CodeWarrior Target Resident Kernel Reference

Document Number: CWPATRKUG
Rev. 10.5.1, 01/2016

Contents

Section number	Title	Page

Chapter 1 Introduction

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

This chapter includes these topics:

- Overview of this Manual
- Related Documentation

1.1 Overview of this Manual

This manual describes CodeWarrior TRK and explains how to customize it for use with your hardware configuration. The table below describes the information contained in each chapter of this manual.

NOTE

In addition to the chapters described in the table below, this manual may have one or more additional processor-specific appendixes, depending on the product you purchased. Refer to the table of contents in the front of this manual to determine whether your version of this manual contains any processor-specific appendixes.

Table 1-1. Contents of this Manual

Chapter Name	Description
Introduction	(this chapter)
CodeWarrior TRK Concepts	Provides an overview of CodeWarrior TRK, describes the various tasks CodeWarrior TRK performs, and how these tasks are implemented.

Table continues on the next page...

Table 1-1. Contents of this Manual (continued)

Chapter Name	Description
CodeWarrior TRK Communications	Describes the CodeWarrior TRK communication protocol. This chapter is useful for debugging CodeWarrior TRK and for those developing software that communicates with CodeWarrior TRK.
Customizing CodeWarrior TRK	Provides details about the specific areas where you can customize or re-target CodeWarrior TRK to work with your hardware configuration.
Debug Message Interface Reference	Describes the CodeWarrior TRK debug message interface.
CodeWarrior TRK Function Reference	Describes CodeWarrior TRK functions that may be relevant for developers who wish to customize CodeWarrior TRK to work with new target boards.
AppTRK Reference	Describes the Linux application version of CodeWarrior TRK (AppTRK), how it differs from the bareboard version of CodeWarrior TRK, and how to use AppTRK.

NOTE

For basic information about loading and using CodeWarrior TRK with the CodeWarrior IDE (information that differs among sets of CodeWarrior tools), refer to the *Targeting* manual for your particular target processor.

If this is your first time customizing the bareboard version of CodeWarrior TRK, we recommend you read these chapters in this order:

- CodeWarrior TRK Concepts
- CodeWarrior TRK Communications
- Customizing CodeWarrior TRK

If you have previously customized the bareboard version of CodeWarrior TRK previously, you can proceed directly to the Customizing CodeWarrior TRK.

To learn about the Linux application version of CodeWarrior TRK, read this chapter:

- AppTRK Reference

1.2 Related Documentation

This section lists related documentation for CodeWarrior TRK. All CodeWarrior manuals mentioned in this section reside in this directory on your CodeWarrior CD:

CodeWarriorCD`\Documentation`

1.2.1 Other CodeWarrior Documentation

For information about using CodeWarrior and CodeWarrior TRK with a particular target processor, see the *Targeting* manual for your target processor.

1.2.2 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 CodeWarrior TRK distribution or on this Web page:

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

Chapter 2
CodeWarrior TRK Concepts

This chapter describes the architecture of CodeWarrior TRK and how CodeWarrior TRK works. This chapter includes these topics:

- CodeWarrior TRK Architecture
- CodeWarrior TRK Memory Layout
- CodeWarrior TRK Initializations

2.1 CodeWarrior TRK Architecture

This section describes these CodeWarrior TRK components:

- CodeWarrior TRK Core
- CodeWarrior TRK Execution States
- Request and Notification Handling
- Message Queues

2.1.1 CodeWarrior TRK Core

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

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

2.1.2 CodeWarrior TRK Execution States

CodeWarrior TRK has two main operating states, the *message-handling state* and the *event-waiting state*.

A Reset request or a hardware reset causes CodeWarrior TRK to enter its board initialization state. After the board initializations complete, CodeWarrior TRK enters its message-handling state. In this state, CodeWarrior TRK continuously services requests from the debugger. CodeWarrior TRK is in a continuous loop, waiting for requests from the debugger. It passes each request that it receives to an appropriate handler function.

NOTE

The target application does not execute while CodeWarrior TRK is in the message-handling state.

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

NOTE

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

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

The figure below shows the state diagram for CodeWarrior TRK.

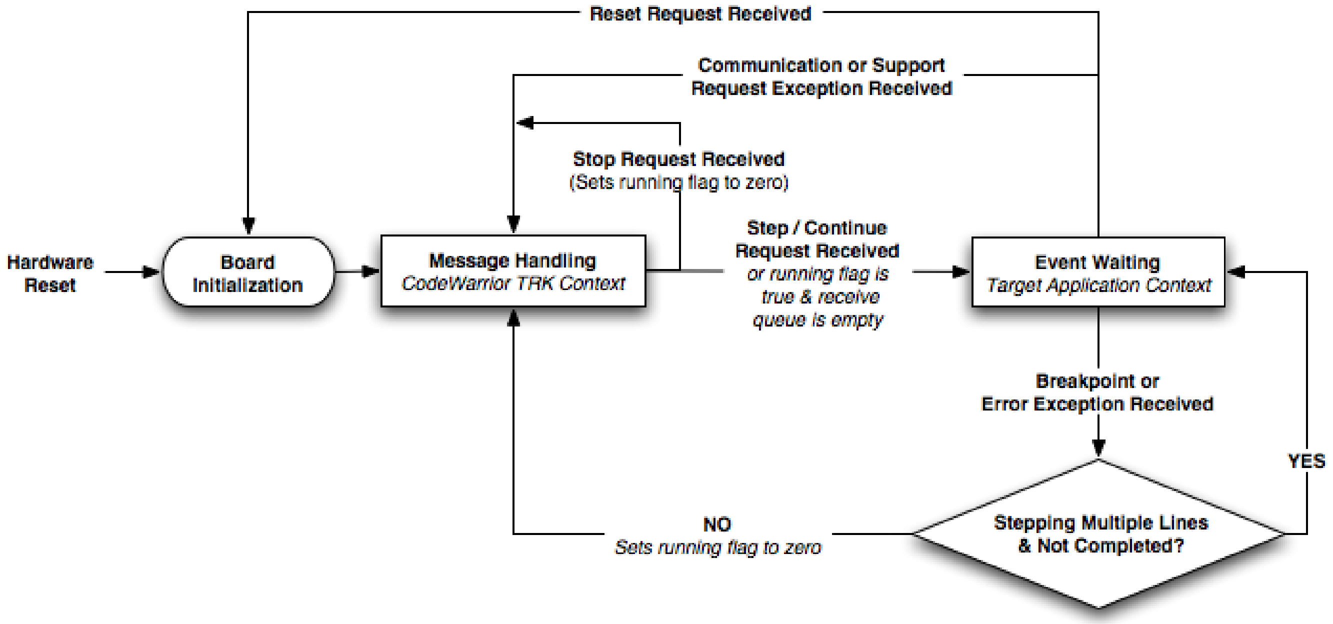

Reset Request Received
Communication or Support Request Exception Received
Stop Request Received (Sets running flag to zero)
Hardware Reset
Board Initialization
Message Handling CodeWarrior TRK Context
Step / Continue Request Received or running flag is true & receive queue is empty
Event Waiting Target Application Context
Breakpoint or Error Exception Received
YES
NO Sets running flag to zero
Stepping Multiple Lines & Not Completed?

Figure 2-1. CodeWarrior TRK State Diagram

2.1.3 Message Queues

While in the message-handling state, CodeWarrior TRK constantly monitors the serial line for incoming requests and stores each one in an incoming message queue. CodeWarrior TRK also maintains another message queue for outgoing messages. To send a message to the debugger, CodeWarrior TRK 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 CodeWarrior TRK, 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 CodeWarrior TRK to use an event-driven design. The CodeWarrior TRK core centrally dispatches all communication with the debugger, which passes first through the messages queues.

