Application Note 31

Using EmbeddedICE

Document number: ARM DAI 0031C
Issued: February 1999
Copyright ARM Limited 1999

ARM

Application Note 31
Using EmbeddedICE

Copyright ©1999 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
February 1999	C	Third release

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

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

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1	Introduction	2
2	EmbeddedICE Principles	3
3	Connecting and Powering up the EmbeddedICE Interface	6
4	Configuring the Debugger and EmbeddedICE Interface	7
5	Watchpoints and Breakpoints	9
6	Vector Breakpoints and Exceptions	11
7	Semihosting	13
8	Debugging Applications in ROM	16
9	Accessing the EmbeddedICE Macrocell Directly	19
10	Timer Accuracy	21
11	Target Board Memory Layout	22
12	System Design Considerations	23

1 Introduction

EmbeddedICE is an extension to the architecture of the ARM family of RISC processors, and provides the ability to debug cores that have been deeply embedded into systems. It consists of three parts:

- A set of debug extensions to the ARM core.
- The *EmbeddedICE macrocell*, which adds a JTAG TAP controller and breakpoint/watchpoint logic to the basic debug extensions.
- A protocol converter, to provide communication between the EmbeddedICE macrocell and the host computer, such as the EmbeddedICE interface or Multi-ICE. For full details on how to use Multi-ICE, refer to the *Multi-ICE User Guide* [ARM DUI 0048].

This application note examines some of the issues involved with debugging a system using an EmbeddedICE interface, and also some of the considerations which need to be made when designing such a system. Much of this information is also applicable if you are using a Multi-ICE.

Throughout this application note, it is assumed that you are using the ARM Software Development Toolkit, version 2.5. If you are using an earlier version of the SDT (such as SDT 2.11a), you should refer to the supplied user guide for details.

2 EmbeddedICE Principles

2.1 Debug extensions to the ARM core

The presence of debug extensions to the ARM core are indicated by the `D` suffix on the core name. The extensions consist of a number of scan chains around the core and some additional pins that are used to control the behavior of the core for debug purposes.

The three most significant pins are:

BREAKPT allows external hardware to halt processor execution for debug purposes. When HIGH, the current memory access is tagged as breakpointed. If the memory access is an instruction fetch, the core enters debug state if and when the instruction reaches the execute stage of the pipeline. If the memory access is for data, the core enters debug state when the current instruction completes execution. The EmbeddedICE macrocell can also assert this input to the core.

DBGRQ is a level-sensitive input that causes the core to enter debug state immediately after the current instruction has completed execution. This allows external hardware to force the ARM into debug state.

DBGACK is an output from the ARM that goes HIGH when the core is in debug state. This allows other peripherals or the debugging system to determine the current state of the core, and act accordingly.

For more details, please refer to the debug interface section of the relevant ARM core datasheet or technical reference manual.

2.2 EmbeddedICE macrocell

The EmbeddedICE macrocell is the integrated on-chip logic that provides debug support for ARM cores. Its presence is indicated by the `I` suffix on the core name.

The EmbeddedICE macrocell is programmed in a serial manner through the *Test Access Port* (TAP) controller on the ARM using the JTAG interface (see ***12 System Design Considerations*** for details of designing this into your own target).

The EmbeddedICE macrocell consists of two real-time watchpoint units, together with a control and status register, and a set of registers implementing a communications link with the debugger, referred to as the Debug Communications Channel. For more details on this link, see Applications Note 38 (Debug Communications Channel). One or both watchpoint units can be programmed to halt the execution of instructions by the ARM core by way of its **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into the EmbeddedICE macrocell and the values currently appearing on the address bus, data bus, and various control signals. Any bit can be masked to prevent it from affecting the comparison. Either watchpoint unit can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches).

For more information, please refer to relevant section of the appropriate ARM data sheet or a technical reference manual.

2.3 The protocol converter

This translates the debug protocol messages sent out by the debugger into JTAG signals that can be sent to the EmbeddedICE macrocell (and vice versa). At the time of writing, ARM has produced two such protocol converters: the EmbeddedICE Interface and the Multi-ICE Unit.

The EmbeddedICE Interface

The EmbeddedICE interface is ARM's original protocol converter. The interface can be connected to the host computer using either the built-in serial port or both the built-in serial port and the parallel port. The serial port can be used for bidirectional transfers, but if the parallel port is used, it will only be used to increase the download speed (up to around 15KB per second).

The Multi-ICE Unit

Short for Multiprocessor EmbeddedICE Interface, this unit is a more recent design that can communicate with a wider range of ARM cores, and also several ARM cores within the same ASIC. Multi-ICE is connected to the host computer by way of the parallel port and has a download performance of around 100KB per second on modern PCs (processor-dependent and parallel port-dependent).

2.4 How EmbeddedICE differs from a debug monitor

A debug monitor, such as Angel, is an application that runs on the board in conjunction with the user application, and requires some resource to be available for it on the board. When the board powers up, Angel installs itself by initializing the vector table so that it takes control of the board when an exception occurs. Communication coming in from the host causes an interrupt, halting the user application and calling the appropriate code within Angel. Angel then returns to the user application. This can complicate matters if the application also requires access to interrupts. Similarly, if the application requests some form of I/O to the host, this is implemented within the application using a SWI instruction that is dealt with by Angel's SWI handler (For further details, see ***7 Semihosting***). This means that Angel requires ROM to store the debug monitor code, RAM to store its data, and control over the exception vectors to allow it to gain control of the ARM while the user application is running.

The EmbeddedICE debug architecture, on the other hand, requires no such resource. Rather than existing as an application on the board, it works by using a combination of the additional debug hardware on the core and the interface box that handles communication between the core's debug hardware and the host. The EmbeddedICE debug architecture has been designed to allow debugging by way of the JTAG port to be as non-intrusive as possible:

- the debuggee needs no special hardware to support debugging (the EmbeddedICE macrocell and the JTAG TAP controller are all that is required)
- no memory in the debuggee system need be set aside for debugging, and no special software need be incorporated to allow debugging

- execution of the debuggee should only be halted when a breakpoint or watchpoint has been hit, or the user requests that the debuggee is halted.

