



Debug of ARM™ based systems using EmbeddedICE™

Version 1.0

ARM, ARM7, ARM7TDMI, ARM9, EmbeddedICE, Thumb, Embedded Trace Macrocell, AXD, ADW, RealMonitor, RMHost, RMTARGET, EmbeddedICE-RT are registered trademarks of ARM Limited. All other trademarks referred to herein are the property of their respective owners.

Ashling Microsystems Ltd.
Tony Garvey, 32-bit Product Manager,
Hugh O'Keefe, R&D Director,
National Technological Park,
Limerick, Ireland
Tel: +353-61-334466
Fax: +353-61-334477
mailto: hugh.okeeffe@ashling.com
<http://www.ashling.com>

doc: ARMDebugv10.doc

1. Table of contents

1.	Table of contents	2
2.	Revision History	2
3.	References:	2
4.	Introduction	2
4.1	ARM EmbeddedICE	3
4.2	Debug features enabled by EmbeddedICE	5
4.3	Debug Communications Channel (DCC)	6
4.4	Semihosting	6
4.4.1	Standard Semihosting	7
4.4.2	DCC-based Semihosting	7
4.5	EmbeddedICE-RT and RealMonitor	8

2. Revision History

Version	Date	Comments
1.0	3 rd July 2002	First External Release

3. References:

1. *ARM7TDMI Technical Reference Manual, Document No. ARM DDI0210B*, September 2001, ARM Ltd.
2. *The ARM7TDMI Debug Architecture, Document No. ARM DAI 0028A*, December 1995, ARM Ltd.
3. *ARM RM Target Integration Guide, Document No. ARM DUI 0142A*, December 2000, ARM Ltd.

4. Introduction

The currently popular high-performance 32-bit microprocessor architectures feature on-chip memory, pipelines, cache memory and high speeds of operation; all of these factors conspire to render the task of producing traditional in-circuit emulators for embedded systems development hugely impractical, if not completely impossible.

Consequently, all major silicon and core Intellectual Property (IP) vendors now include on-chip silicon resources or core extensions to alleviate the difficulties facing developers of software for embedded systems based on such architectures. Such resources allow debug access to the core, and much of the functionality of traditional in-circuit emulators is thus enabled using a new breed of emulator which can access and control these on-chip debug resources.

In very many cases, the device's JTAG pins, normally used for boundary scan test purposes, are 'hijacked' to allow access to the on-chip debug resource. Typical core IP examples include the products from ARM Ltd and MIPS Inc, which include EmbeddedICE and EJTAG on-chip debug facilities respectively, both accessed over JTAG. Motorola, for many years to the forefront in on-chip debug technology, offers BDM, OnCE, COP, Nexus and other debug schemes based on on-chip aids to emulation. Many other vendors, including Hitachi, ST Microelectronics, SuperH and Infineon, have developed similar on-chip debug facilities, not to mention the wide variety of semiconductor vendors whose devices include on-chip debug through licence arrangements with IP vendors such as the aforementioned ARM and MIPS.

Owing to the enormous popularity of the ARM cores, with 77 licensees shipping over 420 million ARM powered devices in 2001, the ARM on-chip debug facilities merit closer examination. This paper looks at the run-time control debug features provided ARM cores. ARM have also developed a core extension called the Embedded Trace Macrocell (ETM) which provides for the provision of compressed program and data trace information via a narrow port. ETM is currently available as a core extension for ARM7 and ARM9 products and will be covered in more detail in a future Ashling paper.

4.1 ARM EmbeddedICE

The current most popular ARM products are based on ARM7 and ARM9 core designs. These are 32-bit RISC microcontroller cores, with the ARM7 being of von Neuman architecture and the ARM9 being of Harvard architecture. Integrated circuit designers and system-on-chip designers combine the basic cores with other peripherals to produce a wide variety of proprietary low cost, low power, high speed integrated circuits.

ARM7 and ARM9 cores both feature the EmbeddedICE core extension as on-chip debug resource. This consists of two real-time watchpoint units, with associated control and status registers, as well as a set of registers implementing a communications link with the debugger, referred to as the Debug Communications Channel (DCC). One or both watchpoint units can be programmed to halt the execution of instructions by the ARM core. Execution is halted when a match occurs between the values programmed into the EmbeddedICE macrocell and the values currently appearing on the address and/or data busses. Any bit can be masked to prevent it from affecting the comparison. Either watchpoint unit can be configured to be a data watchpoint (monitoring data accesses) or an instruction breakpoint.

