd.h` file).
`file_handle`	`ui32`	The handle of the file to write. (`stdout` and `stderr` have predefined handles.) For more information, see the `DSFileHandle` definition in `msgcmd.h`.
`length`	`ui16`	The length of the data to write to the file (a maximum of 2048 bytes).
`file_data`	`ui8[]`	The data (a maximum of 2048 bytes).

Return Values — This message causes the debugger to return the following values:

Return Value Field	Size	Description
`io_result`	`ui8`	Standard I/O result returned by the debugger (`kDSIONoError`, `kDSIOError`, or `kDSIOEOF`).
`length`	`ui16`	The amount of data written.

Remarks — The startup welcome message is sent as a `kDSWriteFile` message, but it is a special case and does not require a reply.

See Also — `mslsupp.c`
`targsupp.h`
"SuppAccessFile()" on page 129

B

MetroTRK Function Reference

This appendix describes MetroTRK functions that may be relevant if you are customizing MetroTRK for new target boards. You may have to change functions that this appendix identifies as board-specific for new target boards.

This appendix describes the following functions:

__reset()

Resets the board and initializes MetroTRK.

Remarks The `__reset()` function calls functions that initialize processor-specific and board-specific items to reset the board. The `__reset()` function also can contain additional initializations. After the initializations, `__reset()` jumps to `__start`.

NOTE You must always examine `__reset()` to determine whether to modify it when customizing MetroTRK for a new target board. For more information, see "MetroTRK Initializations" on page 21 and "Customizing MetroTRK Initializations" on page 47.

Source File For source file information, see the processor-specific appendixes in this manual.

Board-specific? Yes.

DoConnect()

Responds to a Connect request from the debugger.

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

`b`

The message buffer that contains the Connect request and the reply to the request. The Connect request message does not contain input arguments. For more information, see "Connect" on page 71.

Returns Returns a `DSError` error code.

Remarks Sends an acknowledgment to the debugger.

Source File `msghndlr.c`

Board-specific? No.

See Also "Connect" on page 71

DoContinue()

Responds to a Continue request from the debugger.

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

`b`

The message buffer that contains the Continue request and the reply to the request. The Continue request message does not contain input arguments. For more information, see "Continue" on page 72.

Returns Returns a `DSError` error code.

Remarks This procedure swaps in the context of the target application and starts running it again. Because `DoContinue()` is processor-specific, most of the work is done in the board-level function `TargetContinue()`.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetContinue()" on page 136
"Continue" on page 72

DoCPUType()

Responds to a CPUType request from the debugger.

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

`b`

The message buffer that contains the CPUType request and the reply to the request. The CPUType request message does not contain input arguments. For more information, see "CPU-Type" on page 73.

Returns Returns a DSError error code.

Remarks The reply message for this function returns relevant information about the CPU and registers of the target board. For more information, see "CPUType" on page 73.

Source File `msghndlr.c`

Board-Specific? No.

See Also `msgcmd.h`

DoFlushCache

Responds to a FlushCache request from the debugger.

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

`b`

The message buffer that contains the FlushCache request and the reply to the request. For information on the arguments contained in this message, see "FlushCache" on page 75.

Returns Returns a `DSError` error code.

Remarks The message buffer contains values that specify the range of memory to flush. The `DoFlushCache()` function calls `TargetFlushCache()`.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetFlushCache()" on page 137
"FlushCache" on page 75
`msgcmd.h`

DoNotifyStopped()

Notifies the debugger that the target application stopped executing.

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

`command`

The type of message to send to the debugger, which can be one of the following:

- `kDSNotifyStopped`
- `kDSNotifyException`

See `msgcmd.h` for more information about these messages.

Returns Returns a `DSError` error code.

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

Source File `notify.c`

Board-specific? No.

See Also
"TargetAddExceptionInfo()" on page 134
"TargetAddStopInfo()" on page 135
"NotifyStopped" on page 98
`msgcmd.h`

DoReadMemory()

Reads a section of memory from the target board in response to a ReadMemory request from the debugger.

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

`b`

The message buffer that contains the ReadMemory request and the reply to the request. For information on the arguments contained in this message, see "ReadMemory" on page 77.

Returns Returns a `DSError` error code.

Remarks The `DoReadMemory()` function checks that the specified memory addresses are within the 32-bit range and that the range of addresses is valid for the target hardware.

NOTE The `DoReadMemory()` function does not support extended memory addresses.

The `DoReadMemory()` function calls `TargetAccessMemory()`.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetAccessMemory()" on page 132
"ReadMemory" on page 77

DoReadRegisters()

Reads a sequence of registers from the target board in response to a ReadRegisters request from the debugger.

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

`b`

The message buffer that contains the ReadRegisters request and the reply to the request. For information on the arguments contained in this message, see "ReadRegisters" on page 79.

Returns

Returns a `DSError` error code.

Remarks

The `DoReadRegisters()` function checks for a valid input sequence. (The number of the first register to read must be smaller than the number of the last register to read.)

Then `DoReadRegisters()` checks the register type. Depending on the register type, `DoReadRegisters()` calls a function as shown in the following list:

- `kDSRegistersDefault`

  Causes `DoReadRegisters()` to call `TargetAccessDefault()`.

- `kDSRegistersFP`

  Causes `DoReadRegisters()` to call `TargetAccessFP()`.

- `kDSRegistersExtended1`

  Causes `DoReadRegisters()` to call `TargetAccessExtended1()`.

- `kDSRegisters`Extended2

  Causes `DoReadRegisters()` to call `TargetAccessExtended2()`.

The `msgcmd.h` file defines the previously listed register type constants.

NOTE For more information on register definitions, see the processor-specific appendixes in this manual.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetAccessDefault()" on page 139
"TargetAccessFP()" on page 145
"TargetAccessExtended1()" on page 141
"TargetAccessExtended2()" on page 143
"ReadRegisters" on page 79
`msgcmd.h`

DoReset()

The `DoReset()` function re-initializes MetroTRK and resets the board hardware in response to a Reset request from the debugger.

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

`b`

The message buffer that contains the Reset request and the reply to the request. The Reset request message does not contain input arguments. For more information, see "Reset" on page 82.

Returns None.

Remarks Calls the __reset code segment, which is the starting point for MetroTRK initialization. Sends an acknowledgment to the debugger before resetting because `DoReset()` does not resume control after calling `__reset`.

Source File `msghndlr.c`

Board-specific? No.

See Also
"__reset()" on page 105
"Reset" on page 82

DoStep()

The `DoStep()` function steps into or over target application instructions as specified by the `options` argument of the Step request message received from the debugger.

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

`b`

The message buffer that contains the Step request and the reply to the request. For information on the arguments contained in this message, see "Step" on page 83.

Returns Returns a `DSError` error code.

Remarks The following list describes what `DoStep()` does based on the value of the `options` input argument (passed as part of the Step request message):

- `kDSStepSingle`

  Causes `DoStep()` to call the processor-specific function `TargetSingleStep()`, which steps the number of steps specified in the message.

- `kDSStepOutOfRange`

  Causes `DoStep()` to call the processor-specific function `TargetStepOutOfRange()`, which runs the code until the PC (program counter) is outside the range of values specified in the message.

Source File `msghndlr.c`

Board-specific? No.

See Also
"TargetSingleStep()" on page 147
"TargetStepOutOfRange()" on page 148
"Step" on page 83

DoStop()

Responds to a Stop request from the debugger.

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

`b`

The message buffer that contains the Stop request and the reply to the request. The Stop request message does not contain input arguments. For more information, see "Stop" on page 87.

Returns Returns a `DSError` error code.

Remarks This function stops running the target application and sets the running flag to false. For more information, see "MetroTRK Execution States" on page 16.

Source File `msghndlr.c`

Board-specific? No.

See Also "Stop" on page 87

DoSupportMask()

The `DoSupportMask()` function sends a vector that indicates which messages MetroTRK supports in response to a SupportMask request from the debugger.

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

`b`

The message buffer that contains the SupportMask request and the reply to the request. The SupportMask request message does not contain input arguments. For more information, see "SupportMask" on page 88.

Returns Returns a `DSError` error code.

Remarks The `DoSupportMask()` function calls `TargetSupportMask()`, which returns a 256-bit vector that indicates which messages of the debug message interface MetroTRK supports. Then `DoSupportMask()` places the vector in a reply message that MetroTRK sends to the debugger.

In the returned bit-vector, each bit corresponds to the message (type `MessageCommandID`) with an ID matching the position of the bit in the array. If the value of the bit is 1, the message is available; if the value of the bit is 0, the message is not available.

For example, if `kDSReset` is available, the value of the fourth bit is 1 because `kDSReset` is the fourth message.

For more information, see `msgcmd.h`.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetSupportMask()" on page 149
"SupportMask" on page 88

DoVersions()

Replies with a set of four version numbers in response to a Versions request from the debugger.

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

`b`

The message buffer that contains the Versions request and the reply to the request. The Versions request message does not contain input arguments. For more information, see "Versions" on page 90.

Returns — Returns a `DSError` error code.

Remarks — The `DoVersions()` function replies to the debugger with a set of four version numbers. These represent two attributes called kernel and protocol, each of which has a major and a minor version number.

The kernel attribute is the version of the MetroTRK build. The protocol attribute is the version of the debug message interface and low-level serial protocols used by MetroTRK.

NOTE If you change MetroTRK or its protocols, you can update the kernel and protocol version numbers, respectively. For more information, see "Changing Versions-Related Code" on page 58.

The `DoVersions()` function calls `TargetVersions()`.

Source File — `msghndlr.c`

Board-specific? — No.

See Also — "TargetVersions()" on page 150
"Versions" on page 90

DoWriteMemory()

Writes values to a segment of memory on the target board in response to a WriteMemory request from the debugger.