Note *Although the EmbeddedICE debug architecture requires no memory on the target to operate, the target will still require some memory for executing its own application code.*

2.5 The effect of EmbeddedICE on the target

Although EmbeddedICE debugging is generally non-intrusive, there are four exceptions:

- When an ARM debugger is started up, it attempts to find out the state of the debuggee. To do this, it halts the debuggee and inspects the state of the ARM registers. This, however, can be considered non-intrusive if the debugging session is started after the debugger has been started.
- Watchpoints on structures or arrays larger than one word may cause the debuggee to halt execution when writes occur close to the watchpointed area. EmbeddedICE will restart execution transparently to the user, but this may still cause problems if the application is real-time. For more information, see ***4.1 Configuring armsd***.
- Semihosting: provides a mechanism for providing I/O facilities from the host computer. However, in order to do this, the protocol converter will take control of the SWI entry in the vector table. See ***7.1 Adding an application SWI handler when using EmbeddedICE*** for more details.
- Vectors: provides a mechanism for trapping exceptions for which your application may not provide a handler, for example, during early development stages. However, this requires breakpoints to be set on the vector table. See ***6 Vector Breakpoints and Exceptions*** for more details.

3 Connecting and Powering up the EmbeddedICE Interface

You will need to connect the EmbeddedICE interface to both a host computer and an ARM target board.

Note *For details on connecting and powering up a Multi-ICE, refer to the Multi-ICE User Guide* [ARM DUI 0048].

To connect and power up the EmbeddedICE interface:

1. Connect the supplied serial cable between the 9-pin serial port of EmbeddedICE and the serial port of the host computer.
2. Optionally, connect the supplied 25-way D-type parallel cable between the EmbeddedICE and the host computer's parallel printer port.
3. Power up the target board, as detailed in the appropriate documentation.
4. Connect the EmbeddedICE interface to an external +7 to +9V DC (unregulated) power supply, 500mA or greater. The recommended voltage is +7.5V.

   Note that the 2.1mm power connector should have the positive supply connected to the center pin, as shown in ***Figure 3-1: Power connection***:

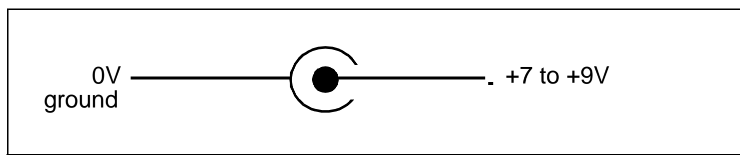

Figure 3-1: Power connection

5. Connect the EmbeddedICE interface to the target board using a14-way IDC JTAG cable. (The actual location of the JTAG connection on the target board depends on the board you are using. Refer to the documentation that accompanied the board for details.)
6. Power up the EmbeddedICE interface. The red LED should light up.
7. You are now ready to run a debug session using the EmbeddedICE interface.

4 Configuring the Debugger and EmbeddedICE Interface

The way in which the EmbeddedICE interface accesses your target hardware is controlled by the software image, or *agent*, in the built-in ROM. This software contains configuration files to allow the EmbeddedICE interface to talk to two ARM variants. These are:

ARM7DI	ARM7 core with debug extensions and EmbeddedICE macrocell (includes ARM7DMI)
ARM7TDI	ARM7 core with Thumb and debug extensions and EmbeddedICE macrocell (includes ARM7TDMI).

When you start the EmbeddedICE interface, it automatically defaults to a particular configuration, according to which version of the agent is installed. This will normally be ARM7TDI.

You can change the current configuration easily from within the debugger. See ***4.1 Configuring armsd*** and ***4.2 Configuring ADW/ADU*** for details.

Note *Always ensure the selected configuration is correct. Otherwise, you may have problems when you try to run the image.*

On ARM7DI, using the ARM7TDI configuration can lead to breakpoints not being hit. If you press ^C (using armsd) or the **Stop** button (using the ADW) to interrupt execution, you enter the undefined instruction trap.

On ARM7TDI, using the ARM7DI configuration can result in a blank execution window in the ADW.

If the following message is displayed, the wrong configuration may be being used and the EmbeddedICE interface has failed to synchronize with the target:

```
Target processor not stopped
```

Note that this error can result from other causes. See ***12 System Design Considerations***.

You can usually ignore the following message if it is displayed when you start up the debugger (or reload an image):

```
Error during initialization:
Recoverable error in RDI initialization
```

If you have problems with your current version, you are advised to contact your supplier and obtain the latest version of the agent. Refer to the website `www.arm.com`. At the time of writing version 2.07 is the latest version (as shipped with the SDT 2.11a and SDT 2.50).

Note *Early EmbeddedICE interfaces were shipped with version 1.0x of the agent software. These communicated with the debugger using the Remote Debug Protocol (RDP), and these were compatible with programs that use Demon SWIs for semihosting (not Angel SWIs). See Application Note 39: Demon and RDP [ARM DAI 0039] for more details. The use of RDP is not supported in SDT v2.5, and therefore, if you are using 1.0x, you will need to upgrade the ROM to 2.0x.*

Using a replacement version of the agent

You can use a replacement version of the agent in either of the following ways:

- Program a new EPROM (AM27C010-120DC or equivalent) with this image, and replace the existing EPROM on the lower of the two circuit boards in the EmbeddedICE interface (taking care to reassemble the two boards correctly).
- Download the new image as the first thing you do during a debug session. This can be done from within the debugger using the `loadagent` command. See ***4.1 Configuring armsd*** for details.

  Note that the image must be reloaded whenever you cycle the power or press the **Reset** button on the EmbeddedICE interface.

Note *If a new image is downloaded to the EmbeddedICE interface, any later debugger startup message will state that the ROM CRC has failed. This is because the downloaded image is CRC-checking against the ROM image whose checksum is usually different. This message can therefore be ignored.*

4.1 Configuring armsd

Details of the options used for controlling the debugging of a remote target can be found in section 7.2.1 of the *ARM Software Development Toolkit Reference Guide* [ARM DUI 0041].

For example, to download the `example` executable to a little-endian target using the EmbeddedICE interface, the standard serial and parallel ports, and at the maximum linespeed supported by the EmbeddedICE interface, enter:

```
armsd -li -adp -port s,p -linespeed 38400 example
```

where:

`-li`	is the little-endian switch. Use `-bi` for big-endian.
`-adp`	is the switch to select remote debugging using the Angel Debug Protocol.
`-port s,p`	indicates that the standard serial and parallel ports are to be used.
`-linespeed`	selects the required serial linespeed to use.
`example`	is the filename.

Once in armsd, there are a number of commands that allow you to change the EmbeddedICE interface configuration and agent facilities, though these will not normally be required by most users. See section 7.8 of the *ARM Software Development Toolkit Reference Guide* [ARM DUI 0041] for more details.

