ARM® RMTarget

Integration Guide

ARM

ARM DUI 0142A

ARM RMTarget
Integration Guide

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
December 2000	A	First release

Proprietary Notice

Contents
Programmer's Guide

Preface

This preface introduces the Integration Guide for the ARM® RMTarget. It contains the following sections:

About this book

This book describes how to integrate ARM RealMonitor into your target application. It documents only the target-side functionality of RealMonitor. It does not discuss RMHost, which is described fully in the *ARM RMHost User Guide*.

Note

It is recommended you read this integration guide before reading the *ARM RMHost User Guide* because this guide describes how to build RMTarget, and integrate it with your own application, which you must do before performing real-time debugging with RMHost.

Intended audience

This book is written for programmers who must integrate RMTarget into an application. It assumes that you are an experienced C programmer who is familiar with the target application itself, and that you are aware of the real-time debugging features you can use with your application. You must also be familiar with the ARM architecture.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction to RealMonitor***

Read this chapter for an introduction to the host-side and target-side functionality of RealMonitor, and to find out the system requirements necessary to use RealMonitor. It also describes the reasons why RealMonitor has been developed.

Chapter 2 ***Introduction to RMTarget***

Read this chapter for a description of RMTarget, and how it communicates with, and complements, RMHost. This chapter also provides the complete procedure for preparing a RealMonitor-enabled image using RMTarget. It also describes the RMTarget source files, and how to port RMTarget to a new processor or board.

Chapter 3 ***RealMonitor Demonstrations***

Read this chapter for a description of RealMonitor demonstrations provided with the *ARM Firmware Suite* (AFS) 1.2, and for step-by-step instructions on setting up the demonstration applications, and debugging them using AXD.

Chapter 4 ***Building RMTarget***

Read this chapter for instructions on building RMTarget, and for a description of the available build options.

Chapter 5 ***Integration***

Read this chapter for instructions on integrating RMTarget with your application or RTOS after you have built RMTarget. There are also instructions for setting up exception handling so that RealMonitor exception handlers can co-exist with those in your application.

Chapter 6 ***Application Program Interface***

Read this chapter for a complete description of the RealMonitor *Application Program Interface* (API).

Chapter 7 ***RealMonitor Protocol***

Read this chapter for a complete description of the RealMonitor protocol, which governs communication between the target and host.

Typographical conventions

The following typographical conventions are used in this book:

bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.
italic	Highlights special terminology, denotes internal cross-references, and citations.
`typewriter`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`typewriter`	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for RealMonitor.

ARM periodically provides updates and corrections to its documentation. See `www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at the ARM website.

ARM® publications

This book contains information that is specific to RMTarget. Refer to the following books for related information:

- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *AXD and armsd Debuggers Guide* (ARM DUI 0066)
- *ADS Compilers and Libraries Guide* (ARM DUI 0067)
- *ADS Debug Target Guide* (ARM DUI 0058)
- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *Multi-ICE User Guide* (ARM DUI 0048)
- *Trace Debug Tools User Guide* (ARM DUI 0118)
- *ARM Firmware Suite Reference Guide* (ARM DUI 0102)
- *ARM Firmware Suite User Guide* (ARM DUI 0136)
- *ARM RMHost User Guid*e (ARM DUI 0137).

Other publications

Refer to the following publications for additional information:

- E3459-97002, *Emulation for the ARM7/ARM9 User's Guide*, Agilent, 1999.

To access this document, see the website `www.agilent.com`.

Feedback

ARM Limited welcomes feedback on both RMTarget and its documentation.

Feedback on RMTarget

If you have any problems with RMTarget, please contact your supplier. To help them provide a rapid and useful response, please give:

- details of the release you are using
- details of the host and target you are running on
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction to RealMonitor

This chapter introduces ARM RealMonitor. It describes the reasons why RealMonitor has been developed, and the advantages of using RealMonitor over other debugging methods. It gives an overview of the host-side and target-side functionality, and how RealMonitor operates as a single mechanism. It also describes the RealMonitor system requirements, both target-side and host-side.

This chapter contains the following sections:

1.1 About RealMonitor

RealMonitor is a small program that, when integrated into your target application or *Real-Time Operating System* (RTOS), allows you to observe and debug your target while parts of your application continue to run.

A debugger such as the *ARM eXtended Debugger* (AXD), that runs on a host computer, can connect to the target to send commands and receive data. This communication between host and target is illustrated in Figure 1-1 on page 13.

The target component of RealMonitor, RMTarget, communicates with the host component, RMHost, using the *Debug Communications Channel* (DCC), which is a reliable link whose data is carried over the JTAG connection.

While your application is running, RMTarget typically uses IRQs generated by the DCC. This means that if your application also wants to use IRQs, it must pass any DCC-generated interrupts to RealMonitor.

To allow nonstop debugging, the EmbeddedICE-RT® logic in the processor generates a Prefetch Abort exception when a breakpoint is reached, or a Data Abort exception when a watchpoint is hit. These exceptions are handled by the RealMonitor exception handlers that inform the user, by way of the debugger, of the event. This allows your application to continue running without stopping the processor.

RealMonitor considers your application to consist of two parts:

- a foreground application running continuously, typically in User, System, or SVC mode
- a background application containing interrupt and exception handlers that are triggered by certain events in your system, including:
  - IRQs or FIQs
  - aborts caused by your foreground application
  - SWIs called by your foreground application
  - undefined instructions in your foreground application.

When one of these exceptions occur that is not handled by your application, the following happens:

- RealMonitor enters a loop, polling the DCC.
- RealMonitor stops the foreground application. Both IRQs and FIQs continue to be serviced if they were enabled by the application at the time the foreground application was stopped.

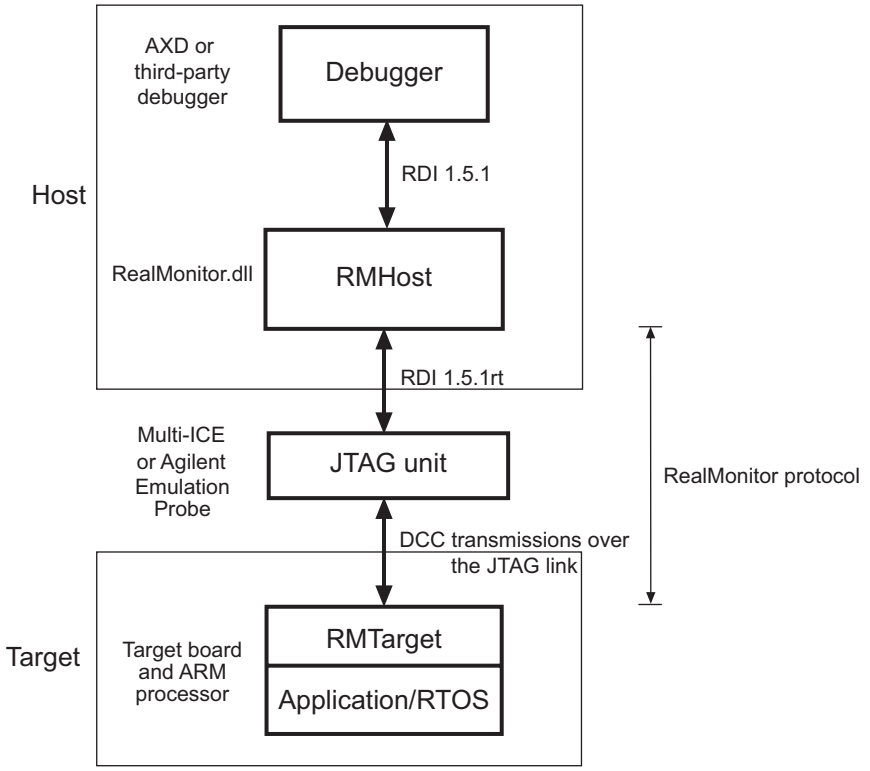

Figure 1-1 RealMonitor components

RealMonitor has the following features:

- It allows time-critical interrupt code to continue executing while other foreground application code is being debugged. This is particularly useful in systems that have a hard real-time requirement for interrupt-driven code that controls physical hardware such as a hard disk controller.
- It allows you to perform the following debug actions on the target:
  - set and clear breakpoints
  - set and clear watchpoints
  - examine and modify memory
  - examine and modify registers while the foreground application is stopped
  - examine and modify coprocessor registers
  - debug both ROM-based and RAM-based code
  - debug from within User, IRQ, Supervisor, and System modes.

Note

The types of debugging operations you can perform on a RealMonitor-enabled application depend on whether your foreground application is stopped or running. See Table 1-1 for a complete list of debugging tasks available, and whether you can perform them on a stopped or running foreground application.

- It allows you to establish a debug session to a currently running system without halting or resetting the system.
- It can pass application-specific or RTOS-specific information asynchronously back to the development host.
- It has minimal system impact on both code and data usage, and in execution overhead. Most of the functionality is implemented in RMHost, so that in a base configuration, RMTarget takes up only about 2KB of code and data.

A subset of RealMonitor debugging features can be utilized while the foreground application continues to run. Table 1-1 shows which features are available. For complete details on debugging a RealMonitor-enabled target application, see the *Debugging* chapter of the *ARM RMHost User Guide*.

Table 1-1 RealMonitor debugging features

Feature	Foreground application stopped	Foreground application running
Examine or modify processor registers	Yes	No
Start foreground process	Yes	N/A
Stop foreground process	N/A	Yes
Set and clear breakpoints	Yes	Yes
Set and clear watchpoints	Yes	Yes
Use SWI semihosting	No	Yes
Examine or modify memory	Yes	Yes
Application-specific or RTOS-specific data logging	Yes	Yes
Examine or modify coprocessor registers	Yes	Yes
Sample the *program counter* (pc)	No	Yes

1.2 About RealMonitor functionality

As shown in Figure 1-1 on page 13, RealMonitor is split into two functional components:

RMHost This is located between a debugger, such as AXD, and a JTAG unit, such as Multi-ICE®. The RMHost controller, `RealMonitor.dll`, converts generic *Remote Debug Interface* (RDI) requests from the debugger into DCC-only RDI messages for the JTAG unit. For complete details on debugging a RealMonitor-integrated application from the host, see the *ARM RMHost User Guide*.

RMTarget This is integrated with your application, and runs on the target hardware. It uses the EmbeddedICE logic, and communicates with the host using the DCC. For more details on RMTarget functionality, see *About RMTarget* on page 22.

These two components communicate with each other over the DCC using the RealMonitor protocol. This protocol reflects both the need to keep the target-side RealMonitor as simple as possible, and the fact that the DCC is highly reliable. See Chapter 7 *RealMonitor Protocol* for a complete description of the protocol.

1.3 The development of RealMonitor

RealMonitor is a lightweight debug monitor that allows interrupts to be serviced while you debug your foreground application. It communicates with the host using the DCC, which is present in most ARM cores. RealMonitor provides advantages over the traditional methods for debugging applications in ARM systems. The traditional methods include:

- Angel® (a target-based debug monitor)
- Multi-ICE and EmbeddedICE logic (a hardware-based debug solution).

Although both of these methods provide robust debugging environments, neither is suitable as a lightweight real-time monitor.

Angel is designed to load and debug independent applications that can run in a variety of modes, and communicate with the debug host using a variety of connections (such as a serial port or ethernet). Angel is required to save and restore full processor context, and the occurrence of interrupts can be delayed as a result. Angel, as a fully functional target-based debugger, is therefore too heavyweight to perform as a real-time monitor.

Multi-ICE is a hardware debug solution that operates using the EmbeddedICE unit that is built into most ARM processors. To perform debug tasks such as accessing memory or the processor registers, Multi-ICE must place the core into a debug state. While the processor is in this state, which can be millions of cycles, normal program execution is suspended, and interrupts cannot be serviced.

RealMonitor combines features and mechanisms from both Angel and Multi-ICE to provide the services and functions that are required. In particular, it contains both the Multi-ICE communication mechanisms (the DCC using JTAG), and Angel-like support for processor context saving and restoring.

1.4 How RealMonitor works

In general terms, the RealMonitor operates as a state machine, as shown in Figure 1-2. RealMonitor switches between running and stopped states, in response to packets received by the host, or due to asynchronous events on the target.

RMTarget supports the triggering of only one breakpoint, watchpoint, stop, or semihosting SWI at a time. There is no provision to allow nested events to be saved and restored. So, for example, if your application has stopped at one breakpoint, and another breakpoint occurs in an IRQ handler, RealMonitor enters a panic state. No debugging can be performed after RealMonitor enters this state.

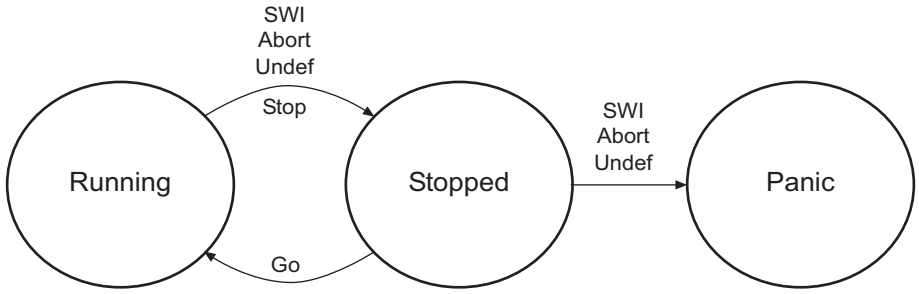

Figure 1-2 RealMonitor as a state machine

1.5 RealMonitor system requirements

ARM RealMonitor 1.0 system requirements are as follows:

Target hardware

Your target processor must contain EmbeddedICE logic. If your EmbeddedICE logic contains a real-time extension (-RT), nonstop breakpoints and nonstop watchpoints are also supported.

If you want to drive RealMonitor using IRQs, you must wire the DCC signals `COMMRX` and `COMMTX`, that are output by the processor, into an interrupt controller on your board. If you do not do this, you must poll RealMonitor from your foreground application (see *Reducing interrupt latency by polling the DCC* on page 512).

RMTarget supports the following boards:

- Integrator®/AP
- Integrator/SP.

RMTarget supports the following processors:

- ARM946E-S (CM946E-S core module)
- ARM966E-S (CM966E-S core module)
- ARM1020T (CM1020T core module).

Note

If you do not have this hardware, you can port RMTarget to your own board or processor (see *Porting RMTarget* on page 211).

JTAG unit RealMonitor works only with a communication transport mechanism that supports the DCC. The following JTAG units are supported by RealMonitor 1.0:

- Multi-ICE version 2.0
- Agilent Emulation Probe (E5900B).

Note

Debugging response times vary depending on the JTAG unit you are using.

All performance targets described in this guide assume the use of Multi-ICE version 2.0 or later.

Host RMHost supports the following platforms:

- Windows 95

- Windows 98
- Windows NT 4.0
- Windows 2000
- HP-UX 10.20
- Solaris 2.6
- Solaris 7.0.

Host tools You can use the version of AXD provided with ADS version 1.1 or later to debug a RealMonitor-enabled application. You can also use RealMonitor with any other RDI 1.5.1rt-compliant debugger.

Note

RealMonitor is not supported when using *ARM Debugger for Windows* (ADW).

Chapter 2
Introduction to RMTarget

This chapter describes the role of RMTarget, and how it communicates with, and complements, RMHost. This chapter also describes the complete procedure for preparing an RM-enabled image using RMTarget. It also describes the RMTarget source files, and how to port RMTarget to a new processor or board.

This chapter contains the following sections:

2.1 About RMTarget

RMTarget refers to the target-side functionality of RealMonitor, as delivered in source form with *ARM Firmware Suite* (AFS) version 1.2. RMTarget consists of a group of files that you must build and integrate into your application or RTOS to allow it to be debugged in real-time. RMTarget is distinct from RMHost, which is delivered on the ARM RealMonitor CD.

The RMHost controller, `RealMonitor.dll`, is used by your debugger to allow you to debug images that have been integrated with RMTarget. Figure 1-1 on page 13 illustrates these two functional parts of RealMonitor, and how they communicate with each other.

Note

This guide describes only the target-side functionality of RealMonitor, and the steps you must perform to integrate RMTarget with your application. Because you must build and integrate RMTarget before you can perform real-time debugging, it is strongly suggested you read this guide before reading the *ARM RMHost User Guide*. The User Guide provides details on selecting the RMHost controller and debugging your RealMonitor-integrated target application.

RMTarget runs only when it is given control of the system when particular exceptions occur. For example, if a software breakpoint occurs, the undefined exception handler is called, which passes control to RMTarget. You must take care to ensure that the exception handlers of your own application can co-exist with the RealMonitor method of exception handling (see *Handling exceptions* on page 55).

RMTarget code runs in the Undef processor mode because this is least likely to interfere with the exception handling of your own application. This can prevent you from debugging application code that runs in undefined mode.

Note

You cannot debug code running in Undef mode while using RealMonitor. When designing your application, you must therefore ensure that as little of your code as possible runs in Undef mode.

RMTarget also features the following:

- *Small code and data size* on page 23
- *Small execution overhead* on page 23
- *Low interrupt latency* on page 23
- *Tool chain support* on page 23
- *Thumb® support* on page 24.

2.1.1 Small code and data size

RMTarget is designed to have as little code and data as possible so that you can install it into production systems with minimal impact. The default configuration of the RMTarget library is approximately 2KB in size (see *Default RMTarget settings* on page 429). For details on how to determine the size of any custom-built RMTarget library, see *Determining library size* on page 411.

2.1.2 Small execution overhead

RealMonitor is designed to have a minimal *execution overhead*. This applies to the number of processor clock cycles per unit of time that RealMonitor claims from your application.

The execution overhead varies depending on both the configuration of your target system, and the type of debug operations you perform.

In any given system, the maximum execution overhead is determined by the rate at which RMTarget and RMHost communicate with each other. If you want to reduce the speed of the communications link (and therefore, the execution overhead of RMTarget), it is suggested that you decrease the speed of your JTAG connection. See the documentation that accompanies your JTAG unit for details on how to do this.

It is expected that the worst-case performance loss occurs if you write escape word value `0xA1 0x46 0xF0 0x98` to memory using a `WriteBytes` opcode, where the last byte written causes a Data Abort, and where the target was not already sending data back to the host (see *Escape sequence handling* on page 74).

2.1.3 Low interrupt latency

Interrupt latency refers to the amount of time that IRQs are disabled by RealMonitor, that is, the time in which no new IRQs can be handled by the processor. It is recommended that, in all cases, you keep interrupt latency as small as possible.

2.1.4 Tool chain support

To ensure RMTarget remains small and efficient, its source code and build system make use of the features and extensions provided in the ADS tool chain. To build RMTarget using a third-party tool chain, you might have to make changes to the source code and build system. See the documentation that accompanies your tool chain for details of the changes you must make.

2.1.5 Thumb® support

Although RealMonitor does not use any Thumb® code itself, you can integrate RMTarget with applications that use Thumb code. This does not require you to make any specific changes to your C or assembly language code. The ARM linker automatically generates interworking veneers for any functions that need to be called from Thumb code, when given the options `-apcs /interwork`. For a description of the linker options, see the *ARM/Thumb Procedure Call Standard* (ATPCS) section in the C and C++ Compilers chapter of the *ADS Compilers and Libraries Guide*.

Similarly, the ability to monitor and debug Thumb code does not require any special measures to be taken by RealMonitor. AXD contains the necessary code to support Thumb debugging.

2.2 Preparing a RealMonitor-enabled image

This section describes the step-by-step procedure required to use RMTarget to prepare an RM-enabled application. Each step provides a cross-reference to a section either in this guide, or in another source that describes the step more fully.

Note

To become familiar with building the RMTarget library and debugging RealMonitor-enabled applications, see Chapter 3 *RealMonitor Demonstrations*.

To prepare a RealMonitor-enabled image using RMTarget:

1. Install AFS version 1.2 if it has not already been installed on your system. The RMTarget source files are installed into the `RealMonitor` directory of the AFS source tree. See *RMTarget source files* on page 27 for a diagram of the RMTarget source tree and a description of each file.
2. Build the RMTarget library with your chosen build options, enabling the RealMonitor features your application will utilize, and disabling the others to save code space. There are two methods available to build RMTarget:
   - *Building using CodeWarrior IDE* on page 42
   - *Building using the Makefile* on page 47.
3. You must integrate RMTarget into your application, as follows:
   - In your initialization code, set up sufficient stacks for the processor modes, such as IRQ mode, required by RealMonitor. The stacks required depend on the build options you have set.
   - Make changes to the source code of both your application and RMTarget to ensure that their exception handling mechanisms complement each other.
   - In your initialization code, call the function `RM_Init()`.

     Note

     When you call this function, your application must be in a privileged mode, with IRQs disabled.

   See *Integration procedure* on page 53 for complete details.

   Note

   For other considerations when integrating, see *Other integration considerations* on page 510 for details.
4. Build your own application using your own method of building.

Note

You can also build RMTarget and your own application as part of the same build. See *Building RMTarget with your application* on page 49 for details on how to do this.

5. Link the object files and libraries of your application with the RMTarget library, `rm.a`, using the method relevant to the way you have built RMTarget, either:
   - *Linking using armlink* on page 49, if you used the Makefile to build
   - *Linking using CodeWarrior IDE* on page 46, if you are using CodeWarrior IDE.

   For details on using `armlink`, see the description of the ARM Linker in the *ADS Compilers and Libraries Guide*.

   A RealMonitor-enabled executable image is produced. You can now debug the RealMonitor-enabled image using RMHost (see the *ARM RMHost User Guide*).

2.3 RMTarget source files

The RMTarget files are provided in source form when you install AFS 1.2. Figure 2-1 shows all RMTarget files, and how they fit into the source tree of AFS.

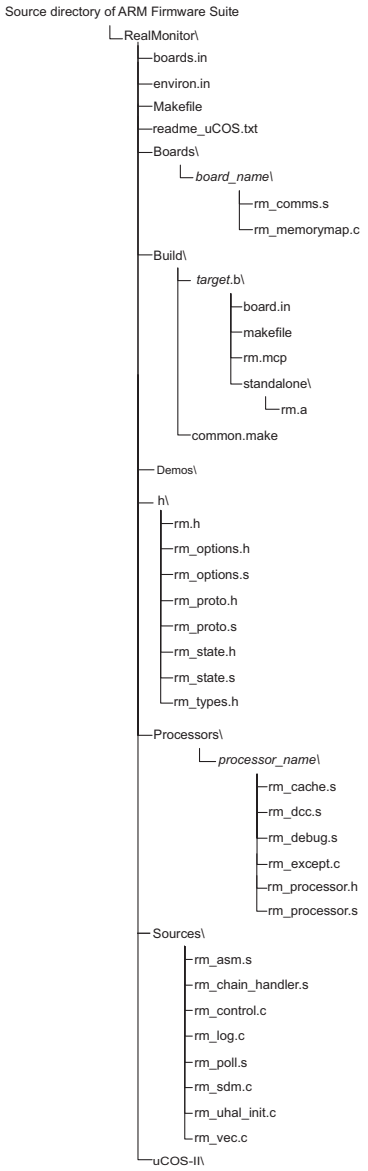

Figure 2-1 RMTarget source files

You must compile the RMTarget source files before you can use them. See Chapter 4 *Building RMTarget* for complete details on building RMTarget. The RMTarget source files are as follows:

`boards.in` Lists the available board/processor variants that are supported by RealMonitor. This file is included when you run `Makefile`.

`environ.in` Lists the available board/processor variants that are supported by RealMonitor. This file is included when you run `Makefile`.

`Makefile` GNU makefile for building all board and processor variants of RMTarget.

`readme_uCOS.txt`

Contains details of an RMTarget port to uC/OS-II (see *Integrating RMTarget into an RTOS* on page 514).

`Boards\`*`board_name`*`\rm_comms.s`

Board-specific assembly language macros relevant to the communications link.

`Boards\`*`board_name`*`\rm_memorymap.c`

Board-specific C code that describes the memory map of the target system.

`Build\`*`target`*`.b\board.in`

Contains rules for building a specific board/processor variant of RMTarget.

`Build\`*`target`*`.b\makefile`

GNU makefile for building one specific board/processor variant of the RMTarget library.

`Build\target.b\rm.mcp`

The CodeWarrior IDE RMTarget project file for building a specific board/processor variant of the RMTarget library.

`Build\target.b\standalone\rm.a`

The RMTarget library that is built for a specific board and processor (see *Building RMTarget using the makefile* on page 47).

`Build\common.make`

GNU makefile containing the rules for building RMTarget that are common to all board and processor variants.

`h\rm.h` A C header file that defines the external function names that can be called by your application.

`h\rm_options.h`

Default values and validation for the RMTarget build options (for C code).

`h\rm_options.s`

Default values and validation for the RMTarget build options (for assembly language code).

`h\rm_proto.h`

C code definitions relating to the RealMonitor protocol.

`h\rm_proto.s`

ARM assembly language definitions relating to the RealMonitor protocol.

`h\rm_state.h`

Defines the data structures that are used to hold the internal state of RealMonitor (see *Data structures* on page 68).

`h\rm_state.s`

Defines the data structure that are used to hold the internal state of RealMonitor (see *Data structures* on page 68).

`h\rm_types.h`

Defines the basic data types used by RealMonitor. See *Basic data types* on page 62 for the data types that are defined in this file.

`Processors\`*`processor_name`*`\rm_cache.s`

Processor-specific assembly language macros for manipulating the cache (see *Cache handling functions* on page 652).

`Processors\`*`processor_name`*`\rm_dcc.s`

Processor-specific assembly language macros for accessing the DCC.

`Processors\`*`processor_name`*`\rm_debug.s`

Processor-specific assembly macros for accessing the debug hardware.

`Processors\`*`processor_name`*`\rm_except.c`

Processor-specific C code for exception handling.

`Processors\processor_name\rm_processor.h`

Processor-specific configuration options.

`Processors\processor_name\rm_processor.s`

Processor-specific configuration options.

`Sources\rm_control.c`

C sources for implementing the RealMonitor protocol.

`Sources\rm_asm.s`

ARM assembly language code for exception handling, saving and restoring registers, and other low-level functionality that cannot be implemented in C.

`Sources\rm_chain_handler.s`

Assembly language code for managing µHAL exception chains.

`Sources\rm_log.c`

C sources for performing data logging.

`Sources\rm_poll.s`

ARM assembly language code for using RealMonitor in a polled mode, rather than an interrupt-driven mode.

`Sources\rm_sdm.c`

C sources for supplying the *Self-Describing Module* (SDM) information string.

`Sources\rm_uhal_init.c`

Initialization code for using RealMonitor with µHAL.

`Sources\rm_vec.c`

C sources for installing exception vectors.

Note

The subdirectories and files that reside in the `Demos` directory are described in *Directory structure of the RealMonitor demonstrations* on page 34.

The subdirectories and files that reside in the `uCOS` directory are described in the `readme_uCOS.txt` file, which is located in the root directory of the RealMonitor installation (see *Integrating RMTarget into an RTOS* on page 514).

2.4 Porting RMTarget

This section describes the procedures you must follow to port RMTarget:

- *Porting to a new processor*
- *Porting to a new board* on page 213
- *Adding a new build target* on page 215.

See *RealMonitor system requirements* on page 18 for details on the target hardware that is supported by RealMonitor.

2.4.1 Porting to a new processor

To port RMTarget to a new processor:

1. Create a new directory `Processors\`*`processor_name`*, where *`processor_name`* is the name of the new processor to which you want to port.

   Note

   The *`processor_name`* must be one that is accepted by both the C Compiler and ARM assembler using the `-cpu` option, as in:

   ```
   armcc -cpu ARM946E-S
   ```

2. Copy all the source files from an existing *`processor_name`* directory into the *`processor_name`* directory you have created (see *RMTarget source files* on page 27). To minimize the amount of work you need to perform, it is recommended that you copy the files from a processor that is most similar to the one to which are attempting to port.

   Make the following modifications:

   - In `rm_cache.s`, modify the implementation of the functions `rm_Cache_SyncAll()`, `rm_Cache_SyncRegion()`, and `rm_Cache_SyncVA()` (see *rm_Cache_SyncAll()* on page 652, *rm_Cache_SyncRegion()* on page 653, and *rm_Cache_SyncVA()* on page 654).
   - In `rm_dcc.s`, modify the implementation of the assembly language macros `RM_DCC_READWORD`, `RM_DCC_WRITEWORD`, and `RM_DCC_GETSTATUS` (see *RM_DCC_READWORD* on page 625, *RM_DCC_WRITEWORD* on page 626, and *RM_DCC_GETSTATUS* on page 627).
   - In `rm_debug.s`, modify the implementation of the assembly language macros `RM_DEBUG_READ_MOE` and `RM_DEBUG_ENABLE` (see *RM_DEBUG_READ_MOE* on page 628 and *RM_DEBUG_ENABLE* on page 629).

- In rm_except.c, modify the implementation of the function rm_GetExceptionTableBase() (see *rm_GetExceptionTableBase()* on page 637).
- In the files rm_processor.h and rm_processor.s, modify the following configuration options:
  - *RM_PROCESSOR_NAME* on page 423
  - *RM_PROCESSOR_REVISION* on page 423
  - *RM_PROCESSOR_HAS_EICE_RT* on page 423
  - *RM_PROCESSOR_HAS_EICE_10* on page 423
  - *RM_PROCESSOR_USE_SYNCCACHES* on page 424
  - *RM_PROCESSOR_USE_SYNC_BY_VA* on page 424
  - *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424.

3. By default, AXD does not display any coprocessor registers relating to your new processor. If you want AXD to display these registers, you must do some additional work, depending on your preference:

   Automatic processor selection

   AXD can automatically select the appropriate processor description upon connection to your target. In this case, coprocessor registers of your processor are displayed in the Registers window when connected. To enable automatic selection, you must:

   1. Add a new description to the file armperip.xml if your processor is not already described within that file. This file is located in the Bin directory of your ADS installation, and can be edited using your preferred text editor. For an example of how to create a new processor description, refer to any of the ADS-supplied processor descriptions that exist within the file.
   2. Update the macros RM_PROCESSOR_NAME and RM_PROCESSOR_REVISION in the file rm_sdm.c to include SDM descriptions of your new processor (see *RMTarget source files* on page 27, *RM_PROCESSOR_NAME* on page 423, and *RM_PROCESSOR_REVISION* on page 423).
   3. Enable the build option RM_OPT_USE_SDM_INFO (see *RM_OPT_SDM_INFO* on page 426).

Note

When you configure AXD to use RMHost, you must also ensure that the option **Fetch module information from target** is selected (the default) in the RealMonitor Configuration dialog box. For details, see the procedure for connecting to RMHost in the *ARM RMHost User Guide*.

Manual processor selection

You can select processor descriptions in AXD at connection time. In this case, coprocessor registers of your chosen processor are displayed in the Registers window when connected. To enable manual selection, you must:

1. Add a new description to the file `armperip.xml` if your processor is not already described within that file. This file is located in the `Bin` directory of your ADS installation, and can be edited using your preferred text editor. For an example of how to create a new processor description, refer to any of the ADS-supplied processor descriptions that exist within the file.
2. Ensure that the build option `RM_OPT_SDM_INFO` is disabled (the default).

Note

When you configure AXD to use RMHost, you must deselect the option **Fetch module information from target** in the RealMonitor Configuration dialog box. The new processor is included in the **Processor** field drop-down list in the dialog box. For details, see the procedure for connecting to RMHost in the *ARM RMHost User Guide*.

Of the three methods available (automatic, manual, or none), you must make sure you use the same method if you are also porting to a new board.

4. Add a new build target for each board with which you want to use the new processor (see *Adding a new build target* on page 215).

2.4.2 Porting to a new board

To port RMTarget to a new board:

1. Create a new directory `Boards\`*`board_name`*, where *`board_name`* is the name of the new board.

2. Copy all the source files from an existing *board_name* directory into the *board_name* directory you have created (see *RMTarget source files* on page 27). To minimize the amount of work you have to perform, it is recommended that you copy the files from a board that is most similar to the one to which you are attempting to port.

   Make the following modifications:

   - In `rm_comms.s`, modify the following assembly language macros:
     - `RM_IRQ_GETSTATUS`
     - `RM_IRQ_ENABLE`
     - `RM_IRQ_DISABLE`.

     (See *RM_IRQ_GETSTATUS* on page 624, *RM_IRQ_ENABLE* on page 624, and *RM_IRQ_DISABLE* on page 625.)
   - In `rm_memory_map.c`, modify the `rm_MemoryMap` data structure to contain memory map descriptions for your new board (see *rm_MemoryMap* on page 622).

3. By default, AXD does not display any memory-mapped registers relating to your new board. If you want AXD to display these registers, you must do some additional work, depending on your preference:

   Automatic board selection

   AXD can automatically select the appropriate board description upon connection to your target. In this case, memory mapped registers of your board are displayed in the Registers window when connected. To enable automatic selection, you must:

   1. Add a description of your board to the file `armperip.xml`, which is located in the `Bin` directory of your ADS installation. You can edit this file using your preferred text editor. For an example of how to create a new board description, refer to any of the ADS-supplied board descriptions that already exist within the file.
   2. Update the macro `RM_BOARD_NAME` in the file `rm_sdm.c` to include SDM descriptions of your new processor (see *RMTarget source files* on page 27).
   3. Enable the build option `RM_OPT_USE_SDM_INFO` (see *RM_OPT_SDM_INFO* on page 426).

Note

When you configure AXD to use RMHost, you must also ensure that the option **Fetch module information from target** is selected (the default) in the RealMonitor Configuration dialog box. For details, see the procedure for connecting to RMHost in the *ARM RMHost User Guide*.

Manual board selection

You can select a board description in AXD at connection time. In this case, memory-mapped registers of the chosen board are displayed in the Registers window when connected. To enable manual selection, you must:

1. Add a description of your board to the file `armperip.xml`, which is located in the `Bin` directory of your ADS installation. You can edit this file using your preferred text editor. For an example of how to create a new board description, refer to any of the ADS-supplied board descriptions that already exist within the file.
2. Ensure that the build option `RM_OPT_SDM_INFO` is disabled (the default).

Note

When you configure AXD to use RHMost, you must deselect the option **Fetch module information from target** in the RealMonitor Configuration dialog box. The new board is included in the **Board** field drop-down list in the dialog box. For details, see the procedure for connecting to RMHost in the *ARM RMHost User Guide*.

Of the three methods available (automatic, manual, or none), you must make sure you use the same method if you are also porting to a new processor.

4. Add a new build target for each processor with which you want to use the new board (see *Adding a new build target*).

2.4.3 Adding a new build target

You must create a new build target whenever you add a new processor or board. To create a new build target:

1. Add a new build directory named `Build\`*`target`*`.b` for each board/processor variant you want to create.

2. If you want to use a makefile to build the RMTarget library for your new target, you must:
   a. Create a new directory named `Build\target.b\standalone`.
   b. Copy the files `board.in` and `makefile` from an existing `Build\target.b` directory into your new target directory (see *RMTarget source files* on page 27).

      In the file `board.in`, modify the following makefile variables:
      - `RM_PLATFORM`
      - `RM_BOARD`
      - `RM_PROCESSOR`
      - `PROCESSOR_NAME`
      - `UHAL_PROCESSOR_ARCHITECTURE`.

   c. Add the new target name to the `boards.in` and `environ.in` files.
3. If you want to use CodeWarrior IDE to build the RMTarget library for your new processor, you must:
   a. Copy the CodeWarrior IDE project file `rm.mcp` from an existing target directory into your new target directory.
   b. Using CodeWarrior IDE, load the new copy of the `rm.mcp` file.
   c. Open the Target window by selecting **rm Settings** from the **Edit** menu.
   d. Select the **Access Paths** window, and change the following paths to correspond to your new processor/board:
      - `{Project}..\..\Boards\board_name`
      - `{Project}..\..\Processors\processor_name`
      - `{Project}..\..\..\uHAL\Boards\board_name`.
   e. Select the Target tab in the ARM Assembler window, and select the name of your processor from the the Architecture or Processor drop-down list.
   f. Select the Target and Source tab in the ARM C Compiler pane, and select the name of your processor from the the Architecture or Processor drop-down list.
   g. Click **Save** and close the window.

Chapter 3
RealMonitor Demonstrations

This chapter describes the RealMonitor demonstrations that are provided when you install AFS 1.2. It contains a step-by-step procedure for setting up the demonstration applications, and provides test scripts for debugging them using AXD.

This chapter contains the following sections:

3.1 About the RealMonitor demonstrations

Demonstrations are provided with the AFS 1.2 source code. These are designed to show how RMTarget can be integrated into an application. It is strongly suggested that you run these demonstrations before attempting to integrate RMTarget into any of your own applications. Before you can use the demonstrations, you must build them (see *Rebuilding the RealMonitor demonstrations* on page 39). The build process creates four excecutable images, one for each demonstration:

`RM.axf` An application that initializes RM, and enters an endless loop (see *Non-mHAL RealMonitor application demonstration* on page 314).

`RM_uHAL.axf`

The µHAL version of the `RM.axf` application (see *mHAL RealMonitor application demonstration* on page 319).

`LEDs.axf` The LEDs demonstration (see *LEDs demonstration* on page 320), which demonstrates how background tasks can continue to run while you are debugging a foreground application.

`LEDs_uHAL.axf`

The µHAL version of the `LEDs.axf` application (see *LEDs demonstration* on page 320).

Note

All demonstrations are designed to be used with the Integrator board, but as with all RealMonitor applications, you can port the demonstrations to a new processor or board (see *Porting RMTarget* on page 211).

The non-µHAL applications demonstrate the process of:

- initializing stacks
- configuring hardware
- sharing interrupts between RealMonitor and an application.

For complete details on sharing interrupts, see *Handling exceptions* on page 55.

In the µHAL versions of the demonstrations, most of this installation is handled automatically by both the µHAL boot code and the function `rm_uHAL_Init()` (see *RM_uHAL_Init()* on page 633).

Optionally, you can build the demonstrations to use RealMonitor in polled mode, so that RealMonitor is not interrupt-driven. In some applications, this is useful if you do not want RealMonitor to affect your interrupt latency (see *Interrupt latency* on page 510 for details).

Though this chapter provides AXD-specific instructions, similar instructions apply to third-party debuggers. Refer to the documentation that accompanies your debugger for details on performing the debugging tasks described in this chapter.

3.2 Directory structure of the RealMonitor demonstrations

When you install AFS 1.2, the RealMonitor demonstration files are provided in source form in the directory `RealMonitor\Demos`, as shown in Figure 3-1.

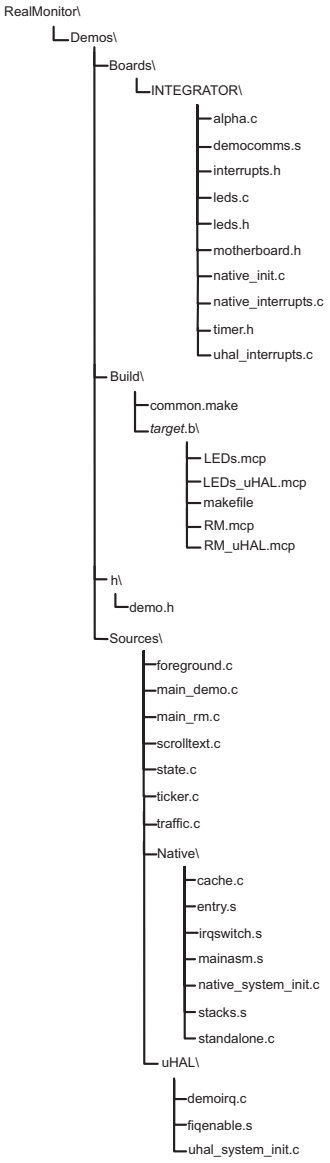

Figure 3-1 RealMonitor demonstrations source files

The RMTarget source code for the demonstrations is contained in the following directories:

`Demos\Boards\INTEGRATOR`

Integrator-specific files that contain definitions for hardware addresses. The demonstrations rely on the these definitions.

`Demos\Build`

Build directory for all supported target platforms. Platform-specific files are located in their relevant *`target`*`.b` subdirectories, such as `Demos\Build\Integrator966E-S.b`.

`Demos\h` Contains the header file for the LEDs demonstrations.

`Demos\Sources`

Shared source code for all RealMonitor demonstrations, including the demonstration tasks and exception handlers.

`Demos\Sources\Native`

Initialization code for the non-µHAL demonstrations.

`Demos\Sources\uHAL`

Initialization code for the µHAL demonstrations.

The RealMonitor demonstration source files are as follows:

`Boards\INTEGRATOR\alpha.c`

Driver for the alphanumeric display.

`Boards\INTEGRATOR\democomms.s`

Defines interrupt numbers for the DCC.

`Boards\INTEGRATOR\interrupts.h`

Defines register addresses for the interrupt controller.

`Boards\INTEGRATOR\leds.c`

Driver for the red, yellow, and green LEDs on the Integrator board.

`Boards\INTEGRATOR\leds.h`

Defines the constants `RED`, `YELLOW`, and `GREEN`, which correspond to their respective LED bit values on the Integrator board.

`Boards\INTEGRATOR\motherboard.h`

Definitions for detecting an Integrator motherboard.

`Boards\INTEGRATOR\native_init.c`

Integrator board-specific initialization code to disable interrupts.

`Boards\INTEGRATOR\native_interrupts.c`

Code to initialize timer-driven IRQ and FIQ routines.

`Boards\INTEGRATOR\timer.h`

Definitions to access the PrimeCell timers on the Integrator board.

`Boards\INTEGRATOR\uhal_interrupts.c`

Supporting code to allow µHAL to install a timer-driven FIQ routine.

`Build\common.make`

A GNU makefile containing the rules for building the RealMonitor application demonstrations that are common to all targets.

`Build\`*`target`*`.b\LEDs.mcp`

A CodeWarrior IDE project file for building the non-µHAL LEDs demonstration.

`Build\`*`target`*`.b\LEDs_uHAL.mcp`

A CodeWarrior IDE project file for building the µHAL LEDs demonstration.

`Build\`*`target`*`.b\makefile`

A GNU makefile containing board-specific and processor-specific rules for building the RealMonitor application demonstration.

`Build\`*`target`*`.b\RM.mcp`

A CodeWarrior IDE project file for building the non-µHAL RealMonitor application demonstration.

`Build\`*`target`*`.b\RM_uHAL.mcp`

A CodeWarrior IDE project file for building the µHAL RealMonitor application demonstration.

`h\demo.h` Contains header definitions for the LEDs demonstrations.

`Sources\foreground.c`

Contains code to poll a function that implements a foreground task. In the LEDs demonstration, this function is `ScrollText()`.

`Sources\main_demo.c`

The main code for the µHAL and non-µHAL LEDs demonstrations.

`Sources\main_rm.c`

The main code for the µHAL and non-µHAL RealMonitor application demonstrations.

`Sources\scrolltext.c`

Contains code for the `ScrollText()` task that scrolls a message on the Integrator alphanumeric display in the LEDs demonstration.

`Sources\state.c`

Code to initialize the global `user_state` structure of the LEDs demonstration.

`Sources\ticker.c`

Contains code for the `TimerTicker()` task that is a simple routine to increment a counter in the LEDs demonstration.

`Sources\traffic.c`

Contains code for the `TrafficTicker()` task that cycles the red, yellow, and green LEDs on the Integrator board, as used by the LEDs demonstration.

`Sources\Native\cache.c`

Contains code for cache initialization.

`Sources\Native\entry.s`

Application startup code. This is the first code that is executed.

`Sources\Native\irqswitch.s`

An IRQ handler that allows the application to share IRQs with RealMonitor when the application is started.

`Sources\Native\mainasm.s`

Contains code to enable IRQs and FIQs.

`Sources\Native\native_system_init.c`

System initialization code for the non-µHAL LEDs and RM application demonstrations.

`Sources\Native\stacks.s`

Initializes stacks for all modes used by RealMonitor.

`Sources\Native\standalone.c`

Provides a runtime environment for the C library. For more details, see the section on building an application for a non-semihosted environment in the *ADS Compilers and Libraries Guide*.

`Sources\uHAL\demoirq.c`

Routines to install and handle timer IRQs.

`Sources\uHAL\fiqenable.s`

Routine to enable FIQs.

`Sources\uHAL\uhal_system_init.c`

System initialization code for the µHAL LEDs and RM application demonstrations.

3.3 Rebuilding the RealMonitor demonstrations

Before you can run the RealMonitor demonstrations, you must first build them. To build the demonstrations:

1. Build the RMTarget library using your preferred method of building (see either *Building RMTarget using CodeWarrior IDE* on page 42 or *Building RMTarget using the makefile* on page 47). Both µHAL and the chaining library are automatically built when the RMTarget library is successfully built.

2. Build a target-specific demonstration in one of the following ways:
   - Run `make` from the directory `Demos\Build\`*`target`*`.b`, where *`target`* is the target platform you are using, such as `Integrator966E-S`. This creates the images in the directory `Demos\Build\`*`target`*`.b\standalone`.
   - Open the CodeWarrior IDE project file in `Demos\Build\`*`target`*`.b`, where *`target`* is the target platform you are using, such as `Integrator966E-S`. This creates the images in the directory `Demos\Build\`*`target`*`.b\standalone_data`.

To build polled versions of the demonstrations, you must rebuild the RMTarget library and the demonstrations using `RM_OPT_USE_INTERRUPTS=FALSE` (see *RM_OPT_USE_INTERRUPTS* on page 427). Depending on the method you are using for building, you can do this in either of the following ways:

- using CodeWarrior IDE (see *Using the Predefines tab of the Assembler window* on page 43 and *Using the Preprocessor tab of the ARM C Compiler window* on page 45)

  Note

  Whenever you specify build options using these two windows in CodeWarrior IDE, you must be sure to apply the same changes to both the `.mcp` demonstration project file (such as `LEDs.mcp`) and the corresponding RMTarget library project file, `rm.mcp` (see *Directory structure of the RealMonitor demonstrations* on page 34 and *RMTarget source files* on page 27).

- using `make` *`options`* on the command line (see *Building using the Makefile* on page 47).

3.4 Initialization of the RealMonitor demonstrations

The following system initialization code is contained in the file `native_system_init.c` (see *Directory structure of the RealMonitor demonstrations* on page 34). It is used by the non-µHAL versions of both the RealMonitor application and LEDs demonstrations. For code specific to µHAL initialization, see *mHAL-specific initialization* on page 313.

```
void SystemInit(void)
{
    SetupCaches();

    /* Disable all interrupt sources. */
    InitBoard();

    /* Initialize stacks for all processor modes RM requires. */
    SetupStacks();

    /* Claim all RM exception vectors. */
    RM_InitVectors();

    /* Initialize RM state. */
    RM_Init();

#if RM_OPT_USE_INTERRUPTS==TRUE
    EnableInterrupts();
#endif
}
```

The non-µHAL versions of the RealMonitor demonstrations call this code at the start of their `main()` function.