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

b

The message buffer that contains the WriteMemory request and the reply to the request. For information on the arguments contained in this message, see "WriteMemory" on page 92.

Returns — Returns a `DSError` error code.

Remarks — The `DoWriteMemory()` function checks that the specified memory addresses are within the 32-bit range and that the range of addresses is valid for the target hardware.

NOTE The `DoWriteMemory()` function does not support extended memory addresses.

The `DoWriteMemory()` function calls `TargetAccessMemory()`.

Source File — `msghndlr.c`

Board-specific? — No.

See Also — "TargetAccessMemory()" on page 132
"WriteMemory" on page 92

DoWriteRegisters()

Writes values to a sequence of registers on the target board in response to a WriteRegisters request from the debugger.

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

`b`

The message buffer that contains the WriteRegisters request and the reply to the request. For information on the arguments contained in this message, see "WriteRegisters" on page 94.

Returns — Returns a `DSError` error code.

Remarks — The `DoWriteRegisters()` function checks for a valid input sequence. (The number of the first register must be smaller than the number of the last register in the specified sequence of registers.)

Then `DoWriteRegisters()` checks the register type. Depending on the register type, `DoWriteRegisters()` calls a function as shown in the following list:

- `kDSRegistersDefault`

  Causes `DoWriteRegisters()` to call `TargetAccessDefault()`.

- `kDSRegistersFP`

  Causes `DoWriteRegisters()` to call `TargetAccessFP()`.

- `kDSRegistersExtended1`

  Causes `DoWriteRegisters()` to call `TargetAccessExtended1()`.

- `kDSRegistersExtended2`

  Causes `DoWriteRegisters()` to call `TargetAccessExtended2()`.

The `msgcmd.h` file defines the previously listed register type constants.

NOTE For more information on register definitions, see the processor-specific appendixes in this manual.

Source File `msghndlr.c`

Board-specific? No.

See Also "TargetAccessDefault()" on page 139
"TargetAccessExtended1()" on page 141
"TargetAccessExtended2()" on page 143
"TargetAccessFP()" on page 145
"WriteRegisters" on page 94
`msgcmd.h`

InitializeIntDrivenUART()

Initializes the UART library when using interrupt-driven I/O.

```
UARTError InitializeIntDrivenUART(
    UARTBaudRate                baudRate,
    unsigned char               intDrivenInput,
    unsigned char               intDrivenOutput,
    volatile unsigned char**    inputPendingPtrRef);
```

`baudRate`

The data transmission rate (baud rate) for the UART.

`intDrivenInput`

Enables interrupt-driven input when set to true.

`intDrivenOutput`

Enables interrupt-driven output when set to true.

NOTE MetroTRK uses interrupt-driven input but not interrupt-driven output.

`inputPendingPtrRef`

On return, a pointer to an input-pending flag that the calling function can use to determine whether input arrived. (When interrupt-driven input is disabled, the value of this flag is always false.)

Returns — Returns a `UARTError` error code.

Remarks — The status of the input-pending flag can change at any time unless you mask the serial interrupt.

Source File — For source file information, see the processor-specific appendixes in this manual.

Board-specific? — Yes.

See Also — "InterruptHandler()" on page 124
"SwapAndGo()" on page 131
"UARTInterruptHandler()" on page 152

InitializeUART()

Initializes the serial hardware on the target board.

NOTE You must change `InitializeUART()` for new target boards.

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

`baudRate`

The rate at which MetroTRK communicates with the debugger. The `UART.h` file defines the type `UARTBaudRate`.

Returns — Returns a `UARTError` error code.

Source File — `uart.c`

Board-specific? — Yes.

InterruptHandler()

Handles an interrupt received by MetroTRK.

```
void InterruptHandler();
```

Returns None.

Remarks After receiving the interrupt, MetroTRK saves the state (context) of the target application (which had been running). MetroTRK then restores its own state and handles the interrupt.

If the value of `TRK_TRANSPORT_INT_DRIVEN` is 1 (indicating that this MetroTRK is interrupt-driven), `InterruptHandler()` first determines whether the interrupt is a communication interrupt. If it is, `InterruptHandler()` calls `UARTInterruptHandler()` to process the interrupt. Otherwise, `InterruptHandler()` processes the non-communication interrupt normally.

NOTE For some target boards, `InterruptHandler()` is found in a C program file; for others, `InterruptHandler()` resides in an assembly language file and its name differs slightly. For more information, see the processor-specific appendixes.

Source File For source file information, see the processor-specific appendixes in this manual.

Board-specific? No.

See Also "InitializeIntDrivenUART()" on page 122
"SwapAndGo()" on page 131
"UARTInterruptHandler()" on page 152

ReadUARTPoll()

Polls the serial device to see whether there is a character to read. If there is, `ReadUARTPoll()` reads it; otherwise, `ReadUARTPoll()` returns an error.

NOTE You must change `ReadUARTPoll()` for new target boards.

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

`c`

Pointer to the output variable for the character read.

Returns

Returns one of the following `UARTError` error codes:

- `kUARTNoData`

  Indicates that no character was available to read.

- `kUARTNoError`

  Indicates that no error occurred.

Source File

`uart.c`

Board-specific?

Yes.

ReadUART1()

Reads one byte from the serial device.

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

`c`

Pointer to the output variable for the character read.

Returns — Returns a `UARTError` error code.

Remarks — The `ReadUART1()` function waits until a character is available to read or an error occurs.

NOTE You must change `ReadUART1()` for new target boards.

Source File — `uart.c`

Board-specific? — Yes.

ReadUARTN()

Reads the specified number of bytes from the serial device.

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

`bytes`

Pointer to the output buffer for the data read.

`limit`

Number of bytes to read and size of output buffer.

Returns — Returns a `UARTError` error code.

Remarks — Returns after reading the specified number of bytes (or encountering an error.)

NOTE The `ReadUARTN()` function calls `ReadUART1()`; consequently, `ReadUARTN()` executes correctly as long as `ReadUART1()` does.

Source File — `uart.c`

Board-specific? — No.

See Also — "ReadUART1()" on page 126

ReadUARTString()

Reads a terminated string from the serial device.

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

`s`

Pointer to the output buffer for the string read.

`limit`

Size of the output buffer.

`termChar`

Character that signals the end of the string (in the input stream.)

Returns Returns a `UARTError` error code.

Remarks The `ReadUARTString()` function terminates the string (in the output buffer) with a null (`\0`) character. Consequently, the output buffer must be one byte longer than the length of the string.

The `ReadUARTString()` function returns after reading a terminating character from the input or when the buffer overflows. If the input stream stops, `ReadUARTString()` does not time-out.

NOTE The `ReadUARTString()` function calls `ReadUART1()`; consequently, `ReadUARTString()` executes correctly as long as `ReadUART1()` does.

Source File `uart.c`

Board-specific? No.

See Also "ReadUART1()" on page 126

SuppAccessFile()

Creates and sends a ReadFile or WriteFile message to the debugger. These messages instruct the debugger to read data from a file or write data to a file on the host.

```
DSError SuppAccessFile(
    ui32        file_handle,
    ui8*        data,
    size_t*     count,
    DSIOResult* io_result,
    bool        need_reply,
    bool        read);
```

`file_handle`

The handle of the file to be read or written. `stdin`, `stdout`, and `stderr` have predefined handles. For more information, see the definition of `DSFileHandle` in `msgcmd.h`.

`data`

Data to be read or written to the file.

`count`

Pointer to the size of the data to be read or written, in bytes. On return, points to the size of the data that was read or written.

`io_result`

Pointer to storage for an I/O result error code. For more information, see the definition of `DSIOResult` in `msgcmd.h`.

`need_reply`

If `TRUE`, `SuppAccessFile()` waits for an acknowledgement from the debugger. If the debugger sends an invalid acknowledgement or `SuppAccessFile()` waits for the duration of the timeout limit, `SuppAccessFile()` resends the message.

`read`

If `TRUE`, a ReadFile message is sent. If `FALSE`, a WriteFile message is sent.

Returns — Returns a `DSError` error code.

Remarks — None.

Source File — `support.c`

Board-specific? — No.

See Also — `msgcmd.h`

SwapAndGo()

Saves the state (context) of MetroTRK, restores the state of the target application, and continues executing the target application from the PC (program counter).

```
void SwapAndGo();
```

Returns None.

Remarks The `TargetContinue()` function calls `SwapAndGo()` to resume running the target application after MetroTRK responds to an interrupt or a message from the debugger.

Source File For source file information, see the processor-specific appendixes in this manual.

Board-specific? No.

See Also
"InitializeIntDrivenUART()" on page 122
"InterruptHandler()" on page 124
"UARTInterruptHandler()" on page 152

TargetAccessMemory()

Reads from or writes to memory in response to a ReadMemory request or a WriteMemory request.

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

`Data`

For a read operation, contains the output of the read. For a write operation, contains a pointer to the data to write.

`virtualAddress`

The starting address in memory for the read or write operation.

`memorySize`

For a read operation this is, on input, the requested size of the area to read and, on output, the size of the area read by `TargetAccessMemory()`. For a write operation this is, on input, the requested amount of data to write and, on output, the amount of data written by `TargetAccessMemory()`.

`accessOptions`

A value currently not used by the `TargetAccessMemory()` function. (The value, while not used by this function, is the same value specified for the options field of the ReadMemory and WriteMemory requests. For more information, see "ReadMemory" on page 77 and "WriteMemory" on page 92.

`read`

A Boolean value that selects a read or write operation. A value of TRUE selects a read operation; a value of FALSE selects a write operation.

Returns Returns a `DSError` error code.

Remarks

Both `DoReadMemory()` and `DoWriteMemory()` call `TargetAccessMemory()` to access memory.

