Hot-Debug for Intel XScale® Core Debug

White Paper

May 2005

Document Number: 273539-005

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel products are not intended for use in medical, life saving, life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel XScale® core may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Copyright© Intel Corporation, 2005

AlertVIEW, i960, AnyPoint, AppChoice, BoardWatch, BunnyPeople, CablePort, Celeron, Chips, Commerce Cart, CT Connect, CT Media, Dialogic, DM3, EtherExpress, ETOX, FlashFile, GatherRound, i386, i486, iCat, iCOMP, Insight960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel ChatPad, Intel Create&Share, Intel Dot.Station, Intel GigaBlade, Intel InBusiness, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Play, Intel Play logo, Intel Pocket Concert, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel TeamStation, Intel WebOutfitter, Intel Xeon, Intel XScale, Itanium, JobAnalyst, LANDesk, LanRover, MCS, MMX, MMX logo, NetPort, NetportExpress, Optimizer logo, OverDrive, Paragon, PC Dads, PC Parents, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, ProShare, RemoteExpress, Screamline, Shiva, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside, The Journey Inside, This Way In, TokenExpress, Trillium, Vivonic, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

The ARM and ARM Powered logo marks (the ARM marks) are trademarks of ARM, Ltd., and Intel uses these marks under license from ARM, Ltd.

Contents

1.0 Introduction 5
- 1.1 Reference Documents 5
- 1.2 Document Organization 5

2.0 Debug Functional Overview 6

3.0 Traditional Debug 8

4.0 Hot-Debug 11

5.0 Hot-Debug Implementation 14

6.0 Requirements and Restriction 18
- 6.1 Addressing 18
- 6.2 Application Code 22
- 6.3 Debugger 23

7.0 Conclusion 24

Figures

1 Debug Components 6
2 Traditional Xscale JTAG Connection Flow Chart 9
3 Code Download During a Cold Reset For Debug 10
4 Hot-Debug JTAG Connection Flow Chart 12
5 Addressing Example 19

Tables

1 Summary of Debugger, Debug Handler, and Application Code Functionality 7
2 DCSR.moe Encodings 14
3 Debugger and Debug Handler Actions During Download 15

Revision History

Date	Revision	Description
April 2005	005	Complete Revision.
December 2002	004	Corrected typographical error on page 11.
May 2002	003	Corrected code in Section 5.0.
October 2001	002	Updated Section 5.0, “Hot-Debug Implementation”. Revised first bullet in Section 6.0, “Requirements and Restriction”.
May 2001	001	Initial release.

1.0 Introduction

The purpose of this white paper is to explain the benefits of and the implementation requirements for Hot-Debug for Intel XScale® microarchitecture I/O processors. Traditionally, when a debugger is used through a Joint Test Action Group (JTAG) connection, the processor is reset at the beginning of the debug session. For standalone applications, this is acceptable, but I/O processors are mostly used for central processing unit (CPU) off load and add-in cards where there is a host system which configures all of the devices on the bus. The start of a debug session on an add-in card or a host bus adaptor causes a reset and consequently the bus configuration of the add-in card is lost. Therefore, a debugger capability to connect through JTAG without causing a reset to the device is required.

1.1 Reference Documents

- *Intel® 80200 Processor based on Intel XScale® Microarchitecture Developer's Manual* (Order Number 273411)—Intel Corporation
- *Intel XScale® Core Developer's Manual* (Order Number: 273473), Intel Corporation
- *ARM Architecture Reference Manual* - ARM Limited. Order number: ARM DDI 0100E.
- *Intel® 80231 I/O Processor Developer's Manual* (Order Number: 273517), Intel Corporation
- *Intel® 80331 I/O Processor Developer's Manual* (Order Number: 273942), Intel Corporation

1.2 Document Organization

The sections of this document are in a sequence to give the reader a thorough understanding of the concepts before detailing the requirements for implementation.

- Section 2.0, "Debug Functional Overview", gives a general overview of the functionality of Intel XScale® core debug through a JTAG connection. The interaction of the mini-instruction cache, the debugger, the debug handler, and the target application are described.
- Section 3.0, "Traditional Debug", provides a description of traditional Intel XScale® core debug.
- Section 4.0, "Hot-Debug", provides a description of Hot-Debug.
- Section 5.0, "Hot-Debug Implementation", details the requirements for a successful implementation of Hot-Debug. Code sections are included and explained. The location for downloading example code is also provided.
- Section 6.0, "Requirements and Restriction", provides details on aspects that will cause problems with Hot-Debug implementation.