The following sections describe each of these function calls or implementation details, and the alternative coding required for µHAL-specific initialization:

- *Caches*
- *Interrupt controller initialization* on page 311
- *Stack initialization* on page 311
- *RealMonitor exception handlers installation* on page 312
- *RMTarget library initialization* on page 312
- *mHAL-specific initialization* on page 313.

3.4.1 Caches

RealMonitor supports debugging on cached processors. You can enable the caches by calling the function `SetupCaches()` in `cache.c`. Details of how caches have been initialized in these demonstrations are not given in this guide because the method varies depending on the exact board and processor you are using.

3.4.2 Interrupt controller initialization

To use RealMonitor in an interrupt-driven manner, you must initialize your interrupt controller. This is performed by the `InitBoard()` function. This function call is not described further in this guide because the code varies depending on the exact board and processor you are using. Some boards might require additional initialization code for components other than the interrupt controller.

3.4.3 Stack initialization

For this demonstration, a distinct fixed-sized stack has been initialized for each processor mode. Each stack is 256 bytes in size, though this value is arbitrary:

```
        AREA    Area_Stacks, DATA, READWRITE
        %       256
UndefStackTop
        %       256
AbortStackTop
        %       256
IRQStackTop
        %       256
FIQStackTop
        %       256
SVCStackTop
```

Each of the processor modes must then be iterated through, and the stack pointer (r13) must be set to contain the address of the top of the appropriate block of memory. The top (highest address) of the block must be used as the starting address of the stacks because the stacks grow downwards as data is pushed onto them:

```
        AREA    Area_SetupStatcks, CODE, READONLY
        EXPORT  SetupStacks
SetupStacks
        MRS     r0, CPSR

        ; Initialise the Undef mode stack.
        BIC     r1, r0, #0x1f
        ORR     r1, r1, #0x1b
        MSR     CPSR_c, r1
        LDR     r13, =UndefStackTop

        ; Initialise the Abort mode stack.
        BIC     r1, r0, #0x1f
        ORR     r1, r1, #0x17
        MSR     CPSR_c, r1
        LDR     r13, =AbortStackTop

        ; Initialise the IRQ mode stack.
```

```
BIC     r1, r0, #0x1f
ORR     r1, r1, #0x12
MSR     CPSR_c, r1
LDR     r13, =IRQStackTop

; Initialise the FIQ mode stack.
BIC     r1, r0, #0x1f
ORR     r1, r1, #0x11
MSR     CPSR_c, r1
LDR     r13, =FIQStackTop

; Initialise the SVC mode stack.
BIC     r1, r0, #0x1f
ORR     r1, r1, #0x13
MSR     CPSR_c, r1
LDR     r13, =SVCStackTop

; Return to the original mode.
MSR     CPSR_c, r0

MOV     pc, lr
```

3.4.4 RealMonitor exception handlers installation

To function properly, RealMonitor must be able to install its own exception handlers. The non-µHAL RealMonitor application, `RM.axf`, has no exception handlers of its own, so it calls the `RM_InitVectors()` function to install the RealMonitor default exception handlers (see *Handling exceptions* on page 55).

3.4.5 RMTarget library initialization

Before you perform any debugging, the RMTarget library must be initialized by calling the function `RM_Init()`.

3.4.6 μHAL-specific initialization

μHAL initialization code performs all necessary target initialization before calling the `main()` function of a μHAL application. In this case, there is no need for the application to initialize stacks, caches, or interrupt controllers. The `SystemInit()` function in the file `uhal_system_init.c` is implemented as:

```
void SystemInit(void)
{
  /* Initialize RM to work with uHAL.
   * This installs handlers for various processor exceptions,
   * calls IMP_Init(), and enables processor interrupts (unless
   * RM is compiled to work in polled mode). */
  RM_uHAL_Init();
}
```

The function `RM_uHAL_Init()` performs all the work necessary to initialize RealMonitor to work with μHAL (see *RM_uHAL_Init()* on page 633).

3.5 Non-μHAL RealMonitor application demonstration

This section describes the non-μHAL RealMonitor application, and provides instructions on how to run it. For initialization details on the non-μHAL RealMonitor demontrations, see *Initialization of the RealMonitor demonstrations* on page 310.

For details on the μHAL version of this demonstration, see *mHAL RealMonitor application demonstration* on page 319.

Caution

You can debug a non RealMonitor application using these demonstration programs only if you have built the demonstration programs to be interrupt-driven (the default). You must not attempt to load any application when using a polled demonstration program.

This application is set to load at a nonstandard address (`0x4000` instead of `0x8000`) so that you can load other simple applications, such as `dhryansi`, without overwriting RealMonitor. This allows you to debug simple applications using RMHost, without having to integrate and link RMTarget with your own application. When you use RMHost to download your application to the target, you are presented with the choice of disabling IRQs (see the description on other messages from RMHost in the Debugging chapter of the *ARM RMHost User Guide*).

Note

A simple application, for the purposes of this documentation, is one that does not have its own exception handlers. If an application installs its own exception handlers, these handlers overwrite the RealMonitor exception handlers. In this case, you must integrate the exception handlers of RMTarget with those in your application (see *Handling exceptions* on page 55).

Alternatively, you can enable chaining support, which allows non-simple applications to co-exist with the μHAL RealMonitor application demonstration (see *Using the AFS chaining library* on page 513).

You can use the RealMonitor application demonstration with any board and processor supported by the RMTarget library, as described in *RealMonitor system requirements* on page 18.

3.5.1 Procedure for running the RealMonitor application demonstration

This section describes the procedure for running the RealMonitor application demonstration.

Although complete information on debugging your own RealMonitor-integrated application using AXD is provided in the *ARM RMHost User Guide*, this section describes all the steps, including the AXD-specific steps you must follow to run this demonstration.

To run the RealMonitor application demonstration:

1. Install RealMonitor, which contains the files necessary to build the RealMonitor application demonstration. See *Directory structure of the RealMonitor demonstrations* on page 34 for a description of these files, and to see where they are installed in relation to the RMTarget source file tree.

2. Connect your JTAG unit, such as Multi-ICE 2.0, to your host computer and target board. If you are using Multi-ICE, see the description of connecting the Multi-ICE hardware in the *Getting Started* chapter of the *Multi-ICE User Guide*. For details on setting up other JTAG units, see the documentation that accompanies your JTAG unit.

   Caution

   If you are using the Multi-ICE sever with RMHost, you must ensure that it is not autoconfigured because this causes the target board to be reset, and disrupts any running program.

   When the Multi-ICE server is not preloaded and preconfigured, you might be given the option to restart the Multi-ICE server when you attempt to connect to the Multi-ICE DLL (as described in steps 5 and 9). In this case, you must click **No** because this would cause an autoconfiguration. Similarly, you must not configure the Multi-ICE server using the **Auto-configure** option.

   To configure the Multi-ICE server for use with RMHost, you must create a configuration file that can be loaded into the Multi-ICE server, as follows:

   a. Start the Multi-ICE server, and select **Auto-Configure** from the **File** menu. The file `Autoconf.cfg` is created (or updated) in the root directory of your Multi-ICE installation.

   b. Make a copy of the file `Autoconf.cfg`, and give the copy an arbitrary name. Renaming this file ensures that it is not overwritten in a future Multi-ICE session.

After you have created and renamed your configuration file, you must load it into your Multi-ICE server using the **Load Configuration...** option from the **File** menu. You must do this every time you load the Multi-ICE server for use with RMHost.

---

3. Build the RealMonitor application demonstration, as described in *Rebuilding the RealMonitor demonstrations* on page 39.

4. Build the semihosted `dhryansi` application as follows:
   a. Select **Open** from the **File** menu. Select the file `dhryansi.mcp` from the `Examples\dhryansi` directory of the ADS installation.
   b. Select **Make** from the **Project** menu. The `dhryansi` project is built and linked.
   c. Exit CodeWarrior IDE.

5. Start AXD, and configure the target as follows:
   a. Select **Configure Target** from the **Options** menu.
   b. Select the DLL for the JTAG unit you are using, such as `Multi-ICE.dll` 2.0, in the Target Environment window. If this is not present in the list, click **Add**, and find the required DLL. (If DLL files do not appear, use Windows Explorer to ensure that files of extension `.dll` are not hidden from view.) If you are using Multi-ICE, the DLL is in the root directory of your Multi-ICE installation.

   Note

   If you have not used the Multi-ICE target in the past, you must configure it after you select it. For details on how to do this, see the *Using the Multi-ICE Server* chapter of the *Multi-ICE User Guide*.

---

6. Load the RealMonitor application image onto your target board by selecting **Load Image** from the **File** menu. Use the file `RM.axf` to run the non-µHAL version. For the µHAL version, use the file `RM_uHAL.axf`. These images are located in one of the following directories, depending on the method you used to build the RealMonitor demonstrations:

   CodeWarrior

   `Demos\Build\`*`target`*`.b\standalone_data`, where *`target`* is the target platform you are using, such as `Integrator966E-S`.

   Makefile

   `Demos\Build\`*`target`*`.b\standalone`, where *`target`* is the target platform you are using, such as `Integrator966E-S`.

Note

If **Load Image** is disabled, you must stop the currently running application by selecting **Stop** from the **Execute** menu.

7. Depending on the target hardware you are using, you might need to change the `$top_of_memory` variable value to correspond to the top-of-memory address of your hardware. For example, if you are using an unexpanded Integrator with a CM966E-S core module, a value of `0x080000` is suitable. See the documentation that accompanies your target hardware for details on the appropriate value to set.

8. Execute the image by selecting **Go** from the **Execute** menu. Configure the interface as follows:
   a. Select **Configure Interface** from the **Options** menu.
   b. Select the General tab.
   c. In the **Target connection** drop-down menu, select the option `ATTACH: Connect according to target properties`. Click **OK**.

9. Configure the target for use with RMHost as follows:
   a. Select **Configure Target** from the **Options** menu.
   b. Select `RealMonitor.dll` as the target DLL. If this DLL is not present in the list, click **Add** and select it from the `Bin` directory of your ADS installation. (If DLL files do not appear, use Windows Explorer to ensure that files of extension `.dll` are not hidden from view.)
   c. Click **Configure** and ensure that a supported JTAG unit, such as Multi-ICE 2.0, appears in the JTAG Controller field. If it does not, click **...** and find the desired DLL. The Multi-ICE DLL is found in the root directory of your Multi-ICE installation.
   d. Click **Configure** to configure your JTAG unit. For details on how to configure Multi-ICE, see the *Using the Multi-ICE Server* chapter of the *Multi-ICE User Guide*. If you are not using Multi-ICE, see the documentation that accompanies your JTAG unit for details on configuration. Click **OK**.

Note

If RMHost detects an error in the configuration you have selected, an error message is displayed. If this happens, you can click one of the following:

OK Stores the configuration, although the error can prevent you from using RMHost.

Cancel Does not store the configuration, and you are returned to the RealMonitor Configuration dialog box.

If configuration is successful, the selected configuration is stored, and the Choose Target dialog box is displayed.

e. Click **OK** to complete the configuration of RMHost.

Caution

When switching targets using AXD, a dialog box is displayed that asks whether you want to stop the currently running program. You must click **No** in this case. AXD also requests whether you want to reload the last image. In this case, you must also click **No** because this would corrupt the currently running image.

f. Select **Stop** from the **Execute** menu. The foreground task in the RealMonitor application demonstration stops, but RealMonitor continues to process interrupts in the background.

10. Load the file `dhryansi.axf` from the ADS installation directory `Examples\dhryansi\dhryansi_Data\DebugRel` onto your target board by selecting **Load Image** from the **File** menu.

Note

A dialog box might be displayed that asks whether you want to disable FIQs and IRQs. If this dialog box is displayed, click **No** to indicate that you do not want these disabled.

Both the RMTarget library and the `dhryansi` program are loaded into memory at the same time.

Execute the image by selecting **Go** from the **Execute** menu. The program runs until the `main()` function is reached. RealMonitor continues to process interrupts.

11. Select **Go** from the **Execute** menu. In the console window, you are prompted to enter the number of runs through the benchmark. Enter a suitable value, such as `50000`.

The `dhryansi` program runs to completion. Results are printed to the console. This demonstrates that RealMonitor allows semihosting operations to be processed.

3.6 μHAL RealMonitor application demonstration

This section describes the μHAL version of the RealMonitor application demonstration. For details on the non-μHAL version of this demonstration, see *Non-mHAL RealMonitor application demonstration* on page 314.

For initialization details on the μHAL RealMonitor demontrations, see *mHAL-specific initialization* on page 313.

When you build the RealMonitor demonstrations, the file `RM_uHAL.axf` is created. The procedure for running the μHAL version of the RealMonitor application is the same as for running the non-μHAL version, except you load `RM_uHAL.axf` instead of `RM.axf`. See *Procedure for running the RealMonitor application demonstration* on page 315 for complete details.

The code in `Sources\Demos\uHAL` implements a μHAL version of the RealMonitor application demonstration. Figure 3-1 on page 34 shows the source code structure for the μHAL subdirectory.

For details on using μHAL with your own application, see *Linking with mHAL* on page 512.

Enabling chaining support allows other applications to add their own exception handlers to RealMonitor exception handlers at runtime. You can use RealMonitor to load other μHAL-based applications, even if they claim interrupts for themselves, as follows:

- When building RMTarget, set `CHAIN_VECTORS=TRUE` (see *CHAIN_VECTORS* on page 427).
- Rebuild μHAL, setting `CHAIN_VECTORS=TRUE`. For details, see the description of rebuilding the μHAL library in the μHAL API chapter of the *ARM Firmware Suite Reference Guide*.

For more details on RealMonitor support for chaining, see *Using the AFS chaining library* on page 513. For complete details on the chaining library, see the chaining library chapter in the *ARM Firmware Suite Reference Guide*.

3.7 LEDs demonstration

This section describes the LEDs demonstrations. It provides a debugging script that introduces you to some of the RMHost debugging capabilities you will typically use with your own application. The two implementations, µHAL and non-µHAL, demonstrate identical functionality. The distinction between the versions is described in *About the RealMonitor demonstrations* on page 32.

For initialization details on all RealMonitor demontrations, see *Initialization of the RealMonitor demonstrations* on page 310.

Details of this demonstration are provided in the following sections:

- *About the LEDs demonstration*
- *Implementation* on page 321
- *Sharing interrupts* on page 322
- *Procedure for running the LEDs demonstration* on page 323
- *Debugging test script for the LEDs demonstration* on page 326.

3.7.1 About the LEDs demonstration

This demonstration shows how a foreground task can be debugged using RMHost while another task continues to execute in the background. The foreground task executes in an endless loop, scrolling text across the two 15-segment alphanumeric LEDs on an Integrator board. The background task is driven by a timer-driven FIQ, and cycles the three colored LEDs on an Integrator board.

The global variable `user_state` defines several runtime parameters, including:

`idle_count` Controls the speed at which text is scrolled. Lower values result in a faster speed.

`invert_text` Determines the display orientation of the scrolled text:

- `0` Normal text orientation.
- `1` Opposite text orientation.

`country` The three colored LEDs on the Integrator board are cycled by the demonstration in one of the following *traffic light* sequences:

- `0` UK mode. The sequence green, yellow, red, and red with yellow is repeated.
- `1` US mode. The sequence green, yellow, and red is repeated.

This demonstration shows how a breakpoint can be reached, and how a value in memory (`country`, in this case) can be modified, while the application continues to run.

3.7.2 Implementation

The µHAL version of the LEDs demonstration shares some code with the non-µHAL LEDs demonstration. It is therefore recommended you read this section regardless of whether you are using the µHAL or non-µHAL version of this demonstration.

Initialization of the LEDs demonstrations is performed using the same `SystemInit()` function calls as used in the RealMonitor application demonstrations (see *Initialization of the RealMonitor demonstrations* on page 310).

Additionally, the initialization code for LEDs demonstrations does the following:

- installs a timer-driven FIQ handler that calls code to cycle the three colored LEDs
- installs a timer-driven IRQ task that increases a global counter variable
- enables IRQ and FIQ interrupts in the processor.

The application then allows the IRQ and FIQ interrupts to occur in the background, while the foreground task scrolls text on the alphanumeric LEDs.

3.7.3 Sharing interrupts

RealMonitor is normally interrupt-driven. As described in *Handling exceptions* on page 55, if an application wants to use interrupts for its own purposes, it must also allow RealMonitor to continue to claim interrupts. The following code from `irqswitch.s` allows the non-µHAL LEDs demonstration to share interrupts with RealMonitor:

```
        IMPORT  RM_IRQHandler2
        EXPORT  App_IRQDispatch
        ROUT
App_IRQDispatch
        ; Point the lr back to the instruction which was interrupted.
        SUB     lr, lr, #4

        ; Save enough registers to call APCS-compliant C functions.
        STMFD   sp!, {r0-r3, ip, lr}

        ; Use uHAL's platform-specific macro to return a bit-mask
        ; containing a '1' for each active interrupt source.
        READ_INT r0, r1, r2

        ; Check whether anything other than a RealMonitor IRQ occurred:
        ; this gives priority to app interrupts over DCC interrupts.
        BICS    r0, r0, #uHAL_COMMRX_MASK :OR: uHAL_COMMTX_MASK
        BEQ     no_app_irqs

app_IRQHandler
        ; An application-specific interrupt has occurred: pass control
        ; to application code.
        ; Insert application specific code here.
        ; Return to the foreground application.
        LDMFD   sp!, {r0-r3, ip, pc}^

        ; No app irqs occurred, so process RealMonitor traffic.
no_app_irqs
        ; Remove r0-r3 from the stack, leaving 'ip' and the adjusted
        ; 'lr', then call RM's interrupt handler.
        LDMFD   sp!, {r0-r3}
        LDR     ip, =RM_IRQ_BASE
        B       RM_IRQHandler2
```

In the case where there are multiple outstanding IRQs, this code gives priority to the IRQs belonging to the application. The RealMonitor handler is called only after all application IRQs have been serviced.

The µHAL version of the LEDs demonstration relies on the µHAL interrupt dispatcher to share interrupts between the application and RealMonitor.

3.7.4 Procedure for running the LEDs demonstration

This section describes the procedure for installing, building, and debugging the LEDs demonstration.

Although complete information on debugging your own RealMonitor-integrated application using AXD is provided in the *ARM RMHost User Guide*, this section and *Debugging test script for the LEDs demonstration* on page 326 describe all the steps, including AXD-specific steps, you must follow to run this demonstration.

To run the LEDs demonstration:

1. Install RealMonitor, which contains the files necessary to build the LEDs demonstration. See *Directory structure of the RealMonitor demonstrations* on page 34 for a description of these files, and to see where they are installed in relation to the RMTarget source file tree.

2. Connect your JTAG unit, such as Multi-ICE 2.0, to your host computer and target board. If you are using Multi-ICE, see the description of connecting the Multi-ICE hardware in the *Getting Started* chapter of the *Multi-ICE User Guide*. For details on setting up other JTAG units, see the documentation that accompanies your JTAG unit.

   Caution

   If you are using the Multi-ICE sever with RMHost, you must ensure that it is not autoconfigured because this causes the target board to be reset, and disrupts any running program.

   When the Multi-ICE server is not preloaded and preconfigured, you might be given the option to restart the Multi-ICE server when you attempt to connect to the Multi-ICE DLL (as described in steps 4 and 8). In this case, you must click **No** because this would cause an autoconfiguration. Similarly, you must not configure the Multi-ICE server using the **Auto-configure** option.

   To configure the Multi-ICE server for use with RMHost, you must create a configuration file that can be loaded into the Multi-ICE server, as follows:

   a. Start the Multi-ICE server, and select **Auto-Configure** from the **File** menu. The file `Autoconf.cfg` is created (or updated) in the root directory of your Multi-ICE installation.

   b. Make a copy of the file `Autoconf.cfg`, and give the copy an arbitrary name. Renaming this file ensures that it is not overwritten in a future Multi-ICE session.

After you have created and renamed your configuration file, you must load it into your Multi-ICE server using the **Load Configuration...** option from the **File** menu. You must do this every time you load the Multi-ICE server for use with RMHost.

3. Build the LEDs demonstration, as described in *Rebuilding the RealMonitor demonstrations* on page 39.

4. Start AXD, and configure the target as follows:
   a. Select **Configure Target** from the **Options** menu.
   b. Select the DLL for the JTAG unit you are using, such as `Multi-ICE.dll` 2.0, in the Target Environment window. If this is not present in the list, click **Add**, and find the required DLL. (If DLL files do not appear, use Windows Explorer to ensure that files of extension `.dll` are not hidden from view.) If you are using Multi-ICE, the DLL is in the root directory of your Multi-ICE installation.

Note

If you have not used the Multi-ICE target in the past, you must configure it after you select it. For details, see the description of connecting an ARM debugger to the Multi-ICE server in the *Using Multi-ICE* chapter of the *Multi-ICE User Guide*.

5. Load the LEDs image into your target board by selecting **Load Image** from the **File** menu. Use the file `LEDs.axf` to run the non-µHAL version. For the µHAL version, use the file `LEDs_uHAL.axf`. These images are located in one of the following directories, depending on the method you used to build the RealMonitor demonstrations:

   CodeWarrior

   `Demos\Build\`*`target`*`.b\standalone_data`, where *`target`* is the target platform you are using, such as `Integrator966E-S`.

   Makefile

   `Demos\Build\`*`target`*`.b\standalone`, where *`target`* is the target platform you are using, such as `Integrator966E-S`.

Note

If **Load Image** is disabled, you must stop the currently running application by selecting **Stop** from the **Execute** menu.

6. Depending on the target hardware you are using, you might need to change the `$top_of_memory` variable value to correspond to the top-of-memory address of your hardware. For example, if you are using an unexpanded Integrator with a CM966E-S core module, a value of `0x080000` is suitable. See the documentation that accompanies your target hardware for details on the appropriate value to set.

7. Execute the image by selecting **Go** from the **Execute** menu. Configure the interface as follows:
   a. Select **Configure Interface** from the **Options** menu.
   b. Select the General tab.
   c. In the **Target connection** drop-down menu, select the option `ATTACH: Connect according to target properties`. Click **OK**.

8. Configure the target for use with RMHost as follows:
   a. Select **Configure Target** from the **Options** menu.
   b. Select `RealMonitor.dll` as the target DLL. If not present in the list, click **Add** and select it from the `Bin` directory of your ADS installation. (If DLL files do not appear, use Windows Explorer to ensure that files of extension `.dll` are not hidden from view.)
   c. Click **Configure** and ensure that a supported JTAG unit, such as Multi-ICE 2.0, appears in the JTAG Controller field. If it does not, click **...** and find the desired DLL. The Multi-ICE DLL is in the root directory of your Multi-ICE installation.
   d. Click **Configure** to configure your JTAG unit. For details on how to configure Multi-ICE, see the *Using the Multi-ICE Server* chapter of the *Multi-ICE User Guide*. If you are not using Multi-ICE, see the documentation that accompanies your JTAG unit for details on configuration. Click **OK**.

      Note

      If RMHost detects an error in the configuration you have selected, an error message is displayed. If this happens, you can click one of the following:

      OK Stores the configuration, although the error can prevent you from using RMHost.

      Cancel Does not store the configuration, and you are returned to the RealMonitor Configuration dialog box.

      If configuration is successful, the selected configuration is stored, and the Choose Target dialog box is displayed.
   e. Click **OK** to complete the configuration of RMHost.

Caution

When switching targets using AXD, a dialog box is displayed that asks whether you want to stop the currently running program. You must click **No** in this case. AXD also requests whether you want to reload the last image. In this case, you must also click **No** because this would corrupt the currently running image.

9. Use AXD to debug the demonstration applications, following the tasks enumerated in *Debugging test script for the LEDs demonstration*. This script is designed to introduce you to the types of debugging tasks you might perform when debugging your own applications in real-time.

3.7.5 Debugging test script for the LEDs demonstration

You must first configure your system, and install and build the LEDs demonstration before you can use AXD to perform debugging. Finally, you must also ensure that the application is executing before you debug it using AXD. See *Procedure for running the LEDs demonstration* on page 323 for complete details.

1. Select **Load Debug Symbols** from the **File** menu, and select the file `LEDs.axf` or `LEDs_uHAL.axf`. The directory in which this file is found depends on the method you used to build the target (see *Procedure for running the LEDs demonstration* on page 323).
2. Restart the application by selecting **Go** from the **Execute** menu. RealMonitor restarts the foreground application.
3. Select **Source** from the **Processor Views** menu. The Open Source window is displayed.
4. Select the file `alpha.c` and click **OK**. The file `alpha.c` opens.
5. In `alpha.c`, locate the following code:

```
void set_alpha(uint32 bits)
{
    /* Wait until hardware is ready for data. */
    do {
       /* nothing */
    } while ((*LED_ALPHA & 1) == 1);

    *LED_ALPHA = bits;
}
```


6. Select the line `*LED_ALPHA = bits;` and press <F9> to set a breakpoint. A red circle appears next to this line in the left-hand margin. The foreground application hits the breakpoint, and stops on that line.

7. In the registers window, change the value of r0 to `0xFFFFFFFF`. Register r0 is used because it stores the variable `bits`.

8. Select **Step** from the **Execute** menu. This single-steps the program, and results in the value `0xFFFFFFFF` being written to the Integrator alphanumeric LEDs. All the alphanumeric LEDs are lit.

9. Remove the breakpoint you set in step 6 by selecting the line `*LED_ALPHA = bits;` and pressing <F9>.

10. Restart the foreground application by selecting **Go** from the **Execute** menu.

11. Select **Variables** from the **Processor Views** menu. The Variables window is displayed.

12. Select the Globals tab in the Variables window, and expand the `user_state` structure by clicking the **+** icon next to this variable. RealMonitor reads the target memory and displays the contents of the `user_state` structure.

13. Select the `country` element of the `user_state` structure. The value is currently `0x00`, which indicates the UK traffic light sequence (see *About the LEDs demonstration* on page 320). Change this value to `0x01`. The new value is written to memory while the foreground application is running, and the traffic light LEDs begin to cycle in the US sequence.

14. The `user_state` structure contains several values that are updated as the program runs. For example, the `alpha_bits` variable is updated as text scrolls in the foreground. You can read the current values at any time by right-clicking in the variables window, and selecting **Refresh** from the pop-up menu to update the display. All values that have changed from the the previous **Refresh** are displayed in red.

Chapter 4
Building RMTarget

This chapter describes how to build RMTarget, and provides a description of the available build options. It contains the following sections:

4.1 Building RMTarget using CodeWarrior IDE

To produce an RealMonitor-enabled executable image, you must:

1. Build the RMTarget library using CodeWarrior IDE.

   Note

   You can also build the RMtarget library using the makefile. See *Building RMTarget using the makefile* on page 47 for complete details.

2. Integrate RMTarget with your application or RTOS (see Chapter 5 *Integration*).
3. Link the RMTarget library with your application or RTOS.

The RMTarget build options provide full control over which features you include in any given build.

This section describes how to build the RMTarget library using CodeWarrior IDE, and how to link the newly built library with your application. It contains the following sections:

- *Building using CodeWarrior IDE*
- *Linking using CodeWarrior IDE* on page 46.

Note

You can also build the RMTarget library using the supplied GNU makefile. See *Building RMTarget using the makefile* on page 47 for complete details.

4.1.1 Building using CodeWarrior IDE

A CodeWarrior IDE project file is supplied for each board/processor variant of RMTarget. The project files are named `rm.mcp`, and are located in each `Build\`*`target`*`.b` directory of the RMTarget source tree (see Figure 2-1 on page 27). For more information about using CodeWarrior IDE, see the *CodeWarrior IDE Guide* in the online suite of books supplied with ADS.

To build the RMTarget library using CodeWarrior IDE:

1. Load the `rm.mcp` project file applicable to your target system into CodeWarrior IDE. To do this, double-click the `rm.mcp` icon in the `Demos\standalone\Build\`*`target`*`.b` directory, where *`target`* is the name of your chosen board/processor variant. CodeWarrior IDE starts and loads the project file.

2. If you want to build the default RMTarget library, proceed to step 3. Alternatively, you can use the build options described in *RMTarget build options and macros* on page 415 to build a customized RMTarget library. To do this, you must edit the CodeWarrior IDE project settings in both of the following ways:
   - *Using the Predefines tab of the Assembler window*
   - *Using the Preprocessor tab of the ARM C Compiler window* on page 45.

   Note

   Ensure that you define each build option in both the ARM Assembler window and the ARM C Compiler window.

3. Build the library by selecting **Make** from the **Project** menu. The build process should complete without error. Any error messages you receive indicate a problem that you must rectify. RealMonitor error messages correspond to individual build options. Each error message is documented with the build option to which it relates (see *RMTarget build options and macros* on page 415).

   The RMTarget library is placed in the `Build\target.b\rm_Data\rm` directory as `rm.a`.

   Note

   You can rebuild the RMTarget library at any time. With CodeWarrior IDE, you do not have to clean the build directories when you change any build options because CodeWarrior IDE automatically rebuilds the necessary files when you select **Make**.

   You cannot use CodeWarrior IDE to determine the size of the RMTarget library. Instead, you can use the `armar` program, as described in *Determining library size* on page 411.

After you have built the RMTarget library, you must integrate RMTarget with your application or RTOS, as described in Chapter 5 *Integration*. After you have fully integrated RMTarget, you must build your own application using your preferred method of building. After you have performed these steps, you must then link the RMTarget library with the files of your own application (see *Linking using CodeWarrior IDE* on page 46).

Using the Predefines tab of the Assembler window

To set build options in this window:

1. Open the Target window by selecting **rm Settings** from the **Edit** menu.

2. Select **ARM Assembler** from the Language Settings category of Target Setting Panels (left side).
3. Select the Predefines tab in the ARM Assembler window, as shown in Figure 4-1.
4. Enter the name of the desired build option, such as `RM_OPT_GETPC`, in the **Variable Name** field.
5. Select the type of definition you require, `SETL` or `SETA`, in the **Directive** field.
6. Set the value of the definition, for example `TRUE`.
7. Click **Add** to ensure that the new definition is added to the command-line arguments. The build option is passed automatically to the ARM assembler, `armasm`, in one of the following formats:
   - `-pr "option_name SETL {option_value}"` for `TRUE` and `FALSE` values
   - `-pr "option_name SETA {option_value}"` for numeric values.
8. Repeat steps 4–7 for all other build options you want to set.

Note

Any build options that do not appear in the ARM Assembler window are included in the build using their default value. To see what the default value is for each build option, see *RMTarget build options and macros* on page 415.

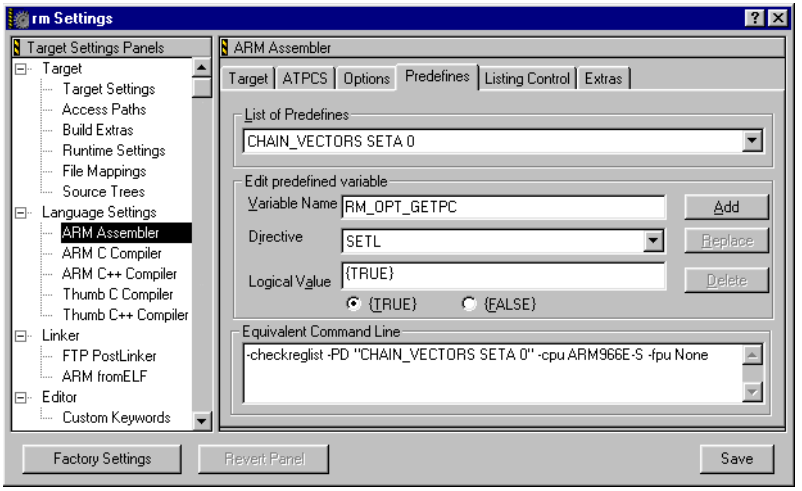

Figure 4-1 Setting build options for the assembler

Using the Preprocessor tab of the ARM C Compiler window

To set build options in this window:

1. Open the Target window by selecting **rm Settings** from the **Edit** menu.
2. Select **ARM C Compiler** from the Language Settings category of Target Setting Panels (left side).
3. Select the Preprocessor tab in the ARM C Compiler window, as shown in Figure 4-2.
4. Enter a build option definition, such as `RM_OPT_GETPC=TRUE`, into the field directly below the **List of #DEFINEs**.
5. Click **Add** to ensure that the new definition is added to the command-line arguments. The build option is passed automatically to the ARM C compiler, `armcc`, in the following format:

   `-Doption_name=option_value`
6. Repeat steps 4–5 for all other build options you want to set.

Note

Any build options that do not appear in the ARM C Compiler window are included in the build using their default value. To see what the default value is for each build option, see *RMTarget build options and macros* on page 415.

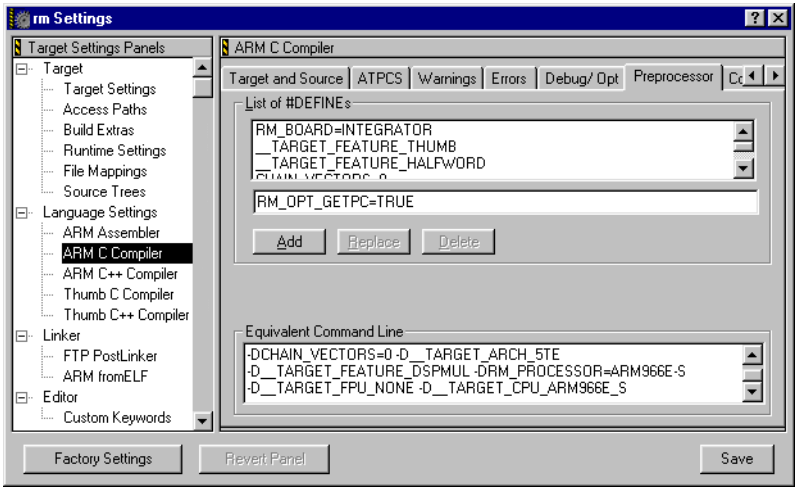

Figure 4-2 Setting build options for the C compiler

4.1.2 Linking using CodeWarrior IDE

Before you link the RMTarget library with the files of your own application, you must first:

1. Build the RMTarget library file `rm.a` (see *Building using CodeWarrior IDE* on page 42).
2. Integrate RMTarget with your application or RTOS (see Chapter 5 *Integration*).
3. Build your own application using your preferred method of building.

To link the RMTarget library with your own application using CodeWarrior IDE, you must create a new CodeWarrior IDE project that comprises the source files and libraries of your own application, plus the RMTarget project file and library. See the description of making a project in the *CodeWarrior IDE Guide* for details on how to make a project and link object files.

Note

After you have successfully created an RM-incorporated project, the RMTarget library builds automatically when you select **make**. This saves you the extra step of having to manually load and build the RMTarget library every time you build your application.

4.2 Building RMTarget using the makefile

This section describes how to build the RMTarget library by setting build options in the RMTarget makefile, or on the command line, and how to link the newly built library with your application.

Note

You must have the GNUmake program to build using the RM-supplied makefiles. For instructions on how to obtain GNUmake, see the description on using GNUmake in the building AFS components chapter of the *ARM Firmware Suite User Guide*.

This section describes the following:

- *Building using the Makefile*
- *Rebuilding the RMTarget library* on page 49
- *Linking using armlink* on page 49
- *Building RMTarget with your application* on page 49.

4.2.1 Building using the Makefile

For build options that you do not have to change frequently, it is recommended that you set them by editing the GNU file `common.make`. This file is located in the `Build` directory in the RMTarget source tree (see Figure 2-1 on page 27). The build options available are described in *RMTarget build options and macros* on page 415.

Note

To reduce the code size of the RMTarget library, you must disable any options in the makefile that you do not intend to use.

You can also include options on the command line when building using the `make` command. This method of settings options is recommended when you want to change options frequently. Any options you set on the command line override any options set in the file `common.make`.

To build RMTarget, go to either the top-level directory, or the directory where the makefile is located, and use the `make` command at the command prompt, for example:

```
cd RealMonitor
make options
```

or

```
cd RealMonitor\Build\Integrator966E-S.b
make options
```

If you perform a `make` in the top-level directory, all board/processor variants of RMTarget are built. If you perform a `make` in a single board/processor directory, only that variant of RMTarget is built.

In either case, *`options`* can be any number of build option settings you choose to include, for example:

```
make RM_OPT_STOPSTART=FALSE RM_OPT_WRITEWORDS=TRUE
```

Note

If you do not include any options on the command line, and you have not edited `common.make`, a library containing the default set of build options is built (see *Default RMTarget settings* on page 429).

If the GNU makefile and the ADS tool chain are installed correctly, the build process completes without error. Any error messages you might receive indicate a problem that you must rectify. RealMonitor error messages correspond to individual build options. Each error message is documented with the build option to which it relates (see *RMTarget build options and macros* on page 415).

After the `make` process is complete, a new version of the library file `rm.a` is created for each board/processor library variant in its relevant `standalone` subdirectory, `Build\`*`BuildProcessor`*`.b\standalone`. Any previously existing versions of `rm.a` are overwritten with the newly built version. After `rm.a` is built, you must link it with the libraries and object files of your own application (see *Linking using armlink* on page 49).

When you build the RMTarget library using this makefile, the `make` scripts automatically pass the build options into the C compiler and ARM assembler, using the respective formats required by these tools. Options passed to the ARM C compiler, `armcc`, take the format:

```
-Doption_name=option_value
```

Options passed to the ARM assembler, `armasm`, take either of the following formats:

- `-pr "`*`option_name`* `SETL {`*`option_value`*`}"` for `TRUE` and `FALSE` values
- `-pr "`*`option_name`* `SETA {`*`option_value`*`}"` for numeric values.

After you have built the RMTarget library, you must integrate RMTarget with your application or RTOS, as described in Chapter 5 *Integration*. After you have fully integrated RMTarget into your application, you must build your own application using your preferred method of building. After you have performed these steps, you must then link the RMTarget library with the files of your own application (see *Linking using armlink* on page 49).

4.2.2 Rebuilding the RMTarget library

You can rebuild the RMTarget library at any time.

Caution

If you do not clean the files and libraries before you rebuild, the RMTarget library can contain a mixture of components using both the old and new build options.

If you intend to use different build options than those used in the previous build, you must first remove the old object and library files by using the `clean` option. To clean these files, use:

```
make clean
```

You can perform this in either the top-level directory, or a single build/processor directory as appropriate.

4.2.3 Linking using armlink

Before you link the RMTarget library with the files of your own application, you must first:

1. Build the RMTarget library file `rm.a` (see *Building using the Makefile* on page 47).
2. Integrate RMTarget with your application or RTOS (see Chapter 5 *Integration*).
3. Build your own application using your preferred method of building.

If you have built `rm.a` using the Makefile, or from the command line, you must link using the `armlink` program. See the description of using armlink in the *ADS Compilers and Libraries Guide*.

For example, using object file *`my_object`*`.o`, create the image *`my_program`*`.axf` using the following command:

```
armlink -O my_program.axf my_object.o rm.a
```

4.2.4 Building RMTarget with your application

You can the build the RMTarget library and your application in one step. To do this, you must add the following lines to the makefile of your own project:

```
cd RealMonitor\Build\target.b; make options
```

where *`options`* can be any number of build option settings you choose to include. For example:

```
make RM_OPT_STOPSTART=FALSE RM_OPT_WRITEWORDS=TRUE
```

The build options available are described in *RMTarget build options and macros* on page 415.

If you include these lines, the RMTarget library is built when you run your makefile. If you include these lines, but do not specify any *options*, the RMTarget library is built using the settings in the RealMonitor makefile `common.make`. For details on editing the RealMonitor makefile, see *Building RMTarget using the makefile* on page 47.

4.3 Determining library size

You can discover the size of an RMTarget library by using the armar program in either a Unix environment, or from an MS-DOS command window. To determine the RMTarget library size, type the following on the command line:

```
armar -sizes rm.a
```

Example 4-1 shows an example of the information that is returned.

Example 4-1 Example of RMTarget library sizes

```
Code    RO Data    RW Data    ZI Data      Debug   Object Name
1436        192          0        304      24532   rm_control.o
1272          0         24          0        800   rm_asm.o
 164          0          0          0      11232   rm_log.o
   0         44          0          0       7484   rm_memorymap.o
   0         36          0          0       3356   rm_sdm.o
 240          0          0          0      12628   rm_vec.o
 200          0        100          0      34760   rm_uhal_init.o
  52          0          0          0        484   rm_poll.o
  84          0          0          0        444   rm_chain_handler.o
3448        272        124        304      95720   TOTAL
```

The sizes shown in Example 4-1 indicate the maximum amount of memory that the RMTarget library will use. In practice, the amount of memory actually used is less than that shown when using armar, because the linker discards sections that are not referenced by any other part of the program.

For a description of the armar program, see the ARM librarian section in the *ADS Compilers and Libraries Guide*.

4.4 Types of build options

This section describes the categories of RMTarget build options available. See *RMTarget build options and macros* on page 415 for a complete list and description of the options available when building RMTarget. The categories of build options are:

- *System configuration build macros*
- *Data logging build option*
- *Debugging build options*
- *Packet support build options* on page 413
- *Automatic build options* on page 413
- *Processor configuration build options* on page 413
- *Miscellaneous build options* on page 414.

4.4.1 System configuration build macros

These options determine the overall behavior of RealMonitor for every debugging session, and in most cases, do not need to be changed from their default settings:

- *RM_BOARD* on page 415
- *RM_PROCESSOR* on page 415
- *RM_BUILD_PATH* on page 415.

Unlike the other build options, you cannot enable or disable them. These options are for the makefile only, and are not used when building with CodeWarrior IDE.

4.4.2 Data logging build option

This option determines whether you can send third-party data between the target and the host:

- *RM_OPT_DATALOGGING* on page 416.

4.4.3 Debugging build options

These options determine the type of debugging you can perform from the host after you have built your RM-enabled executable image:

- *RM_OPT_STOPSTART* on page 416
- *RM_OPT_SOFTBREAKPOINT* on page 417
- *RM_OPT_HARDBREAKPOINT* on page 417
- *RM_OPT_HARDWATCHPOINT* on page 417
- *RM_OPT_SEMIHOSTING* on page 418
- *RM_OPT_SAVE_FIQ_REGISTERS* on page 418.

For details on how to debug the image, see the *ARM RMHost User Guide*.

4.4.4 Packet support build options

These options enable or disable support for certain packets on the RealMonitor channel:

- *RM_OPT_READBYTES* on page 418
- *RM_OPT_WRITEBYTES* on page 419
- *RM_OPT_READHALFWORDS* on page 419
- *RM_OPT_WRITEHALFWORDS* on page 419
- *RM_OPT_READWORDS* on page 419
- *RM_OPT_WRITEWORDS* on page 420
- *RM_OPT_EXECUTECODE* on page 420
- *RM_OPT_GETPC* on page 420.

4.4.5 Automatic build options

These options are set automatically by RM:

- *RM_OPT_COMMANDPROCESSING* on page 421
- *RM_OPT_READ* on page 421
- *RM_OPT_WRITE* on page 421
- *RM_OPT_READWRITE* on page 422
- *RM_OPT_SYNCCACHES* on page 422
- *RM_OPT_SYNC_BY_VA* on page 422.

4.4.6 Processor configuration build options

These configuration options are processor-specific, and are set in the files `rm_processor.h` and `rm_processor.s` (see *RMTarget source files* on page 27). The values of these options are decided when porting RMTarget to a new processor. You do not have to change them for subsequent use:

- *RM_PROCESSOR_NAME* on page 423
- *RM_PROCESSOR_REVISION* on page 423
- *RM_PROCESSOR_HAS_EICE_RT* on page 423
- *RM_PROCESSOR_HAS_EICE_10* on page 423
- *RM_PROCESSOR_USE_SYNCCACHES* on page 424
- *RM_PROCESSOR_USE_SYNC_BY_VA* on page 424
- *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424.

4.4.7 Miscellaneous build options

The following miscellaneous build options are also available:

- *RM_EXECUTECODE_SIZE* on page 425
- *RM_OPT_GATHER_STATISTICS* on page 425
- *RM_DEBUG* on page 426
- *RM_OPT_BUILDIDENTIFIER* on page 426
- *RM_OPT_SDM_INFO* on page 426
- *RM_OPT_MEMORYMAP* on page 426
- *RM_OPT_USE_INTERRUPTS* on page 427
- *RM_FIFOSIZE* on page 427
- *CHAIN_VECTORS* on page 427.

4.5 RMTarget build options and macros

This section describes all RMTarget build options and macros that you can:

- set in the `common.make` file
- pass on the command line when using the the `make` command
- use within CodeWarrior IDE.

See *Building RMTarget using the makefile* on page 47 and *Building RMTarget using CodeWarrior IDE* on page 42 for details on setting options and building RMTarget.

4.5.1 RM_BOARD

RMTarget contains code that is specific to particular boards. This board-specific code is located in the `Boards` directory in the RMTarget source tree (see Figure 2-1 on page 27). If you are using the supplied makefiles and the CodeWarrior IDE project files, this build option is set automatically.

Note

See *RealMonitor system requirements* on page 18 for a list of supported boards.

4.5.2 RM_PROCESSOR

RMTarget contains code that is specific to particular processors. This processor-specific code is located in the `Processors` directory in the RMTarget source tree (see Figure 2-1 on page 27). If you are using the supplied makefiles and the CodeWarrior IDE project files, this build option is set automatically.

Note

See *RealMonitor system requirements* on page 18 for a list of supported processors.

4.5.3 RM_BUILD_PATH

Because of the large number of build variants, conflicts can arise between different users and different projects if all object files are placed in the same directory. You can set this macro to change the directory into which the builds are placed. By default, builds are placed into the `Build` directory of the RMTarget source tree (see Figure 2-1 on page 27).

Note

You can only set this option when using the makefile to build the RMTarget library. This cannot be used with CodeWarrior IDE.

4.5.4 RM_OPT_DATALOGGING

This option enables or disables support for any target-to-host packets sent on a non RealMonitor (third-party) channel.

