Application Note 39

Demon and RDP

Document number: ARM DAI 0039B

Issued: July 1998

ENGLAND
ARM Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY
ARM Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN
ARM K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA
ARM INC.
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.

ARM	*XXX*	*0000*	*X*	*- 00*

(On review drafts only) Two-digit draft number

Release code in the range A-Z

Unique four-digit number

Document type

Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	March 1998	SWS	Initial version
B	July 1998	paw	Text edits, formatting

Table of Contents

1. Introduction

The *Remote Debug Protocol (RDP)* is used for communication between a host debugger and remote debug target hardware in the ARM *Software Development Toolkit (SDT)* 2.02 and earlier. RDP is used by the EmbeddedICE agent software (version 1.0x), and also by the Demon debug monitor running on ARM boards.

The *Angel Debug Protocol (ADP)* was introduced in the ARM SDT v2.1x. This new protocol is used for communication between the debugger and the Angel debug monitor on EmbeddedICE (v2.0x and later). For details of ADP, Angel, and EmbeddedICE, refer to the SDT v2.1x (or later) documentation.

In versions of the SDT prior to 2.1x, RDP was the only communication protocol supported. Although support for RDP is provided in SDT2.1x, this is only for backwards compatibility, and it is not the default protocol.

This application note details both Demon and RDP, as well as how you can make use of the backwards compatibility built into SDT v2.1x.

Note *Do not use RDP/Demon unless your project was begun using an earlier version of SDT, and it is not possible to change to ADP/Angel. All new projects should make use of ADP/Angel.*

Note *Configuring SDT 2.1x to use RDP instead of the default ADP will not be supported in any future versions of the ARM Toolkit. This means that SDT 2.11a is the last Toolkit version that can provide communication to a remote target using the EmbeddedICE interface v1.xx or the Demon debug monitor.*

2. Updating ADP from RDP

When using SDT v2.1x it is recommended that you use the new ADP version of EmbeddedICE (v2.0x) or the Angel debug monitor running on your target. However if you are already developing a product using the RDP version of EmbeddedICE (v1.0x), or if you have the Demon debug monitor running on your target, you have to do one of the following in order to use SDT v2.1x or later:

- Upgrade your EmbeddedICE from v1.0x to v2.0x (see section ***2.1 Upgrading EmbeddedICE***).
- Port the Angel debug monitor to your target to replace the Demon debug monitor (for details on how to do this refer to *Angel Porting Guide* - Application Note 54).
- Configure SDT v2.1x for RDP (see section ***3 Configuring SDT v2.1x for RDP***).

2.1. Upgrading EmbeddedICE

To verify which EmbeddedICE version you are using, press the reset:

- If you have version 1.x (RDP), the LED light goes out briefly, and then comes on and stays on.
- If you have version 2.x (ADP), the LED lights up and remains lit.

To upgrade EmbeddedICE from v1.x to v2.x:

- Install the Angel ROM images from the SDT v2.1x release CD. You can find these images at the following location:

  ```
  \<install_directory>\Angel\images\iceagent\iceagent.rom
  ```

- Request an upgrade from your supplier. At the time of writing, the ARM address is:

  ```
  http://www.arm.com/DevSupp/Sales+Support/download
  ```

3. Configuring SDT v2.1x for RDP

If it is not possible to upgrade to ADP at the current time, you must perform the following to continue working with either your existing EmbeddedICE interface software or with the target resident Demon debug monitor. To use SDT 2.1x you must:

- Set up the build environment to include the Demon libraries.
- Configure a project to use the Demon C libraries (*ARM Project Manager 2 (APM)* only).
- Configure the debugger to use Remote_D (RDP).

3.1. Setting the build environment

You can set up the build environment to include the Demon libraries:

- during the initial installation
- after you have already installed SDT without having to reinstall

3.1.1. During initial installation

When you install the SDT, the Angel version of the libraries is normally used when building applications. To ensure that the Demon libraries are used, simply chose Demon as the default runtime library from the installation options screen..

3.1.2. Without reinstalling

If you have already installed SDT for use with ADP and Angel, you can change it for RDP and Demon without having to re-install:

APM	Change the Library path to `\lib\demon\`

Command Line	Modify the `armlib` environment variable from:
	`\<install_directory>\lib`
	to
	`<install_directory>\lib\demon`

Note *If you fail to link with the Demon C libraries, the debugger is likely to stop with the error message:*

```
Program stopped: SWI
```

pointing to an Angel SWI 0x123456 instruction.

3.2. Configuring a project to use Demon C libraries

If you have APM you can configure armlink to use Demon or Angel C libraries on a per project basis:

1. Select the root directory of the project in the project view.
2. From the **Project** menu select **Tool Configuration for project.apj**, select **armlink,** and then **Set**. The Linker Configuration dialog is displayed.
3. Select the **General** options tab.

4 Ensure the **Search standard libraries** option is selected.

5 Change **Search path for libraries** to:

```
\<install_directory>\lib\demon
```

6 Click the on **OK** button.

When the project is rebuilt, it will be linked with the Demon C libraries.

3.3. Configure the debugger

The method for configuring a debugger for RDP depends on which debugger you are using:

- ARMulator with armsd or *ARM Debugger for Windows (ADW)*
- ADW with a remote target
- armsd with a remote target

Using ARMulator

The ARMulator provided with SDT v2.1x can handle both Angel and Demon semihosted SWIs by default. Therefore, an application linked with a Demon semihosted C library should run under ARMulator without any need to reconfigure the debugger.

Note *You can turn this backwards compatibility off by modifying the* `armul.cnf` *file (refer to ARMulator Configuration File* - Application Note 52).

Using ADW with a remote target

To configure ADW to use RDP:

1 Start ADW directly (not via APM).

2 Select **Configure Debugger** from the **Options** menu. The Debugger Configuration dialog is displayed.

3 Click on the **Target** tab.

4 Click on the **Add** button.

5 Select:

```
\<install_directory>\Bin\remote_d.dll
```

6 Ensure **remote_d** is selected as the Target Environment.

7 Click on the **Configure** button.

8 Select the **Remote Connection** that you are using (either **Serial/Parallel** or **Serial**).

9 Select the **Ports** and the **Serial Line Speed**.

10 Click on the **OK** button.

11 Click on the **Debugger** tab.

12 Select the **Endian** mode the target board runs in (little-endian is the default).

13 Click on the **OK** button.

The Debugger is restarted. The Restarting box is displayed, with rapidly changing numbers indicating that the Debugger is reading from and writing to the target board.

If the target board does not respond, the numbers in the Restarting box do not change. The debugger times out within approximately five seconds, the message 'ADP Error - target did not respond' is displayed, and the configuration defaults to back to the ARMulator.

When the configuration has completed successfully, a message similar to the following is displayed in your host console window (if you are using EmbeddedICE):

```
EmbeddedICE v1.06 512K ROM CRC Ok. Little Endian.
```

Or, if you are using the Demon debug monitor:

```
ARM60, DEMON V1.1, 0x080000 bytes RAM, ROM CRC OK, Little endian,
FPE.
```

When the Debugger exits, it retains the remote debugger option settings so you can automatically make use of them again next time you use the Debugger.

For further information on using EmbeddedICE see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040).

3.3.1. Using armsd with a remote target

To use armsd with a remote target using RDP, start armsd using the `–rdp` option. For full details, see the debugger section of the *SDT Reference Guide* (ARM DUI 0041).

4. Demon Debug Monitor

The Demon debug monitor was supplied on ARM's PIE60 and PIE7 target boards. It was also supplied in source form with SDT 1.x and 2.0x. The source is not supplied with SDT2.1x as Demon's replacement (the Angel debug monitor) is supplied instead.

The Demon debug monitor source code can be split into three parts:

- Driver code
  This is a very simple device driver for the debug channel. The debug channel only needs to support byte read and write operations.
- Level 0 code
  This code:
  1. Initializes the processor.
  2. Performs memory checks.
  3. Initializes all exception mode stacks.
  4. Installs optional floating-point support.
  5. Initializes memory management.
  6. Installs the debug port driver.
  7. Returns an ASCII string representing the state of the machine.
- Level 1 code
  This code contains:
  - An RDP interpreter that allows the machine to be debugged.
  - A software interrupt handler (SWI) that provides C library support over the debug channel.

