XAPP1117 (v1.0) August 21, 2008

Software Debugging Techniques for PowerPC 440 Processor Embedded Platforms

Author: Brian Hill

Summary

This application note discusses the use of the Xilinx Microprocessor Debugger (XMD) and the GNU software debugger (GDB) to debug software defects.

Included Systems

Included with this application note is one ML507 board with PowerPC® (PPC) 440 processor reference system:

https://secure.xilinx.com/webreg/clickthrough_do?cid=111913

Introduction

XMD is used to download executables to the system, to control running these applications with breakpoints, and to examine or modify memory and CPU registers. XMD includes a TCL parser. TCL is a full featured industry standard scripting language. This combination of built-in command and scripting provides powerful debugging possibilities.

GDB is a full featured symbolic software debugger. It can make certain tasks which are cumbersome to accomplish with XMD more streamlined. GDB can be used to debug software locally - a local process running on the same machine and operating system as GDB itself, or remotely. In this document, GDB running on the local machine is used to connect to the GDB stub (also called GDB server) running within XMD. XMD automatically starts the GDB server after the user connects to the target processor.

An implemented system is provided with sample applications which contain intentional software defects. This document discusses the tools available to identify these defects.

Hardware and Software Requirements

The hardware and software requirements are:

- Xilinx ML507 Development Board
- Xilinx Platform USB Cable or Parallel IV Cable
- RS232 Cable
- Serial Communications Utility Program (e.g. HyperTerminal)
- Xilinx Platform Studio 10.1.02
- Xilinx Integrated Software Environment (ISE®) 10.1.02i

System Specifics

The included ML507 system was created with Base System Builder. The system includes a PPC440, XPS UART 16550, XPS Interrupt Controller, XPS LL TEMAC Ethernet controller, PowerPC 440 Processor DDR2 Memory Controller (ppc440mc_ddr2), XPS GPIO connected to on-board LEDs and switches, XPS Multi-Channel Memory Controller (MCH EMC) connected to on-board flash, XPS IIC Controller, and 32k of BRAM. The system address map is shown in Table 1.

Address Map

Table 1: **Reference System Address Map**

Instance	Peripheral	Base Address	High Address
xps_bram_if_cntlr_1	xps_bram_if_cntlr	0xFFFF8000	0xFFFFFFFF
FLASH	xps_mch_emc	0x86000000	0x87FFFFFF
RS232_Uart_1	xps_uart16550	0x83E00000	0x83E0FFFF
Hard_Ethernet_MAC	xps_ll_temac	0x81C00000	0x81C0FFFF
xps_intc_0	xps_intc	0x81800000	0x8180FFFF
IIC_EEPROM	xps_iic	0x81600000	0x8160FFFF
DIP_Switches_8Bit	xps_gpio	0x81460000	0x8146FFFF
Push_Buttons_5Bit	xps_gpio	0x81440000	0x8144FFFF
LEDs_Positions	xps_gpio	0x81420000	0x8142FFFF
LEDs_8Bit	xps_gpio	0x81400000	0x8140FFFF
DDR2_SDRAM	ppc440mc_ddr2	0x00000000	0x0FFFFFFF

Software Applications

This system contains applications which use the Xilinx standalone software library. These applications contain intentional software defects which will be investigated using the procedures outlined in this application note.

The TestApp_Crash application will *crash* by intentionally causing the processor to perform various errors. How to find the cause of application crashes is discussed using this application as a model. Gathering crash debug information from deployed systems is discussed.

TestApp_malloc is used to demonstrate how memory leaks may be identified.

The TestApp_temac application attempts to operate the TEMAC in loopback mode. In its present form, it does not successfully loop packets. This will be investigated.

TestApp_DCache is used to examine PPC440 Data Cache behavior.

Compiler Options

When debugging an application, it is best to compile images with symbolic debugging information included (-g), and with no optimization (-O0). Images which have been stripped (contain no symbols) will be difficult to use in a meaningful way with the debugger. Highly optimized images (-O2) will be more confusing to debug as operations may not occur in the sequence that they appear in the source text (if, in fact, they occur at all). All applications provided with this application note have been configured to compile with "-g -O0" compiler options for these reasons.

Command Conventions

This application note will often instruct the user to enter various commands. All commands are displayed as **bold**. Furthermore, the prompt displayed with the command indicates the environment for which the command is intended is shown in Table 2.

Table 2: **Command Prompts**

Prompt	Environment
$	EDK Shell
(gdb)	GDB
XMD%	XMD

Executing the Reference System

Executing the Reference System using the Pre-Built Bitstream and the Compiled Software Applications

To execute the system using files inside the `ready_for_download/` directory in the project root directory, follow these steps:

1. Change directories to the `ready_for_download` directory.
2. Use iMPACT to download the bitstream by using the following:
```
$ impact -batch xapp1117.cmd
```
3. Invoke XMD and connect to the processor by the following command:
```
$ xmd -opt xapp1117.opt
```
4. Download the executables by using the following command:
```
XMD% dow <executable name>.elf
```

Executing the Reference System from XPS

To execute the system using XPS, follow these steps:

1. Open `system.xmp` in XPS.
2. Select **Hardware** → **Generate Bitstream** to generate a bitstream for the system.
3. Download the bitstream to the board with **Device Configuration**→**Download Bitstream**.
4. Launch XMD with **Debug** → **Launch XMD..**.
5. Download the executables by the following XMD command:
```
XMD% dow <executable name>.elf
```

Console Connection

Connect a serial cable to the RS232 port on the ML507. The terminal application, such as HyperTerminal, is configured as **9600** BPS, **8** Data Bits, No Parity, **1** Stop Bit, and No Flow Control.

XMD and TCL Scripting

Note: This section of the software debugging document is meant to introduce some of the capabilities of TCL scripting within XMD. The TCL language is beyond the scope of this document.

XMD includes a TCL parser. TCL is a full featured industry standard scripting language. This combination provides powerful debugging possibilities. All the functionality of XMD (read/write registers, memory, and memory mapped devices, breakpoints, etc...) is available to user-supplied scripts which can enhance the base functionality of XMD. Any valid TCL command can be entered interactively at the XMD prompt:

```
XMD% expr 8 + 1
9
XMD% puts "hello world"
hello world
```

By writing TCL procedures, it is possible to extend XMD.

```
XMD% proc myprocedure {mynumber1 mynumber2} {
> set retval [expr $mynumber1 + $mynumber2]
> return $retval
> }
XMD%
XMD% myprocedure 8 1
9
```

The script files can be loaded into XMD (rather than typing them in, as above) with the **source** command:

```
XMD% source <myscriptfile.tcl>
XMD% myprocedure 8 1
9
```

The real power of XMD becomes evident when scripting is combined with the ability of XMD to access CPU registers and memory. XMD can access CPU registers interactively, as shown below:

```
XMD% rrd msr
   msr: 00000000
```

This command reads the PPC440 **MSR** register. The small script shown below is an example of how to use this ability to access registers or memory to display information:

```
# Read PPC440 MSR Register and examine the EE bit.
# Print in plain English if External Exceptions(interrupts) are presently
# enabled.
proc ppc440_intenable_print {} {
    # Read the MSR register.  Trim off extra text, keeping only the number.
    # "   msr:  12345678 " becomes "12345678"
    set regval [string trimleft [rrd msr] "msr: "]

    # make the number read above appear like conventional hexadecimal
    # "12345678" becomes "0x12345678"
    set regval [format "0x%08x" 0x$regval ]

    puts -nonewline "PPC External Interrupts "
    # Test the 'EE' bit
    if {$regval & 0x00008000} {
        puts "ENABLED"
    } else {
        puts "DISABLED"
    }
}
```

This simple example script is provided in the `xmd_tcl_scripts` directory as `ppc440_intenable_print.tcl`. If presently in the `ready_for_download` directory, the script would be loaded as shown below:

```
XMD% source ../../xmd_tcl_scripts/ppc440_intenable_print.tcl
```

When the procedure is executed, human-readable state information is displayed:

```
XMD% ppc440_intenable_print
PPC External Interrupts DISABLED
XMD%
```

When XMD starts, it automatically executes any TCL commands in a file called `.xmdrc` (if it exists). This file should be placed in the user's home directory. Commands can be placed here to source all of the debugging scripts when XMD is started.

This application note includes several TCL scripts found in the `xmd_tcl_scripts` directory for use with XMD as debugging aids for Xilinx embedded systems. These scripts display CPU and peripheral register values, and decode many register fields. To utilize these scripts, copy ***dotxmdrc*** from the `xmd_tcl_scripts` directory to **$HOME/.xmdrc**. The user may also want to copy the `xmd_tcl_scripts` directory to a more general area apart from where the project was unzipped.

An example, as entered from the EDK Shell within the `xmd_tcl_scripts` directory, is shown:

```
$ cp dotxmdrc $HOME/.xmdrc
```

The user's `.xmdrc` file should be edited to reflect the directory where these TCL scripts have been placed.

Now, when XMD is started, these scripts alert the user about the new commands that are available:

```
$ xmd
...
Loading custom commands from c:/data/tcl:
mem_read_byte
mem_write_byte
mem_read_hwd
memcpy
ppc405_print
ppc405_rm_sa_print
ppc405_read_tlb
ppc405_dcache_print
ppc440_print
ppc440_read_tlb
ppc440_dcache_print
ppc_bt
mb_print
mb_bt
emaclite_print
lltemac_print
lltemac_read_phy
marvell_phy_print
marvell_phy_probe
national_phy_print
national_phy_probe
lldma_mm_print
lldma_desc_print
uartlite_print
uartns550_print
xps_intc_print
xiic_print
xiic_read_byte
xiic_write_byte
```

```
ipv4_packet_decode
en_packet_decode
csum16
XMD%
```

Note: The remainder of this application note assumes that the user has configured XMD as described in this section.

TestApp_Crash

Using XMD, download and run TestApp_crash from the `ready_for_download` area (assumes bitstream is already downloaded):

```
$ xmd
XMD% connect ppc hw
XMD% dow TestApp_Crash.elf
XMD% run
```

`When run, the following menu is displayed:`

```
-- Entering main() --
Choose:
1) Illegal Instruction (Program Exception)
2) Machine Check
3) Enable Machine Check Exceptions
4) Alignment
5) Enable Alignment Exceptions
6) Data Storage
7) TLB Miss (data)
8) Stack corruption
9) Setup software exception handler
```

The user selects **`1) Illegal Instruction (Program Exception)`** in the terminal console at this time. By examining the source code, the expected output would be:

```
Entering gen_illegal_instr
Branching to 0x000070C4
Exiting gen_illegal_instr
```

Identifying the Problem

When option 1 (Illegal Instruction) is chosen in the application's present form, only the first two lines are printed correctly, which indicates that the application has *crashed*, thereby resulting in the appearance of only some of the expected output. The cause of this error in such a small application as TestApp_Crash is easily found by examining the source code, but if this were a much larger application the task would be much more difficult. If the application were a Unix process, the user would expect the a message from the operating system to tell them that the process had been terminated, and some indication of why. The Xilinx standalone library is a very minimal environment. Unless the application explicitly sets up exception handling for software errors there will be no indication of what has happened.

A *crash* can be one of several events: The processor executed an invalid instruction, an attempt to execute privileged code from user mode, or any other access violation (violating memory protections set in TLB entries). When any such event occurs, the processor generates an exception, often referred to as an interrupt. Exceptions cause the processor to execute code at the appropriate exception vector for the type of exception encountered. In the example, TestApp_Crash has executed an invalid instruction (how this can be determined shall be demonstrated below), which generates a PROGRAM exception on the PowerPC 440 processor. Program exceptions are associated with IVOR6.