A Boolean parameter passed by the calling function specifies the read or write operation. The `TargetAccessMemory()` function calls `ValidMemory32()`, which verifies the target addresses based on the memory configuration of the board.

Source File

`targimpl.c`

Board-specific?

No.

See Also

"DoReadMemory()" on page 111
"DoWriteMemory()" on page 119
"ValidMemory32()" on page 153
"ReadMemory" on page 77
"WriteMemory" on page 92
`msgcmd.h`
`targimpl.h`

TargetAddExceptionInfo()

Builds a `NotifyException` message when notifying the debugger that an exception occurred on the board.

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

`b`

The message buffer that contains the `NotifyException` notification. For information on the arguments contained in this message, see "NotifyException" on page 97.

Returns Returns a `DSError` error code.

Remarks The contents of the message buffer differs depending on the processor. Examples of information that the message can contain follow:

- The PC (Program Counter) at the time the exception was generated
- The instruction at that value of the PC
- The exception ID

The register definition files define the specific information returned for your target processor. For more information on the register definition files and exceptions, see the processor-specific appendixes in this manual.

Source File `targimpl.c`

Board-specific? No.

See Also "DoNotifyStopped()" on page 110
"NotifyException" on page 97

TargetAddStopInfo()

Builds a `NotifyStopped` message when notifying the debugger that the target application stopped.

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

`b`

The message buffer that contains the NotifyStopped notification. For information on the arguments contained in this message, see "NotifyStopped" on page 98.

Returns Returns a `DSError` error code.

Remarks The contents of the message buffer differs depending on the processor. Examples of information that the message can contain follow:

- The PC (Program Counter) at the time the exception was generated
- The instruction at that value of the PC
- The exception ID

Examine the MetroTRK source code to see the specific information returned for your target processor.

Source File `targimpl.c`

Board-specific? No.

See Also "DoNotifyStopped()" on page 110
"NotifyStopped" on page 98

TargetContinue()

The `TargetContinue()` function starts the target application running and then blocks until control returns to MetroTRK (because a relevant exception occurred).

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

`b`

This message buffer has no input parameters and no reply message. For more information, see "Continue" on page 72.

Returns Returns a `DSError` error code.

Remarks The `TargetContinue()` function starts running the program by calling `SwapAndGo()` and sets the running flag to true. When MetroTRK regains control (because of an unhandled exception or breakpoint), control returns to the MetroTRK core, which properly handles the exception.

Source File `targimpl.c`

Board-specific? No.

See Also
"DoContinue()" on page 107
"Continue" on page 72
"SwapAndGo()" on page 131

TargetFlushCache()

Flushes the caches as specified by the input parameters.

```
DSError TargetFlushCache(
    ui8    options
    void*  start
    void*  end);
```

`options`

The type of cache to flush.

`start`

The starting address of the memory to flush in the cache.

`end`

The ending address of the memory to flush in the cache.

Returns — Returns a `DSError` error code.

Remarks — You may have to modify `TargetFlushCache()` if you create a new MetroTRK implementation to work with a currently unsupported processor. In this case, your new version of `TargetFlushCache()` must do one of the following:

- Flush the caches as specified by the `options`, `start`, and `end` parameters
- Flush more than the specified amount of cache

For example, if you choose not to examine the `options` parameter to see which type of cache to flush, you must flush all the caches on the board. If you examine the `options` parameter but not the `start` and `end` parameters, you must flush the entire cache of the type specified by the `options` parameter.

Source File — `targimpl.c`

Board-specific? — No.

See Also — “DoFlushCache” on page 109
“FlushCache” on page 75

TargetInterrupt()

Handles an exception by notifying the debugger.

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

`event`

The original event triggered by an exception or breakpoint.

Returns Returns a `DSError` error code.

Remarks The `TargetInterrupt()` function, which is called when an exception or breakpoint occurs, calls `DoNotifyStopped()` to notify the debugger. The `TargetInterrupt()` function also sets the running flag to false unless MetroTRK is stepping through multiple lines and stepping is not complete.

Source File `targimpl.c`

Board-specific? No.

See Also "DoNotifyStopped()" on page 110

TargetAccessDefault()

Reads data from or writes data to a sequence of registers in the default register block.

```
DSError TargetAccessDefault(
    unsigned int     firstRegister,
    unsigned int     lastRegister,
    MessageBuffer*   b,
    size_t*          registerStorageSize
    bool             read);
```

`firstRegister`

The number of the first register in the sequence.

`lastRegister`

The number of the last register in the sequence.

`b`

The message buffer that contains the ReadRegisters or WriteRegisters request and the reply to the request. For information on the arguments contained in this message, see "ReadRegisters" on page 79 or "WriteRegisters" on page 94.

NOTE The `DoReadRegisters()` or `DoWriteRegisters()` functions pass the message buffer to `TargetAccessDefault()`.

`registerStorageSize`

On output, the number of bytes read or written (a maximum of 2048 bytes).

`read`

A Boolean variable that instructs MetroTRK to read the specified registers if the variable is true. Otherwise, MetroTRK writes the specified registers.

Returns Returns a `DSError` error code.

Remarks

The `TargetAccessDefault()` function verifies the range of the specified registers before accessing them and attempts to catch exceptions that occur while reading or writing the registers.

The `DoReadRegisters()` or `DoWriteRegisters()` functions, which call `TargetAccessDefault()`, place the position pointer at the correct position in the message buffer before calling this function.

NOTE For more information on registers, see the processor-specific appendixes in this manual.

Source File `targimpl.c`

Board-specific? No.

See Also
"DoReadRegisters()" on page 112
"DoWriteRegisters()" on page 120
"ReadRegisters" on page 79
"WriteRegisters" on page 94

TargetAccessExtended1()

Reads data from or writes data to a sequence of registers in the `extended1` register block.

```
DSError TargetAccessExtended1(
    unsigned int     firstRegister,
    unsigned int     lastRegister,
    MessageBuffer*   b,
    size_t*          registerStorageSize
    bool             read);
```

`firstRegister`

The number of the first register in the sequence.

`lastRegister`

The number of the last register in the sequence.

`b`

The message buffer that contains the ReadRegisters or WriteRegisters request and the reply to the request. For information on the arguments contained in this message, see "ReadRegisters" on page 79 or "WriteRegisters" on page 94.

NOTE The `DoReadRegisters()` or `DoWriteRegisters()` functions pass the message buffer to `TargetAccessExtended1()`.

`registerStorageSize`

On output, the number of bytes read or written (a maximum of 2048 bytes).

`read`

A Boolean variable that instructs MetroTRK to read the specified registers if the variable is true. Otherwise, MetroTRK writes the specified registers.

Returns Returns a `DSError` error code.

Remarks — The `TargetAccessExtended1()` function verifies the range of the specified registers before accessing them and attempts to catch exceptions that occur while reading from or writing to the registers.

NOTE For more information on registers, see the processor-specific appendixes in this manual.

Source File — `targimpl.c`

Board-specific? — No.

See Also — "DoReadRegisters()" on page 112
"DoWriteRegisters()" on page 120
"ReadRegisters" on page 79
"WriteRegisters" on page 94

TargetAccessExtended2()

Reads data from or writes data to a sequence of registers in the `extended2` register block.

```
DSError TargetAccessExtended2(
    unsigned int     firstRegister,
    unsigned int     lastRegister,
    MessageBuffer*   b,
    size_t*          registerStorageSize
    bool             read);
```

`firstRegister`

The number of the first register in the sequence.

`lastRegister`

The number of the last register in the sequence.

`b`

The message buffer that contains the ReadRegisters or WriteRegisters request and the reply to the request. For information on the arguments contained in this message, see "ReadRegisters" on page 79 or "WriteRegisters" on page 94.

NOTE The `DoReadRegisters()` or `DoWriteRegisters()` functions pass the message buffer to `TargetAccessExtended2()`.

`registerStorageSize`

On output, the number of bytes read or written (a maximum of 2048 bytes).

`read`

A Boolean variable that instructs MetroTRK to read the specified registers if the variable is true. Otherwise, MetroTRK writes the specified registers.

Returns Returns a `DSError` error code.

Remarks — The `TargetAccessExtended2()` function verifies the range of the specified registers before accessing them and attempts to catch exceptions that occur while reading the registers.

NOTE For more information on registers, see the processor-specific appendixes in this manual.

Source File — `targimpl.c`

Board-specific? — No.

See Also — "DoReadRegisters()" on page 112
"DoWriteRegisters()" on page 120
"ReadRegisters" on page 79
"WriteRegisters" on page 94

TargetAccessFP()

Reads data from or writes data to a sequence of registers in the floating point register block.

```
DSError TargetAccessFP(
    unsigned int     firstRegister,
    unsigned int     lastRegister,
    MessageBuffer*   b,
    size_t*          registerStorageSize
    bool             read);
```

`firstRegister`

The number of the first register in the sequence.

`lastRegister`

The number of the last register in the sequence.

`b`

The message buffer that contains the ReadRegisters or WriteRegisters request and the reply to the request. For information on the arguments contained in this message, see "ReadRegisters" on page 79 or "WriteRegisters" on page 94.

NOTE The `DoReadRegisters()` or `DoWriteRegisters()` functions pass the message buffer to `TargetAccessFP()`.

`registerStorageSize`

On output, the number of bytes read or written (a maximum of 2048 bytes).

`read`

A Boolean variable that instructs MetroTRK to read the specified registers if the variable is true. Otherwise, MetroTRK writes the specified registers.

Returns Returns a `DSError` error code.

Remarks — The `TargetAccessFP()` function verifies the range of the specified registers before accessing them and attempts to catch exceptions that occur while reading the registers.