2.0 Debug Functional Overview

This functional overview pertains to the following Intel® processors:

- 80200
- 80310 (80200 core)
- 80315 (80200 core)
- 80321
- 80219
- 80331
- 8033X
- IXP2400
- IXP2800
- IXP420
- IXP421
- IXP422
- IXP425
- PXA800F
- PXA800EF
- PXA26X
- PXA255

The components of software debug through a JTAG connection are illustrated in Figure 1. The debugger is a software application that runs on a separate computer and communicates with the target processor through a JTAG connection on the circuit board. For this family of Intel XScale® core processors, the debugger downloads a debug handler into the mini-instruction cache of the target processor. The debug handler is code that runs on the target processor and communicates with the debugger through a set of JTAG registers. Normally the target code for debug, the application code for the target processor is typically stored in the Flash of the target circuit board, but since Flash access is slower than RAM access, the application code normally decompresses itself into RAM and then runs from there. The mini-instruction cache is only used by the debug handler and cannot be accessed by the application code running on the circuit board. The debugger, the debug handler, and the application code have distinct roles that are illustrated in Table 1, "Summary of Debugger, Debug Handler, and Application Code Functionality" on page 7. For more information on software debug and test features, refer to the *Intel® 80200 Processor based on Intel XScale® Microarchitecture Developer's Manual*, Chapter 13 and appendix C respectively.

Figure 1. Debug Components

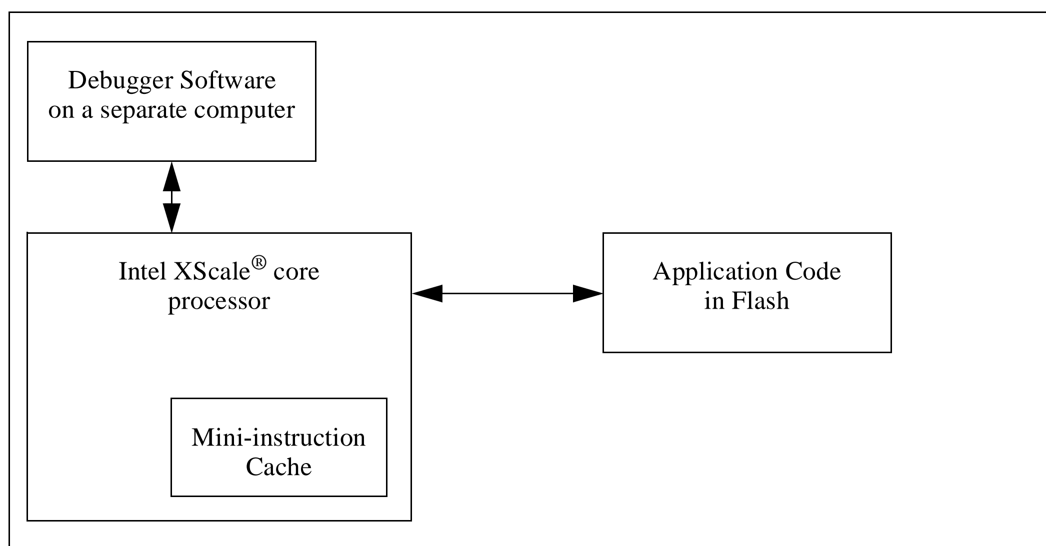


Table 1. Summary of Debugger, Debug Handler, and Application Code Functionality

Debugger	Debug Handler	Application
Located on a separate computer.	Located in the mini-instruction cache.	Normally resides in the Flash at start up. Can run out of memory after initialization.
Selects Halt or Monitor mode in the Debug Control and Status Register (DCSR).	Performs commands issued from the debugger.	Initializes the processor and circuit board.
Performs an external debug break to connect to the target processor.	The code in the debug handler uses exception traps and breakpoints to stop application code at desired locations.	The code that is being debugged.
Downloads the debug handler.		
Issues commands to the debug handler.		

3.0 Traditional Debug

When the processor is reset, it goes to address 0x0 to find the pointer to the beginning of the reset and initialization code. When halt mode is active, the processor uses the reset vector as the debug vector.

Addresses 0x0 through 0x1C contain the exception vector table. The debugger scans the vector table at address 0x0 in Flash and scans it into the first cache line of the mini-instruction cache which is also addressed to 0x0. The vector table in the mini-instruction cache is the override vector table and will be used by the processor even when there is another vector table at the same address in Flash or RAM. To keep the vector table in the mini-instruction cache current, the debugger must periodically scan the vector table in the application code for changes and scan any changes into the vector table in the mini-instruction cache.

At connection with a traditional debug session, the debugger holds the processor in reset until the debug handler is loaded into the mini-instruction cache. On releasing reset, the debugger takes initial control of the system. Figure 2 and Figure 3, "Code Download During a Cold Reset For Debug" on page 10 illustrate the traditional JTAG connection flow.

