<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of this Manual</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Related Documentation</td>
<td>10</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Other CodeWarrior Documentation</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Other Documentation</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>CodeWarrior TRK Architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1</td>
<td>CodeWarrior TRK Core</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2</td>
<td>CodeWarrior TRK Execution States</td>
<td>14</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Message Queues</td>
<td>15</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Request and Notification Handling</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>CodeWarrior TRK Memory Layout</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1</td>
<td>CodeWarrior TRK RAM Sections</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Target Application RAM Sections</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>CodeWarrior TRK Initializations</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Transport Level</td>
<td>20</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Serial Communications Settings</td>
<td>20</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Data Transmission Rate</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Framing Level</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1</td>
<td>CodeWarrior TRK Data Frames</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Checksum Values</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2.1</td>
<td>Encoding Single-Byte Checksum Values</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2.2</td>
<td>Verifying Single-Byte Checksum Values</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2.3</td>
<td>Using Multi-Byte Checksum Values</td>
<td>26</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Escape Sequences</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Reliable Message Delivery</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4.1</td>
<td>Responding to a NAK Reply Message</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4.2</td>
<td>Responding When No Reply Message is Received</td>
<td>29</td>
</tr>
<tr>
<td>3.2.4.3</td>
<td>Preventing Transmission Failure</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Debug Message Interface Level</td>
<td>30</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Request and Notification Messages</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.1</td>
<td>Alignment</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.2</td>
<td>Byte Order</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1.3</td>
<td>Message Length</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Reply Messages</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2.1</td>
<td>ACK Messages</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2.2</td>
<td>NAK Messages</td>
<td>34</td>
</tr>
</tbody>
</table>

Chapter 4

Customizing CodeWarrior TRK

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Customizing CodeWarrior TRK Initializations</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Customizing Serial Communications</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Modifying Serial Communication Functions</td>
<td>38</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Modifying Existing UART Drivers</td>
<td>39</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>Building TI TL16C552a UART Driver</td>
<td>40</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Changing Data Transmission Rate</td>
<td>41</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Customizing CodeWarrior TRK to be Interrupt-Driven</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Customizing CPU Speed</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Customizing Debug Services</td>
<td>43</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Changing ReadMemory-Related Code</td>
<td>43</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Changing WriteMemory-Related Code</td>
<td>44</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Changing SupportMask-Related Code</td>
<td>44</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Changing Versions-Related Code</td>
<td>45</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Changing Maximum Message Length</td>
<td>46</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.5</td>
<td>Customizing Memory Locations</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Customizing Exception Handling</td>
<td>47</td>
</tr>
<tr>
<td>4.7</td>
<td>Customizing Checksum Values</td>
<td>47</td>
</tr>
<tr>
<td>4.8</td>
<td>Customizing Target Board Name</td>
<td>48</td>
</tr>
<tr>
<td>4.9</td>
<td>Customizing <code>usr_put_config.h</code> for Debugging</td>
<td>48</td>
</tr>
</tbody>
</table>

### Chapter 5

**Debug Message Interface Reference**

5.1 Command Sets.......................................................................................................................... 51

5.2 Messages Sent by Debugger...................................................................................................... 51

5.2.1 Connect................................................................................................................................... 52

5.2.2 Continue............................................................................................................................... 53

5.2.3 CPUType............................................................................................................................... 54

5.2.4 FlushCache............................................................................................................................ 55

5.2.5 ReadMemory.......................................................................................................................... 57

5.2.6 ReadRegisters....................................................................................................................... 58

5.2.7 Reset...................................................................................................................................... 60

5.2.8 Step........................................................................................................................................ 61

5.2.9 Stop........................................................................................................................................ 63

5.2.10 SupportMask......................................................................................................................... 64

5.2.11 Versions.............................................................................................................................. 65

5.2.12 WriteMemory......................................................................................................................... 66

5.2.13 WriteRegisters..................................................................................................................... 68

5.3 Messages Sent by CodeWarrior TRK............................................................................................ 70

5.3.1 NotifyException...................................................................................................................... 70

5.3.2 NotifyStopped....................................................................................................................... 71

5.3.3 ReadFile............................................................................................................................... 72

5.3.4 WriteFile.............................................................................................................................. 73