4.2 Configuring ADW/ADU

For details of how to configure ADW/ADU to communicate with an EmbeddedICE interface, please see section 3.7.3 and 3.7.4 of the *ARM Software Development Toolkit User Guide* [ARM DUI 0040].

5 Watchpoints and Breakpoints

As with ARMulator, the ARM debuggers provide break, watch, unbreak, and unwatch facilities with EmbeddedICE linked targets. However, you should be aware of the following issues.

5.1 Watchpoints

All ARM debugger watchpoints are data-changed watchpoints, that is, they are not activated if the data point is read or written to with the same data value as the one currently in memory.

See ***9 Accessing the EmbeddedICE Macrocell Directly*** for details of how to implement other forms of watchpoint.

Hardware versus software watchpoints

Hardware watchpoints are implemented using an EmbeddedICE macrocell point to spot data writes to addresses that fall inside a mask. This type of watchpoint is efficient because execution stops only when the relevant data is written. However, it completely ties up an EmbeddedICE macrocell point. Note also that if a structure or an array is being watchpointed, the mask is likely to include some addresses that are not part of the object being watchpointed. In this case, writes to these unwanted addresses are filtered out by the EmbeddedICE interface. Execution performance is slightly degraded because the processor is stopped when the unwanted watchpoint is hit, and then restarted automatically by the EmbeddedICE interface.

Software watchpoints make no use of the EmbeddedICE macrocell. Instead, after each instruction is executed, the data locations concerned are examined to see whether their values have changed. If a value has changed, execution is halted. Otherwise, execution is resumed. This type of watchpoint drastically reduces execution performance. In addition, it clearly cannot be used on write-only areas of memory, such as some memory-mapped device registers.

5.2 Inspecting points

When you inspect the current breakpoints and watchpoints (using the `watch` or `break` commands without arguments within armsd, or by choosing **Breakpoints** or **Watchpoints** from the **View** menu within ADW/ADU), the output specifies whether they are hardware or software points.

Hardware versus software breakpoints

Hardware breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch from the appropriate address. This works in all cases, even if the program being debugged modifies itself as it executes, or if the code is in ROM. However, it completely ties up one of the two available EmbeddedICE macrocell point units.

Software breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch of a particular bit-pattern. This bit-pattern will have been previously stored at the appropriate location, and the real instruction stored in the host debugger memory. Therefore, self-modifying code, or code in ROM, cannot be debugged using this type of breakpoint.

(EmbeddedICE will detect non writeable memory and not attempt to use software breakpoints for that region.) Any number of software breakpoints can be supported using a single EmbeddedICE macrocell point.

5.3 Watchpoints, breakpoints, and the program counter

Watchpoints are taken when the data being watchpointed has changed. When this happens, the program counter is updated to point to the instruction following the one that caused the watchpoint to be taken. The value of the watchpointed data is therefore the new value, not the old value.

Breakpoints are taken when the instruction being breakpointed reaches the execution stage of the pipeline, but before it is executed. So, when the breakpoint is taken, the program counter is not updated, and retains the address of the breakpointed instruction.

Note *Inside the core of an ARM, the program counter typically points to two instructions beyond the currently executing instruction (historically, this is the address of the instruction currently being loaded into the fetch stage of the pipeline). However, the ARM debuggers simplify this by reporting a modified value for the program counter, so that when it is displayed within the debugger, its contents are the address of the instruction being (or about to be) executed.*

6 Vector Breakpoints and Exceptions

When the debugger starts executing the target application, the EmbeddedICE interface puts into place any breakpoints specified by the debugger internal variable `$vector_catch`. (This means that when you start executing, user breakpoints and watchpoints may have to be downgraded from hardware ones to software ones without warning.) The `$vector_catch` variable is used to indicate whether or not execution should be trapped when various conditions arise. The default value is `%RUsPDAifE`, where capital letters indicate that the condition is to be intercepted.

Breakpoint	Description
R	Reset
U	Undefined instruction
S	SWI
P	Prefetch abort
D	Data abort
A	Address exception
I	IRQ
F	FIQ
E	Error (reserved for possible, future software fault detection)

Table 6-1: Breakpoints

This is useful if the application contains no specific handler for a particular exception.

On ARM9TDMI family devices, additional hardware in the core allows vector catching to take place without the need to set breakpoints. See the ARM9TDMI Technical Reference Manual for more details.

In normal usage, the SWI flag within `$vector_catch` remains lowercase, as finer control is provided by the debugger internal variables `$semihosting_enabled` and `$semihosting_vector` (see ***7 Semihosting*** for further details). Setting the `s` bit to `$vector_catch` will result in unpredictable behavior.

In the case of a system with no interrupt handler that has an active source of interrupts, set up the exception vectors to mask out all the interrupts by loading the instructions shown below.

You can do this from the armsd prompt using the following commands. You do not need to type the text shown after the semicolon (;) because these are comments showing the instruction encoded in each hexadecimal value.

```
0x18 = 0xe1a00000; NOP
0x1C = 0xe14fd000; MRS r13, spsr
0x20 = 0xe38dd0c0; ORR r13, r13, 0xC0
0x24 = 0xe169f00d; MSR spsr, r13
0x28 = 0xe25ef004; SUBS pc, lr, #4
```

You can simplify the above by placing the commands in an `armsd.ini` startup file. All the commands are then executed when armsd is invoked. For information on setting up an armsd startup file, refer to the *ARM Software Development Toolkit Reference Guide* [ARM DUI 0041].

If you are using ADW or ADU, choose **Memory** from the **View** menu, and then modify the contents of the appropriate addresses. Alternatively, invoke a script from the command window

There may be slight problems with applications that rely on interrupts occurring in real time. When doing any operation by way of EmbeddedICE, other than `go` (for example hitting a breakpoint), interrupts may occur during the operation, as going into debug state takes time. If this causes major problems, the workaround is to turn off interrupts (either patch the start-up code, or manually set the CPSR). Then, when you require an interrupt, use the debugger to enter the correct mode, set `pc` equal to vector address, and restart execution.

6.1 $vector_catch with ROM at 0x0

In systems where there is ROM at address 0x0, care needs to be taken with the setting of `$vector_catch`. See ***8 Debugging Applications in ROM*** for more details.

7 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.

On target boards containing Angel, this is implemented using a set of defined SWIs. Therefore, Angel installs a SWI handler when the board is powered up, and, when the target executes a SWI instruction, Angel carries out the required communication with the host.