The following sections describe these areas of code in more detail.

4.1. Driver code

The `driver.s` file contains the debug channel device driver support code, and is automatically included by the level 0 code (`level0.s`)

4.2. Level 0 code

The `level0.s` file (and corresponding `level0_h.s` header file) contains the level 0 code. This file automatically includes debug channel device driver support code (`driver.s`).

The level code works in the following way:

1. Ensures the processor is in 32-bit mode (ARM6 and ARM7 series processors with MMUs begin execution in 26-bit mode).
2. Branches to the beginning of ROM (using LDR PC).

   This supports systems that are reset with ROM overlaying the hardware vectors at location 0, but whose real ROM is elsewhere in the memory map.
   Note: *Only register 14 is used here; all other registers are preserved to support a manual postmortem after reset (that is, the user can examine the registers to find out what was happening before the reset).*

3 Does a double write to the bottom of memory.

These writes remove ROM from location 0 so that RAM is now accessible.

This is the recommended method of supporting boot ROM for the reset vectors, while allowing the reset vector to contain RAM at a later time.

4 All registers are dumped in the `SavedRegs` location defined in the header file (`level0_h.s`).

Level 1 code can, at a later time, examine or even change the register values in `SavedRegs`. This allows the state of the task being debugged to be altered.

5 A dummy, undefined instruction handler is installed.

6 The ID register is read into register 0 by an MRC being performed from coprocessor 15. If the processor does not have an MMU then the dummy handler sets register 0.

7 `ResetDriver` is called. The stack pointers of all exception modes and the debug channel are initialized.

8 `SetLED` is called. Progress is indicated on the status LEDs.

9 `FindRAMSize` is called. The RAM is checked and sized.

The RAM size returned by `FindRAMSize` is used as the start address of the user mode stack (that is, this code assumes that RAM starts at zero and extends upwards—the stack grows down from the top).

10 The status LEDs are updated to show progress.

11 A fast CRC16 algorithm, found in `CheckSubroutine`, is run. The ROM is checksummed.

The endianness of the ROM does not matter as the ROM content is loaded one word at a time.

12 The status LEDs are updated to show progress.

13 A copy of the instruction in the reset vector is saved in `ResetVectorCopy`, and the vector replaced.

Demon uses an STR instruction with an addressing mode like [r14, -r14] to store a register in location zero, which preserves the state without destroying anything except the reset vector.

14 The reset vector is restored from `ResetVectorCopy`.

15 The exception vectors are initialized with a PC-relative LDR PC.

This allows an exception handler to be anywhere in the 4GB address space.

16 Initial handlers are installed, and the status LEDs flash.

Any unhandled exceptions can be identified at this point.

17 The IRQ handler and the FIQ handler are both initialized directly to point to `SerialInt`.

18 The status LEDs are updated to show progress.

19 The debug port drive is initialized again.

This ensures any power up errors are cleared down.

20 127 0x7f (DEL) characters are sent down the line.

This terminates any partial RDP commands that were in progress before the reset was pressed.

21 The banner is sent to the debugger.

This includes initializing the floating-point system, any MMU present, and calling `Level1Init` to set up the next layer.

22 The status LEDs are updated to show progress.

23 The interrupts are enabled.

24 The code now loops until level 1 code is called by RDP.

At this stage the memory map is as follows:

Address	Description
0x00000	CPU Reset Vector
0x00004	CPU Undefined Instruction Vector
0x00008	CPU Software Interrupt Vector
0x0000c	CPU Prefetch Abort Vector
0x00010	CPU Data Abort Vector
0x00014	CPU Address Exception Vector
0x00018	CPU Interrupt Vector
0x0001c	CPU Fast Interrupt Vector
0x00020	1KB for FIQ routine and FIQ mode stack
0x00400	256 bytes for IRQ mode stack
0x00500	256 bytes for Undefined mode stack
0x00600	256 bytes for Abort mode stack
0x00700	256 bytes for SVC mode stack
0x00800	Debug monitor private workspace
0x01000	Free — for user supplied debug monitor
0x02000	Floating point emulator space
0x08000	Application space
Top of memory	

Table 4-1: Initial memory map

At the start of ROM, a jump table is provided for the level 1 code to interface through, as follows:

Offset	Description
0x00000	Reset instruction
0x00004	Address of `Reset` routine
0x00008	Address of `InstallRDP` routine
0x0000c	Address of `ResetChannel` routine
0x00010	Address of `ChannelSpeed` routine
0x00014	Address of `GetByte` routine
0x00018	Address of `PutByte` routine
0x0001c	Address of `ReadTimer` routine
0x00020	Address of `SetLED` routine

Table 4-2: Jump table

The initial entry code is provided by mapping the first two words into address 0 and 4 when the CPU is reset.

`InstallRDP` Used by the level 1 code to register a handler of all RDP messages.

The address of the handler is passed in register 0, and the address of the previous handler returned in register 0. If level 1 code does not handle all RDP messages, then unhandled messages return to the previous handler.

The RDP handler is entered in FIQ mode, with the interrupts disabled and the RDP message number in register 0.

To exit from the handler, load the program counter from the stack. This can be done using an instruction such as:

```
LDMFD sp!, {pc}
```

As all register values are saved before entry to the handler, they are all restored after exit.

When the level 1 RDP handler has been entered, it can read successive bytes from the debug channel using `GetByte`, and send replies using `PutByte`. RDP messages can also be formulated to support the application code that it has been called by—these are also sent by `PutByte`.

`ResetChannel` Used to reset the debug channel driver, if an error is detected by the level 1 code.

`ChannelSpeed`	Used to change the speed of the debug channel in an implementation designed fashion. When powered on, `ChannelSpeed` is set (by value 0) to the default speed of 9600bps. By sending values 1, 2 or 3 to `ChannelSpeed` the speed can be set to 9600bps, 19200bps or 38400bps respectively. The meaning of any other value is undefined.
`GetByte`	Used to read bytes from the debug channel.
`PutByte`	Used to write bytes to the debug channel.
`ReadTimer`	Returns a 10ms count from an on-board timer to register 0. If an on-board timer is not available, 0xFFFFFFFF (-1) is returned.
`SetLED`	Allows the on-board status LED to be set and cleared. The action required is dictated by register 0. The LED is turned off by a 0 or on by any other value. If multiple status LEDs are supported by a board, the argument should be treated as a binary value with each bit controlling a different LED.

4.3. Level 1 code

The level 1 code provides two functions:

- RDP interpreter, which implements the RDP functions (described in ***5 Remote Debug Protocol*** on page 17)
- SWI handler, which provides SWI functions

The SWI functions provided fully support the Demon variant of the ARM ANSI C Semihosted library and are described below; these SWI functions are also implemented by EmbeddedICE agent software version 1.x:

`SWI_WriteC (SWI 0)`
Writes a byte, passed in register 0, to the debug channel. When executed under an ARM debugger, the character appears on the display device connected to an ARM debugger.

`SWI_Write0 (SWI 2)`
Writes the null-terminated string, pointed to by register 0, to the debug channel. When executed under an ARM debugger, the characters appear on the display device connected to an ARM debugger.

`SWI_ReadC (SWI 4)`
Reads a byte from the debug channel and returns it in register 0. The read is notionally from the keyboard attached to the ARM debugger.

`SWI_CLI (SWI 5)`
Passes the string pointed to by register 0 to the host's command line interpreter.

`SWI_GetEnv (SWI 0x10)`
Returns the address of the command-line string used to invoke the program in register 0 and returns the highest available address in user memory in register 1.

`SWI_Exit (SWI 0x11)`
Halts emulation. This is the way a program exits cleanly, returning control to the debugger.

`SWI_EnterOS (SWI 0x16)`
Puts the processor into supervisor mode. If the processor is currently in 26-bit mode, SVC26 is entered, otherwise SVC32 is entered.