### Chapter 6

**CodeWarrior TRK Function Reference**
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>_reset()...........................................</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>DoConnect()........................................</td>
<td>76</td>
</tr>
<tr>
<td>6.3</td>
<td>DoContinue().......................................</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>DoCPUType().........................................</td>
<td>77</td>
</tr>
<tr>
<td>6.5</td>
<td>DoFlushCache......................................</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>DoNotifyStopped()...............................</td>
<td>79</td>
</tr>
<tr>
<td>6.7</td>
<td>DoReadMemory()....................................</td>
<td>80</td>
</tr>
<tr>
<td>6.8</td>
<td>DoReadRegisters()...............................</td>
<td>81</td>
</tr>
<tr>
<td>6.9</td>
<td>DoReset()..........................................</td>
<td>82</td>
</tr>
<tr>
<td>6.10</td>
<td>DoStep()...........................................</td>
<td>83</td>
</tr>
<tr>
<td>6.11</td>
<td>DoStop()...........................................</td>
<td>84</td>
</tr>
<tr>
<td>6.12</td>
<td>DoSupportMask()...................................</td>
<td>84</td>
</tr>
<tr>
<td>6.13</td>
<td>DoVersions()......................................</td>
<td>85</td>
</tr>
<tr>
<td>6.14</td>
<td>DoWriteMemory()...................................</td>
<td>86</td>
</tr>
<tr>
<td>6.15</td>
<td>DoWriteRegisters()..............................</td>
<td>87</td>
</tr>
<tr>
<td>6.16</td>
<td>InitializeIntDrivenUART().....................</td>
<td>88</td>
</tr>
<tr>
<td>6.17</td>
<td>InitializeUART()..................................</td>
<td>90</td>
</tr>
<tr>
<td>6.18</td>
<td>InterruptHandler().............................</td>
<td>90</td>
</tr>
<tr>
<td>6.19</td>
<td>ReadUARTPoll()....................................</td>
<td>91</td>
</tr>
<tr>
<td>6.20</td>
<td>ReadUART1().......................................</td>
<td>92</td>
</tr>
<tr>
<td>6.21</td>
<td>ReadUARTN().......................................</td>
<td>92</td>
</tr>
<tr>
<td>6.22</td>
<td>ReadUARTString()..................................</td>
<td>93</td>
</tr>
<tr>
<td>6.23</td>
<td>SuppAccessFile()..................................</td>
<td>94</td>
</tr>
<tr>
<td>6.24</td>
<td>SwapAndGo()......................................</td>
<td>95</td>
</tr>
<tr>
<td>6.25</td>
<td>TargetAccessMemory()............................</td>
<td>96</td>
</tr>
<tr>
<td>6.26</td>
<td>TargetAddExceptionInfo()......................</td>
<td>98</td>
</tr>
<tr>
<td>6.27</td>
<td>TargetAddStopInfo().............................</td>
<td>98</td>
</tr>
<tr>
<td>6.28</td>
<td>TargetContinue()..................................</td>
<td>99</td>
</tr>
<tr>
<td>6.29</td>
<td>TargetFlushCache()..............................</td>
<td>100</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.30</td>
<td>TargetInterrupt()</td>
<td>101</td>
</tr>
<tr>
<td>6.31</td>
<td>TargetAccessDefault()</td>
<td>102</td>
</tr>
<tr>
<td>6.32</td>
<td>TargetAccessExtended1()</td>
<td>103</td>
</tr>
<tr>
<td>6.33</td>
<td>TargetAccessExtended2()</td>
<td>105</td>
</tr>
<tr>
<td>6.34</td>
<td>TargetAccessFP()</td>
<td>106</td>
</tr>
<tr>
<td>6.35</td>
<td>TargetSingleStep()</td>
<td>107</td>
</tr>
<tr>
<td>6.36</td>
<td>TargetStepOutOfRange()</td>
<td>108</td>
</tr>
<tr>
<td>6.37</td>
<td>TargetSupportMask()</td>
<td>109</td>
</tr>
<tr>
<td>6.38</td>
<td>TargetVersions()</td>
<td>110</td>
</tr>
<tr>
<td>6.39</td>
<td>TerminateUART()</td>
<td>111</td>
</tr>
<tr>
<td>6.40</td>
<td>TransportIrqHandler()</td>
<td>112</td>
</tr>
<tr>
<td>6.41</td>
<td>ValidMemory32()</td>
<td>112</td>
</tr>
<tr>
<td>6.42</td>
<td>WriteUART1()</td>
<td>113</td>
</tr>
<tr>
<td>6.43</td>
<td>WriteUARTN()</td>
<td>114</td>
</tr>
<tr>
<td>6.44</td>
<td>WriteUARTString()</td>
<td>115</td>
</tr>
</tbody>
</table>