Figure 2. Traditional Xscale JTAG Connection Flow Chart

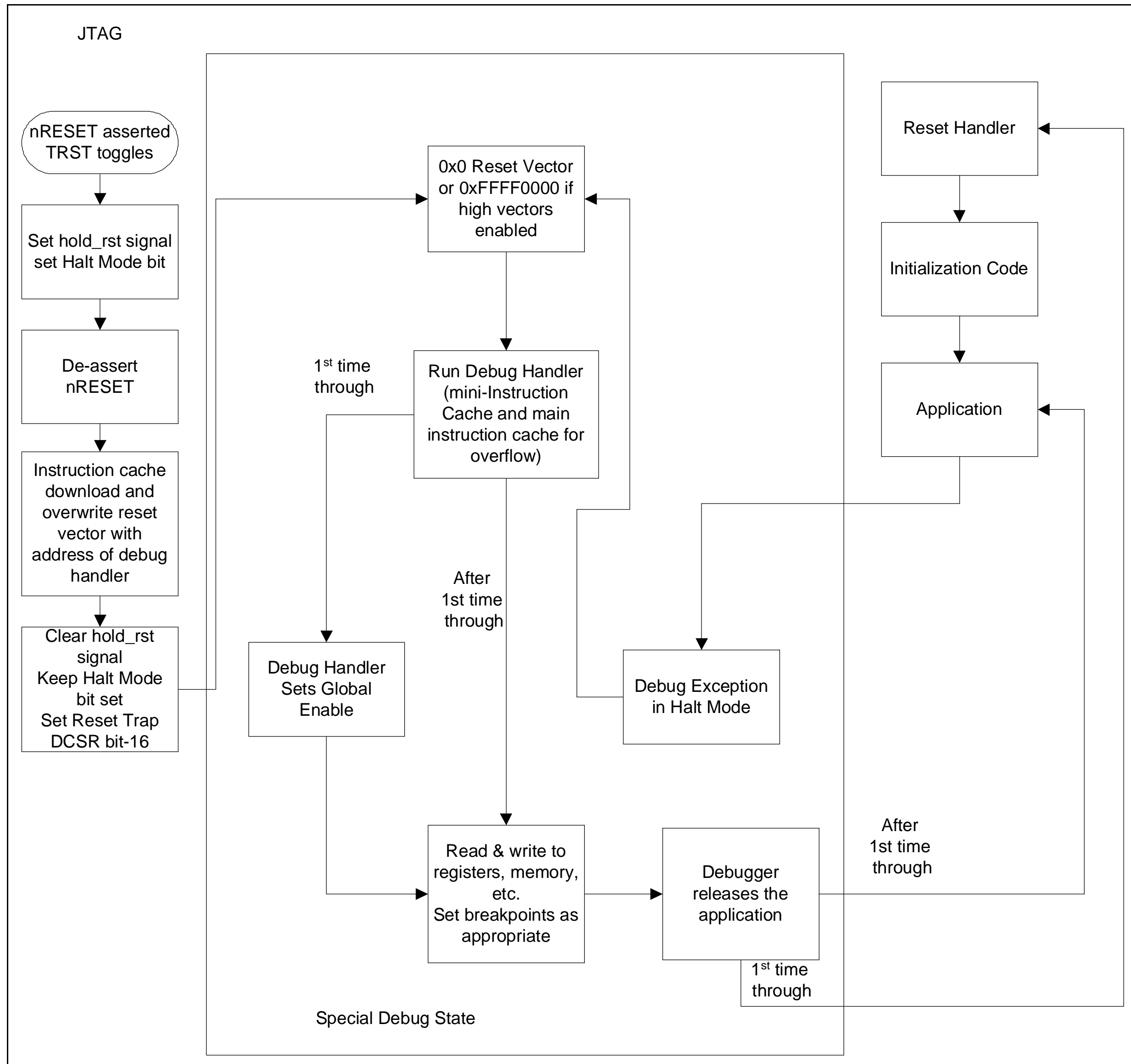
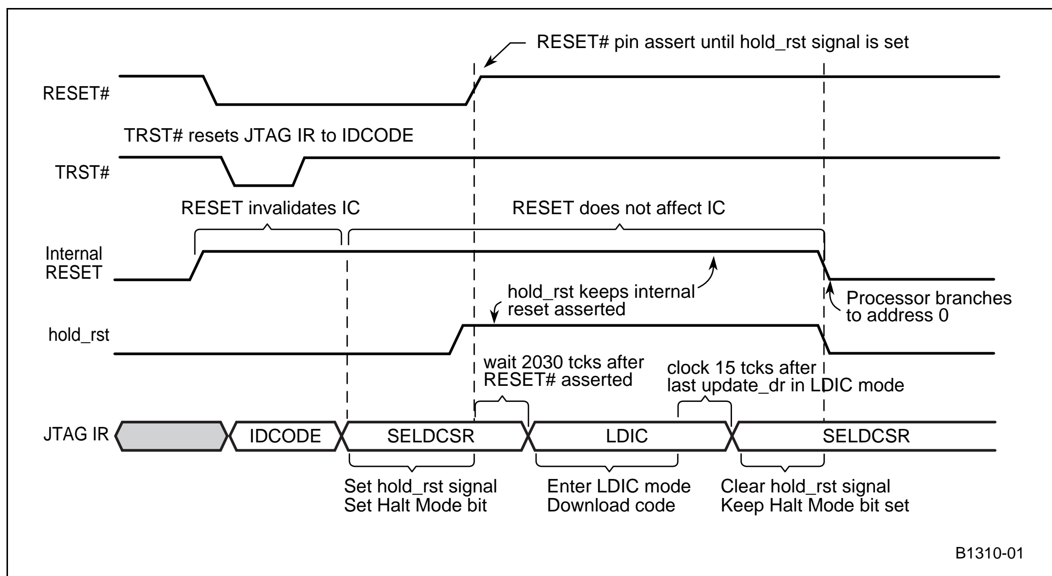