`SWI_GetErrno (SWI 0x60)`
Returns the value of the C library `errno` variable associated with the host support for this debug monitor in register 0. `errno` can be set by a number of C library support SWIs, (for example, `SWI_Remove`, `SWI_Rename`, `SWI_Open`, `SWI_Close`, `SWI_Read`, `SWI_Write`, `SWI_Seek`). Whether or not, and to what value `errno` is set is completely host-specific, except where the ANSI C standard defines the behavior.

`SWI_Clock (SWI 0x61)`
Returns the number of centi-seconds since the support code began execution in register 0. In general, only the difference between successive calls to `SWI_Clock`, can be meaningful.

`SWI_Time (SWI 0x63)`
Returns the number of seconds since January 1, 1970 (the UNIX time origin) in register 0.

`SWI_Remove (SWI 0x64)`
Deletes and unlinks the file named by the NULL-terminated string addressed by register 0. This returns a zero if the removal succeeds, or a non-zero, host-specific error code if it fails, in register 0.

`SWI_Rename (SWI 0x65)`
Register 0 and register 1 address NULL-terminated strings, the old-name and new-name, respectively. If the rename succeeds, zero is returned in register 0; otherwise, a non-zero, host-specific error code is returned.

`SWI_Open (SWI 0x66)`
Register 0 addresses a NUL-terminated string containing a file or device name; register 1 is a small integer specifying the file-opening mode, as shown in ***Table 4-3: File opening modes*** on page 13. If the open succeeds, a non-zero handle is returned in register 0, which can be quoted to `SWI_Close`, `SWI_Read`, `SWI_Write`, `SWI_Seek`, `SWI_Flen` and `SWI_IsTTY`. Nothing else can be asserted about the value of the handle. If the open fails, the value 0 is returned in register 0.

Register 1 value	ANSI C `fopen()` mode
0	"r"
1	"rb"
2	"r+"
3	r+b"
4	"w"
5	"wb"
6	"w+"
7	"w+b"
8	"a"
9	“ab"
10	"a+"
11	"a+b"

Table 4-3: File opening modes

`SWI_Close (SWI 0x68)`

On entry, register 0 must be a handle for an open file, previously returned by `SWI_Open`. If the close succeeds, zero is returned in register 0; otherwise, a non-zero value is returned.

`SWI_Write (SWI 0x69)`

On entry, register 0 must contain a handle for a previously opened file; register 1 points to a buffer in the callee; and register 2 contains the number of bytes to be written from the buffer to the file. `SWI_Write` returns the number of bytes not written in register 0 (and so indicates success with a zero return value).

`SWI_Read (SWI 0x6a)`

On entry, register 0 must contain a handle for a previously opened file or device; register 1 points to a buffer in the callee; and register 2 contains the number of bytes to be read from the file into the buffer. `SWI_Read` returns the number of bytes not read in register 0, and so indicates the success of a read from a file with a zero return value. If the handle is for an interactive device (`SWI_IsTTY` returns non-zero for this handle), a non-zero value is returned from `SWI_Read`, indicating that the line read did not fill the buffer.

`SWI_Seek (SWI 0x6b)`

On entry, register 0 must contain a handle for a seekable file object, and register 1 must contain the absolute byte position to move to. If the request can be honored, `SWI_Seek` returns a 0 in register 0, otherwise it returns a host-specific, non-zero value.
Note: The effect of seeking outside of the current extent of the file object is undefined.

`SWI_Flen (SWI 0x6c)`

On entry, register 0 contains a handle for a previously opened, seekable file object. `SWI_Flen` returns the current length of the file object in register 0, otherwise it returns -1.

`SWI_IsTTY (SWI 0x6e)`

On entry, register 0 must contain a handle for a previously opened file or device object. On exit, register 0 contains 1 if the handle identifies an interactive device, otherwise register 0 contains 0.

`SWI_TmpNam (SWI 0x6f)`

On entry, register 0 points to a buffer and register 1 contains the length of the buffer (register 1 should be at least the value of `L_tmpnam` on the host system). On successful return, register 0 points to the buffer that contains a host temporary file name. If the request cannot be satisfied (for example, because the buffer is too small) then 0 is returned in register 0.

`SWI_InstallHandler (SWI 0x70)`

`SWI_InstallHandler` installs a handler for a hardware exception. On entry, register 0 contains the exception number (see ***Table 4-4: Hardware exception handling*** on page 15), register 1 contains a value to pass to the handler when it is eventually invoked and register 2 contains the address of the handler. On return, register 2 contains the address of the previous handler and register 1 contains its argument.

When the exception occurs, the handler is entered in the appropriate non-user mode, with register 10 holding a value dependent on the exception type, and register 11 holding the handler argument (as passed to `InstallHandler` in register 1). Registers 10, 11, 12, and 14 (for the processor mode in which the handler is entered) are saved on the stack of the mode in which the handler is entered. All other registers are as at the time of the exception. (Any effects of the instruction causing the exception have been unwound, and the saved register 14 points at the instruction that failed, rather than one or two words ahead.) If the handler returns, the exception is passed to the debugger (regardless of the value of the debugger variable `$vector_catch`).

No.	Exception	Mode	r10 value
0	branch through zero	svc32	-
1	undefined instruction	undef32	-
2	SWI	svc32	SWI number
3	prefetch abort	abort32	-
4	data abort	abort32	-
5	address exception	svc32	-
6	IRQ	irq32	-
7	FIQ	fiq32	-
8	Error	svc32	error pointer

Table 4-4: Hardware exception handling

`SWI_GenerateError (SWI 0x71)`

On entry, register 0 points to an error block (containing a 32-bit error number, followed by a zero-terminated error string). Register 1 points to a 17-word block containing the values of the ARM CPU registers at the instant the error occurred (the 17th word contains the PSR). `SWI_GenerateError` calls the (software) error vector—if bit 8 of `$vector_catch` is set, the debugger is entered directly, otherwise, the installed error handler is called (see `SWI_InstallHandler`).

4.4. RDP support

The level 1 RDP support code is called from the interrupt handler for the debug channel. When a byte arrives on the channel, it is sent to the RDP handler in the level 1 code. The level 1 code performs all subsequent transfers (that is, not under interrupt control). The level 1 code installs handlers for exceptions. Unexpected exceptions are handled and passed back via the RDP protocol.

4.5. Breakpoint support

Under Demon, breakpoints are supported in two ways depending on where the breakpoint is set:

- If the breakpoint is set in the lower 32MB of memory, branch instructions are used to replace the instruction that is breakpointed. These instructions make a call back to Demon.

  As branch instructions have a 32MB branch range, this type of breakpoint is only used on the lower 32MB of memory.

- If the breakpoint is set in any part of the memory except the lower 32MB of memory, an Undefined Instruction is used to generate an exception, and the Undefined Instruction handler passes control back to Demon.

  This type of breakpoint cannot be used to breakpoint code running in Undefined mode.

- As breakpoints require memory to be changed, Demon cannot set breakpoints on code in ROM.

4.6. Watchpoint support

Demon does not support true watchpoints, instead it provides *changepoints*. After each instruction has executed, Demon checks the “watched” locations to see if their values have changed. If a change is detected, the application stops and Demon returns control to the debugger.

Note *This means that Demon has to single step the application code; setting watchpoints will dramatically slow down execution speed.*

5. Remote Debug Protocol

5.1. Introduction

The *Remote Debug Protocol (RDP)* is the byte stream communication protocol used between the ARM debuggers and a remote debuggee, using a debug monitor or controlling debug agent. Usually, the RDP is used via stub functions implementing the *Remote Debug Interface (RDI)*—there is a one-to-one correspondence between RDI calls and RDP messages. For details see the documentation supplied with the ARM Software Development Toolkit.