This means that the processor will execute instructions at the address determined by the vector stored in Interrupt Vector Offset Register 6 (IVOR6) after the start of the exception vector table. Often, the exception vector table begins at `0x00000000`, but that is not the case for the test application. The base of the exception vector table can be set by software with the IVPR Special Purpose Register. **The test application has not yet initialized this register, and it will contain a random value.**

To determine the cause of this crash, set a hardware breakpoint at the PROGRAM exception vector. Because software has not initialized the exception registers, IVPR and IVOR6 are set to known values with the debugger:

```
$ xmd
XMD% connect ppc hw
XMD% rst
XMD% dow TestApp_Crash.elf
XMD% rwr ivpr 0x00000000
XMD% rwr ivor6 0x00000060
```

A hardware breakpoint is set at IVPR + IVOR6:

```
XMD% bps 0x00000060 hw
Setting breakpoint at 0x00000060

XMD% run
Info:Processor started. Type "stop" to stop processor

<Choose 1) Illegal Instruction (Program Exception) in terminal.>
RUNNING> XMD% Info:Hardware Breakpoint 0 Hit, Processor Stopped at
0x00000060

XMD%
```

The processor is now stopped at the hardware breakpoint set at the PROGRAM exception vector.

The `ppc440.tcl` script, provided in the `xmd_tcl_scripts` directory, can provide some information about the cause of this crash. The commands that the script contains should already be available if the instructions in the "XMD and TCL Scripting" section of this application note have been completed.

Execute the "**`ppc440_print`**" procedure after the application hits the breakpoint as shown below:

```
XMD% ppc440_print
PVR:  0x7ff21912 Silicon: 2.1
MSR:  0x00000000 PRpriv
TCR:  0x00000000
TSR:  0xcc000000 ENW WIS
ESR:  0x08000000 PIL
```

This script prints the PVR, MSR, TCR, TSR, and ESR PowerPC 440 processor registers, and decodes many of the bits. This is applicable to TestApp_Crash because the Exception Syndrome Register (ESR) contains the specific reason an exception has occurred (there are many possibilities). The text, PIL (*Program - Illegal Instruction)*, corresponds with the name used in the PowerPC Processor Reference Guide.

Explanation of the Problem

TestApp_Crash executes an invalid PPC instruction. It accomplishes this by filling an array with arbitrary data, and then executing this arbitrary data as code.

Excerpt from the `TestApp_Crash/src/TestApp_Crash.c` file:

```
/*
 * gen_illegal_instr:
 * This function will generate an Illegal Instruction Program Exception
 * by filling the array crash_instructions[] with
 * arbitrary data, then branching to this array address causing the PPC
 * to attempt to execute this data as code.
 */
void
gen_illegal_instr (void) {
    void (*fp) (void);

    xil_printf("Entering gen_illegal_instr\n\r");

    /*
     * Fill crash_instructions with arbitrary data.
     */
    crash_instructions[0] = 0;
    crash_instructions[1] = 1;
    crash_instructions[2] = 2;
    crash_instructions[3] = 3;
    /*
     * Branch to address of the array crash_instructions[]
     * (which contains only data, not valid instructions)
     */
    fp = (void*)crash_instructions;
    xil_printf("Branching to 0x%08x\n\r", (unsigned)fp);
    (*fp)();

    xil_printf("Exiting gen_illegal_instr\n\r");
}
```

When a PROGRAM exception occurs, the processor sets the effective address of the instruction that caused the exception in SRR0:

```
XMD% srrd srr0
  srr0: 000070c4
```

This is the address of the instruction (`0x000070C4`) which, when executed, generated the exception. The **`nm`** command is used to display the addresses of all symbols in the executable:

```
$ powerpc-eabi-nm --numeric-sort TestApp_Crash.elf
```

Note: A premade symbol listing is provided in <EDK_Project>/TestApp_Crash/TestApp_Crash.sym

It can now be seen that the illegal instruction is at the address of the array crash_instructions, the address to where gen_illegal_instr() branched.

```
...
00007080 ? __tdata_start
00007084 b object.3143
0000709c B __malloc_current_mallinfo
000070c4 B crash_instructions
000070ec B XExc_VectorTable
000071ec B __bss_end
000071ec B _stack_end
...
```

Other useful information is found in the contents of the Link Register which contains the return address for the last function call (the last time a branch-with-link type instruction was executed, including bl, bla, bnel, bnela, etc...).

```
 XMD% srrd lr
     lr: 00000f9c
```

The symbol listing shows that the last function call was made from somewhere within gen_illegal_instr() -- the return address of `0x00000f9C` is greater than the start of gen_illegal_instr() at `0x00000f10` and less than the end gen_illegal_instr() at `0x00000fc0`:

```
 ...
00000e30 T setup_crashtrace
00000f10 T gen_illegal_instr
00000fc0 T gen_machine_check
...
```

It could be helpful to see what function was called from within gen_illegal_instr(). To do this, the executable file is disassembled with the **`objdump`** utility:

```
 $ powerpc-eabi-objdump -S TestApp_Crash.elf > TestApp_Crash.dis
```

Note: Objdump will only include source in the disassembly (as shown here) if run on the executable in the directory where it was compiled. A premade disassembly is provided in <EDK_Project>/TestApp_Crash/TestApp_Crash.dis

```
---
00000f10 <gen_illegal_instr>:
...
     (*fp)();
      f90:       80 1f 00 08     lwz     r0,8(r31)
      f94:       7c 09 03 a6     mtctr   r0
      f98:       4e 80 04 21     bctrl

     xil_printf("Exiting gen_illegal_instr\n\r");
      f9c:       3d 20 00 00     lis     r9,0
      fa0:       38 69 63 e0     addi    r3,r9,25568
      fa4:       48 00 0d 4d     bl      1cf0 <xil_printf>
```

It is found that the last line of “C” language executed was a function call, via the function pointer fp, as implemented in the Branch to CTR with Link (bctrl) instruction at `0xF98`. This resulted in the branch to crash_instructions, where the invalid instructions in question are found.

How to Solve the Problem

TestApp_Crash is not a realistic example of any common programming error. No application design would ever specifically include code to execute arbitrary data. This application does provide a useful framework to locate software errors with the debugger, and an introduction to other useful software tools. This framework is a foundation for the other applications in this application note. The “problem” in TestApp_Crash is easily solved by simply removing the code inside gen_illegal_instr() which branched to the array crash_instructions.

Disassembling Instructions in Memory With XMD

Identifying the Problem

It has been shown in the “TestApp_Crash” section, that examining a disassembly of the application can be a useful debugging aide. At times it would be useful to produce a disassembly of instructions in memory directly within XMD. XMD does not natively provide this capability.

Solving the Problem

The TCL scripting abilities of XMD are used to expand the build-in functionality of XMD. The `ppc_dis.tcl` script, provided with this application note, will produce a disassembly as shown below, where the first argument is the address to begin disassembling, and the second argument is the number of instructions to disassemble:

```
XMD% stop
XMD% ppc_dis 0xf90 4
 f90:    80 1f 00 08     lwz     r0,8(r31)
 f94:    7c 09 03 a6     mtctr   r0
 f98:    4e 80 04 21     bctrl
 f9c:    3d 20 00 00     lis     r9,0

XMD%
```

The disassembly shown corresponds to that of TestApp_Crash produced in the "TestApp_Crash" section.

If the user has properly configured `.xmdrc` as detailed in "XMD and TCL Scripting", this command will be available for immediate use.

The script operates by saving the specified section of memory to a binary file `memory.bin`. This file is converted to a PPC object file `memory.o` with the `powerpc-eabi-objcopy` executable, which must be in the PATH for the script to operate properly. This object file is then disassembled with the `powerpc-eabi-objdump` executable. For this script to function properly, the following conditions are necessary:

1. The script creates temporary files. As such, the user must have write permission on the current directory.
2. The `powerpc-eabi-objcopy` and `powerpc-eabi-objdump` executables must be in the PATH.

Introduction to GDB

Identifying the Problem

Using the TestApp_Crash application again, the same problem debugged previously with XMD is re-examined, this time using GDB. As previously observed, this application will "crash" because it executes an illegal instruction. The same information previously gathered with XMD is collected again with GDB, with some of the features present only in GDB introduced.

Explanation of the Problem

Before beginning with GDB, the bitstream should already have been downloaded with XMD started and connected to the processor.

Start GDB. GDB is used in textual mode as indicated by the -nw switch.

Note: Most of the advanced features of GDB are only available through the GDB command prompt. The GDB GUI provides no graphical access to these features. For this reason, GDB is used entirely in textual mode throughout this application note.

```
$ powerpc-eabi-gdb -nw TestApp_Crash.elf
```

Inform GDB that the processor it will be debugging is a PowerPC 440 processor by using the command:

```
(gdb) set processor powerpc:440
```

Next, have GDB connect to the *target* — in this case the GDB server within XMD. Because this is a network connection, GDB and XMD can be running on different machines:

```
(gdb) target remote (hostname or ip address of machine running
XMD):1234
Remote debugging using (remote machine ip address):1234
0xfffffffc in ?? ()
```

If GDB and XMD are run on the same machine, `localhost` may be used to specify the machine to which GDB should connect by using the command:

```
(gdb) target remote localhost:1234
```

Now, tell GDB to download the application into memory:

```
(gdb) load
```

As was done with XMD, a hardware breakpoint is set at the PROGRAM exception vector:

Note: Exception handling has not been initialized by software. Due to this, some output in the remainder of this section may vary from one instance to another, and is unlikely to match this text exactly.

```
(gdb) info registers ivpr
ivpr           0x0      0
(gdb) info registers ivor6
ivor6          0x60     96
(gdb) hbreak *0x00000060
Hardware assisted breakpoint 1 at 0x60
```

Now, start the application. The menu shall be display on the console.

```
(gdb) continue
Continuing.
```

As before, **`1) Illegal Instruction`** is chosen. The application will “crash” and GDB will stop at the breakpoint.

```
Breakpoint 1, IVOR6 () at xvectors.S:511
```

Execution stops at the PROGRAM exception. GDB can display the nesting of function calls (the callstack) which have occurred up to the time of the exception. Have GDB display the callstack with a backtrace command (bt):

```
(gdb) bt
#0  IVOR6 () at xvectors.S:511
#1  0x00000f9c in gen_illegal_instr ()
#2  0x00001518 in choose_crash ()
#3  0x000015d0 in main ()
```

It is seen that from somewhere in gen_illegal_instr() or a function called within gen_illegal_instr() has caused the application to crash. The numbers (#0 - #3) indicate stack frames, one for each function in the callstack. The stack and stack frames are discussed in detail in the “Debugging Stack Errors” section of this application note. For now, it is not necessary to understand how GDB accomplishes this.