Note

If this option is disabled, `RM_OPT_COMMANDPROCESSING` must be enabled (see *RM_OPT_COMMANDPROCESSING* on page 421). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
Neither RM_OPT_DATALOGGING nor RM_OPT_COMMANDPROCESSING are enabled!
```

Values

`FALSE`	Disabled (default).
`TRUE`	Enabled.

4.5.5 RM_OPT_STOPSTART

This option enables or disables support for all stop and start debugging features. This includes support for the RealMonitor Stop and Go packets (see *Stop* on page 714 and *Go* on page 714).

Note

If this option is enabled, `RM_OPT_COMMANDPROCESSING` must also be enabled (see *RM_OPT_COMMANDPROCESSING* on page 421). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_STARTSTOP requires RM_OPT_COMMANDPROCESSING!
```

Values

`FALSE`	Disabled.
`TRUE`	Enabled (default).

4.5.6 RM_OPT_SOFTBREAKPOINT

This option enables or disables support for software breakpoints.

Note

If this option is enabled, `RM_OPT_STOPSTART` must also be enabled (see *RM_OPT_STOPSTART* on page 416). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_SOFTBREAKPOINT requires RM_OPT_STOPSTART!
```

Values

`FALSE`	Disabled.
`TRUE`	Enabled (default).

4.5.7 RM_OPT_HARDBREAKPOINT

This option enables or disables support for nonstop hardware breakpoints. Nonstop hardware breakpoints are available only on processors that contain EmbeddedICE-RT logic.

Note

If this option is enabled, `RM_OPT_STOPSTART` must also be enabled (see *RM_OPT_STOPSTART* on page 416). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_HARDBREAKPOINT requires RM_OPT_STOPSTART!
```

Values

`FALSE`	Disabled (default if the processor does not contain EmbeddedICE-RT logic).
`TRUE`	Enabled (default if the processor contains EmbeddedICE-RT logic).

4.5.8 RM_OPT_HARDWATCHPOINT

This option enables or disables support for nonstop watchpoints. Nonstop watchpoints are available only on processors that contain EmbeddedICE-RT logic.

Note

If this option is enabled, RM_OPT_STOPSTART must also be enabled (see *RM_OPT_STOPSTART* on page 416). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_HARDWATCHPOINT requires RM_OPT_STOPSTART!
```

Values

FALSE	Disabled (default if the processor does not contain EmbeddedICE-RT logic).
TRUE	Enabled (default if the processor contains EmbeddedICE-RT logic).

4.5.9 RM_OPT_SEMIHOSTING

This option enables or disables support for SWI semihosting.

Note

If this option is enabled, RM_OPT_STOPSTART must also be enabled (see *RM_OPT_STOPSTART* on page 416). Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_SEMIHOSTING requires RM_OPT_STOPSTART!
```

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.10 RM_OPT_SAVE_FIQ_REGISTERS

This option determines whether the FIQ-mode registers are saved into the registers block when RealMonitor stops.

Values

FALSE	Do not save FIQ-mode registers.
TRUE	Save FIQ-mode registers (default).

4.5.11 RM_OPT_READBYTES

This option enables or disables support for the RealMonitor ReadBytes packet (see *ReadBytes* on page 718).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.12 RM_OPT_WRITEBYTES

This option enables or disables support for the RealMonitor WriteBytes packet (see *WriteBytes* on page 721).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.13 RM_OPT_READHALFWORDS

This option enables or disables support for the RealMonitor ReadHalfWords packet (see *ReadHalfWords* on page 723).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.14 RM_OPT_WRITEHALFWORDS

This option enables or disables support for the RealMonitor WriteHalfWords packet (see *WriteHalfWords* on page 726).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.15 RM_OPT_READWORDS

This option enables or disables support for the RealMonitor ReadWords packet (see *ReadWords* on page 729).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.16 RM_OPT_WRITEWORDS

This option enables or disables support for the RealMonitor WriteWords packet (see *WriteWords* on page 731).

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.17 RM_OPT_EXECUTECODE

This option enables or disables support for the RealMonitor ExecuteCode packet (see *ExecuteCode* on page 717).

Note

If this option is enabled, one or more of the following build options must also be enabled:

- *RM_OPT_WRITEBYTES* on page 419
- *RM_OPT_WRITEHALFWORDS* on page 419
- *RM_OPT_WRITEWORDS*.

Otherwise, the following error message is returned when you attempt to build the RMTarget library:

```
RM_OPT_EXECUTECODE requires at least one write to memory packet to be supported!
```

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.18 RM_OPT_GETPC

This option enables or disables support for the RealMonitor GetPC packet (see *GetPC* on page 715).

Note

It is recommended that you do not enable this option if you intend to use RealMonitor in polled mode because, in that case, the GetPC opcode always returns the same address (see *Reducing interrupt latency by polling the DCC* on page 512).

Values

FALSE	Disabled (default).
TRUE	Enabled.

4.5.19 RM_OPT_COMMANDPROCESSING

This specifies whether any host-to-target packet processing code must be built into RMTarget.

Caution

This build option is set automatically by RealMonitor. You must not attempt to set this build option manually.

Values

FALSE	No host-to-target packets are supported.
TRUE	One or more host-to-target packets is supported.

4.5.20 RM_OPT_READ

This option specifies whether any of the read memory packets are supported:

- *RM_OPT_READBYTES* on page 418
- *RM_OPT_READHALFWORDS* on page 419
- *RM_OPT_READWORDS* on page 419.

Caution

This build option is set automatically by RealMonitor. You must not attempt to set this build option manually.

Values

FALSE	None of the read memory packets are enabled.
TRUE	One or more of the read memory packets is enabled.

4.5.21 RM_OPT_WRITE

This option specifies whether any of the write memory packets are supported:

- *RM_OPT_WRITEBYTES* on page 419
- *RM_OPT_WRITEHALFWORDS* on page 419
- *RM_OPT_WRITEWORDS* on page 420.

Caution

This build option is set automatically by RealMonitor. You must not attempt to set this build option manually.

Values

FALSE	None of the write memory packets are enabled.
TRUE	One or more of the write memory packets is enabled.

4.5.22 RM_OPT_READWRITE

This option specifies whether any of the read memory or write memory packets (*RM_OPT_READ* on page 421 or *RM_OPT_WRITE* on page 421) is supported.

Caution

This build option is set automatically by RealMonitor. You must not attempt to set this build option manually.

Values

FALSE	None of the read memory and write memory packets is enabled.
TRUE	One or more read memory or write memory packets is enabled.

4.5.23 RM_OPT_SYNCCACHES

This specifies whether the RealMonitor SyncCaches packet is supported (see *SyncCaches* on page 716).

Values

FALSE	The SyncCaches packet is not supported.
TRUE	The SyncCaches opcode is supported.

4.5.24 RM_OPT_SYNC_BY_VA

This specifies whether cache coherency is maintained automatically by the write-to-memory opcodes.

Note

This option is enabled only when both the options RM_OPT_WRITE and RM_PROCESSOR_USE_SYNC_BY_VA are set to TRUE (see *RM_OPT_WRITE* on page 421 and *RM_PROCESSOR_USE_SYNC_BY_VA* on page 424).

Values

FALSE	The write-to-memory opcodes do not maintain cache coherency.
TRUE	The write-to-memory opcodes maintain cache coherency.

4.5.25 RM_PROCESSOR_NAME

This specifies the name of the processor. The name is used in the *Self-Describing Module* (SDM) extension to RDI, provided you have enabled the build option RM_OPT_SDM_INFO (see *RM_OPT_SDM_INFO* on page 426). The name must correspond exactly with a processor name listed in the file armperip.xml, which is located in the Bin directory of your ADS installation.

This macro is defined only in the file rm_processor.h.

4.5.26 RM_PROCESSOR_REVISION

This specifies the revision of the processor, and is used in the SDM extension to RDI.

This macro is defined only in the file rm_processor.h.

4.5.27 RM_PROCESSOR_HAS_EICE_RT

This specifies whether the RealMonitor is being compiled for a processor that contains EmbeddedICE-RT logic.

Values

FALSE	The processor does not contain EmbeddedICE-RT logic.
TRUE	The processor contains EmbeddedICE-RT logic.

4.5.28 RM_PROCESSOR_HAS_EICE_10

This specifies whether RMTarget is being compiled for a processor that contains ARM10-type EmbeddedICE-RT logic. EmbeddedICE-RT logic is of the ARM10 type if the breakpoint and watchpoint registers cannot be accessed directly over the JTAG.

Values

FALSE	The processor does not contain ARM10-type EmbeddedICE-RT logic.
TRUE	The processor contains ARM10-type EmbeddedICE-RT logic.

4.5.29 RM_PROCESSOR_USE_SYNCCACHES

This specifies whether the processor you are using supports the SyncCaches opcode. The SyncCaches opcode is needed only on Harvard-architecture processors, and where cache coherency operations are slow.

This option is enabled only when the functions `rm_Cache_SyncAll()` and `rm_Cache_SyncRegion()` are implemented (see *rm_Cache_SyncAll()* on page 652 and *rm_Cache_SyncRegion()* on page 653).

4.5.30 RM_PROCESSOR_USE_SYNC_BY_VA

This specifies whether the write-to-memory opcodes automatically maintain ICache and DCache coherency on Harvard-architecture processors. This option must be enabled only when:

- the processor supports cache coherency instructions using virtual addresses
- cache coherency operations are fast.

This option is enabled only when the function `rm_Cache_SyncVA()` is implemented (see *rm_Cache_SyncVA()* on page 654).

4.5.31 RM_PROCESSOR_NEEDS_NTRST_FIX

On the ARM966E-S, while the **nTRST** signal is asserted, coprocessor instructions that are used to access the DCC can generate undefined instruction exceptions.

Caution

You must be careful if you need to modify the DCC access code in RMTarget because the `RM_PROCESSOR_NEEDS_NTRST_FIX` build option is specific to RMTarget.

The **nTRST** signal is typically asserted when establishing a connection to an ARM processor using the JTAG unit (such as Multi-ICE). When using the ARM966E-S, you must therefore enable this build option to ensure that RealMonitor continues to work when **nTRST** is asserted.

Enabling this build option allows RealMonitor to detect any failed accesses to the DCC, and consequently stops RealMonitor from attempting to transmit any further data until **nTRST** is de-asserted.

Note

To decrease the code size of the RMTarget library, it is suggested you disable this option if you are not using the ARM966E-S .

Values

FALSE	Disabled.
TRUE	Enabled (default).

4.5.32 RM_EXECUTECODE_SIZE

This option specifies the size, in words, of the buffer that is used for storing code and data for the ExecuteCode packet.

Note

If you set the buffer size to less than zero, the following error message is returned when you attempt to build the RMTarget library:

```
RM_EXECUTECODE_SIZE is set to an invalid value!
```

Values

size_of_buffer

The size, in words, of the ExecuteCode buffer. The default size is `0x06`.

Note

In most cases, you do not have to change the buffer size because the default value is sufficient when using the RMHost controller. For any given processor, there is a minimum execute buffer size. If you try to set the ExecuteCode buffer to less than the minimum value, the build system ignores your setting and chooses the minimum size instead. If the buffer is too big, memory is wasted.

4.5.33 RM_OPT_GATHER_STATISTICS

This option enables or disables the code for gathering statistics about the internal operation of RealMonitor.

Note

This option is intended primarily for internal ARM usage, and must not normally be enabled.

Values

FALSE	Disabled (default).
TRUE	Enabled.

4.5.34 RM_DEBUG

This option enables or disables additional debugging and error-checking code in RealMonitor.

Note

This option is intended primarily for internal ARM usage, and must not normally be enabled.

Values

FALSE	Disabled (default).
TRUE	Enabled.

4.5.35 RM_OPT_BUILDIDENTIFIER

This option determines whether a build identifier is built into the capabilities table of RMTarget (see *rm_CapabilitiesTable* on page 615).

Values

FALSE	Disabled (default).
TRUE	Enabled.

4.5.36 RM_OPT_SDM_INFO

This option determines whether SDM information is built into the capabilities table (see *rm_CapabilitiesTable* on page 615).

Values

FALSE	Do not include SDM information (default).
TRUE	Include SDM information.

4.5.37 RM_OPT_MEMORYMAP

This option determines whether a memory map of the board is built into the target and made available through the capabilities table (see *rm_CapabilitiesTable* on page 615).

Values

FALSE	Do not include a memory map of the board (default).
TRUE	Include a memory map of the board.

4.5.38 RM_OPT_USE_INTERRUPTS

This option specifies whether RMTarget is built for interrupt-driven mode or polled mode. For details on changing your application to use polled mode, see *Reducing interrupt latency by polling the DCC* on page 512.

Note

It is recommended that you do not enable the build option RM_OPT_GETPC if you intend to use RealMonitor in polled mode because, in that case, the GetPC opcode always returns the same address (see *RM_OPT_GETPC* on page 420).

Values

FALSE	No interrupt handling is provided, and you must poll RealMonitor.
TRUE	Interrupt handling is provided (default).

4.5.39 RM_FIFOSIZE

This option specifies the size, in words, of the data logging FIFO buffer.

Note

If you set the FIFO size to less than zero, the following error message is returned when you attempt to build the RMTarget library:

```
RM_FIFOSIZE is set to an invalid value!
```

Values

size_of_buffer

The size, in words, of the FIFO buffer. The default size is 0x10.

4.5.40 CHAIN_VECTORS

This option allows RMTarget to support vector chaining through µHAL.

Values

FALSE	Disable support for vector chaining (default).

TRUE	Enable support for vector chaining.

4.6 Default RMTarget settings

If you do not set any build options, the default set of build options is used when you build the RMTarget library, `rm.a`. This default set of options is also used when you build the demonstration RealMonitor applications provided with ARM RealMonitor. For details on running these examples, see Chapter 3 *RealMonitor Demonstrations*.

The object code for this default configuration is approximately 2KB:

```
RM_OPT_DATALOGGING=FALSE
RM_OPT_STOPSTART=TRUE
RM_OPT_SOFTBREAKPOINT=TRUE
RM_OPT_HARDBREAKPOINT=processor_dependent
RM_OPT_HARDWATCHPOINT=processor_dependent
RM_OPT_SEMIHOSTING=TRUE
RM_OPT_SAVE_FIQ_REGISTERS=TRUE
RM_OPT_READBYTES=TRUE
RM_OPT_WRITEBYTES=TRUE
RM_OPT_READHALFWORDS=TRUE
RM_OPT_WRITEHALFWORDS=TRUE
RM_OPT_READWORDS=TRUE
RM_OPT_WRITEWORDS=TRUE
RM_OPT_EXECUTECODE=TRUE
RM_OPT_GETPC=FALSE
RM_EXECUTECODE_SIZE=0x06
RM_OPT_GATHER_STATISTICS=FALSE
RM_DEBUG=FALSE
RM_OPT_BUILDIDENTIFIER=FALSE
RM_OPT_SDM_INFO=FALSE
RM_OPT_MEMORYMAP=FALSE
RM_OPT_USE_INTERRUPTS=TRUE
RM_FIFOSIZE=0x10
CHAIN_VECTORS=FALSE
```


Chapter 5
Integration

This chapter describes how to integrate RMTarget into your application or RTOS. It also describes how to set up exception handling in your application so that, when RMTarget is built and integrated, it can co-exist effectively with your application.

This chapter contains the following sections:

5.1 About RMTarget integration

After you have built the RMTarget library (see Chapter 4 *Building RMTarget*), there is some integration work you must perform prior to linking the RMTarget library into your application. Most of this work relates to the RealMonitor exception handlers.

You must first ensure that sufficient stack space is allocated within your application to support each processor mode RealMonitor uses. See *Adding stacks* on page 53 for complete details.

You must also ensure that the RealMonitor exception handlers are installed appropriately. Some of these handlers might share an exception with application code. For example, RealMonitor typically uses IRQs to drive the DCC link, so if your application uses IRQs, you must arrange to pass DCC interrupts to RealMonitor.

Finally, your application must call the RMTarget initialization function to start the communications link before you can perform any debugging.

Other integration considerations on page 510 describes some different ways of integrating RMTarget to meet different application requirements.

5.2 Integration procedure

Before you can use RMTarget with your application or RTOS, you must perform the following tasks:

- *Adding stacks*
- *Handling exceptions* on page 55
- *Calling the RMTarget initialization function* on page 59.

Note

Your foreground application might call RealMonitor to poll the DCC, which effectively disables RealMonitor from being interrupt-driven. In these cases, you do not need to consider the integration of IRQ handlers as described in *Handling exceptions* on page 55. Instead, see *Reducing interrupt latency by polling the DCC* on page 512.

In addition to performing these tasks, you might also have to perform other integration work, depending on your application requirements (see *Other integration considerations* on page 510).

Note

You must not perform any integration until you have built RMTarget (see *Building using the Makefile* on page 47 or *Building using CodeWarrior IDE* on page 42).

After you have performed integration (as described in this chapter), you must build your own application or RTOS using your preferred method of building. Then, you must link the RMTarget library you have built with the files and libraries of your own application (see *Linking using armlink* on page 49 or *Linking using CodeWarrior IDE* on page 46).

5.2.1 Adding stacks

You must ensure that stacks are set up within your application for each of the processor modes used by RealMonitor. For each mode, RealMonitor requires a fixed number of words of stack space. You must therefore allow sufficient stack space for both RealMonitor and your application. The RealMonitor demonstrations allocate 256 bytes for each mode, which is sufficient in most applications (see Chapter 3 *RealMonitor Demonstrations*).

RealMonitor has the following stack requirements:

FIQ mode RealMonitor makes no use of this stack.

IRQ mode A stack for this mode is always required. RealMonitor uses two words on entry to its interrupt handler. These are freed before nested interrupts are enabled.

Undef mode A stack for this mode is always required. RealMonitor uses 12 words while processing an undefined instruction exception.

SVC mode A stack for this mode is required only if you have built RMTarget to support SWI semihosting, that is, if the build option `RM_OPT_SEMIHOSTING` is enabled (see *RM_OPT_SEMIHOSTING* on page 418). RealMonitor uses four words on entry to its SWI interrupt handler.

If `RM_OPT_EXECUTECODE` is enabled, one additional word is required (see *RM_OPT_EXECUTECODE* on page 420).

Abort mode

Your application must initialize an abort mode stack if you have built RMTarget with any of the options that can install Data Abort or Prefetch Abort handlers. These include options that enable support for hardware breakpoints, watchpoints, memory reads, and memory writes. Therefore, a stack for this mode is required only if you have enabled any of the following options:

- *RM_OPT_HARDBREAKPOINT* on page 417
- *RM_OPT_HARDWATCHPOINT* on page 417
- *RM_OPT_READBYTES* on page 418
- *RM_OPT_WRITEBYTES* on page 419
- *RM_OPT_READHALFWORDS* on page 419
- *RM_OPT_WRITEHALFWORDS* on page 419
- *RM_OPT_READWORDS* on page 419
- *RM_OPT_WRITEWORDS* on page 420
- *RM_OPT_EXECUTECODE* on page 420.

Aborts can also be caused by application code attempting to access invalid areas of memory. RealMonitor uses four words on entry to its abort interrupt handler.

User/System mode

RealMonitor makes no use of this stack.

For an example of a stack implementation that is suitable for RealMonitor, see *Stack initialization* on page 311.

5.2.2 Handling exceptions

This section describes the importance of sharing exception handlers between RealMonitor and your application, and provides code examples to show how you can either use the RealMonitor default method of handling exceptions, or write your own code to share exceptions between RealMonitor and your application:

- *Overview of RealMonitor exception handling*
- *Example of installing default RealMonitor default exception handling* on page 57
- *Example of an application-specific IRQ handler* on page 58.

Overview of RealMonitor exception handling

To function properly, RealMonitor must be able to intercept certain interrupts and exceptions. Figure 5-1 on page 56 illustrates how exceptions can be claimed by RealMonitor itself, or shared between RealMonitor and your application. For each exception type, RealMonitor provides an exception handler that must be called when that exception occurs, such as `RM_SWIHandler()`.

If your application requires that you share the exception, you must provide your own function (such as *`app_IRQDispatch`*`()`). Depending on the nature of the exception, this handler can either:

- pass control to the RealMonitor processing routine, such as `RM_IRQHandler2()`
- claim the exception for the application itself, such as *`app_IRQHandler`*`()`.

In a simple case where an application has no exception handlers of its own, the application can install the RealMonitor low-level exception handlers directly into the vector table of the processor. The easiest way to do this is to write a branch instruction (`B <`*`address`*`>`) into the vector table, where the target of the branch is the start address of the relevant RealMonitor exception handler.

Note

If you do not want to change the interrupt handling mechanisms of your application, you can change your foreground application to call RealMonitor to poll the DCC. This allows RealMonitor to run without an IRQ handler. See *Reducing interrupt latency by polling the DCC* on page 512 for details.

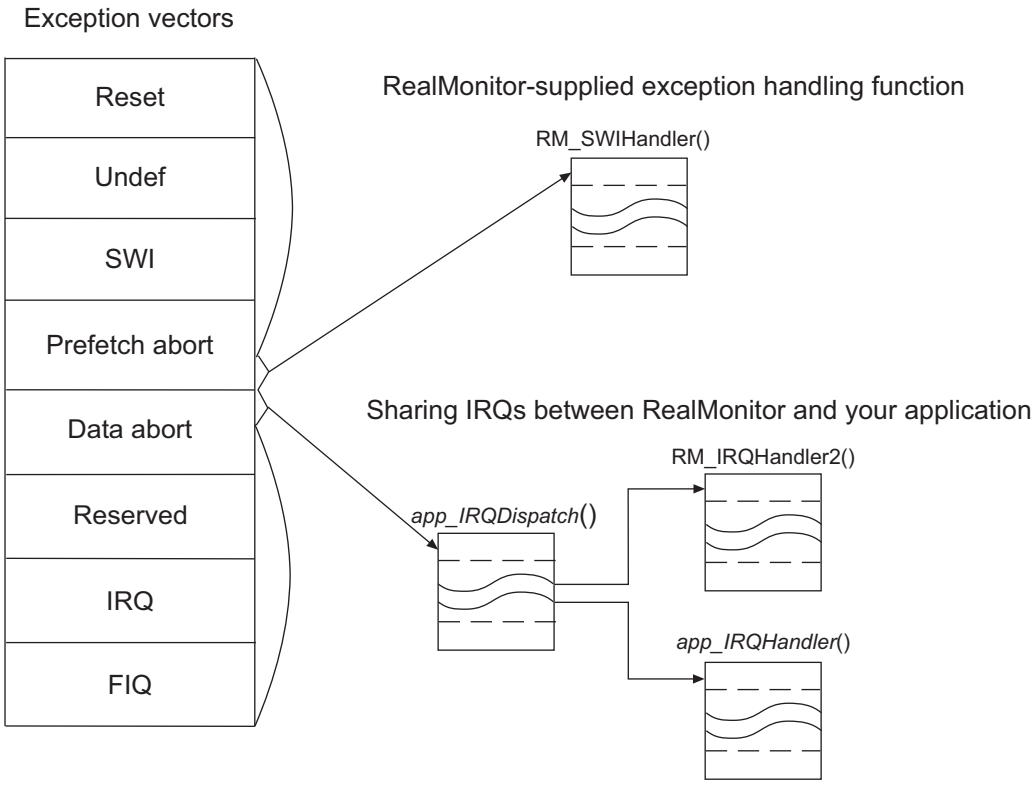

Figure 5-1 Exception handlers

Example of installing default RealMonitor default exception handling

Consider an application that uses only FIQs. RealMonitor provides handlers for IRQs and other exceptions that can be called directly. The application must only install the relevant RealMonitor handler on each exception vector.

Example 5-1 shows how to call the function `RM_InitVectors()`, which installs the RealMonitor default exception handlers. This function must be called prior to calling the function `RM_Init()`.

Example 5-1 Installing RealMonitor default exception handlers

```
#include "rm.h"

int main(void)
{
  RM_InitVectors();
  RM_Init():

  /* Your application code starts here. */
  .
  .
  .
}
```

Example of an application-specific IRQ handler

Consider an application that shares vectors with RealMonitor. This typically occurs in applications that must use IRQs for themselves, but it is possible the application might want to handle SWIs, undefined instructions, and other exceptions. In this case, the application must provide the initial vector handling, which can detect whether the exception is for RealMonitor, and branch to the relevant code.

Example 5-2 shows an IRQ handler that has been designed to distinguish interrupts intended for RealMonitor, and interrupts intended for a μHAL application. This is installed in place of the RealMonitor low-level interrupt handler, `RM_IRQHandler()`. In this example, the application IRQs are given a higher priority than RealMonitor IRQs.

Example 5-2 Sharing interrupt handlers with RealMonitor

```
        ; Get interrupt controller definitions from uHAL.
        INCLUDE platform.s
        INCLUDE target.s
        INCLUDE rm_comms.s

        ; Get definitions of uHAL_COMMRX_MASK and uHAL_COMMTX_MASK.
        INCLUDE democomms.s

        IMPORT RM_IRQHandler2
        IMPORT uHALir_DispatchIRQ

        AREA demo_trap, CODE, READONLY

        EXPORT App_IRQDispatch
        ROUT
App_IRQDispatch
        ; Point the lr back to the instruction that was interrupted.
        SUB lr, lr, #4

        ; Save enough registers to call APCS-compliant C functions.
        STMFD sp!, {r0-r3, ip, lr}

        ; Use uHAL's platform-specific macro to return a bit-mask containing
        ; a '1' for each active interrupt source.
        READ_INT r0, r1, r2

        ; Check whether anything other than a RealMonitor IRQ occurred.
        ; This gives priority to app interrupts over DCC interrupts.
       BICS r0, r0, #uHAL_COMMRX_MASK :OR: uHAL_COMMTX_MASK
       BEQ no_app_irqs
```



```
app_IRQHandler
        ; An application-specific interrupt has occurred. Pass control to
        ; application code (in this case, uHAL's interrupt dispatcher).
        BL uHALir_DispatchIRQ

        ; Return to the foreground application.
        LDMFD sp!, {r0-r3, ip, pc}^


        ; No app IRQs occurred, so process RealMonitor traffic.
no_app_irqs
        ; Remove r0-r3 from the stack, leaving 'ip' and the adjusted 'lr', then
        ; call RM's interrupt handler.
        LDMFD sp!, {r0-r3}
        LDR ip, =IMP_IRQ_BASE
        B RM_IRQHandler2
```

5.2.3 Calling the RMTarget initialization function

While the processor is in a privileged mode, and IRQs are disabled, you must include a line of code within the startup sequence of your application to call `RM_Init()`, an example of which is shown in Example 5-3.

Example 5-3 Calling RM_Init()

```
#include "rm.h"

int main(void)
{
  /* Cannot debug these statements */
  .
  .
  .
  RM_Init();
  /* Can debug these statements */
  .
  .
  .
}
```

Note

You cannot perform any debugging until the `RM_Init()` function is called.

5.3 Other integration considerations

After you have performed the *Integration procedure* on page 53, you must also ensure that your application is not affected adversely when it is RealMonitor-enabled, because there are additional coding considerations for RealMonitor-enabled applications. This section describes the additional considerations and restrictions you must be aware of, and for which you might have to take counteractive measures, to ensure that your application is integrated properly. These considerations can be categorized as follows:

- *SWI numbers*
- *Interrupt latency*
- *Linking with mHAL* on page 512.
- *Using the AFS chaining library* on page 513.

For details on integrating RMTarget to an RTOS, see *Integrating RMTarget into an RTOS* on page 514.

Note

RealMonitor is not compatible with systems that are based on the (obsolete) 26-bit ARM architecture.

Because of the default *Field-Programmable Gate Array* (FPGA) configuration supplied with most Integrator core modules, you cannot use images built for big-endian targets with these boards.

5.3.1 SWI numbers

If your application defines its own SWIs, and also wants to use ARM semihosting SWIs, you must modify your SWI handler to detect semihosting SWIs, and pass them to the RealMonitor SWI handler. Semihosting SWIs have the number `0x123456` for ARM code, and `0xAB` for Thumb code. For more details about semihosting SWIs, see the Semihosting chapter in the *ADS Debug Target Guide*.

Caution

You must not attempt to use semihosting SWIs from within an IRQ or FIQ handler. If an interrupt handler attempts to call a semihosting SWI while RealMonitor is processing data, RealMonitor enters the panic state (see Figure 1-2 on page 17).

5.3.2 Interrupt latency

Adding RealMonitor to your application can result in increased interrupt latency.

This section describes the interrupt latency considerations when integrating with RealMonitor:

- *Interrupt latency using the default RMTarget*
- *Reducing interrupt latency by polling the DCC* on page 512.

Interrupt latency using the default RMTarget

If you do not bypass the default RealMonitor method of handling interrupts, as described in *Reducing interrupt latency by polling the DCC* on page 512, the RealMonitor default method for reducing interrupt latency is used when your application is running.

RealMonitor is designed to keep interrupt latency to a minimum by ensuring that it never disables FIQs. It also avoids leaving IRQs disabled for long periods.

When a DCC interrupt occurs, RealMonitor does the following:

1. Disables both DCC TX and RX interrupts.
2. Saves the processor state.
3. Switches to Undef mode.
4. Re-enables IRQs in the processor. (IRQs are automatically disabled when the processor enters IRQ mode.)
5. Processes the DCC interrupt.

This allows an application interrupt to be serviced while RealMonitor is processing the original DCC interrupt.

If any other exceptions occur during application processing (such as a Prefetch Abort when a breakpoint is hit), RealMonitor also attempts to re-enable interrupts as soon as possible.

Note

IRQs in the processor are not re-enabled if they were disabled by the application when an exception occurred. In this case, RealMonitor leaves IRQs disabled until it has finished processing the exception.

Reducing interrupt latency by polling the DCC

In cases where your application cannot tolerate any increase in interrupt latency, you can use RealMonitor in a polled mode. To do this, make the following changes:

1. Disable the build option `RM_OPT_USE_INTERRUPTS` at build-time to ensure that no DCC interrupt-handling code is generated.
2. Call the function `RM_PollDCC()` periodically from the foreground task in your application (see *RM_PollDCC()* on page 661).

Example 5-4 shows how the initialization sequence of your application should appear if you make these changes.

Example 5-4 Polling the DCC

```
#include "rm.h"

int main(void)
{
    /* Initialize the RM state machine. */
    RM_InitVectors();
    RM_Init();

  while(1)
  {
    /* Application polling code */
    .
    .
    .
    /* Poll DCC for RM commands. */
    RM_PollDCC();
  }
}
```

5.3.3 Linking with μHAL

As described in *mHAL RealMonitor application demonstration* on page 319, you can link RealMonitor with μHAL to simplify the task of initializing the target. This allows you to share interrupts between RealMonitor and your application with no additional work.

The μHAL library handles all the board initialization. Your application simply needs to call `RM_uHAL_Init()` in its initialization sequence (instead of `RM_Init()` and `RM_InitVectors()`) to initialize RealMonitor support. This also installs

`rm_uHAL_IRQHandler()` on the IRQ vector, which provides support for sharing IRQs between RealMonitor and µHAL (see *RM_uHAL_Init()* on page 633 and *rm_uHAL_IRQHandler()* on page 634). Therefore, if your application uses the µHAL interface to claim IRQs, no further integration work is necessary, and none of the tasks described in *Integration procedure* on page 53 is relevant.

5.3.4 Using the AFS chaining library

The chaining library in AFS provides support for separately loaded applications to share exception handling. Chaining support needs to be present in the first application loaded (the *chain controller*) and in any subsequent applications that want to share exceptions with the controller (the *chain clients*). For complete details on the chaining library, see the chapter on the chaining library in the *ARM Firmware Suite Reference Guide*.

RealMonitor can benefit from integrating µHAL with the chaining library. To enable this integration, you must specify `CHAIN_VECTORS=1` when you rebuild µHAL, and set `CHAIN_VECTORS=TRUE` when you build the RMTarget library.

The µHAL RealMonitor application demonstration is useful as a chain controller. If the image `RM_uHAL.axf` is running on your target, you can connect to a target using RMHost, and download non RealMonitor-enabled applications (those with µHAL and the chain library enabled), such as the `Dhrystone` application in *Procedure for running the RealMonitor application demonstration* on page 315.

Applications that use the chaining library can claim exceptions in a way that co-operates with the RealMonitor method of exception handling.

5.4 Integrating RMTarget into an RTOS

The existing RealMonitor code does not a support a task-switching environment unless you first perform specific porting work, which varies according to the interrupt demands of your RTOS.

The benefits to integrating your RTOS with RMTarget are as follows:

- Standalone RealMonitor does all processing within the DCC interrupt handler, whereas the RTOS version does as much as possible within a thread. By configuring the priority of this thread, you can control how much of an effect large DCC transfers have on the RealMonitor CPU usage.
- You can identify which threads in your target are responsible for dealing with real-time aspects of the system. RealMonitor can be instructed not to stop these threads when it receives a `Stop` request from the host. This differs from standalone applications, where RealMonitor can leave interrupts and FIQs running, but not foreground code. The RTOS version therefore gives you more control over what to leave running.

If you do not perform this porting work, RealMonitor can behave in unexpected ways. The following sequence of events shows the problem that occurs if you do not integrate your RTOS with RMTarget:

1. The RealMonitor IRQ handler receives a `stop` command from the host.
2. RealMonitor saves the state of whichever foreground task is currently running, into the register block.
3. RealMonitor re-enables IRQs and goes into a poll loop.
4. A timer interrupt occurs, causing the time slice of the current task to expire.
5. The RTOS performs a context switch, replacing the RealMonitor poll loop with a different task context.

When this sequence of events occurs, RealMonitor does not run again until the RTOS decides to re-activate the original foreground task. Only one task, the one running when the `stop` command was received, has effectively been stopped. This stops any debugger target communication.

There are simple ways to avoid this situation, but these do not make use of the ability of RealMonitor to leave real-time tasks running while the target is stopped. For example, RealMonitor can:

- disable interrupts when the target is stopped
- request that the RTOS disables the task-switcher.

However, to take full advantage of RealMonitor functionality in an RTOS context, it is recommended that you follow the approach described in this section. This approach allows a specific subset of the tasks to be stopped by RealMonitor, leaving the rest of the system to respond to real-time events:

- *Overview of the threads solution*
- *DCC driver* on page 516
- *The monitor thread* on page 517
- *The exception-processing thread* on page 517
- *Stopping and restarting* on page 518
- *Implementation* on page 518.

Note

The approach described in these subsections is demonstrated in the uC/OS-II port provided when you install RealMonitor. For more details, see `readme_uCOS.txt` in the `RealMonitor` directory of the AFS 1.2 installation.

The code for this port can be used for porting to other operating systems.

This approach is designed to allow RMTarget to work with your RTOS so that:

- you can select those threads that are to be stopped when the target is stopped (with a `stop` command), and those that are to be left running
- you can select a particular thread to be displayed in the debugger when the target is stopped, allowing you to view the saved register contents of that thread.

5.4.1 Overview of the threads solution

For the solution described in the following sections, you must create two new threads that are required for RealMonitor operation:

- A monitor thread that processes communication with the debugger, and that controls the stopping and starting of the target (see *The monitor thread* on page 517). The work of transmitting and receiving DCC data is shared between the monitor thread and an interrupt-driven DCC driver (see *DCC driver* on page 516).
- A thread to handle breakpoints and other processor exceptions (see *The exception-processing thread* on page 517).

Additionally, various semaphores are required to allow these components to communicate with each other.

5.4.2 DCC driver

You must write a DCC driver to allow the sending and receiving of words of data in an interrupt-driven manner (see the uC/OS-II port example described in `readme_uCOS.txt`). This must buffer one word of data in each direction to allow for rapid transmission (between target and host).

The DCC driver has two main responsibilities:

- *Transmitting DCC data*
- *Receiving DCC data*.

Transmitting DCC data

The transmit component of the driver must operate as follows:

1. Wait for a COMMTX interrupt.
2. Send the next word of data, over the DCC, from the one-word transmit buffer.
3. Disable the COMMTX interrupt source so that no further interrupts are received until the monitor thread has placed the next word of data in the buffer.
4. Post a semaphore.

The semaphore that is posted then allows the monitor thread to generate new data for transmission (see *The monitor thread* on page 517).

Receiving DCC data

The receive component of the driver must operate in a similar manner:

1. Wait for a COMMRX interrupt.
2. Read the next word of data, from the DCC, and store it in the one-word receive buffer.
3. Disable the COMMRX interrupt source so that no further interrupts are received until the monitor thread has processed the word of data in the buffer.
4. Post a semaphore.

The semaphore that is posted then allows the monitor thread to process the incoming data (see *The monitor thread* on page 517).

5.4.3 The monitor thread

You must create a thread that initializes the RealMonitor state by calling the function `RM_Init()` (see *RM_Init()* on page 630). It must then enter a loop and perform the following:

1. Wait for one of the components (transmit or receive) of the DCC driver, or the exception handler, to post a semaphore (see *DCC driver* on page 516 and *The exception-processing thread*).
2. If data is received, it must do the following:
   a. Process escape sequences, and send the word to the RealMonitor protocol handler by calling the `IMP_GlobalState->rxproc` handler in the `IMP_GlobalState` structure (see *IMP_GlobalState* on page 69).
   b. Re-enable the COMMRX interrupt source.
3. If an exception has occurred, or a `stop` command has been received from the host, it must enter the stopped state (see *Stopping and restarting* on page 518).
4. If the transmit buffer is empty, it must do the following:
   a. Check whether the second word of an escape sequence must be sent. If this is the case, place this word into the target buffer.
      Otherwise, call the `IMP_GlobalState->txproc` handler in the `IMP_GlobalState` structure to generate the next word of data to be transmitted (see *IMP_GlobalState* on page 69). If there is no data to transmit, `IMP_GlobalState->tx_flag` is cleared by the `txproc` handler.
   b. Re-enable COMMTX interrupts if `tx_flag` is set.

Note

If you integrate RMTarget using this threads solution, the monitor thread and the IRQ handlers continue to run while the target is stopped, that is, control is never passed to the function `rm_StoppedLoop()` (see *rm_StoppedLoop()* on page 670).

You can decrease the priority of the monitor thread to allow other tasks to take priority over processing DCC traffic.

5.4.4 The exception-processing thread

The task of processing exceptions is shared by the RealMonitor exception handlers and a dedicated thread. You must write specific exception handlers that:

- Record into memory which exception occurred, and which thread was running at the time.

- Post a semaphore to wake up the exception-processing thread.
- Return control to the RTOS scheduler.

You must also create a high-priority exception-processing thread that does the following:

1. Suspend the task that caused the exception.
2. Post a semaphore to the monitor thread, causing RealMonitor to enter the stopped state.

5.4.5 Stopping and restarting

When RealMonitor enters the stopped state, you must ensure that the monitor thread does the following:

1. Call the RTOS to suspend the threads you have designated as non real-time. The remaining high-priority threads must continue to run to service real-time events.
2. To allow the debugger to display the CPU registers of a designated thread, it must copy the saved thread context information into `RM_Registers` (see *RM_Registers* on page 620).

To restart RealMonitor from a stopped state, you must ensure that the monitor thread does the following:

1. Copy information from `RM_Registers` into the thread context structure.
2. Call the RTOS to restart the threads that were previously stopped.

5.4.6 Implementation

To create an RTOS-aware implementation of this threads solution, you must create code that implements the following:

- exception handling
- saving and restoring registers
- DCC communication.

The file `rm_asm.s` implements these for a non RTOS-aware environment (see *RMTarget source files* on page 27).

You must therefore code a RTOS-specific implementation that uses the RMTarget API in the same manner as the file `rm_asm.s`. This requires that you:

- implement the following functions:
  - *rm_EnableRXTX()* on page 668

  - *rm_EnableTX()* on page 668
  - *rm_ResetComms()* on page 669
  - *rm_Go()* on page 677
  - *rm_DoGetPC()* on page 6106.
- call the following functions appropriately:
  - *rm_InitCommsState()* on page 630
  - *rm_RX()* on page 658
  - *rm_SetPending()* on page 664
  - *rm_TXDone()* on page 666.

Chapter 6
Application Program Interface

This chapter describes the *Application Program Interface* (API) to ARM RealMonitor. It contains the following sections:

6.1 RealMonitor naming conventions and data types

This section describes the naming conventions used for the functions and variables in this API, and describes the data types that are used:

- *Naming conventions*
- *Basic data types*
- *Complex data types*.

6.1.1 Naming conventions

All functions and variables in this API are prefixed with one of the following:

`RM_` A name that is external to RealMonitor.

`rm_` A name that is internal to RealMonitor.

`IMP_` A name that is external to the RealMonitor protocol.

A name is considered external if it can be used by an application or RTOS to integrate with RealMonitor. A name is defined as internal if it is unlikely to be used by an application or RTOS. Internal names are not always exported by the RMTarget library, and might not have C header file definitions.

Note

When performing a simple integration, only `RM_` functions and variables are most likely to be applicable.

6.1.2 Basic data types

RealMonitor relies very closely on the size and signedness of its basic data types for efficient and correct operation. These basic data types are defined in `rm_types.h` (see *RMTarget source files* on page 27):

`int32` A 32-bit signed integer.

`uint8` An 8-bit unsigned integer.

`uint16` A 16-bit unsigned integer.

`uint32` A 32-bit unsigned integer.

6.1.3 Complex data types

RealMonitor defines the following complex data types, which are defined in the files `rm_proto.h` and `rm_state.h` (see *RMTarget source files* on page 27).

DispatchProc

Functions of this type are called when the first word of a new packet is received. The definition is:

```
typedef void DispatchProc(uint32 length)
```

RxHandler

Functions of this type are called to process each word of data that is received over the DCC. The definition is:

```
typedef void RxHandler(uint32 data)
```

TxHandler

Functions of this type are called by `rm_FillTransmitBuffer()` to obtain each successive word of a packet. The returned word is either sent directly over the DCC, or buffered for later transmission. The definition is:

```
typedef uint32 TxHandler(void)
```

ReadProc

Functions of this type are called to read memory as part of a ReadBytes, ReadHalfWords, or ReadWords packet. The definition is:

```
typedef uint32 ReadProc(uint32 dummy, uint32 address, uint32 length)
```

WriteProc

Functions of this type are called to write to memory as part of a WriteBytes, WriteHalfWords, or WriteWords packet. The definition is:

```
typedef void WriteProc(uint32 address, uint32 length, uint32 word)
```

RM_ControllerOpcodes

This is an enumeration that defines all possible opcodes that the target expects to receive from the host. The values assigned to each item in the enumeration correspond to the opcode numbers defined in the RealMonitor protocol (see *Host-to-target messages* on page 77). The definition is:

```
typedef enum {
    RM_Cmd_NOP              = 0x00,
    RM_Cmd_GetCapabilities  = 0x01,
```



```
    RM_Cmd_Stop             = 0x02,
    RM_Cmd_Go               = 0x03,

    RM_Cmd_GetPC            = 0x04,
    RM_Cmd_SyncCaches       = 0x05,
    RM_Cmd_ExecuteCode      = 0x06,
    RM_Cmd_InitialiseTarget = 0x07,
    RM_Cmd_ReadBytes        = 0x08,
    RM_Cmd_WriteBytes       = 0x09,
    RM_Cmd_ReadHalfWords    = 0x0a,
    RM_Cmd_WriteHalfWords   = 0x0b,
    RM_Cmd_ReadWords        = 0x0c,
    RM_Cmd_WriteWords       = 0x0d,

    RM_Cmd_ReadRegisters    = 0x10,
    RM_Cmd_WriteRegisters   = 0x11,

    RM_Cmd_ReadCP           = 0x12,
    RM_Cmd_WriteCP          = 0x13
} RM_ControllerOpcodes;
```

RM_TargetOpcodes

This is an enumeration that defines all the possible opcodes that the target can send to the host. The values assigned to each item in the enumeration correspond to the opcode numbers defined in the RealMonitor protocol (see *Target-to-host controller messages* on page 79). The definition is:

```
typedef enum
{
    RM_Msg_NOP              = 0x00,
    RM_Msg_Okay             = 0x80,
    RM_Msg_Error            = 0x81,
    RM_Msg_Stopped          = 0x90,
    RM_Msg_SoftBreak        = 0x91,
    RM_Msg_HardBreak        = 0x92,
    RM_Msg_HardWatch        = 0x93,
    RM_Msg_SWI              = 0x94,
    RM_Msg_Undef            = 0xA0,
    RM_Msg_PrefetchAbort    = 0xA1,
    RM_Msg_DataAbort        = 0xA2
} RM_TargetOpcodes;
```


RM_Errors

This is an enumeration that defines the errors that can be stored in the error block. The values assigned to each item in the enumeration correspond to the error numbers defined by the RealMonitor protocol (see *RealMonitor packets* on page 712). This enumeration is defined in `rm_proto.h` as follows:

```
typedef enum
{
    RM_Error_None               = 0x00,
    RM_Error_UnsupportedOpcode  = 0x01,
    RM_Error_DataAbort          = 0x02
} RM_Errors;
```

where:

`RM_Error_None`

Indicates that no error has occurred. There is no additional data associated with this error.

`RM_Error_UnsupportedOpcode`

Indicates that a packet was received on the RealMonitor channel with an unknown or unsupported opcode. There is no additional data associated with this error.

`RM_Error_DataAbort`

Indicates that a Data Abort occurred during the processing of a ReadBytes, ReadHalfWords, ReadWords, WriteBytes, WriteHalfWords, WriteWords, or ExecuteCode opcode. The address that generated the Data Abort is written into `error_block[2]`.

Escape sequences

These define the values used as part of an escape sequence. The syntax is:

```
#define IMP_ESCAPE_QUOTE 0
#define IMP_ESCAPE_RESET 1
#define IMP_ESCAPE_GO    2
```

Channel numbers

This enumeration defines the logical channels that can be used when transferring packets over the DCC. The syntax is:

```
typedef enum
{
    IMP_Chan_RTM     = 0x00,
    IMP_Chan_DCCSemi = 0x01,
} IMP_Channel;
```

State

This defines the various states that RealMonitor can be in at any given time. The syntax is:

```
#define RM_State_Running        (0x00)
#define RM_State_Stopped        (0x01)
#define RM_State_Panic          (0x02)
```

where:

`RM_State_Running`

Indicates that the foreground application is running.

`RM_State_Stopped`

Indicates that the foreground application is stopped. Interrupt-driven tasks can continue to execute in the background.

`RM_State_Panic`

Indicates that RealMonitor is in a situation from which it is unable to recover.