The RDI gives the ARM debugger core a uniform way to communicate with:

- a controlling debug agent or debug monitor linked with the debugger
- a debug agent executing in a separate operating system process
- a debug monitor running on ARM-based hardware accessed over a communication link
- a debug agent controlling an ARM processor using hardware debug support

Structure 1 (RDI)
: This arises in the variant of the debuggers that are linked with ARM's standard ARM emulation environment (for the PC- and Sun-hosted cross-development variants of the debugger), and in the self-hosted, single address-space variant of armsd (for Acorn's RISC OS operating system). No direct use of the RDP is involved.

Structure 2 (RDI over IPC/RPC)
: This would arise in an ARM-UNIX-hosted variant of the symbolic debugger, if the debugger and the ARM emulator (the ARMulator) were run in separate UNIX processes (perhaps on separate machines). In the second case, the RDI would consist of two stubs using UNIX's remote procedure calls to effect the inter-process message passing. Again, no direct use of the RDP is involved.

Structures 3 and 4 (RDP)
: These arise when the debugger is used to control a debuggee executing on ARM-based hardware (for instance on the *Platform Independent Evaluation (PIE)* card) connected to the debugger's host using a hardware debugging channel, (for instance, over RS232 as used on the PIE card).

5.2. Terminology

The *program counter (PC)* is the address of the currently-executing instruction in the debuggee.

An ARM processor can be configured to operate with either:

- *little-endian* memory (in which the *least significant byte (lsb)* of a word has the lowest address of any byte in the word)
- *big-endian* memory (in which the *most significant byte (msb)* of a word has the lowest address of any byte in the word)

The endianness of a memory system and processor configuration is also called its byte sex.

In the following sections, pseudo-C declarations are used to specify the content of messages and the types of arguments to message functions. In these declarations:

- *byte* means an 8-bit unsigned value
- *word* means a 4-byte unsigned value, transmitted lsb first (little-endian)

The types *bytes* and *words* (plural) mean, respectively, a sequence of bytes and a sequence of words.

Values enclosed in braces { } are present only in certain contexts, as clarified by the explanatory text.

Each message of the RDP is encoded as a single function byte, followed immediately by its arguments, if any.

The return message acts as an acknowledgement as well as returning values. If the request is meaningful and successful, a zero-status byte is returned, possibly preceded by requested data.

The reply to an unsatisfied request (failed request) is always padded to the same length that it would have had, if it had been successful. The reply is folled by a non-zero error code byte (see ***5.7 Error Codes*** on page 40).

As RDP has evolved, new levels of specification have been added, and within any level of specification there are implementation options. This approach was taken so that a variety of minimal debug monitors and controlling debug agents can be accommodated without excessive overhead, and so there can be compatibility between debuggers and debug monitors released at different times. As a result, a debugger using the RDP must negotiate to establish its debuggee's capabilities and must not use capabilities that its debuggee does not have. This is done using the info message. These issues are highlighted in the following sections.

5.3. Message summary

Message Name	Hexadecimal Function Code
Open and/or Initialize	00
Close and Finalize	01
Read Memory Address	02
Write Memory Address	03
Read CPU State	04
Write CPU State	05
Read Coprocessor State	06
Write Coprocessor State	07
Set Breakpoint	0A
Clear Breakpoint	0B
Set Watchpoint	0C
Clear Watchpoint	0D
Execute	10
Step	11
Info	12
OS Operation Reply	13
Add Configuration	14
Load Configuration	15
Select Configuration	16
Load Debug Agent	17
Interrupt Request	18
Comms Channel To Host (Reply)	19
Comms Channel From Host (Reply)	1A
Reset	7F

Table 5-1: Debugger to debuggee messages

Message name	Hexadecimal Function Code
Stopped Notification message	20
OS Operation Request	21
Comms Channel to Host	22
Comms Channel From Host	23
Fatal Protocol Error	5E
Return Value/Status Message	5F
Reset	7F

Table 5-2: Debuggee to debugger messages

5.4. Debugger to debuggee messages

Open and/or initialize message (00)

```
Open(byte type, word memorysize {, byte speed})
return(byte status)
```

Upon receipt of this message a debuggee prepares itself for an imminent debugging session, bootstrapping and/or initializing itself. This message is always sent as the first message. If for some reason initialization is impossible, a non-zero status value must be returned. The `type` argument can be used to distinguish between types of initialization:

Bit 0 = 0	cold start (bootstrap, initialize MMU, and so on)
Bit 0 = 1	warm start (terminate current execution, clear all breakpoints/ watchpoints, and so on)
Bit 1 = 1	reset the communication link
Bit 2 = 0	debugger requires little-endian debuggee
Bit 2 = 1	debugger requires big-endian debuggee
Bit 3 = 1	debuggee must return its sex

The *`memorysize`* argument is used to specify the minimum number of bytes of memory that the debuggee's environment must have. A value of zero can be used if the debugger is not concerned with the memory size, for example when the debuggee is running under an ARM emulator, which allocates memory dynamically, as needed.

If bit 1 of the `type` argument is set, a single byte specifying the debug channel speed follows the *`memorysize`* argument. A value of zero sets the default speed. Other values are target dependent.

The return value `RDIError_WrongByteSex` indicates that the debuggee has the opposite byte order to the byte order requested in bit 2 of the argument, and therefore the request has failed. If bit 3 of the *`type`* argument is set, the debuggee ignores bit 2 and returns a status of either `RDIError_LittleEndian` or `RDIError_BigEndian`.

Close and finalize message (01)

```
Close()
return(byte status)
```

This message indicates the termination of the current debugging session. If for some reason the current debugging session cannot be terminated, a non-zero status value is returned. Only the `Open` message can follow the `Close` message.

Read memory address message (02)

```
Read(word address, word nBytes)
return(bytes data, byte status {, word nBytes})
```

This message requests the transfer of memory contents for the debuggee to the debugger. The transfer begins at *address*, and transfers *nBytes* of data in increasing address order.

On successful completion, the requested bytes are returned, followed by a zero status value.

On unsuccessful completion, the number of bytes requested are returned (some are garbage padding), followed by a non-zero error code byte, followed by the number of bytes successfully transferred. This number can be added to the base address to calculate the address where the transfer failed.

Write memory address message (03)

```
Write (word address, word nBytes, Bytes data)
return(Byte status {, word nBytes})
```

This message transfers data from the debugger to the debuggee’s memory. The *address* argument specifies the location where the first byte of data is to be stored, and the *nBytes* argument gives the number of bytes to be transferred, followed by the byte sequence to transfer.

A zero status value is returned on successful completion.

On failure, a non-zero error code byte is returned, followed by the number of bytes successfully transferred, just as with the `Read` message.

Read CPU state message (04)

```
ReadCPU(Byte mode, word mask)
return(words data, byte status)
```

This message is a request to read the values of registers in the CPU.

The *mode* argument defines the processor mode from which the transfer must be made. The mode number is the name as the mode number used by ARM6—a value of 255 indicates the current mode.

The *mask* argument indicates which registers must be transferred. Setting a bit to 1 causes the designated register to be transferred.

Bit [0:14]	request register r0–r14
Bit 15	requests the PC (including the mode and flag bits in 26-bit modes)

Bit 16	requests the transfer of the value of the PC (without the mode and flag bits in a 26-bit mode)
Bit 17	requests the address of the currently executing instruction (often 8 bytes less than the PC, because of instruction prefetching)
Bit 18	(in 32-bit modes) requests transfer of the CPSR—in 32-bit processor modes with an SPSR (non-user modes), bit 19 requests its transfer
Bit 20	requests the transfer of the value of the flag and mode bits in a 26-bit mode (in the same bit positions as in register 15)

Upon successful completion, the values of the register are returned (the number depending on the number of bits set in the *mask* argument), followed by a zero status value. The lowest numbered register is transferred first.

On unsuccessful completion, the number of words specified in the *mask* is returned, followed by a non-zero error code byte.

Write CPU state message (05)

```
WriteCPU(byte mode, word mask, words data)
return (byte status)
```

This message is a request to set the values of registers in the debuggee's CPU.

The *mode* argument defines the processor mode to which the transfer must be made.