As was done with XMD, the exception address the CPU placed in SRR0 is examined:

```
(gdb) info registers srr0
srr0           0x70c4   28868
```

GDB will look up an address in the symbol table using the command:

```
(gdb) info symbol 0x70c4
crash_instructions in section .bss
```

The backtrace provided by GDB is very useful, but GDB can examine the Link Register directly as was done with XMD:

```
(gdb) info registers lr
lr              0xf9c    3996
(gdb) info symbol 0xf9c
gen_illegal_instr + 140 in section .text
```

GDB will also disassemble instructions in memory. To see what is at the address stored in the Link Register, use the command:

```
(gdb) disassemble gen_illegal_instr
Dump of assembler code for function gen_illegal_instr:
...
0x00000f74 <gen_illegal_instr+100>:     addi    r0,r9,28868
0x00000f78 <gen_illegal_instr+104>:     stw     r0,8(r31)
0x00000f7c <gen_illegal_instr+108>:     lwz     r0,8(r31)
0x00000f80 <gen_illegal_instr+112>:     lis     r9,0
0x00000f84 <gen_illegal_instr+116>:     addi    r3,r9,25544
0x00000f88 <gen_illegal_instr+120>:     mr      r4,r0
0x00000f8c <gen_illegal_instr+124>:     bl      0x1cf0 <xil_printf>
0x00000f90 <gen_illegal_instr+128>:     lwz     r0,8(r31)
0x00000f94 <gen_illegal_instr+132>:     mtctr   r0
0x00000f98 <gen_illegal_instr+136>:     bctrl
0x00000f9c <gen_illegal_instr+140>:     lis     r9,0
```

And again, it is observed that the instruction preceding `0x00000f9C` (gen_illegal_instr + 140) is a branch to crash_instructions() as achieved with a Branch to CTR with Link instruction. The contents of the CTR register are examined:

```
(gdb) info registers ctr
ctr             0x70c4   28868
```

The branch was taken to `0x70C4`, the address of `crash_instructions`.

How to Solve the Problem

TestApp_Crash has once again served as a useful model to demonstrate how to find software errors, this time using GDB. As before, the offending code in gen_illegal_instr has been identified. To prevent TestApp_Crash from causing an exception, remove the function call to the array crash_instructions.

Debugging Stack Errors

CPU registers are finite in number. The data which one individual function must work with can easily exceed this set. This limitation is resolved with a construct known as the stack. The stack is an area of memory used to hold temporary data - variables local to a function and saved register values. The PowerPC processor architecture, unlike some other CPU architectures (such as Intel) does not architecturally require the use of a stack, nor is explicit hardware support provided for a stack. There is no hardware-defined stack pointer, and when function calls are made or interrupts occur, the PPC does not automatically store any data on the stack (as occurs with Intel x86 and its successors). Because there is no register specifically assigned by hardware as a stack pointer, and the processor does not directly involve itself with stack manipulation, **all PPC stack usage is by convention only.** This means that the caller and the callee have an implicit agreement between themselves which General Purpose Register is to be considered the stack pointer, and where all parties assume certain values to have been placed on the stack. By convention, General Purpose Register 1 (GPR 1) is the stack pointer on a PPC system. The stack grows with each function call in a downward direction -- from higher memory addresses towards lower memory addresses. Each function has its own **stack frame** pointed to by the stack pointer in that function. A function, when called, will create a new stack frame for itself by decrementing the stack pointer (r1) by the appropriate amount to create a scratch pad for itself. Before returning, it will restore the stack pointer to that of the calling function. Stack usage will be discussed in more detail later in this section.

Identifying the Problem

Earlier usage of TestApp_Crash has been a useful introduction to the debugger, but it does not represent any likely errors. It does not demonstrate how a real-world application might suddenly execute unexpected code. One common reason is stack corruption. When function calls are made, register values from the calling function are saved on the stack so that they can be restored when the called function returns. Among items saved are the stack pointer and the contents of the Link Register. In the C programming language, local variables are also placed on the stack. This provides the opportunity for bugs, which shall be examined.

Explanation of the Problem

A different TestApp_Crash menu choice is used in this section: **`8) Stack Corruption.`** The pertinent pieces of TestApp_Crash are now examined. The skeleton of the application resembles the following:

Note: this is ***not*** the actual program text!

```
void
myfunction3 (void) {
    unsigned mylocalarray[THE_SIZE];

    printf("-- Entering myfunction3() --\r\n");
    mylocalarray[THE_INDEX] = (something);
    printf("-- Exiting myfunction3() --\r\n");
}

void
myfunction2 (void) {
    printf("-- Entering myfunction2() --\r\n");
    myfunction3();
    printf("-- Exiting myfunction2() --\r\n");
}

void
myfunction1 (void) {
    printf("-- Entering myfunction1() --\r\n");
    myfunction2();
    printf("-- Exiting myfunction1() --\r\n");
}

int main (void) {
   print("-- Entering main() --\r\n");
   myfunction1();
   print("-- Exiting main() --\r\n");
   return 0;
}
```

At first glance, the expected output would be:

```
-- Entering main() --
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
-- Exiting myfunction3() --
-- Exiting myfunction2() --
-- Exiting myfunction1() --
-- Exiting main() --
```

But this is not the behavior observed, it is instead:

```
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
-- Exiting myfunction3() --
Entering gen_illegal_instr
Branching to 0x000070C4
```

Nowhere is gen_illegal_instr() explicitly called or branched to. To investigate what has happened, the program is run with the debugger:

```
$ powerpc-eabi-gdb -nw TestApp_Crash.elf
(gdb) set processor powerpc:440
```

Connect to the XMD GDB server:

```
(gdb) target remote <host>:1234
Remote debugging using <host>:1234
0xfffffffc in ?? ()
```

Have GDB download the executable to memory:

```
(gdb) load
```

Set a hardware breakpoint at the location observed in IVPR + IVOR6:

```
(gdb) hbreak *0x60
Hardware assisted breakpoint 1 at 0x60
```

Continue program execution from the present PC:

```
(gdb) c
Continuing.
```

Choose `8) Stack corruption` from the application menu. The application will "crash", and GDB will stop at the breakpoint.

Note: The backtrace produced below may not appear exactly like that seen by the user. When an application "crashes", subsequent behavior varies based upon un-initialized data (memory, CPU registers). The same application, run on the same hardware, will not necessarily always produce the same results. This is one of many important debugging lessons.

Display the callstack backtrace:

```
(gdb) bt
#0  IVOR6 () at xvectors.S:511
#1  0x00000f9c in gen_illegal_instr () at
TestApp_Crash/src/TestApp_Crash.c:95
#2  0x00000f10 in setup_crashtrace ()
#3  0x00001414 in myfunction1 () at TestApp_Crash/src/TestApp_Crash.c:301
#4  0x00001568 in choose_crash () at TestApp_Crash/src/TestApp_Crash.c:342
#5  0x000015d0 in main () at TestApp_Crash/src/TestApp_Crash.c:363
```

By looking at the source, it is seen that chose_crash() calls myfunction1(), which calls myfunction2(), which calls myfunction3(). There is no call in myfunction1() to gen_illegal_instr() as the above backtrace seems to indicate. In addition, the program output indicates that myfunction2() and myfunction3() have actually been executed.

Note: Anytime that the backtrace appears improbable, the developer should suspect that stack corruption or a stack overflow has occurred.

Examine the myfunction1 stack frame more closely. The frame number is displayed in the above backtrace output.

```
(gdb) info frame 3
Stack frame at 0x75b0:
 pc = 0x1414 in myfunction1 (TestApp_Crash/src/TestApp_Crash.c:301); saved
pc 0x1568
 called by frame at 0x75d8, caller of frame at 0x75a0
 source language c.
 Arglist at 0x75a0, args:
 Locals at 0x75a0, Previous frame's sp is 0x75b0
 Saved registers:
  r31 at 0x75ac, pc at 0x75b4, lr at 0x75b4
```

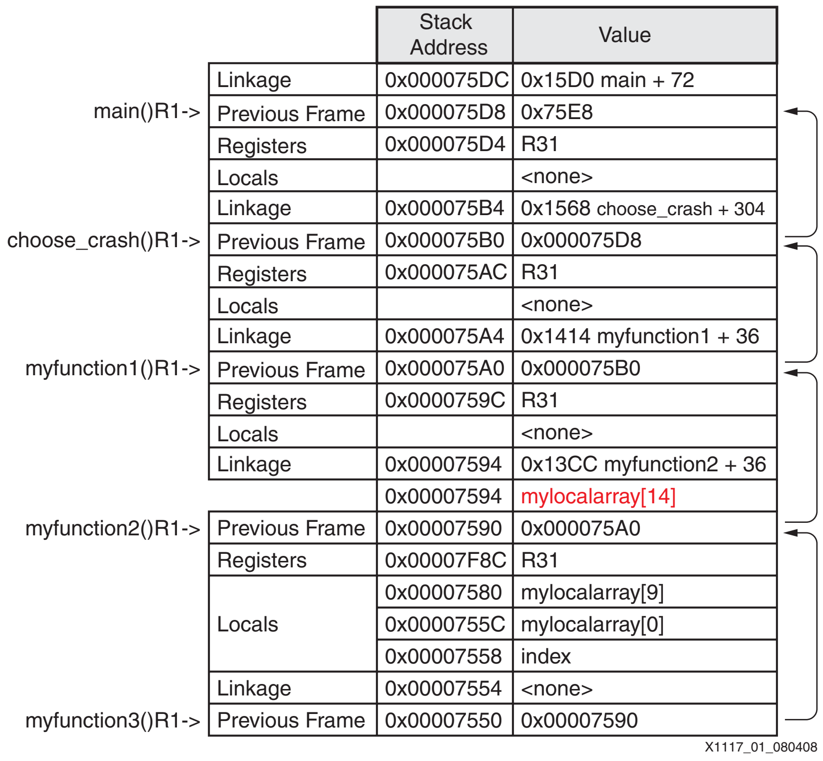
This shows that myfunction1() left its frame at pc `0x1414`. Disassemble the function to see what instruction is at `0x1414`:

```
(gdb) disassemble myfunction1
Dump of assembler code for function myfunction1:
0x000013f0 <myfunction1+0>:     stwu    r1,-16(r1)
0x000013f4 <myfunction1+4>:     mflr    r0
0x000013f8 <myfunction1+8>:     stw     r31,12(r1)
0x000013fc <myfunction1+12>:    stw     r0,20(r1)
0x00001400 <myfunction1+16>:    mr      r31,r1
0x00001404 <myfunction1+20>:    lis     r9,0
0x00001408 <myfunction1+24>:    addi    r3,r9,26200
0x0000140c <myfunction1+28>:    bl      0x2d9c <puts>
0x00001410 <myfunction1+32>:    bl      0x13a8 <myfunction2>
0x00001414 <myfunction1+36>:    lis     r9,0
```

So, the last thing myfunction1() did was call myfunction2() (which is clearly not a call to gen_illegal_instr()).

This calls for some logical thinking. Clearly, myfunction2() calls myfunction3(). The first thing any function will do is set up a stack frame. It saves the link register it will use to return to the caller on the stack frame of the caller — not it's own stack frame. This is at a known, fixed offset (stackpointer + 4 bytes). The link register is saved because function calls may occur in myfunction3 (and, in fact, do occur), which would overwrite the current link register contents. If the value of the LR saved on the stack were to be modified before the function returned, the function would then return to the incorrect address. Examine if this is what is happening with TestApp_Crash.

For a model of the stack in this application, see Figure 1.