Breakpoints are classified as hardware or software. Hardware breakpoints typically monitor the address bus value and can be set anywhere in the address space, including ROM. Software breakpoints monitor a particular bit pattern being fetched from any address. One EmbeddedICE watchpoint unit may be used to support any number of software breakpoints. Software breakpoints can normally only be set in RAM, because an instruction has to be replaced (patched) by the special bit pattern chosen to cause a software breakpoint.

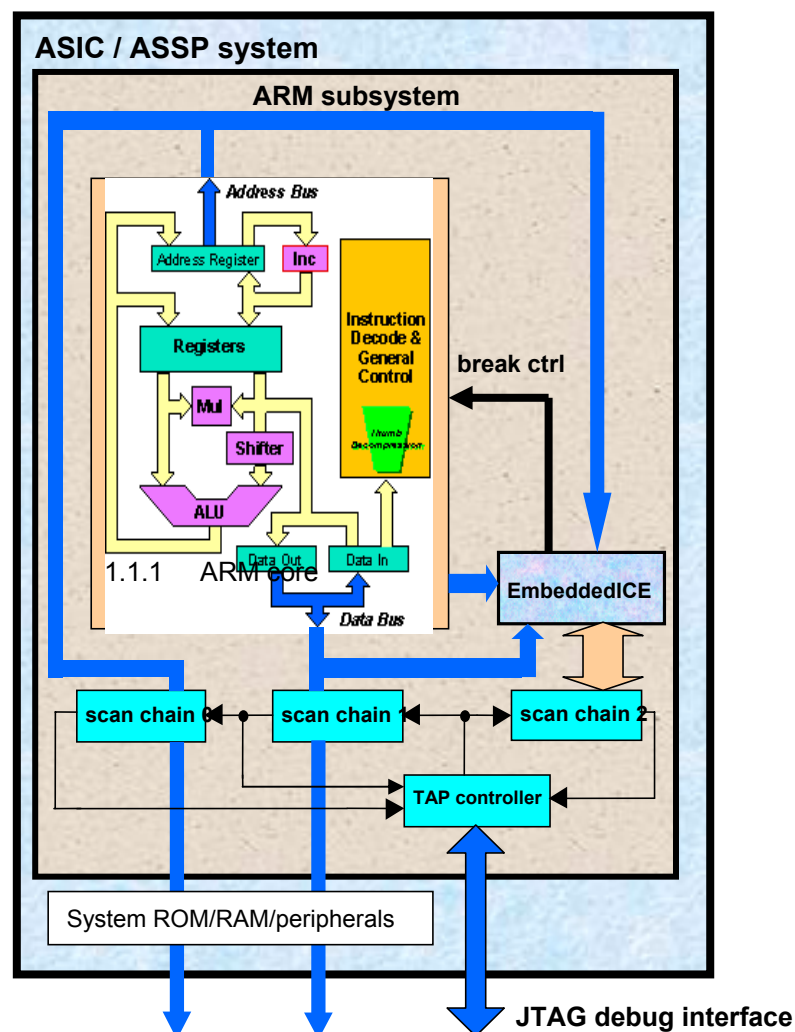


Figure 1. Typical ARM system with JTAG accessed scan chains and EmbeddedICE unit

The EmbeddedICE registers as shown in **Figure 2** are memory-mapped within the EmbeddedICE module and are accessed via the JTAG interface on Scan Chain 2. (See **Figure 1**). The Control Value registers in the Watchpoint units permit complex breakpoint conditions to be set up. Read/write accesses; byte, word or halfword accesses; instruction fetch or data accesses – all can be differentiated between. Complex breakpoint dependencies can be configured.

The EmbeddedICE Control register permits asynchronous debug requests to be asserted via the JTAG scan chain. The EmbeddedICE Status register allows for core state to be determined; and whether 32 bit ARM or 16 bit Thumb instruction mode is active on entry to debug mode. (A major advantage of ARM core technology is the ability to support a 16 bit instruction set, named Thumb, as well as the full 32 bit instruction set. This permits code density improvements and savings to be made on memory width requirements in the 32 bit environment. Zero overhead on-chip conversion from Thumb compressed instructions to the equivalent full 32 bit instruction is achieved.)