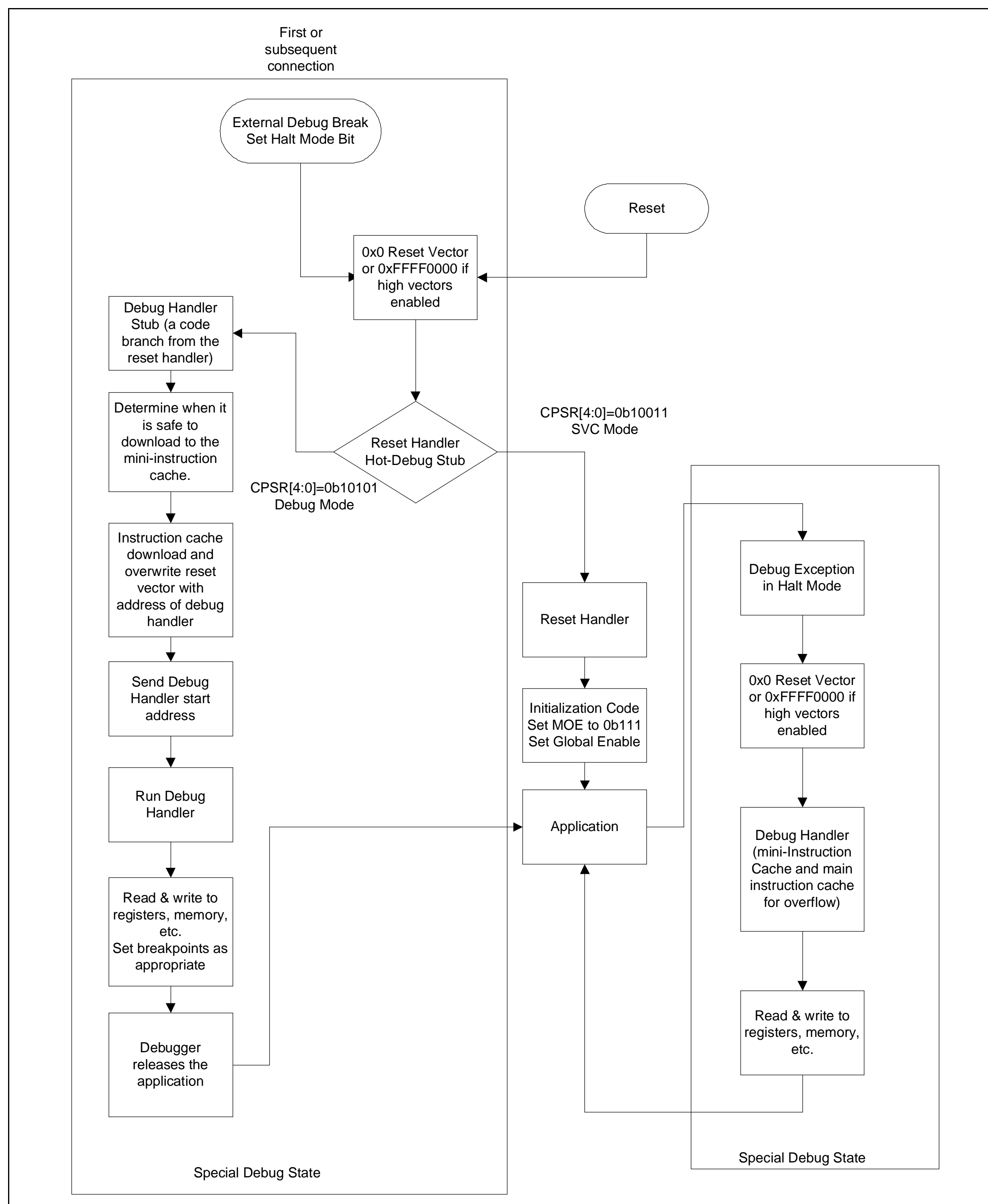
Figure 3. Code Download During a Cold Reset For Debug



4.0 Hot-Debug

Due to the Intel XScale® core debug architecture, a traditional debug session resets the processor when the debugger connects. Hot-Debug is a software solution that allows a debugger to connect without resetting the processor. To connect in Hot-Debug, additional code at the beginning of the reset code determines whether a reset occurred or a Hot-Debug session was initiated. When a reset occurred, the reset and initialization code continues. When a Hot-Debug session was initiated, the reset code does not run, the processor is polled for the appropriate point to download the debug handler into the mini-instruction cache and then control is redirected to the debug handler. Figure 4, "Hot-Debug JTAG Connection Flow Chart" on page 12 illustrates the Hot-Debug connection flow.