NOTE For more information on registers, see the processor-specific appendixes in this manual.

Source File — `targimpl.c`

Board-specific? — No.

See Also — "DoReadRegisters()" on page 112
"DoWriteRegisters()" on page 120
"ReadRegisters" on page 79
"WriteRegisters" on page 94

TargetSingleStep()

Steps into or over a specified number of instructions.

```
DSError TargetSingleStep(
    unsigned    count,
    bool        stepOver);
```

`count`

The number of instructions to step over or into.

`stepOver`

A Boolean value that instructs MetroTRK to either step into or step over the number of instructions specified by the `count` parameter. (A value of 1 indicates a request to step over; a value of 0 indicates a request to step into.)

Returns Returns a `DSError` error code.

Remarks The `TargetSingleStep()` function sets up the trace exception and steps into or over the requested number of instructions. (After each instruction, `TargetSingleStep()` checks whether that instruction is the last instruction to step through.)

Source File `targimpl.c`

Board-specific? No.

See Also "DoStep()" on page 115
"Step" on page 83

TargetStepOutOfRange()

Runs the target application until the PC (Program Counter) is outside a specified range of values.

```
DSError TargetStepOutOfRange(
    ui32   start,
    ui32   end
    bool   stepOver);
```

`start`

The starting address of the range.

`end`

The ending address of the range.

`stepOver`

A Boolean value that instructs MetroTRK to either step into or step over instructions until the PC (Program Counter) is outside a specified range of values. (A value of 1 indicates a request to step over; a value of 0 indicates a request to step into.)

Returns Returns a `DSError` error code.

Remarks The `TargetStepOutOfRange()` function sets up the trace exception. After each instruction, `TargetStepOutOfRange()` checks whether the PC is outside the specified range of values.

Source File `targimpl.c`

Board-specific? No.

See Also "DoStep()" on page 115
"Step" on page 83

TargetSupportMask()

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

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

`mask`

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 value of the bit is 1, the message is available; if the value of the bit is 0, the message is not available.

For example, if `kDSReset` is available, the value of the fourth bit is 1 because `kDSReset` is the fourth message.

For more information, see `msgcmd.h`.

Returns None.

Remarks Changing the support mask values does not require changing this function because the values are defined in the file `default_supp_mask.h`. For more information, see "Changing SupportMask-Related Code" on page 57.

Source File `targimpl.c`

Board-specific? No.

See Also

"DoSupportMask()" on page 117
"SupportMask" on page 88
`msgcmd.h`
`target.h`

TargetVersions()

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

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

`versions`

Output variable containing version information for the running MetroTRK build.

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

Remarks — The `TargetVersions()` function replies to the debugger with a set of four version numbers. These represent two attributes called kernel and protocol, each of which has a major and a minor version number.

The kernel attribute is the version of the MetroTRK build. The protocol attribute is the version of the debug message interface and low-level serial protocols used by MetroTRK.

NOTE If you change MetroTRK or its protocols, you can update the kernel and protocol version numbers, respectively. For more information, see "Changing Versions-Related Code" on page 58.

Source File — `targimpl.c`

Board-specific? — No.

See Also — "DoVersions()" on page 118
"Versions" on page 90
`target.h`

TerminateUART()

Deactivate the serial device.

```
UARTError TerminateUART ( void );
```

Returns — Returns a `UARTError` error code.

Remarks — The default implementation of MetroTRK does not call this function. However, you can run an operating system on the target board while you are debugging using MetroTRK. When you finish debugging, you can use this function to release the UART device so that you can run a different program on the target board.

NOTE You must implement `TerminateUART()` for your target board before attempting to call it.

Source File — `uart.c`

Board-specific? — Yes.

UARTInterruptHandler()

Handles a UART interrupt.

NOTE You must change `UARTInterruptHandler()` for new target boards if you are using interrupt-driven communication with MetroTRK.

```
asm void UARTInterruptHandler(void);
```

Returns None.

Remarks The `UARTInterruptHandler()` function is part of the UART driver code. The MetroTRK `InterruptHandler()` function calls `UARTInterruptHandler()`, which disables interrupts while it is running. For a serial input interrupt, `UARTInterruptHandler()` gets the incoming characters from the UART, stores them in a buffer, and sets the input-pending flag to true. The `ReadUARTPoll()` function gets the next character from the buffer and clears the input-pending flag if the buffer becomes empty.

When creating your own implementation of `UARTInterruptHandler()`, do not assume that the registers hold any particular values. The `UARTInterruptHandler()` function always must return to the calling function.

Source File For source file information, see the processor-specific appendixes in this manual.

Board-specific? Yes.

See Also "InitializeIntDrivenUART()" on page 122
"InterruptHandler()" on page 124
"SwapAndGo()" on page 131

ValidMemory32()

Verifies the range of addresses for the target board when MetroTRK reads or writes to memory.

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

`addr`

The starting address of the memory segment.

`length`

The length of the memory segment.

`readWriteable`

This parameter must be one of the following values:

- `kValidMemoryReadable`
- `kValidMemoryWriteable`

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

Remarks — The `ValidMemory32()` function is not board-specific. However, `ValidMemory32()` uses a global variable called `gMemMap`, which contains board-specific memory layout information.

NOTE To customize the memory layout information for a new target board, change the definition of `gMemMap` in the `memmap.h` file.

Source File — `targimpl.c`

Board-specific? — No.

See Also — `memmap.h`

WriteUART1()

Writes one byte to the serial device.

NOTE You must change `WriteUART1()` for new target boards.

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

`c`

The character to write.

Returns — Returns a `UARTError` error code.

Source File — `uart.c`

Board-specific? — Yes.

See Also — "WriteUARTN()" on page 155
"WriteUARTString()" on page 156

WriteUARTN()

Writes *n* bytes to the serial device.

NOTE The `WriteUARTN()` function calls `WriteUART1()`; consequently, `WriteUARTN()` executes correctly as long as `WriteUART1()` does.

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

`bytes`

Pointer to the input data.

`length`

The number of bytes to write.

Returns — Returns a `UARTError` error code.

Source File — `uart.c`

Board-specific? — No.

See Also — "WriteUART1()" on page 154
"WriteUARTString()" on page 156

WriteUARTString()

Writes a character string to the serial device.

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

`string`

Pointer to the input data.

Returns 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.

NOTE The `WriteUARTString()` function calls `WriteUARTN()`, which calls `WriteUART1()`. Consequently, `WriteUARTString()` executes correctly as long as `WriteUART1()` does.

Source File `uart.c`

Board-specific? No.

See Also "WriteUART1()" on page 154
"WriteUARTN()" on page 155

C

MIPS-Specific Information

This appendix provides MetroTRK-related information that is specific to MIPS processors.

This appendix contains the following topics:

MIPS-Specific Location of __reset()

Table C.1 shows the location of `__reset()` for the currently supported MIPS reference boards.

NOTE Some reference boards use a function called `__reset()`; other reference boards use assembly language code that is labeled `__reset`.

Table C.1 MIPS-specific __reset() location

Reference Board	File Containing __reset()
Midas RTE VR4x00 PC	`\Processor\mips\board\midas\rte_vr4x00_pc\reset_midas_vr4k.c`
Midas RTE VR5000 PC	`\Processor\mips\board\midas\rte_vr5000_pc\reset_midas_vr5k.c`
IDT 79S381	`Processor\mips\board\idt\79s381\reset_idt.c`
LSI BDMR 4101	`Processor\mips\board\lsi\bdmr_4101\reset_lsi_tr4k.c`
UEB 41xx	`Processor\mips\board\nec\41xx_ueb\reset_ueb_vr4k.c`
DDB VRC4373	`Processor\mips\board\nec\ddb_vrc4373\reset_ddb_vr4k.c`
DDB VRC5074	`Processor\mips\board\nec\ddb_vrc5074\reset_ddb_vr5k.c`

MIPS-Specific Memory Locations

This section discusses the default memory locations of the MetroTRK RAM sections and of your target application for MIPS.

This section contains the following topics:

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains the following topics:

Exception vectors

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 in low memory) and spans 4096 bytes to end at 0x80001000. For a ROM-based MetroTRK, the MetroTRK initialization process copies the exception vectors from ROM to RAM.

NOTE Do not change the location of the exception vectors because the processor expects the exception vectors to reside at the set location.

Data and code sections

In the default implementation of MetroTRK, which is ROM-based, no code exists in RAM because the code executes directly from ROM.

You can set the location of the data and code sections in the linker command file in your MetroTRK project. (The linker command file is the file in your project that has the extension `.lcf`.) For more information, see the *Targeting* manual for your target processor.

The stack

In the default implementation, the MetroTRK stack resides in high memory and grows downward. The default implementation of MetroTRK requires a maximum of 8KB of stack space.