Physical access to the EmbeddedICE resource is via JTAG, and together with the JTAG TAP controller and associated scan chains, it enables powerful debug access to the target system. EmbeddedICE is incorporated on practically all ARM core implementations. Debug based on EmbeddedICE is independent of target speed. It offers the following major advantages over alternative debug schemes (including target monitors and traditional in-circuit emulators);

- Non intrusive, no target resources are required (memory, comms port, etc.),
- Low cost, (no emulation processor required, so no expensive probes),
- Full speed operation,
- Debug silicon is the release silicon.

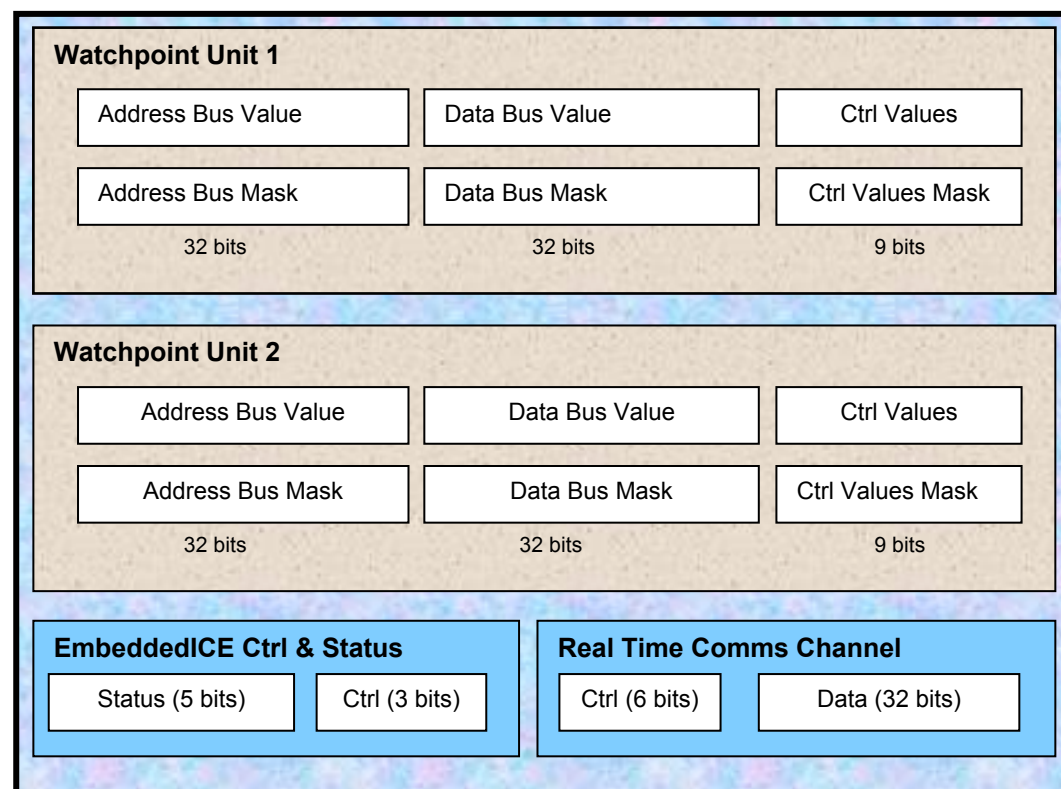


Figure 2. EmbeddedICE internal registers (as implemented on ARM7 cores)

Ashling offer a range of tools to support EmbeddedICE debugging on a wide selection of ARM7 and ARM9 cores. Opella and Genia offer the functionality described in this document, while Vitra extends the use of the JTAG port to also allow configuration of on-chip ETM resources for trace debugging (when present).

4.2 Debug features enabled by EmbeddedICE

The target system's JTAG port and EmbeddedICE unit provide the following basic core access

- program download,
- execution control, including start/stop, stepping, breakpoints and watchpoints
- system interrogation, including registers and memory
- EmbeddedICE reset.

Building on these capabilities, suitable debugger software, such as Ashling's PathFinder for ARM, interacting with the target system's EmbeddedICE debug port via suitable debug hardware tools such as the Ashling Opella or Genia, can then offer the following functionality;

- source code display,
- assembly code display
- code browser,
- stack,
- core registers (all banks),
- peripheral registers (via memory window),
- memory locations,
- I/O locations (via memory window),
- HLL variables, global and local.