The figure below shows how data flows through CodeWarrior TRK when it is in the message-handling state, including the message queues described in this section. In the event-waiting state, CodeWarrior TRK is inactive while waiting for the next exception to occur. The message queues are not dependent on the target board configuration.

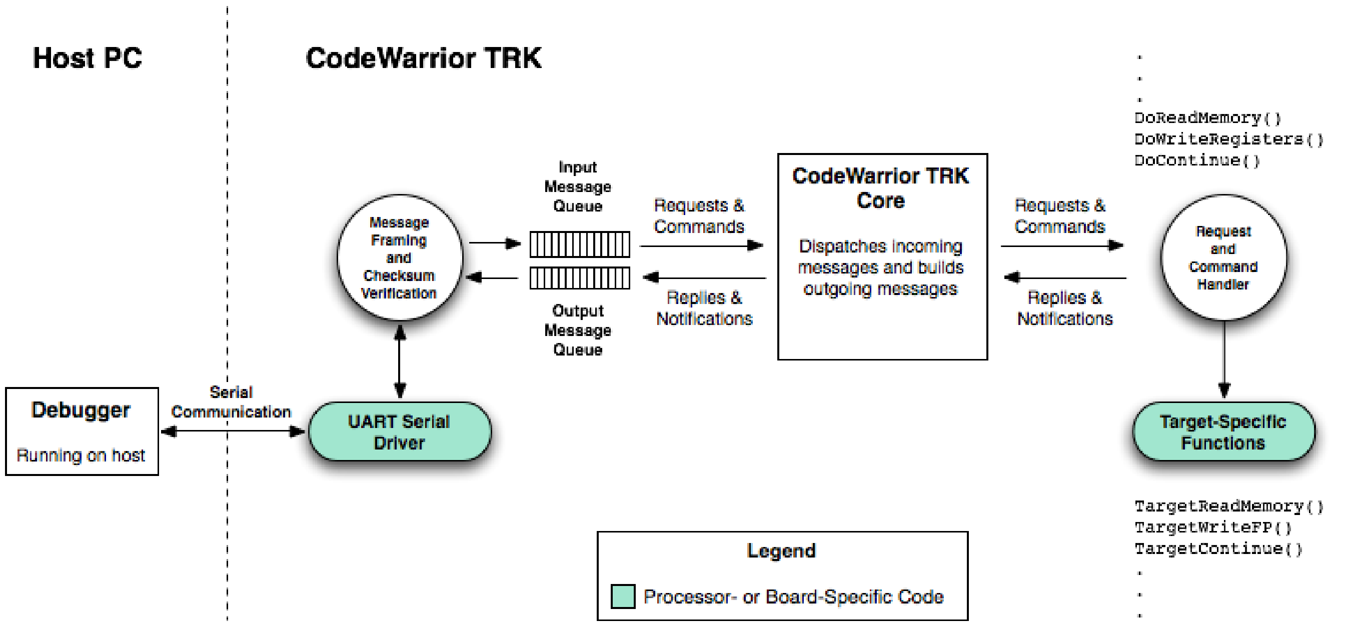


Figure 2-2. CodeWarrior TRK Data-Flow Diagram

2.1.4 Request and Notification Handling

A set of handler functions separate from the CodeWarrior TRK core comprises another module of CodeWarrior TRK 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 this directory:

```
CWTRKDir\Processor\
ProcessorType\Board
```

For more information, see Customizing Debug Services.

2.2 CodeWarrior TRK Memory Layout

This section contains these topics:

- CodeWarrior TRK RAM Sections
- Target Application RAM Sections

2.2.1 CodeWarrior TRK RAM Sections

The table below shows the memory sections that exist in RAM when you run CodeWarrior TRK from ROM.

Table 2-1. CodeWarrior TRK 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 CodeWarrior TRK, CodeWarrior TRK copies any initial values from ROM to RAM. CodeWarrior TRK 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	CodeWarrior TRK requires at least 8KB of RAM for its stack, which is also the maximum amount of RAM that the stack occupies.

2.2.2 Target Application RAM Sections

Specify target application memory sections so that they do not overwrite the CodeWarrior TRK memory sections. For more information, see CodeWarrior TRK RAM Sections.

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 CodeWarrior TRK. You then can place the stack of the target application below the stack of CodeWarrior TRK. You must allow enough room for the CodeWarrior TRK stack to grow downward.

For more information, see Customizing Memory Locations.

2.3 CodeWarrior TRK Initializations

CodeWarrior TRK 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 `_in it_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 CodeWarrior TRK is targeted) `_init_board` moves ROM in memory. In this case, you can directly reset the program counter in `_reset()` following `_init_board`.

The listing below shows the preceding initialization sequence.

Listing 2-1. CodeWarrior TRK Initialization Sequence

```
 _reset() calls:
   _init_
processor
   _init_board

   (Before jump to _start, _reset() can contain other code

   performs direct initialization.)

   _start
```

NOTE

The `_reset()` function either uses the initialization sequence described 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 about the location of _reset() or _reset, see the processor-specific appendixes in this manual.

Chapter 3
CodeWarrior TRK Communications

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

CodeWarrior TRK continuously communicates with the debugger. This communication has these levels:

- Transport Level
- Framing Level
- Debug Message Interface Level

NOTE

When describing CodeWarrior TRK 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 CodeWarrior TRK can be either a sender or a receiver, depending on the action each is performing at a particular moment.

The figure below shows the interaction of the communication levels.

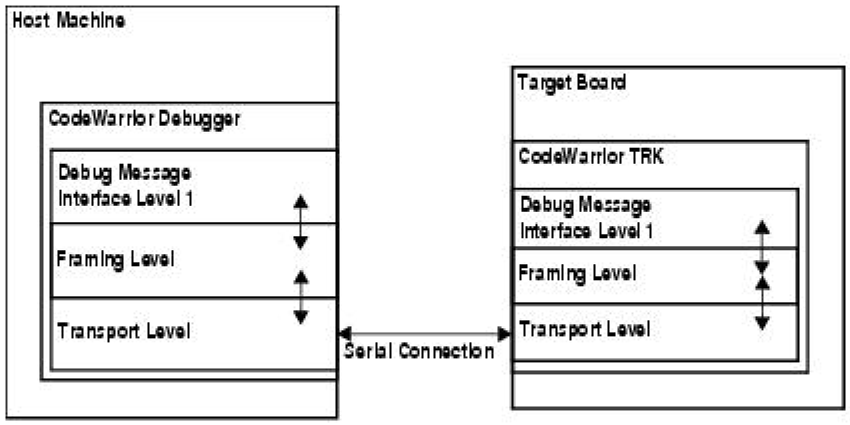


Figure 3-1. CodeWarrior TRK Communication Levels

3.1 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 or Ethernet connection.

For maximum portability, the low-level code that drives the serial/Ethernet controller is separated from the CodeWarrior TRK core. CodeWarrior TRK 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.

You must configure both CodeWarrior TRK 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.

This section contains these topics:

- Serial Communications Settings
- Data Transmission Rate

3.1.1 Serial Communications Settings

Your target board must have a serial port so that CodeWarrior TRK can communicate with the debugger running on the host computer. For most target boards, the data transmission rate when communicating with CodeWarrior TRK 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.)

CodeWarrior TRK usually communicates using these 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.