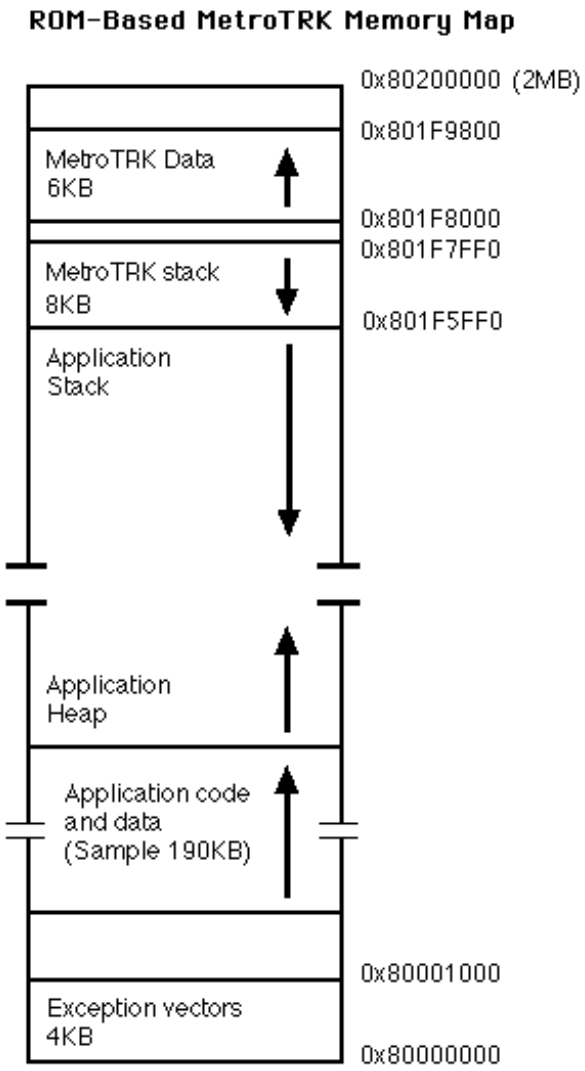
You can set the location of the stack section in the linker command file in your MetroTRK project. (The linker command file is the file in your project that has the extension `.lcf`.) For more information, see the *Targeting* manual for your target processor.

MetroTRK Memory Map

Figure C.1 shows a sample map of RAM sections as configured when running MetroTRK with a sample target application.

NOTE The memory maps for other target boards may differ from the memory map shown in Figure C.1.

Figure C.1 MIPS-specific MetroTRK RAM map



MIPS-Specific Exception Handling

MetroTRK initializes the memory system before copying the exception vectors to RAM because this operation depends on safe memory access. The location in memory of the exception vectors is processor-specific. By default, MetroTRK responds to certain exceptions and notifies the debugger of all other exceptions.

The vector-copying process is not board-dependent because the processor determines the memory locations of the exception vectors. Consequently, the vector-copying process does not differ for differing board configurations.

For MIPS, `InitializeTarget()`, which resides in `targimpl.c`, calls `__copy_vectors()`, which resides in `copy_vectors.c`, to copy the exception vectors to RAM.

You can customize exception handling for MIPS; your application can handle any exception. However, the MIPS version of MetroTRK must receive all `Breakpoint` exceptions. Therefore, after handling a `Breakpoint` exception, your application must call the MetroTRK exception-handling code so that MetroTRK also can handle the `Breakpoint` exception. (This is called sharing the exception.)

In addition, if MetroTRK is using interrupt-driven communications, you must share the interrupt with MetroTRK after you handle it with your customized code. Otherwise, your application can overwrite any other exceptions that it handles.

The following file defines the exceptions for MIPS target boards:

`Processor\mips\export\mips_except.h`

For more information, see "Exception vectors" on page 159.

MIPS-Specific Register Definitions

The following file defines all MIPS registers with mnemonic names and the sizes (bit-lengths) of the registers in the different register blocks:

`Processor\mips\export\mips_reg.h`

MIPS-Specific Version Location

For MIPS, the `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` constants reside in the following file:

`Processor\mips\generic\mips_version.h`

MIPS-Specific Locations of target.h

The location of `target.h` for supported reference boards differs. The following list shows the MIPS-specific directory paths for `target.h`:

- `Processor\mips\board\midas\rte_vr4x00_pc`
- `Processor\mips\board\midas\rte_vr5000_pc`
- `Processor\mips\board\idt\79s381`
- `Processor\mips\board\lsi\bdmr_4101`
- `Processor\mips\board\ueb\41xx_ueb`
- `Processor\mips\board\ddb\ddb_vrc4373`

Interrupt-Driven Communication for MIPS

For all target boards, you must define the value of `TRK_TRANSPORT_INT_DRIVEN` as 1 to indicate that MetroTRK uses interrupt-driven communication. (For more information, see "Customizing MetroTRK to Be Interrupt-Driven" on page 54.) In addition, for MIPS, you must define the following items in `target.h`:

- `TRK_TRANSPORT_INT_MASK`
- `TRK_TRANSPORT_INT_KEY`

Define the interrupt mask and the interrupt key so that MetroTRK can identify the interrupt that corresponds to the communication transport. Using the cause register, it is the register for which:

```
cr & TRK_TRANSPORT_INT_MASK
     == TRK_TRANSPORT_INT_KEY
```

NOTE For MIPS boards for which interrupt-driven MetroTRK communication is enabled, `TRK_TRANSPORT_INT_DRIVEN`, `TRK_TRANSPORT_INT_MASK`, and `TRK_TRANSPORT_INT_KEY` reside in `target.h`.

For MIPS, mask off the exception code and the IP bit corresponding to the transport interrupt. The key must correspond to an exception code of 0 (interrupt) and the appropriate IP bit must be set. For example, on the Midas RTE-VR4x00-PC reference board, the UART uses interrupt 0; consequently, the mask is 0x0000047C and the key is 0x00000400.

For a MIPS-specific example implementation of `UARTInterruptHandler()` used for the Midas RTE-VR4100-PC/VR4300-PC target board, see the following file:

`Transport\uart\scx26x1\SCx26x1_mips.c`

The `UARTInterruptHandler()` function calls a C function called `IRQHandler()` that stores incoming characters in a buffer for later retrieval by the various ReadUART functions.

Table C.2 shows the MIPS-specific location of several functions related to interrupt-driven communication with MetroTRK. (For more information, see "MetroTRK Function Reference" on page 103.)

Table C.2 MIPS-specific interrupt-driven communication functions

Function Name	Location
`InitializeIntDrivenUART()`	`Transport\uart\scx26x1\uart.c`
`InterruptHandler()`	`Processor\mips\generic\targimpl.c`
`SwapAndGo()`	`Processor\mips\generic\targimpl.c`
`UARTInterruptHandler()`	`Transport\uart\scx26x1\SCx26x1_mips.c`

D

PowerPC-Specific Information

This appendix provides MetroTRK-related information that is specific to the PowerPC processor.

This appendix contains the following topics:

PowerPC-Specific Location of __reset

For the currently supported PowerPC reference boards, Table D.1 shows the location of `__reset`.

NOTE Some reference boards use a function called `__reset()`; other reference boards use assembly language code that is labeled `__reset`.

Table D.1 PowerPC-specific __reset location

Reference Board	File Containing __reset
Cogent CMA102	`Processor\ppc\Board\cogent\cma102\reset_7xx_603e_cogent.asm`
Motorola MPC 8260 VADS	`Processor\ppc\Board\motorola\mpc_8260_vads\reset_mpc_8260_vads.asm`
Motorola MPC 8xx ADS	`Processor\ppc\Board\motorola\mpc_8xx_ads\reset_8xx_ads.asm`
Motorola MPC 8xx MBX	`Processor\ppc\Board\motorola\mpc_8xx_mbx\reset_8xx_mbx.asm`
Motorola MPC 5xx EVB	`Processor\ppc\Board\motorola\mpc_5xx_evb\reset_5xx_evb.asm`
Motorola Yellow Knife x4	`Processor\ppc\Board\motorola\yellowknife_x4\reset_7xx_603e_yk.asm`
Motorola Excimer	`Processor\ppc\Board\motorola\excimer\reset_excimer.asm`
Motorola MPC 555 ETAS	`Processor\ppc\Board\motorola\mpc_555_etas\reset_555_etas.asm`
Phytec miniMODUL-505 and miniMODUL-509	`Processor\ppc\Board\phytec\minimodul_ppc_505_509\reset_5xx_phytec.asm`

PowerPC-Specific Memory Locations

This section discusses the default memory locations of the MetroTRK code and data sections and of your target application for PowerPC.

This section contains the following topics:

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains the following topics:

Exception vectors

For a ROM-based MetroTRK, the MetroTRK initialization process copies the exception vectors from ROM to RAM.

NOTE For the MPC 555 ETAS board, the exception vectors remain in ROM.

The location of the exception vectors in RAM is a set characteristic of the processor. For PowerPC, the exception vector must start at 0x000100 (which is in low memory) and spans 7936 bytes to end at 0x002000.

NOTE Do not change the location of the exception vectors because the processor expects the exception vectors to reside at the set location.

Data and code sections

In the default implementation of MetroTRK used with most supported target boards, which is ROM-based, no MetroTRK code section exists in RAM because the code executes directly from ROM. However, for some PowerPC target boards, some MetroTRK code does reside in RAM, usually for one of the following reasons:

- Executing from ROM is slow enough to limit the MetroTRK data transmission rate (baud rate).
- For the 603e and 7xx processors, the main exception handler must reside in cacheable memory if the instruction cache is enabled. On some boards the ROM is not cacheable; consequently, the main exception handler must reside in RAM if the instruction cache is enabled.

RAM does contain a MetroTRK data section. For example, on the Motorola 8xx ADS and Motorola 8xx MBX boards, the MetroTRK

data section starts, by default, at the address `0x3F8000` and ends at the address 0x3FA000. (For more information, see "MetroTRK Memory Map" on page 168.)

You can change the location of the data and code sections in your MetroTRK project using one of the following methods:

- By modifying settings in the EPPC Linker target settings panel
- By modifying values in the linker command file (the file in your project that has the extension `.lcf`)

NOTE To use a linker command file, you must select the Use Linker Command File checkbox on the EPPC Linker target settings panel.

For more information, see the *Targeting* manual for your target processor.

The stack

In the default implementation, the MetroTRK stack resides in high memory and grows downward. The default implementation of MetroTRK requires a maximum of 8KB of stack space.