Chapter 7
AppTRK Reference

7.1 What is AppTRK?                                      | 117  |
7.2 Differences Between AppTRK and CodeWarrior TRK     | 117  |
7.3 Using AppTRK                                       | 118  |
7.4 Modifying AppTRK                                   | 121  |
7.5 Creating AppTRK for a Non-Freescale Linux Image    | 122  |
7.6 Conditions under which AppTRK Must Be Modified      | 123  |
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>
<thead>
<tr>
<th>Chapter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>(this chapter)</td>
</tr>
<tr>
<td>CodeWarrior TRK Concepts</td>
<td>Provides an overview of CodeWarrior TRK, describes the various tasks CodeWarrior TRK performs, and how these tasks are implemented.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 1-1. Contents of this Manual (continued)

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

**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.
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.
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:

\[\text{CWTRKDir}\backslash\text{Processor}\backslash\text{ProcessorType}\backslash\text{Board}\]

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>
<thead>
<tr>
<th>RAM Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>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.</td>
</tr>
<tr>
<td>Exception vectors</td>
<td>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.</td>
</tr>
<tr>
<td>The stack</td>
<td>CodeWarrior TRK requires at least 8KB of RAM for its stack, which is also the maximum amount of RAM that the stack occupies.</td>
</tr>
</tbody>
</table>

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

• _start

The _start assembly language code starts the runtime code.

If needed, you can place code for other initializations in _reset() immediately before the jump to _start. For example, sometimes (depending on the reference board for which your default implementation of 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

```c
_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.
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 physical 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.

![Figure 3-2. Creating a CodeWarrior TRK Data Frame](image-url)
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**