Memory map attributes

These macros are defined in `rm_proto.h` (see *RMTarget source files* on page 27), and describe the bit fields that are used to create memory maps. See *Tag 0x00000016, Pointer to memory descriptor block* on page 746 for details on the meanings of these attributes. The definition is:

```
#define RM_MD_READABLE          (1 << 16)
#define RM_MD_WRITABLE          (1 << 17)

#define RM_MD_ENDIAN_LITTLE     (0 << 18)
#define RM_MD_ENDIAN_BIG        (1 << 18)
#define RM_MD_ENDIAN_EITHER     (2 << 18)
#define RM_MD_ENDIAN_RESERVED   (3 << 18)
#define RM_MD_ENDIAN_MASK       (3 << 18)

#define RM_MD_STOPPABLE         (1 << 20)

#define RM_MD_ACCESS_ANY        (0 << 21)
```



```
#define RM_MD_ACCESS_BYTE       (1 << 21)
#define RM_MD_ACCESS_HALFWORD   (2 << 21)
#define RM_MD_ACCESS_WORD       (3 << 21)
#define RM_MD_ACCESS_MASK       (3 << 21)
#define RM_MD_CACHE_NOSYNC      (1 << 23)
```

6.2 Data structures

This section describes the data structures used by RealMonitor:

- *rm_DispatchTable*
- *IMP_GlobalState* on page 69
- *rm_CapabilitiesTable* on page 615.

6.2.1 rm_DispatchTable

This array contains a function pointer for each opcode in the RealMonitor protocol. The functions are called by `rm_RX()` when a packet header is received on the RealMonitor channel (see *rm_RX()* on page 658).

Index 0 in the array contains the pointer for opcode `0x00`, index 1 in the array contains the pointer for opcode `0x01`, and so on. Opcodes that are unsupported by a particular build of RealMonitor (as specified by the relevant build option) must have their function pointer set to NULL.

If the end of the array contains a sequence of NULL pointers, those function pointers are omitted from the array. This reduces the amount of data memory required by RealMonitor.

The contents of this array are never modified at runtime, so you can safely place it into ROM.

Syntax

```
static const DispatchProc *rm_DispatchTable[] =
    {
    rm_NOP,               /* opcode 0x00 - always required */
    rm_GetCapabilities,   /* opcode 0x01 - always required */
    rm_Stop,              /* opcode 0x02 - optional (RM_OPT_STOPSTART) */
    rm_Go,                /* opcode 0x03 - optional (RM_OPT_STOPSTART) */
    rm_GetPC,             /* opcode 0x04 - optional (RM_OPT_GETPC) */
    rm_SyncCaches,        /* opcode 0x05 - optional (RM_OPT_SYNCCACHES) */
    rm_ExecuteCode,       /* opcode 0x06 - optional (RM_OPT_EXECUTECODE) */
    NULL,                 /* opcode 0x07 - unimplemented (InitializeTarget) */
    rm_ReadBytes,         /* opcode 0x08 - optional (RM_OPT_READBYTES) */
    rm_WriteBytes,        /* opcode 0x09 - optional (RM_OPT_WRITEBYTES) */
    rm_ReadHalfWords,     /* opcode 0x0A - optional (RM_OPT_READHALFWORDS) */
    rm_WriteHalfWords,    /* opcode 0x0B - optional (RM_OPT_WRITEHALFWORDS) */
    rm_ReadWords,         /* opcode 0x0C - optional (RM_OPT_READWORDS) */
    rm_WriteWords,        /* opcode 0x0D - optional (RM_OPT_WRITEWORDS) */
    NULL,                 /* opcode 0x0E - unimplemented (RESERVED) */
    NULL,                 /* opcode 0x0F - unimplemented (RESERVED) */
    rm_ReadRegisters,     /* opcode 0x10 - optional (RM_OPT_READREGISTERS) */
    rm_WriteRegisters,    /* opcode 0x11 - optional (RM_OPT_WRITEREGISTERS) */
    rm_ReadCP,            /* opcode 0x12 - optional (RM_OPT_READCP) */
    rm_WriteCP            /* opcode 0x13 - optional (RM_OPT_WRITECP) */
    };
```

6.2.2 IMP_GlobalState

This data structure contains the entire state of RMTarget. Because this structure is accessed very frequently, the base address of the data structure is allocated permanently to ARM register r4 using the `__global_reg()` keyword provided by the ARM C compiler.

Register r4 is used because it is nonbanked in all processor modes. You must not change the register number because a large amount of assembly code is hardwired to use this register.

RealMonitor must ensure that the global register is initialized properly when an application or operating system calls any of its external functions (those prefixed with `RM_`). When an external function is called, it must save the previous value of r4 into another register, or onto the stack, and initialize r4 to point to the global state data structure. When an external function returns, it must restore the previous value of r4 from wherever it was saved.

The global state data structure is defined as follows:


```
typdef struct {
    RxHandler *rxproc;
    TxHandler *txproc;
    TxHandler *txpending;

    uint32  stop_code;
    uint8   state;
    uint8   exception_flag;
    uint8   operation_aborted;
    uint32  header;

    uint32  rwaddress;
    uint32  rwlength;
    uint32  rwmask;
    void    *rwproc;

    uint32  pending_data;
    struct {
        rm_bool hasdata;
        rm_bool off;
        uint32  data[RM_FIFOSIZE];
        uint32  insert;
        uint32  remove;
        int32   current;
    } fifo;

    uint8   rx_escape_flag;
    uint8   tx_flag;
    uint32  tx_buffer;
    uint32  tx_escape_data;

    uint32  error_block[3];
} State;
```

where:

`RxHandler *rxproc`

A pointer to a function that processes each successive word of a packet as it is received over the DCC. The function is called with a single argument containing the word just read from the DCC.

`TxHandler *txproc`

A pointer to a function that generates each successive word of a packet. When called, the function returns the next word of data to be sent over the DCC.

`TxHandler *txpending`

A pointer to a function that handles the DCC transmit interrupt.

`uint32 stop_code`

Stores the packet header that is generated when RealMonitor stops (see *Starting and stopping the foreground application* on page 670). This packet header always has a length field of zero, indicating that there is no payload.

`uint8 state`

Contains the coarsest indicator of the current state of RealMonitor. See *State* on page 66 for the possible values.

`uint8 exception_flag`

Indicates the action to be taken when a Data Abort is detected. Possible values include:

zero Any Data Aborts are treated as a software breakpoint. The foreground application is stopped, and the host is notified.

nonzero An opcode is currently in progress. The Data Abort is reported in the error block (see *Using the error_block array of IMP_GlobalState* on page 614), and execution continues with the instruction after the one which caused the exception.

`uint8 operation_aborted`

Indicates whether a Prefetch Abort, Data Abort, or undefined exception has occurred since this flag was last cleared. This flag is used by the memory access opcodes so that they can prevent further memory accesses from occurring if the previous memory accesses resulted in a Data Abort. Possible values include:

zero No Prefetch Abort, Data Abort, or undefined exception has occurred since the start of the current packet.

nonzero A Prefetch Abort, Data Abort, or undefined exception has occurred since the start of the current packet.

`uint32 header`

A copy of the packet header for the RealMonitor opcode currently being executed.

`uint32 rwaddress`

The default address from which the next read or write memory opcode begins.

`uint32 rwlength`

The number of bytes remaining to be transferred to or from memory as part of a read or write memory opcode.

`void *rwproc`

A pointer to a function that handles the specific memory read or write opcode.

`uint32 pending_data`

Stores the last word of a packet, prior to it being sent by `rm_SendPendingData()` (see *rm_SendPendingData()* on page 674). This is typically used as an efficient way of sending one or two word packets, and packets where the last word represents the success or failure status of a RealMonitor operation.

`rm_bool hasdata`

A Boolean variable that indicates whether the data logging FIFO buffer contains any data. This variable is available only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416). It is part of the `fifo` structure.

`rm_bool off`

A Boolean variable that indicates whether the data logging FIFO buffer is enabled or disabled. This variable is available only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416). It is part of the `fifo` structure.

`uint32 data[RM_FIFOSIZE]`

An array that stores the data logging packets before they are transmitted over the communications link. Packets are added to the array by the function `RM_SendPacket()` (see *RM_SendPacket()* on page 655), and are removed by the function `rm_EmptyFifo()` (see *rm_EmptyFifo()* on page 656).

The size of the array, and therefore the maximum packet length that can be transmitted, is determined by the build option `RM_FIFOSIZE` (see *RM_FIFOSIZE* on page 427).

This array is available only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416). It is part of the `fifo` structure.

`uint32 insert`

This variable gives the index within the data logging FIFO buffer to where the next word of data will be written. The function `RM_SendPacket()` is responsible for writing data into the FIFO and updating this variable to point to the next free location (see *RM_SendPacket()* on page 655).

This variable is available only when the build option RM_OPT_DATALOGGING is set to TRUE (see *RM_OPT_DATALOGGING* on page 416). It is part of the fifo structure.

`uint32 remove`

This variable gives the index within the data logging FIFO buffer from where the next word of data will be read. The function rm_EmptyFifo() is responsible for reading data from the FIFO buffer, and updating this variable to point to the next data item (see *rm_EmptyFifo()* on page 656).

This variable is available only when the build option RM_OPT_DATALOGGING is set to TRUE (see *RM_OPT_DATALOGGING* on page 416) . It is part of the fifo structure.

`int32 current`

This variable is used to store the length of the data logging packet that is currently being transmitted. The variable is set by the function rm_NextMessage() whenever it transmits a packet header (see *rm_NextMessage()* on page 657).

This variable is available only when the build option RM_OPT_DATALOGGING is set to TRUE (see *RM_OPT_DATALOGGING* on page 416). It is part of the fifo structure.

`uint8 rx_escape_flag`

Indicates whether the next word received on the DCC is part of an escape sequence, or is part of the normal data stream (see *Escape sequence handling* on page 74). Possible values include:

zero The next word received over the DCC is part of the normal data stream (or might be the start of an escape sequence).

nonzero The next word received over the DCC is the end of an escape sequence (or might be the start of another escape sequence).

`uint8 tx_flag`

Indicates whether the tx_buffer and tx_escape_data (by implication) variables contain data to be sent over the DCC. Possible values include:

zero There is no data waiting to be sent, and the value of tx_buffer is undefined.

nonzero There is data waiting to be sent. The next word is stored in tx_buffer, and depending on the exact value of tx_buffer, tx_escape_data might contain the next word after that.

`uint32 tx_buffer`

Contains the next word of data to be sent over the DCC. Whenever the escape word is stored in this variable, `tx_escape_data` contains the corresponding escape value.

`uint32 tx_escape_data`

Contains the next-but-one word of data to be sent over the DCC. The contents are valid only when an escape sequence is being sent (when `tx_buffer` contains the escape word).

`uint32 error_block[3]`

An array of words containing the details of the last error that occurred. See *Using the error_block array of IMP_GlobalState* for complete details.

Using the error_block array of IMP_GlobalState

The `IMP_GlobalState->error_block` is an array of words containing the details of the last error that occurred. The host reads the contents of the error block by sending appropriate read memory requests, using the address stored within the `rm_CapabilitiesTable` (see *rm_CapabilitiesTable* on page 615).

The error block is formatted in a style similar to the format used by RealMonitor protocol packets, that is, there is a packet header and a payload, as shown in Figure 6-1 on page 615. The first word of the error block contains the packet header. The packet header always contains the channel `IMP_Chan_RTM` and the Ok opcode (see the description of channel numbers in *Complex data types* on page 62). The length field indicates the amount of data relevant to the current error.

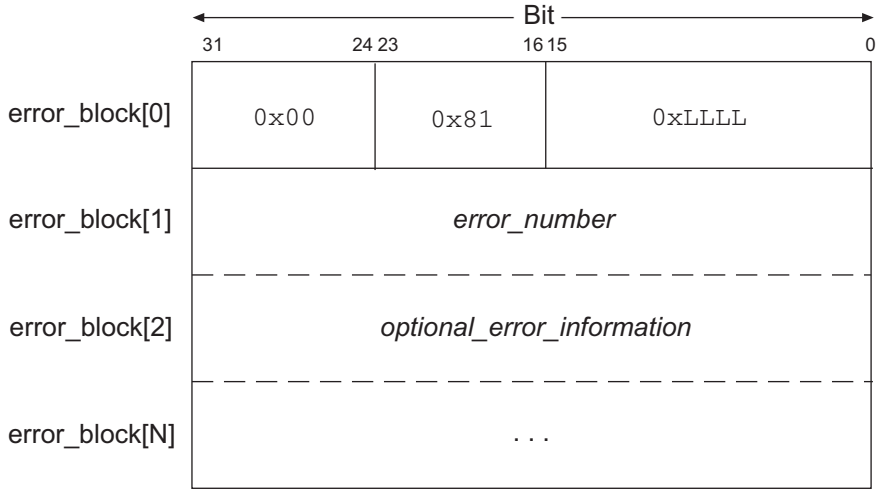

Figure 6-1 Format of the error block

With the errors defined in *RM_Errors* on page 65, the maximum size of the error block is three words long. In the case that none of the read, write memory, or ExecuteCode opcodes are supported, the error block is reduced to just two words.

The details of an error remain in the error block until a subsequent error occurs. Subsequent errors overwrite older errors, so the host must attempt to query the error block as soon as possible after an error has been detected.

When `RM_Init()` is called (see *RM_Init()* on page 630), the error block is set to contain `RM_Error_None`. This prevents unpredictable behavior from occurring if the host reads the error block and no error has occurred.

6.2.3 rm_CapabilitiesTable

This array allows the host to discover the build-time configuration of RMTarget, and enables the host to read or write internal RealMonitor data structures.

The capabilities table is stored in a word array called `rm_CapabilitiesTable`. This array is not part of the global state data structure because the contents of the capabilities block is fixed at compile-time, and never modified by RealMonitor at runtime. This allows the capabilities table to be placed into ROM.

The capabilities table consists of the following tag-value pairs:

`RM_Cap_ProtocolVersion (0x00000001)`

Indicates the version of the protocol that RealMonitor supports. In the current implementation, the version is `0x41000200`.

`RM_Cap_ConfigWord (0x00000002)`

The value of the configuration word relies on the fact that the configuration options take the value 0 or 1. The configuration options are multiplied by (1 << bit) to toggle that bit on or off in the configuration word.

`RM_Cap_BuildIdentifier (0x00000003)`

Pointer to a zero-terminated ASCII string. The string provides you with a means of identifying different builds of RealMonitor. The string is currently defined in `rm_control.c` as:

```
__TIME__ " " __DATE__
```

You can customize this string as required. For example, you can include the name, version, and build number of your product (see *RMTarget source files* on page 27).

This tag-value pair and the corresponding string are available only when the build option `RM_OPT_BUILDIDENTIFIER` is set to `TRUE` (see *RM_OPT_BUILDIDENTIFIER* on page 426).

`RM_Cap_TargetState (0x00000010)`

Pointer to `IMP_GlobalState->state` (see *State* on page 66). This tag-value pair is available only when the build option `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416).

`RM_Cap_RegisterBlockPtr (0x00000012)`

Pointer to the register block (see *rm_ExecuteCodeBuffer* on page 618). This tag-value pair is available only when the build option `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416).

`RM_Cap_ErrorBlockPtr (0x00000013)`

Pointer to the error block (see *Using the error_block array of IMP_GlobalState* on page 614).

`RM_Cap_RegisterAccessPtr (0x00000014)`

Pointer to the register accessibility block (see *RM_RegisterAccess* on page 620). This tag-value pair is available only when the build option `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416).

`RM_Cap_ExecuteCodeBlockPtr (0x00000015)`

Pointer to the execute code block (see *RM_ExecuteCodeBlock* on page 619), which describes the execute code buffer (see *rm_ExecuteCodeBuffer* on page 618). This tag-value pair is available only when the build option `RM_OPT_EXECUTECODE` is set to `TRUE` (see *RM_OPT_EXECUTECODE* on page 420).

`RM_Cap_MemoryDescriptorPtr (0x00000016)`

Pointer to the memory map data structure. This tag-value pair is available only when the build option `RM_OPT_MEMORYMAP` is set to `TRUE` (see *RM_OPT_MEMORYMAP* on page 426).

`RM_Cap_SDMInfoPtr (0x00000017)`

Pointer to the *Self-Describing Module* (SDM) information string (see *rm_SDM_Info* on page 623). This tag-value pair is available only when the build option `RM_OPT_SDM_INFO` is set to `TRUE` (see *RM_OPT_SDM_INFO* on page 426).

`RM_Cap_EndOfTable (0x00000000)`

This is always the last tag-value pair in the capabilities table.

6.3 Control and monitoring structures

These data structures and arrays are used to access or manage coprocessors, registers, or executable code:

- *rm_ExecuteCodeBuffer*
- *RM_ExecuteCodeBlock* on page 619
- *RM_Registers* on page 620
- *RM_RegisterAccess* on page 620
- *RM_Statistics* on page 621
- *rm_MemoryMap* on page 622
- *rm_SDM_Info* on page 623.

6.3.1 rm_ExecuteCodeBuffer

This array is used by the host to access coprocessors, without the need for coprocessor support code to be built permanently into the target.

Caution

The contents of the `rm_ExecuteCodeBuffer` array is read as instructions, and is both read and written as data. This causes problems on processors like the ARM966E-S that contain tightly-coupled SRAM, because locations in the SRAM can only be accessed as instructions or data, but not both. On these processors, you must be careful to ensure that the `rm_ExecuteCodeBuffer` is located in off-chip RAM.

The host downloads a series of ARM instructions and data into the buffer, which RealMonitor then executes when it receives an ExecuteCode opcode.

Syntax

```
    AREA   AREA_ExecuteCodeBuffer, CODE, READWRITE
    EXPORT rm_ExecuteCodeBuffer
rm_ExecuteCodeBuffer
    NOP
    SPACE  ((RM_EXECUTECODE_SIZE-1) * 4)
```

Implementation

The size of the buffer is determined by the `RM_EXECUTECODE_SIZE` build option (see *RM_EXECUTECODE_SIZE* on page 425). However, to prevent the linker from generating a warning about branches into data symbols, a NOP instruction must be inserted into the top of the buffer. To compensate for the NOP instruction, the remainder of the buffer, as defined using the `SPACE` directive, must be reduced by four bytes.

The following example shows how to read a coprocessor register using a *Move ARM Register from Coprocessor* (MRC) instruction. The code must be downloaded into the array using normal write-to-memory packets. After sending an ExecuteCode packet, the coprocessor value can be retrieved by reading from the fourth word in the array:

```
MRC  p15, 0, r0, c0, c0, 0  ; read coprocessor register
STR  r0, [pc, #0]           ; store register value to memory
BX   lr                     ; return
%    4                      ; reg. value will be stored here
```

The following example shows how to write to a coprocessor register using a *Move Coprocessor from ARM Register* (MCR) instruction. The code must be downloaded into the array using normal write-to-memory packets. The fourth word in the buffer must be set to the value that is to be written into the coprocessor. An ExecuteCode packet must then be sent for the code to be executed, and for the value to be written to the coprocessor:

```
LDR  r0, [pc, #4]           ; load value to be written
MCR  p15, 0, r0, c1, c0, 0  ; write value to coprocessor
BX   lr                     ; return
DCD  #value                 ; reg. value must be stored here
```

The `rm_ExecuteCodeBuffer` array is available only when the build option `RM_OPT_EXECUTECODE` is set to `TRUE` (see *RM_OPT_EXECUTECODE* on page 420).

6.3.2 RM_ExecuteCodeBlock

This data structure is pointed to by an entry in the capabilities table (see *rm_CapabilitiesTable* on page 615), and is used by the host to discover the start address and length of the execute code buffer (see *rm_ExecuteCodeBuffer* on page 618).

Note

This structure is available only when the build option `RM_OPT_EXECUTECODE` is set to `TRUE` (see *RM_OPT_EXECUTECODE* on page 420).

Syntax

```
static const RTM_ExecuteCode_Struct RM_ExecuteCodeBlock =
{
    rm_ExecuteCodeBuffer,
    (RM_EXECUTECODE_SIZE * sizeof(uint32))
};
```

6.3.3 RM_Registers

The registers block is used to store copies of various ARM registers when the foreground application is stopped (see *Starting and stopping the foreground application* on page 670). A subset of these registers is restored when the foreground application is restarted.

The register block always contains space for the same number of registers, even if a particular build of RealMonitor does not save and restore the entire set of registers.

Note

The register block is compiled into RMTarget only when `RM_OPT_STOPSTART` value is set to TRUE (see *RM_OPT_STOPSTART* on page 416).

Syntax

```
typedef struct
{
    uint32 r[15];                   /* User mode r0 to r14 */
    uint32 pc;                      /* PC */
    uint32 cpsr;                    /* CPSR */
    uint32 svc_r[2], svc_spsr;      /* SVC mode r13, r14, SPSR */
    uint32 abort_r[2], abort_spsr;  /* Abort mode r13, r14, SPSR */
    uint32 undef_r[2], undef_spsr;  /* Undef mode r13, r14, SPSR */
    uint32 irq_r[2], irq_spsr;      /* IRQ mode r13, r14, SPSR */
    uint32 fiq_r[7], fiq_spsr;      /* FIQ mode r8 to r14, SPSR */
} RM_RegisterBlock;
RM_RegisterBlock RM_Registers;
```

6.3.4 RM_RegisterAccess

The register accessibility block contains pairs of interleaved words, with each word containing a set of flags representing 32 registers. The flags in the first word indicate whether the corresponding register is stored into the registers block when the foreground application is stopped. The flags in the second word indicate whether the corresponding register is restored from the registers block when the foreground application is restarted.

Caution

If read access to the *program counter* (pc) is disabled, you cannot use breakpoints, watchpoints, or SWI semihosting. If write access to the pc is disabled, you cannot use breakpoints.

Note

You must ensure that you do not mark any of the registers as writable, but not readable.

This data structure is compiled into RMTarget only when `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416).

Syntax

```
typedef struct
{
    uint32 reg00_31_read;
    uint32 reg00_31_write;
    uint32 reg32_63_read;
    uint32 reg32_63_write;
} RM_RegisterAccess_Struct;

static const RM_RegisterAccess_Struct RM_RegisterAccess =
{
#if RM_OPT_SAVE_FIQ_REGISTERS
    /* Read all registers. */
    /* Write to R0_USR to R14_USR, PC, CPSR, */
    /* R13_SVC, R14_SVC, SPSR_SVC. */
    0xFFFFFFFF, 0x000FFFFF, 0x0000001F, 0x00000000
#else
    /* Same as above, but without FIQ registers. */
    0x1FFFFFFF, 0x000FFFFF, 0x00000000, 0x00000000
#endif
};
```

6.3.5 RM_Statistics

The statistics data structure is used to hold various counters and flags used for monitoring the internal operation of RealMonitor. All counters are initialized to zero when RealMonitor is first loaded. The counters are then incremented as follows:

`RX_count` Incremented by one each time `RM_IRQHandler()` is called (see *RM_IRQHandler()* on page 643) as the result of a DCC RX interrupt.

`TX_count` Incremented by one each time `RM_IRQHandler()` is called (see *RM_IRQHandler()* on page 643) as the result of a DCC TX interrupt.

`failed_DCC_count`

Incremented by one each time a DCC coprocessor access instruction fails as the result of the assertion of the **nTRST** signal (see *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424 for details).

Note

This data structure is intended primarily for ARM use, and is compiled into RMTarget only when the build option RM_OPT_GATHER_STATISTICS is set to TRUE (see *RM_OPT_GATHER_STATISTICS* on page 425).

Syntax

```
typedef struct
{
    uint32 RX_count;
    uint32 TX_count;
    uint32 failed_DCC_count;
} RM_STATISTICS;

RM_STATISTICS RM_Statistics;
```

6.3.6 rm_MemoryMap

This array describes the memory map of a target system. The memory map is accessed by the host using the RM_Cap_MemoryDescriptorPtr in the capabilities table (see *rm_CapabilitiesTable* on page 615).

The memory map is defined in a board-specific manner using the file rm_memorymap.c. For an Integrator board, the memory map is defined as follows:

- memory in the range 0x00000000 to 0x0FFFFFFF (the core module memory) is readable, writable, and stoppable, and has the same endianness as the processor
- memory in the range 0x20000000 to 0x2FFFFFFF (the core module *External Bus Interface*, or EBI, memory) is readable, writable, and stoppable, and has the same endianness as the processor
- memory in the range 0x80000000 to 0xBFFFFFFF (the core module alias memory) is readable, writable, and stoppable, and has the same endianness as the processor
- all other memory is readable and writable, and does not require cache synchronization.

When the memory map does not specify the attributes of a memory range, the following default attributes apply:

- not readable
- not writable
- little-endian
- not stoppable

- no default access type
- no cache synchronization required.

If the memory of the board has been remapped in a different way than the standard configuration, or you want to add to, or restrict, the existing memory map, you must edit the file `rm_memorymap.c`.

Note

This array is compiled into RMTarget only when the build option `RM_OPT_MEMORYMAP` is set to `TRUE` (see *RM_OPT_MEMORYMAP* on page 426).

Syntax

```
const uint32 rm_MemoryMap[];
```

6.3.7 rm_SDM_Info

This is a string that allows the *Remote Debug Interface* (RDI) module server on the host to determine which SDMs must be used. The SDMs provide the debugger with additional information about the processor and board that are being debugged.

The string is defined in `rm_sdm.c` (see *RMTarget source files* on page 27), and is made available to the host using the `RM_Cap_SDMInfoPtr` in the capabilities table (see *rm_CapabilitiesTable* on page 615). The contents of the string are determined at compile time using the board and processor build options (see *RM_PROCESSOR_NAME* on page 423 and *RM_PROCESSOR_REVISION* on page 423). For example, when using an ARM966E-S processor and an Integrator board, the string contains the following:

```
"ARM966E-S\n0\nCM966E-S + Integrator"
```

Syntax

```
const char rm_SDM_Info[];
```


6.4 Assembly language macros

This section describes the low-level assembly language macros:

- *RM_IRQ_GETSTATUS*
- *RM_IRQ_ENABLE*
- *RM_IRQ_DISABLE* on page 625
- *RM_DCC_READWORD* on page 625
- *RM_DCC_WRITEWORD* on page 626
- *RM_DCC_GETSTATUS* on page 627
- *RM_DEBUG_READ_MOE* on page 628
- *RM_DEBUG_ENABLE* on page 629.

6.4.1 RM_IRQ_GETSTATUS

This is a board-specific macro that is defined in `rm_comms.s`. The status is stored into the register specified by *`$status`*, and can be tested by ANDing it with one of the following macros:

`RM_IRQ_RX` To determine whether the COMMRX interrupt bit is asserted.

`RM_IRQ_TX` To determine whether the COMMTX interrupt bit is asserted.

`RM_IRQ_RXTX` To determine whether both the COMMRX and COMMTX interrupt bits are asserted.

Syntax

`RM_IRQ_GETSTATUS` *`$status`*

where:

`$status` The name of an ARM register that is used to return the status of the DCC interrupts in the interrupt controller.

6.4.2 RM_IRQ_ENABLE

This is a board-specific macro that is defined in `rm_comms.s`. This macro enables, in the interrupt controller, each of the interrupts specified in the register *`$enable`*. The possible values for this register are:

`RM_IRQ_NONE` To leave the interrupt enable status unchanged.

`RM_IRQ_RX` To enable the COMMRX interrupt.

`RM_IRQ_TX` To enable the COMMTX interrupt.

RM_IRQ_RXTX To enable both the COMMRX and COMMTX interrupts.

Syntax

RM_IRQ_ENABLE *$enable*, *$temp*

where:

$enable The name of an ARM register that specifies which DCC interrupts to enable.

$temp The name of an ARM register that can be used to store temporary data.

6.4.3 RM_IRQ_DISABLE

This is a board-specific macro that is defined in rm_comms.s. This macro disables, in the interrupt controller, each of the interrupts specified in the register *$disable*. The possible values for this register are:

RM_IRQ_NONE To leave the interrupt enable status unchanged.

RM_IRQ_RX To disable the COMMRX interrupt.

RM_IRQ_TX To disable the COMMTX interrupt.

RM_IRQ_RXTX To disable both the COMMRX and COMMTX interrupts.

Syntax

RM_IRQ_DISABLE *$disable*, *$temp*

where:

$disable The name of an ARM register that specifies which DCC interrupts to disable.

$temp The name of an ARM register that can be used to store temporary data.

6.4.4 RM_DCC_READWORD

This is a processor-specific macro to read a single word of data from the DCC.

Caution

This macro must be executed from within any privileged mode, with the exception of Undef mode. It must not be executed in Undef mode because of the problem that arises when the **nTRST** signal is asserted. See *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424 for a complete description of this behavior.

The data read from the DCC is placed into the register specified by *$reg*. If there is no data to read, an undefined value is returned.

The macro expects to be executed from within any privileged processor mode, and preserves all registers.

Syntax

```
RM_DCC_READWORD $reg
```

where:

$reg The register into which the single word of data read from the DCC is placed.

6.4.5 RM_DCC_WRITEWORD

This is a processor-specific macro to write a single word of data to the DCC.

Caution

This macro must be executed from within any privileged mode, with the exception of Undef mode. It must not be executed in Undef mode because of the problem that arises depending on when the **nTRST** signal is asserted. See *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424 for a complete description of this behavior.

The data to be written is contained in the register specified by *$reg*. If the DCC write buffer already contains data, the value written to the DCC is undefined.

The macro expects to be executed from within any privileged processor mode, and preserves all registers.

Syntax

```
RM_DCC_WRITEWORD $reg
```


where:

$reg The register that contains the word of data that is to be written to the DCC.

6.4.6 RM_DCC_GETSTATUS

This is a processor-specific macro to return the status of the DCC.

Caution

This macro must be executed from within any privileged mode, with the exception of Undef mode. It must not be executed in Undef mode because of the problem that arises depending on when the **nTRST** signal is asserted. See *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424 for a complete description of this behavior.

The status is written to the register specified by *$reg*, and can mean one of the following:

- `($reg AND RM_DCC_RBIT)`

  where:

  zero Read register is empty.

  nonzero Read register is full.

- `($reg AND RM_DCC_WBIT)`

  where:

  zero Write register is free.

  nonzero Write register is busy.

  Note

  If you are using an ARM10 processor, the meanings of these values are reversed. This is indicated by the value of `RM_DCC_WBIT_SENSE` (`TRUE` on an ARM10), which is processor-dependent. You must consider this when attempting to port RMTarget to a new processor (see *Porting to a new processor* on page 211).

This macro must be executed from within a privileged processor mode, with the exception of Undef mode. This macro must not be executed in Undef mode because of the possibility of the **nTRST** problem (see *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424).

Syntax

```
RM_DCC_GETSTATUS $reg
```

6.4.7 RM_DEBUG_READ_MOE

This is a processor-specific macro that determines the *Method Of Entry* (MOE) for the immediately preceding Data Abort or Prefetch Abort.

This macro must be called only once for each abort. If it is called more than once for the same abort, the value returned in the second and subsequent calls is undefined.

The macro expects to be executed from within any privileged processor mode, and preserves all registers.

Syntax

```
RM_DEBUG_READ_MOE $result, $temp
```

where:

$result The name of a register that will be used to store the method of entry.

$temp The name of a register that can be used to store temporary data.

The MOE is written to the register specified by *$result*, and the following assembly language definitions must be used to compare against it:

`RM_MOE_HARDBREAK`

A hardware breakpoint has been triggered.

`RM_MOE_HARDWATCH`

A watchpoint has been triggered.

`RM_MOE_BKPT`

A `BKPT` instruction has been reached.

`RM_MOE_DATAABORT`

A D-side abort has occurred (Data Abort).

`RM_MOE_PREFETCHABORT`

An I-side abort has occurred (Prefetch Abort).

Return value

On processors where the MOE cannot be determined exactly, some of the definitions of this macro map onto the same return value. In these cases, the return value always indicates the base exception. Therefore, hardware and software breakpoints are reported as Prefetch Aborts, and watchpoints are reported as Data Aborts.

6.4.8 RM_DEBUG_ENABLE

This is a processor-specific macro that enables the debugging hardware in the processor. On processors such as the ARM966E-S, this macro does nothing, because the debugging hardware is always enabled. On processors like the ARM1020T, the debugging hardware is disabled by default (to conserve power), and must be enabled by setting the Global Debug Enable bit in the *Debug Status and Control Register* (DSCR).

This macro must be executed from within a privileged processor mode.

Syntax

```
RM_DEBUG_ENABLE $temp
```

where:

$temp The name of a register that can be used to store temporary data.

6.5 Initialization functions

This section describes the functions that are used to initialize the RealMonitor state, the communications channel, and the exception handlers:

- *RM_Init()*
- *rm_InitCommsState()*
- *RM_InitVectors()* on page 632.

6.5.1 RM_Init()

This function initializes the internal state of RealMonitor, and then enables the COMMRX and COMMTX interrupts in the interrupt controller. It must be called once after the exception handlers have been installed, and can be called at any future time to reset the state. When this function is called, the following variable is set:

```
IMP_GlobalState->state = RM_State_Running
```

This indicates that RealMonitor is running, which must always be the case when RealMonitor is initialized.

A call is also made to `rm_InitCommsState(IMP_ESCAPE_RESET)` to initialize the communications link, and to synchronize with the host.

This function must be called from a privileged processor mode.

Syntax

```
void RM_Init(void)
```

6.5.2 rm_InitCommsState()

This function initializes the state pertaining to the communications link. By keeping this initialization code separate from `RM_Init()`, you can reset only the communications link, without affecting the rest of RealMonitor (see *RM_Init()*).

In particular, rm_InitCommsState() sets the variables described in Table 6-1.

Table 6-1 rm_InitCommsState() variables

Variable	Resulting action
IMP_GlobalState->rxproc = rm_NOP	All incoming data is discarded, but escape sequence processing continues.
IMP_GlobalState->txproc = imp_TX	Allow data to be transmitted.
IMP_GlobalState->txpending = NULL	There is no RealMonitor packet waiting to be transmitted.
IMP_GlobalState->stop_code = NULL	There is no stop code awaiting transmission.
IMP_GlobalState->exception_flag = 0	Any read or write memory opcode currently being processed is abandoned. Treat Data Aborts as software breakpoints.
IMP_GlobalState->rx_escape_flag = 0	No escape sequence is currently being received.
IMP_GlobalState->tx_flag = 1 IMP_GlobalState->tx_buffer = IMP_ESCAPE IMP_GlobalState->tx_escape_data = reset_type	Send an escape sequence to the host. The parameter to this function is used as the escape value.
IMP_GlobalState->error_block[0] = IMP_Header(IMP_Chan_RTM,RM_Msg_Okay,0x0004) IMP_GlobalState->error_block[1] = RM_Error_None	No error has occurred.

Syntax

void rm_InitCommsState(uint32 *reset_type*)

where:

reset_type

The type of reset that needs to be performed, which can be either of the following:

- IMP_ESCAPE_RESET
- IMP_ESCAPE_GO.

6.5.3 RM_InitVectors()

This function installs the exception handlers required by RealMonitor. It causes any existing entries in the processor exception vector table to be overwritten. If the function is being used on a Harvard-architecture processor (with split ICaches and DCaches), the processor caches are synchronized using either the function `rm_Caches_SyncRegion()` or `rm_Cache_SyncVA()` (see *rm_Cache_SyncRegion()* on page 653 and *rm_Cache_SyncVA()* on page 654). The installation of exception handlers is dependent on the build options you set:

Undef This handler is installed only if either of the build options `RM_OPT_SOFTBREAKPOINT` or `RM_PROCESSOR_NEEDS_NTRST_FIX` is set to `TRUE` (see *RM_OPT_SOFTBREAKPOINT* on page 417 and *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424).

SWI This handler is installed only if the build option `RM_OPT_SEMIHOSTING` is set to `TRUE` (see *RM_OPT_SEMIHOSTING* on page 418).

Prefetch Abort

This handler is installed only if the build option `RM_OPT_HARDBREAKPOINT` is set to `TRUE` (see *RM_OPT_HARDBREAKPOINT* on page 417).

Data Abort This handler is installed only if the build option `RM_OPT_HARDWATCHPOINT` is set to `TRUE` (see *RM_OPT_HARDWATCHPOINT* on page 417).

IRQ This handler is installed only if the build option `RM_OPT_USE_INTERRUPTS` is set to `TRUE` (see *RM_OPT_USE_INTERRUPTS* on page 427).

This function must be called from a privileged processor mode.

Syntax

```
void RM_InitVectors(void)
```


6.6 μHAL interfacing functions

This section describes the functions that allow you to use RealMonitor with μHAL:

- *RM_uHAL_Init()*
- *rm_uHAL_IRQTest()*
- *rm_uHAL_IRQHandler()* on page 634.

6.6.1 RM_uHAL_Init()

Applications that are μHAL-based must call this function to initialize RealMonitor, rather than calling `RM_Init()`. This function installs exception handlers and initializes interrupts in a manner that is compatible with the μHAL method of interrupt handling. For details on linking with μHAL, see *Linking with μHAL* on page 512.

Syntax

```
void RM_uHAL_Init(void)
```

6.6.2 rm_uHAL_IRQTest()

This function tests whether a DCC interrupt is pending. It is called by the chaining library to test whether RealMonitor wants to claim the IRQ.

Syntax

```
int rm_uHAL_IRQTest(void *lr)
```

where:

lr The address of the code where the interrupt occurred.

Return value

Returns one of the following:

1 `TEST_CLAIM`. There is a DCC interrupt pending to be handled by RealMonitor. This value instructs the chaining library to pass control to the RealMonitor interrupt handler so that it can process the interrupt.

2 `TEST_SKIP`. There is no DCC interrupt pending. This value instructs the chaining library to offer the interrupt to the next handler in the chain.

6.6.3 rm_uHAL_IRQHandler()

This function provides an interrupt handler that is installed by `RM_uHAL_Init()` if chaining is not in use (see *RM_uHAL_Init()* on page 633). It passes DCC interrupts to the function `RM_IRQHandler2()` (see *RM_IRQHandler2()* on page 644). It passes any other interrupts to the µHAL low-level handler.

Syntax

```
rm_uHAL_IRQHandler(void)
```


6.7 Exception handling functions

This section describes the functions that are called directly from the exception vector table of the processor, or by a chained exception handler. They must not be called in any other context. The exception handling functions are:

- *RM_InstallVector()* on page 636
- *rm_GetExceptionTableBase()* on page 637
- *RM_UndefHandler()* on page 638
- *rm_nTRST_Fix()* on page 639
- *RM_SWIHandler()* on page 640
- *RM_PrefetchAbortHandler()* on page 641
- *RM_DataAbortHandler()* on page 642
- *RM_IRQHandler()* on page 643
- *RM_IRQHandler2()* on page 644
- *rm_RunningToStopped()* on page 646
- *rm_Common_RunningToStopped()* on page 647
- *rm_RestoreUndefAndReturn()* on page 648
- *rm_ExceptionDuringProcessing()* on page 649
- *rm_Panic()* on page 651.

The exception handlers are installed by calling `RM_InitVectors()` (see *RM_InitVectors()* on page 632). Alternatively, your application can install the RealMonitor exception handlers itself. For details on installing exception handlers, see *Handling exceptions* on page 55.

Note

RealMonitor makes no use of FIQs, so the application is expected to provide its own FIQ handler if necessary.

6.7.1 RM_InstallVector()

This function installs an exception handler into the processors exception vector table, and returns the address of the previous exception handler.

Syntax

`uint32 *RM_InstallVector(uint32 vector, uint32 *handler)`

where:

vector The number of the vector that is to be installed, which can be any of the following:

- **0** Reset.
- **1** Undefined instruction.
- **2** Software interrupt.
- **3** Prefetch Abort.
- **4** Data Abort.
- **6** IRQ.
- **7** FIQ.

handler The address of the new vector that is to be installed.

Return value

Returns the address of the previous exception vector, or NULL if the new exception vector could not be installed.

Implementation

New exception handlers are always installed using the same method as any existing exception handler, which can be one of the following:

The `B <address>` **instruction**

The existing branch instruction in the exception vector table is examined to obtain the address of the previous exception handler. The existing branch instruction is then overwritten with a new branch instruction that points to the new exception handler. On processors with a Harvard cache, the writing of a branch instruction is followed by a synchronization of caches.

With this method, the exception table must reside in RAM.

The `LDR PC, [PC, #<offset>]` instruction

The existing load instruction in the exception vector table is examined to obtain the address of the memory location that this instruction tries to access. The memory location is read to obtain the address of the previous exception handler, and is then written with the address of the new exception handler.

With this method, the exception table can reside in ROM, but the address pointed to by the load instruction must reside in RAM.

On processors that support relocatable exception tables, such as the ARM720 and ARM920, coprocessor 15 is examined to determine the current exception table base address. Address `0x00000000` is used as the base address on processors that do not support relocatable exception tables.

This function must always be called from a privileged processor mode.

6.7.2 rm_GetExceptionTableBase()

This processor-specific function returns the base address of the exception table. It must be called from within a privileged processor mode.

Syntax

```
__inline uint32 *rm_GetExceptionTableBase(void)
```

Return value

Returns the base address of the exception table.

Implementation

On processors with a fixed exception table, this function always returns `0x00000000`.

On processors with a relocatable exception table, this function returns either `0x00000000` or `0xFFFF0000`, depending on the exception table setting in CP15.

6.7.3 RM_UndefHandler()

This is the entry point into the RealMonitor Undef exception handler. This entry point is designed to be branched directly from the exception table of the processor. That is, it can be used when the application does not provide any Undef handlers of its own.

Syntax

`RM_UndefHandler`

where, on entry, the registers must contain the following:

`lr` Address of the undefined instruction + 4.

`SPSR` `CPSR` from the previous mode.

`CPSR` Undef mode, with IRQs disabled.

Implementation

On entry to the function the registers r3, r4, ip, and lr are pushed onto the stack, and register r4 is set to contain the address of the `rm_State` data structure (see *State* on page 66).

Depending on the cause of the exception, there are three possible actions to be taken:

- If the build option `RM_PROCESSOR_NEEDS_NTRST_FIX` is set to `TRUE`, the instruction that caused the exception (at address lr-4) is examined. If this instruction corresponds to an MRC or MCR using coprocessor 14, the **nTRST** problem been detected, and a branch is made to `rm_nTRST_Fix()` (see *rm_nTRST_Fix()* on page 639).
- If the variable `IMP_GlobalState->exception_flag` is nonzero, the undefined instruction occurred during the processing of a RealMonitor packet (see *IMP_GlobalState* on page 69). The value of lr that is stored on the stack is modified to point to the instruction that caused the exception, and the function `rm_ExceptionDuringProcessing()` is called, passing the adjusted value of lr in register r3.
- The undefined instruction is assumed to have been caused either by an error in the foreground application, or by a software breakpoint that the debugger has set. A branch is made to `rm_RunningToStopped()`, with an Undef packet header placed in register r3 (see *rm_RunningToStopped()* on page 646).

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_SOFTBREAKPOINT` or `RM_PROCESSOR_NEEDS_NTRST_FIX` is set to `TRUE` (see *RM_OPT_SOFTBREAKPOINT* on page 417 and *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424).

6.7.4 rm_nTRST_Fix()

This function is called when an undefined instruction occurs that was caused by the **nTRST** problem (described in *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424). Register r13 must be set up as shown in Figure 6-2.

	Stack
r13[6]	Register lr
r13[5]	Register ip
r13[4]	Register r4
r13[3]	Register r3
r13[2]	Register r2
r13[1]	Register r1
r13[0]	Register r0

Figure 6-2 Format of the stack

Syntax

`RM_nTRST_Fix`

where, on entry, the registers must contain the following:

`r4` Pointer to the `rm_State` data structure (see *State* on page 66).

`r13[0]` Saved value of r0.

`r13[1]` Saved value of r1.

`r13[2]` Saved value of r2.

`r13[3]` Saved value of r3.

`r13[4]` Saved value of r4.

`r13[5]` Saved value of ip.

r13[6]	Saved value of lr (a pointer to the instruction that caused the exception).
SPSR	CPSR from the previous mode.
CPSR	Undef mode, with IRQs disabled.

Return value

Returns the following:

r1	The value 2.
r3-r4, IP	The corresponding values popped off the stack.
CPSR	Value of SPSR on entry.

Implementation

To avoid undefined instructions from occurring continuously while **nTRST** is asserted, the transmission of data must be disabled. This is achieved by setting the variable IMP_GlobalState->tx_flag to zero (see *IMP_GlobalState* on page 69).

The function then returns to the instruction directly following the one that caused the exception. Registers r3, r4, and ip are popped from the stack, and r1 is set to contain the value 0x02. This value must be returned in r1 so that, if the undefined instruction was attempting to read from the DCC status register, the status appears as if no reads or writes are currently possible.

Note

This function is only compiled into RMTarget when the build option RM_PROCESSOR_NEEDS_NTRST_FIX is set to TRUE (see *RM_PROCESSOR_NEEDS_NTRST_FIX* on page 424).

6.7.5 RM_SWIHandler()

This provides the entry point into the RealMonitor SWI exception handler. This entry point is designed to be branched directly from the exception table of the processor. That is, it can be used when the application does not provide any SWI handlers of its own.

Syntax

```
RM_SWIHandler
```


where, on entry, the registers must contain the following:

`lr` Address of the SWI instruction + 4.

`SPSR` `CPSR` from the previous mode.

`CPSR` Supervisor mode, with IRQs disabled.

Implementation

On entry to the function, registers r3, r4, ip, and lr are pushed onto the stack, and register r4 is set up to contain the address of the `rm_State` data structure (see *State* on page 66). A branch is then made to `rm_RunningToStopped()`, with a SWI packet header placed in register r3.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_SEMIHOSTING` is set to `TRUE` (see *RM_OPT_SEMIHOSTING* on page 418).

6.7.6 RM_PrefetchAbortHandler()

This provides the entry point into the RealMonitor Prefetch Abort exception handler. This entry point is designed to be branched directly from the exception table of the processor. That is, it can be used when the application does not provide any Prefetch Abort handlers of its own.

Syntax

```
RM_PrefetchAbortHandler
```

where, on entry, the registers must contain the following:

`lr` Address of the aborted instruction + 4.

`SPSR` `CPSR` from the previous mode.

`CPSR` Abort mode, with IRQs disabled.

Implementation

On entry to this function, register lr is decremented by four so that it points to the address of the instruction that caused the exception, then registers r3, r4, ip, and lr are pushed onto the stack. Register r4 is set to contain the address of the `rm_State` data structure (see *State* on page 66).