		Stack Address	Value
	Linkage	0x000075DC	0x15D0 main + 72
main()R1->	Previous Frame	0x000075D8	0x75E8
	Registers	0x000075D4	R31
	Locals		<none>
	Linkage	0x000075B4	0x1568 choose_crash + 304
choose_crash()R1->	Previous Frame	0x000075B0	0x000075D8
	Registers	0x000075AC	R31
	Locals		<none>
	Linkage	0x000075A4	0x1414 myfunction1 + 36
myfunction1()R1->	Previous Frame	0x000075A0	0x000075B0
	Registers	0x0000759C	R31
	Locals		<none>
	Linkage	0x00007594	0x13CC myfunction2 + 36
		0x00007594	mylocalarray[14]
myfunction2()R1->	Previous Frame	0x00007590	0x000075A0
	Registers	0x00007F8C	R31
	Locals	0x00007580	mylocalarray[9]
		0x0000755C	mylocalarray[0]
		0x00007558	index
	Linkage	0x00007554	<none>
myfunction3()R1->	Previous Frame	0x00007550	0x00007590

Figure 1: **TestApp_StackOverflow Stack**

The disassembly of myfunction3() is examined below to get a better understanding of the stack manipulation that occurs in this function. Additional commentary has been added to the listing to explain what is occurring.

```
$ powerpc-eabi-objdump -S TestApp_Crash.elf
```

Note: Objdump will only include source in the disassembly (as shown here) if run on the executable in the directory where it was compiled. So that the user need not recompile TestApp_Crash, a premade disassembly is provided in the <EDK_Project>/TestApp_Crash/TestApp_Crash.dis file.

```
void
myfunction3 (void) {
# At call time, R1(the stack pointer) = 0x00007550 + 64
# store word and update (stwu) the stack pointer.
# *sp = 0x00007590; sp = sp - 64bytes;
    1338:       94 21 ff c0      stwu    r1,-64(r1)
# Copy the contents of the link register to r0
    133c:       7c 08 02 a6      mflr    r0
# Store contents of R31 on the stack at 0x00007550 + 60 (0x0000758C)
    1340:       93 e1 00 3c      stw     r31,60(r1)
# Store the contents of the LR at 0x00007550 + 68 (0x00007594)
# This offset is within the caller's frame (68 > 64).
    1344:       90 01 00 44      stw     r0,68(r1)
# copy stack pointer to R31
    1348:       7c 3f 0b 78      mr      r31,r1
    unsigned mylocalarray[10];
    int index;

#define CALLBYFRAME 1
#define CALLERFRAME 2
#define PREVFRAMESP 3
#define SAVEDLR     4

    printf("-- Entering myfunction3() --\r\n");
    134c:       3d 20 00 00     lis     r9,0
    1350:       38 69 65 d8     addi    r3,r9,26072
    1354:       48 00 1a 49     bl      2d9c <puts>

    index = (sizeof(mylocalarray)/sizeof(unsigned)) + SAVEDLR;
# index = (10 + 4) = 14
# index is beyond the end of the array. Using this index will corrupt
# the stack.
    1358:       38 00 00 0e     li      r0,14
    135c:       90 1f 00 08     stw     r0,8(r31)
    /* the value of index is out of bounds - this will write past the end
     * of the array.
     */
# the address of mylocalarray[14]      = 0x00007594
# the link address register was saved at 0x00007594
# the data saved in this array has overwritten the LR -- stack corruption.
# when this function returns, it will now branch to gen_illegal_instr()
    mylocalarray[index] = (unsigned) gen_illegal_instr;
    1360:       81 7f 00 08     lwz     r11,8(r31)
    1364:       3d 20 00 00     lis     r9,0
    1368:       38 09 0f 10     addi    r0,r9,3856
    136c:       7c 0a 03 78     mr      r10,r0
    1370:       55 69 10 3a     rlwinm  r9,r11,2,0,29
    1374:       38 1f 00 08     addi    r0,r31,8
    1378:       7d 29 02 14     add     r9,r9,r0
    137c:       39 29 00 04     addi    r9,r9,4
    1380:       91 49 00 00     stw     r10,0(r9)

    printf("-- Exiting myfunction3() --\r\n");
    1384:       3d 20 00 00     lis     r9,0
    1388:       38 69 65 f8     addi    r3,r9,26104
    138c:       48 00 1a 11     bl      2d9c <puts>
}
# Load R11 with previous stack pointer
    1390:       81 61 00 00     lwz     r11,0(r1)
# Get saved link register from the stack
    1394:       80 0b 00 04     lwz     r0,4(r11)
# move saved LR value to the link register
    1398:       7c 08 03 a6     mtlr    r0
# place saved R31 value into R31
    139c:       83 eb ff fc     lwz     r31,-4(r11)
# restore previous stack pointer
    13a0:       7d 61 5b 78     mr      r1,r11
# branch to the link register, returning from this function.
# this will branch to crash_function()
    13a4:       4e 80 00 20     blr
```

There is a local variable called *mylocalarray*. Because it is a local variable, this is instantiated on the stack. This array is only written to once:

```
mylocalarray[index] = (unsigned) gen_illegal_instr;
```

This appears to be a correct line of C code, yet it causes myfunction3() to return to the incorrect address. This is because *index* is outside the bounds of the array. *mylocalarray* has 10 elements only. In this case, index = 14. This code will write to a location on the stack that it should not — in this case, the location to where myfunction3() has saved the contents of its Link Register. This is commonly called *Smashing the stack*.

How to Solve the Problem

This application contains a common error — accessing an array outside of its bounds. The problem can be resolved by increasing the size of the array to at least 15 elements. Ordinarily, it is desirable for software to verify that any value to be used as an index to an array will not be beyond the end of the array.

Another common cause of stack problems is a stack overflow. Standalone applications are assigned a fixed amount of stack space at compile time. Too many nested function calls or local variables can exceed this amount. If problems which appear to be stack corruption appear, it may be a useful test to try increasing the stack space available to the application. For Xilinx standalone executables, this can be set by modifying the applications linker script.

Debugging Crashes in the Field

Identifying the Problem

When an embedded system is deployed in the field, debugging application crashes takes on new challenges. It is generally not possible to run the application with the debugger in this environment. Some kind of useful error reporting from the field is necessary. Software is installed at the PROGRAM exception vector which will supply debugging information that customers can forward to the appropriate technical support staff.

Explaining the problem

In the TestApp_Crash application there is provided a **minimal** example of crash debugging software. The project file, `ppc440_crashtrace.c`, contains code to set up a handler for this exception and print out useful information when a crash occurs. The necessary function, setup_crashtrace(), is not called by default. Before choosing to generate a particular error, such as **`8) Stack corruption`**, choose **`9) Setup software exception handler`**:

```
<choose 9) Setup software exception handler>
<choose 8) Stack corruption>
```

Program output is now:

```
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
-- Exiting myfunction3() --
Entering gen_illegal_instr
Branching to 0x000070C4


STACK MAXIMUM DEPTH: 0x258 bytes out of 0x404 total.
========================================================
Program - Illegal Instruction Exception at: 0x000070C4
FRAME: 0x000074D0 RETURN 0x00000F9C
FRAME: 0x00007570 RETURN 0x00000F90
FRAME: 0x00007590 RETURN 0x00000F10
FRAME: 0x000075A0 RETURN 0x00001414
FRAME: 0x000075B0 RETURN 0x00001568
FRAME: 0x000075D8 RETURN 0x000015D0
FRAME: 0x000075E8 RETURN 0x00000444
========================================================
```

This information from the serial console should be saved to a text file, such as `TestApp_Crash-stack_corrupt.crash`, for later analysis. A TCL script **stackscan** is provided with this application note which will look up addresses from the crash output in a symbol file:

Produce a symbol file for the application:

```
$ powerpc-eabi-nm --numeric-sort TestApp_Crash.elf >
TestApp_Crash.sym
```

Execute the script using XMD as the TCL parser in the `ready_for_download` directory:

```
$ xmd -tcl ../../xmd_tcl_scripts/stackscan.tcl TestApp_Crash-
stack_corrupt.crash TestApp_Crash.sym
EXCEPTION ADDR: 0x000070C4 B crash_instructions
FRAME:          0x00000F9C T gen_illegal_instr+140
FRAME:          0x00000F90 T gen_illegal_instr+128
FRAME:          0x00000F10 T gen_illegal_instr
FRAME:          0x00001414 T myfunction1+36
FRAME:          0x00001568 T choose_crash+304
FRAME:          0x000015D0 T main+72
FRAME:          0x00000444 T __vectors_end+196
```

The script can also be executed with **tclsh** as the TCL interpreter. **TCLSH** is part of the standard TCL installation available at http://sourceforge.net/projects/tcl and is NOT part of the Xilinx EDK install. If TCL is installed, the script can be run like any ordinary executable installed somewhere in a place specified with the users PATH environment variable:

```
$ stackscan.tcl TestApp_Crash-stack_corrupt.crash
TestApp_Crash.sym
```

How to Solve the Problem

The callstack output with symbols provided by stackscan can give developers a good clue of what activity was taking place when the application crashed. From this output it is seen that an Illegal instruction was executed from within crash_instructions -- an array. The processor should not be attempting to execute code from an array, which contains data. It is seen that the last function called was gen_illegal_instr(), which seems to have been called by myfunction1(). It is known from examining the source that myfunction1() never calls gen_illegal_instr(). Any irregularities in the callstack should make the developer suspect that stack corruption has occurred. The developer now knows to begin looking at code within myfunction1() and any functions that it calls, perhaps specifically looking for any references to gen_illegal_instr or the variable crash_instructions.

Backtrace with XMD

Identifying the Problem

The ability of GDB to provide a backtrace, previously introduced in the "Debugging Stack Errors" section, is an invaluable debugging aid. XMD does not natively provide this capability.

How to Solve the Problem

The TCL scripting abilities of XMD are used to expand the built-in functionality of XMD. The `ppc_bt.tcl` script provided with this application note will produce a symbolic backtrace as shown below:

```
XMD% stop
XMD% ppc_bt
PC:    0x00000c8c            ProgramExceptionHandler ppc440_crashtrace.c:380
LR:    0x00000c80            ProgramExceptionHandler ppc440_crashtrace.c:377
R1:    0x00007488 0x00000c80 ProgramExceptionHandler ppc440_crashtrace.c:377
R1:    0x000074d0 0x00000188 _vector110        xvectors.S:572
R1*:   0x000074d0 0x000070c4 __bss_start       ??:0
R1*:   0x000074d0 0x00000f9c gen_illegal_instr TestApp_Crash.c:97
```

```
R1:   0x00007570 0x00000f90 gen_illegal_instr TestApp_Crash.c:95
R1:   0x00007590 0x00000f10 gen_illegal_instr TestApp_Crash.c:77
R1:   0x000075a0 0x00001414 myfunction1       TestApp_Crash.c:303
R1:   0x000075b0 0x00001568 choose_crash      TestApp_Crash.c:342
R1:   0x000075d8 0x000015d0 main              TestApp_Crash.c:364
XMD%
```

The backtrace shown corresponds to that produced by `ppc440_backtrace.c` in the "Debugging Crashes in the Field" section.

If the user has properly configured the `.xmdrc` file as detailed in "XMD and TCL Scripting" this command will already be available for immediate use.

The script operates by reading the pertinent addresses from memory. These addresses are resolved to symbols with the `powerpc-eabi-addr2line` executable, which must be in the PATH for the script to operate properly. The name of the last ELF file downloaded by XMD is provided to addr2line so that it may match the address to a symbol in that executable. For this to function properly, the following conditions are necessary:

1. The user must not have changed directories within XMD since downloading (via the **`cd`** command).
2. The ELF file must still exist, unmodified, and be readable by the user.
3. The ELF file must not have been stripped of symbols.

If these conditions are not met, the backtrace will not contain symbolic information. The file which shall be used may be seen as follows:

```
XMD% puts $elf_file
TestApp_Crash.elf
```

It is seen that TestApp_Crash.elf will be used. It is expected to be in the **current directory.**

Debugging TLB Errors

Identifying the Problem

The PPC440 may generate several TLB related exceptions, including Data Storage, TLB Miss (data), and TLB Miss (instruction). The TLB Miss (data) exception is discussed as a representative example in this section.

Explanation of the Problem

A TLB Miss (data) occurs when the processor performs a load or store to an address for which there is no virtual->physical mapping in the TLB. On a "real" operating system, such as Linux, this is not an unusual occurrence, and will be part of the ordinary workings of the virtual memory system. For the standalone BSP, any such exception should be considered an unrecoverable error.