3.1.2 Data Transmission Rate

The default implementation of CodeWarrior TRK 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 kilobytes. Consequently, if you are using the Solaris-hosted debugger, you must set the data transmission rate in CodeWarrior TRK to 38.4 kilobytes, 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.

3.2 Framing Level

The framing level:

- is responsible for reliably transporting messages between CodeWarrior TRK and the debugger

- transmits the messages as arbitrarily sized segments of data
- places each data segment in a CodeWarrior TRK 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 CodeWarrior TRK data frame contains one message. (For more information, see Debug Message Interface Level.)

NOTE

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

This section contains these topics:

- CodeWarrior TRK Data Frames
- Checksum Values
- Escape Sequences
- Reliable Message Delivery

3.2.1 CodeWarrior TRK Data Frames

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

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

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

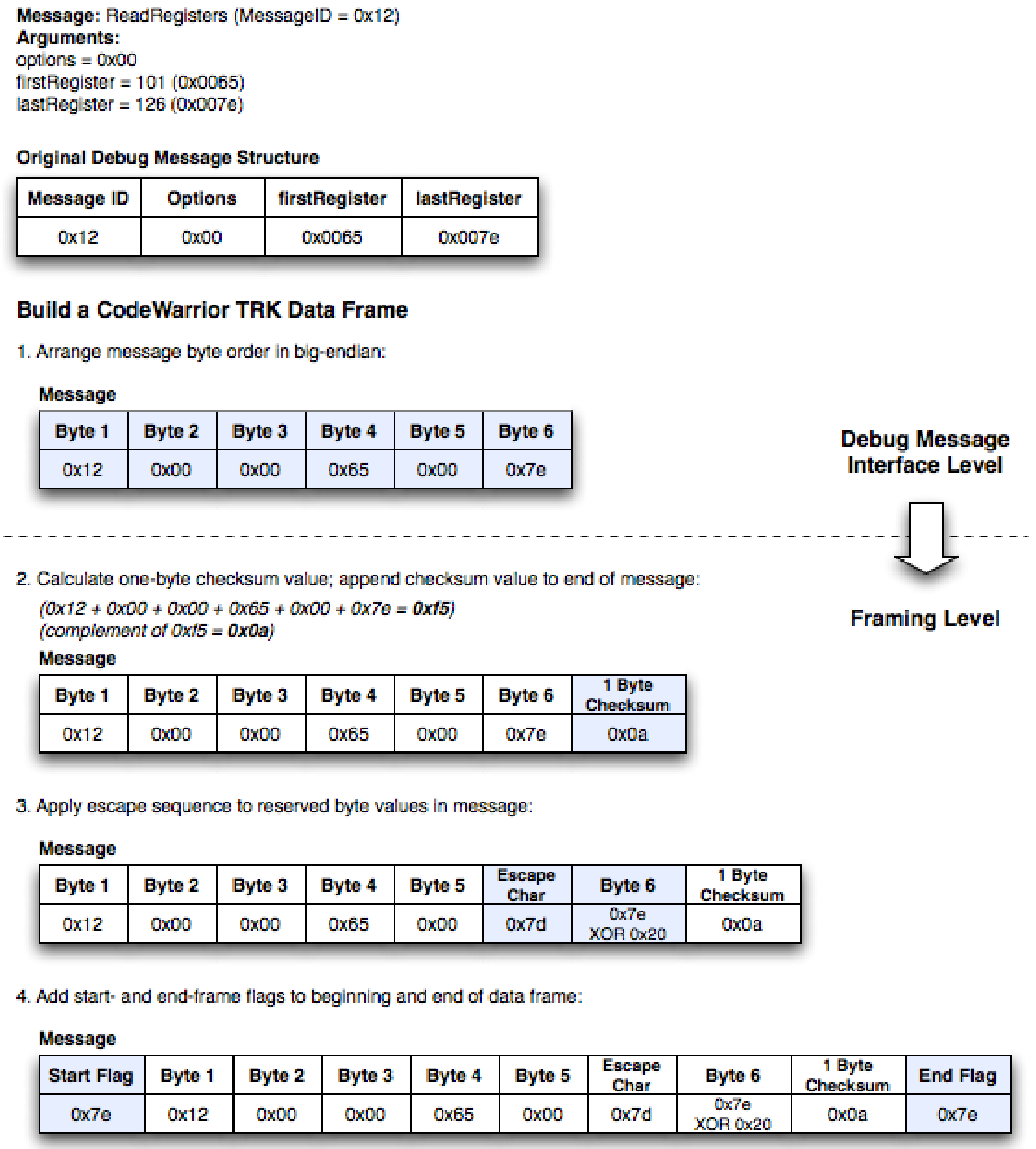
   NOTE

   Step 1 executes at the debug message interface level rather than the framing level, as shown in the figure below.

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 CodeWarrior TRK data frame.

The figure below shows how CodeWarrior TRK and the debugger create a CodeWarrior TRK data frame.



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

Original Debug Message Structure

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

Build a CodeWarrior TRK Data Frame

1. Arrange message byte order in big-endian:

Message

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

2. Calculate one-byte checksum value; append checksum value to end of message:

*(0x12 + 0x00 + 0x00 + 0x65 + 0x00 + 0x7e = **0xf5**)*
*(complement of 0xf5 = **0x0a**)*

Message

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	1 Byte Checksum
0x12	0x00	0x00	0x65	0x00	0x7e	0x0a

3. Apply escape sequence to reserved byte values in message:

Message

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Escape Char	Byte 6	1 Byte Checksum
0x12	0x00	0x00	0x65	0x00	0x7d	0x7e XOR 0x20	0x0a

4. Add start- and end-frame flags to beginning and end of data frame:

Message

Start Flag	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Escape Char	Byte 6	1 Byte Checksum	End Flag
0x7e	0x12	0x00	0x00	0x65	0x00	0x7d	0x7e XOR 0x20	0x0a	0x7e

Figure 3-2. Creating a CodeWarrior TRK Data Frame

3.2.2 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.

By default, CodeWarrior TRK uses a single-byte checksum value. However, you can customize CodeWarrior TRK to use a two- or four-byte checksum value. For more information, see Customizing Checksum Values.

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 these sections, which describe how CodeWarrior TRK 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

CodeWarrior TRK contains implementations for computing checksum values. If you are developing your own debugger to communicate with CodeWarrior TRK, you may want to borrow your implementation directly from CodeWarrior TRK. These source files implement checksums:

```
CWTRKDir\Export\serframe.h
CWTRKDir\Transport\protocol\rfc_1662\rfc1662.c
```

3.2.2.1 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 CodeWarrior TRK data frame.

The listing below 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

```
#define    FCSBITSIZE FCS8
#if FCSBITSIZE == FCS8

    /* Definitions for 8-bit simple Frame Check Sequences */

    typedef unsigned char FCSType; /* 8-bit type */

    #define PPPINITFCS8     0x00    /* Initial FCS value */

    #define PPPGOODFCS8     0xFF    /* Good final FCS value */

    #define PPPINITFCS    PPPINITFCS8

    #define PPPGOODFCS    PPPGOODFCS8

    #define PPPCOMPFCS    0xFF    /* complement FCS value */

    #ifdef _cplusplus

        inline FCSType PPPFCS(FCSType fcs, unsigned char c)

        {

            return (fcs + c);

        }

    #else

        #define PPPFCS(fcs, c)  (fcs + c)

    #endif
```

3.2.2.2 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.