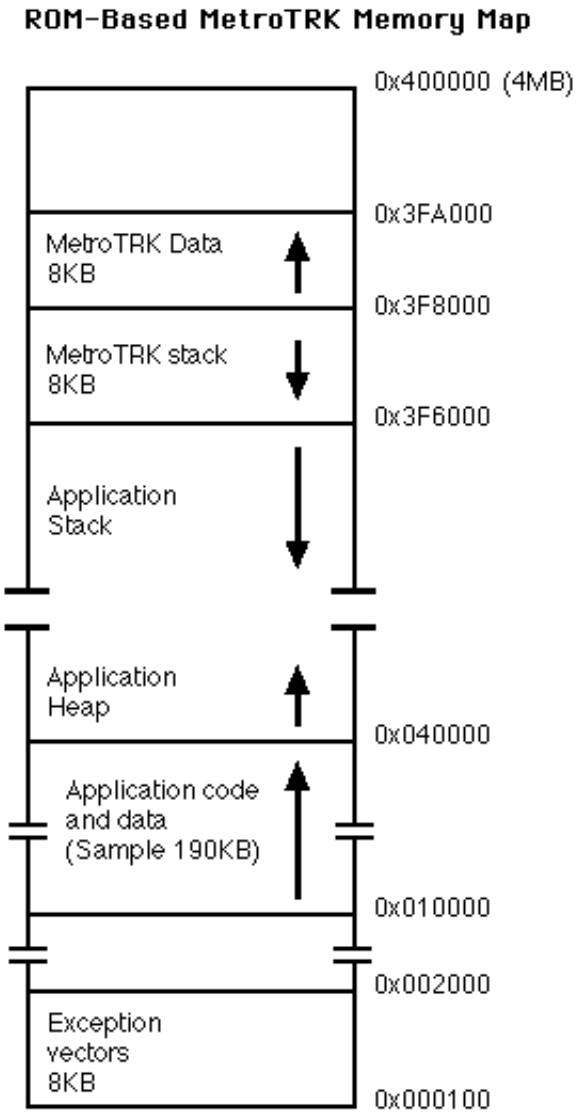
For example, on the Motorola 8xx ADS and Motorola 8xx MBX boards, the MetroTRK stack resides between the address 0x3F6000 and 0x3F8000. (For more information, see "MetroTRK Memory Map" on page 168.

You can change the location of the stack section by modifying settings on the EPPC Linker target settings panel and rebuilding the MetroTRK project. For more information, see the *Targeting* manual for your target processor.

MetroTRK Memory Map

Figure D.1 shows a sample map of RAM memory sections as configured when running MetroTRK with a sample target application on the Motorola 8xx MBX boards.

Figure D.1 MetroTRK memory map (Motorola 8xx MBX board)



PowerPC-Specific Exception Handling

MetroTRK initializes the memory system before copying the exception vectors to RAM because this operation depends on safe memory access. The location in memory of the exception vectors is

processor-specific. By default, MetroTRK responds to certain exceptions and notifies the debugger of all other exceptions.

The vector-copying process is not board-dependent because the processor determines the memory locations of the exception vectors. Consequently, the vector-copying process does not differ for differing board configurations.

For PowerPC, `InitializeTarget()`, which resides in `mpc_5xx.c`, `mpc_7xx_603e.c`, or `mpc_8xx.c`, calls `__copy_vectors()`, which resides in `copy_vectors.c`, to copy the exception vectors to RAM.

You can customize exception handling for PowerPC so that your application handles exceptions. However, MetroTRK must receive all the following types of exceptions:

- `Program Error` (used to signal program termination)
- `Trace` (used for stepping)
- `Software Emulation` (used for breakpoints)

Your application can handle any exception other than those in the preceding list.

WARNING! On PowerPC, MetroTRK must handle `Software Emulation` exceptions for breakpoints to work properly.

The following file contains the exception definitions for PowerPC target boards:

`Processor\ppc\Export\ppc_except.h`

For more information, see "Exception vectors" on page 167.

PowerPC-Specific Register Definitions

The following files define all PowerPC registers with mnemonic names and the sizes (bit-lengths) of the registers in the different register blocks:

- `Processor\ppc\Export\ppc_reg.h`

- `Processor\ppc\Export\m5xx_reg.h`
- `Processor\ppc\Export\m7xx_m603e_reg.h`
- `Processor\ppc\Export\m8xx_reg.h`

The `ppc_reg.h` file contains definitions that are common to all the PowerPC boards; the other files contain definitions that are board-specific.

PowerPC-Specific Version Location

For PowerPC, the `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` constants reside in the following file:

`Processor\ppc\Generic\ppc_version.h`

PowerPC-Specific Locations of target.h

The location of `target.h` for supported reference boards differs. The following list shows the PowerPC-specific directory paths for `target.h`:

- `Processor\ppc\Board\cogent\cma102`
- `Processor\ppc\Board\motorola\mpc_8260_vads`
- `Processor\ppc\Board\motorola\mpc_8xx_ads`
- `Processor\ppc\Board\motorola\mpc_8xx_mbx`
- `Processor\ppc\Board\motorola\mpc_5xx_evb`
- `Processor\ppc\Board\motorola\yellowknife_x4`
- `Processor\ppc\Board\motorola\excimer`
- `Processor\ppc\Board\motorola\mpc_555_etas`
- `Processor\ppc\Board\phytec\minimodul_ppc_505_509`

Unsupported Step Over Feature

MetroTRK does not support stepping over instructions when debugging for the PowerPC processor. (However, MetroTRK does support stepping through instructions.)

E

M•Core-Specific Information

This appendix provides MetroTRK-related information that is specific to the M•Core processor.

This appendix contains the following topics:

Supported M•Core Target Boards

MetroTRK currently supports the following target boards:

- MMCCMB1200 Controller and Memory Board (abbreviated as CMB1200)
- MMCEVB1200 Evaluation Board (abbreviated as EVB1200)
- MMCCMB2080 Controller and Memory Board (abbreviated as CMB2080)
- MMCEVB2080 Evaluation Board (abbreviated as EVB2080)
- MMCCMB2107 Controller and Memory Board (abbreviated as CMB2107
- MMCEVB2107 Evaluation Board (abbreviated as EVB2107)

- MMCCMB2103 Controller and Memory Board (abbreviated as CMB2103)
- MMCEVB2103 Evaluation Board (abbreviated as EVB2103)

M•Core-Specific Location of __reset

When the target board performs its hardware power-up sequence, it executes the __reset function first. The location of __reset differs depending on the target board (Table E.1).

Table E.1 M•Core-specific location of __reset

Target Board	Location of __reset
CMB1200	`\Processor\mcore\board\motorola\cmb_evb_1200`
EVB1200	
CMB2080	`\Processor\mcore\board\motorola\cmb_evb_2080`
EVB2080	
CMB2107	`\Processor\mcore\board\motorola\cmb_evb_2107`
EVB2107	
CMB2103	`\Processor\mcore\board\motorola\cmb_evb_2103`
EVB2103	

NOTE Some reference boards use a function called `__reset()`; other reference boards use assembly language code that is labeled `__reset`.

M•Core-Specific Memory Locations

This section discusses the default memory locations of the MetroTRK code and data sections and of your target application for M•Core.

This section contains the following topics:

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains the following topics:

Exception vectors

All exception vectors reside in the supervisor address space and are accessed using program relative references. Only the reset and soft reset are fixed in the memory map of the processor.

During the board initialization process, MetroTRK sets the VBR system register to point to the start of the RAM exception vector, which differs depending on the target board (Table E.2).

Table E.2 Starting address of the RAM exception vector

Target Board	Starting Address of the RAM Exception Vector
CMB1200	0x30009000
EVB1200	
CMB2080	0x02000000
EVB2080	

Target Board	Starting Address of the RAM Exception Vector
CMB2107	0x81007000
EVB2107	
CMB2103	0x80800000
EVB2103	

After initialization completes, you can relocate the base address of the exception vector table after reset by programming the VBR (Vector Base Register).

Data and code sections

In the default implementation of MetroTRK, no MetroTRK code exists in RAM because the code executes directly from ROM. For example, for the CMB/EVB1200 board, the MetroTRK code section resides in the flash memory starting at address 0x2D000000.

At startup, MetroTRK copies its initialized data from ROM to RAM. For example, the data section for the CMB/EVB1200 board, by default, resides in the address range from 0x30004000 to 0x300069BC.

You can change the location of the data and code sections in your MetroTRK project using one of the following methods:

- By modifying settings in the Section Mappings target settings panel
- By modifying values in the linker command file (the file in your project that has the extension `.cmd`)

NOTE If your project contains a linker command file, CodeWarrior uses the values in the file rather than any values you set in the Section Mappings target settings panel.

The stack

In the default implementation, the MetroTRK stack resides in high memory and grows downward. The default implementation of MetroTRK requires a maximum of 8KB of stack space. The linker automatically assigns the beginning address of the stack.

For example, for the CMB/EVB1200 board, the size of the MetroTRK stack is 8KB, and, by default, the stack starts at the address 0x300079C0 and grows downward to the address 0x300059C0.

MetroTRK Memory Map

The MetroTRK memory map differs depending on the target board. This section contains the following topics:

- Memory map: CMB1200 and EVB1200 target boards
- Memory map: CMB2080 and EVB2080 target boards
- Memory map: CMB2107 and EVB2107 target boards
- Memory map: CMB2103 and EVB2103 target boards

Memory map: CMB1200 and EVB1200 target boards

Table E.3 shows the default address ranges that MetroTRK uses for its stack, data, and exception vectors for the CMB1200 and EVB1200 target boards.

Table E.3 MetroTRK memory map for CMB1200 and EVB1200 boards

Memory Type	Address Range
Stack	0x300059C0 - 0x300079C0
Data	0x30004000 - 0x300069BC
Exception vector	0x30009000 - 0x300091FC

Table E.4 shows the default address ranges that are available for the target application when using the CMB1200 and EVB1200 target boards.