Figure 4. Hot-Debug JTAG Connection Flow Chart



When Hot-Debug is supported, the debugger uses an external debug break through JTAG to cause a debug exception in the core. The core redirects the execution to a debug handler stub that is linked into the application reset handler. The stub sends a message to the debugger through JTAG indicating when it is safe to download the debug handler code into the mini-instruction cache. After the debugger completes the download, it sends the starting address of the downloaded debug handler to the stub. The stub then branches to the beginning of the debug handler and the debug session begins.

When Hot-Debug is supported, the debugger can initiate a Hot-Debug session at any time provided that physical vector at address 0x00000000 points to the beginning of the reset code that contains the debug handler stub. The debugger can connect, disconnect, and reconnect multiple times when needed.

5.0 Hot-Debug Implementation

Hot-Debug code must be added to the application code. The code with comments is provided at the end of this section.

The application code determines whether the reset handler was entered due to a reset or a debug exception. When the processor is in Supervisor (SVC) mode where the Current Program Status Register bits 4:0 equal 0x13 (CPSR[4:0] =0x13), then a normal reset occurred. When the processor is in debug (DBG) mode (CPSR[4:0]=0x15), then a debug exception occurred. In this case, the reset handler branches to the linked debug handler stub, which allows the debugger to download the debug handler into the mini-instruction cache. Once the download is complete, the stub branches to the downloaded debug handler.

The application code must enable debug in the DCSR following every hardware reset. At reset, the processor clears the debug enable bit (DCSR[31]) and the Method of Entry (MOE) bits (DCSR[4:2]) in the DCSR. To support Hot-Debug, software must set the debug enable bit to enable debug exceptions and set the MOE bits with 111b. Table 2 details the meaning of the Method of Entry bits. Though these bits will normally change during the debug session, the debug handler must set them to 111 binary (Hot-Debug mode) on exit.

Table 2. DCSR.moe Encodings

Bit Encoding	Description
000b	Indicates Hot-Debug is not supported. The processor must be held in reset while downloading the debug handler into the mini-instruction cache.
001b - 110b	Indicates a monitor is actively running. Hot-Debug may or may not be supported. The debugger must wait for the debug monitor to exit to determine whether Hot-Debug is supported.
111b	Indicates Hot-Debug is supported, and a debug monitor is not actively running.

The debugger initiates a debug session with an external debug break via the DCSR JTAG data register. (For details, refer to the *Intel XScale® Core Developer's Manual*, Chapter 9, Section 9.11.) The processor redirects execution to the debug/reset vector and enters the reset handler. At this point, the reset handler checks the mode in the CPSR and detects that the processor is in DBG mode, and branches to the debug handler stub. Table 3 shows the actions taken by the debug handler stub and the debugger to ensure that the debug handler is correctly downloaded into the mini-instruction cache. Subsequent debug breaks during the current debug session are intercepted by the debug/reset vector. The processor then branches directly to the downloaded debug handler. When there are subsequent debug sessions, the debug vector and the debug handler should be downloaded every time at the beginning of the new debug session.

The external debug break should be generated as follows:

1. Scan a value into the DCSR JTAG data register to set the Halt mode bit.
2. Scan a value into the DCSR JTAG data register to set the external debug break bit (and keeping the Halt mode bit set).

Refer to the *Intel XScale® Core Developer's Manual*, Chapter 9, Section 9.11.

Table 3. Debugger and Debug Handler Actions During Download

<table>
<tr><th>Debug Handler Stub Actions</th><th>Debugger Actions</th></tr>
<tr><td>Executes synchronization routine to ensure all outstanding instruction fetches have completed. Invalidate the BTB.</td><td rowspan="2">Polls the DBG_TX JTAG data register for a 'ready-for-download' message indicating it is safe to begin the download.</td></tr>
<tr><td>Following sync routine, writes the 'ready-for-download' message to the TX register. 'ready for download' = 0x00B00000</td></tr>
<tr><td rowspan="3">Spins in a loop, polling RX register, waiting for debugger to indicate the download is complete.</td><td>Detects the write to TX and reads the value through the DBG_TX JTAG data register. Sees the 'ready-for-download' message and begins the download.</td></tr>
<tr><td>Downloads vector table and debug handler code into mini-instruction cache through the LDIC JTAG data register1. After completion of the download, waits ~50 TCKs w/ LDIC JTAG instruction in the JTAG IR before continuing.</td></tr>
<tr><td>Writes the debug handler start address to the DBG_RX JTAG data register. This signals the download is complete and also provides the debug handler stub with the start address of for the full debug handler.</td></tr>
<tr><td>Detects valid data in RX and comes out of its polling loop. Reads start address from RX and branches to the debug handler.</td><td>Polls DGB_TX for debug handler entry message.</td></tr>
</table>

NOTE: Each cache line written to must first be invalidated through JTAG.