The listing below 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);

}
```

3.2.2.3 Using Multi-Byte Checksum Values

For multi-byte checksum values, CodeWarrior TRK uses an algorithm called FCS (Frame Check Sequence). Similar 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 CodeWarrior TRK 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 CodeWarrior TRK source code in this file provide details on calculating multi-byte FCS values:

`CWTRKDir\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`, CodeWarrior TRK or the debugger sends these bytes in the order shown:

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

3.2.3 Escape Sequences

The CodeWarrior TRK communications protocol has these 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 this 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.

3.2.4 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. For more information, see Reply Messages.

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.

3.2.4.1 Responding to a NAK Reply Message

The sender can receive a NAK reply for these reasons:

- The receiver did not receive the CodeWarrior TRK data frame correctly.
- The format of the CodeWarrior TRK 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 NAK Messages.

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

3.2.4.2 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 these 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 (CodeWarrior TRK 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), CodeWarrior TRK 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 CodeWarrior TRK. However, this amount may sometimes require adjustment.

3.2.4.3 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.

3.3 Debug Message Interface Level

The debug message interface level defines these messages that CodeWarrior TRK 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, CodeWarrior TRK can request that the debugger read information from a file and return the information to CodeWarrior TRK by sending a ReadFile request to the debugger. CodeWarrior TRK 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 CodeWarrior TRK. After receiving a request from the debugger, the CodeWarrior TRK core examines the fields of the message to determine which handler function to call. The CodeWarrior TRK 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 CodeWarrior TRK passes to its handler functions as parameters. For more information, see Debug Message Interface Reference and CodeWarrior TRK Function Reference.)

CodeWarrior TRK 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 and Debug Message Interface Reference.

This section contains these topics:

- Request and Notification Messages
- Reply Messages

3.3.1 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 about the structure of each request and notification message, see Debug Message Interface Reference.)

3.3.1.1 Alignment

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

3.3.1.2 Byte Order

Multi-byte data in debug messages uses big-endian byte order (most significant byte first). The table below 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]`

Table continues on the next page...

Table 3-1. Data in Big-endian Byte Order (continued)

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

3.3.1.3 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.

3.3.2 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 these topics:

- ACK Messages
- NAK Messages

3.3.2.1 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 CodeWarrior TRK 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 CodeWarrior TRK constant `kDSReplyNoError`). If the error code byte contains any other value, an error occurred.

The table below 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.
`0x10`	`kDSReplyUnsupportedCommandError`	The request was of an invalid type. You can query CodeWarrior TRK to determine which requests are supported. For more information, see SupportMask.
`0x11`	`kDSReplyParameterError`	The values of one or more fields in the message were incorrect. These 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. These 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.
`0x15`	`kDSReplyCWDSException`	An exception was generated while processing the request. These 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, these requests reply with the error code:

Table continues on the next page...

Table 3-2. Possible Error Codes in an ACK Reply Message (continued)

Value	Error Code Name	Description
		• Continue • FlushCache • ReadMemory • ReadRegisters • Step • WriteMemory • WriteRegisters This applies only if CodeWarrior TRK and the debugger are using interrupt-driven communication. Otherwise, CodeWarrior TRK 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 about return values, see the descriptions of individual debug messages in Debug Message Interface Reference.

3.3.2.2 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.)

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

The table below 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.

Table continues on the next page...

Table 3-3. Possible Error Codes in a NAK Reply Message (continued)

Value	Error Code Name	Description
`0x02`	`kDSReplyPacketSizeError`	The length of the received message was zero.
`0x05`	`kDSReplyBadFCS`	The contents of the CodeWarrior TRK data frame did not match the FCS checksum value. For more information, see Checksum Values.
`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 CodeWarrior TRK.
`0x01`	`kDSReplyError`	Unknown problem in transmission.
`0x07`	`kDSReplySequenceMissing`	Gap in reply sequence.
`0x17`	`kDSReplyBreakpointsFull`	The breakpoint resources (hardware or software) are exhausted.
`0x18`	`kDSReplyBreakpointConflict`	The requested breakpoint conflicts with an existing breakpoint.
`0x20`	`kDSReplyOsError`	General OS error.
`0x21`	`kDSReplyInvalidProcessId`	The request specified an invalid process.
`0x22`	`kDSReplyInvalidThreadId`	The request specified an invalid thread.

Chapter 4
Customizing CodeWarrior TRK

This chapter shows how to customize CodeWarrior TRK to work with new target boards.

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

Table 4-1. Required CodeWarrior TRK customizations

Customization Section	Customization Required?
Customizing CodeWarrior TRK Initializations	Yes
Modifying Serial Communication Functions	Yes
Modifying Existing UART Drivers	Yes
Changing the Data Transmission Rate	Yes
Customizing CodeWarrior TRK to be Interrupt-Driven	No
Customizing the CPU Speed	Yes
Changing ReadMemory-Related Code	Yes
Changing WriteMemory-Related Code	Yes
Changing SupportMask-Related Code	No
Changing Versions-Related Code	No
Changing the Maximum Message Length	No
Customizing Memory Locations	No
Customizing Exception Handling	No
Customizing Checksum Values	No
Customizing the Target Board Name	No
Customizing usr_put_config.h for Debugging	No

NOTE

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

4.1 Customizing CodeWarrior TRK Initializations

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

1. Examine the existing initialization sequence for your default implementation of CodeWarrior TRK.
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 Customizing CodeWarrior TRK.

For more information, see CodeWarrior TRK Initializations.

4.2 Customizing Serial Communications

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

This section contains these topics:

- Modifying Serial Communication Functions
- Modifying Existing UART Drivers
- Changing Data Transmission Rate
- Customizing CodeWarrior TRK to be Interrupt-Driven

4.2.1 Modifying Serial Communication Functions

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

CodeWarrior TRK provides configurable driver code for TI TL16C552a (works with most 16552-compatible UARTs).

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

NOTE

This is a required customization. For more information, see Customizing CodeWarrior TRK.

If your UART is not compatible with the supplied driver code, you must implement your own driver. If your UART is compatible with one of the preceding drivers, see Modifying Existing UART Drivers.

If you are using the UART library only to support CodeWarrior TRK, you must change these UART functions for new target boards:

- InitializeUART()
- ReadUARTPoll()
- WriteUART1()

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

- ReadUART1()
- TerminateUART()

This file prototypes the UART functions in this section:

```
CWTRKDir/Export/UART.h
```

NOTE

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

4.2.2 Modifying Existing UART Drivers

CodeWarrior TRK provides configurable driver code for TI TL16C552a (works with most 16552-compatible UARTs).

NOTE

This is a required customization. For more information, see Customizing CodeWarrior TRK.

4.2.2.1 Building TI TL16C552a UART Driver

This driver is located in this directory:

CWTRKDir/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 CodeWarrior TRK project.)

   Include these driver files:

   - `tl16c552a.c` (main driver code)
   - One of these 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 `tl16c552a_config_sample.h` to your local project directory.
3. Change the name of the driver configuration file `tl16c552a_config_sample.h` to:

   `tl16c552a_config.h`
4. Include the renamed driver configuration file `tl16c552a_config.h` in your library project.
5. 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 this UART header file is in your include path: *CWTRKDir*/Export/ UART.h
7. Build your library project.

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

   NOTE

   For information about building projects, see the *CodeWarrior IDE User's Guide*.

4.2.3 Changing Data Transmission Rate

CodeWarrior TRK 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 CodeWarrior TRK.

NOTE

This is a required customization. For more information, see Customizing CodeWarrior TRK.

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 CodeWarrior TRK, 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 must also set the debugger to communicate at the same data transmission rate as CodeWarrior TRK. 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 CodeWarrior TRK to 38.4 KB, even if the target board accepts a faster rate.

4.2.4 Customizing CodeWarrior TRK to be Interrupt-Driven

Depending on the target board, CodeWarrior TRK 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 CodeWarrior TRK currently supports that communication method.

When using serial polling, CodeWarrior TRK does not respond to messages from the debugger while the target application is running. However, when using interrupt-driven communication, CodeWarrior TRK 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, CodeWarrior TRK calls `TransportIrqHandler ()` when a UART interrupt occurs and `Read UARTPoll ()` 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, CodeWarrior TRK returns an error, and the debugger must stop the target application before resending the request.

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

For any target board for which CodeWarrior TRK 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 CodeWarrior TRK 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 these functions for your target board:

- InitializeIntDrivenUART()
- TransportIrqHandler()

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.)

4.3 Customizing 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.

NOTE
This is a required customization. For more information, see Customizing CodeWarrior TRK.

4.4 Customizing Debug Services

CodeWarrior TRK provides debug services using a debug message interface that consists of debug requests and debug notifications. (For more information, see Debug Message Interface Reference.)

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

This section describes 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 Maximum Message Length

NOTE
If you are customizing CodeWarrior TRK, ensure that you implement all messages in the primary command set. For more information, see Command Sets.

4.4.1 Changing ReadMemory-Related Code

After receiving a ReadMemory request, CodeWarrior TRK reads the specified section of memory and returns the result. To perform this task, CodeWarrior TRK 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 Customizing CodeWarrior TRK.

For more information, see ReadMemory, TargetAccessMemory(), and ValidMemory32().

4.4.2 Changing WriteMemory-Related Code

After receiving a WriteMemory request, CodeWarrior TRK writes the specified data in memory at the specified address. To perform this task, CodeWarrior TRK 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 Customizing CodeWarrior TRK.

For more information, see WriteMemory, TargetAccessMemory(), and ValidMemory32().

4.4.3 Changing SupportMask-Related Code

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

CWTRKDir/ `Portable/ default_smask.h`

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

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 CodeWarrior TRK 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_smask.h` to `target.h` and change the definitions as needed.

NOTE

You can also 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, DoSupportMask(), and TargetSupportMask().

4.4.4 Changing Versions-Related Code

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

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

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

These 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.

These 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, DoVersions(), and TargetVersions().

4.4.5 Changing 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 Customizing CodeWarrior TRK.

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

NOTE

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

4.5 Customizing Memory Locations

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

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

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

- Modifying variables in the Linker target settings panel in your CodeWarrior TRK project
- Modifying the linker command file in your CodeWarrior TRK 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.

4.6 Customizing Exception Handling

You can customize exception handling by overriding the default exception-handling code so that your application handles some exceptions. CodeWarrior TRK must handle certain exceptions; in those cases, your application must accommodate CodeWarrior TRK 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 Customizing CodeWarrior TRK.

4.7 Customizing Checksum Values

By default, CodeWarrior TRK uses a one-byte checksum value for error-checking when it frames messages. (For more information, see Checksum Values.)

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

However, you can specify that the CodeWarrior TRK 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 this file and recompile the CodeWarrior TRK:

CWTRKDir`/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.)

4.8 Customizing Target Board Name

The name of the target board displays in the startup welcome message for CodeWarrior TRK. 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.

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

4.9 Customizing usr_put_config.h for Debugging

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

NOTE

This customization is not required. For more information, see Customizing CodeWarrior TRK.

You can customize these values:

- `DEBUGIO_SERIAL`

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

NOTE

Usually, you define only `DEBUGIO_SERIAL` when debugging CodeWarrior TRK.

- `DEBUGIO_RAM`

  When you define `DEBUGIO_RAM`, CodeWarrior TRK writes the debug information to a RAM buffer on the target board. If you define `DEBUGIO_RAM`, you also must define these 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 CodeWarrior TRK because the debugging output displays in the console window of the debugger.

NOTE

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

Chapter 5
Debug Message Interface Reference

This chapter describes the debug message interface, that is, the set of debug messages that the debugger and CodeWarrior TRK use to communicate.

This chapter contains these topics:

- Command Sets
- Messages Sent by Debugger
- Messages Sent by CodeWarrior TRK

5.1 Command Sets

Each message described in this chapter 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 CodeWarrior TRK, ensure that you implement all messages in the primary command set.

5.2 Messages Sent by Debugger

This section describes the messages that the debugger can send to CodeWarrior TRK, which are all requests.

NOTE

See this 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. CodeWarrior TRK and the debugger place return values for a request in a separate reply message. For more information, see Reply Messages.

The debugger can send these messages:

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

5.2.1 Connect

Requests that CodeWarrior TRK begin a debug session.

Command Set

Primary command set (level 1).

Fields

This message contains this 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()

5.2.2 Continue

Requests that CodeWarrior TRK start running the target application.

Command Set

Primary command set (level 1).

Fields

This message contains this field:

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

Return Values

None.

Error Codes

CodeWarrior TRK can return this 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 CodeWarrior TRK to resume executing the target application. After receiving a Continue request, CodeWarrior TRK 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 CodeWarrior TRK Execution States.

Handler Function

`DoContinue()`

See Also

DoContinue()

5.2.3 CPUType

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

Command Set

Extended command set (level 2).

Fields

This message contains this field:

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

Return Values

This message causes CodeWarrior TRK to return these values:

Return Value Field	Size	Description
`cpuMajor`	`ui8`	The major CPU type, which indicates the processor family of the target board.

Table continues on the next page...

Return Value Field	Size	Description
`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.
`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

CodeWarrior TRK can return these 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()

5.2.4 FlushCache

Requests that CodeWarrior TRK 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 CodeWarrior TRK.)

Command Set

Secondary command set (level 2).

Fields

This message contains these fields:

Field	Size	Description
`command`	`ui8`	`kDSFlushCache` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain these values, which specify the type of cache to flush: `DS_MSG_CACHE_TYPE_INSTRUCTIONDS_MSG_CACHE_TYPE_DATADS_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

CodeWarrior TRK can return these 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
- `msgcmd.h`

5.2.5 ReadMemory

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

Command Set

Primary command set (level 1).

Fields

This message contains these fields:

Field	Size	Description
`command`	`ui8`	`kDSReadMemory` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of these 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 CodeWarrior TRK to return these 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

CodeWarrior TRK can return these error codes:

Error Code	Description
kDSReplyCWDSError	An unknown error occurred while processing the request.

Table continues on the next page...

Error Code	Description
`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

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

Handler Function

`DoReadMemory()`

See Also

- DoReadMemory()
- `msgcmd.h`

5.2.6 ReadRegisters

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

Command Set

Primary command set (level 1).

Fields

This message contains these fields:

Field	Size	Description
`command`	`ui8`	`kDSReadRegisters` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of these values: • `kDSRegistersDefault`

Table continues on the next page...

Field	Size	Description
		• `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 CodeWarrior TRK to return this 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

CodeWarrior TRK can return these 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, CodeWarrior TRK reads the specified sequence of registers from the processor, returning the resulting values to the debugger. CodeWarrior TRK attempts to catch and handle any access exceptions that occur while reading.

NOTE

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

Handler Function

`DoReadRegisters()`

See Also

- DoReadRegisters()
- `msgcmd.h`

5.2.7 Reset

Requests that CodeWarrior TRK reset the target board.

Command Set

Extended command set (level 2).

Fields

This message contains this 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, CodeWarrior TRK calls its own reset code. CodeWarrior TRK restarts and performs all hardware initializations as if the board were being manually reset.

Handler Function

`DoReset()`

See Also

DoReset()

5.2.8 Step

Requests that CodeWarrior TRK 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 these fields:

Field	Size	Description
`command`	`ui8`	`kDSStep` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of these 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 these fields:

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

Table continues on the next page...

Field	Size	Description
`rangeEnd`	`ui32`	The end address of the specified memory range.

Return Values

None.

Error Codes

CodeWarrior TRK can return these 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.
`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.
`kDSReplyUnsupportedOptionError`	The specified value of the `options` field is unsupported.

Remarks

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

This message specifies

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

If the value of the `options` parameter is `kDSStepIntoCount` or `kDSStepOverCount`, CodeWarrior TRK 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`, CodeWarrior TRK continues running the program until it encounters an instruction whose address is outside the range specified by `rangeStart` and `rangeEnd`. CodeWarrior TRK then returns control to the host.

CodeWarrior TRK notifies the debugger that the end condition was reached by sending a NotifyStopped notification. For more information, see NotifyStopped.

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

This 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 CodeWarrior TRK 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 CodeWarrior TRK 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`, CodeWarrior TRK 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()
- NotifyStopped
- `msgcmd.h`

5.2.9 Stop

Requests that CodeWarrior TRK stop running the target application.

Command Set

Extended command set (level 2).

Fields

This message contains this field:

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

Return Values

None.

Error Codes

CodeWarrior TRK can return this error code:

Error Code	Description
`kDSReplyError`	Unknown problem in transmission.

Remarks

None.

Handler Function

`DoStop()`

See Also

DoStop()

5.2.10 SupportMask

Requests that CodeWarrior TRK return a list of supported messages.

Command Set

Primary command set (level 1).

Fields

This message contains this field:

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

Return Values

This message causes CodeWarrior TRK to return these 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 CodeWarrior TRK. For more information, see Command Sets.

Error Codes

CodeWarrior TRK can return these 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 about how the default values are set, see `target_supp_mask.h` and Changing SupportMask-Related Code.

Handler Function