Dedicated support for execution control using the EmbeddedICE WatchPoint registers is offered via dialog boxes

- breakpoints, software & hardware
- watchpoints,

In addition, support for system reset control and detection as well as auto target voltage matching is incorporated in the Ashling tools. File handling and scripting capabilities of PathFinder enable the user to develop automated debug/test sessions.

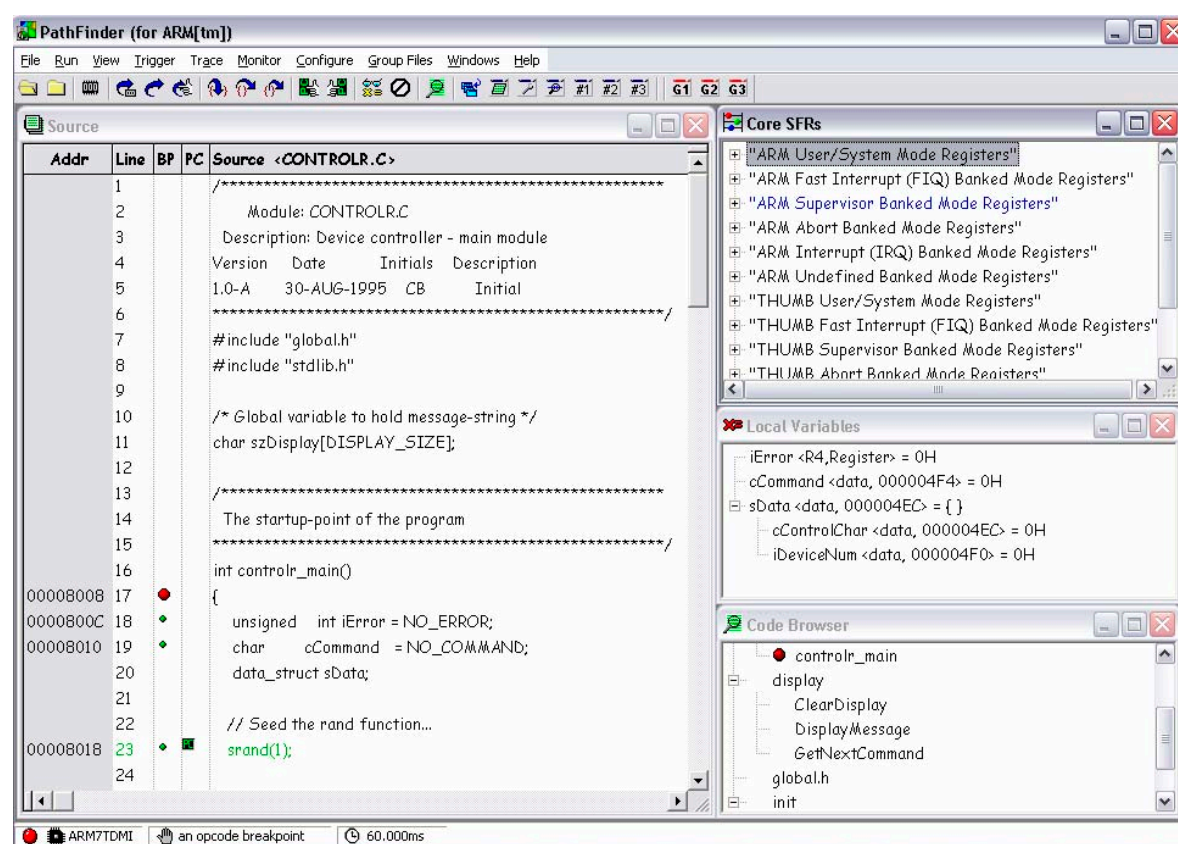


Figure 3. Screenshot of PathFinder, Ashling's source level debugger for ARM system debug

Feature	Opella	Genia	Vitra
Code download	√	√	√
Go/stop	√	√	√
Core registers read/write	√	√	√
Peripheral registers read/write	√	√	√
On-chip memory read/write	√	√	√
External memory read/write	√	√	√
System I/O read/write	√	√	√
Hardware breakpoints	√	√	√
Software breakpoints	√	√	√
Data watchpoints	√	√	√
Semihosting support	√	√	√
Debug Communications Channel	√	√	√
Single step assembly line	√	√	√
Single step high level line	√	√	√
Function step in/over/out	√	√	√
Scripting language	√	√	√
EmbeddedICE reset control	√	√	√
System reset control	√	√	√
Simultaneous assembly/source display	√	√	√
EmbeddedICE-RT/RealMonitor	√	√	√

Table 1. Ashling tools' support for EmbeddedICE features

4.3 Debug Communications Channel (DCC)

To overcome the requirement for the core to be stopped to interrogate the target system, the EmbeddedICE macrocell contains a Debug Communication Channel (DCC), which is available for data transfers to and from the host debugger during runtime. The DCC and its associated registers are shown as the Real Time Comms Channel in **Figure 2**.