```c
#define    FCSBITSIZE FCS8
#if FCSBITSIZE == FCS8
    /* Definitions for 8-bit simple Frame Check Sequences */
    typedef unsigned char FCSType; /* 8-bit type */
    #define PPPINITFC88 0x00    /* Initial FCS value */
    #define PPPGOODFC88 0xFF    /* Good final FCS value */
    #define PPPINITFC8 PPPINITFC88
    #define PPPGOODFC8 PPPGOODFC88
    #define PPPCOMPFCS 0xFF    /* complement FCS value */
#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**

```c
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:

\[
\text{originalChar} = \text{escapedChar} ^ {0x20};
\]

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/end-frame 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>
<thead>
<tr>
<th>Type</th>
<th>Hex Value</th>
<th>Big-Endian Byte Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui8</td>
<td>0x12</td>
<td>[0x12]</td>
</tr>
<tr>
<td>ui16</td>
<td>0x1234</td>
<td>[0x12][0x34]</td>
</tr>
<tr>
<td>ui32</td>
<td>0x12345678</td>
<td>[0x12][0x34][0x56][0x78]</td>
</tr>
<tr>
<td>ui8[]</td>
<td>{0x12,0x34}</td>
<td>[0x12][0x34]</td>
</tr>
<tr>
<td>ui16[]</td>
<td>{0x1234,0x5678}</td>
<td>[0x12][0x34][0x56][0x78]</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### 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.

---

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Hex Value</th>
<th>Big-Endian Byte Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>u132[]</td>
<td><code>{0x12345678,0x9ABCDEF0}</code></td>
<td><code>[0x12][0x34][0x56][0x78][0x9A][0xBC][0xDE][0xF0]</code></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Error Code Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>kDSReplyNoError</td>
<td>The request was handled successfully.</td>
</tr>
<tr>
<td>0x10</td>
<td>kDSReplyUnsupportedCommandError</td>
<td>The request was of an invalid type. You can query CodeWarrior TRK to determine which requests are supported. For more information, see SupportMask.</td>
</tr>
</tbody>
</table>
| 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 3-2. Possible Error Codes in an ACK Reply Message (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Error Code Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>kDSReplyCWDSerror</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Error Code Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>kDSReplyEscapeError</td>
<td>An escape character was immediately followed by a start-frame/end-frame flag. For more information, see Escape Sequences.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
<table>
<thead>
<tr>
<th>Value</th>
<th>Error Code Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02</td>
<td>kDSReplyPacketSizeError</td>
<td>The length of the received message was zero.</td>
</tr>
<tr>
<td>0x05</td>
<td>kDSReplyBadFCS</td>
<td>The contents of the CodeWarrior TRK data frame did not match the FCS checksum value. For more information, see Checksum Values.</td>
</tr>
<tr>
<td>0x06</td>
<td>kDSReplyOverflow</td>
<td>The message exceeded the maximum length of the buffer. By default, the maximum length of a debug message is 2176 bytes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can change the maximum length by changing the value of the variable kMessageBufferSize in the file msgbuf.h and recompiling CodeWarrior TRK.</td>
</tr>
<tr>
<td>0x01</td>
<td>kDSReplyError</td>
<td>Unknown problem in transmission.</td>
</tr>
<tr>
<td>0x07</td>
<td>kDSReplySequenceMissing</td>
<td>Gap in reply sequence.</td>
</tr>
<tr>
<td>0x17</td>
<td>kDSReplyBreakpointsFull</td>
<td>The breakpoint resources (hardware or software) are exhausted.</td>
</tr>
<tr>
<td>0x18</td>
<td>kDSReplyBreakpointConflict</td>
<td>The requested breakpoint conflicts with an existing breakpoint.</td>
</tr>
<tr>
<td>0x20</td>
<td>kDSReplyOsError</td>
<td>General OS error.</td>
</tr>
<tr>
<td>0x21</td>
<td>kDSReplyInvalidProcessId</td>
<td>The request specified an invalid process.</td>
</tr>
<tr>
<td>0x22</td>
<td>kDSReplyInvalidThreadId</td>
<td>The request specified an invalid thread.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Customization Section</th>
<th>Customization Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customizing CodeWarrior TRK Initializations</td>
<td>Yes</td>
</tr>
<tr>
<td>Modifying Serial Communication Functions</td>
<td>Yes</td>
</tr>
<tr>
<td>Modifying Existing UART Drivers</td>
<td>Yes</td>
</tr>
<tr>
<td>Changing the Data Transmission Rate</td>
<td>Yes</td>
</tr>
<tr>
<td>Customizing CodeWarrior TRK to be Interrupt-Driven</td>
<td>No</td>
</tr>
<tr>
<td>Customizing the CPU Speed</td>
<td>Yes</td>
</tr>
<tr>
<td>Changing ReadMemory-Related Code</td>
<td>Yes</td>
</tr>
<tr>
<td>Changing WriteMemory-Related Code</td>
<td>Yes</td>
</tr>
<tr>
<td>Changing SupportMask-Related Code</td>
<td>No</td>
</tr>
<tr>
<td>Changing Versions-Related Code</td>
<td>No</td>
</tr>
<tr>
<td>Changing the Maximum Message Length</td>
<td>No</td>
</tr>
<tr>
<td>Customizing Memory Locations</td>
<td>No</td>
</tr>
<tr>
<td>Customizing Exception Handling</td>
<td>No</td>
</tr>
<tr>
<td>Customizing Checksum Values</td>
<td>No</td>
</tr>
<tr>
<td>Customizing the Target Board Name</td>
<td>No</td>
</tr>
<tr>
<td>Customizing usr_put_config.h for Debugging</td>
<td>No</td>
</tr>
</tbody>
</table>

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:
Customizing Memory Locations

- 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.
Customizing usr_put_config.h for Debugging
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
- CPUPType
- 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSConnect (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSContinue (defined in the msgcmd.h file)</td>
</tr>
</tbody>
</table>

Return Values
None.

Error Codes
CodeWarrior TRK can return this error code:
## Messages Sent by Debugger

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyNotStopped</td>
<td>The target application is running and must be stopped before the debugger issues this request.</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSCPUType (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>