```
DoSupportMask()
```

See Also

- DoSupportMask()
- `msgcmd.h`

5.2.11 Versions

Requests that CodeWarrior TRK return version information.

Command Set

Primary command set (level 1).

Fields

This message contains this field:

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

Return Values

This message causes CodeWarrior TRK to return these values:

Return Value Field	Size	Description
`kernelMajor`	`ui8`	The major version number for CodeWarrior TRK. (In version 1.2, the `kernelMajor` is 1.)
`kernelMinor`	`ui8`	The minor version number for CodeWarrior TRK. (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

CodeWarrior TRK can return these 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()

5.2.12 WriteMemory

Requests that CodeWarrior TRK write data to a specified memory location.

Command Set

Primary command set (level 1).

Fields

This message contains these fields:

Field	Size	Description
`command`	`ui8`	`kDSWriteMemory` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of these 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 CodeWarrior TRK to return this value:

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

Error Codes

CodeWarrior TRK can return these 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.

Table continues on the next page...

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.
`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

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

Handler Function

`DoWriteMemory()`

See Also

- DoWriteMemory()
- `msgcmd.h`

5.2.13 WriteRegisters

Requests that CodeWarrior TRK write data to a specified sequence of registers.

Command Set

Primary command set (level 1).

Fields

This message contains these fields:

Field	Size	Description
`command`	`ui8`	`kDSWriteRegisters` (defined in the `msgcmd.h` file).
`options`	`ui8`	This field can contain one of these values: • `kDSRegistersDefault` • `kDSRegistersFP` • `kDSRegistersExtended1` • `kDSRegistersExtended2`

Table continues on the next page...

Field	Size	Description
		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

CodeWarrior TRK can return these 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, CodeWarrior TRK writes the specified data into the specified register sequence. CodeWarrior TRK attempts to catch and handle any access exceptions that occur while writing.

NOTE

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

Handler Function

```
DoWriteRegisters()
```

See Also

- DoWriteRegisters()
- `msgcmd.h`

5.3 Messages Sent by CodeWarrior TRK

This section describes the messages that CodeWarrior TRK can send to the debugger. Some messages are notifications; others are requests.

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.

CodeWarrior TRK can send these messages:

- NotifyException
- NotifyStopped
- ReadFile
- WriteFile

5.3.1 NotifyException

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

Command Set

Primary command set (level 1).

Fields

This message contains these 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.

Field	Size	Description
		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().

Return Values

None.

Remarks

None.

See Also

- DoNotifyStopped
- TargetAddExceptionInfo()
- TargetAddStopInfo()
- TargetInterrupt()

5.3.2 NotifyStopped

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

Command Set

Primary command set (level 1).

Fields

This message contains these 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().

Return Values

None.

Remarks

None.

See Also

- TargetInterrupt()
- TargetAddStopInfo()

5.3.3 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 these 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 these 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, CodeWarrior TRK passes the data to the target application.

See Also

- `mslsupp.c`
- `targsupp.h`
- SuppAccessFile()

5.3.4 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 these 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 these 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()

Chapter 6 CodeWarrior TRK Function Reference

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

This chapter describes these functions:

_reset()	DoConnect()
DoContinue()	DoCPUType()
DoFlushCache	DoNotifyStopped()
DoReadMemory()	DoReadRegisters()
DoReset()	DoStep()
DoStop()	DoSupportMask()
DoVersions()	DoWriteMemory()
DoWriteRegisters()	InitializeIntDrivenUART()
InitializeUART()	InterruptHandler()
ReadUARTPoll()	ReadUART1()
ReadUARTN()	ReadUARTString()
SuppAccessFile()	SwapAndGo()
TargetAccessMemory()	TargetAddExceptionInfo()
TargetAddStopInfo()	TargetContinue()
TargetFlushCache()	TargetInterrupt()
TargetAccessDefault()	TargetAccessExtended1()
TargetAccessExtended2()	TargetAccessFP()
TargetSingleStep()	TargetStepOutOfRange()
TargetSupportMask()	TargetVersions()
TerminateUART()	TransportIrqHandler()
ValidMemory32()	WriteUART1()
WriteUARTN()	WriteUARTString()

6.1 _reset()

Resets the board and initializes CodeWarrior TRK.

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 CodeWarrior TRK for a new target board. For more information, see CodeWarrior TRK Initializations and Customizing CodeWarrior TRK Initializations.

Source File

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

Board-specific?

Yes.

6.2 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.

Returns

Returns a `DSError` error code.

Remarks

Sends an acknowledgment to the debugger.

Source File

`msghndlr.c`

Board-specific?

No.

See Also

Connect

6.3 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.

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()
- Continue

6.4 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 CPUType.

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.

Source File

`msghndlr.c`

Board-Specific?

No.

See Also

`msgcmd.h`

6.5 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 about the arguments contained in this message, see FlushCache.

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()
- FlushCache
- `msgcmd.h`

6.6 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 these:

- `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()
- TargetAddStopInfo()
- NotifyStopped
- `msgcmd.h`

6.7 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 about the arguments contained in this message, see ReadMemory.

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()
- ReadMemory

6.8 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 about the arguments contained in this message, see ReadRegisters.

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 one of these functions:

- `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 about register definitions, see the processor-specific appendixes in this manual.

Source File

`msghndlr.c`

Board-specific?

No.

See Also

- TargetAccessDefault()
- TargetAccessFP()
- TargetAccessExtended1()
- TargetAccessExtended2()
- ReadRegisters
- `msgcmd.h`

6.9 DoReset()

The `DoReset()` function re-initializes CodeWarrior TRK 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.

Returns

None.

Remarks

Calls the _reset code segment, which is the starting point for CodeWarrior TRK 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()
- Reset

6.10 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 about the arguments contained in this message, see Step.

Returns

Returns a `DSError` error code.

Remarks

This 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()

- TargetStepOutOfRange()
- Step

6.11 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.

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 CodeWarrior TRK Execution States.

Source File

`msghndlr.c`

Board-specific?

No.

See Also

Stop

6.12 DoSupportMask()

The `DoSupportMask()` function sends a vector that indicates which messages CodeWarrior TRK 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.

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 CodeWarrior TRK supports. Then `DoSupportMask()` places the vector in a reply message that CodeWarrior TRK 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()
- SupportMask

6.13 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.

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

NOTE

If you change CodeWarrior TRK or its protocols, you can update the kernel and protocol version numbers, respectively. For more information, see Changing Versions-Related Code.

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

Source File

`msghndlr.c`

Board-specific?

No.

See Also

- TargetVersions()
- Versions

6.14 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 about the arguments contained in this message, see WriteMemory.

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()
- WriteMemory

6.15 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 about the arguments contained in this message, see WriteRegisters.

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 this list:

- `kDSRegistersDefault`

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

- `kDSRegistersFP`

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

- `kDSRegistersExtended1`

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

- `kDSRegisters`Extended2

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

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

NOTE

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

Source File

`msghndlr.c`

Board-specific?

No.

See Also

- TargetAccessDefault()
- TargetAccessExtended1()
- TargetAccessExtended2()
- TargetAccessFP()
- WriteRegisters
- `msgcmd.h`

6.16 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

CodeWarrior TRK 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()

- SwapAndGo()
- TransportIrqHandler()

6.17 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 CodeWarrior TRK 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.

6.18 InterruptHandler()

Handles an interrupt received by CodeWarrior TRK.