There are two possible causes for a Prefetch Abort that RealMonitor must distinguish:

- If the variable `IMP_GlobalState->exception_flag` has a nonzero value, an abort has occurred during the processing of a packet. The only situation when this is expected to occur is when an ExecuteCode packet is being processed. This is handled by branching to `rm_ExceptionDuringProcessing()`, with the address of the aborted instruction in r3 (see *rm_ExceptionDuringProcessing()* on page 649).
- The Prefetch Abort is assumed to have been caused by one of the following:
  — a hardware breakpoint being triggered
  — a `BKPT` instruction being executed
  — an error in the application being debugged.

  This is handled by branching to `rm_RunningToStopped()`, with an appropriate packet header placed in r3. This results in the host being notified, and the foreground application being stopped. The packet header is determined using the assembly language macro `RM_DEBUG_READ_MOE` (see *RM_DEBUG_READ_MOE* on page 628):

  `HardBreak` If a hardware breakpoint was triggered.

  `SoftBreak` If a software breakpoint was triggered.

  `PrefetchAbort`
  For all other causes.

6.7.7 RM_DataAbortHandler()

This provides the entry point into the RealMonitor Data Abort exception handler. This entry point is designed to be branched directly from the exception table of the processor. That is, it can be used when the application does not provide any Data Abort handlers of its own.

Syntax

`RM_DataAbortHandler`

where, on entry, you must set the registers to contain the following:

`lr` Address of the aborted instruction + 8.

`SPSR` `CPSR` from the previous mode.

`CPSR` Abort mode, with IRQs disabled.

Implementation

On entry to this function, register lr is decremented by eight so that it points to the address of the instruction that caused the exception, then registers r3, r4, ip, and lr are pushed onto the stack. Register r4 is set to contain the address of the `rm_State` data structure (see *State* on page 66).

There are two possible causes for a Data Abort that RealMonitor must distinguish:

- If the variable `IMP_GlobalState->exception_flag` has a nonzero value, an abort has occurred during the processing of a packet. The only situation when this is expected to occur is when a read/write packet accesses an invalid or restricted area of memory, or when an ExecuteCode packet is being processed. This is handled by branching to `rm_ExceptionDuringProcessing()`, with the address that caused the Data Abort in r3 (see *rm_ExceptionDuringProcessing()* on page 649).
- The Data Abort is assumed to have been caused by either a hardware watchpoint being triggered, or due to an error in the application being debugged.

  This is handled by branching to `rm_RunningToStopped()`, with an appropriate packet header placed in r3. This results in the host being notified, and the foreground application being stopped. The packet header is determined using the assembly language macro `RM_DEBUG_READ_MOE` (see *RM_DEBUG_READ_MOE* on page 628):

  `HardWatch` If a hardware watchpoint was triggered.

  `DataAbort` For all other causes.

6.7.8 RM_IRQHandler()

This function provides the first entry point into the RealMonitor IRQ exception handler. The IRQ handler is responsible for processing DCC data quickly when RealMonitor is operating with interrupts enabled. The entry point is designed to be branched directly from the exception table of the processor. That is, it can be used when the application does not provide any IRQ handlers of its own.

The COMMRX and COMMTX interrupt sources are signaled when the DCC has received, or is ready to transmit, data. As RealMonitor changes state, it enables or disables the appropriate interrupt sources.

The COMMRX and COMMTX interrupts are disabled when RealMonitor is in the stopped state, forcing RealMonitor to poll the DCC. This is necessary because the IRQ bit in the processor status register is in an unknown state when RealMonitor stopped, and it is not safe for RealMonitor to re-enable IRQs if the application has disabled them.

When the foreground application is running, RealMonitor leaves the COMMRX interrupt enabled. This ensures that the host can initiate a synchronization, irrespective of what the target is currently doing. The COMMTX interrupt is enabled when RealMonitor has data to transmit.

Both interrupt sources are disabled when the interrupt handler starts running, and the appropriate source, or sources, is re-enabled when the interrupt handler returns. The function `rm_RestoreUndefAndReturn()` restores control to the foreground application (see *rm_RestoreUndefAndReturn()* on page 648).

Syntax

`RM_IRQHandler`

where, on entry, the registers must contain the following:

`lr` Address of the next instruction to be executed + 4.

`SPSR` `CPSR` from the previous mode.

`CPSR` IRQ mode, with IRQs disabled.

Implementation

On entry to the function, register lr is decremented by four so that it points to the address of the next instruction, and then registers ip and lr are pushed onto the stack.

The status of the interrupt controller is examined, and if neither the COMMRX nor COMMTX interrupts is asserted, execution falls through to an unspecified user code. By default, the user code simply returns to the interrupted task.

Otherwise, if either the COMMRX or COMMTX interrupts was asserted, and no other interrupts were asserted, execution then falls through to the function `RM_IRQHandler2()` (see *RM_IRQHandler2()*).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_USE_INTERRUPTS` is set to `TRUE` (see *RM_OPT_USE_INTERRUPTS* on page 427).

6.7.9 RM_IRQHandler2()

This function provides a second entry point to the RealMonitor IRQ exception handler. This entry point is intended to be used when another IRQ handler must chain onto the beginning of the RealMonitor IRQ handler, having already stacked ip and lr.

Syntax

`RM_IRQHandler2`

where, on entry, you must set the registers to contain the following:

`sp[0]` Saved value of ip.

`sp[1]` Address of the next instruction to be executed.

`lr` Address of the next instruction to be executed.

`SPSR` `CPSR` from the previous mode.

`CPSR` IRQ mode, with IRQs disabled.

Return value

`CPSR` Value of `SPSR` on entry.

Implementation

To minimize interrupt latency, this function re-enables IRQs as soon as possible by performing the following actions:

1. Disables the COMMRX and COMMTX interrupt sources to prevent the interrupt handler from being called again when processor interrupts are enabled.
2. Saves registers r0-r4 and sp into the registers block (see *RM_Registers* on page 620).
3. Switches into Undef mode, with IRQs enabled in the processor.

These steps enable the processor to handle interrupts from sources other than the DCC, while RealMonitor continues to process DCC traffic. After these steps are performed, the Undef mode registers r13, r14, and spsr are saved into the registers block, and register r4 is set to contain the address of the `rm_State` data structure (see *State* on page 66).

A check is performed to determine whether RealMonitor was in the running state when the IRQ occurred. If not, a COMMRX or COMMTX IRQ has occurred while the foreground application was already stopped, and RealMonitor enters the panic state by branching to `rm_Panic` (see *rm_Panic()* on page 651).

To send or receive a word of data, the function `rm_Poll()` is called (see *rm_Poll()* on page 661). If, on returning from this function, the `IMP_GlobalState->state` variable indicates that RealMonitor is still in the running state, a branch is made to

rm_RestoreUndefAndReturn() (see *rm_RestoreUndefAndReturn()* on page 648). Otherwise, the foreground application must be stopped, which is performed by branching to rm_RunningToStopped() (see *rm_RunningToStopped()*).

Note

This function is compiled into RMTarget only when the build option RM_OPT_USE_INTERRUPTS is set to TRUE (see *RM_OPT_USE_INTERRUPTS* on page 427).

6.7.10 rm_RunningToStopped()

This function is called when it is necessary to stop the foreground application. It must be called from the same privileged mode that caused RealMonitor to stop, and the *stack pointer* (sp) register must be set up as shown in Figure 6-3.

Figure 6-3 Format of the stack when stopping

Syntax

rm_RunningToStopped

where, on entry, registers must contain the following:

r3 — Stop code. A packet header for the RealMonitor channel, with an opcode explaining why RealMonitor is stopping, and with a length of zero.

r4 — Pointer to rm_State.

sp[0] — Saved value of r3.

sp[1] — Saved value of r4.

sp[2] — Saved value of ip.

sp[3] — Saved value of lr.

SPSR — CPSR from the previous mode.

CPSR — Any privileged mode, with IRQs disabled.

Implementation

A check is made to see whether RealMonitor was in the running state when it was stopped. If RealMonitor was not in the running state when it stopped, an error has occurred from which RealMonitor cannot recover, and a branch is made to `rm_Panic()` (see *rm_Panic()* on page 651). The parameter header is stored into `IMP_GlobalState->stop_code` for future reference, and `IMP_GlobalState->state` is set to `RM_State_Stopped`. The communications link COMMRX and COMMTX interrupts are disabled in the interrupt controller to prevent any possibility of the RealMonitor IRQ handler being re-entered when IRQs in the processor are later enabled.

Finally, the state of the processor is stored into the register block using the following method (see *RM_Registers* on page 620):

1. Store `R0_USER` to `R2_USER` into the registers block.
2. Remove r3, r4, and ip from the stack, and store them into the registers block.
3. Store `R3_USER` to `R14_USER` using a banked *STore Multiple* (STM) instruction (these will not have been corrupted because the foreground application was interrupted, and can therefore be saved directly).
4. Remove lr from the stack, and store it into the registers block.
5. Store lr and spsr into the registers block.

After the state is saved, RealMonitor switches into Undef mode, and re-enables IRQs if they were enabled prior to RealMonitor being stopped. Execution then falls through to the function `rm_Common_RunningToStopped()`.

6.7.11 rm_Common_RunningToStopped()

This function loops through each of the processor modes and saves sp, lr, and spsr into the registers block (see *RM_Registers* on page 620).

Syntax

```
rm_Common_RunningToStopped
```

where, on entry, registers must contain the following:

`r5` Number of the last processor mode that must be saved.

Implementation

The order in which the modes must be iterated through is the same order that the registers are stored into the registers block, as follows:

1. SVC mode.
2. Abort mode.
3. Undef mode.
4. IRQ mode.

The value in r5 is a mode number that is used to determine the last set of registers to be saved. For example, if r5 is `0x17`, which is the mode number for Abort, only the registers for SVC and Abort mode are saved.

If the build option `RM_OPT_SAVE_FIQ_REGISTERS` is set to `TRUE`, the FIQ registers r8-14 and spsr are saved into the registers block (see *RM_OPT_SAVE_FIQ_REGISTERS* on page 418).

Finally, this function:

1. Switches into Undef mode.
2. Enables IRQs if they were enabled when the exception occurred.
3. Branches to `rm_StoppedLoop()` (see *rm_StoppedLoop()* on page 670).

6.7.12 rm_RestoreUndefAndReturn()

This function restores various registers from the registers block, and restarts the foreground application (see *RM_Registers* on page 620).

Syntax

```
rm_RestoreUndefAndReturn
```

where, on entry, registers must contain the following:

`r3` Pointer to the registers block.

`CPSR` Undef mode. IRQs might be enabled.

On exit, the foreground application is restarted.

Implementation

While in Undef mode, the following registers are restored:

- `sp_undef` = `RM_Registers.undef_r[0]`
- `lr_undef` = `RM_Registers.undef_r[1]`
- `SPSR_undef` = `RM_Registers.undef_spsr`
- `ip` = `RM_Registers.r[12]`

It then switches to IRQ mode with IRQs disabled. If the build option `RM_OPT_USE_INTERRUPTS` is set to `TRUE`, it re-enables COMMRX and COMMTX in the interrupt controller, if necessary (see *RM_OPT_USE_INTERRUPTS* on page 427). COMMRX is always enabled. COMMTX is enabled only if the `IMP_GlobalState->tx_flag` variable is not equal to zero.

It then restores the following registers:

- `r0` = `RM_Registers.r[0]`
- `r1` = `RM_Registers.r[1]`
- `r2` = `RM_Registers.r[2]`
- `r3` = `RM_Registers.r[3]`
- `SPSP_irq` = `RM_Registers.cpsr`
- `pc` = `RM_Registers.pc`

The last register transfer causes the foreground application to restart.

6.7.13 rm_ExceptionDuringProcessing()

This function handles Prefetch Abort, Data Abort, and undefined instruction exceptions that occur during the processing of a packet. These exceptions are expected only to occur during the processing of read memory, write memory, or ExecuteCode opcodes.

Syntax

```
rm_ExceptionDuringProcessing
```

where, on entry, registers must contain the following:

`r0`	Address to be stored into the error block.
`r4`	Pointer to the `rm_State` data structure (see *State* on page 66).
`r13[0]`	Saved value of r0.
`r13[1]`	Saved value of r1.
`r13[2]`	Saved value of r2.

r13[3]	Saved value of r3.
r13[4]	Saved value of r4.
r13[5]	Saved value of ip.
r13[6]	Saved value of lr (the address of the instruction that caused the exception).
SPSR	CPSR from the previous mode.
CPSR	Any privileged mode.

Return value

Returns one of the following:

r0-r4, IP	The corresponding values popped off the stack.
CPSR	Value of SPSR on entry.

Implementation

The function performs the following actions:

1. Sets IMP_GlobalState->operation_aborted to 1 to indicate that an exception has occurred during the processing of a packet (see *IMP_GlobalState* on page 69).
2. Stores an Error packet header (0x00810000) into pending_data so that when the response is sent back to the host, the host can detect that an error has occurred.
3. Sets up the error block, so that the host can determine the cause of the error:
   ```
   error_block[0] = 0x00800008
   error_block[1] = 0x00000002
   error_block[2] = r0
   ```
4. Restores r0–r4, ip, and lr from the stack.
5. Adds four to lr so that it points to the instruction after the one that caused the exception.
6. Returns to the mode that caused the exception (pc=lr, CPSR=SPSR).

6.7.14 rm_Panic()

This function is called when an error occurs from which RealMonitor is unable to recover. The errors from which RealMonitor cannot recover are:

- hitting a breakpoint or watchpoint while the foreground application is already stopped
- starting a semihosting SWI request while the foreground application is already stopped.

This function sets `IMP_GlobalState->state` to `RM_State_Panic`, and then attempts to repeatedly send an ESCAPE-PANIC escape sequence over the DCC. At this point, all command processing ceases, and the host is unable to communicate with the target. This ensures that no further errors can occur, and the state of the target is preserved. To find the cause of the error, you must manually debug the target using only your JTAG unit, that is, without using RMHost.

Syntax

```
rm_Panic
```

where, on entry, registers must contain the following:

`r4` Pointer to `rm_State`.

6.8 Cache handling functions

The section describes the functions that allow the processors caches to be manipulated:

- *rm_Cache_SyncAll()*
- *rm_Cache_SyncRegion()* on page 653
- *rm_Cache_SyncVA()* on page 654.

Cache manipulation code is required only on Harvard-architecture processors (with a split ICache and DCache).

You can implement cache manipulation code in either of the following ways:

- Implement the functions `rm_Cache_SyncAll()` and `rm_Cache_SyncRegion()` (see *rm_Cache_SyncAll()* and *rm_Cache_SyncRegion()* on page 653). The host must then send the SyncCaches opcode after each write (or sequence of writes) to memory. You must use this method where synchronizing the caches is a time-consuming operation.
- Implement the function `rm_Cache_SyncVA()` (see *rm_Cache_SyncVA()* on page 654), so that the write-to-memory opcodes can automatically synchronize the caches after each write. You must use this method where the processor supports efficient cache synchronization for a single virtual address.

6.8.1 rm_Cache_SyncAll()

This function contains processor-specific code for ensuring coherency between the ICaches and DCaches.

Syntax

```
void rm_Cache_SyncAll(void)
```

Implementation

This function only has to be implemented when the build option `RM_PROCESSOR_USE_SYNCCACHES` is enabled (see *RM_PROCESSOR_USE_SYNCCACHES* on page 424).

When implemented, this function performs the following actions:

1. Cleans the entire DCache.
2. Flushes the entire ICache.

Note

This function can take a long time (thousands of cycles) to execute on some processors, and can affect interrupt response time adversely.

This function must always be called from within a privileged processor mode.

6.8.2 rm_Cache_SyncRegion()

This function contains processor-specific code for ensuring coherency between the ICaches and DCaches for a specified memory region.

Syntax

```
void rm_Cache_SyncRegion(uint32 address, uint32 length)
```

where:

address The word-aligned start address of the region.

length The length, in bytes, of the region. The length must be a multiple of four, and must be greater than zero.

Implementation

This function only has to be implemented when the build option RM_PROCESSOR_USE_SYNCCACHES is enabled (see *RM_PROCESSOR_USE_SYNCCACHES* on page 424).

When implemented, this function performs the following actions:

1. Clean the specified range in the DCache. An implementation can clean more than the specified region.
2. Flush the specified range in the ICache. An implementation can clean more than the specified region.

Note

This function can take a long time (thousands of cycles) to execute on some processors, and can affect interrupt response time adversely.

This function must always be called from within any privileged processor mode.

6.8.3 rm_Cache_SyncVA()

This function contains processor-specific code for ensuring consistency between (synchronizing) the ICaches and DCaches for a specified virtual address.

Syntax

```
void rm_Cache_SyncVA(uint32 address)
```

where:

address The virtual address to be synchronized between the ICache and the DCache.

Implementation

This function only has to be implemented when the build option RM_PROCESSOR_USE_SYNC_BY_VA is enabled (see *RM_PROCESSOR_USE_SYNC_BY_VA* on page 424).

When implemented, this function performs the following actions:

1. Cleans at least one word from the DCache, starting from the specified virtual address.
2. Flushes at least one word from the ICache, starting from the specified virtual address.

This function must be called from a privileged processor mode.

6.9 Data logging functions

The data logging functions allow data to be sent and received between the host and target in a controlled manner. The data logging functions are:

- *RM_SendPacket()*
- *rm_EmptyFifo()* on page 656
- *rm_NextMessage()* on page 657.

6.9.1 RM_SendPacket()

This function is used to send a packet of data over the communications link to the host. The function returns immediately, although the packet might not actually be sent until some time later. Packets are sent in the order in which this function is called.

Syntax

```
RM_SentCode RM_SendPacket(uint32 const *buffer)
```

where:

buffer A pointer to a word-aligned buffer that contains the information shown in Figure 6-4.

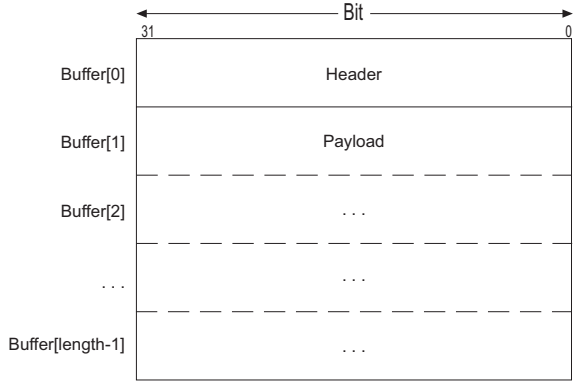

Figure 6-4 Packet buffering

Return value

Returns one of the following:

`RM_Sent` RealMonitor has accepted the buffer, and sends the packet as soon as possible. The application must wait until the length field reaches zero before re-using the buffer.

`RM_FilteredOut`

The packet has been rejected because the channel/opcode has been filtered out.

Note

This is currently unsupported by RealMonitor.

`RM_FIFOFull`

The packet has not been sent because there is insufficient free space for RealMonitor to hold the contents of the buffer. The application can either discard the packet, or attempt to send it again.

`RM_NotSupported`

Data logging is not supported in this build of RealMonitor.

`RM_Disabled`

Data logging is disabled.

Interrupt handlers must always discard packets that cannot be sent. In all other cases, the caller can either discard its packet, or attempt to send it again.

Implementation

If there is space in the RealMonitor data logging FIFO buffer, the data is copied from the buffer of the application into this FIFO buffer. The packet data is then sent by RealMonitor under interrupt.

Because the data is copied by RealMonitor, the caller does not have to keep the buffer contents unchanged.

6.9.2 rm_EmptyFifo()

This function sends the next word of a data logging packet over the communications link.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416).

Syntax

```
uint32 rm_EmptyFifo(void)
```


Implementation

A word of data is read from the data logging FIFO buffer at index `IMP_GlobalState->fifo.remove`, and `IMP_GlobalState->fifo.remove` is updated to contain the index of the subsequent word.

6.9.3 rm_NextMessage()

This function is called by `rm_TX()` to start transmitting a new data logging packet (see *rm_TX()* on page 659). This function reads the next word of data (a packet header) from the data logging FIFO buffer, and stores the payload length field into the `IMP_GlobalState->current` variable.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416).

Syntax

```
uint32 rm_NextMessage(void)
```

Implementation

The transmission of data logging packets is handled by the function `rm_EmptyFifo()` (see *rm_EmptyFifo()* on page 656), so `IMP_GlobalState->txproc` is set to contain the address of that function. Finally, this function returns by calling `rm_EmptyFifo()` to cause the packet header to be transmitted.

6.10 Communication functions

This section describes the functions that manage the communications channel. The main communication functions are:

- *rm_RX()*
- *rm_TX()* on page 659
- *rm_Poll()* on page 661
- *RM_PollDCC()* on page 661
- *rm_TransmitData()* on page 662
- *rm_FillTransmitBuffer()* on page 663
- *rm_ReceiveData()* on page 663.

The following auxiliary communications functions are defined and used in `rm_control.c`:

- *rm_SetPending()* on page 664
- *rm_SetRX()* on page 664
- *rm_RXDone()* on page 665
- *rm_RXDoneSetPending()* on page 665
- *rm_SetTX()* on page 665
- *rm_TXDone()* on page 666
- *rm_SendPendingData()* on page 666
- *rm_SetPendingSendPending()* on page 667
- *rm_RXDoneSetPendingSendPending()* on page 667
- *rm_TXPend()* on page 667
- *rm_PendTX()* on page 668
- *rm_EnableRXTX()* on page 668
- *rm_EnableTX()* on page 668
- *rm_ResetComms()* on page 669.

6.10.1 rm_RX()

This function is called when a packet header is received. It examines the packet header and decides the action that is required to process the rest of the packet.

Syntax

```
void rm_RX(uint32 header)
```

where:

header A word received from the DCC, containing a packet header.

Implementation

If the channel number is not equal to IMP_Chan_RM, the packet cannot be handled by RealMonitor, and must be skipped. To initiate the skipping of the packet, the function rm_SkipPacketPayload() is called. If you want to implement your own channels, you must modify this code to recognize and handle your own packets.

Your packet must be on the RealMonitor channel, and you must also check whether the opcode field corresponds to a supported opcode. The rm_DispatchTable array contains a pointer for each supported opcode (see *rm_DispatchTable* on page 68). If the value of the opcode field is out of bounds with respect to the dimensions of the array, or the contents of the indexed array element is NULL, there is no handler function installed for this channel.

If a handler for the channel is found, the following occurs:

1. The variable IMP_GlobalState->operation_aborted is set to zero to indicate that no Data Abort has occurred for this packet.
2. An Ok packet header is stored into IMP_GlobalState->pending_data.
3. The handler is called, passing the packet header as the only argument.

If there is no handler for the opcode, the following occurs:

1. The error block is set to contain the RM_Error_UnsupportedOpcode error.
2. IMP_GlobalState->pending_data is set to contain an Error packet, and the function rm_SetPendingSendPending() is called to cause that packet to be sent to the host.
3. Control is passed to the rm_SkipPacketPayload() function, which causes the rest of the packet to be discarded.

6.10.2 rm_TX()

This function is called when the last word of a packet has just been sent over the DCC, and the transmit buffer has now become empty.

Syntax

```
uint32 rm_TX(void)
```

Return value

Returns a word of data to be written to the DCC.

Implementation

This function chooses the packet to be sent next. It does this by setting `IMP_GlobalState->txproc` to point to an appropriate channel handler. The header of the chosen packet is then returned to the caller (see *IMP_GlobalState* on page 69).

The next packet to be sent can be generated from a variety of different sources. The sources are examined in the following order, until a packet is found:

1. **Asynchronous stop packet**

   If the value of the variable `IMP_GlobalState->stop_code` does not equal zero, the value represents an asynchronous stop packet header.

   Asynchronous packets are generated when RealMonitor stops, for example due to a breakpoint being hit. These packets never have a payload.

   The variable `IMP_GlobalState->stop_code` is set to zero, and the previous value of the variable is returned to the caller.

   Note

   The code for handling asynchronous packets is compiled into RMTarget only when the build option `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416)

2. **RM packet**

   If the value of the variable `IMP_GlobalState->txpending` ≠ `NULL`, there is a packet on the RealMonitor channel that must be sent. These RealMonitor packets are generated in response to RealMonitor packets received from the host.

   The variable `IMP_GlobalState->txproc` is set to `IMP_GlobalState->txpending`, and `IMP_GlobalState->txpending` is set to `NULL`. Finally, the function pointed to by `IMP_GlobalState->txproc` is called, and its return value (which represents a packet header) is returned to the caller of this function.

3. **Data logging packet**

   If the flag `IMP_GlobalState->fifo.hasdata` does not equal zero, there are one or more packets waiting to be sent in the FIFO buffer. Packets in the FIFO buffer are generated by your application or RTOS.

   To send a data logging packet, the following steps are performed:

   1. The variable `IMP_GlobalState->txproc` is set to the address of the function `rm_NextMessage()` (see *rm_NextMessage()* on page 657).
   2. The function `rm_NextMessage()` is called to obtain the packet header of the first data logging packet to be sent.

3. The packet header is returned to the caller of this function.

Note

The code for handling data logging packets is compiled into RMTarget only when the build option `RM_OPT_DATALOGGING` is set to `TRUE` (see *RM_OPT_DATALOGGING* on page 416).

If none of these sources have any packets to send, the variable `IMP_GlobalState->tx_flag` is set to zero to indicate that no data must be sent over the communications link. This function must return a value, so it returns a NOP packet because this has no adverse effect if it gets sent to the host (see *NOP* on page 713).

6.10.3 rm_Poll()

This function queries the status of the DCC.

Syntax

```
static void rm_Poll(void)
```

Implementation

If the DCC read buffer is full, control is passed to `rm_ReceiveData()` (see *rm_ReceiveData()* on page 663). If the DCC write buffer is free, control is passed to `rm_TransmitData()` (see *rm_TransmitData()* on page 662). If there is nothing else to do, the function returns to the caller.

The ordering of the above comparisons gives reads from the DCC a higher priority than writes to the communications link.

6.10.4 RM_PollDCC()

This function must be called from your own application when you want to use RealMonitor in a polled mode, rather than in an interrupt-driven mode. It returns after it has sent or received some data over the communications link, or after it has determined that there is nothing to do. Therefore, to allow RealMonitor to remain in contact with the host, you must call this function on a regular basis.

This function must be called from a privileged processor mode.

Note

The frequency with which this function is called is directly related to the maximum data transfer rate over the communications link, and therefore to the responsiveness of the debugger. To avoid receiving data link timeout errors from the host, it is necessary to call this function a minimum of once per second. It is recommended that this function is called at least 100 times per second to obtain a responsive debug session.

Syntax

```
void RM_PollDCC(void)
```

6.10.5 rm_TransmitData()

This function is responsible for transmitting data over the DCC, and for ensuring that any escape words in the normal data stream are quoted in accordance with the escape sequence protocol (see *Escape sequence handling* on page 74).

Syntax

```
static void rm_TransmitData(void)
```

Implementation

The next word of data to be transmitted is always found in `IMP_GlobalState->tx_buffer`. This word is sent over the DCC at the beginning of the function, to minimize the amount of time that the DCC is idle (see *IMP_GlobalState* on page 69).

The beginning of an escape sequence is identified when the value of `IMP_GlobalState->tx_buffer` equals `IMP_ESCAPE`, and the remainder of the escape sequence (the escape value) is found in `IMP_GlobalState->tx_escape_data`. The escape value is copied into `IMP_GlobalState->tx_buffer` so that it is automatically sent the next time this function is called. This function then returns.

If the original value in `IMP_GlobalState->tx_buffer` was not an escape sequence, it is necessary to obtain a new value to store in the buffer. The new value is obtained by calling `IMP_GlobalState->txproc`.

If the new value stored in `IMP_GlobalState->tx_buffer` equals `IMP_ESCAPE`, it must convert this word into an `ESCAPE-QUOTE` escape sequence. This conversion is performed by storing `IMP_ESCAPE_QUOTE` into `IMP_GlobalState->tx_escape_data`.

6.10.6 rm_FillTransmitBuffer()

This function is called by `rm_TransmitData()` to obtain the next word of data to be sent over the DCC.

Syntax

```
static void rm_FillTransmitBuffer(void)
```

Implementation

The variable `IMP_GlobalState->tx_flag` is set to 1. Then, the function pointed to by `IMP_GlobalState->txproc` is called. The return value from the function pointed to by `IMP_GlobalState->txproc` represents the next word of data to be sent, and is stored in `IMP_GlobalState->tx_buffer` (see *IMP_GlobalState* on page 69).

If the function pointed to by `txproc` does not have any data to send, that function must set `IMP_GlobalState->tx_flag` to zero explicitly. This notifies RealMonitor that the return value (to be stored into `IMP_GlobalState->tx_buffer`) is undefined, and must be ignored.

If the value stored in `IMP_GlobalState->tx_buffer` equals `IMP_ESCAPE`, it must send this word as an escape sequence. The conversion into an escape sequence is performed by storing `IMP_ESCAPE_QUOTE` into `IMP_GlobalState->tx_escape_data`.

6.10.7 rm_ReceiveData()

This function reads data from the DCC, and routes it to the correct destination. Also, the function must detect escape sequences in the data stream, and handle them according to the escape sequence protocol (see *Escape sequence handling* on page 74).

Syntax

```
static void rm_ReceiveData(void)
```

Implementation

The first task this function performs is to read a single word of data from the DCC. This word is compared against `IMP_ESCAPE` and, if equal, this marks the beginning of an escape sequence. When the beginning of an escape sequence is detected, then `IMP_GlobalState->rx_escape_flag` is set to a nonzero value, and the function returns (see *IMP_GlobalState* on page 69).

If the word received was not the beginning of an escape sequence, then IMP_GlobalState->rx_escape_flag must be checked. If the flag value was zero, the word received is part of the normal data stream, and a jump is performed to the function pointed to by IMP_GlobalState->rxproc, passing the word as the single parameter.

Otherwise, if the word received was not the start of an escape sequence, and the IMP_GlobalState->rx_escape_flag value was nonzero, this marks the end of an escape sequence. In this case, the action taken depends on the value of the word received:

IMP_ESCAPE_QUOTE

Jump to IMP_GlobalState->rxproc, passing IMP_ESCAPE as the single parameter.

IMP_ESCAPE_RESET

Cause an ESCAPE-GO escape sequence to be sent to the host by calling the function rm_InitCommsState(), with IMP_ESCAPE_GO as the single parameter. Then, set IMP_GlobalState->rxproc to contain the address of the function rm_RX(), so that the next word received is considered the start of a new packet.

IMP_ESCAPE_GO

Set imp_GlobalState->rxproc to contain the address of the function rm_RX(), so that the next word received is considered the start of a new packet.

6.10.8 rm_SetPending()

This function sets IMP_GlobalState->txpending to *pending*.

Syntax

```
static void rm_SetPending(TxHandler *pending)
```

where:

pending The address of any TX handler function.

6.10.9 rm_SetRX()

This function sets IMP_GlobalState->rxproc to *rx*.

Syntax

```
static void rm_SetRX(RxHandler *rx)
```


where:

rx The address of any RX handler function.

6.10.10 rm_RXDone()

This function calls `rm_SetRX()`, passing the address of `rm_RX()` as the single parameter (see *rm_SetRX()* on page 664).

Syntax

```
static void rm_RXDone(void)
```

6.10.11 rm_RXDoneSetPending()

This function sets `IMP_GlobalState->rxproc` to the address of `rm_RX()`, and then calls `rm_SetPending()` with *pending* as the single parameter.

Syntax

```
static void rm_RXDoneSetPending(TxHandler *pending)
```

where:

pending The address of any TX handler function.

6.10.12 rm_SetTX()

This function sets `IMP_GlobalState->txproc` to *tx*, and returns *data* to the caller. Although this function does not use *data*, it is passed through so that the caller can avoid having to both save *data* onto the stack before the function call, and then restore it.

Syntax

```
static uint32 rm_SetTX(uint32 data, TxHandler *tx)
```

where:

data A word of data that is unused by this function.

tx The address of any TX handler function.

Return value

Returns *data* unmodified.

6.10.13 rm_TXDone()

This function sets `IMP_GlobalState->txproc` to the address of `rm_TX()`, and returns *data* to the caller (see *rm_TX()* on page 659). Although this function does not use *data*, it is passed through so that the caller can avoid having to both save *data* onto the stack before the function call, and then restore it.

Syntax

```
uint32 rm_TXDone(uint32 data)
```

where:

data A word of data that is unused by this function.

Return value

Returns *data* unmodified.

6.10.14 rm_SendPendingData()

This function sets `IMP_GlobalState->txproc` to the address of `rm_TX()`, and returns *data* to the caller (see *rm_TX()* on page 659).

Syntax

```
static uint32 rm_SendPendingData(void)
```

Return value

Returns the word of data that is waiting to be transmitted over the DCC.

Implementation

This function sets `IMP_GlobalState->txproc` to the address of `rm_TX()`, and returns the value stored in `IMP_GlobalState->pending_data`. Although this function does not use *data*, it is passed through so that the caller can avoid having to both save *data* onto the stack before the function call, and then restore it (see *IMP_GlobalState* on page 69).

6.10.15 rm_SetPendingSendPending()

This function sets `IMP_GlobalState->txpending` to the address of `rm_SendPendingData()` (see *rm_SendPendingData()* on page 666.

Syntax

```
void rm_SetPendingSendPending(void)
```

6.10.16 rm_RXDoneSetPendingSendPending()

This function performs the following actions:

1. Sets `IMP_GlobalState->rxproc` to the address of `rm_RX()` (see *rm_RX()* on page 658).
2. Sets `IMP_GlobalState->txpending` to the address of `rm_SendPendingData()` (see *rm_SendPendingData()* on page 666).

Syntax

```
void rm_RXDoneSetPendingSendPending(void)
```

6.10.17 rm_TXPend()

This function sets `IMP_GlobalState->txproc` to the address of `rm_SendPendingData()`, and returns *data* to the caller.

Syntax

```
static uint32 rm_TXPend(uint32 data)
```

where:

data A word of data that is unused by this function.

Return value

Returns *data* unmodified.

Implementation

Although this function does not use *data*, it is passed through so that the caller can avoid having to both save *data* onto the stack before the function call, and then restore it.

6.10.18 rm_PendTX()

This function sets `IMP_GlobalState->pending_data` to *data2*, sets `IMP_GlobalState->txproc` to the address of `rm_SendPendingData()`, and returns *data1* to the caller (see *rm_SendPendingData()* on page 666).

Syntax

```
static uint32 rm_PendTX(uint32 data1, uint32 data2)
```

where:

data1 A word of data that is unused by this function.

data2 A word of data that is to be sent over the DCC after *data1* has been sent.

Return value

Returns *data1* unmodified.

Implementation

Although this function does not use *data1*, it is passed through so that the caller can avoid having to both save *data1* onto the stack before the function call, and then restore it.

6.10.19 rm_EnableRXTX()

This function enables the COMMRX and COMMTX interrupts in the interrupt controller.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_USE_INTERRUPTS` is set to `TRUE` (see *RM_OPT_USE_INTERRUPTS* on page 427).

Syntax

```
void rm_EnableRXTX(void)
```

6.10.20 rm_EnableTX()

This function enables the COMMTX interrupt in the interrupt controller.

Note

This function is compiled into RMTarget only when the build option RM_OPT_DATALOGGING is set to TRUE (see *RM_OPT_DATALOGGING* on page 416).

Syntax

```
void rm_EnableTX(void)
```

6.10.21 rm_ResetComms()

This function is used to reset the communications link.

Syntax

```
void rm_ResetComms(uint32 escape_value)
```

where:

escape_value

The escape value to be sent over the communications link. It can be either IMP_ESCAPE_RESET or IMP_ESCAPE_GO.

Implementation

The variable IMP_GlobalState->tx_buffer is set to IMP_ESCAPE, and the parameter *escape_value* is stored into IMP_GlobalState->tx_escape_data.

Finally, a nonzero value is stored into IMP_GlobalState->tx_flag to indicate that there is now some data to send.

6.11 Starting and stopping the foreground application

This section describes the functions that control stopping and starting the foreground application:

- *rm_StoppedLoop()*
- *rm_StoppedToRunning()*.

6.11.1 rm_StoppedLoop()

This function is called after RealMonitor has stopped. It polls the DCC continually to send and receive data. The loop terminates when `IMP_GlobalState->state` equals `RM_State_Running`.

Syntax

```
void rm_StoppedLoop(void)
```

Implementation

A polled mechanism must be used because interrupts might be disabled. Interrupts are in the state in which the application left them prior to being stopped. In general, it is not safe for RealMonitor to re-enable interrupts without the permission of the application.

As soon as the loop has terminated, control passes to `rm_StoppedToRunning()` to enable the foreground application to be restarted (see *rm_StoppedToRunning()*).

6.11.2 rm_StoppedToRunning()

This function is called to change from the state `RM_State_Stopped` to `RM_State_Running`, and to restart the foreground application. This occurs when RealMonitor receives a Go packet.

Syntax

```
void rm_StoppedToRunning(void)
```

Implementation

The function expects to be called with the processor in Undef mode, that is, the mode in which `rm_RunningToStopped()` leaves the processor (see *rm_RunningToStopped()* on page 646).

To restore the foreground application, the following actions are performed:

1. Restore R5_USER to R14_USER.
2. Restore R13_SVC, R14_SVC, and SPSR_SVC.
3. Restore R0_USER to R4_USER.
4. Restore pc and cpsr.

Because this function requires a number of User mode registers for its own purposes, the restoration of the User mode registers must be performed in two parts. The final User mode registers can be restored only at the end of the function.

Step 4 causes the foreground application to start running again.

6.12 Opcode handlers for the RealMonitor channel

This section describes the functions that process RealMonitor packets sent by the host on the RealMonitor channel:

- *rm_SendPendingData()* on page 674.

The following function is used to process NOP packet requests:

- *rm_NOP()* on page 674.

The following functions are used to process GetCapabilities requests:

- *rm_GetCapabilities()* on page 675
- *rm_ReturnGetCapabilitiesPayload()* on page 676.

The following function is used to handle Stop requests:

- *rm_Stop()* on page 677.

The following function is used to handle Go requests:

- *rm_Go()* on page 677.

The following function is used with all read payload functions:

- *rm_ReadData()* on page 679.

The following functions are used to handle ReadBytes requests:

- *rm_ReadBytes()* on page 679
- *rm_ReadBytesPayload()* on page 681.

The following functions are used to handle ReadHalfWords requests:

- *rm_ReadHalfWords()* on page 683
- *rm_ReadHalfWordsPayload()* on page 684.

The following functions are used to handle ReadWords requests:

- *rm_ReadWords()* on page 686
- *rm_ReadWordsPayload()* on page 687.

The following auxiliary memory read functions are provided:

- *rm_SetupRead()* on page 688
- *rm_GetReadAddress()* on page 689
- *rm_GetReadLength()* on page 690
- *rm_ReadHeader()* on page 690
- *rm_ContinueRead()* on page 691.

The following functions are used to handle WriteBytes requests:

- *rm_WriteBytes()* on page 693

- *rm_WriteBytesPayload()* on page 694.

The following functions are used to handle WriteHalfWords requests:

- *rm_WriteHalfWords()* on page 695
- *rm_WriteHalfWordsPayload()* on page 696.

The following functions are used to handle WriteWords requests:

- *rm_WriteWords()* on page 697
- *rm_WriteWordsPayload()* on page 698.

The following auxiliary memory write functions are provided:

- *rm_SetupWrite()* on page 699
- *rm_SetWriteAddress()* on page 6100
- *rm_WriteData()* on page 6100
- *rm_ContinueWrite()* on page 6101.

The following functions are used to handle ExecuteCode requests:

- *rm_ExecuteCode()* on page 6102
- *rm_CallExecuteCodeBuffer()* on page 6103.

The following functions are used to handle GetPC requests:

- *rm_GetPC()* on page 6103
- *rm_GetPCCounts()* on page 6105
- *rm_GetPCHeader()* on page 6106
- *rm_DoGetPC()* on page 6106.

The following functions are used to handle SyncCaches requests:

- *rm_DoGetPC()* on page 6106
- *rm_SyncCaches()* on page 6107
- *rm_SyncCaches_Address()* on page 6108
- *rm_SyncCaches_Length()* on page 6109.

When a packet header is received that is unsupported in the current build, RealMonitor must attempt to read and discard the remainder of the packet. This ensures that synchronization is not lost on the DCC. The packet skipping functions are:

- *rm_SkipPacketPayload()* on page 6110
- *rm_DoSkipPacketPayload()* on page 6110.

6.12.1 rm_SendPendingData()

This function is called to send the last word of a packet over the DCC.

Syntax

```
static uint32 rm_SendPendingData(void)
```

Return value

Returns a word of data to be written to the DCC.

Implementation

The word of data has been stored previously by another channel handler into `IMP_GlobalState->pending_data`. After setting `IMP_GlobalState->txproc` to `rm_TX()`, the word of data is returned (see *IMP_GlobalState* on page 69).

6.12.2 rm_NOP()

This function is called by `rm_RX()` when a NOP packet header is received on the RealMonitor channel (see *rm_RX()* on page 658), as shown in Figure 6-5.

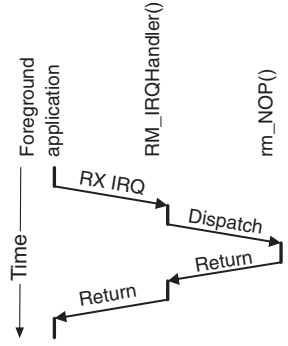

Figure 6-5 Processing a NOP packet

Syntax

```
static void rm_NOP(uint32 length)
```


where:

length A word received from the DCC, containing the length field from a NOP packet header. As defined in *NOP* on page 713, *length* must always be zero.

Implementation

This function only discards a NOP packet header, because a NOP packet must not have a payload, and no response packet must be sent.

6.12.3 rm_GetCapabilities()

This function is called by `rm_RX()` when a GetCapabilities packet header is received on the RealMonitor channel, as shown in Figure 6-6 (see *rm_RX()* on page 658).

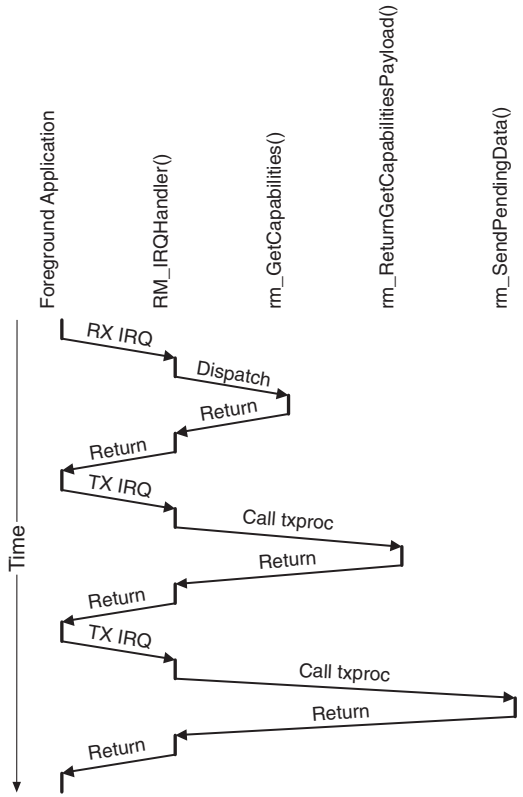

Figure 6-6 Processing a GetCapabilities packet

Syntax

```
static void rm_GetCapabilities(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a GetCapabilities packet header. As defined in *GetCapabilities* on page 713, *length* must always be zero.

Implementation

The function sets `IMP_GlobalState->txpending` to point to the `rm_ReturnGetCapabilitiesPayload()` and returns (see *IMP_GlobalState* on page 69 and *rm_ReturnGetCapabilitiesPayload()*).

The *length* argument is ignored because a GetCapabilities packet must not have a payload.

6.12.4 rm_ReturnGetCapabilitiesPayload()

This function generates the response packet to a `GetCapabilities` request.

Syntax

```
static uint32 rm_ReturnGetCapabilitiesPayload(void)
```

Return value

Returns an Ok packet header to be written to the DCC.

Implementation

The `rm_GetCapabilities()` function arranges for this function to be called when a TX interrupt occurs (see *rm_GetCapabilities()* on page 675).

The base address of the `rm_CapabilitiesTable` is written into `IMP_GlobalState->pending_data`, and `IMP_GlobalState->txproc` is set to `rm_SendPendingData()` (see *IMP_GlobalState* on page 69). Finally, `rm_ReturnGetCapabilitiesPayload()` returns a packet header for the RealMonitor channel, with the opcode `RM_Msg_Okay`, and the length 4.

6.12.5 rm_Stop()

This function is called by `rm_RX()` when a Stop packet is received from the host, as shown in Figure 6-7 (see *rm_RX()* on page 658).

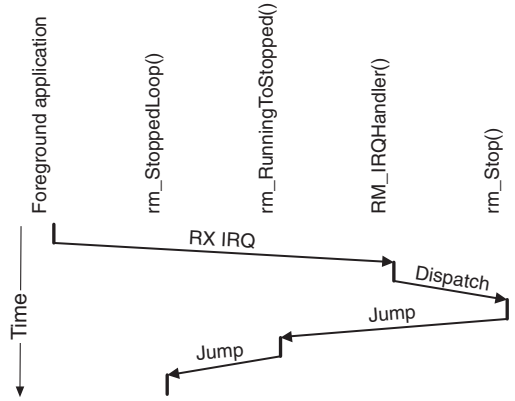

Figure 6-7 Processing a Stop packet

Syntax

```
void rm_Stop(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a Stop packet header. As defined in *Stop* on page 714, *length* must always be zero.

Implementation

This function sets `IMP_GlobalState->state` to `RM_State_Stopped` (see *IMP_GlobalState* on page 69), and then jumps to `rm_RunningToStopped(RM_Msg_Stopped)`.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_STOPSTART` is set to `TRUE` (see *RM_OPT_STOPSTART* on page 416).

6.12.6 rm_Go()

This function restarts the foreground application after a Go packet is received, as shown in Figure 6-8 on page 678.