When using the EmbeddedICE interface, semihosting is handled differently. It is implemented by *faking* the SWI handler that Angel would have installed on the SWI vector. The EmbeddedICE interface installs a breakpoint on the SWI vector, and, when this breakpoint is hit, checks to see what the SWI number was. If the SWI is recognized, the EmbeddedICE interface emulates it and transparently restarts execution of the application. If the SWI is not recognized as a semihosting SWI, the EmbeddedICE interface halts the processor and reports an error.

This semihosting mechanism can be disabled or changed by making use of the following debugger internal variables:

`$semihosting_enabled`
: By default, this variable is set to 1 to enable semihosting. Setting it to 0 disables semihosting. This can be useful, for example, when debugging applications running from ROM. Disabling semihosting in such situations frees up another watchpoint unit. The `s` bit in `$vector_catch` should not be enabled as an alternative to `$semihosting` enabled.

`$semihosting_vector`
: This variable controls the location of the breakpoint set by the EmbeddedICE interface to detect a semihosted SWI. It is set to 8 by default. Note that the EmbeddedICE interface will return directly to the instruction following the SWI instruction in your code after handling the semihosted SWI, completely bypassing the contents of the `$semihosting_vector` address. (This is done by examining the contents of `lr`.)

  Note that if this variable is set to zero, this does not imply address 0. Address 8 is used instead. However, all exceptions and interrupts would be trapped and reported as an error condition, no matter what the value of `$vector_catch`.

In ADW/ADU, both of these variables can be accessed by selecting **Debugger Internals** from the **View** menu or through the command line window.

Note *When using Multi-ICE, an additional way of implementing semihosting is available. This uses the debug communications channel so that the core is not stopped while semihosting takes place. This is enabled by setting `$semihosting_enabled` to `2`. Refer to the Multi-ICE User Guide [ARM DUI 0048] for more details.*

7.1 Adding an application SWI handler when using EmbeddedICE

In addition to using semihosted SWIs, many applications will also need to install their own SWI handlers into the vector table. This must be done in such a way that the application SWI handler will successfully cooperate with the EmbeddedICE semihosting mechanism. To do this, the application SWI handler must be installed into the vector table. Then, the `$semihosting_vector` must be modified to point to a location at the end of the application handler that is only reached if your handler does not recognize the SWI (or recognizes it as a semihosting SWI).

There are two vital points to note when doing this:

1. It is essential that the actual position within the application handler to which the `$semihosting_vector` points is correct.
2. At the point the EmbeddedICE interface traps the SWI, your own SWI handler *must* have restored all registers to their original values at the moment your SWI handler was entered. Typically, this means that your SWI handler should store the register to a stack on entry, and restore them before falling through to the semihosting vector address. If this is not done, semihosting will be final.

For example, a particular SWI handler may detect if it has failed to handle an SWI and branch to an error handler (see ***Exception Handling*** in the *ARM Software Development Toolkit User Guide* [ARM DUI 0040] for further details of writing SWI handlers):

```
                                ; r0 = 1 if SWI handled
    CMP r0, #1                  ; Test if SWI has been handled.
    BNE NoSuchSWI               ; Call unknown SWI handler.
    LDMFD sp!, {r0}             ; Unstack SPSR...
    MSR spsr, r0                ; ...and restore it.
    LDMFD sp!, {r0-r12,pc}^     ; Restore registers and return.
```

This code could be modified for use in conjunction with EmbeddedICE interface semihosting as follows:

```
                                ; r0 = 1 if SWI handled
    CMP r0, #1                  ; Test if SWI has been handled.
    LDMFD sp!, {r0}             ; Unstack SPSR...
    MSR spsr, r0                ; ...and restore it.
    LDMFD sp!, {r0-r12,lr}      ; Restore registers.
    MOVEQS pc, lr               ; Return if SWI handled.
Semi_SWI
    MOVS pc,lr                  ; Fall through to EmbeddedICE
                                ; interface handler.
```

The `$semihosting_vector` variable should then be set up to point to the address of `Semi_SWI`. Note that the instruction at this address never gets executed because the EmbeddedICE interface returns directly to the application after processing the semihosted SWI. However, using a normal SWI return instruction ensures that the application does not crash if the semihosting breakpoint is not set up. The semihosting action requested is not carried out, and the handler simply returns.

There is one slight complication if the application is linked with the semihosted ARM C library, and therefore uses the C library's startup code. If `$semihosting_vector` is set to the *fall through* part of the application SWI handler before the application starts execution, the semihosted SWIs that are called by the library startup (for example, the SWI to get heap and stack information) triggers an unknown watchpoint error.

This happens because at this point, the SWI vector has not yet had the application handler installed into it, and may still contain the software breakpoint bit pattern. This triggers a breakpoint that the EmbeddedICE interface no longer knows about because the `$semihosting_vector` address has moved to a place that cannot currently be reached. To prevent this from happening, change the contents of `$semihosting_vector` just before the application installs its own handler, typically by setting a breakpoint in the main code.

Note *If semihosting is not required at all by an application, this process can be simplified. All that is needed is to set* `$semihosting_enabled` *to 0.*

Care is therefore required when moving an application that previously ran in conjunction with Angel onto an EmbeddedICE-based system. On Angel-based systems, application SWI handlers are typically added by moving (and adjusting) the contents of the SWI vector (as installed by Angel) to another place, and installing the application SWI handler into the SWI vector. Such a method will not work correctly under the EmbeddedICE interface because there is no instruction to move out of the SWI vector. So, when the application handler fails to handle a particular SWI, it will jump to a storage location and try to execute a completely random instruction that will typically be undefined. Therefore, when moving an application onto an EmbeddedICE-based system, it is essential to convert to the correct way of installing the application and semihosted SWI handlers

8 Debugging Applications in ROM

This section examines some of the issues involved with debugging applications in ROM using an EmbeddedICE interface (or Multi-ICE).

8.1 Debugging from reset

The EmbeddedICE interface can be used to debug systems running in ROM. Typically, when a target board with an application stored in ROM is powered up, the application begins running. Therefore, when the debugger is started up on the host, the processor on the target is stopped. At this stage, the application could be at any point in its execution lifetime, depending on when the debugger was started.

This means that the state of the system can be examined and execution can be restarted from the current place. In some cases, this may be sufficient to achieve what is required. However, in many cases, it is preferable to restart execution of the application as if from power-on. There are two ways of approaching this:

- by faking the reset
- by carrying out an actual reset of the target.