The *mask* argument is the same as for the `ReadCPU` message, and is followed by the sequence of word values to be written to the register specified in *mask*. The first value is written to the lowest numbered register given in *mask*.

The status value returned is zero if the request is successful, otherwise, the error is specified (see ***5.7 Error Codes*** on page 39).

Read coprocessor state message (06)

```
ReadCoPro(byte Cpnum, word mask)
return(words data, byte status)
```

This message is a request to read a coprocessor's internal state. Its operation is similar to `ReadCPU`, except that register values are transferred from the coprocessor numbered *Cpnum*.

The registers to be transferred are specified by the *mask* argument.

The registers are transferred and their sizes are coprocessor-specific. Currently the following coprocessor are understood:

- Coprocessor 1 (and 2 in the case of FPA) is a floating-point unit.
  - Bits [0:7] of *mask* request the transfer of floating-point registers [0:7].
  - Bit 8 requests the FPSR.
  - Bit 9 requests the FPCR.
- Coprocessor 15 is an MMU, for example ARM600s.
  - Bits [0:7] of *mask* request the transfer of internal registers [0:7].

On successful completion, the values of the requested registers are returned, followed by a zero status value. The lowest numbered register is transferred first.

On unsuccessful completion, the number of words implied by *mask* are transferred, followed by a non-zero error code byte.

For more details on the structure of a coprocessor description, including the format of coprocessor registers, see `dbg_cp.h`.

Write coprocessor state message (07)

```
WriteCoPro(byte Cpnum, word mask, words data)
return(byte status)
```

This message is a request to write a coprocessor's internal state. This operation is similar to that of `WriteCPU`, except that register values are transferred to the coprocessor numbered *Cpnum*. The registers to be written are specified by the *mask* argument.

The registers are transferred, and their sizes, depend on the coprocessor. Currently, the following coprocessors are understood:

- Coprocessor 1 (and 2 in the case of FPA) is a floating-point unit.
  - Bits [0:7] of *mask* request the setting of floating-point registers [0:7].
  - Bit 8 requests a write to the FPSR.
  - Bit 9 requests a write to the FPCR.
- Coprocessor 15 is an MMU, for example ARM600s.
  - Bit [0:7] of *mask* requests the setting of internal registers [0:7].

The status value returned is zero when the request is successful, otherwise the error is specified (see ***5.7 Error Codes*** on page 39).

For more details on the structure of a coprocessor description, including the format of coprocessor registers, see `dbg_cp.h`.

Set breakpoint message (0x0A)

```
SetBreak(word address, byte type {, word bound})
return({word pointhandle} byte status)
or
return({word address{word bound}} byte status)
```

This message requests the debuggee to set an execution breakpoint at *address*. The least significant 4 bits of *type* define the sort of breakpoint to set:

0	halt if the PC is equal to *address*
1	halt if the PC is greater than *address*
2	halt if the PC is greater than or equal to *address*
3	halt if the PC is less than *address*
4	halt if the PC is less than or equal to *address*
5	halt if the PC is in the address range from *address* to `bound`, inclusive
6	halt execution if the PC is not in the address range from *address* to `bound`, inclusive
7	halt execution if PC AND *bound* = *address*

At RDI/RDP specification levels 1 and above, bits [4:7] of *type* have further significance:

Bit 4 set	If set, the breakpoint is on a 16-bit (Thumb) instruction, rather than a 32-bit (ARM) instruction.
Bit 5 set	Requests that the breakpoint is conditional (execution halts only if the breakpointed instruction is executed, not if it is conditionally skipped). If bit 5 is not set, execution halts whenever the breakpointed instruction is reached (whether executed or skipped).
Bit 6 set	Requests a dry run; the breakpoint is not set and the *address*—and if appropriate the *bound*—that would be used are returned (for comparison and range breakpoints, the *address* and *bound* used need not be exactly as requested). A zero status byte indicates that resources are currently available to set the breakpoint; `RDIError_NoMorePoints` indicates that the required breakpoint resources are not currently available.
Bit 7 set	Requests that a handle, *pointhandle*, is returned for the breakpoint by which it is subsequently identified. If bit 7 is set, a handle is returned whether the request succeeds or fails (but, obviously, it is only meaningful if the request succeeds).

Note *Bits 6 and 7 must not be set simultaneously.*

Upon completion a zero status byte is returned. On unsuccessful completion, a non-zero error code byte is returned.

If the request is successful, but there are no more breakpoint registers (of the requested *type*), the message `RDIError_NoMorePoints` is returned.

On unsuccessful completion, a non-zero error code byte is returned.

Clear breakpoint message (0x0B)

```
ClearBreak(word pointhandle)
return(byte status)
```

This message requests the clearing of the execution breakpoint (identified by *pointhandle*) that was set by an earlier *SetBreak* request. At level 0 of the RDI/RDP specification, *pointhandle* is the address at which the breakpoint was set.

On successful completion, a zero status byte is returned.

On unsuccessful completion, a non-zero error code byte is returned.

Set watchpoint message (0x0C)

```
SetWatch(word address, byte type, byte datatype {, word bound})
return({word pointhandle} byte status)
or
return({word address {,word bound}} byte status)
```

This message requests the debuggee to set a data access watchpoint at *address*. The least significant 4 bits of *type* define the sort of watchpoint to set:

0	data access to the address equal to *address*
1	halt on a data access to an address greater than *address*
2	halt on a data access to an address greater than or equal to *address*
3	halt on a data access to an address less than *address*
4	halt on a data access to an address less than or equal to *address*
5	halt on a data access to an address in the range from *address* to *bound*, inclusive
6	halt on a data access to an address not in the range from *address* to *bound*, inclusive
7	halt if `(`*data–access–address* `&` *bound*`) =` *address*

At RDI/RDP specification levels 1 and above, bits 6 and 7 of *type* have further significance:

Bit 6 of *type* set	Requests a dry run; the watchpoint is not set and the *address*—and if appropriate, the *bound*—that would be used are returned (for range and comparison watchpoints, the *address* and *bound* used need not be exactly as requested). A zero status byte indicates that resources are currently available to set the watchpoint; `RDIError_NoMorePoints` indicates that the required watchpoint resources are not currently available.
Bit 7 of *type* set	Requests that a handle is returned for the watchpoint by which it is subsequently identified. If bit 7 is set, a handle is returned, whether the request succeeds or fails (but obviously, it is only meaningful if the request succeeds).

Note *Bits 6 and 7 must not be set simultaneously.*

The *datatype* argument defines the type of data access to watch for:

1	watch for byte reads
2	watch for halfword reads
4	watch for word reads
8	watch for byte writes
16	watch for halfword writes
32	watch for word writes

Values can be summed or ORed together in order to halt on any of a set of sorts of memory access. For example, to watch for any write access to the specified location(s):

8 + 16 + 18

Upon successful completion, a zero status byte is returned. On unsuccessful completion, a non-zero error code byte is returned. If the request is successful, but there are no more watchpoint registers (of the requested *type*), the value `RDIError_NoMorePoints` is returned.

If a watchpoint is set on a location that already has a watchpoint, the first watchpoint is removed before the new watchpoint is set.

Clear watchpoint message (0x0D)

```
ClearWatch(word pointhandle)
return(byte status)
```

This message requests the clearing of the data access watchpoint (identified by *pointhandle*) which was set by an earlier *SetWatch* request. At level 0 of the RDI/RDP specification, *pointhandle* is the address at which the watchpoint was set.

Upon successful completion, a zero status byte is returned. On unsuccessful completion, a non-zero error code byte is returned.

Execute message (0x10)

```
Execute(byte return)
return({word pointhandle} byte status)
```

This message requests that the debuggee commences execution at the address currently loaded into the CPU PC.

If the lsb of *return* is 1, and commencing execution is viable, a return message is sent immediately and execution commences asynchronously.

If the lsb of *return* is 0, execution commences synchronously, and the return message is not sent until execution is suspended because:

- the end of the program is reached
- a breakpoint or watchpoint is reached
- an exception occurs
- the user interrupts the program

At RDI/RDP specification levels 1 and above, bit 7 of *return* has further significance—if it is set, an extra word is returned. This extra word is the identifying handle of the breakpoint or watchpoint that suspended execution, if execution is suspended because of a breakpoint or watchpoint.

