Application Note
0380-0245-10 Rev 1.2

Embedded Performance, Inc.
606 Valley Way, Milpitas, CA 95035
Telephone: (408) 957-0350
FAX: (408) 957-0307
e-mail: support@epitools.com
web: www.epitools.com

Using A MAJIC® Probe With The Intel XScale® Microarchitecture

0380-0245-10 Rev 1.2

April 21, 2004

Introduction

The details of the Intel XScale® Debug Support Unit (DSU) raise some special configuration considerations. Many of these issues are managed automatically by the MAJIC probe, but some of the settings must be specified by the user to fit within your hardware and software environment. This application note describes

- The special configuration settings when using a MAJIC® Probe with a processor based on Intel XScale® technology.
- Target initialization files.
- MMU Utilities.
- Execution trace details specific to the Intel XScale® technology.

Additional Documentation

Additional documentation for the EDT software package and MAJIC® probe is installed as part of the EDT software package. On a Windows PC, use the **`[Start]/Programs...`** menu to browse into the **`EPI Tools/EDT`** folder and open **`EDT Documentation Index`**. On a Linux PC view **`./manuals/edtX_doc_index.html`** in your EDT installation.

MAJIC® User's Manual Complete information on configuration and operation of the MAJIC® probe, and the MON command language.

MAJIC® Windows CE .NET eXDI User's Manual
Details on using the eXDI driver for Microsoft Platform Builder, and the optional Platform Builder Plug-In.

Using A MAJIC® Probe with Intel® SDT
Application note providing details on setting up and using XDB-MAJIC with the Intel® C++ Software Development Tool Suite.

Using a MAJIC® Probe With A Linux PC
Using a MAJIC® Probe in a Cygwin Environment
Application notes providing details on setting up and using GDB with a MAJIC® Probe.

Using the EPI Flash Programming Utility
Explains how to program the flash memory on your target board using a MAJIC® probe and the flash utility included in the EDT software package.

MAJIC® Interface Specifications Application note providing details on the debug connector options for standard and small form factor connectors, and the electrical specifications of different MAJIC® Probe models.

Getting Support

Please do not hesitate to contact our technical support group if you have any questions or need assistance in configuring the MAJIC® probe for your system. We recognize that these issues are complex, and are committed to making sure our tools work well for you.

MAJIC® Configuration for Intel XScale® Cores

Chapter 3 of the *MAJIC User's Manual* describes the basic MAJIC® configuration process—this application note is an addendum to that chapter. The special configuration commands described herein should be placed in a custom initialization file, which you can then select with the MAJIC Setup Wizard. See *Advanced MAJIC Configuration* in the *MAJIC® User's Manual* for additional information on this procedure.

Note: The EDTA Software Package that comes with the MAJIC® probe includes board initialization files for several standard reference platforms (board plus OS). If you are using one of the standard reference platforms or something similar, then you may simply select the appropriate start up files with the MAJIC Setup Wizard instead of creating your own.

Debug Mode

When the program hits a breakpoint or completes a single step, the processor itself does not actually halt, as many other processor architectures do. Instead, the processor takes a *Debug Exception* and enters *Debug Mode*. The debug exception vector then transfers control to the debug exception handler, which the MAJIC® probe will have preset in a special *Debug Memory* space within the processor. This debug handler provides the interface between the MAJIC probe and the processor.

Note: When the Intel XScale® core enters sleep mode, its JTAG interface and debug memory are depowered. Therefore, it is not possible to continue a debug session once the processor enters sleep mode (even after it wakes up again).

Reset Management

When a MAJIC® probe connects to the target processor, it needs to load the debug exception handler into the debug memory before it can take control of the processor. Normally it does this by performing a sequence of JTAG operations to initialize the debug memory while holding the target processor in reset state. Then the MAJIC® probe releases the processor from reset, the CPU immediately enters debug mode, and begins executing the debug exception handler.

Note: In order for this to work, the target board must allow the MAJIC® probe to independently control the JTAG reset (nTRST) and system reset (nRESET) signals. Please see the *MAJIC® Interface Specifications* application note for more information on this and other hardware considerations.