Data may be passed between the target and the host debugger using the JTAG port and an EmbeddedICE interface, **without stopping** the program flow or entering debug state.

The 32-bit data register is used to pass information from the core to the host and vice versa. Reads/writes by the core are effected using standard core instructions (coprocessor `MCR` and `MRC` instructions in ARM mode). Host debugger communications is via the JTAG interface.

This functionality is very useful as it allows data transactions between the target and the host debugger without stopping the target – an important constraint in automotive, motor control and disk drive applications, to name but some.

4.4 Semihosting

Semihosting is a mechanism whereby the ARM target communicates I/O requests made in the application code, such as those provided in the standard ANSI C library (`printf()`, `scanf()`, etc.), up to the host computer running the debugger, rather than having a screen/keyboard/disk on the target system itself.

Semihosting capability is very useful during code development by catering for diagnostic messages, user inputs, etc.

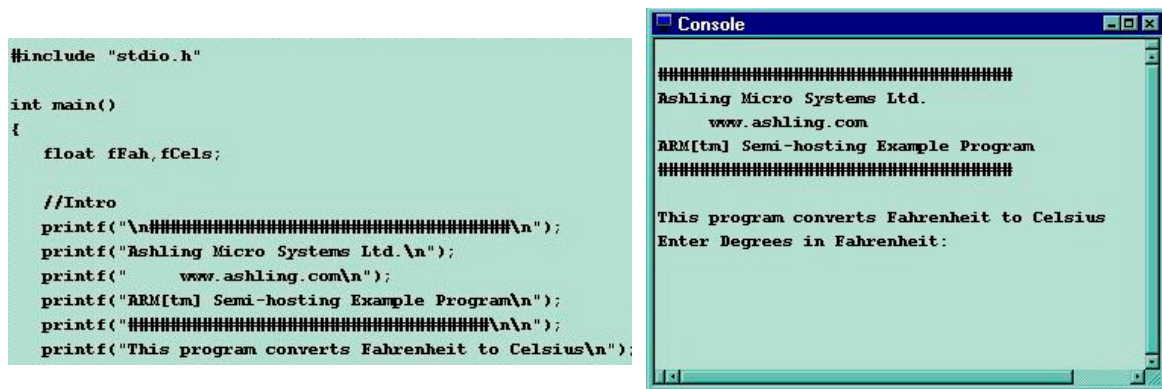


Figure 4. All semi-hosted `printf` statements are redirected to PathFinder's *Console* window when executed.

Semihosting instructions cause a software interrupt (SWI) to occur. Debugger support for semihosting can then be effected in either of two ways;

4.4.1 Standard Semihosting

Standard Semihosting involves a breakpoint being set on the SWI vector or handler, the target system stops when the breakpoint is encountered and control passes to the debugger, which then handles the semihosting operation. For example, for the `printf()` operation described above, the debugger displays the message to be printed in a dedicated Console window, having retrieved the information for display via debugger access to the target memory system. When the operation is complete, control is returned to the target and normal operation resumes.

Note that semihosting support is transparent to the user, i.e. while he has the option to enable or disable it, the actual breakpoint setting etc. is handled by the debugger. Standard Semihosting may cause problems to time critical applications due to the target being stopped while the I/O is being handled.)

4.4.2 DCC-based Semihosting

DCC Semihosting on the other hand does not cause the target processor to stop. Instead a SWI handler is installed in the target's memory which intercepts semihosting SWIs and sends a request for a semihosting operation to be carried out using the processor's Debug Comms Channel.

The debugger hardware receives the request and communicates with the SWI handler on the target, requesting that memory is read and written as necessary. On completion, execution restarts from the instruction after the semihosting SWI.

While such a method has the advantage of not stopping the target processor, it does suffer the drawback of requiring target resident code. Also its use precludes simultaneous use of the DCC for other purposes.