On successful completion of the request, a zero status byte is returned. On completion of a synchronous request, a non-zero *status* byte may indicate the reason the debuggee suspended. Examples of possible return codes are:

2	Undefined instruction
3	A SWI happened (only if watching for SWIs, see ***Info*** message (0x12) on page 28)
4	Prefetch Abort—instruction fetch from unmapped memory
5	Data Abort—no memory at the accessed address
6	Address Exception—26-bit mode access to address >= 2**26
7	IRQ
8	FIQ
9	Error
10	Branch through location 0
143	Breakpoint Reached It is the responsibility of the RDP's caller to remove the breakpoint before continuing execution, otherwise the debuggee stops immediately at the same breakpoint.
144	Watchpoint Accessed It is not defined whether the PC addresses the accessing instruction or a subsequent instruction, nor whether the access has been performed.
147	User Pressed Escape

Step message (0x11)

```
Step(byte return, word ninstr)
return({word pointhandle} byte status)
```

This message requests the debuggee to execute *ninstr* instructions, starting at the address currently loaded into the CPU PC.

If *ninstr* is 0, the debuggee executes instructions up to the next instruction that explicitly alters the PC (that is, a branch or ALU operation with the PC as the destination).

If the lsb of *return* is 1, and starting execution is viable, a return message is sent immediately and execution commences asynchronously.

If the lsb of *return* is 0, execution starts synchronously, and the return message is not sent until execution suspends, because:

- the requested number of instructions have been executed
- a breakpoint or watchpoint is reached
- an exception occurs

- the user interrupts the program.

At RDI/RDP specification levels 1 and above, bit 7 of *`return`* has further significance—if it is set an extra word is returned. This extra word is the identifying handle of the breakpoint or watchpoint that suspended execution, if execution is suspended because of a breakpoint or watchpoint.

On successful completion of the request, a zero status byte is returned. On completion of a synchronous request, a non-zero status byte indicates the reason that the debuggee is suspended, exactly as for the `Execute` message (see ***Execute message*** (0x10) on page 26)

Info message (0x12)

```
Info(word info {, argument})
return({words returninfo,} byte status)
```

This message requests the transfer of information between the debugger and the debuggee. The information transferred and the direction of transfer depends on the value of *`info`*. In each case, a non-zero status byte indicates an unsuccessful request (see ***5.7 Error Codes*** on page 39 for details).

The definition of *`info`* is as follows:

`info` = 0 Returns information about the debuggee in the same way as:

```
return(word data, word model, byte status)
```

Bit 16	1 => the debuggee has a communications channel
Bit 15	1 => the debuggee can cope with 16-bit (Thumb) code
Bit 14	1 => the debug agent can do profiling
Bit 13	1 => the debug agent supports `RDP_Interrupt`
Bit 12	1 => the debug agent supports inquires about the download block size that it supports
Bit 11	1 => the debug agent can be reloaded
Bits [8:10]	the minimum RDI specification level (0–7) required of the debugger
Bits [5:7]	the maximum RDI specification level (0–7) implemented by the debuggee
Bit 4	0=> debuggee is running under a software emulator 1=> debuggee is running on ARM hardware
Bits [0:3]	host speed as 10∗∗(bits [0:3]) *instructions per second (IPS)* (0=>1IPS, 1 => 10IPS, 2 => 100IPS, 3 = 1000IPS, …, 6 => 1MIPS, …)

The value of *`model`* is a unique identifier for the ARM processor or the emulator model under which the debuggee is running.

`info` = 1 — Returns information about the debuggee's breakpointing and watchpointing capabilities, in the same way as:

`return(word breakinfo, byte status)`

The value of *breakinfo* must be interpreted as a set of bits, as follows:

Bit 0	comparison breakpoints are supported
Bit 1	range breakpoints are supported
Bit 2	watchpoints for byte reads are supported
Bit 3	watchpoints for halfword reads are supported
Bit 4	watchpoints for word reads are supported
Bit 5	watchpoints for byte writes are supported
Bit 6	watchpoints for halfword writes are supported
Bit 7	watchpoints for word writes are supported
Bit 8	mask breakpoints/watchpoints are supported
Bit 9	thread-specific breakpoints are supported
Bit 10	thread-specific watchpoints are supported
Bit 11	conditional breakpoints are supported
Bit 12	status inquires about the capabilities of (hardware) breakpoints and watchpoints are allowed

All debuggees must support breakpoints of `RDIPoint_EQ` (break at specified address).

`info` = 2 — Returns information about the debuggee's single-stepping capabilities, in the same way as:

`return(word stepinfo, byte status)`

The value of *stepinfo* is interpreted as follows:

Bit 0	single stepping of more than one instruction is supported
Bit 1	single stepping to the next direct PC alteration is supported
Bit 2	single stepping of a single instruction is supported

`info` = 3 — Returns information about the debuggee's memory management system (if any), in the same way as:

`return(word meminfo, byte status)`

info = 4 Inquires whether configuration download and selection are supported:

```
return(byte status)
```

A *status* return of 0 indicates that these facilities are supported.

info = 5 Inquires whether Info Calls 0x181 to 0x184 (semihosting Get/Set State/Vector are supported):

```
return(byte status)
```

A *status* return of 0 indicates that these facilities are supported.

info = 6 Inquires whether Info Calls 0x400 and 0x401 (Describe Coprocessor and Request Coprocessor Description) are supported:

```
return(byte status)
```

info = 7 Inquires whether the debuggee is controlled by EmbeddedICE:

```
return(byte status)
```

A status return of 0 indicates that the debuggee is controlled by EmbeddedICE.

info = 8 Asks for the memory access statistics for the block of memory indicated by handle. For full details, see `RDI_MemAccessStats` in `dbg_stat.h`.

```
arguments(word handle)
return (word nreads, word nwrites, word sreads,
           word swrites, word ns, word s,
           byte status)
```

info = 9 Sets the characteristics for *n* regions of memory:

```
arguments(word n, {word handle, word start,
                word limit, byte width, byte
                access, word Nread_ns,
                word Nwrite_ns, word Sread_ns,
                word Swrite_ns} …)
return(byte status)
```

For further details, see `dbg_stat.h`.

info = 10 Sets the simulated CPU speed in nanoseconds:

```
arguments(word speed)
return(byte status)
```

info = 12 Reads the simulated CPU time:

```
return(word ns, word s, byte status)
```

info = 13 Inquires whether the debug agent supports Info calls 08–12 (Memory Statistics).

```
return(byte status)
```

A *status* return of 0 indicates that these facilities are supported.

`info` = 14 Inquires about the number of configuration blocks known to the debug agent:

```
return(byte status, word count)
```

count is present only if *status* indicates an error.

`info` = 0x80 Returns the hardware resource number and *type* of that resource, when given a watchpoint *handle*. This can be used only if Info Call 1 (`Info_Points`) returns bit 12 (`RDIPointCapability_Status`) set.

```
arguments(word handle)
return(word hwresource, word type, byte status)
```

`info` = 0x81 Similar to Info Call 0x80, except that it works for a breakpoint handle.

`info` = 0x100 Requests that the debuggee immediately halts execution. If the debuggee is not executing or is running synchronously (see ***Execute message*** (0x10) on page 26), a return message and the status `RDIError_UserInterrupt` is returned. If the debuggee is running asynchronously, a `Stopped` message is returned, with status `RDIError_UserInterrupt`.

`info` = 0x180 Tells the debuggee which hardware exceptions must be reported to the debugger. *argument* is a bit-mask of exceptions to be reported:

Bit 0	Reset (branch through 0)
Bit 1	Undefined Instruction
Bit 2	Software Interrupt (SWI)
Bit 3	Prefetch Abort
Bit 4	Data Abort
Bit 5	Address Exception
Bit 6	Interrupt (IRQ)
Bit 7	Fast Interrupt (FIQ)
Bit 8	Error

A set bit in *argument* indicates that the exception must be reported to the debugger; a clear bit indicates that the corresponding exception vector must be taken. When an exception is reported to the debugger, the state of the debuggee is rewound to the state in effect just before executing the instruction that caused the exception.