The PPC440 has no Real addressing mode - the TLB must always be used. The Xilinx standalone BSP emulates Real mode by creating 1:1 mappings of the entire address space in 256MB chunks.

TestApp_Crash will be used to generate a TLB Miss (data) exception by intentionally unmapping a page within the TLB, and then accessing memory which was within that range (which shall now access a virtual address with no virtual->physical mapping). The following excerpt shows how this is performed:

```
#define UNMAPPED_DATA_ADDRESS          0xB000000C

/*
 * gen_data_tlb_miss:
 * Cause a TLB Miss (data) exception to be generated.
 * Unmap a page of memory, and then access that virtual address.
 */
void gen_data_tlb_miss (void) {
    volatile unsigned *ptr;
    unsigned val, page;

    xil_printf("Entering gen_data_tlb_miss\n\r");

    /* The virtual address to access */
    ptr = (unsigned*) UNMAPPED_DATA_ADDRESS;

    /*
     * Unmap this page
     */
    page = XTlb_FindPage(UNMAPPED_DATA_ADDRESS, 0, TS_DATA);
    xil_printf("Erasing TLB entry #%d\n\r", page);
    XTlb_ErasePage(page);
    /* Without a context synchronization here, the following will not
     * cause the desired TLB miss.
     */
    XTlb_Commit();

    xil_printf("Reading from 0x%x\n\r", (unsigned)ptr);
    /*
     * An exception will occur here.
     */
    val = *ptr;
    xil_printf("0x%x read from 0x%x\n\r", val, (unsigned)ptr);

    xil_printf("Exiting gen_data_tlb_miss\n\r");
}
```

Download and run the TestApp_Crash.elf file:

```
XMD% dow TestApp_Crash.elf
XMD% run
```

From the application menu, choose

```
9) Setup software exception handler
```

and then choose

```
7) TLB Miss (data)
```

The application will crash with a TLB Miss exception:

```
Entering gen_data_tlb_miss
Erasing TLB entry #23
Reading from 0xB000000C


STACK MAXIMUM DEPTH: 0x1D8 bytes out of 0x4004 total.
=======================================================
Data Exception Address: 0xB000000C
TLB miss (data) - load Exception at: 0x00001164
FRAME: 0x0000A390 RETURN 0x00001160
FRAME: 0x0000A430 RETURN 0x00001160
FRAME: 0x0000A450 RETURN 0x00001550
FRAME: 0x0000A478 RETURN 0x000015C0
FRAME: 0x0000A488 RETURN 0x00000444
=======================================================
```

The processor has placed the unmapped address on which software attempted a load in the Data Exception Address Register (DEAR). The exception handler software displays this value when appropriate. It is seen that this value, `0xB000000C`, matches UNMAPPED_DATA_ADDRESS used by TestApp_Crash.

Within XMD, produce a symbolic backtrace using the `ppc_bt.tcl` script:

```
XMD% stop
XMD% ppc_bt
PC:   0x00000c8c            ProgramExceptionHandler ppc440_crashtrace.c:380
LR:   0x00000c80            ProgramExceptionHandler ppc440_crashtrace.c:377
R1:   0x000074a8 0x00000c80 ProgramExceptionHandler ppc440_crashtrace.c:377
R1:   0x000074f0 0x00000188 _vector110              xvectors.S:572
R1*:  0x000074f0 0x00001174 gen_data_tlb_miss TestApp_Crash.c:180
R1*:  0x000074f0 0x00001170 gen_data_tlb_miss TestApp_Crash.c:180
R1:   0x00007590 0x00001170 gen_data_tlb_miss TestApp_Crash.c:180
R1:   0x000075b0 0x00001560 choose_crash      TestApp_Crash.c:341
R1:   0x000075d8 0x000015d0 main              TestApp_Crash.c:364
R1:   0x000075e8 0x00000444 _start            ??:0
XMD%
```

When `TestApp_Crash.c` line 180 is examined, the load which caused the exception is seen:

```
        /*
         * An exception will occur here.
         */
        val = *ptr;
```

Debugging the TLB with XMD

The exception generated in the previous step indicated that the software attempted to do a load from `0xB000000C`, but there was no virtual->physical mapping for this virtual address. To debug such issues it is useful to view the present TLB contents. XMD provides access to TLB entries by pseudo memory mapping them. When XMD connects to the processor, it displays certain debug addresses, TLB entries included:

```
XMD% connect ppc hw
....
User Defined Address Map to access Special PowerPC Features using XMD:
        I-Cache (Data)........0x70000000 - 0x70007fff
        I-Cache (TAG).........0x70008000 - 0x7000ffff
        D-Cache (Data)........0x78000000 - 0x78007fff
        D-Cache (TAG).........0x78008000 - 0x7800ffff
        DCR...................0x78020000 - 0x78020fff
        TLB...................0x70020000 - 0x70023fff
```

The script, `ppc440_tlb.tcl,` is provided with this application note to decode and display TLB entries using this XMD feature.

If the procedures to setup the users `.xmdrc` have been followed as outline in "XMD and TCL Scripting" this command will already be available for use.

The contents of the TLB are displayed:

```
XMD% ppc440_read_tlb 0x70020000
#  RPN        EPN        TS SIZE  ERPN U0U1U2U3 W I M G E UX UW UR SX SW SR
 0 0x00000000 0x00000000    256MB 0               I       UX UW UR SX SW SR
 1 0x00000000 0x00000000 TS 256MB 0               I       UX UW UR SX SW SR
 2 0x10000000 0x10000000    256MB 0               I       UX UW UR SX SW SR
 3 0x10000000 0x10000000 TS 256MB 0               I       UX UW UR SX SW SR
 4 0x20000000 0x20000000    256MB 0               I       UX UW UR SX SW SR
 5 0x20000000 0x20000000 TS 256MB 0               I       UX UW UR SX SW SR
 6 0x30000000 0x30000000    256MB 0               I       UX UW UR SX SW SR
 7 0x30000000 0x30000000 TS 256MB 0              I       UX UW UR SX SW SR
 8 0x40000000 0x40000000    256MB 0              I       UX UW UR SX SW SR
 9 0x40000000 0x40000000 TS 256MB 0              I       UX UW UR SX SW SR
10 0x50000000 0x50000000    256MB 0              I       UX UW UR SX SW SR
11 0x50000000 0x50000000 TS 256MB 0              I       UX UW UR SX SW SR
12 0x60000000 0x60000000    256MB 0              I       UX UW UR SX SW SR
13 0x60000000 0x60000000 TS 256MB 0              I       UX UW UR SX SW SR
14 0x70000000 0x70000000    256MB 0              I       UX UW UR SX SW SR
15 0x70000000 0x70000000 TS 256MB 0              I       UX UW UR SX SW SR
16 0x80000000 0x80000000    256MB 0              I       UX UW UR SX SW SR
17 0x80000000 0x80000000 TS 256MB 0              I       UX UW UR SX SW SR
18 0x90000000 0x90000000    256MB 0              I       UX UW UR SX SW SR
19 0x90000000 0x90000000 TS 256MB 0              I       UX UW UR SX SW SR
20 0xa0000000 0xa0000000    256MB 0              I       UX UW UR SX SW SR
21 0xa0000000 0xa0000000 TS 256MB 0              I       UX UW UR SX SW SR
22 0xb0000000 0xb0000000    256MB 0              I       UX UW UR SX SW SR
24 0xc0000000 0xc0000000    256MB 0              I       UX UW UR SX SW SR
25 0xc0000000 0xc0000000 TS 256MB 0              I       UX UW UR SX SW SR
26 0xd0000000 0xd0000000    256MB 0              I       UX UW UR SX SW SR
27 0xd0000000 0xd0000000 TS 256MB 0              I       UX UW UR SX SW SR
28 0xe0000000 0xe0000000    256MB 0              I       UX UW UR SX SW SR
29 0xe0000000 0xe0000000 TS 256MB 0              I       UX UW UR SX SW SR
30 0xf0000000 0xf0000000    256MB 0              I       UX UW UR SX SW SR
31 0xf0000000 0xf0000000 TS 256MB 0              I       UX UW UR SX SW SR
XMD%
```

How to Solve the Problem

It is seen that there is no virtual->physical mapping for `0xB0000000` Translation Space 1 (the standalone BSP uses Translation Space 1 for data, Translation Space 0 for instructions). This is because TestApp_Crash deleted TLB entry 23, which previously provided this mapping.

If the application did not modify this TLB entry in gen_data_tlb_miss() as shown:

```
    XTlb_ErasePage(page);
```

it would not crash.

Debugging Memory Allocation

Identifying the Problem

Stack errors are not the only memory pool errors that can occur. Most software of any complexity will require dynamic memory allocation where the memory allocated is more persistent than the life of a single function call (as happens with the stack). The well known libc functions **malloc**() and **free**() are used to dynamically carve up a block of dynamic memory known as the **heap**. Like all system resources, space on the heap is finite. One of the more difficult software errors to track down is a **memory leak**. A memory leak occurs when memory which was allocated for a transient purpose is never freed, and is therefore lost to the system until a reboot occurs. Eventually no memory will be left in the pool, and all calls to **malloc**() will fail.

Explanation of the Problem

The most effective method to debug any heap related issue is to replace the library versions of **malloc**() and **free**() with debug versions which have been specially instrumented. The Xilinx EDK does not provide source code for the malloc implementation used. This prevents easy modification of malloc itself, or close examination of the data structures it uses.

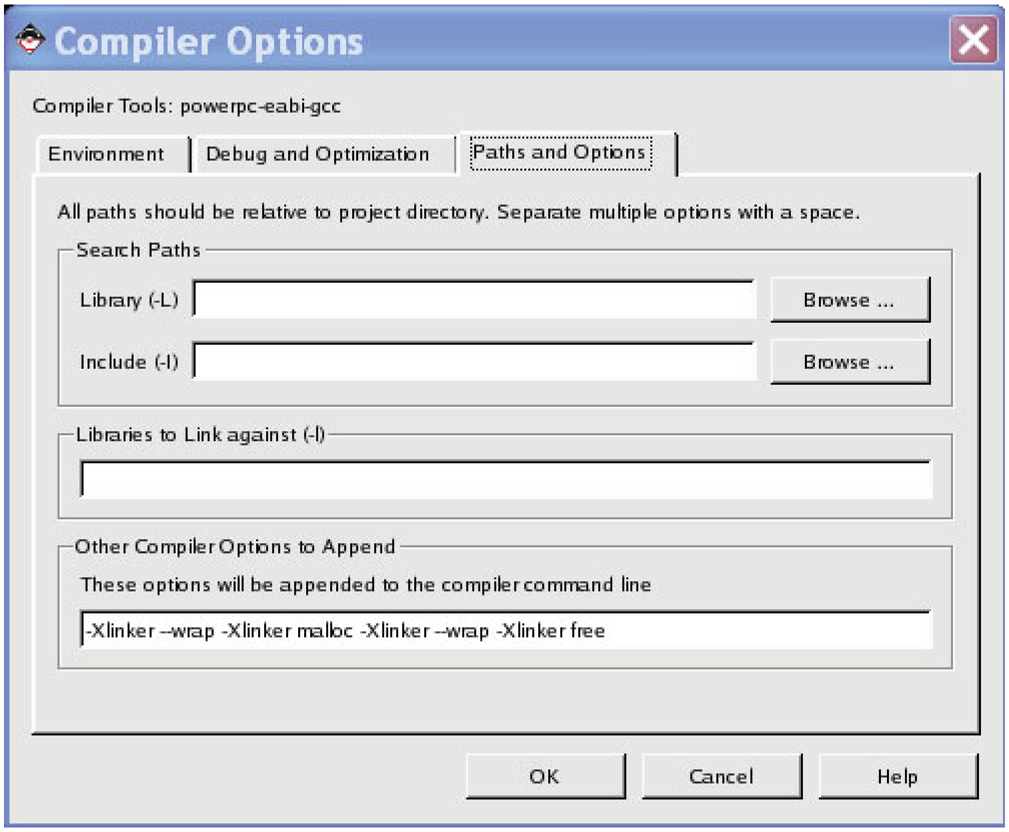
At first glance, it would seem that the proper course of action is to download a suitable library source (such as newlib) and build it. With the assistance of a special linker feature, however, it is possible to produce a debug solution without modifying the library version of malloc() or knowledge of any of its internals. A **wrapper** shall be created.