Note that the debugger must allow for the SWI handler to be installed in target memory in addition to the user's program and data. The handler is under 1kbyte in size and it is necessary that there is free RAM space to accommodate it – ARM Ltd recommend locating this handler at 0x10000 less than the target system's top of memory.

4.5 EmbeddedICE-RT and RealMonitor

Recently EmbeddedICE has been developed to permit debug actions without stopping the core. Whereas the standard EmbeddedICE silicon permits breakpoints/watchpoints to be set which will cause the core to be stopped, EmbeddedICE-RT permits debug actions as well as memory accesses to be carried out **without stopping** the target system. It is achieved by causing breakpoints/watchpoints to cause debug interrupt service routines (ISR) to be activated rather than stopping the core and entering debug mode. It is enabled by connecting EmbeddedICE signals to the ARM-powered device's Interrupt Controller (EmbeddedICE-RT).

The RealMonitor software consists of a target-resident component **RMTarget** which communicates with the host debugger component **RMHost** using the *Debug Communications Channel* (DCC) over JTAG. **RMHost** converts generic *Remote Debug Interface* (RDI) requests from the debugger into DCC-only RDI messages which are sent to **RMTarget** via the JTAG unit (e.g. Opella).

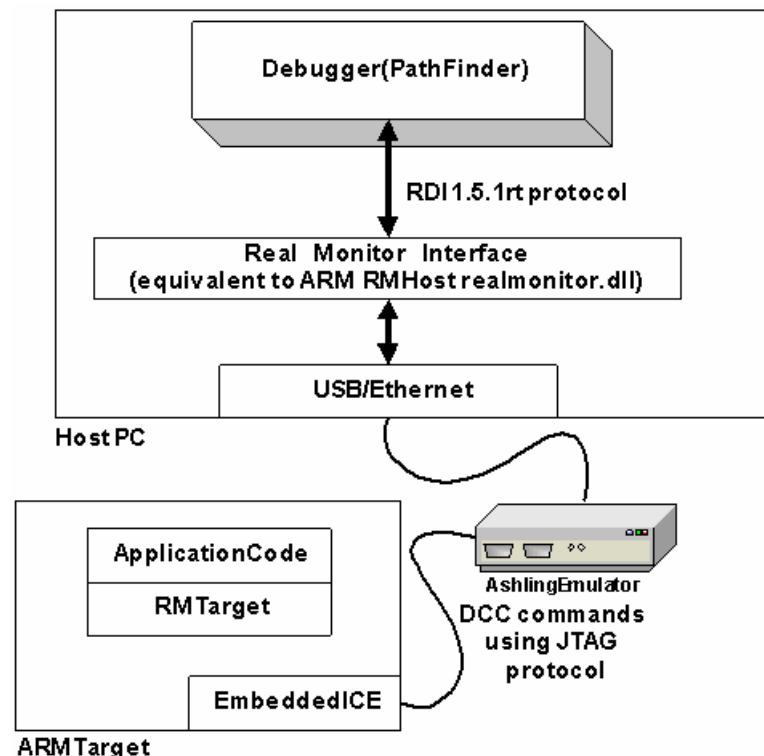


Figure 5. RealMonitor Overview

To allow non-stop 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 of the event. This allows the user's application to continue running without stopping the processor.

RealMonitor considers the target application to consist of two parts:

- a foreground application running continuously,
- a background application containing interrupt and exception handlers that are triggered by certain events (interrupts, exceptions) in the target system,

When one of these exceptions occur and the target code does not handle it, the following happens:

- RealMonitor enters a loop, polling the DCC while also stopping 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.

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 the user 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.
- It allows the user to establish a debug session to a currently running system.
- It can pass application-specific or RTOS-specific information asynchronously back to the debugger.
- 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, RMTTarget takes up only about 2KB of code and data.

To summarise, run-time debug requests cause target program execution to vector to an Interrupt Service Routine (ISR) which carries out the required debug activity, after which normal code execution resumes. Higher priority interrupts (i.e. all of the user's systems interrupts) continue to be serviced throughout this time (as per normal interrupt handling). At all times, the core is running from the main system clock.

To read or write memory without stopping the processor, a similar mechanism is used. The user selects a memory location to be read or written. The debugger sends commands via JTAG to the monitor program, which momentarily interrupts the application. The monitor program carries out the required operation; control is then returned to the application.