`info` = 0x181 Set whether or not semihosting is enabled. It can be used only if Info Call 5 (`Info_SemiHosting`) returned 0.

```
arguments (word semihostingstate)
result (byte status)
```

info = 0x182 Reads whether or not semihosting is enabled. It can be used only if Info Call 5 (`Info_SemiHosting`) returned 0.

```
result (word semihostingstate, byte status)
```

info = 0x183 Set the semihosting vector. It may be used only if Info Call 5 (`Info_SemiHosting`) returned 0.

```
arguments (word semihostingvector)
result (byte status)
```

info = 0x184 Reads the semihosting vector. It may be used only if Info Call 5 (`Info_SemiHosting`) returned 0.

```
result(word semihostingvector, byte status)
```

info = 0x185 Reads a bitmap that indicates which of the EmbeddedICE macrocell breakpoints have been locked. This can be used only if Info Call 7 (`Info_ICEBreaker`) returned 0.

```
result(word lockedstate, byte status)
```

info = 0x186 Writes a bitmap that indicates which of the EmbeddedICE macrocell breakpoints are locked. This may be used only if Info Call 7 (`Info_ICEBreaker`) returned 0.

```
arguments(word lockedstate)
```

info = 0x187 Requests the maximum length of data the debug agent can receive in one block. This can be used only if Info Call 0 (`Info_Target`) returned bit 12 set.

```
result (word maxloadsize, byte status)
```

info = 0x188 Indicates whether data must be transferred from the debuggee to the debugger over the Debug Comms Channel. This can be used only if Info Call 0 (`Info_Target`) returned bit 16 set.

```
arguments(byte connect)
return (byte status)
```

info = 0x189 The same as Info Call 0x188 except that it refers to data transfer from the debugger to the debuggee.

info = 0x200 Requests the debuggee to return the number of instructions and cycles executed since initialization, in the same way as:

```
return(word ninstr, word S-cycles, word N-cycles,
           word I-cycles, word C-cycles, word F-
cycles,
           byte status)
```

info = 0x201 Requests the debuggee to return the error pointer associated with the last return to an `Execute` or `Step` request with *status* `Error`.
Note: The error is returned as a word rather than a byte.

info = 0x300 Requests that the debuggee's command line be set to *argument*. *argument* must be a zero-terminated string of bytes, and no longer than 256 bytes (including the terminating 0).

info = 0x301 Requests that the RDI specification level be set to *argument*, a byte value lying between the limits returned by a call with Info Call = 0. From receipt of an open request with bit 0 of `type` = 0 (a cold start open request) until receipt of this request, the debuggee operates with the RDI level set to its lower limit.

info = 0x302 Requests that the thread context (`SetBreak` and `SetWatch` messages) be set to *argument*, a 32-bit handle identifying a thread of execution. The distinguished handle `RDINoHandle` requests resetting the thread context to be the underlying hardware processor.

info = 0x400 Describes the registers of a coprocessor. *argument* has the form:

```
byte cpnum {byte rmin, byte rmax, byte nBytes,
               byte access, byte r0, byte r1,
               byte w0, byte w1}* byte = 0xff
```

where:

nBytes is the size in bytes of the register(s)

access is a bitmask:

Bit 0 register(s) readable with this bit set

Bit 1 register(s) writeable with this bit set

Bit 2 register(s) read or written using CPDT instructions (else CPRT) with this bit set. If bit 2 is set, the registers provide bits as follows:

r0 Bits [0:7]

r1 Bits [16:23] of a CPRT instruction to read the register

w0 Bits [0:7]

w1 Bits [16:23] of a CPRT instruction to write the register

Otherwise, r0 provides Bits [12:15] and r1 bit 22 of CPDT instructions to read and write the register (and w0 and w1 are junk).

info = 0x401 Has *argument* as a byte coprocessor number. It requests that the debuggee describe the registers of the coprocessor if it is known. The description is:

```
return({byte rmin, byte rmax, byte nBytes,
           byte access}*, byte = 0xff)
```

where *rmin*, *rmax*, *nBytes* and *access* are as above.

info = 0x500 Requests that the collection of profiling data is stopped. This must only be used if Info Call 0 (`Info_Target`) returned bit 14 set.

```
return (byte status)
```

info = 0x501 Requests that profiling data must be collected. This must be used only if Info Call 0 (`Info_Target`) returned bit 14 set.

```
arguments(word interval)
```

PC samples are taken every *interval* microseconds.

info = 0x502 Must be used only if Info Call 0 (`Info_Target`) returned bit 14 set.

```
arguments(word len, word size, word offset,
    words mapdata
result(byte status)
```

This downloads a map array that describes the PC ranges for profiling. See the documentation supplied with the ARM Software Development Toolkit for more details.

This is downloaded over the RDP in a series of messages:

len The number of elements in the entire map array being downloaded.

size The number of map array elements being downloaded in this message.

offset The offset into the entire map array from which this message starts.

mapdata Consists of *size* words of *mapdata*.

info = 0x503 Must be used only if Info Call 0 (`Info_Target`) returned bit 14 set.

```
arguments(word offset, word size)
result(words counts, byte status)
```

This uploads a set of profile counts that correspond to the current profile map. See the documentation supplied with the ARM Software Development Toolkit for more details.

This is uploaded over the RDP in a series of messages:

offset The offset in the entire array of counts from which this message starts.

size The number of *counts* being uploaded in this message.

counts Consists of *size* words of profiling counts data.

info = 0x504 Requests that profiling counts are reset to zero. This must be used only if Info Call 0 (`Into_Target`) returned bit 14 set.

```
result(byte status)
```

OS operation reply message (0x13)

```
OSOpReply(byte info, {data})
```
no reply

This message signals completion of the last requested OS Operation request.

info describes the type of the value returned by the operation:

0	return value
1	`data` comprises a single byte, to be placed in the debuggee's r0
2	`data` comprises a word, to be placed in the debuggee's r0

OS operations that return more complicated values must do so by using the appropriate combination of Write Memory and Write CPU state operations

Add configuration message (0x14)

```
AddConfig(word nBytes)
return(byte status)
```

On receiving this message, the debug agent stores the configuration data ready for it to be selected. If there is an error during download, a non-zero status is returned.

Load configuration message (0x15)

```
LoadConfigData(word nBytes, words data)
return(status)
```

On receiving this message, the debug agent stores the configuration data ready for it to be selected. If there is an error during download, a non-zero status is returned. For more information about the content of the configuration data, see the documentation for the debug agent concerned.

Select configuration message (0x16)

```
SelectConfig(byte aspect, byte namelen, byte matchtype,
                  word vsn_req, bytes name)
return(word vsn_sel, byte status)
```

On receiving this message, the debug agent selects one of the sets of configuration data blocks and re-initialize the debug agent to use that configuration:

aspect	either of `RDI_ConfigCPU` or `RDI_ConfigSystem`
namelen	the number of bytes in *name*
name	*namelen* bytes that comprise the name of the configuration
vsn_req	the requested version of the named configuration
matchtype	specifies how the selected version must match the specified version, and must be one of `RDI_MatchAny`, `RDIMatchExactly`, or `RDI_MatchNoEarlier`

The debug agent returns the version number of the configuration selected and a status byte.

Load debug agent (0x17)

```
LoadAgent(word loadaddress, word size)
return(byte status)
```

On receiving this message, the debug agent prepares to receive configuration data that it interprets as a new version of the debug agent code. The new debug agent code is *size* bytes long, and must be loaded at *loadaddress*. The data is downloaded using `LoadConfigurationData (0x15)`.

Interrupt execution (0x18)

```
Interrupt()
```
no return value

On receiving this message, the debug agent attempts to halt execution of the debuggee. No return value is sent.