If you have set up a basic **nTRST** connection, as described in ***Basic nTRST connection*** in ***12.3 Reset and JTAG signal connection***, you may only fake the reset. If you have implemented additional use of **nICERST** as described in ***Separate control of nTRST and nRESET***, you may choose either method.

When you debug code running from ROM, ensure that at least one watchpoint unit remains available to allow breakpoints to be set on code in ROM (as software breakpoints cannot be used). The chances of the debugger taking these units for its own use can be reduced by not using semihosting or vector catching. To do this, the following debugger internal variables should be set as soon as possible after starting up the debugger:

```
$semihosting_enabled = 0
$vector_catch = 0
```

Then, setup any ROM breakpoints before any non-ROM breakpoints or watchpoints are set. Otherwise, again, the watchpoint units may be fully occupied, causing the attempt to set the ROM breakpoint to fail:

```
Error: Too many breakpoints
```

Watchpoint units must then be freed (by deleting breakpoints / watchpoints) before the ROM breakpoints can be set.

Another complication with debugging a system in ROM is that the ROM image cannot contain any debug information. When debugging using the EmbeddedICE interface, symbol or source code information is available by loading the relevant information into the debugger from a file on the host. This is described in ***8.2 Accessing debug information***.

Faking a reset

Typically, you can simulate a reset from within the debugger by setting:

PC to 0	address of the reset vector
CPSR to %IF_SVC32	to change into supervisor mode with interrupts disabled.

This simulates the state of the ARM at power on/reset, but it does not allow for a reset memory map or the initialization of any target-specific features (such as registers). It is therefore advisable to modify any such target-specific features to resemble their startup configuration before executing the application again, if this is possible. You can automate this procedure with the debugger's scripting facility (the `obey` command or `-script` option).

Carrying out a real reset

Depending on the design of the reset circuitry, it may be possible to carry out a real reset of the board. However, care is required when doing this, because if the EmbeddedICE macrocell is also reset, the debugger becomes out of sync with the macrocell. Two forms of reset are required on the board:

- a full power-on reset that resets everything on the board
- a reset button that resets everything on the board except the EmbeddedICE macrocell.

See also ***12 System Design Considerations***.

Therefore, if a *hardware* breakpoint is set on the reset vector (or the address in ROM of the reset routine to which the reset vector branches), when the target is reset, the target will halt on reset as required. Note that the EmbeddedICE macrocell is not reset. This would cause the breakpoint to be lost.

Note *It is advisable to delete all other software breakpoints/watchpoints before resetting, because although they may be deleted by the reset, the debugger will think they still exist. This can cause problems when you try to delete them after the reset.*

Example: The ARM development board (PID7T)

The ARM development board also implements the required two levels of reset. In this case, the reset switch carries out the required initialization reset, thereby allowing debug from reset. All that is required in this case is to set the hardware breakpoint, and then press the **Reset** button.

8.2 Accessing debug information

In normal usage (with SDT 2.50 and later), armlink will produce an application image in ELF format. In order to program this into ROM/FLASH, this will need to be converted into a suitable binary format. This will usually be done using the *fromelf* tool.

The binary file can then be programmed into the ROM, and the debug information, from the ELF file, can be loaded into the debugger:

For armsd

Use one of the following:

```
armsd -symbols filename.axf
```

when invoking the debugger, or:

```
readsyms filename.axf
```

when armsd is running (where *filename*.axf is the ELF file produced by armlink).

For ADW/ADU

Choose **Load symbols only** from the **File** menu, and then select the ELF file corresponding to the image in the system ROM.

For more information, please see ***Writing Code for ROM*** in the *ARM Software Development Toolkit User Guide* [ARM DUI 0040].

8.3 Debugging systems with ROM at zero

When debugging ARM7-based systems with ROM at zero, rather than RAM, it is necessary to set `$vector_catch` to 0. This prevents the EmbeddedICE interface from trying to set software breakpoints on the vector table.

Note *When debugging ARM9 family systems using Multi-ICE, this is not as important because they contain special vector catch hardware.*

9 Accessing the EmbeddedICE Macrocell Directly

No new debugger commands have been added to allow you to manipulate EmbeddedICE macrocell registers. Instead, you can use the commands that display and set coprocessor registers, with coprocessor number 0 specified. Coprocessor 0 is defined so that it is never implemented, and therefore cannot clash with user-developed or ARM-developed coprocessors.

For example, the armsd command `cregisters 0` displays the contents of a number of registers that are, in fact, EmbeddedICE macrocell registers:

```
ARMSD: cregisters 0
c0 =       0x05
c1 =       0x09
c4 =       0x00
c5 = 0x00000000
c8 = 0x516ce8da
c9 = 0xbfdf0ea6
c10 = 0xbff6fd7d
c11 = 0xfbaffbff
c12 =     0x0000
c13 =       0xff
c16 = 0x00000008
c17 = 0x00000003
c18 = 0x7dfeeffb
c19 = 0xffffffff
c20 =     0x0100
c21 =       0xf6
```

To access the coprocessor 0 registers using ADW, choose **Registers** from the **View** menu, and then display the Coprocessor dialog box. Enter `0` for the coprocessor number and check the Raw (unformatted) display option.

The correspondence between coprocessor 0 registers displayed and EmbeddedICE macrocell registers is as follows: the `register address` field in the EmbeddedICE macrocell scan chain *is* the register number. For more information about the EmbeddedICE macrocell, refer to one of the ARM data sheets on an ARM core with debug capabilities (for example, ARM7DI or ARM7TDMI).

You may read EmbeddedICE macrocell registers freely in this manner, but writing them requires more care. This is because EmbeddedICE also makes use of EmbeddedICE macrocell registers to set up breakpoints and watchpoints. When you write to an EmbeddedICE macrocell register (for example, using the armsd command `cwrite 0 20 0x44`), EmbeddedICE checks to see if the breakpoint (of which that register is a part) is in use. If it is, EmbeddedICE attempts to free it (by degrading hardware breakpoints to software breakpoints), and then sets a lock on the breakpoint so that EmbeddedICE makes no further attempt to use it.

It is possible to see which breakpoints have been locked in this way by displaying the value of `$icebreaker_lockedpoints`. This debugger internal variable can also be set to unlock breakpoints. In the ARM7DI and ARM7TDI, the breakpoints are numbered 1 and 2, and bits 1 and 2 in `$icebreaker_lockedpoints` indicate their status.