The linker parameter **--wrap** allows for any library function to be intercepted. For example, "--wrap malloc" would cause any call to malloc() to be intercepted with the function __wrap_malloc() which is provided in the application. The __wrap_malloc() function will perform appropriate debugging tasks, and then allocate the requested memory by calling __real_malloc(), which is the actual library malloc function.

XPS provides an option for user-specified compiler flags. The compiler, in turn, provides a method for flags to be passed to the linker with the **-Xlinker** compiler option. The proper compiler options to create a wrapper for the library function malloc() are:

-Xlinker --wrap -Xlinker malloc

See Figure 2 for an example XPS compiler options configuration to create a wrapper for malloc() and free(). These are used in the provided application TestApp_malloc.


Compiler Options
Compiler Tools: powerpc-eabi-gcc
Environment
Debug and Optimization
Paths and Options
All paths should be relative to project directory. Separate multiple options with a space.
Search Paths
Library (-L)
Browse ...
Include (-I)
Browse ...
Libraries to Link against (-l)
Other Compiler Options to Append
These options will be appended to the compiler command line
OK
Cancel
Help
X1117_01_080408

-Xlinker --wrap -Xlinker malloc -Xlinker --wrap -Xlinker free

Figure 2: **Setting the Linker Wrapper Option**

TestApp_malloc

TestApp_malloc provides code useful for locating memory leaks. This is done by building upon lessons learned with TestApp_Crash. As discussed previously, it is possible to determine the precise location where a function call occurred all the way up the call tree. This is a useful tool which can be applied to determine the callers of malloc in the application. Statistics are kept on a per-caller basis. In this way it can be seen what memory has been allocated and never freed.

The caller of a function is determined, as seen previously, by examining the contents of the Link Register in this excerpt from `TestApp_malloc/src/ppc_debug_malloc.c`:

```
/*
 * Determine where malloc() was called from
 */
lr_val = mfspr(XREG_SPR_LR);
```

This information is stored in the allocated memory itself. Additional bytes are added to the users requested memory to hold this debug info, and the user is returned a pointer past the beginning of allocated memory. See Figure 3.

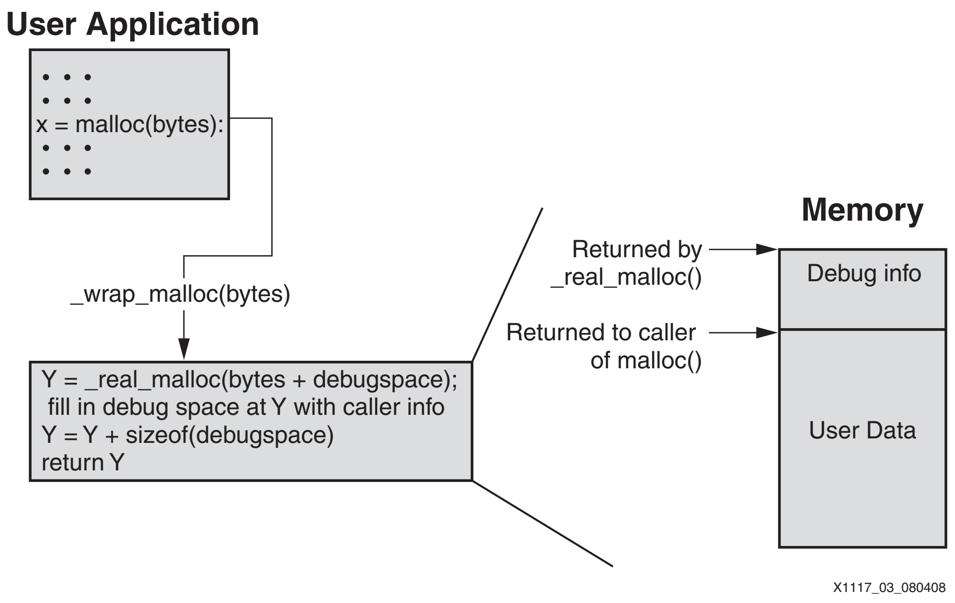

User Application
• • •
• • •
x = malloc(bytes):
• • •
• • •
_wrap_malloc(bytes)
Y = _real_malloc(bytes + debugspace);
fill in debug space at Y with caller info
Y = Y + sizeof(debugspace)
return Y
Memory
Returned by _real_malloc()
Returned to caller of malloc()
Debug info
User Data
X1117_03_080408

Figure 3: **Malloc Wrapper Flow**

The array *malloc_log_data[]* is used to track total bytes allocated from an individual call location. Since an array is statically allocated, the maximum number of callers to malloc() which will be tracked is determined at compile time.

```
/*
 * Data to be recorded in debug array when malloc() is called
 */
typedef struct malloc_data malloc_data_t;
struct malloc_data {
    u32    caller_return; /* Stats per caller PC */
    Xint32 bytes;         /* Bytes allocated from this caller */
   u32    total_allocs;  /* Total times malloc() called from this caller */
   u32    total_frees;   /* number of these allocs which were later freed */
};

/*
 * data from each malloc() and free() are stored here. This is statically
 * sized since it is memory allocation itself being debugged.
 */
static malloc_data_t malloc_log_data[MAX_MALLOC_CALLER_LOG];
```

Calls to free() are intercepted in __wrap_free(). Here, the saved caller to malloc() is retrieved from the debug space. This caller information is used to find the applicable entry within *malloc_log_data[]* and decrement the amount of memory logged to the appropriate caller of malloc().

When a call to malloc fails (when malloc() returns NULL), the gathered statistics are viewed with a call to malloc_log_print().

The output of TestApp_malloc is shown below:

```
-- Entering main() --
MemWaster4 TestApp_Malloc/src/main.c:148 malloc failed!
MALLOC Statistics:
Total calls to malloc: 87
Total calls to free:   27
Malloc failures:       1
MALLOC CALLER: 0x000001f8 TOTAL BYTES:      448 ALLOCS:    40 FREES    26
MALLOC CALLER: 0x000002e0 TOTAL BYTES:     2610 ALLOCS:    45 FREES     0
MALLOC CALLER: 0x000003cc TOTAL BYTES:      100 ALLOCS:     1 FREES     0
```

The address of each MALLOC CALLER is looked up in the symbol table for this image, as described earlier in this document with TestApp_Crash, by using the command:

```
$ powerpc-eabi-nm --numeric-sort TestApp_malloc.elf >
TestApp_malloc.sym
```

Excerpt from generated TestApp_malloc.sym file:

```
...
00000164 t frame_dummy
000001d0 T MemWaster1
000002b8 T MemWaster2
00000360 T MemWaster3
00000434 T MemWaster4
000004c0 T main
...
```

Each of the addresses provided in the MALLOC statistics is looked up in the symbol table by hand following the procedure first outlined with "TestApp_Crash". It is found that:

```
MALLOC CALLER 0x000001f8 MemWaster1+40 presently has 448 bytes allocated
MALLOC CALLER 0x000002e8 MemWaster2+48 presently has 2610 bytes allocated
MALLOC CALLER 0x000003cc MemWaster3+108 presently has 100 bytes allocated
```

MemWaster2() is the largest consumer of dynamically allocated memory in the application at the time when malloc_log_print() was called. This information provides an initial suspect for the memory leak.

How to Solve the Problem

The best that any debug tool can do is provide data - a clue where the investigation should proceed. That is the case observed with TestApp_malloc. The memory usage statistics identify which places within the application allocate the most memory in a persistent manner (memory which has not been freed). Higher memory usage does not guarantee a memory leak -- there may really be a large amount of data to store. Therefore, knowledge of what type of data is stored and how much of it there is will be necessary to debug a possible memory leak.

A Real Application

In this section, a more substantial application TestApp_temac is debugged. This application will place the ethernet PHY into loopback and then loop a few packets. DMA will be used to supply packets for transmission to the MAC and for packets received by the MAC to make their way to memory.

Identifying the Problem

When run successfully, the expected output is:

```
Starting Application.
XLlDma_Initialize:
XLlTemac_SetMacAddress:
XLlDma_mBdClear:
RX XLlDma_BdRingCreate:
```

```
XLlDma_BdRingClone:
TX XLlDma_BdRingCreate:
XLlDma_BdRingClone:
Wait for XTE_RDY_HARD_ACS_RDY_MASK:
TemacUtilEnterLoopback:
XLlTemac_SetOperatingSpeed:
TemacUtilPhyDelay:
TemacSetupIntrSystem:
TemacSgDmaIntrSingleFrameExample:
Success!
TemacSgDmaIntrCoalescingExample:
Success!
TemacDisableIntrSystem:
XLlTemac_Stop:
Success
```

However, as in previous examples, this application is not behaving as desired. Instead, the output will be:

```
Starting Application.
XLlDma_Initialize:
XLlTemac_SetMacAddress:
XLlDma_mBdClear:
RX XLlDma_BdRingCreate:
XLlDma_BdRingClone:
TX XLlDma_BdRingCreate:
XLlDma_BdRingClone:
Wait for XTE_RDY_HARD_ACS_RDY_MASK:
TemacUtilEnterLoopback:
XLlTemac_SetOperatingSpeed:
TemacUtilPhyDelay:
TemacSetupIntrSystem:
TemacSgDmaIntrSingleFrameExample:
```

No further output is printed. The application stops functioning.

Explanation of the Problem

To investigate, run the application with GDB.

```
$ powerpc-eabi-gdb -nw TestApp_temac.elf
(gdb) set processor powerpc:440
(gdb) target remote <host>:1234
(gdb) load
(gdb) c
```

```
Continuing.
```

Now, wait for the application to progress to the point where it does not proceed. At this point, enter `^C` to interrupt the program. Program execution will stop, and GDB will display a prompt again:

Examine the current status:

```
(gdb) bt
#0  0x0000a174 in XIo_In32 (InAddress=2172649480) at xio.c:216
#1  0x00008424 in XIntc_DeviceInterruptHandler ()
#2  0x00008350 in XIntc_InterruptHandler ()
#3  0x00000188 in _vector110 () at xvectors.S:572
#4  0x00004884 in XCache_InvalidateDCacheRange ()
#5  0x000038d8 in XLlDma_BdRingFromHw ()
#6  0x00000f88 in TemacSgDmaIntrSingleFrameExample ()
#7  0x000009ec in TemacSgDmaIntrExample ()
#8  0x000005c0 in main ()
```

The application appears to remain stuck endlessly in an interrupt handler. If the debugger were told to continue, and the application interrupted again, the backtrace would always appear similar.

Note: The external interrupt vector, and the functions it will call, are a big software "loop". The debugger can interrupt this loop at any point. Each time the program is executed and interrupted with the debugger this location may be different.