CCTHostReply (0x19)

```
CCToHostReply()
return(byte status)
```

On receiving this message, the debug agent knows if the host successfully received the data it sent to `CCTHost (0x22)`.

CCFromHostReply (0x20)

```
CCFromHostReply(byte valid, word data)
return(byte status)
```

This message returns data to be sent to the debuggee using the Debug Comms channel. If *valid* is 0, there is no data, otherwise *data* is the word to be transferred.

Reset message (0x7F)

```
RequestReset()
```
no return value

Requests that the debuggee reset itself. No return value is sent

5.5. Debuggee to debugger messages

The debugger does not acknowledge debuggee-to-debugger messages, as there is no point in trying to handle errors in the debuggee.

All responses to debugger-to-debuggee requests are of the `Return` message type, described in ***Return message*** (0x5F) on page 38.

Stopped message (0x20)

`Stopped({word` *`pointhandle`*`} byte` *`status`*`)`

To indicate that execution of the debuggee has suspended, this message is sent to a debugger by an asynchronously executing debuggee. Execution of the debuggee was previously started by an `Execute(X)` or `Step(X, n)` message with bit 1 of `X` = 1.

The status value indicates the type of suspension; see ***Execute message*** (0x10) on page 26.

At RDI specification levels 1 and above, if bit 7 of `X` was 1, a word *`pointhandle`* is returned. If execution is suspended because a breakpoint was reached or a watchpoint was accessed, the value of *`pointhandle`* identifies the breakpoint or watchpoint concerned.

OS operation request message (0x21)

`OSOp(word` *`op`*`, byte` *`argdesc`*`, {args})`

This message is sent by the debuggee to request execution of a call to the host operation system.

`op` Identifies which call.

`argdesc` Allows the description of up to four arguments to the call; if there are more, or their format is more complicated than can be described by *`argdesc`*, their values must be acquired by the appropriate combination of the Read Memory and Read CPU State operations. *`argdesc`* is a sequence of two-bit fields starting from the least significant end of the word, each of which describes the corresponding argument, as shown below:

0	no such argument
1	a single byte
2	a word
3	a string

The format of a string value is determined by its first byte:

[0:32]	the string is of this length, and its component bytes follow (the terminating 0 byte is omitted).
[33:254]	the string is of this length. A word containing the address of the string follows. The read memory message can be used to access the string.

255 a word follows containing the string's length and a word containing its address. The read memory message can be used to access the string.

Comms channel to host message (0x22)

```
CCToHost(word data)
return(byte status)
```

This message sends a word of data that has been transferred from the debuggee using the Debug Comms Channel up to the host.

Comms channel from host message (0x23)

```
CCFromHost()
return(byte valid, word data, byte status)
```

This message requests a word of data from the host to be sent to the debuggee using the Debug Comms Channel. If `valid` is 0, the host has no data to transfer. Otherwise, `data` is valid.

Fatal message (0x5E)

```
Fatal(byte error)
```

The `Fatal` message indicates that the debuggee could not make sense of the last message sent.

Return message (0x5F)

```
Return(…, byte status)
```

The `Return` message is used to acknowledge a recognized request message.

A status value is always returned, indicating that the syntax of the original request was understood, and whether or not it was satisfied.

The arguments to `Return` depend on the message being acknowledged, and are described in ***5.4 Debugger to debuggee messages*** on page 20.

Reset message (0x7F)

This message indicates that the debuggee has reset itself, either because of a hardware reset, or in response to a reset request.

When a reset occurs, the debuggee sends a stream of reset messages (0x7F) to ensure that the debugger realizes that there has been a reset (irrespective of the state the protocol was in before the reset occurred). Following the stream of 0x7F, the debuggee sends an ASCII text banner message followed by a 0x00 character. The banner message gives identification of the target system (processor, memory and so on) and is normally displayed by the debugger.

5.6. Notes

5.6.1. Address spaces

In debuggee environments that support different address spaces in different processor modes, the address space that corresponds to the processor mode at the time the message is sent is used by all memory access, breakpoint and watchpoint instructions.

5.6.2. Minimum support

There is a minimum subset of requests that all debuggees must support. The `info` message is used to inquire whether a debuggee can support operations outside the minimum subset, consisting of:

Message	Function code
Open and/or Initialize	00
Close and Finalize	01
Read Memory Address	02
Write Memory Address	03
Read CPU State	04
Write CPU State	05
Set Breakpoint	0A (with first argument = 0)
Clear Breakpoint	0B
Execute	10
Info	20
Reset	7F

5.7. Error Codes

5.7.1. Debuggee status

Code	Error Name	Possible Cause
0	`No error`	Everything worked.
1	`Reset`	Debuggee reset.
2	`Undefined instruction`	Tried to execute the undefined instruction.
3	`Software interrupt`	A SWI happened (when tracing SWIs).
4	`Prefetch abort`	Execution ran into unmapped memory.
5	`Data abort`	No memory at the specified address.
6	`Address exception`	Accessed > 26-bit address in 26-bit mode.
7	`IRQ`	An interrupt occurred.
8	`FIQ`	A fast interrupt occurred.
9	`Error`	An error occurred.
10	`BranchThrough0`	Branch through location 0.
142	`No more points`	The last of the breakpoints/watchpoints has been reached.
143	`Breakpoint reached`	Breakpoint reached—returned by Execute or Step.
144	`Watchpoint accessed`	Watchpoint reached—returned by Execute or Step.
146	`Program finished`	End of the program reached while stepping.
147	`User interrupt`	User pressed Escape.
148	`CantSetPoint`	Breakpoint/watchpoint resources exhausted.
150	`CantLoadConfig`	Configuration Data could not be loaded.
151	`BadConfigData`	Configuration data was corrupt.
152	`NoSuchConfig`	The request configuration has not been loaded.
153	`BufferFull`	Buffer was filled during the operation.
154	`OutOfStore`	The debug agent ran out of memory.
155	`NotInDownLoad`	Illegal request made during download.
156	`PointInUse`	EmbeddedICE breakpoint is already being used.
157	`BadImageFormat`	Debug agent could not make sense of AIF image supplied.
158	`TargetRunning`	Target processor could not be halted (probably with EmbeddedICE system).
159	`DeviceWouldNotOpen`	Failed to open serial or parallel port.
160	`NoSuchHandle`	No such memory description handle exists.
161	`ConflictingPoint`	Incompatible breakpoint already exists.

Table 5-3: Debuggee status

5.7.2. Internal Fault or limitation messages

Code	Error Name	Possible Cause
253	`InsufficientPrivilege`	Supervisor state was not accessible to this debug monitor.
254	`Unimplemented message`	Debuggee cannot honor this RDP request.
255	`Undefined message`	Garbled RDP request.

Table 5-4: Internal faults

5.7.3. Information messages

Code	Error Name	Possible Cause
240	`LittleEndian`	The debuggee is little-endian.
241	`BigEndian`	The debuggee is big-endian.
242	`SoftInitialiseError`	A recoverable error occurred during initialization. Perhaps different configuration data is required.

Table 5-5: Informational messages

5.7.4. RDI error messages

Code	Error Name	Possible Cause
128	`Not initialised`	Open must be the first call.
129	`Unable to initialise`	The target world is broken.
130	`WrongByteSex`	The debuggee cannot operate with the requested byte sex.
131	`Unable to terminate`	Target world was smashed by the debuggee.
132	`Bad instruction`	It is illegal to execute this instruction.
133	`Illegal instruction`	The effect of executing is undefined.
134	`Bad CPU state`	Tried to set the SPSR of user mode.
135	`Unknown co-processor`	This coprocessor is not connected.
136	`Unknown co-proc state`	Do not know how to handle this request.
137	`Bad co-proc state`	Recognizably broken coprocessor request.
138	`Bad point type`	Misuse of the RDI.
139	`Unimplemented type`	Misuse of the RDI
140	`Bad point size`	Misuse of the RDI
141	`Unimplemented size`	Halfwords not yet implemented
145	`No such point`	Tried to clear an unset breakpoint or watchpoint

Table 5-6: RDI error messages