If your board does not provide independent JTAG reset (nTRST) and system reset (nRESET) signals, then the only way for a MAJIC® probe to take control of the processor is with Intel's *Hot Debug* driver. The hot debug technique is also useful if you want to connect a MAJIC® probe to a running target system without resetting it, for in-service testing or diagnostics.

The driver, which is available from Intel, must be linked into and called from your boot code, and you must choose the Non-Intrusive connection method in the MAJIC Setup Wizard. For more information on integrating the hot debug driver into your build, please refer to Intel's white paper on this topic, which is available at

http://developer.intel.com/design/iio/applnots/27353904.pdf

Watchdog Timers

Embedded systems often utilize a watchdog timer to reset the system in the event that it stops operating normally. In order to prevent being reset, the software running on the system must periodically access the watchdog timer to refresh its counter. Since reset cycles initiated by the target system interfere with the debug environment, it is best to avoid using the watchdog timer while debugging.

However, if you do wish to leave the watchdog timer active while debugging, you can configure the MAJIC® probe to assume responsibility for refreshing the watchdog timer while stopped in debug mode. This feature is not specific to the Intel XScale® microarchitecture, and so is outside of the scope of this application note. For further information on this topic, and examples its usage, please read the comments in the `bin/cmd_desc.cmd` command file in your EDTA software installation.

Intercepting the Debug Exception

The processor reuses the reset vector for the debug exception vector. When using a MAJIC® probe, therefore, that shared vector must be modified to branch to the debug exception handler instead of the reset boot code, so that debug exceptions will invoke the debug handler rather than reboot the system. Unfortunately, while the fundamental concept is straightforward, there are a number of details that make this more complex than it first appears. The next sections of this application note discuss the issues inherent in intercepting exceptions, and how the MAJIC® probe manages this requirement.

The first and most obvious problem is how to replace the instruction at the reset vector with one that jumps to the debug exception vector. When the system is first powered up, some form of non-volatile read-only memory must be located at 0 because that is where the processor will fetch the first instruction. This means that the MAJIC® probe cannot simply overwrite the vector location.

Usually the MAJIC® probe overlays a 32 byte block of debug memory over the vector table area so that it can change the contents of the reset/debug vector, and thereby intercept the debug exception. By default, the MAJIC® probe initializes this *Exception Interceptor* block from the corresponding physical memory before starting program execution, and only changes the shared reset/debug vector, so the MAJIC® probe does not interfere with other exceptions.

However, there are some side effects of this technique that require consideration. In particular, since debug memory only responds to instructions fetches, none of the vectors in the exception interceptor can be directly modified by the program under test. This is because all loads and stores access target memory, as usual, but instruction fetches within the exception interceptor are serviced by debug memory instead. Furthermore, if the address translation of the vector area is changed to a different physical memory range, then the contents of the exception interceptor will not be consistent with the actual memory.

The following sections explain how the MAJIC® probe supports applications that change their vectors by writing there, by relocating the vectors to their high location, or by remapping the vector area to a different area of physical memory.

Run-Time Vector Initialization

The instructions in the debug memory, and the exception interceptor in particular, cannot be modified by software running on the target processor. The only means of writing to debug memory is via the JTAG interface. If the vector area is in RAM then the processor can read and write target memory at those locations, but the processor will fetch instructions from the exception interceptor, not the target memory. This means that the program under test cannot change the exception vectors by directly writing to them.

If your application requires changing exception vectors at run-time, EPI recommends using the following procedure instead of changing the instruction opcodes within actual vector area.

```
            IMPORT MyVectorTable
00000000    LDR    pc, MyVectorTable     //NEVER MODIFY
00000004    LDR    pc, MyVectorTable+4   //NEVER MODIFY
00000008    LDR    pc, MyVectorTable+8   //NEVER MODIFY
0000000C    LDR    pc, MyVectorTable+12  //NEVER MODIFY
00000010    LDR    pc, MyVectorTable+16  //NEVER MODIFY
00000014    LDR    pc, MyVectorTable+20  //NEVER MODIFY
00000018    LDR    pc, MyVectorTable+24  //NEVER MODIFY
0000001C    LDR    pc, MyVectorTable+28  //NEVER MODIFY
```

To change an exception vector, simply set the appropriate word in the `MyVectorTable` array (a buffer of 8 words allocated somewhere within your program) to the address of the exception handler's entry point, and the actual vector opcode need not change. This example shows the normal exception vector area, but the same technique can be used with the upper vector area as well.