```
  void InterruptHandler();
```

Returns

None.

Remarks

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

If the value of `TRK_TRANSPORT_INT_DRIVEN` is 1 (indicating that this CodeWarrior TRK is interrupt-driven), `InterruptHandler()` first determines whether the interrupt is a communication interrupt. If it is, `InterruptHandler()` calls `TransportIrqHandler()` 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()
- SwapAndGo()
- TransportIrqHandler()

6.19 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 these `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.

6.20 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.

6.21 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()

6.22 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()

6.23 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`

6.24 SwapAndGo()

Saves the state (context) of CodeWarrior TRK, 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 CodeWarrior TRK 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()
- InterruptHandler()
- TransportIrqHandler()

6.25 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 and WriteMemory.

`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()
- DoWriteMemory()
- ValidMemory32()
- ReadMemory
- WriteMemory

6.26 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 about the arguments contained in this message, see NotifyException.

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 about the register definition files and exceptions, see the processor-specific appendixes in this manual.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoNotifyStopped()
- NotifyException

6.27 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 about the arguments contained in this message, see NotifyStopped.

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 CodeWarrior TRK source code to see the specific information returned for your target processor.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoNotifyStopped()
- NotifyStopped

6.28 TargetContinue()

The `TargetContinue()` function starts the target application running and then blocks until control returns to CodeWarrior TRK (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.

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 CodeWarrior TRK regains control (because of an unhandled exception or breakpoint), control returns to the CodeWarrior TRK core, which properly handles the exception.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoContinue()
- Continue
- SwapAndGo()

6.29 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 CodeWarrior TRK implementation to work with a currently unsupported processor. In this case, your new version of `TargetFlushCache()` must do one of these:

- 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
- FlushCache

6.30 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 CodeWarrior TRK is stepping through multiple lines and stepping is not complete.

Source File

`targimpl.c`

Board-specific?

No.

See Also

DoNotifyStopped()

6.31 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 about the arguments contained in this message, see ReadRegisters or WriteRegisters.

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 CodeWarrior TRK to read the specified registers if the variable is true. Otherwise, CodeWarrior TRK 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 about registers, see the processor-specific appendixes in this manual.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoReadRegisters()
- DoWriteRegisters()
- ReadRegisters
- WriteRegisters

6.32 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 about the arguments contained in this message, see ReadRegisters or WriteRegisters.

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 CodeWarrior TRK to read the specified registers if the variable is true. Otherwise, CodeWarrior TRK 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 about registers, see the processor-specific appendixes in this manual.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoReadRegisters()

- DoWriteRegisters()
- ReadRegisters
- WriteRegisters

6.33 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 about the arguments contained in this message, see ReadRegisters or WriteRegisters.

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 CodeWarrior TRK to read the specified registers if the variable is true. Otherwise, CodeWarrior TRK 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 about registers, see the processor-specific appendixes in this manual.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoReadRegisters()
- DoWriteRegisters()
- ReadRegisters
- WriteRegisters

6.34 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 about the arguments contained in this message, see ReadRegisters or WriteRegisters.

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 CodeWarrior TRK to read the specified registers if the variable is true. Otherwise, CodeWarrior TRK 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 about registers, see the processor-specific appendixes in this manual.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoReadRegisters()
- DoWriteRegisters()
- ReadRegisters
- WriteRegisters

6.35 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 CodeWarrior TRK 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()
- Step

6.36 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 CodeWarrior TRK 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()
- Step

6.37 TargetSupportMask()

Returns a mask that indicates which debug messages the current CodeWarrior TRK 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.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoSupportMask()
- SupportMask

6.38 TargetVersions()

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

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

`versions`

Output variable containing version information for the running CodeWarrior TRK 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 CodeWarrior TRK build. The protocol attribute is the version of the debug message interface and low-level serial protocols used by CodeWarrior TRK.

NOTE

If you change CodeWarrior TRK or its protocols, you can update the kernel and protocol version numbers, respectively. For more information, see Changing Versions-Related Code.

Source File

`targimpl.c`

Board-specific?

No.

See Also

- DoVersions()
- Versions
- `target.h`

6.39 TerminateUART()

Deactivate the serial device.

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

Returns

Returns a `UARTError` error code.

Remarks

The default implementation of CodeWarrior TRK does not call this function. However, you can run an operating system on the target board while you are debugging using CodeWarrior TRK. 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.

6.40 TransportIrqHandler()

Handles a UART interrupt.

NOTE

You must change `TransportIrqHandler()` for new target boards if you are using interrupt-driven communication with CodeWarrior TRK.