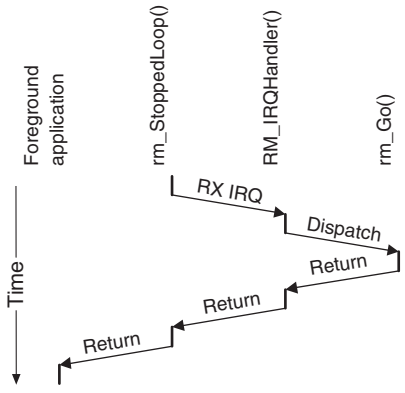

Figure 6-8 Processing a Go packet

Syntax

```
void rm_Go(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a Go packet header. As defined in *Go* on page 714, *length* must always be zero.

Implementation

When a Go packet is received on the RealMonitor channel, the foreground application must be restarted.

First, RealMonitor sets the state variable in the global state data structure to `RM_State_Running` (see *IMP_GlobalState* on page 69).

RealMonitor then switches into the processor mode in which the application was originally running, and restores all the visible registers in that mode from the registers block. However, where the foreground application was running in User mode, the registers are restored by switching into System mode.

The final registers to be transferred are the pc and the *program status register* (psr). This causes the foreground application to begin executing again.

It is not possible to restore all of the registers in the registers block because this corrupts the registers being used by interrupt-driven tasks executing in the background.

Note

This function is compiled into RMTarget only when the build option RM_OPT_STOPSTART is set to TRUE (see *RM_OPT_STOPSTART* on page 416).

6.12.7 rm_ReadData()

This function reads the rwaddress, rwlength, and rwproc fields from the IMP_GlobalState data structure. It is used only to assist the read payload functions.

Syntax

```
static uint32 rm_ReadData(void)
```

Return value

Returns the value returned by IMP_GlobalSate->rwproc, that is, a word to be written to the DCC.

Implementation

This function calls the function pointer contained in the variable IMP_GlobalState->rwproc, passing the values IMP_GlobalState->rwaddress as arguments.

Note

This function is optimized in the case where only one of the three read build options (RM_OPT_READBYTES, RM_READ_HALFWORDS, and RM_OPT_READWORDS) is set to TRUE (see *RMTarget build options and macros* on page 415).

For example, if RM_OPT_READBYTES is TRUE, but RM_READ_HALFWORDS and RM_OPT_READWORDS are both FALSE, the function rm_ReadBytesPayload() is inlined within rm_ReadData() because this is the only value the variable IMP_GlobalState->rwproc can ever take. This both reduces the size of the RMTarget library, and improves its performance.

6.12.8 rm_ReadBytes()

This function is called by rm_RX() when a ReadBytes packet header is received on the RealMonitor channel (see *rm_RX()* on page 658). The typical execution graph is shown in Figure 6-9 on page 680.

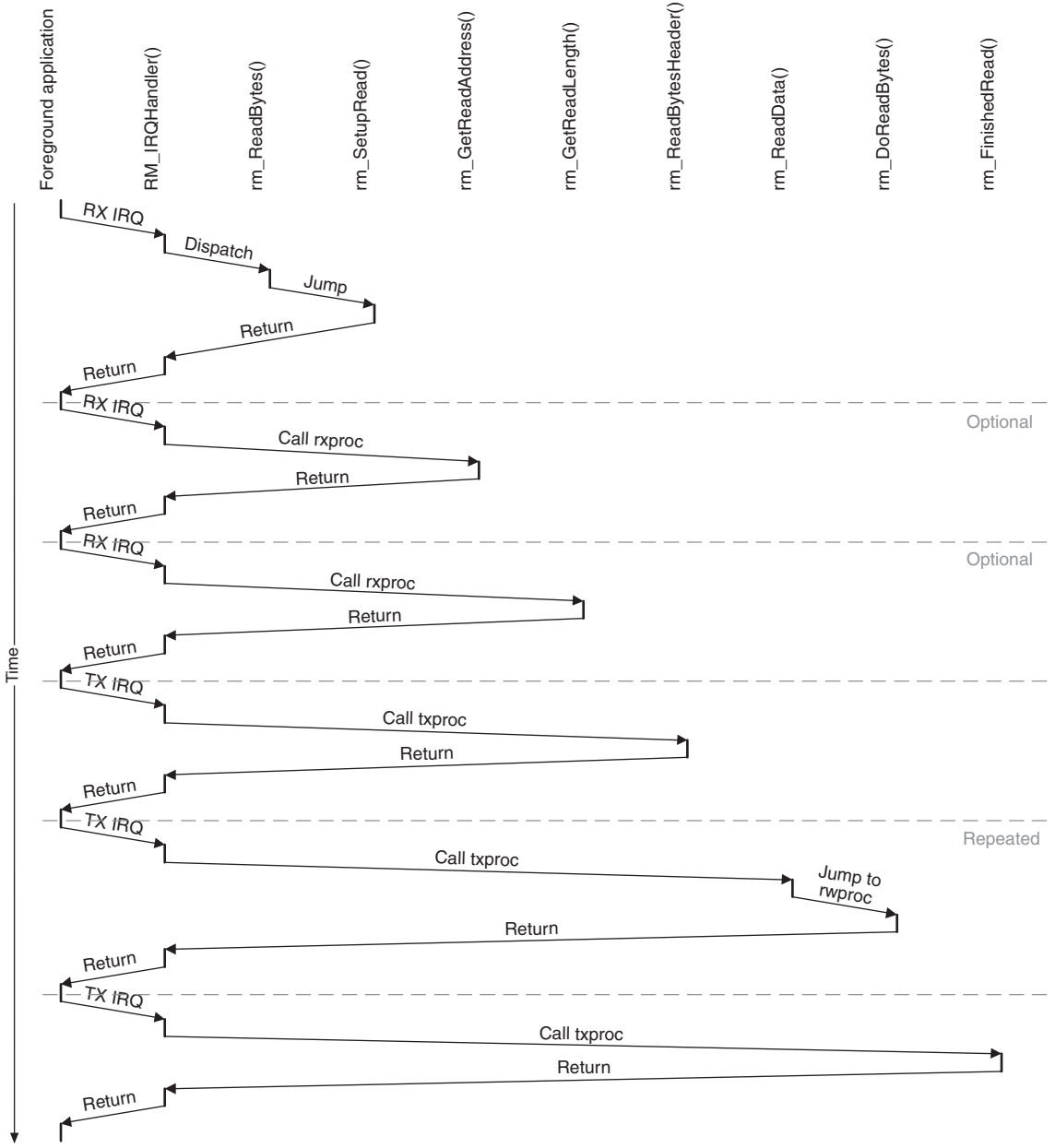

Figure 6-9 Processing a ReadBytes packet

Syntax

```
static void rm_ReadBytes(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a ReadBytes packet header. As defined in *ReadBytes* on page 718, *length* can take the values 0, 4, or 8.

Implementation

The variable IMP_GlobalState->rwlength is set to 1. This is used as the default length if the optional length field in the packet payload has not been found. The variable IMP_GlobalState->rwproc is set to rm_ReadBytesPayload() (see *IMP_GlobalState* on page 69).

If *length* is zero, there are no optional address and size words in the packet payload. In this case, IMP_GlobalState->txproc is set to rm_ReadHeader().

If the length argument is not zero, there is an address word (and possibly a size word) in the packet payload. In this case, IMP_GlobalState->rxproc is set to rm_GetReadAddress().

Note

This function is compiled into RMTarget only when the build option RM_OPT_READBYTES is set to TRUE (see *RM_OPT_READBYTES* on page 418).

6.12.9 rm_ReadBytesPayload()

This function performs the actual reads from memory for a ReadBytes packet.

Syntax

```
static uint32 rm_ReadBytesPayload(uint32 length, uint32 address, uint32 length)
```

where:

dummy Not used. Its purpose is to shift registers used by the other parameters (to improve C compiler-generated code).

address The address to read from.

length Number of bytes remaining to be read.

Return value

Returns a word of data to be written to the DCC.

Implementation

Every time this function is called, it returns a word containing up to four bytes read from the address given by the variable `IMP_GlobalState->rwaddress`. This variable is incremented by the number of bytes read, and the `IMP_GlobalState->rwlength` variable is decremented by the number of bytes read (see *IMP_GlobalState* on page 69).

After reading the data from memory, a call is made to `rm_ContinueRead()`, passing the word of data, the new address, and the new length as the three arguments (see *rm_ContinueRead()* on page 691).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_READBYTES` is set to `TRUE` (see *RM_OPT_READBYTES* on page 418).

6.12.10 rm_ReadHalfWords()

This function is called by `rm_RX()` when a ReadHalfWords packet header is received on the RealMonitor channel (see *rm_RX()* on page 658). The typical execution graph is shown in Figure 6-10.

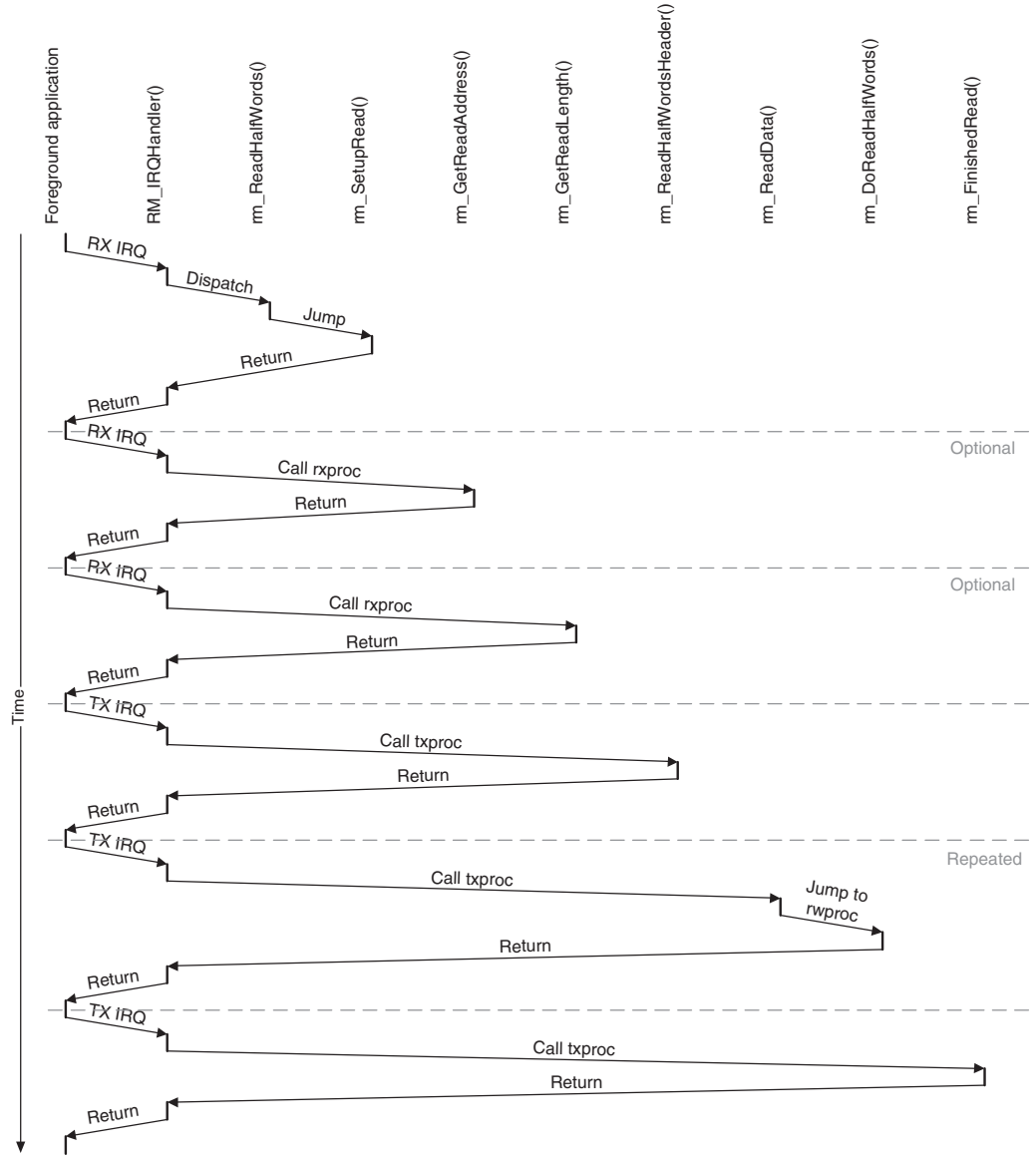

Figure 6-10 Processing a ReadHalfWords packet

Syntax

```
static void rm_ReadHalfWords(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a ReadHalfWords packet header. As defined in *ReadHalfWords* on page 723, *length* can take the values 0, 4, or 8.

Implementation

The variable IMP_GlobalState->rwlength is set to 2. This is used as the default length if the optional length field in the packet payload has not been used by the host. The variable IMP_GlobalState->rwproc is set to rm_ReadHalfWordsPayload() (see *IMP_GlobalState* on page 69).

If the *length* argument is zero, there are no optional address and size words in the packet payload. In this case, IMP_GlobalState->txproc is set to rm_ReadHeader().

If the *length* argument is not zero, there is an address word (and possibly a size word) in the packet payload. In this case IMP_GlobalState->rxproc is set to rm_GetReadAddress().

Note

This function is compiled into RMTarget only when the build option RM_OPT_READHALFWORDS is set to TRUE (see *RM_OPT_READHALFWORDS* on page 419).

6.12.11 rm_ReadHalfWordsPayload()

This function performs the actual reads from memory for a ReadHalfWords packet.

Syntax

```
static uint32 rm_ReadHalfWordsPayload(void)
```

Return value

Returns a word of data to be written to the DCC.

Implementation

Every time this function is called, it returns a word containing up to two halfwords read from the address given by the variable `IMP_GlobalState->rwaddress`. This variable is incremented by the number of bytes read, and the `IMP_GlobalState->rwlength` variable is decremented by the number of bytes read (see *IMP_GlobalState* on page 69).

After reading the data from memory, a call is made to `rm_ContinueRead()`, passing the word of data, the new address, and the new length as the three arguments (see *rm_ContinueRead()* on page 691).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_READHALFWORDS` is set to `TRUE` (see *RM_OPT_READHALFWORDS* on page 419).

6.12.12 rm_ReadWords()

This function is called by `rm_RX()` when a ReadWords packet header is received on the RealMonitor channel (see *rm_RX()* on page 658). The typical execution graph is shown in Figure 6-11.

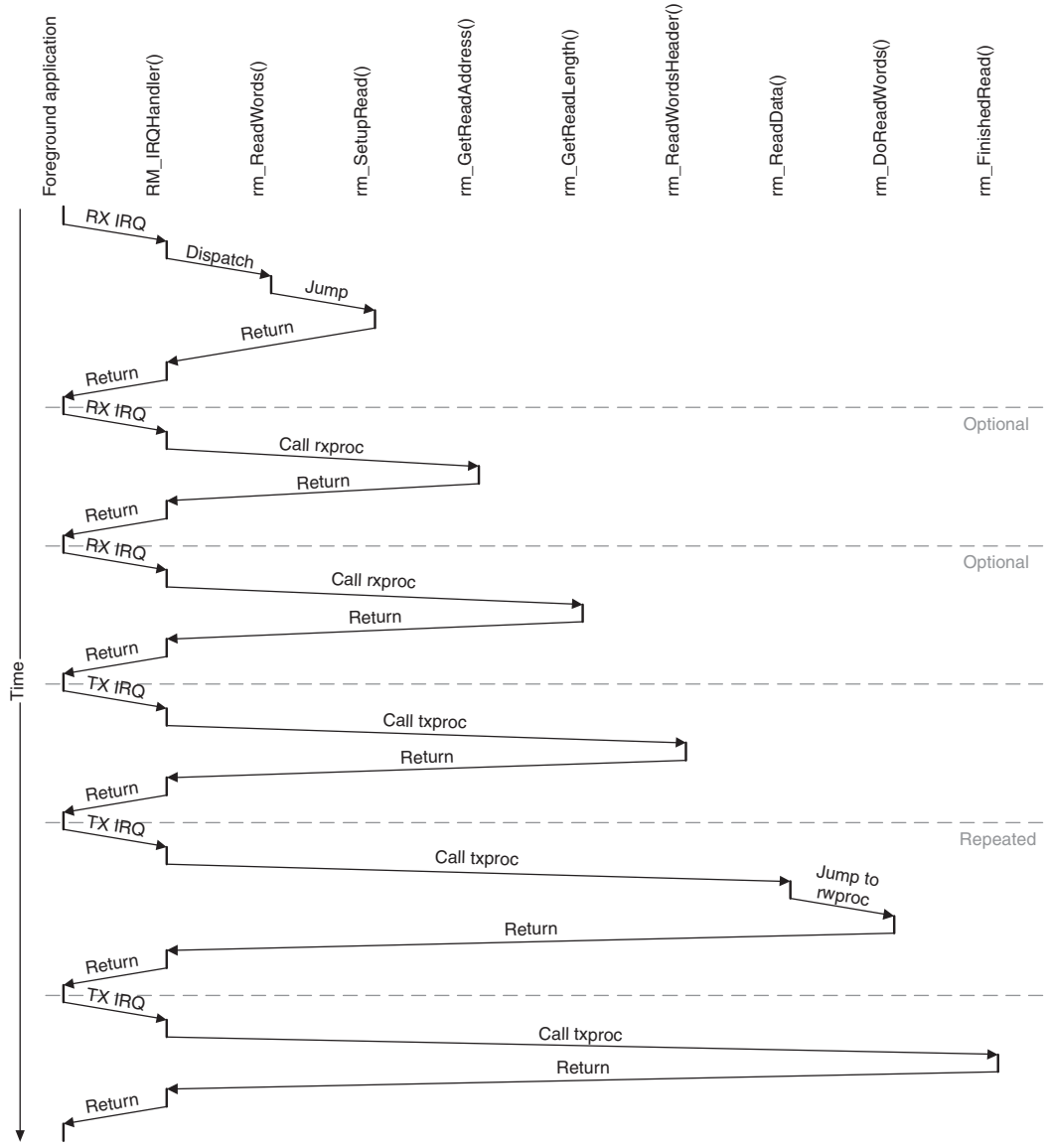

Figure 6-11 Processing a ReadWords packet

Syntax

```
static void rm_ReadWords(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a ReadWords packet header. As defined in *ReadWords* on page 729, *length* can take the values 0, 4, or 8.

Implementation

The variable IMP_GlobalState->rwlength is set to 4. This is used as the default length if the optional length field in the packet payload has not been supplied. The variable IMP_GlobalState->rwproc is set to rm_ReadWordsPayload() (see *IMP_GlobalState* on page 69).

If *length* is zero, there are no optional address and size words in the packet payload. In this case, IMP_GlobalState->txproc is set to the address of the function rm_ReadHeader().

If *length* is not zero, there is an address word (and possibly a size word) in the packet payload. In this case, IMP_GlobalState->rxproc is set to rm_GetReadAddress().

Note

This function is compiled into RMTarget only when the build option RM_OPT_READWORDS is set to TRUE (see *RM_OPT_READWORDS* on page 419).

6.12.13 rm_ReadWordsPayload()

This function performs the actual reads from memory for a ReadWords packet.

Syntax

```
static uint32 rm_ReadWordsPayload(void)
```

Return value

Returns a word of data to be written to the DCC.

Implementation

Every time this function is called, it returns a word read from the address given by the variable IMP_GlobalState->rwaddress. This variable is incremented by four, and the IMP_GlobalState->rwlength variable is decremented by four (see *IMP_GlobalState* on page 69).

After reading the data from memory, a call is made to rm_ContinueRead(), passing the word of data, the new address, and the new length as the three arguments (see *rm_ContinueRead()* on page 691).

Note

This function is compiled into RMTarget only when the build option RM_OPT_READWORDS is set to TRUE (see *RM_OPT_READWORDS* on page 419).

6.12.14 rm_SetupRead()

This function initializes the variables that are used when processing the remainder of a ReadBytes, ReadHalfWords, or ReadWords packet:

- IMP_GlobalState->rwlength is set to *size*
- IMP_GlobalState->rwproc is set to *read_func*
- IMP_GlobalState->rwmask is set to *size*–1.

See *IMP_GlobalState* on page 69 for details on these variables.

Syntax

```
static void rm_SetupRead(uint32 length, TxHandler *read_func, uint32 size)
```

where:

length	Contains the length field from the packet header.
read_func	A pointer to a function that performs the reading of data from memory.
size	Contains the default size of the memory transfer.

Implementation

If *length*>0, the packet payload contains an address field. In this case, a call is made to rm_SetRX(), passing the address of rm_GetReadAddress() as the only argument (see *rm_GetReadAddress()* on page 689).

If *length*=0, the packet payload contains no address or size fields. In this case, a call is made to `rm_ReadHeader()` so that the response packet header can be sent (see *rm_ReadHeader()* on page 690).

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_READBYTES`, `RM_OPT_READHALFWORDS`, or `RM_OPT_READWORDS` is set to `TRUE` (see *RM_OPT_READBYTES* on page 418, *RM_OPT_READHALFWORDS* on page 419, and *RM_OPT_READWORDS* on page 419).

6.12.15 rm_GetReadAddress()

This function reads the address word from the packet payload for a ReadBytes, ReadHalfWords, or ReadWords memory opcode on the RealMonitor channel.

Syntax

```
static void rm_GetReadAddress(uint32 address)
```

where:

address A word received from the DCC, containing the address field from a read memory packet payload.

Implementation

The variable `IMP_GlobalState->rwaddress` is set to *address*. The variable `IMP_GlobalState->header` is then used to work out whether there is another word of data to be read from the packet payload (see *IMP_GlobalState* on page 69).

If there is any additional data in the payload, it is the optional length word. In this case, `IMP_GlobalState->rxproc` is set to `rm_GetReadLength()`.

If there is no additional data in the payload, then `IMP_GlobalState->txproc` is set to `IMP_GlobalState->rwproc`. This allows the actual data transfer to begin.

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_READBYTES`, `RM_OPT_READHALFWORDS`, or `RM_OPT_READWORDS` is set to `TRUE` (see *RM_OPT_READBYTES* on page 418, *RM_OPT_READHALFWORDS* on page 419, and *RM_OPT_READWORDS* on page 419).

6.12.16 rm_GetReadLength()

This function reads the length word from the packet payload for a ReadBytes, ReadHalfWords, or ReadWords memory opcode on the RealMonitor channel.

Syntax

```
static void rm_GetReadLength(uint32 length)
```

where:

length A word received from the DCC. It contains the length field from a read memory packet payload.

Implementation

The variable `IMP_GlobalState->rwlength` is set to the length argument passed to this function (see *IMP_GlobalState* on page 69).

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_READBYTES`, `RM_OPT_READHALFWORDS`, or `RM_OPT_READWORDS` is set to `TRUE` (see *RM_OPT_READBYTES* on page 418, *RM_OPT_READHALFWORDS* on page 419, and *RM_OPT_READWORDS* on page 419).

6.12.17 rm_ReadHeader()

This function returns a word containing the response packet header for a ReadBytes, ReadHalfWords, or ReadWords memory opcode on the RealMonitor channel.

Syntax

```
static uint32 rm_ReadHeader(void)
```

Return value

Returns an Ok packet header to be written to the DCC.

Implementation

The header consists of:

Channel `IMP_Chan_RTM`.

Opcode RM_Msg_Okay.

Length 4 + IMP_GlobalState->rwlength.

The address of the function rm_ReadData() is stored in the variable IMP_GlobalState->txproc . This allows the actual data transfer to begin.

Note

This function is compiled into RMTarget only when at least one of the build options RM_OPT_READBYTES, RM_OPT_READHALFWORDS, or RM_OPT_READWORDS is set to TRUE (see *RM_OPT_READBYTES* on page 418, *RM_OPT_READHALFWORDS* on page 419, and *RM_OPT_READWORDS* on page 419).

6.12.18 rm_ContinueRead()

This function is called by rm_ReadBytesPayload(), rm_ReadHalfWordsPayload(), and rm_ReadWordsPayload() to send up to four bytes of data, that have been read from memory, over the DCC (see *rm_ReadBytesPayload()* on page 681, *rm_ReadHalfWordsPayload()* on page 684, and *rm_ReadWordsPayload()* on page 687).

Syntax

```
static uint32 rm_ContinueRead(uint32 data, uint32 address, uint32 length)
```

where:

data A word of data that has been read from memory, and is to be sent over the DCC.

address The address from where the next memory access starts.

length The number of bytes that remain to be read from memory as part of this read memory packet.

Return value

Returns a word of data to be written to the DCC.

Implementation

The *address* parameter is stored into IMP_GlobalState->rwaddress so that the next memory access can continue from where this memory access left off (see *IMP_GlobalState* on page 69).

The *length* parameter is examined as follows:

length **= 0** Indicates that this is the last read from memory that must be performed for this packet. This function returns by calling `rm_TXPend()`, passing *data* as the single argument (see *rm_TXPend()* on page 667). This causes the value of *data* to be sent over the DCC, and schedules the contents of `IMP_GlobalState->pending_data` to be sent after that.

length $\neq$ **0** Indicates that more memory access must be performed before the packet is complete. This function stores the *length* parameter into `IMP_GlobalState->rwlength`, and then returns the *data* parameter to the caller so that it can be sent over the DCC.

6.12.19 rm_WriteBytes()

This function is called when a WriteBytes packet header is received over the DCC. The typical execution graph is shown in Figure 6-12.

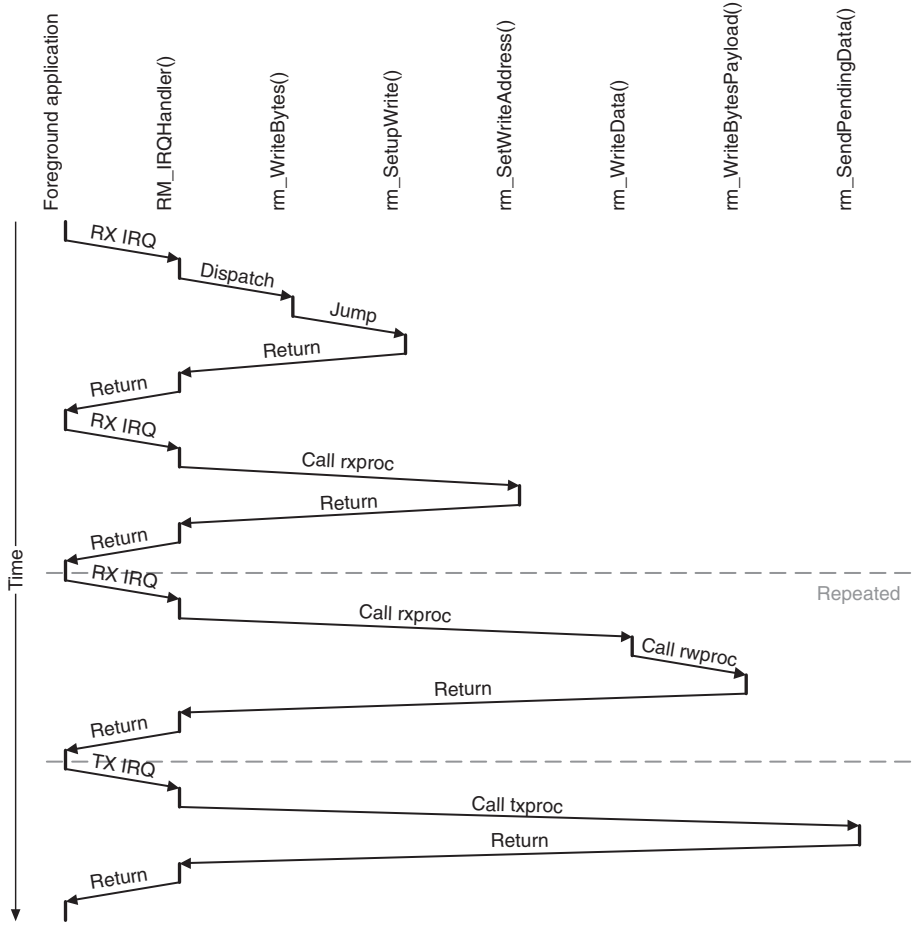

Figure 6-12 Processing a WriteBytes packet

Syntax

```
static void rm_WriteBytes(uint32 length)
```

length A word received from the DCC, containing the length field from a WriteBytes packet header.

Implementation

A call is made to `rm_SetupWrite(length, rm_WriteBytesPayload, 0x0)`. The length of the packet needs to be passed to ensure that the memory write function can detect the end of the packet. The address of `rm_WriteBytesPayload()` must be passed so that this function can be installed as the `txproc` after the address field in the payload has been received.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_WRITEBYTES` is set to `TRUE` (see *RM_OPT_WRITEBYTES* on page 419).

6.12.20 rm_WriteBytesPayload()

This function performs the actual writes to memory.

Syntax

```
static void rm_WriteBytesPayload(uint32 data, uint32 address, uint32 length)
```

where:

data	A word received from the DCC, containing up to four bytes of data to be written to memory.
address	The address to which to write the word data.
length	The number of bytes remaining to be written.

Implementation

Every time this function is called, it writes up to four bytes of data to the address given by the variable `IMP_GlobalState->rwaddress`. This variable is incremented by the number of bytes written, and the `IMP_GlobalState->rwlength` variable is decremented by the number of bytes written (see *IMP_GlobalState* on page 69).

After writing the data to memory, a call is made to the function `rm_ContinueWrite()`, passing the new *address* and new *length* as arguments (see *rm_ContinueWrite()* on page 6101).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_WRITEBYTES` is set to `TRUE` (see *RM_OPT_WRITEBYTES* on page 419).

6.12.21 rm_WriteHalfWords()

This function is called when a WriteHalfWords packet header is received over the DCC. The typical execution graph is shown in Figure 6-13.

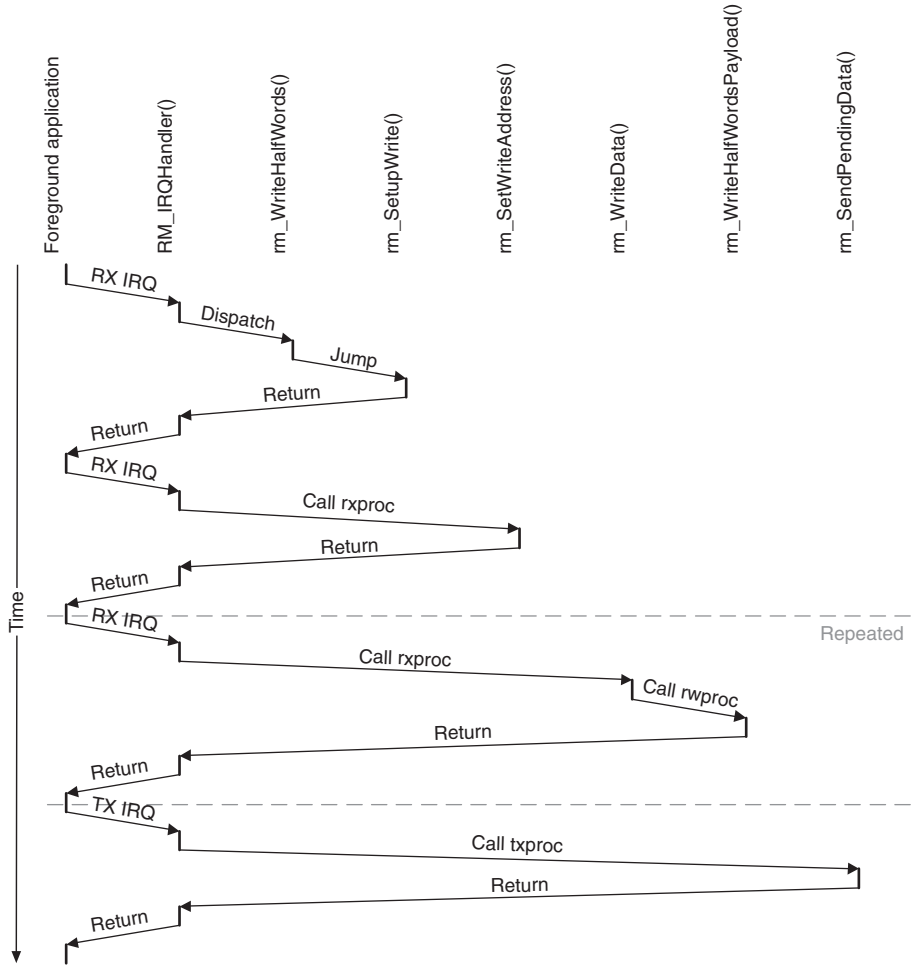

Figure 6-13 Processing of a WriteHalfWords packet

Syntax

```
static void rm_WriteHalfWords(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a WriteHalfWords packet header.

Implementation

A call is made to `rm_SetupWrite(length, rm_WriteHalfWordsPayload, 0x1)`. The length of the packet needs to be passed to ensure that the memory write function can detect the end of the packet. The address of `rm_WriteHalfWordsPayload()` must be passed so that this function can be installed as the `txproc` after the address field in the payload has been received.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_WRITEHALFWORDS` is set to `TRUE` (see *RM_OPT_WRITEHALFWORDS* on page 419).

6.12.22 rm_WriteHalfWordsPayload()

This function performs the actual writes to memory.

Syntax

```
static void rm_WriteHalfWordsPayload(uint32 data, uint32 address, uint32 length)
```

where:

data A word received from the DCC, containing up to two halfwords of data to be written to memory.

address The address to which to write the word data.

length The number of bytes remaining to be written.

Implementation

Every time this function is called, it writes up to two halfwords of data to the address given by the variable `IMP_GlobalState->rwaddress`. This variable is incremented by the number of bytes written, and the `IMP_GlobalState->rwlength` variable is decremented by the number of bytes written (see *IMP_GlobalState* on page 69).

After writing the data to memory, a call is made to the function `rm_ContinueWrite()`, passing the new *`address`* and new *`length`* as arguments (see *rm_ContinueWrite()* on page 6101).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_WRITEHALFWORDS` is set to `TRUE` (see *RM_OPT_WRITEHALFWORDS* on page 419).

6.12.23 rm_WriteWords()

This function is called when a WriteWords packet header is received over the DCC. The typical execution graph is shown in Figure 6-14.

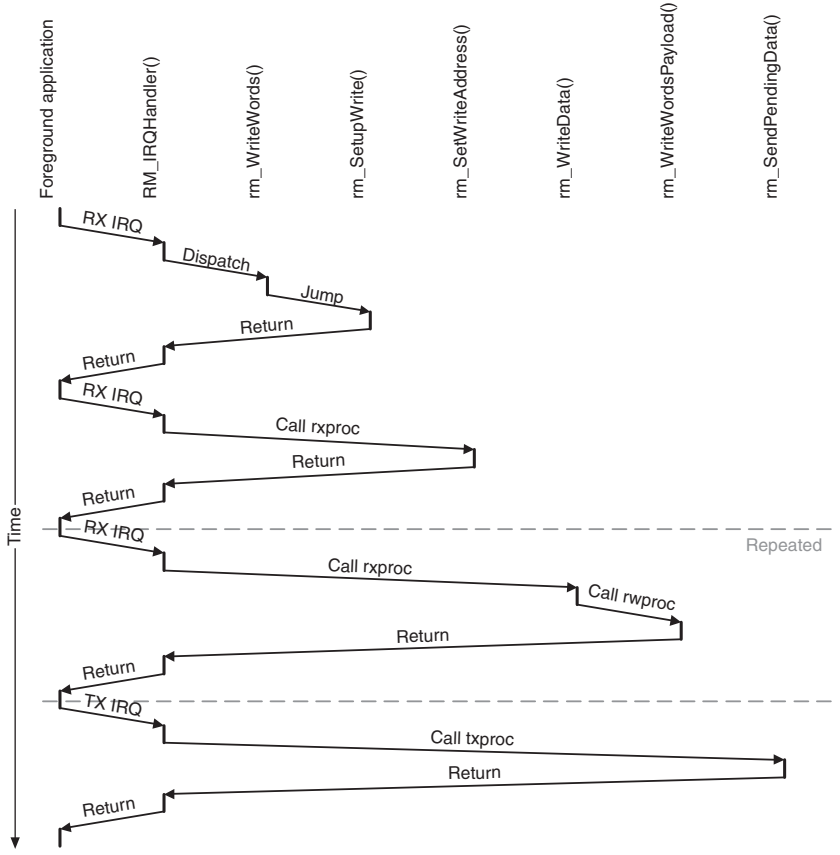

Figure 6-14 Processing of a WriteWords packet

Syntax

```
static void rm_WriteWords(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a WriteWords packet header.

Implementation

A call is made to `rm_SetupWrite(length, rtm_WriteWordsPayload,0x3)`. The length of the packet must be passed to ensure that the memory write function can detect the end of the packet. The address of `rm_WriteWordsPayload()` must be passed so that this function can be installed as the `txproc` after the address field in the payload has been received.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_WRITEWORDS` is set to `TRUE` (see *RM_OPT_WRITEWORDS* on page 420).

6.12.24 rm_WriteWordsPayload()

This function performs the actual writes to memory.

Syntax

```
static void rm_WriteWordsPayload(uint32 data, uint32 address, uint32 length)
```

where:

data A word received from the DCC to be written to memory.

address The address to which to write the word data.

length The number of bytes remaining to be written.

Implementation

Every time this function is called, it writes one word of data to the address given by the variable `IMP_GlobalState->rwaddress`. This variable is incremented by four, and the `IMP_GlobalState->rwlength` variable is decremented by 4 (see *IMP_GlobalState* on page 69).

After writing the data to memory, a call is made to the function rm_ContinueWrite(), passing the new *address* and new *length* as arguments (see *rm_ContinueWrite()* on page 6101).

Note

This function is compiled into RMTarget only when the build option RM_OPT_WRITEWORDS is set to TRUE (see *RM_OPT_WRITEWORDS* on page 420).

6.12.25 rm_SetupWrite()

This function is called by any of rm_WriteBytes(), rm_WriteHalfWords(), or rm_WriteWords() to set up the state of RealMonitor so that a write-to-memory packet can be processed (see *rm_WriteBytes()* on page 693, *rm_WriteHalfWords()* on page 695, and *rm_WriteWords()* on page 697).

Syntax

```
static void rm_SetupWrite(uint32 length, WriteProc *write_proc, uint32 mask)
```

where:

length Contains the length field from the packet header.

write_proc A pointer to a function that performs the writing to memory.

mask Contains the default size of the memory transfer–1.

Implementation

This function performs the following:

- IMP_GlobalState->rwmask is set to *mask*.
- IMP_GlobalState->rwlength is set to *length*–4
- IMP_GlobalState->rwproc is set to *write_proc*.

Finally, a call is made to rm_SetRX(), passing the address of the function rm_SetWriteAddress() as the only argument (see *rm_SetRX()* on page 664 and *rm_SetWriteAddress()* on page 6100).

Note

This function is compiled into RMTarget only when at least one of the build options RM_OPT_WRITEBYTES, RM_OPT_WRITEHALFWORDS, or RM_OPT_WRITEWORDS is set to TRUE (see *RM_OPT_WRITEBYTES* on page 419, *RM_OPT_WRITEHALFWORDS* on page 419, and *RM_OPT_WRITEWORDS* on page 420).

6.12.26 rm_SetWriteAddress()

This function reads the address word from the packet payload for a WriteBytes, WriteHalfWords, or WriteWords memory opcode on the RealMonitor channel.

Syntax

```
static void rm_SetWriteAddress(uint32 address)
```

where:

address A word received from the communications link, containing the address field from a write memory packet payload.

Implementation

The variable `IMP_GlobalState->rwaddress` is set to the address passed as the argument to the function. Then, `IMP_GlobalState->rwproc` is set to the address of the function `rm_WriteData()` to allow the data transfer to begin (see *rm_WriteData()*).

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_WRITEBYTES`, `RM_OPT_WRITEHALFWORDS`, or `RM_OPT_WRITEWORDS` is set to `TRUE` (see *RM_OPT_WRITEBYTES* on page 419, *RM_OPT_WRITEHALFWORDS* on page 419, and *RM_OPT_WRITEWORDS* on page 420).

6.12.27 rm_WriteData()

This function is called when a word of data is received over the DCC during a write-to-memory packet. It passes the word to any of `rm_WriteBytesPayload()`, `rm_WriteHalfWordsPayload()`, or `rm_WriteWordsPayload()`, as appropriate (see *rm_WriteBytesPayload()* on page 694, *rm_WriteHalfWordsPayload()* on page 696, and *rm_WriteWordsPayload()* on page 698).

Note

This function is compiled into RMTarget only when at least one of the build options `RM_OPT_WRITEBYTES`, `RM_OPT_WRITEHALFWORDS` or `RM_OPT_WRITEWORDS` is set to `TRUE` (see *RM_OPT_WRITEBYTES* on page 419, *RM_OPT_WRITEHALFWORDS* on page 419, *RM_OPT_WRITEWORDS* on page 420).

If only one of these options is set to TRUE, the appropriate payload function is compiled inline. This both reduces the size of the RMTarget library, and improves its performance.

Syntax

```
static void rm_WriteData(uint32 word)
```

where:

word The next word of data received over the DCC.

Implementation

To process the *word* of data, this function calls `IMP_GlobalState->rwproc(word, IMP_GlobalState->rwlength, IMP_GlobalState->rwaddress)` (see *IMP_GlobalState* on page 69).

6.12.28 rm_ContinueWrite()

This function is called by `rm_WriteBytesPayload()`, `rm_WriteHalfWordsPayload()`, and `rm_WriteWordsPayload()` after one of these functions has just written some data to memory (see *rm_WriteBytesPayload()* on page 694, *rm_WriteHalfWordsPayload()* on page 696, and *rm_WriteWordsPayload()* on page 698).

Syntax

```
static void rm_ContinueWrite(uint32 dummy, uint32 address, uint32 length)
```

where:

dummy Ignored by this function. It is included so that the caller does not need to rearrange its registers when calling this function.

address The address from where the next memory access starts.

length The number of bytes that remain to be written as part of this write memory packet.

Implementation

The *address* parameter is stored into `IMP_GlobalState->rwaddress`, and *length* is stored into `IMP_GlobalState->rwlength` so that the next memory access can continue from where this memory access finished (see *IMP_GlobalState* on page 69).

Finally, when *length* equals zero, this is the last write to memory that must be performed for this packet. In this case, a call is made to `rm_RXDoneSetPendingSendPending()` (see *rm_RXDoneSetPendingSendPending()* on page 667).

Note

This function is compiled into RMTarget when at least one of the build options RM_OPT_WRITEBYTES, RM_OPT_WRITEHALFWORDS, or RM_OPT_WRITEWORDS is set to TRUE (see *RM_OPT_WRITEBYTES* on page 419, *RM_OPT_WRITEHALFWORDS* on page 419, and *RM_OPT_WRITEWORDS* on page 420).

6.12.29 rm_ExecuteCode()

This function is called when an ExecuteCode packet header is received over the communications link. The typical execution graph is shown in Figure 6-15.

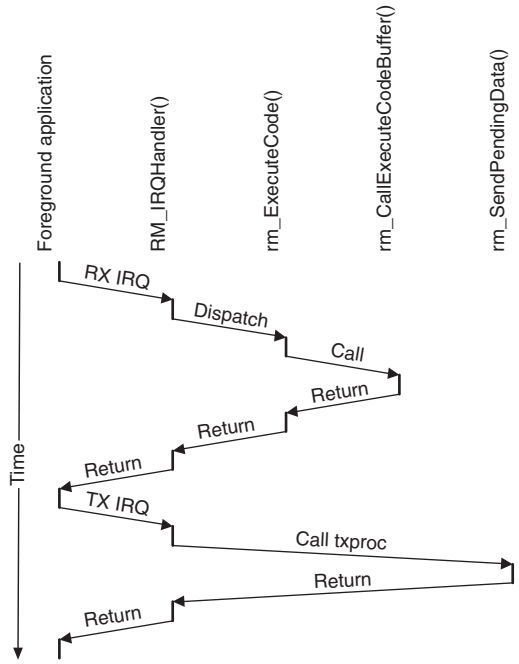

Figure 6-15 Processing an ExecuteCode Packet

Syntax

```
static void rm_ExecuteCode(uint32 length)
```

where:

length A word received from the DCC, containing the length field from an ExecuteCode packet header.

Implementation

If the host adheres to the RealMonitor protocol, *length* must be zero. That is, it has no payload. For efficiency, RMTarget never checks this.

The function `rm_CallExecuteCodeBuffer()` is called to execute the code in the code buffer (see *rm_CallExecuteCodeBuffer()*).

Finally, the function `rm_RXDoneSetPending(rm_SendPendingData)` is called (see *rm_RXDoneSetPending()* on page 665). This schedules the value stored in `IMP_GlobalState->pending_data` to be sent to the host (see *IMP_GlobalState* on page 69).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_EXECUTECODE` is set to `TRUE` (see *RM_OPT_EXECUTECODE* on page 420).

6.12.30 rm_CallExecuteCodeBuffer()

This function switches into Supervisor mode and calls the first instruction in the execute code buffer (see *RM_ExecuteCodeBlock* on page 619). Switching into Supervisor mode is necessary so that aborts and undefined instructions in the execute code buffer can be detected without corrupting any important registers.

After the code in the execute code buffer has completed, this function switches back into the original processor mode.

Note

The function is compiled into RMTarget only when the build option `RM_OPT_EXECUTECODE` is set to `TRUE` (see *RM_OPT_EXECUTECODE* on page 420).

Syntax

```
void rm_CallExecuteCodeBuffer(void)
```

6.12.31 rm_GetPC()

This function is called when a GetPC packet header is received over the DCC. The typical execution graph is shown in Figure 6-16 on page 6104.