If your application only needs to initialize the vector area once prior to taking any exceptions, then you may simply configure the MAJIC® probe to initialize the exception interceptor(s) to user defined values instead of from target memory. That way, you can declare what the vectors will eventually be initialized to before you execute the boot code, so that when exceptions start happening the intended vectors will be seen. User defined vectors are selected and initialized with configuration options that are explained in *Exception Interceptor Settings*, below.

However, if your system architecture requires that you modify the vector table dynamically, then please see *Dynamic Vector Management*, below.

Relocating the Vector Area

The vector table in the Intel XScale® microarchitecture starts at address zero, but many operating systems relocate the vector table to an alternate address range in high memory. Therefore, there are actually two exception interceptor areas, each of which may be independently intercepted (or not) depending on how your operating system manages exceptions while booting and during normal operation. The exception interceptors are controlled with configuration options that are explained in *Exception Interceptor Settings*, below.

Remapping the Vector Area

If your application changes the memory mapping of the vector areas via the MMU, and thereby changes the vector area's contents, then the MAJIC® probe cannot simply initialize the exception interceptor from memory prior to starting execution. Despite the remapping, the vector area will still be overlaid by the interceptor, so the original vectors are seen by the processor instead of your intended vectors.

Providing that you don't expect to take any exceptions prior to remapping the table, then you may simply configure the MAJIC® probe to initialize the exception interceptor(s) to user defined values instead of from target memory. That way, you can declare what the vectors will eventually be initialized to before you execute the boot code, so that when exceptions start happening the intended vectors will be seen. User defined vectors are selected and initialized with configuration options that are explained in *Exception Interceptor Settings*, below.

However, if your systems needs to change the vector area mapping dynamically, then please see *Dynamic Vector Management*, below.

Exception Interceptor Settings

The **`vector_load_low`** and **`vector_load_high`** options control how the MAJIC® probe manages each of the vector areas. If either of these options is set to **`target`**, then that interceptor will be automatically initialized from target memory at the corresponding physical address range before the MAJIC® probe (re)starts program execution. If either of these options is set to **`user`**, then the corresponding interceptor will be initialized with user defined instructions before the MAJIC® probe (re)starts program execution. If an option is **`off`**, then that interceptor is disabled, and instruction fetches in that area are serviced by the target memory system.

The default settings for these options are **`vector_load_low = target`** and **`vector_load_high = off`**. To change how the MAJIC® probe manages the vector areas, add commands such as the following examples to your user initialization file:

```
eo vector_load_low = target  // load low interceptor from target
eo vector_load_high = user   // load high interceptor from user
```

Note: If a debug exception is raised, and the currently active vector area is not intercepted by the MAJIC® probe, then it is the user's responsibility to ensure that the debug exception vector points to the debug kernel entry point. Otherwise, the MAJIC® probe will not be able to service the debug exception. EPI does not recommend disabling vector interception for the vector area(s) that are used in your system.

Dynamic Vector Management

If your system architecture requires dynamic changes to the vector area(s), either by directly modifying the vector location(s) or by remapping the vector area in the MMU, then the MAJIC® probe needs to know when such changes take place so that it can update the exception interceptor. This is accomplished by momentarily pausing program execution so that the MAJIC® probe can reinitialize the exception interceptor(s), by embedding a special instruction in your code, after the line(s) which modify the vector area or mapping but before the next exception occurs.

```
//  Modify vector area contents or mapping here
//
    BKPT   0x1234    // MAJIC pauses for vector reinitialization
//
//  Changes are now in effect
```

When no probe is connected, this instruction is executed as a NOP and is therefore completely benign. However, when a MAJIC® probe is connected, it is executed as a special software breakpoint. Execution stops, then the MAJIC® probe reinitializes the exception vector area(s) whose vector load option is set to **`target`**, then program execution is automatically restarted from the instruction following this **`BKPT`**.