#### Return Values

This message causes CodeWarrior TRK to return these values:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuMajor</td>
<td>ui8</td>
<td>The major CPU type, which indicates the processor family of the target board.</td>
</tr>
</tbody>
</table>
### Error Codes

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
<tr>
<td>kDSReplyCWDSError</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
</tbody>
</table>

### Handler Function

DoCPUPerType()

### See Also

DoCPUPerType()

### 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSFlushCache (defined in the msgcmd.h file).</td>
</tr>
<tr>
<td>options</td>
<td>ui8</td>
<td>This field can contain these values, which specify the type of cache to flush: DS_MSG_CACHE_TYPE_INSTRUCTION D_MSG_CACHE_TYPE_DATA D_MSG_CACHE_TYPE_SECONDARY For more information, see the msgcmd.h file.</td>
</tr>
<tr>
<td>start</td>
<td>ui32</td>
<td>The starting address of the specified memory section in the cache.</td>
</tr>
<tr>
<td>end</td>
<td>ui32</td>
<td>The end address of the specified memory section in the cache.</td>
</tr>
</tbody>
</table>

Error Codes

CodeWarrior TRK can return these error codes:

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

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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td><code>kDSReadMemory</code> (defined in the <code>msgcmd.h</code> file).</td>
</tr>
<tr>
<td>options</td>
<td>ui8</td>
<td>This field can contain one of these values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DS_MSG_MEMORY_SEGMENTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DS_MSG_MEMORY_PROTECTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DS_MSG_MEMORY_USERVIEW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see the <code>msgcmd.h</code> file.</td>
</tr>
<tr>
<td>length</td>
<td>ui16</td>
<td>The length of the memory section (a maximum of 2048 bytes).</td>
</tr>
<tr>
<td>start</td>
<td>ui32</td>
<td>The starting address of the memory section.</td>
</tr>
</tbody>
</table>

**Return Values**

This message causes CodeWarrior TRK to return these values:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>ui16</td>
<td>The length of the data read (a maximum of 2048 bytes).</td>
</tr>
<tr>
<td>data</td>
<td>ui8[]</td>
<td>The data read (a maximum of 2048 bytes).</td>
</tr>
</tbody>
</table>

**Error Codes**

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyCWDSError</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
</tbody>
</table>
### Messages Sent by Debugger

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyCWDSException</td>
<td>An exception was thrown while processing the request.</td>
</tr>
<tr>
<td>kDSReplyInvalidMemoryRange</td>
<td>The specified memory range is invalid.</td>
</tr>
<tr>
<td>kDSReplyNotStopped</td>
<td>The target application is running and must be stopped before the debugger issues this request.</td>
</tr>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
<tr>
<td>kDSReplyParameterError</td>
<td>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.</td>
</tr>
<tr>
<td>kDSReplyUnsupportedOptionError</td>
<td>The specified value of the options field is unsupported.</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSReadRegisters (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>
| options      | ui8   | This field can contain one of these values:  
• kDSRegistersDefault |

*Table continues on the next page...*
### Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstRegister</td>
<td>ui16</td>
<td>The number of the first register in the sequence.</td>
</tr>
<tr>
<td>lastRegister</td>
<td>ui16</td>
<td>The number of the last register in the sequence.</td>
</tr>
</tbody>
</table>

**Return Values**

This message causes CodeWarrior TRK to return this value:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerData</td>
<td>void*</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Error Codes**

CodeWarrior TRK can return these error codes:

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

**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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSReset (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSStep (defined in the msgcmd.h file).</td>
</tr>
<tr>
<td>options</td>
<td>ui8</td>
<td>This field can contain one of these values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSStepIntoCount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSStepOverCount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see the msgcmd.h file.</td>
</tr>
<tr>
<td>count</td>
<td>ui8</td>
<td>The number of instructions to step over.</td>
</tr>
</tbody>
</table>

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

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

Table continues on the next page...
### Field Size Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeEnd</td>
<td>ui32</td>
<td>The end address of the specified memory range.</td>
</tr>
</tbody>
</table>

### Return Values

None.

### Error Codes

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyNotStopped</td>
<td>The target application is running and must be stopped before the debugger issues this request.</td>
</tr>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
<tr>
<td>kDSReplyParameterError</td>
<td>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.</td>
</tr>
<tr>
<td>kDSReplyUnsupportedOptionError</td>
<td>The specified value of the options field is unsupported.</td>
</tr>
</tbody>
</table>

### 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:
### Messages Sent by Debugger

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSStop (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>

**Return Values**

None.

**Error Codes**

CodeWarrior TRK can return this error code:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyError</td>