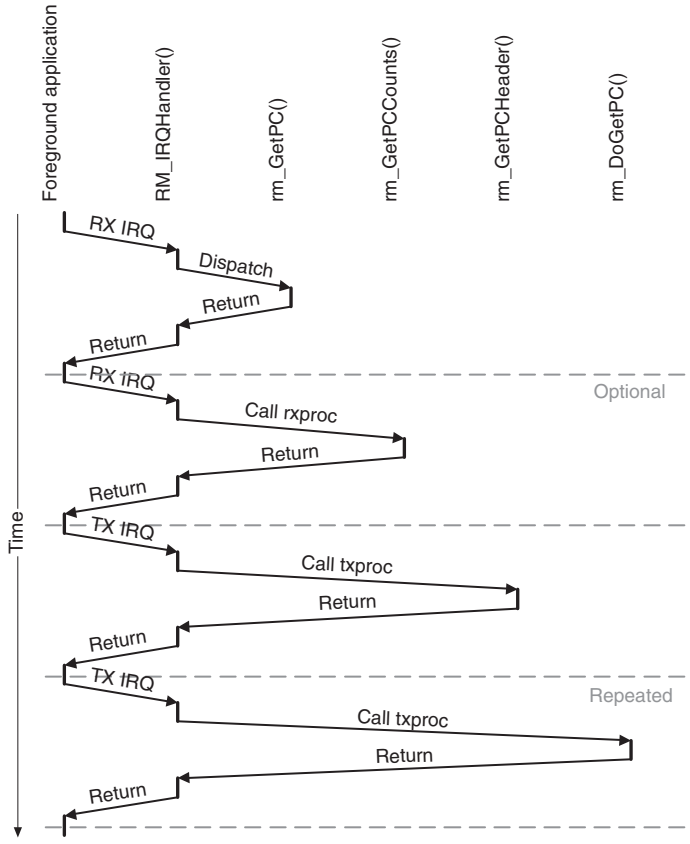

Figure 6-16 Processing a GetPC packet

Syntax

```
static void rm_GetPC(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a GetPC packet header.

Implementation

The action to be taken depends on the *length* parameter. The *length* parameter is examined as follows:

length **= 0** The packet does not have a payload, so the default sample size of 1 is used by calling `rm_GetPCCounts(1)` (see *rm_GetPCCounts()*).

length **≠ 0** The packet has a payload where the first word represents the number of samples required. The first word of the packet payload is read by calling `rm_SetRX()`, passing the address of `rm_GetPCCounts()` as the only parameter (see *rm_SetRX()* on page 664 and *rm_GetPCCounts()*).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_GETPC` is set to `TRUE` (see *RM_OPT_GETPC* on page 420).

6.12.32 rm_GetPCCounts()

This function handles GetPC requests. It is called when the first word of a GetPC packet payload is received over the DCC.

Syntax

```
static void rm_GetPCCounts(uint32 counts)
```

where:

counts A word received from the DCC, containing the number of *program counter* (pc) samples to be made.

Implementation

The word represents the number of pc samples to be taken, and is stored into `IMP_GlobalState->rwlength`. The `rm_GetPCCounts()` function then sets `IMP_GlobalState->txpending` to `rm_GetPCHeader()` to allow the actual capture of pc samples to begin (see *IMP_GlobalState* on page 69).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_GETPC` is set to `TRUE` (see *RM_OPT_GETPC* on page 420).

6.12.33 rm_GetPCHeader()

This function is called after a GetPC packet header has been received over the DCC, and has been processed by `rm_GetPC()` (see *rm_GetPC()* on page 6103).

Syntax

```
static uint32 rm_GetPCHeader(void)
```

Return value

Returns an Ok packet header.

Implementation

This function sets `IMP_GlobalState->txproc` to `rm_DoGetPC()`, and then returns an Ok packet header with a length equal to (`IMP_GlobalState->rwlength * 4`) (see *IMP_GlobalState* on page 69).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_GETPC` is set to `TRUE` (see *RM_OPT_GETPC* on page 420).

6.12.34 rm_DoGetPC()

This function is repeatedly called to send pc samples over the DCC.

Syntax

```
uint32 rm_DoGetPC(void)
```

Return value

Returns a pc sample.

Implementation

Every time this function is called, the GetPC sample counter in `IMP_GlobalState->rwlength` is decremented by 1 (see *IMP_GlobalState* on page 69). When the sample counter reaches zero, `IMP_GlobalState->txproc` is set to `rm_TX()` to indicate that no more samples have to be taken. This function then returns the pc of the foreground application (stored in lr on the stack).

If RealMonitor is in the stopped state, the value 0 is returned.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_GETPC` is set to `TRUE` (see *RM_OPT_GETPC* on page 420).

6.12.35 rm_SyncCaches()

This function is called when a SyncCaches packet header is received over the DCC. The typical execution graph is shown in Figure 6-17.

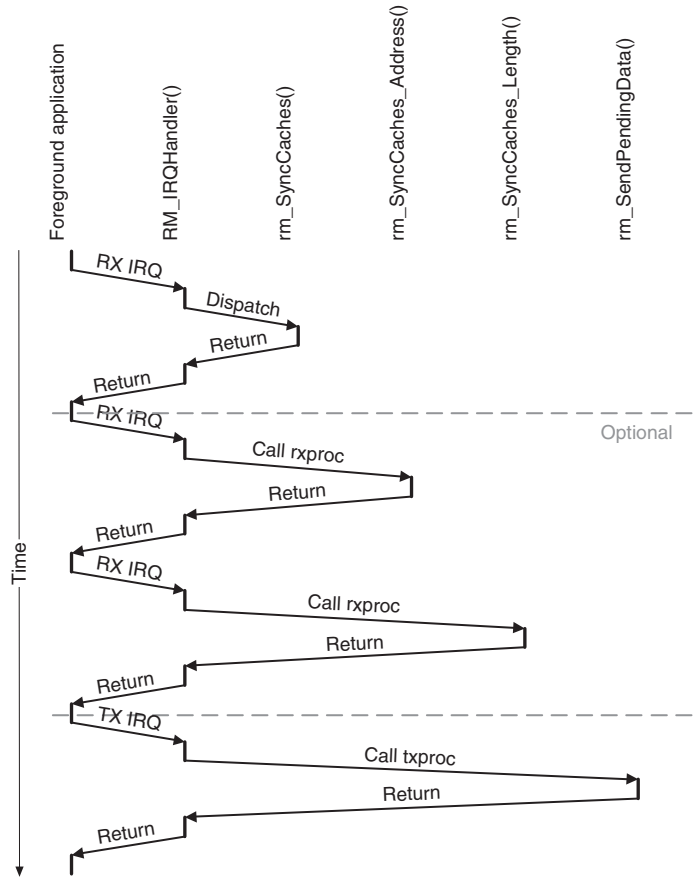

Figure 6-17 Processing of a SyncCaches packet

Syntax

```
static void rm_SyncCaches(uint32 length)
```

where:

length A word received from the DCC, containing the length field from a SyncCaches packet header.

Implementation

An Ok packet header is stored into `IMP_GlobalState->pending_data`. The data is sent when the rest of the SyncCaches packet has been received and processed (see *IMP_GlobalState* on page 69).

If the length field in the packet header equals zero, this is a request to synchronize the entire cache. The synchronization of the caches is performed by calling `rm_Cache_SyncAll()` (see *rm_Cache_SyncAll()* on page 652). Finally, the pending data (the Ok response) is scheduled to be sent by calling `rm_RXDoneSetPending(rtm_SendPendingData)`.

If the length field in the packet header is nonzero, this is a request to synchronize a specific region of the cache. A call is made to `rm_SetRX(rtm_SyncCaches_Address)` to allow the next word of the packet to be received.

Note

This function is compiled into RMTarget only when the build option `RM_OPT_SYNCCACHES` is set to `TRUE` (see *RM_OPT_SYNCCACHES* on page 422).

6.12.36 rm_SyncCaches_Address()

This function is called when the first word of a SyncCaches packet payload is received over the DCC.

Syntax

```
static void rm_SyncCaches_Address(uint32 address)
```

where:

address A word received from the DCC, containing the start address of the region that is to be synchronized.

Implementation

The address is stored into IMP_GlobalState->rwaddress to ensure that it can be accessed after the rest of the packet has been received (see *IMP_GlobalState* on page 69).

Finally, rm_SetRX(rtm_SyncCaches_Length) is called to allow the next word of the packet to be received.

Note

This function is compiled into RMTarget only when the build option RM_OPT_SYNCCACHES is set to TRUE (see *RM_OPT_SYNCCACHES* on page 422).

6.12.37 rm_SyncCaches_Length()

This function is called when the second word of a SyncCaches packet payload is received over the DCC.

Syntax

```
static void rm_SyncCaches_Length(uint32 length)
```

where:

length A word received from the DCC, containing the length, in bytes, of the region that is to be synchronized.

Implementation

The function rm_Cache_SyncRegion() is called to synchronize the specified region of the cache. Finally, the pending data (the Ok response) is scheduled by calling rm_RXDoneSetPending() with the parameter rm_SendPendingData.

Note

This function is compiled into RMTarget only when the build option RM_OPT_SYNCCACHES is set to TRUE (see *RM_OPT_SYNCCACHES* on page 422).

6.12.38 rm_SkipPacketPayload()

This function causes an entire packet payload to be discarded.

Syntax

```
static void rm_SkipPacketPayload(void)
```

Implementation

The action to be taken depends on the value of the length field in the packet header that is stored in `IMP_GlobalState->header` (see *IMP_GlobalState* on page 69). The *length* parameter is examined as follows:

length **= 0** The packet does not have a payload, so no more data has to be received. Return to the caller.

length **≠ 0** The packet has a payload, so call the function `rm_SetRX()`, passing the address of `rm_DoSkipPacketPayload()` as the single argument. This causes the payload of the packet to be discarded (see *rm_SetRX()* on page 664 and *rm_DoSkipPacketPayload()*).

Note

This function is compiled into RMTarget only when the build option `RM_OPT_GETPC` is set to `TRUE` (see *RM_OPT_GETPC* on page 420).

6.12.39 rm_DoSkipPacketPayload()

This function discards packet payloads. It is called every time a word of a packet payload is received.

Syntax

```
static void rm_DoSkipPacketPayload(uint32 data)
```

where:

data A word of data received from the DCC.

Implementation

The action to be taken depends on the value of the length field in the packet header that is stored in `IMP_GlobalState->header` (see *IMP_GlobalState* on page 69). The *`length`* parameter is examined as follows:

`length` ≤ **4** This is the last word of the packet payload, so call the function `rm_RXDone()` to prevent `rm_DoSkipPacketPayload()` from being called anymore (see *rm_RXDone()* on page 665).

`length` > **4** This is not the last word of the packet payload, so decrement the length field in `IMP_GlobalState->header` by four, and return to the caller.

Chapter 7
RealMonitor Protocol

This chapter describes the RealMonitor protocol. It contains the following sections:

7.1 About the RealMonitor protocol

Data is transferred between RMHost and RMTarget using the RealMonitor protocol. The RealMonitor protocol uses 32-bit words as its basic unit of data. This is a consequence of the protocol being implemented using the *Debug Communications Channel* (DCC). The communication between host and target is illustrated in Figure 1-1 on page 13.

The DCC is used as the physical communications medium for the following reasons:

- the hardware required to implement the DCC is present in most ARM cores
- the DCC is highly reliable, removing the need for error detection and error correction in the protocol
- the software interface to the DCC is very simple, allowing the RMTarget size to remain small.

The protocol consists of simple commands that allow you to, for example, start or stop your application, or read and write memory. More complex debugging and application-specific or OS-specific operations can be layered upon this simple protocol.

The protocol can be extended, if required, by third-party toolkits to support particular applications and operating systems. In this case, RMHost and RMTarget act simply as a messaging system that operates between the debugger and the application. These third-party message types are tagged explicitly using distinct channel numbers so they can be distinguished from ARM RealMonitor messages. Channel numbers are assigned by ARM as part of a registration policy administered by the Development Systems Business Unit third-party program. Over the RealMonitor channel, the protocol is asymmetric. That is, only the target is required to acknowledge receipt of commands.

The RealMonitor protocol handles three types of messages:

Host-to-target messages

Messages that originate in the host and demand some action by RMTarget.

Target-to-host responses

Responses by RMTarget to the commands from the host.

Asynchronous messages from the target

Messages originated in RMTarget, informing the host that an event has occurred.

Packet formats on page 76 describes the message formats, their meaning, and usage.

Note

Although the protocol is designed to support third-party messages in either direction, RealMonitor version 1.0 does not support third-party messages sent from RMHost to RMTarget.

7.2 Escape sequence handling

Escape sequences enable you to send out-of-band data across the communications link. Handling escape sequences is crucial for both synchronization and reset handling.

Escape sequence processing does not affect packet processing. Channel handlers are unaware of any escape sequence activity that might be occurring.

An escape sequence consists of two 32-bit quantities, an *escape word* followed by an *escape value*. The escape word always has the value `0xA146F098`.

There are four defined escape values:

`QUOTE (0x00000000)`

When the normal data stream contains the escape word, there must be a way to send that word so that it is not mistaken for an actual escape sequence. To do this, the sender must replace the escape word in the normal data stream with an `ESCAPE-QUOTE` escape sequence. At the other end of the link, the receiver converts the `ESCAPE-QUOTE` back into the original escape word.

Figure 7-1 shows how three words in the normal data stream are affected by escape sequence quoting. The first and third words are sent unmodified. The second word corresponds to the escape word, and is therefore converted into an `ESCAPE-QUOTE` escape sequence.

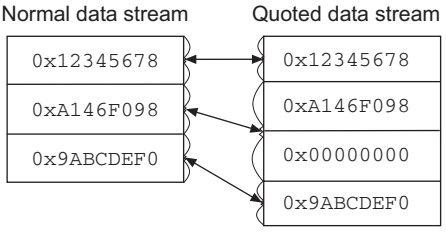

Figure 7-1 Normal compared to quoted data stream

`RESET (0x00000001)`

This escape sequence can be sent at any time to reset the communications link. Any packet that is currently being transmitted by the sender of this escape sequence must be aborted. The next word that the sender transmits must be the start of a new packet. To complete the reset, the receiver of an `ESCAPE-RESET` escape sequence must respond by sending back an `ESCAPE-GO` escape sequence.

`GO (0x00000002)`

This sequence is sent as a direct consequence of receiving an `ESCAPE-RESET` escape sequence. The `ESCAPE-GO` escape sequence completes the reset of the communications link.

`PANIC (0x00000003)`

This escape sequence is sent repeatedly by the target when it enters the panic state (see Figure 1-2 on page 17). The host must not send this escape sequence to the target.

Note

All other escape values are reserved, and must not be used.

Repeat escape words (`ESCAPE`, `ESCAPE`, `ESCAPE`…) are treated as a single escape word. This situation can occur if there is an asynchronous reset while another escape sequence is being transmitted. The number of repeat escape words allowed before a timeout error occurs is implementation-specific.

7.3 Packet formats

This section describes the message formats of the RealMonitor protocol, their meaning, and their usage. It contains the following sections:

- *Packet structure*
- *Host-to-target messages* on page 77
- *Target-to-host controller messages* on page 79.

7.3.1 Packet structure

RealMonitor protocol packets are structured as shown in Figure 7-2. There is a single-word header, followed by an optional number of payload words.

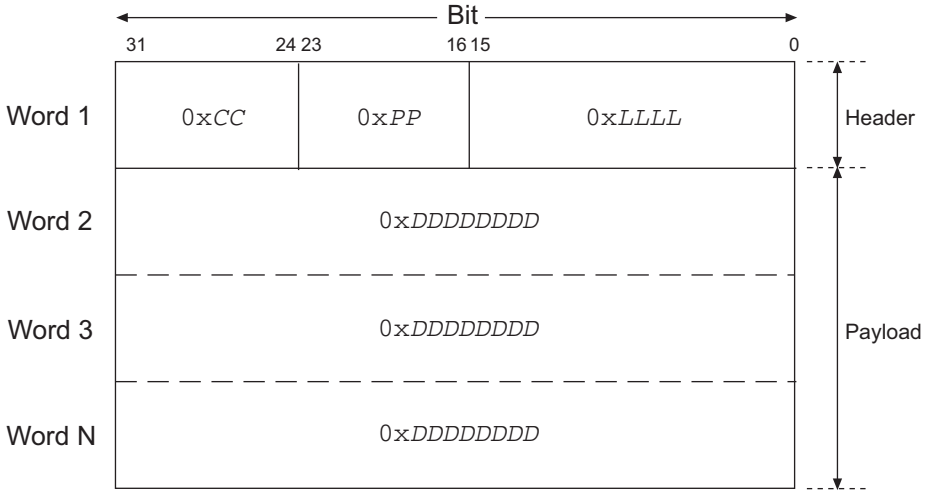

Figure 7-2 Structure of RealMonitor protocol packets

The packet header comprises three fields:

Bits 0–15 Represent the length in bytes of the payload, in the range 0 to 65535.

Bits 16–23 Represent the opcode, in the range 0 to 255.

Bits 24–31 Represent the channel number, in the range 0 to 255.

The channel number allows packets from *logically* distinct senders and receivers to be differentiated. The packets are logically distinct because they are used for different purposes by different pieces of software.

ARM assigns the following channel numbers:

Channel 0 Assigned to RealMonitor.

Channel 1 Provides a replacement for the DCC, for example, for sending data to the AXD channel viewer.

Channel 2–127

Reserved by ARM.

Channel 128–255

Available for third-party use, registered through ARM.

Whole packets from different channels can be interleaved arbitrarily while being sent across the DCC. The only restriction is that packets on the same channel must be delivered in the same order in which they were sent.

The words of packets being transmitted in opposite directions can be interleaved. This allows, for example, the target to start sending a response packet as soon as it has received the header of an incoming packet.

The meaning of the opcode field is dependent on the channel number. The RealMonitor protocol defines the opcodes that are to be used on the RealMonitor channel.

The length field in the packet header specifies the number of bytes in the payload. However, the communications medium (the DCC) is word-based (32-bit), so the actual amount of data sent is rounded up to the nearest multiple of four, that is: `((`*`payload_length`*` + 3) AND 0xFFFC)`.

Where the length field is not a multiple of four, the designer of a channel must specify which bytes in the payload are unused. On the RealMonitor channel, the most significant bytes in the last word of the payload always remain unused.

7.3.2 Host-to-target messages

Table 7-1 shows the set of messages that a host controller can send to RMTarget on channel 0.

Table 7-1 Host-to-target opcodes

Opcode	Name	Status	Description
`0x00`	NOP	Mandatory	Does nothing
`0x01`	GetCapabilities	Mandatory	Get the address of the capabilities table
`0x02`	Stop	Optional	Stop the foreground task

Table 7-1 Host-to-target opcodes (continued)

Opcode	Name	Status	Description
0x03	Go	Optional	Start the foreground task
0x04	GetPC	Optional	Return *program counter* (pc) value (profiling)
0x05	SyncCaches	Optional	Synchronize ICaches and DCaches
0x06	ExecuteCode	Optional	Execute instructions in the ExecuteCode buffer
0x07	InitializeTarget	Optional	Initialize the exception handlers on the target
0x08	ReadBytes	Mandatory	Read from memory as bytes
0x09	WriteBytes	Optional	Write to memory as bytes
0x0A	ReadHalfWords	Optional	Read from memory as halfwords
0x0B	WriteHalfWords	Optional	Write to memory as halfwords
0x0C	ReadWords	Optional	Read from memory as words
0x0D	WriteWords	Optional	Write to memory as words
0x0E	N/A	Reserved	N/A
0x0F	N/A	Reserved	N/A
0x10	ReadRegisters	Optional	Read a subset of the processor registers
0x11	WriteRegisters	Optional	Write a subset of the processor registers
0x12	ReadCPRegister	Optional	Read a coprocessor register
0x13	WriteCPRegister	Optional	Write a coprocessor register

Whenever RMTarget receives a packet on the RealMonitor channel, it must send back either an Ok or Error packet in response (except for the NOP packet, which does not require a response). The target must receive the whole packet, but it can acknowledge the packet any time after it has received the first word.

If the target receives an erroneous or unsupported opcode, it must:

1. Receive and discard the remainder of the packet.
2. Return an Error packet to the host.

3. Store an UnsupportedOpcode error into the ErrorBlock.

7.3.3 Target-to-host controller messages

Table 7-2 shows the packets that RMTarget can send to RMHost on channel 0.

Table 7-2 Target-to-host opcodes

Opcode	Name	Status	Description
0x00	NOP	Mandatory	Does nothing
0x80	Ok	Mandatory	Sent in response to a host-to-target packet
0x81	Error	Mandatory	Sent in response to a host-to-target packet
0x90	Stopped	Optional	Stopped due to a Stop packet being received
0x91	SoftBreak	Optional	Stopped due to a software breakpoint being hit
0x92	HardBreak	Optional	Stopped due to a hardware breakpoint being hit
0x93	HardWatch	Optional	Stopped due to a watchpoint being triggered
0x94	SWI	Optional	Stopped due to a semihosting SWI being hit
0xA0	Undef	Optional	Stopped due to an undefined instruction exception
0xA1	PrefetchAbort	Optional	Stopped due to a Prefetch Abort exception
0xA2	DataAbort	Optional	Stopped due to a Data Abort

The Ok and Error packets are sent by the target only in direct response to a host packet. The other target-to-host packets are triggered by asynchronous events occurring on the target, and can therefore be sent at any time. If the host sends a packet to the target for which it expects a reply, it must be prepared to receive an asynchronous packet prior to receiving its response packet.

7.4 Connecting to a target

When establishing a connection to the target, the host cannot assume anything about the current state of the target. The target can be in the running, stopped, or panic state, and might already have a queue of packets to send. If the target has been disconnected uncleanly in a previous debug session, it might even be part way through sending or receiving a packet.

To place the target into a consistent state, the host must use the `ESCAPE-RESET` and `ESCAPE-GO` escape sequences described in *Escape sequence handling* on page 74.

In some extreme cases, the target might be too confused to process any packets, or give the correct response. It is therefore important that the host has a timeout on the connection process. (A timeout of five seconds is sufficient in most situations.)

After establishing a connection to the target, the host must remove all hardware breakpoints and watchpoints. The host is then able to begin debugging activities.

Note

The host is not expected to be able to remove software breakpoints because it does not necessarily know where, in memory, any software breakpoints exist, and what the original instruction was.

7.5 Disconnecting from a target

To disconnect from a target, the host must perform the following:

1. Remove all breakpoints.
2. Remove all watchpoints.
3. Remove any prescribed filtering described using the capabilities block (see *Tag 0x00000011, Pointer to the channel filter block* on page 743).

This ensures that the target is left in a consistent state.

Caution

Forced disconnection by other means, such as the host crashing, can leave the target in an inconsistent or unrecoverable state.

7.6 RealMonitor packets

This section describes the opcodes used with different packet types:

7.6.1 NOP

This packet takes the opcode `0x00`, and has no payload. It can be sent at any time by either the target or the host. The receiver discards the packet without sending any response. The transmission of the NOP packet is illustrated in Figure 7-3.

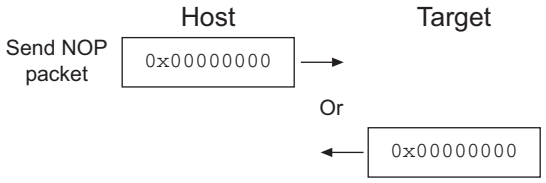

Figure 7-3 NOP packet transmission

7.6.2 GetCapabilities

This packet takes the opcode `0x01`, and has no payload. When the target receives this packet, it sends an Ok packet in response, with the base address of the capabilities table as the single word in the payload. The capabilities table is described in *rm_CapabilitiesTable* on page 615 and *Format of the capabilities table* on page 739.

If the GetCapabilities opcode is unsupported by the target, an Error packet is sent to the host. The transmission of the GetCapabilities packet is illustrated in Figure 7-4.

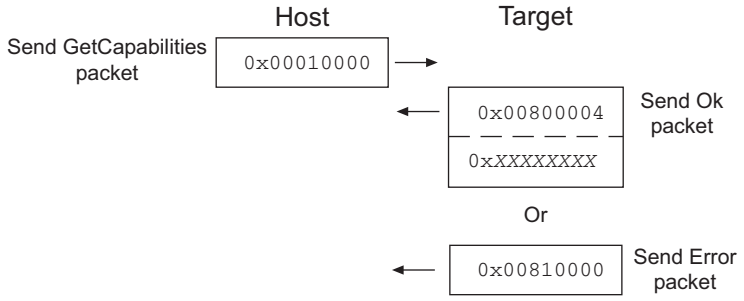

Figure 7-4 GetCapabilities packet transmission

7.6.3 Stop

This packet takes the opcode `0x02`, and has no payload. Support for this packet is optional, and its presence is indicated by bit 0 in the configuration word of the capabilities table (see *GetCapabilities* on page 713).

When the target receives this packet, it sends an Ok packet in response, and then stops execution of the foreground application. After the foreground application is stopped, the target sends an asynchronous Stopped packet to the host. If the target is already stopped when it receives this packet, it still sends an Ok packet, followed by a Stopped packet in response.

If the Stop opcode is unsupported by the target, an Error packet is sent to the host. The transmission of the Stop packet is illustrated in Figure 7-5.

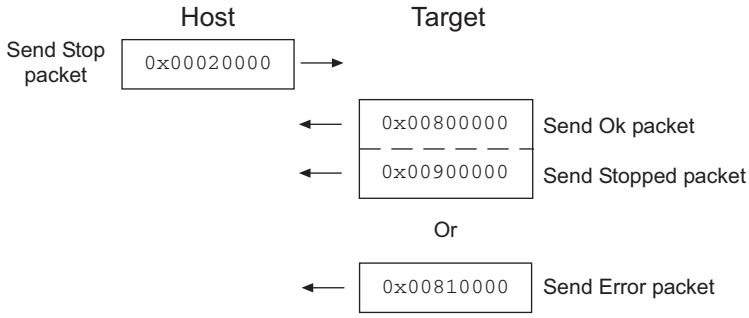

Figure 7-5 Stop packet transmission

7.6.4 Go

This packet takes the opcode `0x03`, and has no payload. Support for this packet is optional, and its presence is indicated by bit 0 in the configuration word of the capabilities table (see *GetCapabilities* on page 713).

When the target receives this packet, it resumes execution of the previously stopped foreground application.

If the Go opcode is unsupported by the target, an Error packet is sent to the host. The behavior of this opcode is undefined if the foreground application on the target is currently running. The transmission of the Go packet is illustrated in Figure 7-6 on page 715.

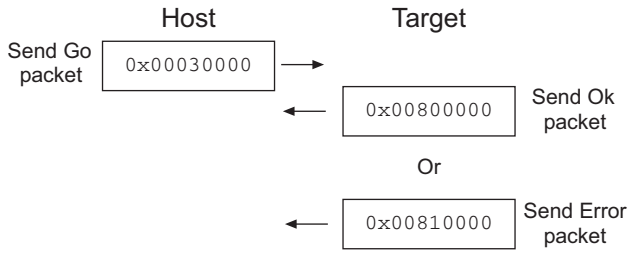

Figure 7-6 Go packet transmission

7.6.5 GetPC

This packet takes the opcode `0x04`, and has a single word of data in the payload. Support for this packet is optional, and its presence is indicated by bit 7 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the GetPC packet is illustrated in Figure 7-7.

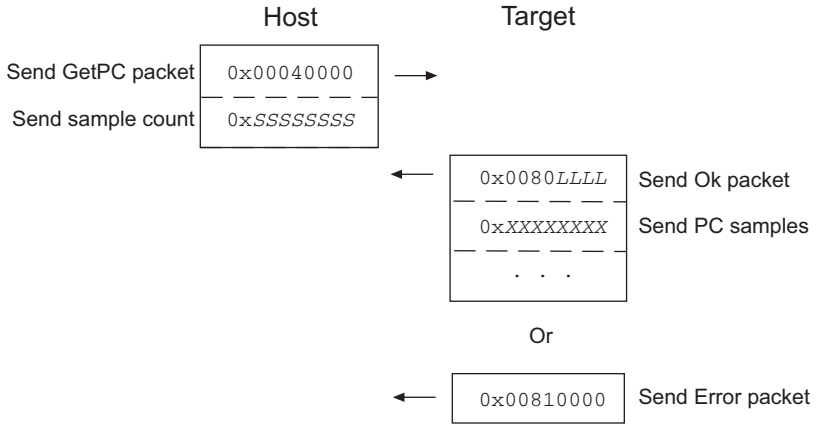

Figure 7-7 GetPC packet transmission

The GetPC command performs sample-based profiling. The single word of the payload specifies how many samples the host is requesting. A sample size of zero is invalid, and the behavior of the target is undefined if a request containing zero is ever received.

The target sends an Ok packet in response, with the payload containing a sequence of pc values. The target is allowed to return fewer samples than the host requested. The host can determine the number of samples actually returned by examining the length field in the response packet header.

The frequency with which samples are made, and the method used to gather samples, are implementation-dependent. The GetPC command produces valid results only when the target is running. When the target is stopped, the GetPC command returns zero.

Code running with IRQs disabled cannot be profiled because RealMonitor is unable to receive the GetPC command. Typically, this prevents profiling of FIQ handlers because the hardware disables IRQs when an FIQ occurs.

If the GetPC opcode is unsupported by the target, an Error packet is sent back to the host.

7.6.6 SyncCaches

This packet takes the opcode `0x05`. Support for this packet is optional, and its presence is indicated by bit 8 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the SyncCaches packet is illustrated in Figure 7-8.

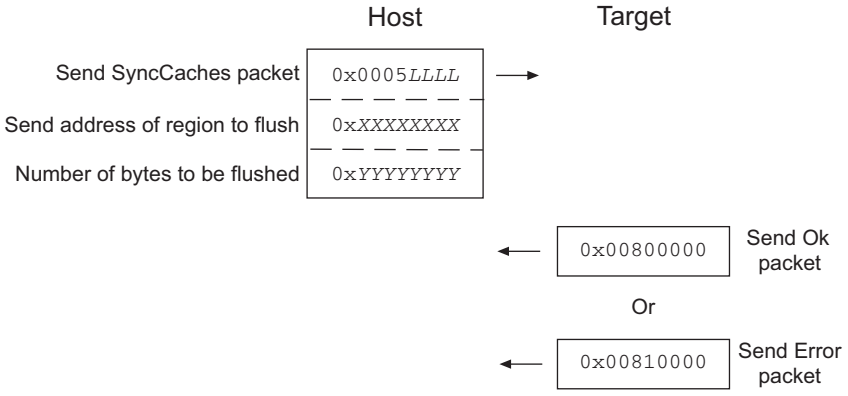

Figure 7-8 SyncCaches packet transmission

The first word in the payload gives the start address of the region to be synchronized. The start address must be word-aligned, and the behavior of the target is undefined if a non word-aligned address is used.

The second word in the payload gives the number of bytes to synchronize, and must be a multiple of four. The behavior of the target is undefined if the number of bytes is not a multiple of four.

The SyncCaches opcode performs the following actions:

1. Cleans the DCache in at least the range X to X+Y–1.

2. Flushes the ICache in at least the range X to X+Y–1.

Any other caches, such as the prefetch buffer and the *Translation Lookaside Buffer* (TLB), are not expected to be flushed (although a specific implementation might choose to do so anyway).

A specific implementation can clean/flush more than the requested memory range, and if it cleans/flushes all of the cache, it can discard the X and Y words in the payload. If the host omits the X and Y fields of the packet, the target is expected to clean/flush the entire cache.

If the SyncCaches opcode is supported by the target, the WriteBytes, WriteHalfWords, and WriteWords opcodes do not flush/clean any caches, and it is the job of the host to issue any necessary SyncCaches opcodes.

If the SyncCaches opcode is not supported by the target, the WriteBytes, WriteHalfWords, and WriteWords opcodes must automatically clean/flush any caches in the affected memory range, if necessary.

7.6.7 ExecuteCode

This packet takes the opcode `0x06`. Support for this packet is optional, and its presence is indicated by bit 9 in the configuration word of the capabilities table, and has no payload (see *GetCapabilities* on page 713).

The host controller determines what target resources are available to it from the Execution Information Block (see *Tag 0x00000015, Pointer to the execution information block* on page 746). A pointer to this structure is included in the capabilities block.

Before the ExecuteCode opcode is called, the host must have downloaded some code (and any other data required) into the ExecuteCode buffer using the normal write memory opcodes. After the ExecuteCode opcode has completed, the host can use the read memory opcodes to retrieve any results that the code might have written to the buffer.

The ExecuteCode opcode causes execution to transfer to the first address in the ExecuteCode buffer. The ARM code in the buffer is executed in an unspecified privileged processor mode. The code can corrupt r0–r3, and must preserve all other registers. The code must exit with a `BX LR` instruction (or an `LDM` instruction if it saved any registers on the stack).

The Execute Code opcode returns an Ok response packet after the code has been executed. An Error response packet is returned if the ExecuteCode opcode is not supported. The transmission of the ExecuteCode packet is illustrated in Figure 7-9 on page 718.

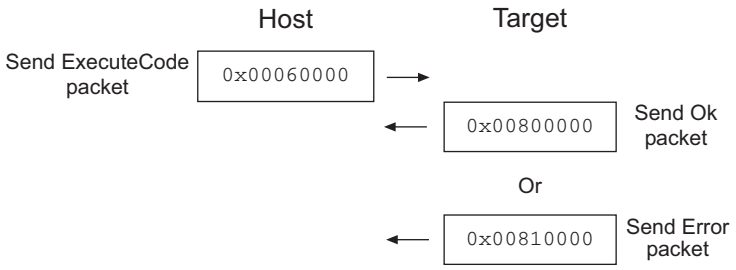

Figure 7-9 ExecuteCode packet transmission

7.6.8 InitializeTarget

This packet takes the opcode `0x07`, and has no payload. Support for this packet is optional, and its presence is indicated by bit 20 in the configuration word of the capabilities table (see *GetCapabilities* on page 713).

This packet is sent immediately before the host downloads a new image. It causes the target to initialize any exception vectors that the previous program might have used.

When the target receives this packet, it sends an Ok packet in response. If the InitializeTarget opcode is unsupported by the target, an Error packet is sent to the host. The transmission of the InitializeTarget packet is illustrated in Figure 7-10.

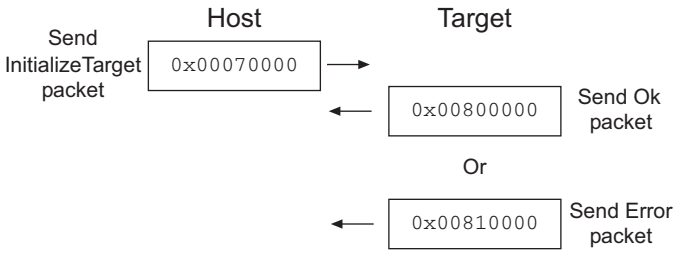

Figure 7-10 InitializeTarget packet transmission

7.6.9 ReadBytes

This packet takes the opcode `0x08`, and takes between zero and two words of data in the payload.

Caution

When reading more than one byte at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

The transmission of the ReadBytes packet is illustrated in Figure 7-11.

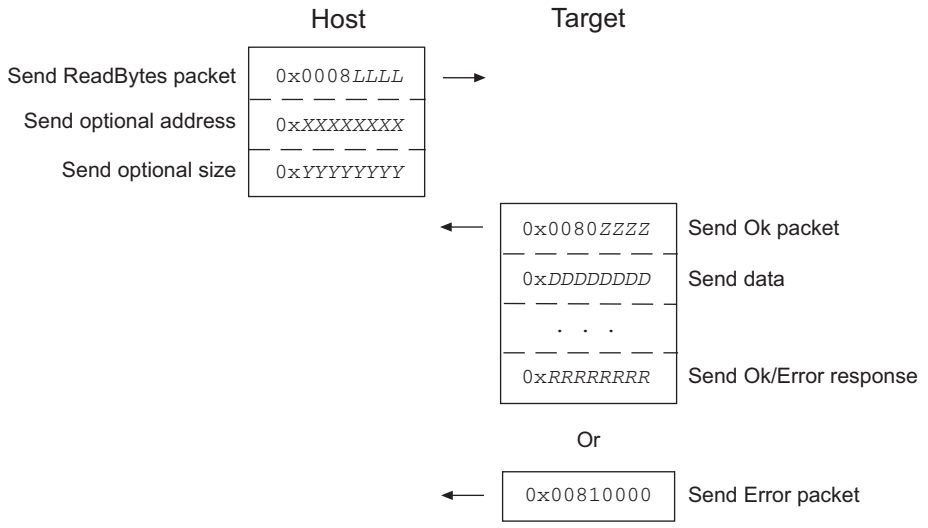

Figure 7-11 ReadBytes packet transmission

The first word of the payload, if present, gives the start address in the memory of the target. If the start address is not specified, RealMonitor uses the address that the previous Read or Write memory operation would have accessed next. The start address can be omitted only when the size is also omitted.

The second word in the payload, if present, gives the number of bytes to be read. If the number of bytes to read is not specified, the default, one, is used. A value of zero is invalid, and the behavior of the target is undefined if a size of zero is received.

If the ReadBytes operation is supported by the target, an Ok packet is sent back to the host containing the data in the payload. At the end of the payload, an additional word specifies whether the ReadBytes operation completed without error.

If the last word of the response packet equals `0x00800000`, the read completed successfully. If the last word equals `0x00810000`, an error occurred, and the host must query the error block to determine exactly what happened (see *Tag 0x00000013, Pointer to the error block* on page 744). Typically, any reported error is due to a Data Abort occurring during the read. In this case, all data returned in the payload after the first Data Abort occurred is undefined, and must be discarded by the host.

The target reads the memory one byte at a time, starting from the specified address, then incrementing the address by one for each subsequent byte. As the bytes are read, they are packed into a word by progressive left shifts into the least significant eight bits of that word. A word of data is sent over the DCC when all four bytes contain data, or there

is no more data to be read. Where the last word contains less than four bytes of useful data, the value of the unused bytes is undefined. See Figure 7-12 for an example of this process.

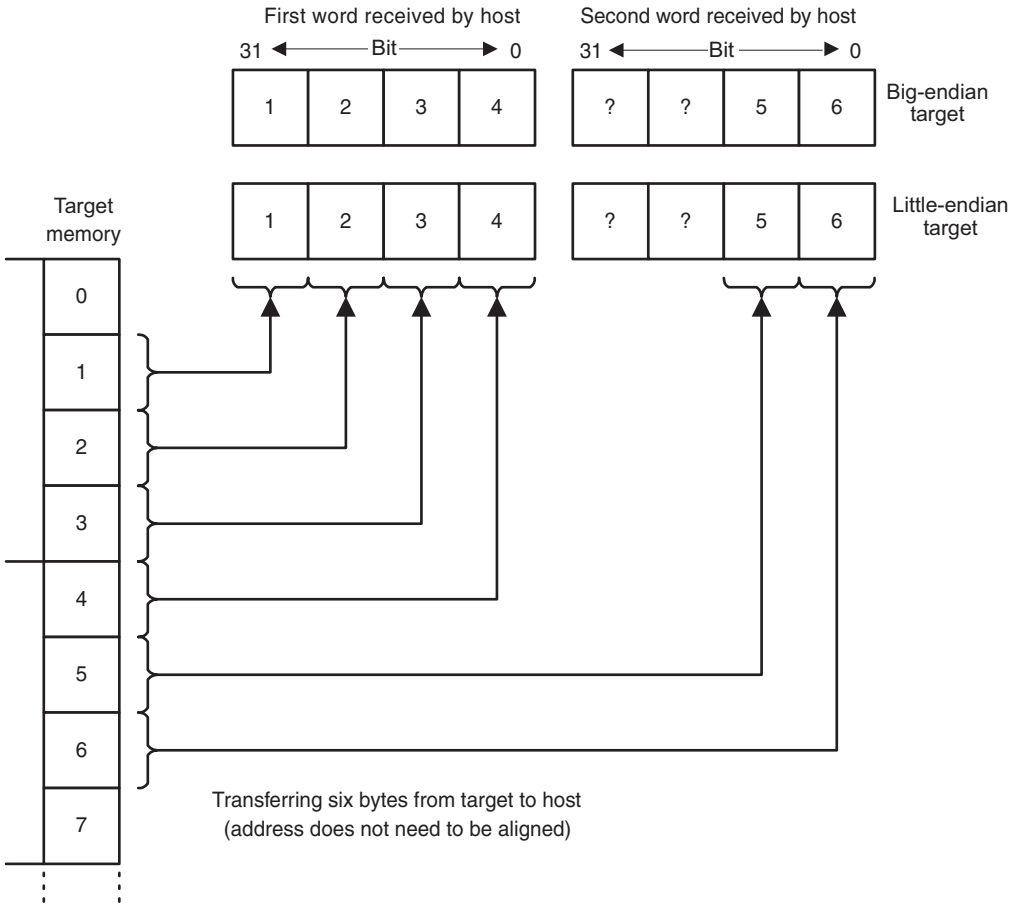

Figure 7-12 Transferring six bytes from target to host

7.6.10 WriteBytes

This packet takes the opcode `0x09`, and takes a minimum of one word in the payload.

Caution

When writing more than one byte at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

Support for this packet is optional, and its presence is indicated by bit 2 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the WriteBytes packet is illustrated in Figure 7-13.

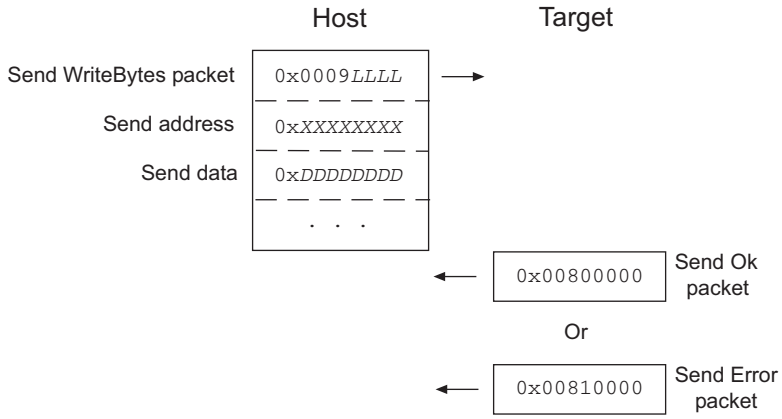

Figure 7-13 WriteBytes packet transmission

The first word in the payload gives the start address in memory of where the subsequent words of the payload will be written. The start address does not have to be word-aligned. The number of bytes of data to be written is equal to the length field in the packet header minus four.

For each word of data that is received, the target writes the least significant eight bits to memory, then shifts the word right by eight bits. This process continues until all four bytes of data have been written. The last word to be received might contain less than four bytes of data. The value of the unused bytes is undefined and is ignored by the target. See Figure 7-14 on page 722 for an example of this process.

If the WriteBytes operation completes successfully, the target sends an Ok packet to the host. If the WriteBytes operation is unsupported by the target, or an error occurs during its execution, an Error packet is sent back to the host. To discover the exact cause of the error, the host queries the error block (see *Tag 0x00000013, Pointer to the error block*

on page 744). Typically, errors during the execution of a command are caused by a Data Abort when writing to memory. As soon as the first Data Abort occurs, no more data from this packet is written to memory.

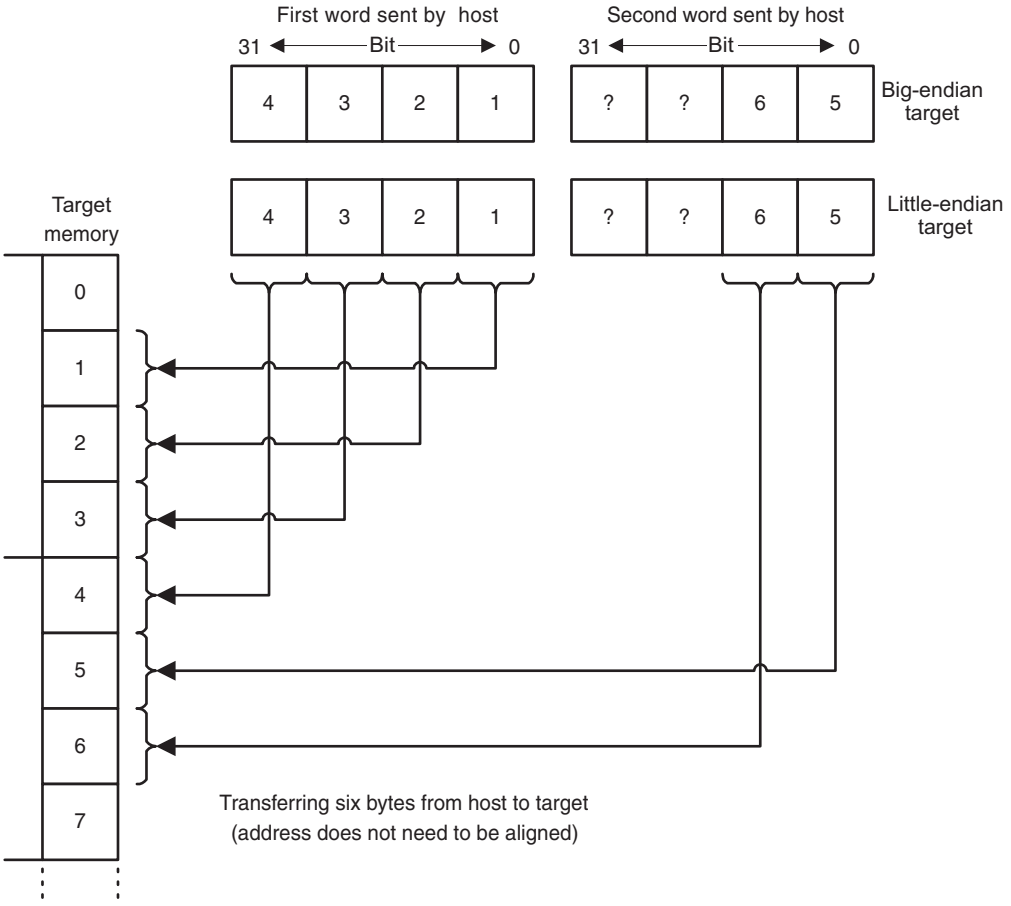

Figure 7-14 Transferring six bytes from host to target

7.6.11 ReadHalfWords

This packet takes the opcode `0x0A`, and takes between zero and two words of data in the payload.

Caution

When reading more than one halfword at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

Support for this packet is optional, and its presence is indicated by bit 3 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the ReadHalfWords packet is illustrated in Figure 7-15.

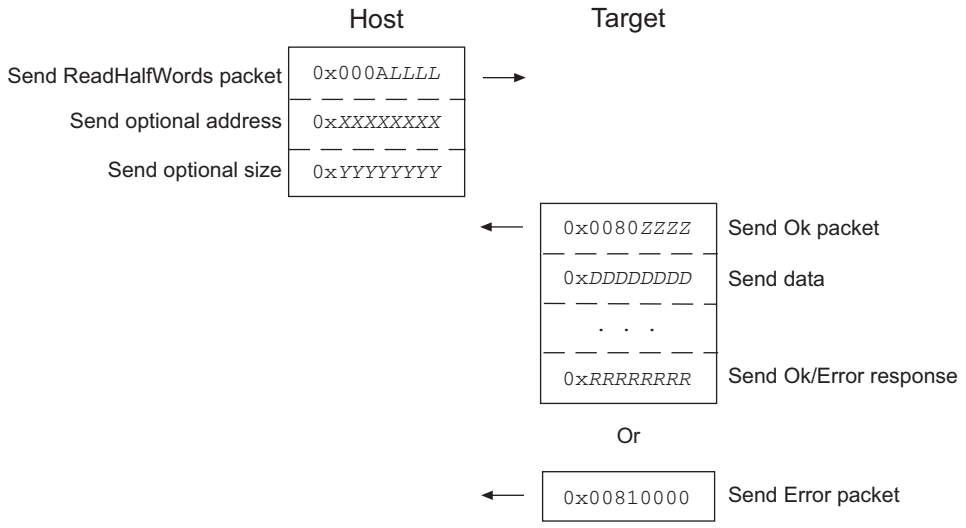

Figure 7-15 ReadHalfWords packet transmission

The first word of the payload, if present, gives the start address in the memory of the target. If the start address is not specified, RealMonitor uses the address that the previous Read or Write memory operation would have accessed next. The start address can be omitted only when the size is also omitted. The start address must be aligned on a halfword boundary. In cases where the address is unaligned, the behavior is undefined.

The second word in the payload, if present, gives the number of bytes to be read. If the number of bytes to read is not specified, the default of two is used. The number of bytes to read must be a multiple of two, and must be greater than zero. The behavior of the target is undefined if a value of zero, or a nonmultiple of two, is ever received.

If the ReadHalfWords operation is supported by the target, an Ok packet is sent back to the host containing the data in the payload. At the very end of the payload, an additional word specifies whether the ReadHalfWords operation completed without error.

If the last word of the payload equals `0x00800000`, the read completed successfully. If the last word equals `0x00810000`, an error occurred, and the host must query the error block to determine exactly what went wrong (see *Tag 0x00000013, Pointer to the error block* on page 744). Typically, any reported error is due to a Data Abort occurring during the read. In this case, all data returned in the payload after the first Data Abort occurred is undefined, and must be discarded by the host.

The target reads the memory, one halfword at a time, starting from the specified address. As the halfwords are read, they are packed into a word by progressive left shifts into the least significant 16 bits of that word. A word of data is sent over the DCC when both halfwords contain data, or there is no more data to be read. Where the last word contains less than two halfwords of useful data, the value of the unused halfword is undefined. See Figure 7-16 on page 725 for an example of this process.

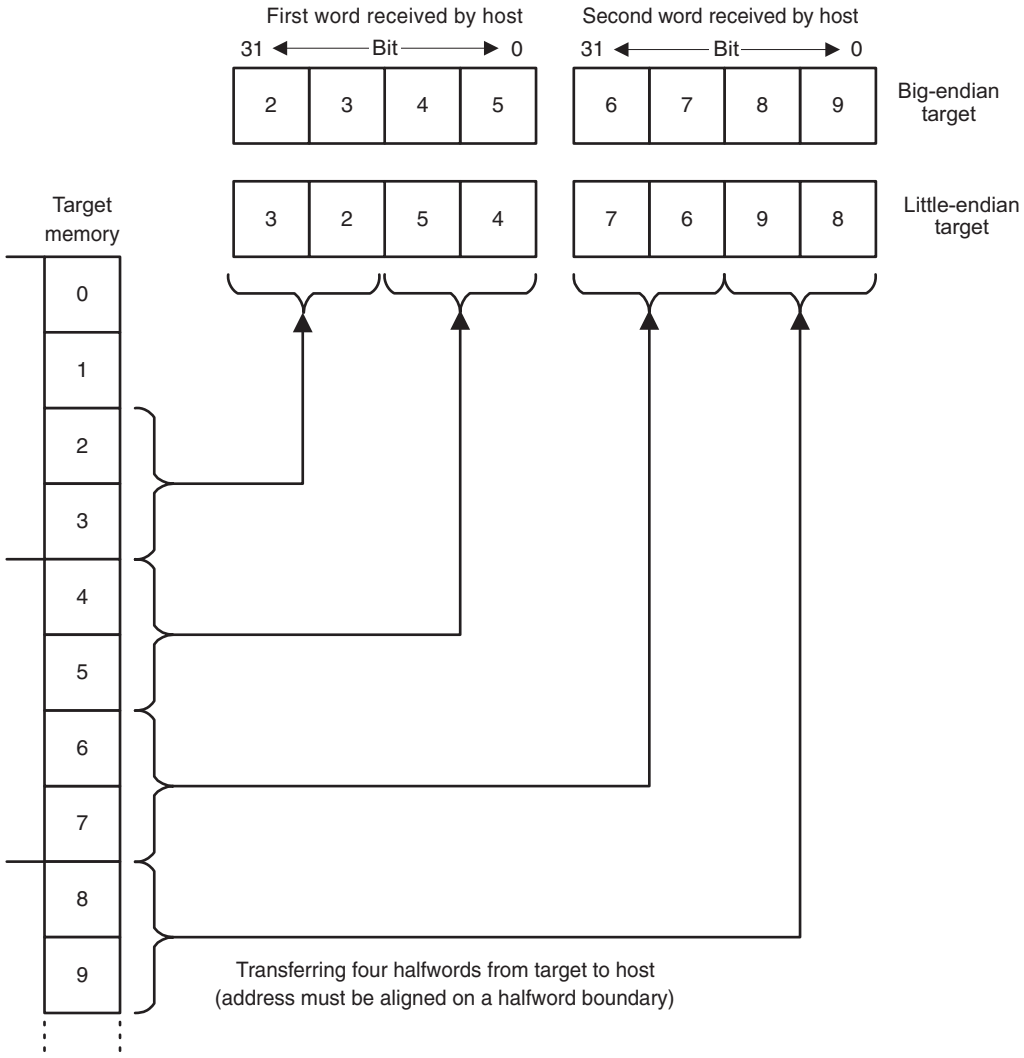

Figure 7-16 Transferring four halfwords from target to host

7.6.12 WriteHalfWords

This packet takes the opcode `0x0B`, and takes a minimum of one word in the payload.

Caution

When writing more than one halfword at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

Support for this packet is optional, and its presence is indicated by bit 4 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the WriteHalfWords packet is illustrated in Figure 7-17.

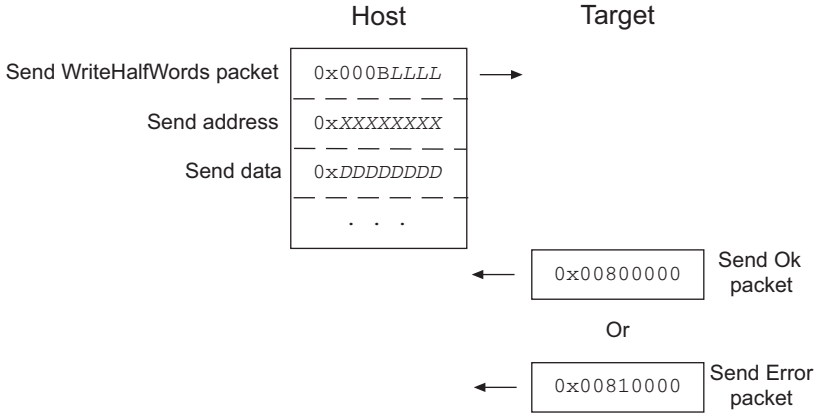

Figure 7-17 WriteHalfWords packet transmission

The first word in the payload gives the start address in memory of where the subsequent words of the payload are written. The start address must be aligned on a halfword boundary. The number of bytes of data to be written is equal to the length field in the packet header minus two, and must be a multiple of two. The behavior of the target is undefined if the number of bytes is not a multiple of two, or is zero.

For each word of data that is received, the target writes the least significant 16 bits to memory, then shifts the word right by 16 bits. This process repeats so that both halfwords of data have been written. The last word to be received might contain less than two half-words of data. The value of the unused half-word is undefined and is ignored by the target. See Figure 7-18 on page 728 for an example of this process.

If the WriteHalfWords operation completes successfully the target sends an Ok packet to the host. If the WriteHalfWords operation is unsupported by the target, or an error occurs during its execution, an Error packet is sent back to the host. To discover the exact cause of the error, the host must query the error block (see *Tag 0x00000013, Pointer to the error block* on page 744).

Typically, errors during the execution of a command are caused by a Data Abort when writing to memory. As soon as the first Data Abort occurs, no more data from this packet is written to memory.

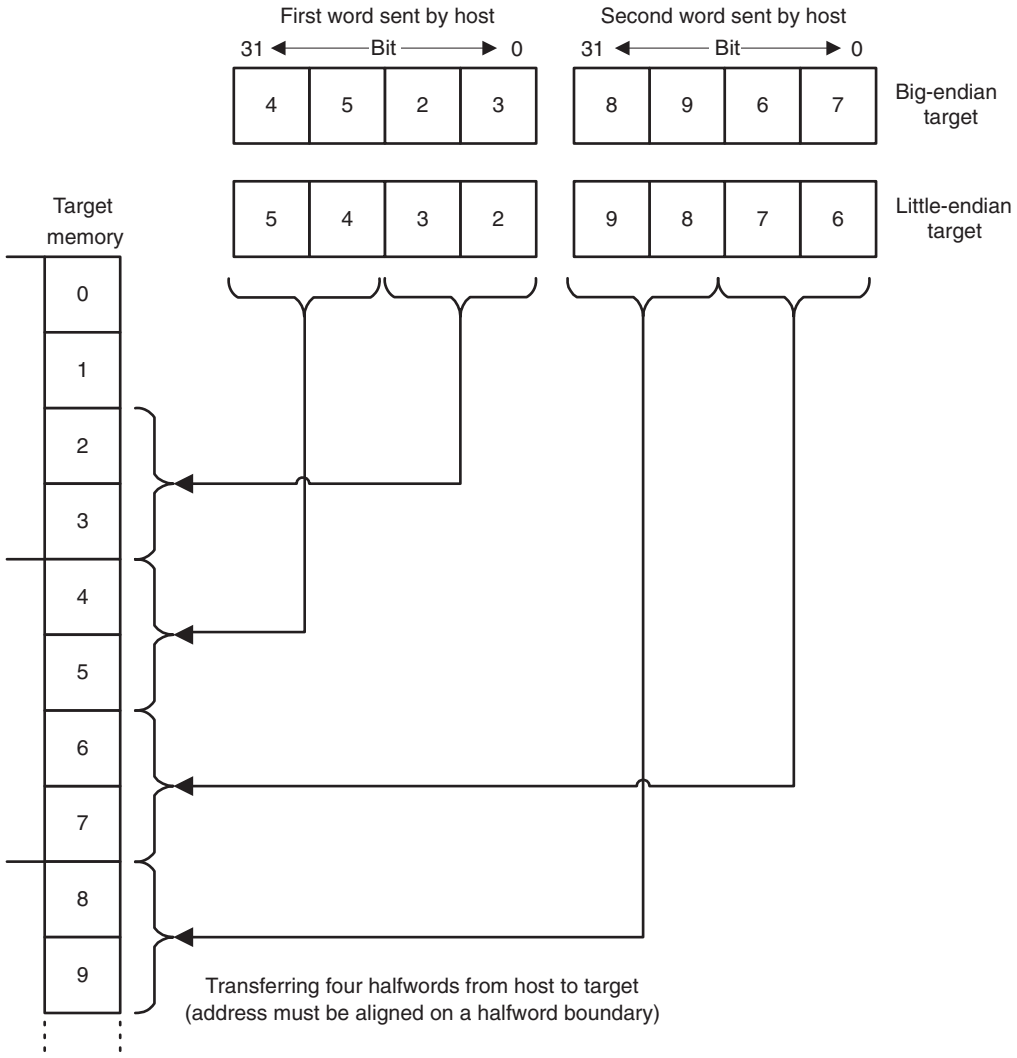

Figure 7-18 Transferring four halfwords from host to target

7.6.13 ReadWords

This packet takes the opcode `0x0C`, and takes between zero and two words of data in the payload.

Caution

When reading more than one word at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

Support for this packet is optional, and its presence is indicated by bit 5 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the ReadWords packet is illustrated in Figure 7-19.

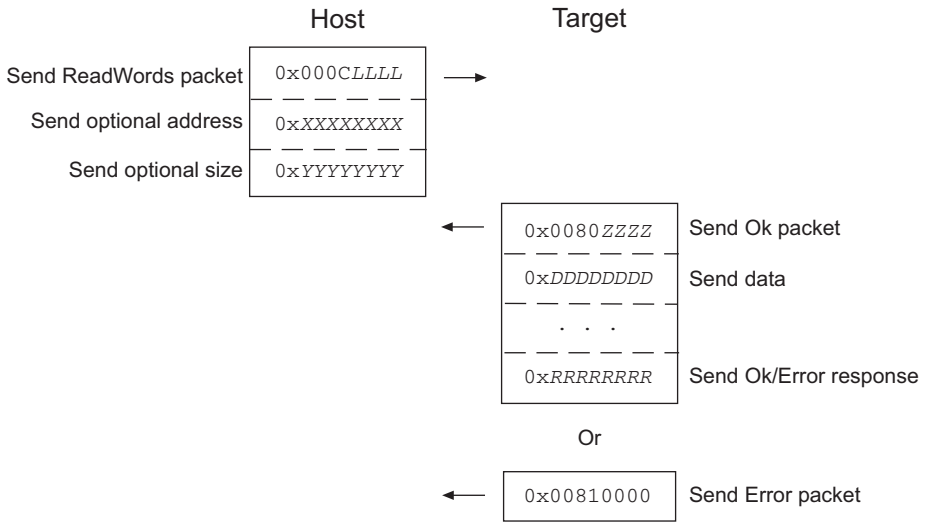

Figure 7-19 ReadWords packet transmission

The first word of the payload, if present, gives the start address in the memory of the target. If the start address is not specified, RealMonitor uses the address that the previous Read or Write memory operation would have accessed next. The start address can be omitted only when the size is also omitted. The start address must be aligned on a word boundary. In all other cases, the behavior is undefined.

The second word in the payload, if present, gives the number of bytes to be read. If the number of bytes to read is not specified, the default of four is used. The number of bytes to be read must be a multiple of four. The behavior of the target is undefined if a size of zero or nonmultiple of four is ever received.

If the ReadWords operation is supported by the target an Ok packet is sent back to the host containing the data in the payload. At the very end of the payload an additional word specifies whether the ReadWords operation completed without error.

If the last word of the payload equals `0x00800000` then the read completed successfully. If the last word equals `0x00810000`, an error occurred, and the host must query the error block to determine exactly what happened. Typically, any reported errors are due to a Data Abort occurring during the read. In this case, all data returned in the payload after the first Data Abort occurred is undefined, and is discarded by the host.

The target reads the memory one word at a time, starting from the specified address. The words of data are sent over the communication link in the order in which they were read. See Figure 7-20 for an example of this process.

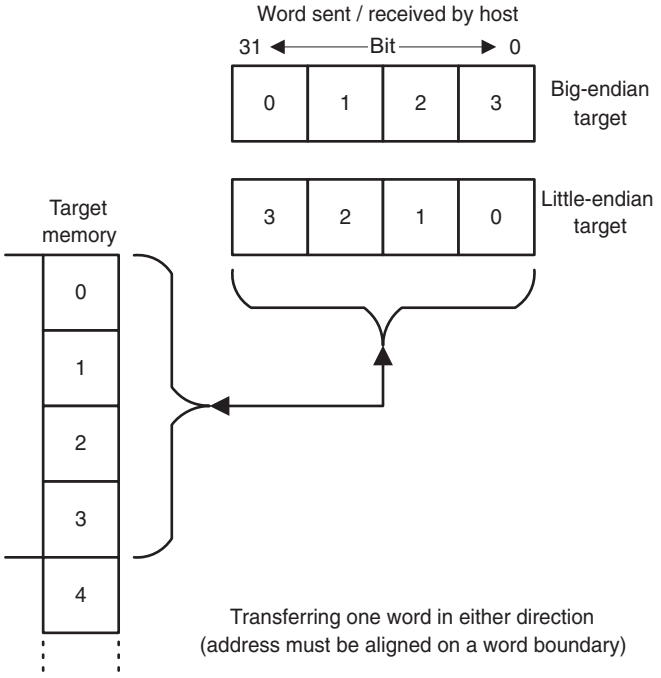

Figure 7-20 Transferring one word in either direction

7.6.14 WriteWords

The WriteWords packet takes the opcode `0x0D`, and takes a minimum of one word in the payload.

Caution

When writing more than one word at a time, memory accesses are not atomic. This can lead to inconsistencies in the system being debugged.

Support for this packet is optional, and its presence is indicated by bit 6 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the WriteWords packet is illustrated in Figure 7-21.

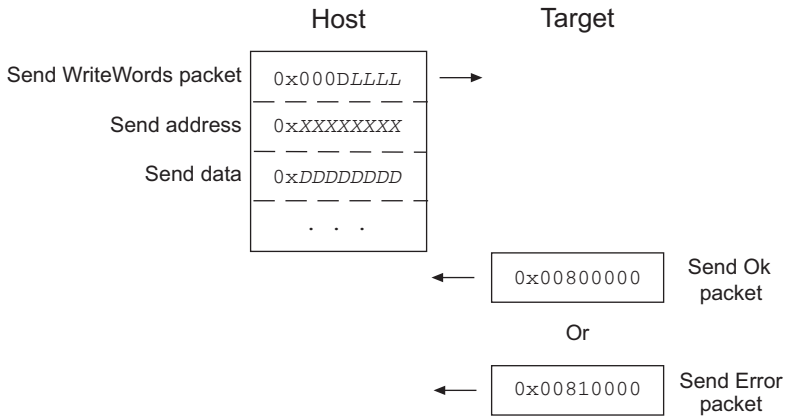

Figure 7-21 WriteWords packet transmission

The first word in the payload gives the start address in memory of where the subsequent words of the payload will be written. The start address must be word-aligned, and the behavior of the target is undefined if a non word-aligned address is received. The number of bytes of data to be written is equal to the length field in the packet header minus four, and must be a multiple of four.

As each word of data is received, it is written to memory, starting from the specified start address. See Figure 7-20 on page 730 for an example of this process. If the WriteWords operation completes successfully, the target sends an Ok packet to the host. If the WriteWords operation is unsupported by the target, or an error occurs during its execution, an Error packet is sent back to the host. To discover the exact cause of the error, the host must query the error block (see *Tag 0x00000012, Pointer to the register block* on page 743).

Typically, errors during the execution of a command are caused by a Data Abort when writing to memory. As soon as the first Data Abort occurs, no more data from this packet is written to memory.

7.6.15 ReadRegisters

This packet takes the opcode `0x10` and either one or two words in the payload. Support for this packet is optional, and its presence is indicated by bit 10 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the ReadRegisters packet is illustrated in Figure 7-22.

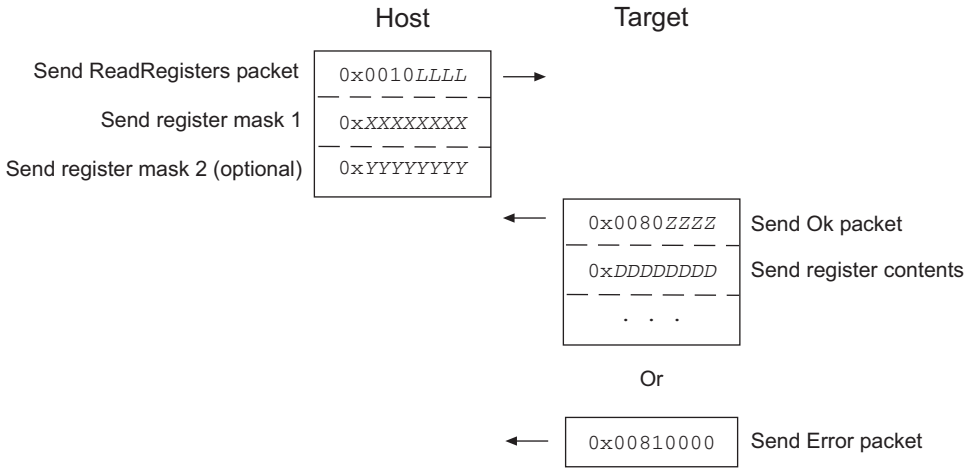

Figure 7-22 ReadRegisters packet transmission

The payload contains masks to indicate which registers are returned to the host. Bits 0 to 31 in the first word correspond to registers 0 to 31. Bits 0 to 31 of the second word (if present) correspond to registers 32 to 63. A register is returned only if its corresponding bit is set. The registers are numbered starting from zero, and are in the same order as given in the registers block (see *Tag 0x00000012, Pointer to the register block* on page 743). All unallocated register numbers are reserved.

Processor register access is only supported when the foreground application is stopped. The behavior of the target is undefined if this packet is received while the target is running.

Note

This packet is currently unsupported by both RMHost and RMTarget. If you attempt to use this packet, you receive an Error packet in reply.

7.6.16 WriteRegisters

This packet takes the opcode `0x11`, and between two and 66 words in the payload. Support for this packet is optional, and its presence is indicated by bit 11 in the configuration word of the capabilities table (see *GetCapabilities* on page 713). The transmission of the WriteRegisters packet is illustrated in Figure 7-23.

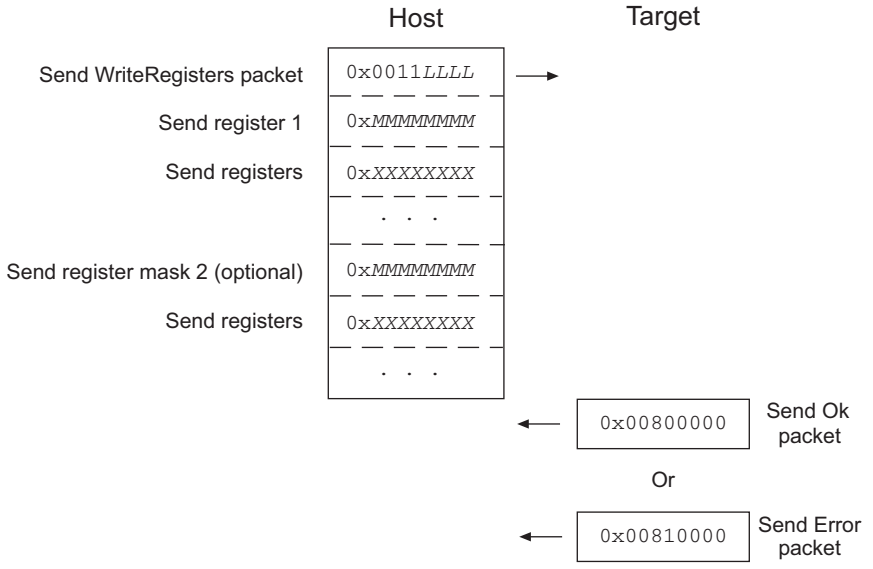

Figure 7-23 WriteRegisters packet transmission

The first word of the payload is a mask for registers 0 to 31. For each bit set in the mask, there is a subsequent word of data in the payload that contains the new value for the corresponding register. There is a second, optional mask for registers 32 to 63, followed by the data for these registers.

If the WriteRegisters operation is unsupported by the target, it responds by sending an Error packet to the host, and the payload of the WriteRegisters packet is discarded.

Processor register access is supported only when the foreground application is stopped. The behavior of the target is undefined if this packet is received while the target is running.

Note

This packet is currently unsupported by both RMHost and RMTarget. If you attempt to use this packet, you receive an Error packet in reply.

7.6.17 ReadCPRegister

This packet takes the opcode `0x12`, and has a payload to describe the coprocessor access. Support for this packet is optional, and its presence is indicated by bit 12 in the configuration word of the capabilities table (see *GetCapabilities* on page 713).

Note

This packet is reserved for future use by ARM. It is currently unsupported by both RMHost and RMTarget. If you attempt to use this packet, you receive an error packet in reply.

7.6.18 WriteCPRegister

This packet takes the opcode `0x13`, and has a payload that both describes the coprocessor access, and provides the data to be written. Support for this packet is optional, and its presence is indicated by bit 13 in the configuration word of the capabilities table (see *GetCapabilities* on page 713).

Note

This packet is reserved for future use by ARM. It is currently unsupported by both RMHost and RMTarget. If you attempt to use this packet, you receive an error packet in reply.

7.6.19 Ok

This packet takes the opcode `0x80`, and has an optional payload. This packet is sent after the target has received the header of a host-to-target packet, and has determined that the opcode is both valid and supported by the current build. The payload contains any data returned from the command. The transmission of the Ok packet is illustrated in Figure 7-24.

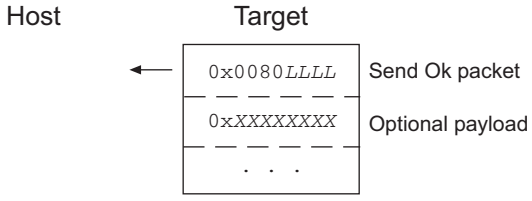

Figure 7-24 Ok packet transmission

7.6.20 Error

This packet takes the opcode `0x81`, and has no payload. It is sent after the target has received the header of a host-to-target packet, and has determined that the opcode is invalid or unsupported in the current build.

To find the exact details of the error, the host must query the error block (see Figure 7-35 on page 745). The transmission of the Error packet is illustrated in Figure 7-25.

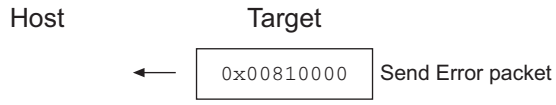

Figure 7-25 Error packet transmission

7.6.21 Stopped

This packet takes the opcode `0x90`, and has no payload. It is sent asynchronously when the foreground task on the target has been stopped due to the host having sent a Stop packet previously. The Stop packet will have already been acknowledged with an Ok packet. The transmission of the Stopped packet is illustrated in Figure 7-26.

Figure 7-26 Stopped packet transmission

7.6.22 SoftBreak

This packet takes the opcode `0x91`, and has no payload. It is sent by the target only when a software breakpoint occurs. After hitting a software breakpoint, the foreground application is stopped, but FIQs and IRQs continue to be serviced (if enabled when the breakpoint occurred).

Software breakpoints are available only when using an RMTarget that has been built to include software breakpoint support. The transmission of the SoftBreak packet is illustrated in Figure 7-27.

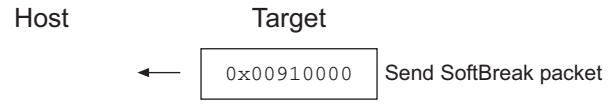

Figure 7-27 SoftBreak packet transmission

7.6.23 HardBreak

This packet takes the opcode `0x92`, and has no payload. It is sent by the target only when a hardware breakpoint occurs. After hitting a hardware breakpoint, the foreground application is stopped, but FIQs and IRQs continue to be serviced (if enabled when the breakpoint occurred).

Hardware breakpoints are available only when using an RMTarget that has been built to include hardware breakpoint support, and when RMTarget is running on a processor with EmbeddedICE-RT logic. The transmission of the HardBreak packet is illustrated in Figure 7-28.

Figure 7-28 HardBreak packet transmission

7.6.24 HardWatch

This packet takes the opcode `0x93`, and has no payload. This packet is sent by the target only when a watchpoint occurs. After hitting a watchpoint, the foreground application is stopped, but FIQs and IRQs continue to be serviced (if enabled when the watchpoint occurred).

Watchpoints are available only when using an RMTarget that has been built to include watchpoint support, and when RMTarget is running on a processor with EmbeddedICE-RT logic. The transmission of the HardWatch packet is illustrated in Figure 7-29.

Figure 7-29 HardWatch packet transmission

7.6.25 SWI

This packet takes the opcode `0x94`, and has no payload. Targets that have been built to include support for SWI semihosting send a SWI packet to the host whenever a semihosting SWI instruction is executed. After a semihosting SWI is encountered, the foreground application is stopped, but FIQs and IRQs continue to be serviced (if enabled when the SWI occurred).

On receiving a SWI packet, it is the responsibility of the host controller to interrogate the target, and to perform any actions needed to process the SWI and restart the foreground application. The transmission of the SWI packet is illustrated in Figure 7-30.

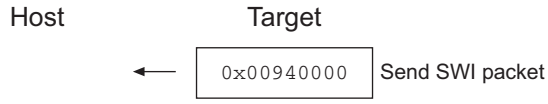

Figure 7-30 SWI packet transmission

7.6.26 Undef

This packet takes the opcode `0xA0`, and has no payload. This packet is generated by the target when an undefined instruction exception occurs. The transmission of the Undef packet is illustrated in Figure 7-31.

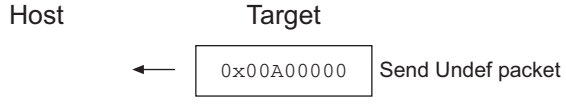

Figure 7-31 Undef packet transmission

7.6.27 PrefetchAbort

This packet takes the opcode `0xA1`, and has no payload. This packet is generated by the target when a Prefetch Abort occurs. Prefetch Aborts can be generated by either errors in the application, or software or hardware breakpoints. If the target has the capability to distinguish breakpoints from other Prefetch Aborts, it must send the SoftBreak and HardBreak packets in preference to this one. The transmission of the PrefetchAbort packet is illustrated in Figure 7-32.

Figure 7-32 PrefetchAbort packet transmission

7.6.28 DataAbort

This packet takes the opcode `0xA2`, and has no payload. This packet is generated by the target when a Data Abort exception occurs. Data Aborts can be generated by either errors in the application, or watchpoints. If the target has the capability to distinguish watchpoints from other Data Aborts, it must send the HardWatch packet in preference to this one. The transmission of the DataAbort packet is illustrated in Figure 7-33.

Figure 7-33 DataAbort packet transmission

7.7 Format of the capabilities table

A pointer to this block is returned by RMTarget in response to a GetCapabilities packet from RMHost. The capabilities table is a set of two or more tag-value pairs that hold the configuration of RMTarget. The tag field is a word that uniquely identifies the contents of the value field. The value field is a word that, depending on the actual tag, contains either the data itself, or a pointer to the data.

From the point of view of the host, the capabilities table is considered read-only, regardless of whether it actually resides in RAM or ROM on the target. For capabilities that need to be writable by the host, the value field always contains a pointer to the actual data in RAM.

Table 7-3 lists the current set of capability tags. All tags with bit 31 clear are reserved for use by ARM. All tags with bit 31 set are free for use by third parties. The Pointer access column shows whether the item being pointed at by a tag-value pair is readable and/or writable.

Table 7-3 Capability tags

Tag	Value type	Pointer access	Status	Description
`0x00000000`	Data	N/a	Mandatory (must be the last entry in table)	Marks the end of the capabilities table.
`0x00000001`	Data	N/a	Mandatory (must be the first entry in table)	RealMonitor version number.
`0x00000002`	Data	N/a	Mandatory	Configuration word.
`0x00000003`	Pointer	Read-only	Optional	Pointer to the build identifier.
`0x00000010`	Pointer	Depends on bit 19 in the configuration word	Start-stop debug only	Pointer to the RealMonitor state byte.
`0x00000011`	Pointer	Read/write	Optional	Pointer to the channel filter block.
`0x00000012`	Pointer	Read-only	Start-stop debug only	Pointer to the registers block.

Table 7-3 Capability tags (continued)

Tag	Value type	Pointer access	Status	Description
0x00000013	Pointer	Read-only	Mandatory	Pointer to the error block.
0x00000014	Pointer	Read-only	Start-stop debug only	Pointer to the register accessibility block.
0x00000015	Pointer	Read-only	Optional	Pointer to the execution information block.
0x00000016	Pointer	Read-only	Optional	Pointer to the memory descriptor areas.
0x00000017	Pointer	Read-only	Optional	Pointer to the SDM information string.
0x00000018 to 0x7FFFFFFF	N/a	N/a	N/a	Reserved by ARM.
0x80000000 to 0xFFFFFFFF	N/a	N/a	N/a	Available for third-party extensions. Registered through ARM.

7.7.1 Tag 0x00000000, End-of-table marker

The last tag-value pair in the table is always the end-of-table marker. This tag is different from all others in that it does not have an associated value. The behavior of the target is undefined if the host attempts to read the value. The host controller must not try to access any memory beyond the end-of-table marker.

7.7.2 Tag 0x00000001, RealMonitor version number

The RealMonitor revision number is mandatory, and must be the first tag-value pair in the capabilities table. The debugger determines the endianness of the target by reading the first tag in the table using four byte reads. If the lowest addressed byte contains the value 0x00, the target is big-endian. If it contains the value 0x01, it is little-endian.

The revision number takes the format 0xMMVVVVRR, where:

M Manufacturer ID.

`V` Version number.

`R` Revision number.

The format of the version word is shown in Figure 7-34.

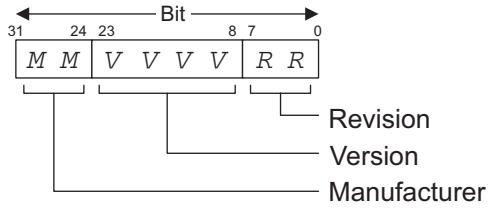

Figure 7-34 Format of the version word

The manufacturer ID is a single ASCII character. For ARM, the character `A` (`0x41`) is used. Other companies creating their own implementation of RealMonitor must use a default value of `0x00`, or register a different character with ARM. The version number starts at one, and is incremented whenever an incompatible change is made to the protocol.

The revision number starts at zero, and is incremented whenever a backwards-compatible change is made to the protocol. For ARM, the initial implementation of the RealMonitor uses the version `0x41000200`.

Note

Both the RealMonitor 1.0 host controller and target support version 2.0 of the RealMonitor protocol. Version 1.0 of the RealMonitor protocol was used solely for the Beta A release of RealMonitor, and is now unsupported.

7.7.3 Tag 0x00000002, Configuration word

The configuration word is used by the host to find out which features are available in a particular build of the RealMonitor. Each bit set in the configuration word indicates a single feature that RealMonitor supports. Table 7-4 shows the meaning of each bit.

Table 7-4 Meaning of bits

Bit	Description
0	Stop and Go opcodes are supported
1	ReadBytes opcode is supported, and must be set
2	WriteBytes opcode is supported
3	ReadHalfWords opcode is supported
4	WriteHalfWords opcode is supported
5	ReadWords opcode is supported
6	WriteWords opcode is supported
7	GetPC opcode is supported
8	SyncCaches opcode is supported
9	ExecuteCode opcode is supported
10	ReadRegisters opcode is supported
11	WriteRegisters opcode is supported
12	ReadCPRegister is supported
13	WriteCPRegister is supported
14	Software breakpoints are supported
15	Hardware breakpoints are supported
16	Watchpoints are supported
17	SWI semihosting is supported

Table 7-4 Meaning of bits (continued)

Bit	Description
18	Data logging is supported
19	Writes to the RealMonitor state byte are supported
20	InitializeTarget is supported

7.7.4 Tag 0x00000003, Pointer to the build identifier string

This optional tag-value pair allows you to distinguish between different builds of RealMonitor. The value is a pointer to a read-only, zero-terminated ASCII string. The contents of the string are implementation-dependent. In the ARM implementation of RealMonitor, the version string contains the time and date that RealMonitor was built.

7.7.5 Tag 0x00000010, Pointer to the RealMonitor state byte

The value field for this tag is a pointer to a byte in RAM that shows the current state of RealMonitor. If bit 19 in the configuration word is set, the host can write to this byte to start and stop the foreground application. If the bit is clear, the host must assume this byte is read-only, and use the Go and Stop packets to start and stop the foreground application. The possible values for the state byte are as follows:

`0x00` Running. The foreground application is running.

`0x01` Stopped. The foreground application is stopped.

`0x02` Panic. RealMonitor is in a state from which it is unable to recover.

7.7.6 Tag 0x00000011, Pointer to the channel filter block

Note

Reserved for future use by ARM.

7.7.7 Tag 0x00000012, Pointer to the register block

In the cases where RealMonitor returns a pointer to a register block, the block is formatted as follows:

```
typedef struct {
    uint32 r[13];                   /* User mode r0 to r12 */
    uint32 usr_r[2]                 /* User mode r13 and r14 */
```



```
    uint32 pc;                      /* the program counter */
    uint32 cpsr;                    /* the CPSR of the foreground task */
    uint32 svc_r[2], svc_spsr;      /* SVC mode r13, r14, SPSR */
    uint32 abort_r[2], abort_spsr;  /* Abort mode r13, r14, SPSR */
    uint32 undef_r[2], undef_spsr;  /* Undef mode r13, r14, SPSR */
    uint32 irq_r[2], irq_spsr;      /* IRQ mode r13, r14, SPSR */
    uint32 fiq_r[7], fiq_spsr;      /* FIQ mode r8 to r14, SPSR */
} RTM_RegisterBlock;
```

The target modifies the value of `lr` before storing it into the registers block. The modification depends on the reason for stopping, as shown in Table 7-5.

Note

`lr` refers to the `lr` delivered to the exception handler that is then stored into the `pc` entry in the registers block, and is not used to indicate the value stored into register `usr_r[1]`.

Table 7-5 Link register adjustment

Vector	Modification	After Modification lr Points to
Interrupt	`SUB lr, lr, #4`	Next instruction to be executed
Undefined instruction	None	Instruction that caused the Undef
Prefetch Abort	`SUB lr, lr, #4`	Instruction that caused the abort
Data Abort	`SUB lr, lr, #8`	Instruction that caused the abort
SWI	None	Instruction after the SWI

The registers block must be read and written while the target is stopped. After the target receives a Go command, an implementation-defined subset of the processor registers is restored from the register block.

7.7.8 Tag 0x00000013, Pointer to the error block

The error block is a sequence of one or more words that contains the details of the last error that occurred. The structure of the error block is shown in Figure 7-35 on page 745.

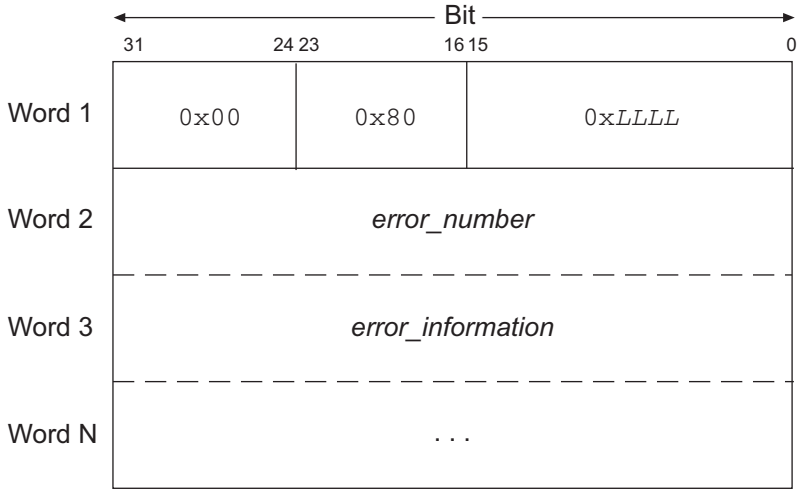

Figure 7-35 Error block format

The error block is formatted in a similar way to an Ok packet, with the length field giving the amount of additional data provided. The second word is the error number, and subsequent words provide any associated information, such as the fault address or status information from an instruction or Data Abort. Currently defined error codes are as follows:

`Error_None`

ErrorBlock[0] = `0x00800004`
ErrorBlock[1] = `0x00000000`

`Error_UnsupportedOpcode`

ErrorBlock[0] = `0x00800004`
ErrorBlock[1] = `0x00000001`

`Error_DataAbort`

ErrorBlock[0] = `0x00800008`
ErrorBlock[1] = `0x00000002`
ErrorBlock[2] = address that caused the Data Abort.

7.7.9 Tag 0x00000014, Pointer to the register accessibility block

You can maintain up to 64 registers in a register block. See *Tag 0x00000012, Pointer to the register block* on page 743 for the current register block definition. The accessibility block uses a bitmap to define valid entries for read access, and valid entries

for write access. The most significant bit set (read or write access) in the accessibility block defines the size of the register block implemented on the target. The host must not access a register beyond the capability indicated by the accessibility block.

Caution

If read access to the pc is disabled, you cannot use breakpoints, watchpoints, or SWI semihosting. If write access to the pc is disabled, you cannot use breakpoints.

The value field for this tag is a pointer to a four-word structure:

Word 0 Bits 0 to 31 correspond to entries 0 to 31 of the register block (read access).

Word 1 Bits 0 to 31 correspond to entries 0 to 31 of the register block (write access).

Word 2 Bits 0 to 31 correspond to entries 32 to 63 of the register block (read access).

Word 3 Bits 0 to 31 correspond to entries 32 to 63 of the register block (write access).

7.7.10 Tag 0x00000015, Pointer to the execution information block

The value field for this tag provides a pointer to a read-only array of two words:

Word 0 Start address of the buffer (word-aligned).

Word 1 Length, in bytes, of the buffer (multiple of four).

7.7.11 Tag 0x00000016, Pointer to memory descriptor block

The value field for this tag provides a pointer to a table containing descriptions of the memory map of the target (see Figure 7-36 on page 747). The table consists of zero or more memory descriptions, followed by a word where the lower 16-bits are zero. Each memory description consists of a 16-bit parameter and 16-bit length field, followed by one or more address ranges.

Each address range consists of a 32-bit inclusive start address, followed by a 32-bit inclusive end address. Address ranges can overlap with each other, in which case, the address ranges that occur nearer to the start of the table have a higher priority than those nearer the end.

The length field at the start of each memory descriptor specifies the number of bytes until the start of the next memory descriptor block. The length is equal to (*number of address ranges* * 8), so for example, if a memory descriptor contain three address ranges, the length field is (3 * 8) = 24.

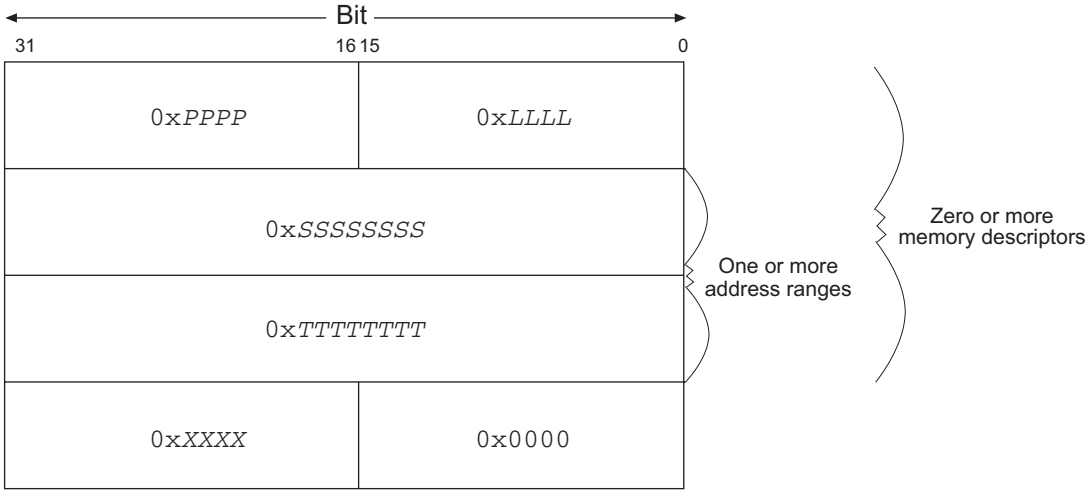

Figure 7-36 Format of the memory descriptor block

The parameter field is a set of bit fields defined as shown in Table 7-6.

Table 7-6 Memory descriptor parameter field

Bits	Description	Value	Definition
16	Read access	0b0	Not readable
		0b1	Readable
17	Write access	0b0	Not writable
		0b1	Writable
19:18	Endianness (Ignored by the controller)	0b00	Little-endian
		0b01	Big-endian
		0b10	Same as processor (default)
		0b11	Reserved
20	Stoppable	0b0	Must not set breakpoints
		0b1	Can set breakpoints

Table 7-6 Memory descriptor parameter field (continued)

Bits	Description	Value	Definition
22:21	Default access size (ignored by the controller)	0b00	Do not care (default)
		0b01	Byte
		0b10	Half-word
		0b11	Word
23	Cache synchronize (ignored by the controller)	0b0	Must synchronize after writes (default)
		0b1	No synchronization required on writes
31:24	Reserved		Must be zero

If the memory descriptor block is not implemented, or if a memory address is not covered by any address ranges, the following defaults apply:

- memory can be read and written
- debugger can set/clear breakpoints
- memory has the same endianness as the processor
- no default access size
- does require cache synchronization after writes.

7.7.12 Tag 0x00000017, Pointer to the SDM information string

The value field for this tag is a pointer to a zero-terminated ASCII string that contains three substrings delimited by a linefeed character. These substrings contain information about the processor name, processor revision number, and board type. A file containing a complete list of supported devices is available to RMHost, and the contents of the string in the target must correspond to this. An example of the string format is as follows:

```
"ARM966E-S\n0\nCM966E-S + Integrator/AP"
```


7.8 Data logging

RealMonitor allows your application or RTOS to pass its own data back to the debug host. This data is queued on the target in a fixed-sized *First-In First-Out* (FIFO) buffer until it can be sent.

All third-party data must be formatted into packets in a similar manner to the way RealMonitor packets are formatted. That is, they must contain a valid packet header. You can use any channel number in the packet header, except for channel 0 because this is used by the RealMonitor protocol.

Glossary

This glossary contains definitions for some of the acronyms contained within this Programmer's Guide.

ADW *See* ARM Debugger for Windows.

AFS *See* ARM Firmware Suite.

API *See* Application Program Interface.

Application Program Interface
The syntax of the functions and procedures within a module or library.

ARM Debugger for Windows
This debugger and *ARM Debugger for UNIX* (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the ARM Software Development Toolkit. It is fully supported and is now supplied as part of the ARM Developer Suite.

ARM eXtended Debugger
The latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.

ARM Firmware Suite A collection of utilities to assist in developing applications and operating systems on ARM-based systems.

ARM/Thumb Procedure Call Standard
Defines how registers and the stack are used for subroutine calls.

ATPCS *ARM/Thumb Procedure Call Standard*.

AXD *See* ARM eXtended Debugger.

DCC *See* Debug Communications Channel.

Debug Communications Channel
A 32-bit, bidirectional communications link between the target processor and the host.

Debug Status and Control Register
Register for DCC status and control.

DSCR *See* Debug Status and Control Register.

EBI *See* External Bus Interface.

External Bus Interface
A bus that allows additional memory to be added to an Integrator board.

FIFO *First-In First-Out*.

FPGA *Field-Programmable Gate Array*.

JTAG *Joint Test Application Group*.

MOE *Method Of Entry*.

RDI *See* Remote Debug Interface.

Remote Debug Interface
An open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged. RDI gives the debugger a uniform way to communicate with the following:

- a debug agent running on the host (such as ARMulator)
- a debug monitor running on ARM-based hardware accessed through a communication line (such as Angel)
- a debug agent controlling an ARM processor through hardware debug support (such as Multi-ICE).

RTOS *Real-Time Operating System*.

SDM *Self-Describing Module*.

TLB *See* Translation Lookaside Buffer.

Translation Lookaside Buffer

A memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Typically, there is a TLB for each memory interface of the ARM implementation.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Symbols