User Defined Vectors

If you set the **`vector_load_low`** or **`vector_load_high`** option(s) to **`user`**, then you must set the desired values for the corresponding vectors in your custom initialization file, as shown in the following example, filling in the blanks with the opcode appropriate for each vector:

```
//  Define user vectors for low area
//  (used if vector_load_low = user)
ew MAJIC_VLOW_U = 0x________       // UNDEF : 0x00000004
ew MAJIC_VLOW_S = 0x________       // SWI   : 0x00000008
ew MAJIC_VLOW_P = 0x________       // PREF  : 0x0000000C
ew MAJIC_VLOW_A = 0x________       // ABORT : 0x00000010
ew MAJIC_VLOW_R = 0x________       // RSVD  : 0x00000014
ew MAJIC_VLOW_I = 0x________       // INTR  : 0x00000018
ew MAJIC_VLOW_F = 0x________       // FIQ   : 0x0000001C
//
//  Define user vectors for high area
//  (used if vector_load_high = user)
ew MAJIC_VHIGH_U = 0x________      // UNDEF : 0xFFFF0004
ew MAJIC_VHIGH_S = 0x________      // SWI   : 0xFFFF0008
ew MAJIC_VHIGH_P = 0x________      // PREF  : 0xFFFF000C
ew MAJIC_VHIGH_A = 0x________      // ABORT : 0xFFFF0010
ew MAJIC_VHIGH_R = 0x________      // RSVD  : 0xFFFF0014
ew MAJIC_VHIGH_I = 0x________      // INTR  : 0xFFFF0018
ew MAJIC_VHIGH_F = 0x________      // FIQ   : 0xFFFF001C
```

If you don't know what the user vectors should be set to for the high or low area, you may use the following technique to find out. First set all the user vectors for that area to **`0xFFFFFFFF`**, then set a hardware breakpoint on the UNDEF vector and run the boot code. When the first exception in that area is hit, the CPU will attempt to execute **`0xFFFFFFFF`** (from the exception interceptor), which is an undefined instruction, so it takes an UNDEF exception and stops at the hardware breakpoint. Now you can view the contents of target memory for that vector area and update your custom initialization file accordingly. With EDBICE or MONICE, you can automate this process for the high vectors using the **`xhivecs.cmd`** command file.

Notes:

- After you change your custom initialization file, you should quit the debugger and power cycle the hardware before attempting to use the system normally.
- If you want to set both vector interceptors to user defined vectors, you should repeat this process to find what each should be set to.

Debug Handler Location

The debug memory containing the debug handler is normally located from 0xFEFFFC00 to 0xFF0003FF in the physical memory space. Debug memory only responds to instruction fetches and ignores loads and stores, so this "hole" in the target's address map is normally transparent to the target system. If your system has instruction memory in that area, then the debug memory must be moved to some other area that won't conflict. Moving the debug memory is outside of the scope of this application note, as very few users will ever have to make this change. If the default range does conflict with your system, please contact EPI technical support for assistance.

Cache Management

The debug memory used for the debug exception handler and exception interceptors is actually a "mini-cache". Therefore, you must take care in your cache management code to ensure that you don't invalidate the debug memory. The cache management code running on the target can not use this instruction (Invalidate I Cache Line with MVA in Rd) where Rd hits either of the exception vectors or the region in memory where the debug handler resides.

```
MCR  p15,0,rd,c7,c5,1
```

Instead, use this instruction (Invalidate I Cache & BTB), which will not affect the debug memory.

```
MCR  p15,0,rd,c7,c5
```

Note: This is documented in the *Intel XScale® Core Developer's Manual*, in the *Mini Instruction Cache Overview* section, found in the *Software Debug* chapter.

Memory Controller Initialization

If you plan to download your code directly into RAM via the MAJIC® probe, then the RAM must first be made accessible. Many boards come up ready to access RAM, so nothing special is required. However, some boards require that the memory controller be initialized before you can access RAM. This may be accomplished in one of two ways:

1. Run the board's boot code to let it perform the required initialization, then stop it and download your program. This may be the easiest way if you have working boot code on your board, but it may cause undesirable side effects like initializing peripheral devices as well.
2. Add commands to your target initialization file to set up your memory controller by poking values into its control registers. The EDTA software package includes target initialization files with memory controller initialization scripts for many standard reference platforms. If your board resembles one of the reference boards, then you can probably use or adapt one of these.