```
asm void TransportIrqHandler(void);
```

Returns

None.

Remarks

The `TransportIrqHandler()` function is part of the UART driver code. The CodeWarrior TRK `InterruptHandler()` function calls `TransportIrqHandler()`, which disables interrupts while it is running. For a serial input interrupt, `TransportIrqHandler()` 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 `TransportIrqHandler()`, do not assume that the registers hold any particular values. The `TransportIrqHandler()` 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()
- InterruptHandler()
- SwapAndGo()

6.41 ValidMemory32()

Verifies the range of addresses for the target board when CodeWarrior TRK 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 these 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`

6.42 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()
- WriteUARTString()

6.43 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()
- WriteUARTString()

6.44 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()
- WriteUARTN()

Chapter 7
AppTRK Reference

This chapter defines what AppTRK is and explains how to use it. This chapter includes these topics:

7.1 What is AppTRK?

AppTRK is a program that runs on development boards that boot embedded Linux. AppTRK is a Linux application that lets the CodeWarrior debugger control other Linux applications running on the board. Essentially, AppTRK is CodeWarrior TRK transformed into a Linux application-hence the name AppTRK.

7.2 Differences Between AppTRK and CodeWarrior TRK

The main difference between AppTRK and CodeWarrior TRK is that AppTRK is a Linux application. However, there are other important differences:

- In most situations, you do not need to change AppTRK source code.

AppTRK does not interact directly with the hardware. Instead, AppTRK uses the API of the embedded Linux operating system. As a result, once Freescale creates an embedded Linux board support package (BSP) for a given development board, AppTRK automatically works on this board.

- Each time you power-up your board, you must reconfigure the link between AppTRK and the CodeWarrior debugger.

  CodeWarrior TRK communicates with the debugger over a serial or Ethernet connection using an asynchronous protocol. Once configured, this setup requires no changes.

  AppTRK communicates with the debugger over an Ethernet connection using the TCP/IP protocol. Because the development board's Ethernet port "forgets" its IP address each time the board is powered off, you must reassign this address each time you power-up the board.

- Each time you reboot embedded Linux, you must launch AppTRK.

  Because AppTRK is just another Linux application, you must launch AppTRK each time you reboot the board, unless you configure the Linux to automatically start the AppTRK. To do so, configure the IP address and start the AppTRK on a default port.

NOTE

For detailed instructions on working with the AppTRK package, refer the SDK user manual. The document is available in the iso/help/documents/pdf folder of the SDK layout.

7.3 Using AppTRK

This section shows you how to set up an embedded Linux development environment, launch AppTRK, and create a CodeWarrior project such that the CodeWarrior debugger and AppTRK can communicate over a TCP/IP link.

To accomplish these tasks, follow these steps:

1. Build the AppTRK package.
2. Include the AppTRK package in the ramdisk

NOTE

For detailed instructions on how to install the SDK and work with the AppTRK package, refer the SDK user manual. The document is available in the iso/help/documents/pdf folder of the SDK layout.

3. Start the Linux BSP
   a. After the kernel boot is finished, start the apptrk agent `apptrk :12345 &`
   b. Test that the network configuration is working as expected:
      - Ping the Linux host machine IP from the simulated Linux, ping the simulated IP from the Linux machine.
      - From the Windows machine try `telnet <IP address> 1234` where `<IP address>` is the address of the Linux alias Ethernet device and `1234` is the port on which apptrk is running

NOTE

You will see some AppTRK information after issuing the telnet command. If no information shows up then there might be a problem with the whole network configuration.

4. Launch CodeWarrior.

   The WorkSpace Launcher dialog box appears, prompting you to select a workspace to use.

NOTE

Click Browse to change the default location for workspace folder. You can also select the Use this as the default and do not ask again checkbox to set default or selected path as the default location for storing all your projects.

5. Click OK.

   The default workspace is accepted. The CodeWarrior IDE launches and the Welcome page appears.

NOTE

The Welcome page appears only if the CodeWarrior IDE or the selected Workspace is opened for the first time. Otherwise, the Workbench window appears.

6. Click Go to Workbench, on the Welcome page.

   The workbench window appears.

7. Select File > New > Power Architecture Project, from the CodeWarrior IDE menu bar.

The New Power Architecture Project wizard launches and the Create a Power Architecture Project page appears.

8. Specify a name for the new project in the Project name text box.

   For example, enter the project name as linux_project.

9. If you do not want to create your project in the default workspace:
   a. Clear the Use default location checkbox.
   b. Click Browse and select the desired location from the Browse For Folder dialog box.
   c. In the Location text box, append the location with the name of the directory in which you want to create your project.

NOTE

An existing directory cannot be specified for the project location.

10. Click Next.

    The Processor page appears.

11. Select the target processor for the new project, from the Processor list.
12. Select a processing model option from the Processing Model group.

NOTE

SMP option is disabled and cannot be selected in the current installation.

- Select AMP (One project per core) to generate a separate project for each selected core. The option will also set the core index for each project based on the core selection.
- Select AMP (One build configuration per core) to generate one project with multiple targets, each containing an .lcf file for the specified core.

13. Select Application from the Project Output group, to create an application with .elf extension, that includes information required to debug the project.
14. Click Next.

    The Build Settings page appears.

15. Select a toolchain for Linux applications from the Toolchain group.

    Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.

NOTE

The current release does not include toolchains for Linux applications by default. To add the required Linux build tools support, you should install the corresponding service pack for the required target. For more information on installing service packs, refer to the Service Pack Updater Quickstart available in the <CodeWarrior-Install-Dir>\PA\ folder.

16. Click Next.

    The Linux Application Launch Configurations page appears.

17. Select New System to create a new Remote system configuration.
    - Select CodeWarrior TRK to use the CodeWarrior Target Resident Kernel (TRK) protocol, to download and control application on the Linux host system.

      NOTE

      When debugging a Linux application, you must use the CodeWarrior TRK to manage the communications interface between the debugger and Linux system.

    - In the IP Address textbox, enter the IP Address of the Linux host system, the project executes on.
    - In the Port textbox, enter the port number that the debugger will use to communicate to the Linux host system.
    - In the Remote Download Path textbox, enter the absolute path for the host directory, into which the debugger downloads the application.
18. Click Finish.

    The wizard creates a Linux application project according to your specifications. You can access the project from the CodeWarrior Projects view on the Workbench.

That's it. You have configured your development environment on a simulator (using Virtutech Simics) for TCP/IP communications, launched AppTRK such that it can communicate with the CodeWarrior debugger, and created a CodeWarrior project configured such that the CodeWarrior debugger can communicate with AppTRK.

NOTE

To start with, you can see the CodeWarrior Examples projects included in the CodeWarrior layout. These include a Fork, Thread, Shared Library, and some other projects.

7.4 Modifying AppTRK

A working AppTRK is provided with each Linux BSP you obtain from Freescale. On rare occasions, however, you may want to modify the AppTRK.

To modify AppTRK and include it in the Linux image running on your development board, follow these steps:

1. Modify and build the libelf and AppTRK projects.

   NOTE

   For details on modifying and building an AppTRK package, refer the SDK user manual. The document is available in the `iso/help/documents/pdf` folder of the SDK layout.

2. Copy the modified `AppTRK.elf` file to the simulated Linux filesystem.
3. Redeploy the kernel package for creating the new ramdisk with AppTRK included.
4. Boot the kernel with the new ramdisk and start the AppTRK agent.

That's it. You have modified AppTRK and included it in the Linux image running on your development board.

7.5 Creating AppTRK for a Non-Freescale Linux Image

NOTE

For details on creating an AppTRK package for a Non-Freescale Linux image, refer the SDK user manual. The document is available in the `iso/help/documents/pdf` folder of the SDK layout.

AppTRK is intended to be run from the command line on the target system. It accepts one parameter which specifies either a TCP port or a serial device to be used for communication with the host debugger. For example:

```
AppTRK.elf :1234
```

```
AppTRK.elf /dev/ttyS0
```

To set the baud rate for serial communication, use `stty`. Alternatively, AppTRK will use stdin/stdout pipes for debug communication if the pipe symbol, '`|`', is specified. Using this option, AppTRK can be run under `inetd`. For example:

```
AppTRK.elf |
```

7.6 Conditions under which AppTRK Must Be Modified

These are the conditions under which the AppTRK must be modified:

- The same gcc toolchain should be used when compiling both BSP and AppTRK.
- The user must re-build AppTRK in case a different gcc is used for building the BSP (for example the customer uses a non -FSL Linux BSP).

  In this case, the user must ensure that the drivers (ethernet/serial) are functional in his BSP. The AppTRK communication with the target relies on drivers provided by the BSP.

Index

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWPATRKUG
Revision 10.5.1, 01/2016