If a breakpoint or watchpoint that has been set up in this way gets taken, EmbeddedICE will not know why execution has stopped (because it was not due to one of the break/watch points it is aware of), and will therefore halt debuggee execution with the report `Unknown watchpoint`.

Note that a debugger hardware watchpoint should not be used in combination with a user-programmed EmbeddedICE watchpoint unit, because EmbeddedICE will not halt execution due to the user-programmed point. In practice, this is unlikely to cause any problems because the EmbeddedICE macrocell has only two watchpoints, and you can also program the second watchpoint directly.

Take care when writing EmbeddedICE macrocell registers 0 and 1, the control and status registers. EmbeddedICE uses these to perform many of its operations. If they are written at all, they should always be returned to their original values afterwards.

Note also that debugger requests to read or write EmbeddedICE macrocell registers do not necessarily cause the registers to be read or written immediately. This is because, in the interests of efficiency, the EmbeddedICE software caches the contents of the EmbeddedICE macrocell registers, only updating changed registers before execution of the debuggee is resumed.

10 Timer Accuracy

When using the EmbeddedICE interface, or Multi-ICE, the standard ANSI `clock()` function can be inaccurate.

There are two reasons for this:

- The implementation of the Angel SWI `SWI_CLOCK` (called by the standard ANSI `clock()`routine) in the EmbeddedICE agent software, and in Multi-ICE, has a resolution of one second rather than one centisecond, despite the fact that the value of `CLK_TCK` (that gives the number of clock ticks per second) is 100.
- Currently, neither the EmbeddedICE agent nor Multi-ICE produces the timing figure itself, but instead, sends a request back to the host for it. This means that the figure returned is larger than it actually should be, because it includes the time it takes for the EmbeddedICE interface to send a request to the host and get the time back.

Combined, these two issues render inaccurate any benchmarks (such as Dhrystone) run on EmbeddedICE-based systems. Extreme care is therefore needed in such cases. This can also help to run benchmarks over a very large number of iterations to minimize the overall affect of the inaccuracies. If the application is modified to use a clock routine that takes its values from a timer routine running on the board itself, more reliable figures can be generated. The example suite provided with the ARM development board (PID7T) contains a conversion of Dhrystone, so that the timer figures are produced from the onboard timer interrupts. This can also be downloaded from the ARM website.

11 Target Board Memory Layout

The debugger variable `$top_of_memory` is read by the EmbeddedICE interface to give the top of the read-write memory, and the value of this is returned by the Angel `heap_info` SWI that is used by the C library for initialization.

The default value for `$top_of_memory` is 512KB, which is correct for the ARM development board (PID7T) without expansion RAM. If you have expanded RAM, or are using a different board with more or less RAM, set `$top_of_memory` to the correct value before you run the application.

The mechanism places the application heap above the application program, and places the application stack at the top of memory. It also requires that memory is contiguous from the end of the application to the top of memory.

If this is not the case, it is possible to define the following symbols in the application to override this mechanism, and set up the application heap and stack precisely:

```
__heap_base,__heap_limit,__stack_base,__stack_limit.
```

For example, in ARM Assembler:

```
    EXPORT __heap_base
__heap_base DCD my_heapbase_value
```

11.1 Privileged modes and stack pointers

It should be noted that the EmbeddedICE interface itself does not set up any stack pointers. Therefore, if you have linked with the full semihosted ANCI C library, you will have a stack pointer for the current mode setup (as described in ***11 Target Board Memory Layout***). However, you will not have stack pointers set up for the other modes. This contrasts when an application is run under ARMulator or Angel where the stacks are set up for you.

You must therefore ensure that your application start up code sets up the stack pointers for any other modes that may be entered, for instance, as a result of exceptions taking place.

If you are not linking with the full semihosted ANSI C library, not even the current mode stack pointer will be set up.

For more details, see **Writing Code for ROM** in the *ARM Software Development Toolkit Version 2.50 User Guide [ARM DUI 0040D]*.

12 System Design Considerations

This section examines some of the issues involved with designing systems which are to be debugged using an EmbeddedICE interface.

12.1 Memory Access Requirements

Background Information

The EmbeddedICE macrocell inside the target device contains two watchpoint units, each of which is capable of being programmed as either a breakpoint or watchpoint. This section describes how breakpoints are set in memory.

If you set a breakpoint in ROM, this will take up one of the breakpoint units in the EmbeddedICE macrocell. The address in ROM of the instruction to be breakpointed is programmed into the address value register of the macrocell by the EmbeddedICE interface. This macrocell unit will then halt instruction execution when the instruction at that address is fetched from ROM and reaches the execute stage of the instruction pipeline.

The process for instructions in RAM is more complex, but more powerful. Rather than use up one macrocell breakpoint unit for each address you want to set breakpoints on in RAM (as is the case for breakpoints in ROM), the instruction in RAM is overwritten by an unused bit pattern by the EmbeddedICE interface, and the overwritten value is stored away in the host debug platform for later replacing the unused pattern when the breakpoint is reached or deleted. Any number of breakpoints can be set in RAM using the one breakpoint unit because the same pattern is searched for all the time.

This makes the EmbeddedICE approach to debugging very powerful, and does not use up any target resources because the debugger memory is used as temporary storage of the overwritten instructions.

When an ARM instruction is breakpointed in memory, the pattern used is 0xdeeedeee, an unused bit pattern in the ARM instruction set. If you want to set breakpoints on Thumb instructions (16-bit instructions), the pattern used is 0xdeee, an unused bit pattern in the Thumb instruction set.

Design Requirement

As virtually any ARM system is capable of word memory accesses, the setting of ARM instruction breakpoints causes no problem. However, the 16-bit read/write operation required for Thumb instructions in RAM is actually performed using two-byte accesses, rather than a single half-word access. Therefore, your memory system has to be capable of both word and byte accesses, even if byte-wide memory is not available in your system.

Note *This is advisable for other reasons too. Byte access is an integral part of the ARM architecture, and the ARM tools will always assume that such accesses are possible, for example, for manipulation of packed structures, accessing character strings, and for debugger access to memory.*

Note *Multi-ICE release version 1.3 (and later) will use halfword instructions to set Thumb software breakpoints, although allowing byte access is still strongly recommended.*

12.2 Connecting to an ARM target board

This section describes the physical connection between the EmbeddedICE interface and an ARM target Board.