<td>Unknown problem in transmission.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSSupportMask (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>

**Return Values**
This message causes CodeWarrior TRK to return these values:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mask</td>
<td>ui8[32]</td>
<td>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.</td>
</tr>
<tr>
<td>protocolLevel</td>
<td>ui8</td>
<td>The protocol level supported by CodeWarrior TRK. For more information, see Command Sets.</td>
</tr>
</tbody>
</table>

**Error Codes**

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyCWDSError</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSVersions (defined in the msgcmd.h file).</td>
</tr>
</tbody>
</table>

**Return Values**

This message causes CodeWarrior TRK to return these values:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernelMajor</td>
<td>ui8</td>
<td>The major version number for CodeWarrior TRK. (In version 1.2, the kernelMajor is 1.)</td>
</tr>
<tr>
<td>kernelMinor</td>
<td>ui8</td>
<td>The minor version number for CodeWarrior TRK. (In version 1.2, the kernelMinor is 2.)</td>
</tr>
<tr>
<td>protocolMajor</td>
<td>ui8</td>
<td>The major version number for the messaging protocol. (In version 1.2, the protocolMajor is 1.)</td>
</tr>
<tr>
<td>protocolMinor</td>
<td>ui8</td>
<td>The minor version number for the messaging protocol. (In version 1.2, the protocolMinor is 2.)</td>
</tr>
</tbody>
</table>

**Error Codes**

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyCWDSError</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
</tbody>
</table>

**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:

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

Return Values

This message causes CodeWarrior TRK to return this value:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>ui16</td>
<td>The amount of memory written.</td>
</tr>
</tbody>
</table>

Error Codes

CodeWarrior TRK can return these error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyCWDError</td>
<td>An unknown error occurred while processing the request.</td>
</tr>
<tr>
<td>kDSReplyCWDSException</td>
<td>An exception was thrown while processing the request.</td>
</tr>
<tr>
<td>kDSReplyInvalidMemoryRange</td>
<td>The specified memory range is invalid.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Error Code

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDSReplyNotStopped</td>
<td>The target application is running and must be stopped before the debugger issues this request.</td>
</tr>
<tr>
<td>kDSReplyPacketSizeError</td>
<td>The length of the message is less than the minimum for a message of that type.</td>
</tr>
<tr>
<td>kDSReplyParameterError</td>
<td>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.</td>
</tr>
<tr>
<td>kDSReplyUnsupportedOptionError</td>
<td>The specified value of the options field is unsupported.</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSWriteRegisters (defined in the msgcmd.h file).</td>
</tr>
<tr>
<td>options</td>
<td>ui8</td>
<td>This field can contain one of these values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSRegistersDefault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSRegistersFP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSRegistersExtended1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• kDSRegistersExtended2</td>
</tr>
</tbody>
</table>

Table continues on the next page...
<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstRegister</td>
<td>ui16</td>
<td>The number of the first register in the sequence.</td>
</tr>
<tr>
<td>lastRegister</td>
<td>ui16</td>
<td>The number of the last register in the sequence.</td>
</tr>
<tr>
<td>registerData</td>
<td>ui32[]</td>
<td>An array of register values. The size of each element depends on the size of the registers.</td>
</tr>
</tbody>
</table>

Return Values

None.

Error Codes

CodeWarrior TRK can return these error codes:

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

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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>ui8</td>
<td><code>kDSNotifyException</code> (defined in the <code>msgcmd.h</code> file).</td>
</tr>
<tr>
<td>target-defined info</td>
<td>target-specific</td>
<td>The value of this field, which provides state information about the target board, differs for each target processor.</td>
</tr>
<tr>
<td>Field</td>
<td>Size</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>command</td>
<td>ui8</td>
<td>kDSNotifyStopped (defined in the msgcmd.h file).</td>
</tr>
<tr>
<td>target-defined info</td>
<td>target-specific</td>
<td>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().</td>
</tr>
</tbody>
</table>

### 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:
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:

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

Return Values

This message causes the debugger to return these values:

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io_result</td>
<td>ui8</td>
<td>Standard I/O result returned by the debugger (kDSIONoError, kDSIOError, or kDSIOEOF).</td>
</tr>
<tr>
<td>length</td>
<td>ui16</td>
<td>The amount of data read.</td>
</tr>
<tr>
<td>file_data</td>
<td>ui8[]</td>
<td>The data (a maximum of 2048 bytes).</td>
</tr>
</tbody>
</table>
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:

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

Return Values
This message causes the debugger to return these values:
### Messages Sent by CodeWarrior TRK

<table>
<thead>
<tr>
<th>Return Value Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io_result</td>
<td>ui8</td>
<td>Standard I/O result returned by the debugger (kDSIONoError, kDSIOError, or kDSIOEOF).</td>
</tr>
<tr>
<td>length</td>
<td>ui16</td>
<td>The amount of data written.</td>
</tr>
</tbody>
</table>

### 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:

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

Responds to a Continue request from the debugger.