The following code must be added to the application code:

```
reset_handler_start:
## reset handler should first check whether this is a debug exception
## or a real RESET event.
## NOTE: r13 is only safe register to use.
## - For RESET, don't really care about which register is used
## - For debug exception, r13=DBG_r13, prevents application registers
## - from being corrupted, before debug handler can save.
mrs r13, cpsr
and r13, r13, #0x1f
cmp r13, #0x15 # are we in DBG mode?
beq dbg_handler_stub # if so, go to the dbg handler stub
mov r13, #0x8000001c # otherwise, enable debug, set MOE bits
mcr p14, 0, r13, c10, c0, 0 # and continue with the reset handler
## normal reset handler initialization follows code here,
## or branch to the reset handler.
.align 5 ## align code to a cache line boundary.
dbg_handler_stub:
## First save the state of the IC enable/disable bit in DBG_LR[0].
mrc p15, 0, r13, c1, c0, 0
and r13, r13, #0x1000
orr r14, r14, r13, lsr #12
## Next, enable the IC.
mrc p15, 0, r13, c1, c0, 0
orr r13, r13, #0x1000
mcr p15, 0, r13, c1, c0, 0
## do a sync operation to ensure all outstanding instr fetches have
## completed before continuing. The invalidate cache line function
## serves as a synchronization operation, that's why it is used
## here. The target line is some scratch address in memory.
adr r13, line2
mcr p15, 0, r13, c7, c5, 1
## invalidate BTB. make sure downloaded vector table does not hit one of
## the application's branches cached in the BTB, branch to the wrong place
mcr p15, 0, r13, c7, c5, 6
## Now, send 'ready for download' message to debugger, indicating debugger
## can begin the download. 'ready for download' = 0x00B00000.
TXloop:
mrc p14, 0, r15, c14, c0, 0 # first make sure TX reg. is available
bvs TXloop
mov r13, #0x00B00000
mcr p14, 0, r13, c8, c0, 0 # now write to TX
## Wait for debugger to indicate that the download is complete.
RXloop:
mrc p14, 0, r15, c14, c0, 0 # spin in loop waiting for data from the
bpl RXloop # debugger in RX.
## before reading the RX register to get the address to branch to, restore
## the state of the IC (saved in DBG_r14[0]) to the value it have at the
## start of the debug handler stub. Also, note it must be restored before
```

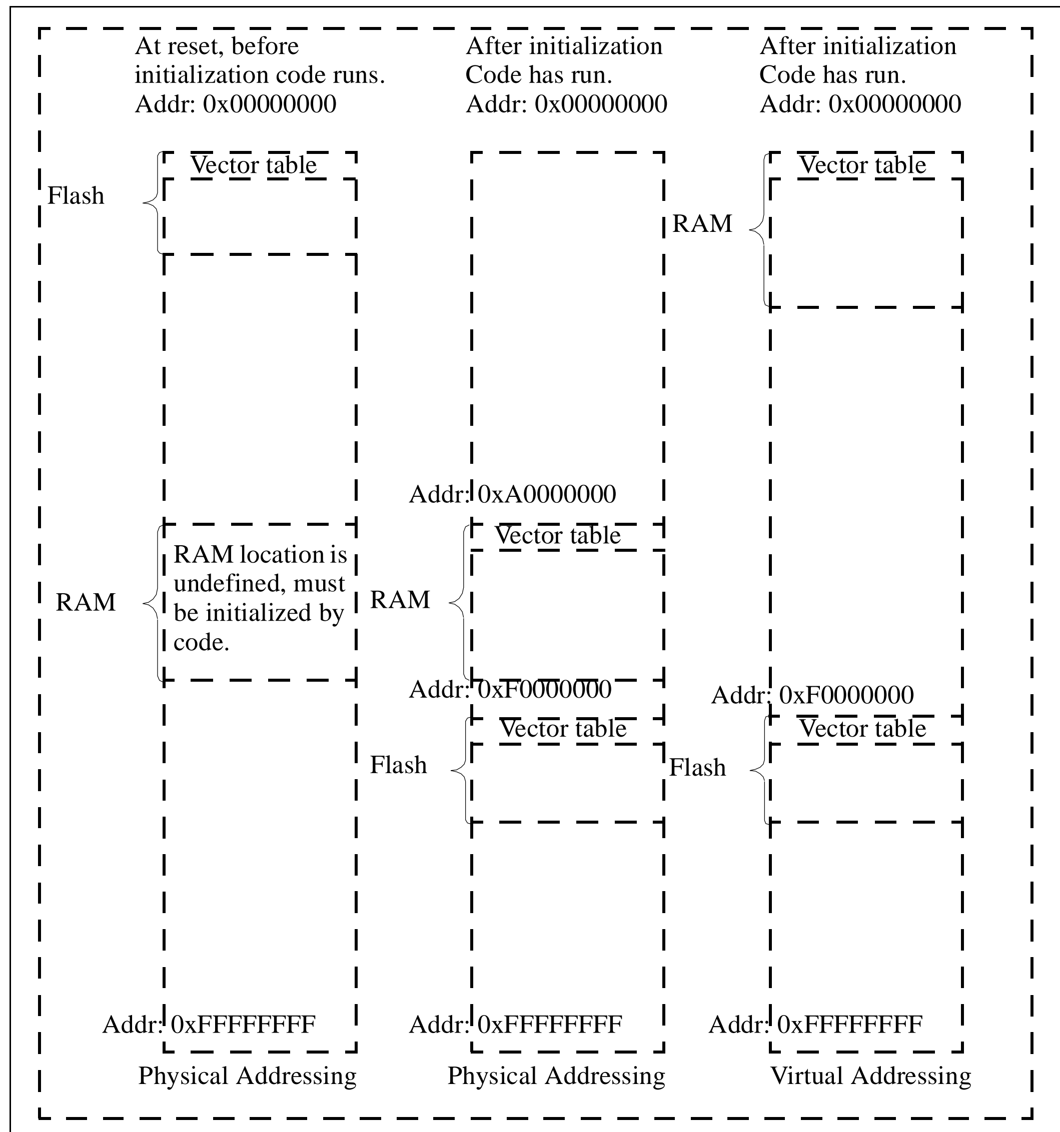
```
## reading the RX register because of limited scratch registers (r13)
mrc p15, 0, r13, c1, c0, 0
### First, check DBG_LR[0] to see if the IC was enabled or disabled
tst r14, #0x1
### Then, if it was previously disabled, then disable it now, otherwise,
### there’s no need to change the state, because its already enabled.
biceq r13, r13, #0x1000
mcr p15, 0, r13, c1, c0, 0
## Restore the link register value
bic r14, r14, #0x1
## Now r13 can be used to read RX and get the target address to branch to.
mrc p14, 0, r13, c9, c0, 0 # Read RX and
mov pc, r13 # branch to downloaded address.
## scratch memory space used by the invalidate IC line function above.
.align 5 # make sure it starts at a cache line
# boundary, so nothing else is affected
line2:
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
```