Note *If using Multi-ICE, refer to the Multi-ICE User Guide [ARM DUI 0048] for connector details.*

EmbeddedICE interface connector pinout

The interface connector on the target board (shown below) is a 14-way box header as shown in ***Figure 12-1: EmbeddedICE interface connector PL1 (viewed from above)***.

This plug is connected to the EmbeddedICE interface module using a short 14-way IDC cable with IDC sockets at each end.

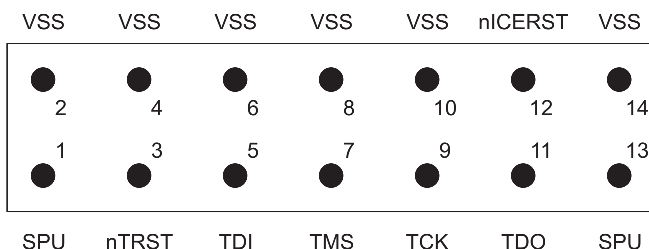

Figure 12-1: EmbeddedICE interface connector PL1 (viewed from above)

Table 12-1: EmbeddedICE interface connector PL1 summarizes the functions of the connector pins.

Pin	Name	Function
1	SPU	System powered up, pin connected to Vdd through a 33 ohm resistor
3	nTRST	Test reset, active low
5	TDI	Test data in
7	TMS	Test mode select
9	TCK	Test clock
11	TDO	Test data out
12	nICERST	Target System Reset (sometimes referred to nSYSRST or nRSTOUT)
13	SPU	System powered up, pin connected to Vdd through a 33 ohm resistor
2, 4, 6, 8, 10, 14	VSS	System ground reference (All VSS pins should be connected to minimize noise pickup)

Table 12-1: EmbeddedICE interface connector PL1

The functions of these signals are mainly self-evident, but there are a few which have special functions.

TDI, **TDO**, **TMS**, **TCK** and **nTRST** are the standard 5-wire JTAG interface signals.

nICERST is an unsupported function in the EmbeddedICE interface, which is intended to allow the target system to be reset from the debugger. Designers should include this signal, if possible, in any reset circuitry because other protocol convertors (for instance, Multi-ICE) can make use of this facility.

SPU is used by EmbeddedICE to monitor the target systems positive supply to ensure that the JTAG lines are driven and received by target-compatible logic.

EmbeddedICE interface driver/receiver circuitry

The actual driving/receiving circuitry inside the EmbeddedICE interface is detailed below to allow selection of suitable interface cells.

- Outputs from the EmbeddedICE interface

  The JTAG outputs (**TDI**, **TMS**, **TCK** and **nTRST**) are driven from 74HC14 Schmitt Trigger Inverter devices by way of 47-ohm serial resistors. The 74HC14 device is powered from a supply derived from the **SPU** signal from the target system to ensure that the drivers are voltage-compatible with the target system.

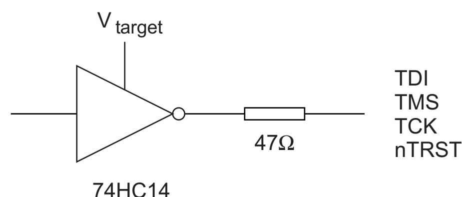

Figure 12-2: EmbeddedICE interface output circuitry

- Inputs to the EmbeddedICE interface

  The JTAG input (TDO) is terminated by a 47-ohm serial resistor, followed by a 10-kohm parallel, pull down resistor. This terminated input is fed into another 74HC14 device, again powered from the SPU signal to ensure voltage-level compatibility.

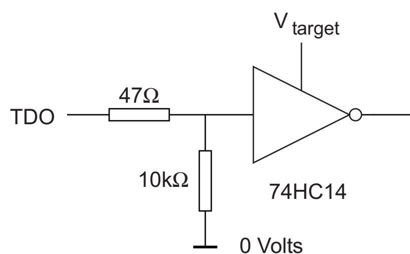

Figure 12-3: EmbeddedICE interface input circuitry

12.3 Reset and JTAG signal connection

When you design a target system, it is important to consider the connection of the JTAG and reset signals. The different approaches that may be used are described here.

nRESET is used to reset the processor core and put it into a known state, while **nTRST** is used to reset the TAP controller and the EmbeddedICE macrocell, including the registers in the breakpoint/watchpoint units. Both these resets must be applied before the device will function correctly.

The system must be designed so that it operates correctly, whether or not the EmbeddedICE interface is connected. It must also allow the EmbeddedICE interface to reset the JTAG TAP and, in some cases, the system itself.

The following sections describe two different approaches to the generation of the **nTRST** signal. The first approach is the simplest and provides a basic reset scheme, whereas the second approach is more sophisticated and allows a greater level of control by the debugger.

Basic nTRST connection

As the system must function correctly when the EmbeddedICE interface is not connected, the following must occur before the system can operate:

- **nRESET** must be held LOW for a number of clock cycles, before going HIGH and allowing the processor out of reset (see the relevant processor data sheet for details)
- **nTRST** must be held LOW to ensure the TAP is reset, and this can be done in either of the following ways:
  - by tying **nTRST** permanently LOW
  - by holding **nTRST** LOW, before going HIGH.

However, if **nTRST** is permanently tied LOW, debugging by way of the JTAG port cannot occur, so **nTRST** must be pulsed LOW. The simplest way to provide this is for **nTRST** to be connected directly to **nRESET**.

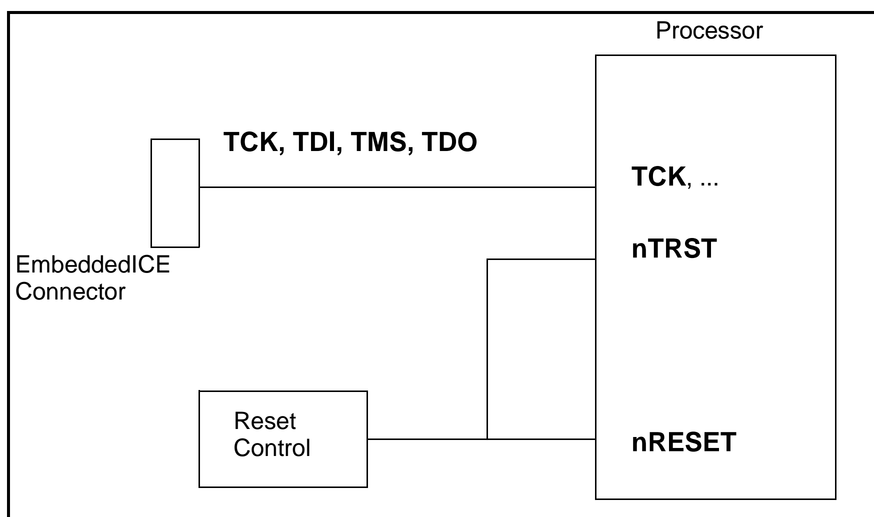

Figure 12-4: Basic nTRST connection

The EmbeddedICE interface is still able to reset the TAP because it is possible to reset the TAP using only a combination of the **TCK** and **TMS** signals.