```c
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**

`msgndlr.c`

**Board-specific?**

No.

**See Also**

- `TargetContinue()`
- `Continue`
6.4 DoCPUPhase()

Responds to a CPUPhase request from the debugger.

```c
DSError DoCPUPhase ( MessageBuffer* b );
```

*b*

The message buffer that contains the CPUPhase request and the reply to the request. The CPUPhase request message does not contain input arguments. For more information, see CPUPhase.

**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 CPUPhase.

**Source File**

msgcmsglrlr.c

**Board-Specific?**

No.

**See Also**

msgcmd.h

6.5 DoFlushCache

Responds to a FlushCache request from the debugger.

```c
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?
6.7 DoReadMemory()

DoReadMemory() reads a section of memory from the target board in response to a ReadMemory request from the debugger.

```c
DSError DoReadMemory ( MessageBuffer* 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.

```c
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().

- **kDSRegistersExtended2**
  
  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**
6.9 DoReset()

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

```c
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**

msghdlr.c

**Board-specific?**

No.

**See Also**
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.

```c
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()
6.11 DoStop()

Responds to a Stop request from the debugger.

```c
DSError DoStop ( MessageBuffer* 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.

```c
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.

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

b
**DoWriteMemory()**

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.

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

```c
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**

msgnhrd1r.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.

```c
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()`.
- `kDSRegistersExtended2`
  - **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.

```c
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.

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

c
Pointer to the output variable for the character read.

**Returns**
Returns one of these UARTError error codes:

- kUARTNoData
6.20 ReadUART1()

Reads one byte from the serial device.

```c
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);

CodeWarrior Target Resident Kernel Reference, Rev. 10.5.1, 01/2016

Freescale Semiconductor, Inc.
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 to be read or written to the file.

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.

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

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.

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

Returns a DSError error code.

None.

support.c

No.

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()`
- `ExceptionHandler()`
- `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.

```c
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).

```c
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.

dend

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.

```c
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).
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.

```c
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()
6.33  TargetAccessExtended2()

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

```c
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.

```c
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.

dend
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.

```c
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.

```c
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.