6.0 Requirements and Restriction

6.1 Addressing

The Hot-Debug code must be placed in one-to-one virtual to physical memory space as defined by the memory management unit (MMU) descriptor tables. When Hot-Debug is entered, the processor enters Special Debug State (SDS) and the instruction memory management unit is disabled. This turns off instruction virtual addressing. When addressing for virtual and physical instructions is not the same, then the instruction pointer will be pointing to an incorrect location as soon as SDS is entered. Many Intel XScale® microarchitecture applications relocate the startup code in Flash to a different physical address and RAM is virtually addressed to address 0x0 as illustrated in Figure 5. To make this work in Hot-Debug, place the Hot-Debug code in Flash and have the moved Flash located in one-to-one virtual to physical memory space.

Figure 5. Addressing Example



When the processor enters Special Debug State and virtual addressing for instructions is disabled, the vector table must be located at physical address 0x0 (or 0xffff0000 when high vectors are enabled) when the table is located in Flash or RAM, or the vector table must be located at virtual address 0x0 (or 0xffff0000 when high vectors are enabled) when the table is locked in cache. When RAM is virtually addressed to 0x0, then the vector table at virtual address 0x0 must be loaded into the instruction cache by the application before the Hot-Debug JTAG connection is made. The cache line at address 0x0 must be loaded and locked into the main instruction cache. When the Hot-Debug code is relocated in RAM then that code must also be lock in cache.

Code to lock the vector table into the main instruction cache at address 0x0 should be similar to the following:

```
@ clean, drain, flush the main Dcache

ldr      r1, =DCACHE_FLUSH_AREA   @ use a CACHEABLE area of memory
mov      r0, #1024                @ number of lines in the Dcache
Loop_1:
mcr      p15, 0, r1, c7, c2, 5    @ allocate a Dcache line
add      r1, r1, #32              @ increment to the next cache line
subs     r0, r0, #1               @ decrement the loop count
bne      Loop_1

@ Prepare to lock vector table into icache line.
@ The icache lock function requires that instruction caching be disabled.
@ When the MMU is on, this is accomplished by setting the descriptor C bit to 0

mrc      p15, 0, r8, c2, c0, 0    @ Translation Table Base address
ldr      r10, =0x0FFFFFFF         @
and      r8, r8, r10              @ Convert to virtual
mov      r9, pc, lsr #20          @ get current address
add      r8, r8, r9               @ pointer to current memory descriptor
ldr      r9, [r8]                 @ get descriptor and preserve value
ldr      r10, =0xFFFFFFF7         @ Mask C bit to disable instruction cache.
and      r10, r9, r10
str      r10, [r8]
mov      r1, #0
mcr      p15, 0, r0, c7, c10, 4   @ drain the write and fill buffers
mcr      p15, 0, r0, c7, c7, 0    @ invalidate Icache, Dcache and BTB
mcr      p15, 0, r0, c8, c7, 0    @ invalidate instruction and data TLBs
mcr      p15, 0, r0, c7, c10, 4   @ drain the write and fill buffers
CPWAIT   r1
mcr      p15, 0, r0, c9, c1, 0;   @ Lock Vector table into icache
CPWAIT   r1
str      r9, [r8]                 @ restore descriptor table value
mcr      p15, 0, r8, c7, c10,1    @ clean Dcache so that descriptor is in RAM
mcr      p15, 0, r0, c7, c10,4    @ drain the write and fill buffers
mcr      p15, 0, r0, c7, c7, 0    @ invalidate Icache, Dcache and BTB
mcr      p15, 0, r0, c8, c7, 0    @ invalidate instruction and data TLBs
CPWAIT   r1
```