So, execution of the application was interrupted somewhere inside of TemacSgDmaIntrSingleFrameExample(). No matter where the debugger has interupted program execution, this function always appears in the backtrace, even if subsequent functions differ. Match this to a line of C code:

```
$ powerpc-eabi-objdump -S TestApp_temac.elf > TestApp_temac.dis
```

Note: When **objdump** is provided the **-S** option, it will produce a disassembly listing with lines of C source code (as seen below). This is only useful if the source file was compiled with all optimization disabled (as TestApp_temac has been). Objdump will only include source in the disassembly if run on the executable in the directory where it was compiled. So that the user need not recompile TestApp_temac, a premade disassembly is provided in `<EDK_Project>/TestApp_temac/TestApp_temac.dis`.

Excerpt from `TestApp_temac.dis`:

```
          /*
           * Wait for transmission to complete
           */

          while (!FramesTx);
       f64:       3d 20 00 01     lis     r9,1
       f68:       80 09 5c e0     lwz     r0,23776(r9)
       f6c:       2f 80 00 00     cmpwi   cr7,r0,0
       f70:       41 9e ff f4     beq+    cr7,f64
<TemacSgDmaIntrSingleFrameExample+0x47c>

          /*
           * Now that the frame has been sent, post process our TxBDs.
          * Since we have only submitted 2 to HW, then there should be only
2 ready
           * for post processing.
           */
          if (XLlDma_BdRingFromHw(TxRingPtr, 2, &Bd1Ptr) == 0) {
       f74:       38 1f 00 28     addi    r0,r31,40
       f78:       80 7f 00 14     lwz     r3,20(r31)
       f7c:       38 80 00 02     li      r4,2
       f80:       7c 05 03 78     mr      r5,r0
       f84:       48 00 28 e1     bl      3864 <XLlDma_BdRingFromHw>
       f88:       7c 60 1b 78     mr      r0,r3
```

Comparing the above disassembly with the source file, it is found that the application was here when interrupted:

```
          /*
           * Wait for transmission to complete
           */

          while (!FramesTx);

          /*
           * Now that the frame has been sent, post process our TxBDs.
```

```
         * Since we have only submitted 2 to HW, then there should be only
2 ready
          * for post processing.
          */
         if (XLlDma_BdRingFromHw(TxRingPtr, 2, &Bd1Ptr) == 0) {
                 TemacUtilErrorTrap("TxBDs were not ready for post
processing");
                 return XST_FAILURE;
         }
```

The application had progressed to `xlltemac_example_intr_sgdma.c`:623 when it was stopped by the debugger. This information is useful because it clarifies exactly what has already happened and what has not yet been done. To get to this point, it is necessary to have transmitted a packet, as the software will have waited for this to happen:

```
xlltemac_example_intr_sgdma.c:616


            while (!FramesTx);
```

These statistics are examined (FramesRx and FramesTx) with GDB using the command:

```
(gdb) p FramesTx
$1 = 1
(gdb) p FramesRx
$2 = 0
```

It can be seen that the frame which was transmitted was never received by software as it should have been.

Now use one of the TCL scripts provided with this application note to look at the DMA engine. This script (`lldma.tcl`) prints out the registers, and decodes the pertinent configuration bits.

DCR Register access with XMD

The PPC440 processor Hard DMA control registers are not on the PLBv46. They are accessible by PowerPC processor Device Control Registers (DCR). Software accesses these registers by using the mtdcr and mfdcr PPC instructions. Since it is undesirable and intrusive for the debugger to execute machine instructions to read register contents, XMD provides a method to access DCR as though it were a memory mapped peripheral, much as it does for TLB and processor cache contents. XMD displays this address when connecting to the processor:

```
XMD% connect ppc hw

PowerPC440 Processor Configuration
-------------------------------------
Version.............................0x7ff21910
User ID.............................0x00f00000
No of PC Breakpoints................4
No of Addr/Data Watchpoints.........2
User Defined Address Map to access Special PowerPC Features using XMD:
        I-Cache (Data)........0x70000000 - 0x70007fff
        I-Cache (TAG).........0x70008000 - 0x7000ffff
        D-Cache (Data)........0x78000000 - 0x78007fff
        D-Cache (TAG).........0x78008000 - 0x7800ffff
        DCR...................0x78020000 - 0x78020fff
        TLB...................0x70020000 - 0x70023fff
```

To determine the address that XMD will use to emulate DCR access, the following formula is used:

```
XMD DCR Base + (DCR * 4)
```

DMA0 is connected to the XPS LL Temac. The base address of DMA0 is DCR 0x80. This corresponds to XMD as:

```
0x78020000 + (0x80 *4) = 0x78020200
```

Run the lldma_mm_print procedure, providing the base address of the Hard DMA device as was just determined:

```
XMD% lldma_mm_print 0x78020200
TX Ring
0x78020200 NDESC 0x00016a00
0x78020204 BUFA  0x00016156
0x78020208 BUFL  0x00000000
0x7802020c CDESC 0x00016a00
0x78020210 TDESC 0x000169c0
0x78020214 CR    0x01010000 IRQ_TMO_0x1 IRQ_THR_0x1 MSB_ADDR_0x0

0x78020218 IRQ   0x00010000 ClscCntValue_1
0x7802021c SR    0x00000014 COMPLETED EOP
RX Ring
0x78020220 NDESC 0x000189c0
0x78020224 BUFA  0x00016746
0x78020228 BUFL  0x000001f8
0x7802022c CDESC 0x000189c0
0x78020230 TDESC 0x00018980
0x78020234 CR    0x01010087 IRQ_TMO_0x1 IRQ_THR_0x1 MSB_ADDR_0x0
                           IRQ_EN IRQ_ERROR_EN IRQ_DELAY_EN IRQ_COALESCE_EN
0x78020238 IRQ   0x00014401 ClscCntValue_1 ClscCnt_1 WRQ_EMPTY COALESCE_IRQ
0x7802023c SR    0x0000001c COMPLETED SOP EOP

0x78020240 DMACR 0x0000001c RX_OVF_ERR_DIS TX_OVF_ERR_DIS TAIL_PTR_EN
XMD%
```

The TX ring has completed at least one packet transmission. It is also seen that a complete packet has been received. This is known by the RX Ring Status Register bits (COMPLETED, SOP, EOP). Interrupts are enabled on the device, and the RX COALESCE_IRQ is presently asserted. The software has not processed this waiting packet (shown above, FramesRx = 0). Refer to the PPC440 Hard DMA data sheet for additional details.

The systems interrupt configuration is now examined with the debugger. When an exception occurs, the code installed at the appropriate vector is run. With the Xilinx standalone library, this small amount of code installed at the vector (see xvectors.S) will then branch to a handler installed for that particular vector source. It is known that this exception code (from ppc440_0/libsrc/standalone_v2_00_a/src/xvectors.S) has been installed at the external interrupt vector because that is where the debugger stopped.

First, instruct GDB to print in a human readable form by using the command:

```
(gdb) set print pretty
(gdb) set radix 16
```

When exception handlers are installed with XExc_RegisterHandler(), they are placed in this array:

```
Excerpts from xexception_l.h and xexception_l.c :
XExc_VectorTableEntry XExc_VectorTable[XEXC_ID_LAST + 1];
typedef struct
{
   XExceptionHandler Handler;
   void *DataPtr;
   void *ReadOnlySDA;
   void *ReadWriteSDA;
} XExc_VectorTableEntry;
```

Now, view what exception handlers are installed:

```
(gdb) p XExc_VectorTable
$1 = {{
    Handler = 0x4b48 <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  }, {
    Handler = 0x4b48 <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  }, {
    Handler = 0x4b48 <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  }, {
    Handler = 0x4b48 <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  }, {
    Handler = 0x8308 <XIntc_InterruptHandler>,
    DataPtr = 0x16940,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  }, {
    Handler = 0x4b48 <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1d6dc,
    ReadWriteSDA = 0x1dcb8
  } <repeats 11 times>}
(gdb)
```

At array index 4, which corresponds to the external interrupt vector, XIntc_InterruptHandler is found. This is as expected - it is the xintc to which external interrupts are connected. When an external interrupt occurs, the CPU will execute code at `IVPR + IVOR4`. This code will find **`XIntc_InterruptHandler`** in the table and call it. The interrupt controller software has its own call vector table with handlers for all the external devices connected to it.

The XIntc call vector table is defined as:

`Excerpts from xintc.h and xintc_g.c`

```
XIntc_Config XIntc_ConfigTable[]
/*
 * This typedef contains configuration information for the device.
 */
typedef struct {
        u16 DeviceId;            /**< Unique ID  of device */
        u32 BaseAddress;         /**< Register base address */
        u32 AckBeforeService;    /**< Ack location per interrupt */
        u32 Options;             /**< Device options */

        /** Static vector table of interrupt handlers */
      XIntc_VectorTableEntry HandlerTable[XPAR_INTC_MAX_NUM_INTR_INPUTS];
} XIntc_Config;
```

The individual *XIntc_VectorTableEntry* fields in *HandlerTable* are set by *XIntc_Connect()*. This structure is defined as:

```
/* The following data type defines each entry in an interrupt vector table.
 * The callback reference is the base address of the interrupting device
 * for the driver interface given in this file and an instance pointer for
the
 * driver interface given in xintc.h file.
 */
typedef struct {
        XInterruptHandler Handler;
        void *CallBackRef;
} XIntc_VectorTableEntry;
```

```
((gdb) p *XIntc_ConfigTable
$2 = {
  DeviceId = 0x0,
  BaseAddress = 0x81800000,
  AckBeforeService = 0x0,
  Options = 0x1,
  HandlerTable = {{
      Handler = 0x9334 <StubHandler>,
      CallBackRef = 0x16940
    }, {
      Handler = 0x2008 <TemacErrorHandler>,
      CallBackRef = 0x1aa1c
    }, {
      Handler = 0x9334 <StubHandler>,
      CallBackRef = 0x16940
    }, {
      Handler = 0x9334 <StubHandler>,
      CallBackRef = 0x16940
    }, {
      Handler = 0x1df8 <TxIntrHandler>,
      CallBackRef = 0x1a984
    }}
}
```

Handler TxIntrHandler is seen, but no RxIntrHandler appears. Examining the xparameters.h file for this system the following is seen:

```
#define XPAR_XPS_INTC_0_SYSTEM_FPGA_0_HARD_ETHERNET_MAC_PHY_MII_INT_INTR
0
#define XPAR_XPS_INTC_0_HARD_ETHERNET_MAC_TEMACINTC0_IRPT_INTR 1
#define XPAR_XPS_INTC_0_RS232_UART_1_IP2INTC_IRPT_INTR 2
#define XPAR_XPS_INTC_0_PPC440_0_DMA0RXIRQ_INTR 3
#define XPAR_XPS_INTC_0_PPC440_0_DMA0TXIRQ_INTR 4
```

An entry in HandlerTable[3] is expected for receive interrupts. Examine the callback data for the handler that is there, StubHandler. This is of type XIntc:

```
(gdb) p *(XIntc*) XIntc_ConfigTable.HandlerTable[3].CallBackRef
$3 = {
  BaseAddress = 0x81800000,
  IsReady = 0x11111111,
  IsStarted = 0x22222222,
  UnhandledInterrupts = 0x98f94,
  CfgPtr = 0x156fc
}
```

Many un-handled interrupts are seen. StubHandler() is the default handler, present when no specific handler has been installed.

So, the interrupt is enabled, and a received packet means that it is presently being generated. With nothing to handle this interrupt (and make it go away) the application remains in the interrupt handler forever.