```c
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.

```c
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:

- What is AppTRK?
- Differences Between AppTRK and CodeWarrior TRK
- Using AppTRK
- Modifying AppTRK
- Creating AppTRK for a Non-Freescale Linux Image
- Conditions under which AppTRK Must Be Modified

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/pd.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
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.

   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.

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

<table>
<thead>
<tr>
<th>_reset()</th>
<th>76</th>
</tr>
</thead>
</table>

## A

- ACK Messages 32
- Alignment 31
- AppTRK Reference 117

## B

- Building TI TL16C552a UART Driver 40
- Byte Order 31

## C

- Changing Data Transmission Rate 41
- Changing Maximum Message Length 46
- Changing ReadMemory-Related Code 44
- Changing SupportMask-Related Code 44
- Changing Versions-Related Code 45
- Changing WriteMemory-Related Code 44
- Checksum Values 24
- CodeWarrior TRK Architecture 13
- CodeWarrior TRK Communications 19
- CodeWarrior TRK Concepts 13
- CodeWarrior TRK Core 13
- CodeWarrior TRK Data Frames 22
- CodeWarrior TRK Execution States 14
- CodeWarrior TRK Function Reference 75
- CodeWarrior TRK Initializations 17
- CodeWarrior TRK Memory Layout 16
- CodeWarrior TRK RAM Sections 17
- Command Sets 51
- Conditions under which AppTRK Must Be Modified 123
- Connect 52
- Continue 53
- CPUType 54
- Creating AppTRK for a Non-Freescale Linux Image 122
- Customizing Checksum Values 47
- Customizing CodeWarrior TRK 37
- Customizing CodeWarrior TRK Initializations 38
- Customizing CodeWarrior TRK to be Interrupt-Driven 41
- Customizing CPU Speed 42
- Customizing Debug Services 43
- Customizing Exception Handling 47
- Customizing Memory Locations 46
- Customizing Serial Communications 38
- Customizing Target Board Name 48
- Customizing usr_put_config.h for Debugging 48

## D

- Data Transmission Rate 21
- Debug Message Interface Level 30
- Debug Message Interface Reference 51
- Differences Between AppTRK and CodeWarrior TRK 117
- DoConnect() 76
- DoContinue() 77
- DoCPUType() 78
- DoFlushCache 78
- DoNotifyStopped() 79
- DoReadMemory() 80
- DoReadRegisters() 81
- DoReset() 82
- DoStep() 83
- DoStop() 84
- DoSupportMask() 84
- DoVersions() 85
- DoWriteMemory() 86
- DoWriteRegisters() 87

## E

- Encoding Single-Byte Checksum Values 24
- Escape Sequences 27

## F

- FlushCache 55
- Framing Level 21

## I

- InitializeIntDrivenUART() 89
- InitializeUART() 90
- InterruptHandler() 90
- Introduction 9

## M

- Message Length 32
- Message Queues 15
- Messages Sent by CodeWarrior TRK 70
- Messages Sent by Debugger 51
- Modifying AppTRK 121
- Modifying Existing UART Drivers 39
- Modifying Serial Communication Functions 38

## N

- NAK Messages 34

---

**CodeWarrior Target Resident Kernel Reference**

Freescale Semiconductor, Inc.

125
NotifyException 70
NotifyStopped 71

O
Other CodeWarrior Documentation 11
Other Documentation 11
Overview of this Manual 9

P
Preventing Transmission Failure 30

R
ReadFile 72
ReadMemory 57
ReadRegisters 58
ReadUART1() 92
ReadUARTN() 92
ReadUARTPoll() 91
ReadUARTString() 93
Related Documentation 10
Reliable Message Delivery 28
Reply Messages 32
Request and Notification Handling 16
Request and Notification Messages 31
Reset 60
Responding to a NAK Reply Message 28
Responding When No Reply Message is Received 29

S
Serial Communications Settings 20
Step 61
Stop 63
SuppAccessFile() 94
SupportMask 64
SwapAndGo() 96

T
TargetAccessDefault() 102
TargetAccessExtended1() 103
TargetAccessExtended2() 105
TargetAccessFP() 106
TargetAccessMemory() 96
TargetAddExceptionInfo() 98
TargetAddStopInfo() 98
Target Application RAM Sections 17
TargetContinue() 99
TargetFlushCache() 100
TargetInterrupt() 101
TargetSingleStep() 107
TargetStepOutOfRange() 108

TargetSupportMask() 109
TargetVersions() 110
TerminateUART() 111
TransportIrqHandler() 112
Transport Level 20

U
Using AppTRK 118
Using Multi-byte Checksum Values 27

V
ValidMemory32() 112
Verifying Single-Byte Checksum Values 25
Versions 65

W
What is AppTRK? 117
WriteFile 73
WriteMemory 67
WriteRegisters 68
WriteUART1() 114
WriteUARTN() 114
WriteUARTString() 115

Index
CodeWarrior Target Resident Kernel Reference
126 Freescale Semiconductor, Inc.
Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, CodeWarrior, QorIQ, QorIQ Qonverge, and StarCore are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2009–2016 Freescale Semiconductor, Inc.