Table E.4 Target Application SRAM Locations

Memory Type	Address Range
Target application	0x2F000000 - 0x2F0FFFFF
Target application	0x30002000 - 0x30003FFF
Target application	0x30007000 - 0x30007FFF
Target application	0x30008000 - 0x3000FFFF (echo of 0x30000000 - 0x30007FFF)

Memory map: CMB2080 and EVB2080 target boards

Table E.5 shows the default address ranges that MetroTRK uses for its stack, data, and exception vectors for the CMB2080 and EVB2080 target boards.

Table E.5 MetroTRK memory map for CMB2080 and EVB2080 boards

Memory Type	Address Range
Stack	0x02007968 - 0x02005968
Data	0x02004400 - 0x02007968
Exception vector	0x02000000 - 0x020001FC

The default address range that is available for the target application when using the CMB2080 and EVB2080 target boards follows:

```
0x02007968 - 0x020FFFF
```

Memory map: CMB2107 and EVB2107 target boards

Table E.6 shows the default address ranges that MetroTRK uses for its stack, data, and exception vectors for the CMB2107 and EVB2107 target boards.

Table E.6 MetroTRK memory map for CMB2107 and EVB2107 boards

Memory Type	Address Range
Stack	0x81009878 - 0x8100B878
Data	0x81007300 - 0x81009870
Exception Vector	0x81007000 - 0x810071FC

Table E.7 shows the default address ranges that are available for the target application when using the CMB2107 or EVB2100 target boards.

Table E.7 Application address range for CMB2107 and EVB2107 boards

Target Board	Application Address Range
CMB2107	0x8100C000 - 0x811FFFFF
EVB2107	0x8100C000 - 0x810FFFFF

Memory map: CMB2103 and EVB2103 target boards

Table E.8 shows the default address ranges that MetroTRK uses for its stack, data, and exception vectors for the CMB2103 and EVB2103 target boards.

Table E.8 MetroTRK memory map for CMB2103 and EVB2103 boards

Memory Type	Address Range
Stack	0x80805858 - 0x80807858
Data	0x80804400 - 0x80807858
Exception Vector	0x80800000 - 0x808001FC

The default address range that is available for the target application when using the CMB2103 and EVB2103 target boards follows:

```
0x80808000 - 0x808FFFFF
```

M•Core-Specific Exception Handling

MetroTRK initializes the memory system before copying the exception vectors to RAM because this operation depends on safe memory access. The location in memory of the exception vectors is processor-specific. By default, MetroTRK responds to certain exceptions and notifies the debugger of all other exceptions.

The vector-copying process is not board-dependent because the processor determines the memory locations of the exception vectors. Consequently, the vector-copying process does not differ for differing board configurations.

For M•Core, the function that copies the exception vector from ROM to RAM is `__copy_vectors()`, which resides in the following file:

```
\Processor\mcore\generic\__copy_vectors.c
```

You can customize exception handling for M•Core; your target application can handle any exception. However, the M•Core version of MetroTRK must receive all `Breakpoint` and `Trace` exceptions. Therefore, after handling a `Breakpoint` or `Trace` exception, the interrupt handler in your target application must call the MetroTRK exception-handling code so that MetroTRK also can handle the exception. (This is called sharing the exception.)

For other exceptions, your target application can replace the MetroTRK interrupt handler with its own interrupt handler in the RAM exception vector. For more information, see Table E.2 on page 175.

The following file defines the exceptions for M•Core:

```
processor\mcore\export\mcore_except.h
```

For more information, see “Exception vectors” on page 175.

M•Core-Specific Register Definitions

The following file defines all M•Core registers with mnemonic names and the sizes (bit-lengths) of the registers in the different register blocks:

`processor\mcore\export\mcore_reg.h`

M•Core-Specific Version Location

For M•Core, the `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` constants reside in the following file:

`Processor\mcore\generic\mcore_version.h`

M•Core-Specific Location of target.h

The location of `target.h` differs among boards (Table 5.1).

Table 5.1 M•Core-specific location of target.h

Target Board	Location of target.h
CMB1200	`\Processor\mcore\board\motorola\cmb_evb_1200`
EVB1200	
CMB2080	`\Processor\mcore\board\motorola\cmb_evb_2080`
EVB2080	
CMB2107	`\processor\mcore\board\motorola\cmb_evb_2107`
EVB2107	
CMB2103	`\Processor\mcore\board\motorola\cmb_evb_2103`
EVB2103	

M•Core-Specific Data Transmission Rates

MetroTRK uses a different default data transmission rate for different M•Core target boards (Table 5.2).

Table 5.2 M•Core-specific data transmission rates

Target Board	Default Data Transmission Rate
CMB1200	115200
EVB1200	
CMB2080	57600
EVB2080	
CMB2107	115200
EVB2107	
CMB2103	115200
EVB2103	

Interrupt-Driven Communication for M•Core

Interrupt-driven communication supports two features that are not available with the serial polling method of communication:

- If the target application executes an infinite loop, the debugger can send a Stop command to stop the application from running.
- The debugger can read and write memory while the target application is running, which allows the debugger to update the information that it displays more often.

For any target board for which MetroTRK uses interrupt-driven communication, the value of `TRK_TRANSPORT_INT_DRIVEN` must be 1. For currently supported target boards, if MetroTRK uses interrupt-driven communication, the file `target.h` defines the correct value. However, if you customize MetroTRK to work with additional boards, you must define the value of `TRK_TRANSPORT_INT_DRIVEN` as 1 to indicate that MetroTRK uses interrupt-driven communication (if appropriate). (For more information, see "Customizing MetroTRK to Be Interrupt-Driven" on page 54.)

For M•Core, MetroTRK currently supports interrupt-driven communication for the following target boards:

- CMB1200
- EVB1200
- CMB2107
- EVB2107

Table E.9 shows the M•Core-specific location of several functions related to interrupt-driven communication with MetroTRK. (For more information, see "MetroTRK Function Reference" on page 103.)

Table E.9 M•Core-specific interrupt-driven communication functions

Function Name	Location
`InitializeIntDrivenUART()`	`\MetroTRK\Transport\mcore\common\uart.c and uart_interface.c`
`InterruptHandler`	`\MetroTRK\Processor\mcore\generic\targimpl.c`
`SwapAndGo`	`\MetroTRK\Processor\mcore\generic\targimpl.c`
`_UARTInterruptHandler`	`\MetroTRK\Transport\mcore\common\targ_mcore.s`

F

NEC V8xx-Specific Information

This appendix provides MetroTRK-related information that is specific to the NEC V8xx processor family.

This appendix contains the following topics:

NEC V8xx-Specific Location of __reset

For the currently supported NEC V8xx reference boards, the location of `__reset` follows:

```
Processor\necv8xx\Generic\__reset.s
```

NOTE Some reference boards use a function called `__reset()`; other reference boards use assembly language code that is labeled `__reset`.

NEC V8xx-Specific Memory Locations

This section discusses the default memory locations of MetroTRK code and data and of your target application for the NEC V8xx processor family.

This section contains the following topics:

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains the following topics:

Exception vectors

The exception handlers reside at a fixed location, usually in ROM.

NOTE Do not change the location of the exception vectors in ROM because the processor expects the exception vectors to reside at the set location.

However, MetroTRK creates a mirror image of the entire interrupt/exception handler table in DRAM from 0xFFFFFE00 to 0xFFFFFFFF. The location of the mirror image differs between NEC target processor families.

NOTE You can change the starting location of the mirror image. To do so, change the value of RAM_VECTOR_DEST in the __rom_vectors.s file in the following directory and then rebuild MetroTRK:

```
Processor\necv8xx\Generic
```

Table 5.3 shows the default location of the mirror image for each NEC V8xx processor family.

Table 5.3 Interrupt/Exception Handler Table Mirror Image Addresses

Processor Type	Interrupt/Exception Handler Table Mirror Image Addresses
NEC V830 processor family (V830, V831, V832)	0x40000000 - 0x400001FF
NEC V810 processor family	0x00000000 - 0x000001FF
NEC V850 processor family	0x00FFE000 - 0x00FFE270
NEC V850E processor family	0x03FFE000 - 0x03FFE400

For example, to install a handler for a division-by-zero exception on the Midas RTE-V830-PC target board, you can install the handler at the RAM location 0x40000180.

Because MetroTRK mirrors the interrupt handlers as well as the exception handlers, you can overwrite them the same way. For the V830 family, you have another option; your software can set the register bit HWCC.IHA to 1 and set up interrupt handlers in the built-in instruction RAM.

For more information on customizing exception handling, see "NEC V8xx-Specific Exception Handling" on page 190.

Data and code sections

The locations of the code and data sections in memory for the ROM-based version of MetroTRK differ depending on the supported reference board.

For example, for the ROM-based version of MetroTRK for the RTE-Midas V8xx board, the following sections are adjacent in memory starting at the address 0xfe078000:

- `.data`
- `.sdata`
- `.sbss`
- `.bss`
- `.itext`

Also, the following sections are adjacent in ROM starting at 0xffffcab8 for the ROM-based version for the RTE-Midas V8xx board:

- `.text`
- `.const`
- `.sconst`

You can set the location of the data and code sections in the linker command file in your MetroTRK project. (The linker command file is the file in your project that has the extension `.lcf`.) For more information, see the *Targeting* manual for your target processor.