If you plan to execute boot code from ROM or flash and have that load your code into RAM either from flash or a file server, then you need not worry about memory controller initialization. However, it is important to realize that in this case you cannot set software breakpoints in the code until after it has been loaded, because the software breakpoint instruction will be overwritten when the code is loaded. Only a hardware breakpoint may be set in code that has not yet been loaded.

MMU Utilities

The Memory Management Unit (MMU) in the Intel XScale® microarchitecture relies on mapping tables in memory to describe virtual to physical mappings, and other memory access parameters. The EDTA software package includes the **MMU_DUMP** and **MMU_XLATE** utilities to provide visibility into the operation of the MMU.

Note: These commands can be entered as shown when using EDBICE or MONICE. In other debug environments they must be passed through to the EPI debugger back-end that is in use. Please see the appropriate user manual or application note for information on entering MON commands in your debugger.

Translation Table Display

The **MMU_DUMP** command alias runs the **xdmp.cmd** file to display the memory translation table, showing how the MMU maps virtual addresses to physical addresses. The translation table may be displayed if the MMU is disabled, but the mappings are only in effect if the MMU is enabled.

Two optional parameters may be specified to control the display. The first optional parameter is a number from 0-3 controlling what to display (0 is assumed if no parameter is given).

```
MON> MMU_DUMP 0    // Dumps mixed table
MON> MMU_DUMP 1    // Dumps mixed and mapped tables
MON> MMU_DUMP 2    // Dumps mixed and not-mapped tables
MON> MMU_DUMP 3    // Dumps mixed, mapped, and not-mapped tables
```

The second parameter, if given, displays help information on the table entries. The value of the second parameter is irrelevant; just the presence of a second parameter enables display of help information.

```
MON> MMU_DUMP 0 H

                        XScale Memory Map
      Virtual              Physical       MAPPED   D1     D2   DM  AP  CB   X
00000000 - 00000fff    a0000000 - a0000fff   YES   CRSE   XSML   1   3   3   1
00001000 - 00001fff    00001000 - 00001fff   YES   CRSE   SMAL   1   3   0   0
00002000 - 000fffff    00002000 - 000fffff   YES   CRSE   SMAL   1   3   2   0
00100000 - 007fffff    00100000 - 007fffff   YES   SECT   ----   1   3   2   0
00800000 - 9fffffff    00800000 - 9fffffff   YES   SECT   ----   1   3   0   0
a0000000 - a0010fff    a0000000 - a0010fff   YES   CRSE   XSML   1   3   3   1
a0011000 - a0014fff    a0011000 - a0014fff   YES   CRSE   SMAL   1   3   0   0
a0015000 - a00fffff    a0015000 - a00fffff   YES   CRSE   XSML   1   3   3   1
a0100000 - cfffffff    a0100000 - cfffffff   YES   SECT   ----   1   3   3   0
d0000000 - d00fffff    00000000 - 000fffff   YES   SECT   ----   1   3   2   0
d0100000 - efffffff    00000000 - 00000000   NO    INV    INV    1   0   0   0
f0000000 - ffffffff    f0000000 - ffffffff   YES   SECT   ----   1   3   0   0

ROM Protection        : 1
System Protection     : 0
Control Register      : 127f
PID Register          : 08000000
Translation Base Addr : a0004000
Domain Access Register: 55555555
```

```
MAJIC XScale MMU Page Table Dump

  D1    = Level 1 Descriptor Type
  D2    = Level 2 Descriptor Type
  DM    = Domain Access bits (from Domain Access Control Register)
  AP    = Access Permission bits
  CB    = Cache and Bufferability bits
  X     = X-bit (XScale only)
MAPPED  = YES, The Virtual Address maps to Physical Address
MAPPED  = NO,  The Virtual Address does NOT map to Physical Address


Level 1 Descriptor Types
  INV   = Invalid descriptor
  CRSE  = Coarse page table descriptor
  SECT  = Section descriptor
  FINE  = Fine page table descriptor


Level 2 Descriptor Types
  INV   = Invalid descriptor
  LRGE  = Large page table descriptor
  SMAL  = Small page table descriptor
  XSML  = Extended Small page table descriptor
  TINY  = Tiny page table descriptor
  ----  = No Level 2 descriptor for section descriptor

Domain Access bits
  0  =  No access
  1  =  Client
  2  =  Reserved
  3  =  Manager

Access Permission bits
        Privileged            User
        Permissions        Permissions
  0  =  Permission dependent on S and R bits
  1  =  Read/Write          No Access
  2  =  Read/Write          Read Only
  3  =  Read/Write          Read/Write
```