The disadvantage of this approach is that it is not possible to set a breakpoint on the processor exiting from reset. This is because the TAP is always held in reset when the processor is in reset, and the breakpoint registers cannot be programmed while the TAP is in this state.

Separate control of nTRST and nRESET

In some situations, it is desirable to be able set a breakpoint on the reset vector, so that the target system is stopped when the processor is exiting from reset.

However, to do this, the breakpoint registers must be programmed prior to the processor exiting from reset, and therefore, it is necessary to have separate control of the **nTRST** and the **nRESET** signals.

Figure 12-5: nTRST and nRESET connection shows a suggested connection method. The reset control logic, which may be a simple PLD, is used to drive the resets, such that **nTRST** is pulsed LOW at initial power-up only, and **nRESET** is pulsed LOW both at power-up and whenever any other form of reset is asserted.

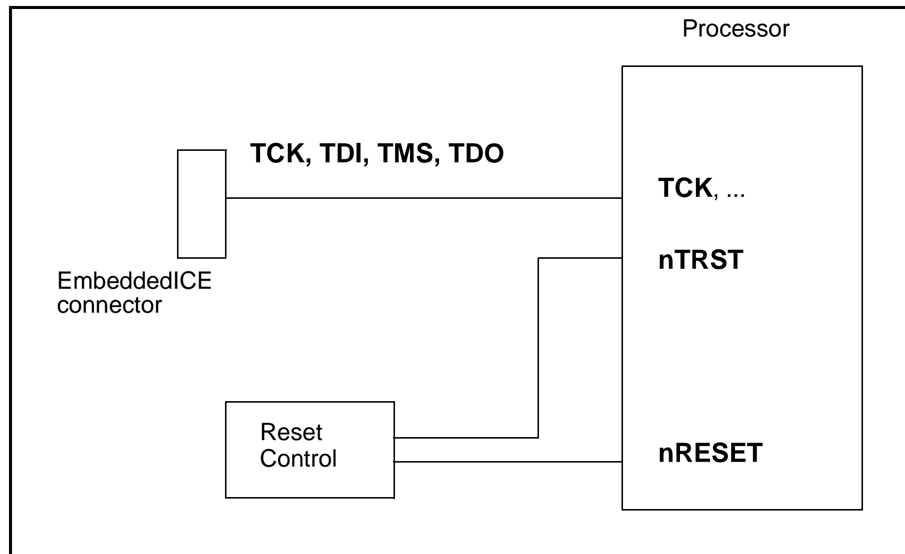

Figure 12-5: nTRST and nRESET connection

This allows the user to set a reset breakpoint by setting a breakpoint on address 0, and then pressing the **Reset** button. If the basic scheme has been used, as described in ***Basic nTRST connection***, pressing the **Reset** button also resets the breakpoint registers.

The EmbeddedICE connector also includes an **nICERST** signal that is asserted by the EmbeddedICE unit when it wishes to reset the target system, as shown in ***Figure 12-6: nTRST, nRESET and nICERST connection***.

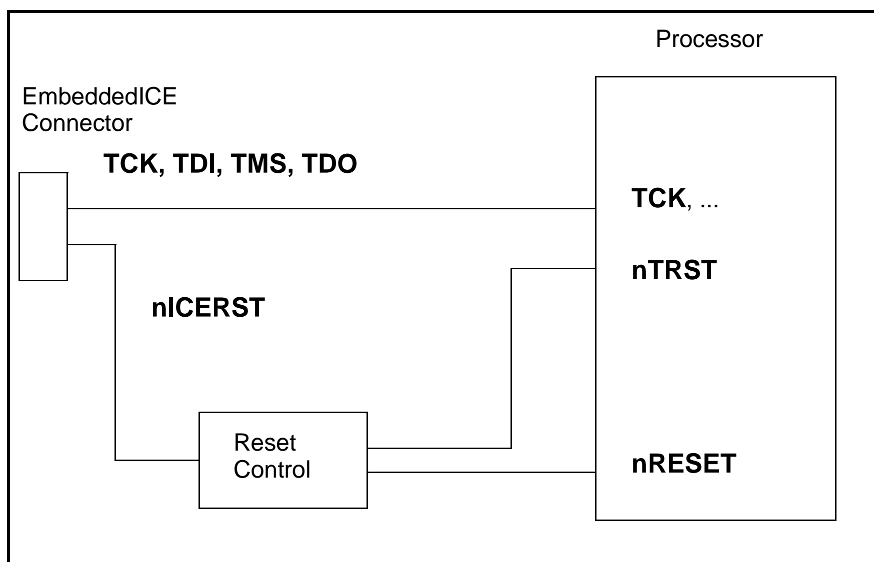

Figure 12-6: nTRST, nRESET and nICERST connection

Note

- In some EmbeddedICE documentation, the **nICERST** signal is called **nSYSRST**.
- The ability to reset the target system from the debugger is not possible when using an EmbeddedICE interface. However, it is possible in a suitably designed target system when using Multi-ICE. See the *Multi-ICE User Guide* [ARM DUI 0048] for details.

12.4 Other points to note

When the debugger starts up, it tries to stop the ARM core by programming the EmbeddedICE macrocell (through scan chain 2) to assert DBGRQ. There are a number of possible reasons why this might not succeed, and the debugger may report `Target Processor not stopped`, or behave erratically during a debug session.

1. DBGEN must be tied high for the core to be able to enter the debug state.
2. The debugger must be configured to use the correct processor. By default, the EmbeddedICE interface will assume an ARM7TDI (as opposed to an ARM7DI).
3. The clock to the core (MCLK) must be running (without infinite wait states).
4. The ARM must be a granted bus master.
5. MCLK must be greater than 100KHz when using an EmbeddedICE interface.
6. The core and macrocell must have been correctly reset (as previously detailed).