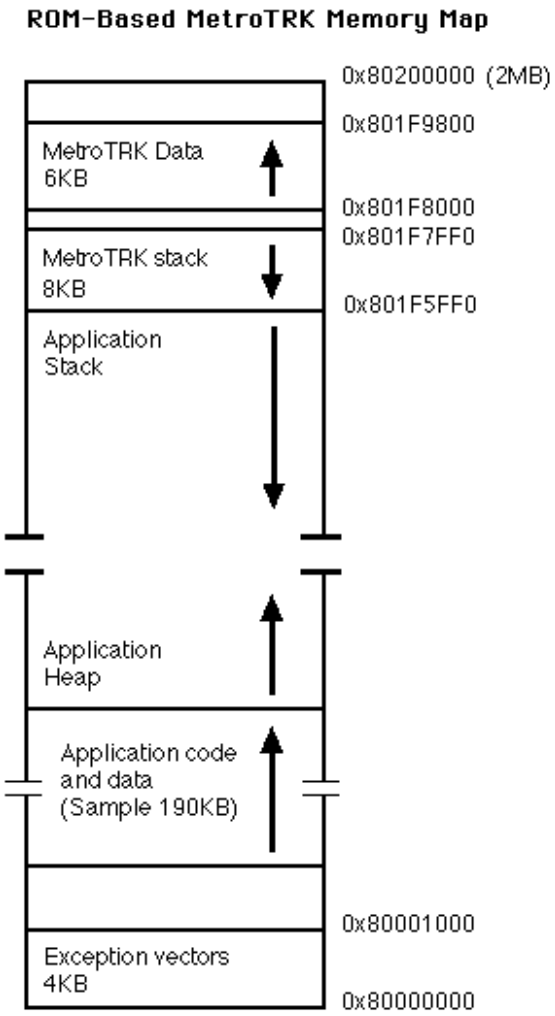
The stack

In the default implementation of MetroTRK, the MetroTRK stack resides in the `.bss` section. The default implementation of MetroTRK requires a maximum of 8KB of stack space.

MetroTRK Memory Map

Figure F.1 shows a sample map of RAM memory sections as configured when running MetroTRK with a sample target application on an RTE-Midas V8xx board.

Figure F.1 MetroTRK memory map (RTE-Midas V8xx board)



NEC V8xx-Specific Exception Handling

MetroTRK initializes the memory system before copying the exception vectors to RAM because this operation depends on safe memory access. The location in memory of the exception vectors is processor-specific. By default, MetroTRK responds to certain exceptions and notifies the debugger of all other exceptions.

The vector-copying process is not board-dependent because the processor determines the memory locations of the exception vectors. Consequently, the vector-copying process does not differ for differing board configurations.

For NEC V8xx, `InitializeTarget()`, which resides in `targimpl.c`, calls `__copy_vectors()`, which resides in `copy_vectors.c`, to copy the exception vectors to RAM.

You can customize exception handling for NEC V8xx target boards; your application can handle any exception. However, the NEC V8xx version of MetroTRK must receive certain exceptions. Therefore, after handling those exceptions, your application must call the MetroTRK exception-handling code so that MetroTRK also can handle the exception. (This is called sharing the exception.)

Table 5.4 shows the exceptions that the NEC V8xx version of MetroTRK must receive and handle for several NEC V8xx target boards.

Table 5.4 NEC V8xx exceptions that MetroTRK must handle

Target Board	Applicable Exceptions
NEC V810, NEC V830	TRAP 0x10 through TRAP 0x1f
NEC V850	TRAP 0x10
	INTCM4
NEC V850e	TRAP 0x10
	INTCM40
	ILGOP

Other than the exceptions shown in Table 5.4, your application can overwrite any other exceptions that it handles.

The following file contains the exception definitions for NEC V8xx target boards:

`Processor/necv8xx/Export/v8xx_except.h`

For more information, see "Exception vectors" on page 186.

NEC V8xx-Specific Register Definitions

The following file contains the V8xx register definition information:

`Processor/necv8xx/Export/v8xx_reg.h`

NEC V8xx-Specific Version Location

For NEC V8xx, the `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` constants reside in the following file:

`Processor\necv8xx\Generic\v8xx_version.h`

NEC V8xx-Specific Locations of target.h

The location of `target.h` for supported reference boards differs. The following list shows the NEC V8xx-specific directory paths for `target.h`:

- `Processor\necv8xx\Board\midas\rte_v832_pc`
- `Processor\necv8xx\Board\midas\rte_v850e_pc`
- `Processor\necv8xx\Board\midas\rte_v853_pc`
- `Processor\necv8xx\Board\midas\rte_v830_pc`
- `Processor\necv8xx\Board\midas\rte_v831_pc`
- `Processor\necv8xx\Board\midas\rte_v821_pc`

- `Processor\necv8xx\Board\nec_mini\rt_v853`
- `Processor\necv8xx\Board\nec_mini\rt_v831`
- `Processor\necv8xx\Board\cosmo\ceb_v850e_t`

Interrupt-Driven Communication for NEC V8xx

For all target boards, you must define the value of `TRK_TRANSPORT_INT_DRIVEN` as 1 to indicate that MetroTRK uses interrupt-driven communication. (For more information, see "Customizing MetroTRK to Be Interrupt-Driven" on page 54.)

NOTE For NEC V8xx boards for which interrupt-driven MetroTRK communication is enabled, `TRK_TRANSPORT_INT_DRIVEN` and any needed transport interrupt keys reside in `__config.i`. (For NEC V8xx, the transport interrupt key is value of the offset into the exception handler table.)

Table F.1 shows the NEC V8xx-specific location of several functions related to interrupt-driven communication with MetroTRK. (For more information, see "MetroTRK Function Reference" on page 103.)

Table F.1 NEC V8xx-specific interrupt-driven communication functions

Function Name	Location
`InitializeIntDrivenUART()`	`Transport\uart\scx26x1\uart.c`
`_InterruptHandler`	`Processor\necv8xx\Generic\targimpl.s`
`_SwapAndGo`	`Processor\necv8xx\Generic\targimpl.s`
`_UARTInterruptHandler`	`Transport\necv8xx\Common\targ_NECv85x.s`

G

68K-Specific Information

This appendix provides MetroTRK-related information that is specific to the 68K processor.

This appendix contains the following topics:

68K-Specific Location of __reset

For the currently supported 68K reference boards, Table G.1 shows the location of `__reset`.

NOTE Some reference boards use a function called `__reset()`; other reference boards use assembly language code that is labeled `__reset`.

Table G.1 68K-specific __reset location

Reference Board	File Containing __reset
Motorola M68328 ADS	`Processor\M68K\Board\motorola\m68328_ads\Reset.s`
Motorola M68EZ328 ADS	`Processor\M68\Board\motorola\m68ez328_ads\Reset.s`

68K-Specific Memory Locations

This section discusses the default memory locations of the MetroTRK RAM sections and of your target application for 68K.

This section contains the following topics:

Locations of MetroTRK RAM Sections

Several MetroTRK RAM sections exist. You can reconfigure some of the MetroTRK RAM sections.

This section contains the following topics:

Exception vectors

The location of the exception vectors in RAM is a set characteristic of the processor. On 68K, the exception vector must start at 0x00000000 (which is in low memory) and spans 1024 bytes to end at 0x00000400. For a ROM-based MetroTRK, the MetroTRK initialization process copies the exception vectors from ROM to RAM.

NOTE Do not change the location of the exception vectors because the processor expects the exception vectors to reside at the set location.

Data and code sections

In the default implementation of MetroTRK, which is ROM-based, no code exists in RAM because the code executes directly from ROM.

You can set the location of the data and code sections in the linker command file in your MetroTRK project. (The linker command file is the file in your project that has the extension `.lcf`.) For more information, see the *Targeting* manual for your target processor.

The stack

In the default implementation, the MetroTRK stack resides in high memory and grows downward. The default implementation of MetroTRK requires a maximum of 8KB of stack space.

You can set the location of the stack section in the linker command file in your MetroTRK project. (The linker command file is the file in your project that has the extension `.lcf`.) For more information, see the *Targeting* manual for your target processor.

MetroTRK Memory Map

As an example, Table G.2 shows the default address ranges that MetroTRK uses for its stack, data, and exception vectors for the M68328 ADS target board.

Table G.2 68K MetroTRK memory map

Memory Type	Address Range
Data and stack	0x00070000 - 0x00078000
Exception vectors	0x00000000 - 0x00000400

68K-Specific Exception Handling

MetroTRK initializes the memory system before copying the exception vectors to RAM because this operation depends on safe memory access. The location in memory of the exception vectors is processor-specific. By default, MetroTRK responds to certain exceptions and notifies the debugger of all other exceptions.

The vector-copying process is not board-dependent because the processor determines the memory locations of the exception vectors. Consequently, the vector-copying process does not differ for differing board configurations.

For 68K, `m68k_step.s` contains assembly code labeled `_Init_InterruptHandler` that copies the exception vectors to RAM.

You can customize exception handling for 68K target boards. Your target application can handle any exceptions other than the following:

- TRAP 0
- TRAP 14
- TRAP 12
- Trace exceptions

The following file defines the exceptions for 68K target boards:

`Processor\M68k\Export\m68k_except.h`

For more information, see "Exception vectors" on page 194.

68K-Specific Register Definitions

The following file defines all 68K registers with mnemonic names and the sizes (bit-lengths) of the registers in the different register blocks:

`Processor\M68k\Export\m68k_reg.h`

68K-Specific Version Location

For 68K, the `DS_KERNEL_MAJOR_VERSION` and `DS_KERNEL_MINOR_VERSION` constants reside in the following file:

`Processor\M68k\Generic\m68k_version.h`

68K-Specific Locations of target.h

The location of `target.h` for supported reference boards differs. The following list shows the 68K-specific directory paths for `target.h`:

- `Processor\M68K\Board\motorola\m68ez328_ads`
- `Processor\M68K\Board\motorola\m68328_ads`

Index

Symbols

A

B

C

D

E

F

N

O

P

Q

R

S

T

U

V

W

Z