Address Translation

The **MMU_XLATE** command alias runs the **xlat.cmd** file to translate the virtual address provided into the corresponding physical address. If an optional second parameter is provided, then details about the page table descriptors used in the translation are displayed as well. The detailed break down uses the same names and terminology as those found in the ARM Technical Reference Manual.

```
MON> MMU_XLATE 80001A80

  Virtual Address       :  80001A80
  Physical Address      :  A0001A80
```

```
MON> MMU_XLATE 80001A80 H

  Virtual Address       :  80001A80
  Physical Address      :  A0001A80
  Translation Base Addr :  A0000000
  1st Level Dscrptr Addr:  A0002000
  2nd Level Dscrptr Addr:  00000000
  1st Level Dscrptr     :  A000040E  SECTION
  2nd Level Dscrptr     :  00000000  ----

1st Level Descriptor Address Break Down
  1st Level Dscrptr Addr:  A0002000
  Translation Base Addr :  A0000000
  Translation Base Mask :  FFFFC000
     Table Index [13:0] :      2000
  VA Table Index [31:20]:  80000000
  VA Table Idx Bit Mask :  FFF00000
  VA Table Idx Shift Val:  18

Physical Address Break Down
   Physical Address     :  A0001A80
   Base Address         :  A0000000
   Base Address Mask    :  FFF00000
   Section Index        :     01A80
   Section Index Mask   :  000FFFFF

  TEX                   :  0
  CB Bits               :  3
  Subpage               :  NA
  Domain Field          :  0
  Domain Access         :  1
  Domain Access Register:  00000001
  Access Permission     :  1
  PID Register          :  00000000
```

Execution Trace

The MAJIC^{MX} and MAJIC^{PLUS} models support the on-chip execution trace buffer provided by the Intel XScale® microarchitecture. This allows you to capture instruction addresses while the program is executing and then display the execution history.

Note: The on-chip trace buffer only records instruction addresses. No information on loads or stores is recorded, nor is any timestamp information recorded.

Trace Acquisition

Normally trace data is continuously captured in a circular buffer as your program runs, with the oldest trace data being replaced by newer trace data. That way, when execution stops the trace buffer presents the instructions leading up to the breakpoint. By setting the **`Trace_Trigger_Action`** option to **`Start`**, though, you can begin capturing when execution starts and automatically stop execution and tracing when the buffer is full. This is most useful if you want to see what happens at the very beginning of a run. Apart from that, no triggering or conditional trace control is available.

Trace Display

EDBICE and the eXDI Plug-In provide a separate trace data window for displaying the trace data as instructions and source lines. In other debug environments, use the **DT** command to display trace data. Please see Chapter 6 of the *MAJIC® User's Manual* for more information on execution tracing.

Notes:

- In order to process the trace data, the trace processing software must analyze the instruction memory as well as the data captured in the on-chip trace buffer. If you have downloaded your program via EDB, or the MON **L** command, then the instructions in the file are used. Otherwise, instructions are fetched from memory to support the trace processing algorithm. This makes it impossible to trace self-modifying code, or code which has been swapped out of memory or otherwise become inaccessible.
- The raw trace data in the on-chip buffer provides no indication of whether ARM or Thumb instructions are being executed. Normally the trace processing algorithm assumes that all instructions are ARM instructions, but if the **Trace_Inst16** option is **On**, then it will assume that all instructions are Thumb instructions. You may change this option and then refresh the display to reprocess the raw trace data as the opposite instruction set.
- If the MMU is enabled when the trace data is processed, then Modified Virtual Addresses (MVA) based on the current Process ID (PID) are presented. If the trace acquisition includes instructions that were executed when the MMU was disabled, or the PID was different than its current value, then those instructions may not be correctly reported. This is because the on-chip trace buffer does not record the PID information.