When the debugger connects to the target in halt mode and halts application program execution, the processor enters Special Debug State and it remains in Special Debug State until the debugger lets the application program run again. When the application is running to the next breakpoint, to a halt, or to the next instruction in a step, it is not in Special Debug State and virtual addressing for data and instructions is valid. When the breakpoint occurs, the processor interrupts the application code again and the processor re-enters Special Debug State.

In Special Debug State, all events (interrupts, aborts, etc.) are disabled and instruction virtual addressing is turned off. When the processor returns to the application, virtual addressing is valid again and any pending exceptions (interrupts, aborts, etc.) are also valid and the application code will need to service them. Special Debug State is described in detail in Section 13.5.1 of the *Intel® 80200 Processor based on Intel XScale® Microarchitecture Developer's Manual* or in Section 9.5.1 of the *Intel XScale® Core Developer's Manual*.

6.2 Application Code

The following restrictions apply to the Hot-Debug code that is added to the application code and must be met to ensure proper Hot-Debug functionality:

The code must be in a cacheable region of memory. The debug handler stub will temporarily enable the instruction cache allowing the polling loops used to communicate with the debugger to get cached. When the code is non-cacheable, the polling loop is small enough such that it can execute out of the fill buffers; however, since these same fill buffers are used when loading code into the mini-instruction cache through JTAG, there is a conflict for these resources.

Use of R13 in DBG mode is very restrictive and must be handled carefully. The code provided above has been tested for correct behavior. Any changes to the code may cause improper Hot-Debug behavior.

When the vector table is in RAM or Flash and it is virtually addressed, the application code must lock the vector table into the main instruction cache prior to the first Hot-Debug connection. Before locking, the application code must do a global invalidate to ensure that the target address is not already resident in the instruction cache. When the vector table is locked into cache, the address of the vector table will be the same whether the processor is in Special Debug State or not.

6.3 Debugger

The following restrictions apply to the debugger and must be met to ensure proper Hot-Debug functionality:

When the debug handler is downloaded into the mini-instruction cache, each line must first be invalidated through JTAG. Then the code for that cache line can be downloaded. This ensures that an address is not valid in multiple places in the instruction cache (main or mini).

After the complete debug handler and vector table is downloaded into the mini-instruction cache, the debugger must wait 50 TCKs before removing the LDIC JTAG instruction from the JTAG IR. This ensures that the last line of the debug handler is correctly updated in the mini-instruction cache. Refer to the *Intel XScale® Core Developer's Manual*, Chapter 9.

The debugger must leave the system in the proper state before exiting. Before ending the debug session, the debugger must do the following:

Clear the Halt Mode bit. The problem occurs when the processor is reset and there is no debug session in progress. The reset clears the global debug enable bit in the DCSR. However, when the Halt

Mode bit is set, application software cannot write to the DCSR (unless the processor is in Special Debug State). So the global debug enable bit does not really get set by the reset handler, and the debugger is then not able to use the external debug break to initiate a debug session.

Invalidate the Branch Target Buffer (BTB). This can only be done via the debug handler. Invalidating the BTB ensures that the BTB does not contain cached branch targets that were there for debug but now are no longer valid. This mainly applies to the vector table since this is probably the only debug code that overlaps the applications code.

Do not invalidate the mini-instruction cache by line or in entirety through JTAG. Invalidating the mini-instruction cache also invalidates the instruction fill buffers and when done when the processor attempts to refill the buffers it incorrectly uses the invalidated address instead of the current instruction address which interferes with the target application that is running. It is not necessary to invalidate the debug handler code because this is placed in a location where the application does not have any code. Even through the debugger leaves the handler there, it should not rely on it remaining intact. On the next Hot-Debug connection, the debug handler should be downloaded again.

Set the DCSR.moe bits to 111b. This allows a debugger to detect that Hot-Debug is supported when attempting a subsequent debug sessions.

At the end of the debug session, the debug handler code must invalidate address 0x0 and 0xffff0000 using the MCR commands. Address 0x0 and 0xffff0000 are the two locations where the processor looks for the vector tables.

7.0 Conclusion

Hot-Debug enables debugging in configurations where a system reset is not appropriate and it is achieved entirely with software.