Now when the source is examined, the following is found:

```
#ifdef NOTNOW_FIXME
        Status |= XIntc_Connect(IntcInstancePtr, DmaRxIntrId,
                                (XInterruptHandler) RxIntrHandler,
                                RxRingPtr);
#endif
```

How to Solve the Problem

The code to install RxIntrHandler was accidently removed or forgotten. By removing the `#ifdef` and `#endif` and recompiling the application the problem is resolved.

GDB Macros

Identifying the Problem

The "A Real Application" section demonstrated the powerful symbolic debugging capabilities available with GDB. It has also made clear that digging through internal data structures can be a cumbersome task. GDB provides a macro capability to facilitate scripting common tasks.

Explanation of the Problem

GDB has a macro feature. The following is a simple example:

```
(gdb) define hello_macro
Type commands for definition of "hello_macro".
End with a line saying just "end".
>printf "hello world\n"
>end
```

The defined macro, consisting of any valid GDB command, can now be executed as desired:

```
(gdb) hello_macro
hello world
(gdb)
```

This feature is utilized to ease cumbersome tasks. The scripts may be sourced from a text file. The following script, provided in `XIntc.txt`, will display all presently installed interrupt handlers:

```
# Display all registered external interrupt handlers
define all_int_info
   # How many interrupt handlers are there?
   set $num_ints = sizeof(((XIntc_Config*)0)->HandlerTable) / \
                   sizeof(XIntc_VectorTableEntry)

   printf "INT HANDLER    CALLBACK\n"
   set $int = 0
   # Print each handler entry
   while $int < $num_ints
      set $handler  = XIntc_ConfigTable.HandlerTable[$int].Handler
      set $callback = XIntc_ConfigTable.HandlerTable[$int].CallBackRef
      printf "%3d 0x%08X 0x%08x ", $int, $handler, $callback
      info sym $handler
```

```
            # Next table entry
            set $int = $int + 1
        end
    end
```

The macro is loaded as follows:

```
    (gdb) source ../../gdb_macros/XIntc.txt
```

All interrupt handlers are displayed:

```
    (gdb) all_int_info
    INT HANDLER    CALLBACK
      0 0x00009334 0x00016940 StubHandler in section .text
      1 0x00002008 0x0001aa1c TemacErrorHandler in section .text
      2 0x00009334 0x00016940 StubHandler in section .text
      3 0x00009334 0x00016940 StubHandler in section .text
      4 0x00001DF8 0x0001a984 TxIntrHandler in section .text
    (gdb)
```

It is quickly seen that no `RxIntrHandler` is present.

PPC440 Data Cache

This section does **not** discuss a PPC440 exception, or any specific software error.

Identifying the Problem

Whether debugging software management of the cache, or resolving a suspected hardware bug, it can be useful to view the contents of the PPC 440 processor data cache and compare that to actual memory contents. XMD provides this functionality in a similar manner to the way TLB entries may be accessed.

Explanation of the Problem

PPC 440 Processor Data Cache Organization

The Virtex®-5 FXT PPC 440 processor data cache is a 32K 64-way set associative cache. There are 16 sets.

```
    64 way X 16 sets X 32 bytes per cache line = 32K bytes.
```

The cache is indexed by address bits 23-26. These four bits determine to which set of the 16 sets an address will belong. Many addresses belong to the same set. The cache may contain up to 64 addresses belonging to the same set (64 *ways*). Each address belonging to a different way of the same set is distinguished by the cache tag. This is comprised of address bits 0-22. Address bits 27-31 (5 bits, sufficient to address 32 bytes) comprise an index/byte offset into the data array for the particular cache line. Refer to IBM PPC 440x5 CPU Core User's Manual.

When connecting to the processor, XMD displays it's pseudo memory map:

```
    User Defined Address Map to access Special PowerPC Features using XMD:
            I-Cache (Data)........0x70000000 - 0x70007fff
            I-Cache (TAG).........0x70008000 - 0x7000ffff
            D-Cache (Data)........0x78000000 - 0x78007fff
            D-Cache (TAG).........0x78008000 - 0x7800ffff
            DCR...................0x78020000 - 0x78020fff
            TLB...................0x70020000 - 0x70023fff
```

The Data Cache Tags and Data are available via separate arrays. Each PPC440 cache line consists of 32 bytes of data. The D-Cache (Data) address used by XMD will access each cache line's data as a single large array, with each entry 32 bytes long. For example, `0x78000000` would be the address for data from one cache line, and `0x78000020` would be the address for data from another. Data cache tags are 2 words for each data cache entry. The D-Cache (TAG) data made available by XMD consumes 32 bytes per cache line (just as the data array). The first two words are the actual tag data, with 6 words of additional padding per entry. Refer to IBM PPC 440x5 CPU Core User's Manual for the format of the two tag words.

The "formula" to determine the address for cache data provided by XMD would be:

```
(xmd base address) + (cache way * (32 * 16)) + (cache set * 32)
```

So that to view (for example) cache set 15 way 1:

```
0x78000000 + (1 * 32 * 16) + (15 * 32) = 0x780003E0 (data)
0x78008000 + (1 * 32 * 16) + (15 * 32) = 0x780083E0 (TAG)
```

Given that there may be 64 ways to check for any given address of interest, this process lends itself to being scripted.

The TestApp_DCache application is used to investigate the PPC440 Data Cache. Download and run this application:

```
XMD% dow TestApp_DCache.elf
XMD% run
```

The program output is the following:

```
Application to assist with Data Cache study:

TLB Alias 0xB0000000 -> 0x00000000 (uncached) created


Data cache is presently disabled for 0x00000000

Wrote 32 bytes to 0x000043E0
DEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF
Wrote 32 bytes to 0x000143E0
DEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF
Wrote 32 bytes to 0x000242E0
DEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF
Wrote 32 bytes to 0x000343E0
DEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEFDEADBEEF

Data cache enabled for 0x00000000 (256MB)

Wrote 32 bytes to 0x000043E0
0000000000000001000000020000000300000004000000050000000600000007
Wrote 32 bytes to 0x000143E0
00000008000000090000000A0000000B0000000C0000000D0000000E0000000F
Wrote 32 bytes to 0x000242E0
0000001000000011000000120000001300000014000000150000001600000017
Wrote 32 bytes to 0x000343E0
0000002000000021000000220000002300000024000000250000002600000027
```

As the program output notes, it has enabled the data cache and written 32 bytes of data to four address ranges - `0x000043E0`, `0x000143E0`, `0x000242E0`, and `0x000343E0`. Each range begins on a cache line (the address is evenly divisible by 32). Each address range has been written to twice - the first time with the data cache inhibited (writing directly to memory), the second time with the data cache enabled for that address range.

The contents of the data cache are now viewed with an XMD TCL script.

```
XMD% stop
XMD% ppc440_dcache_print 0x78008000 0x78000000
Set Way TRA        Address  Drt Data
 7   0  0x00024200 000242e0 0xf 00000010000000110000001200000013
```

```
                                    00000014000000150000001600000017
15    0   0x00004200  000043e0  0xf 00000000000000010000000200000003
                                    00000004000000050000000600000007
15    1   0x00014200  000143e0  0xf 00000008000000090000000a0000000b
                                    0000000c0000000d0000000e0000000f
15    2   0x00034200  000343e0  0xf 00000020000000210000002200000023
                                    00000024000000250000002600000027
```

Note: This command should already be available if the user has followed the instruction in the "XMD and TCL Scripting" section.

The script has displayed the contents of the Data Cache. Each Set and Way within that set are displayed, along with the Tag Real Address (TRA) (address bits 0-22). The TRA and Set are used to calculate the address which has been cached, which is also displayed. The **Drt** field indicates whether or not this cache line is **dirty** (modified in the cache and not written back to memory), which is the case for all four entries.

One of the memory addresses written by the software is read with XMD to verify that the data match the output from the cache display script:

```
XMD% mrd 0x000143e0 8
   143E0:   00000008
   143E4:   00000009
   143E8:   0000000A
   143EC:   0000000B
   143F0:   0000000C
   143F4:   0000000D
   143F8:   0000000E
   143FC:   0000000F
```

The data match.

The application has performed various operations. Before caching was turned on, the application wrote one pattern to memory. Then, caching was enabled and a different pattern was written to the same memory locations. The cache was never flushed (data written back to memory). As was seen in the script output, the cache lines are all dirty. This means that the original pattern written to memory is still there.

Note: XMD "sees" memory just like software running on the processor -- through the TLB and Data Caches. Memory reads and writes performed on a cacheable memory region by XMD will affect cache contents.

The behavior does not lend itself to verifying that the contents of memory and the data cache match (for those instances when they **should** match -- when the cache line is not dirty).

To view the contents of actual memory some help is needed from software. A second TLB entry is set up for the same physical memory. The standalone BSP creates a 1:1 mapping at the time of initialization. This `0x00000000` (virtual) -> `0x00000000` (physical) mapping has been marked as caching enabled. TestApp_DCache also creates a mapping of `0xB0000000` (virtual) -> `0x00000000` (physical) which shall remain uncached. See Figure 4.

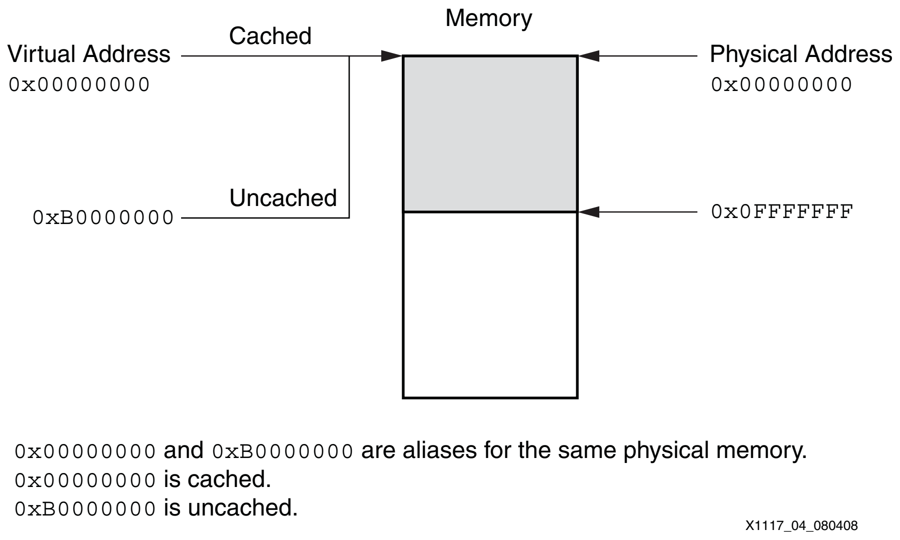


Figure 4: **Multiple TLB mappings for physical address `0x00000000`**

Verify that software has set the TLB up as described with the ppc440_read_tlb script:

```
XMD% ppc440_read_tlb 0x70020000
#  RPN        EPN        TS SIZE  ERPN U0U1U2U3 W I M G E UX UW UR SX SW SR
 0 0x00000000 0x00000000    256MB 0               I       UX UW UR SX SW SR
 1 0x00000000 0x00000000 TS 256MB 0                       UX UW UR SX SW SR
23 0x00000000 0xb0000000 TS 256MB 0               I       UX UW UR SX SW SR
...
```

It is seen that entry 23 and entry 1 both refer to the same RPN (physical address), Translation Space 1 (data). Entry 23 has the Cache **Inhibited** attribute set (cache disabled), entry 1 does not. Entry 0, which uses Translation Space 0 (instruction fetches) is not pertinent to the current discussion.

Use XMD to read the actual memory contents via the uncached memory alias. The memory address with the cached mapping is `0x000143E0`. To read this with the uncached alias, `0xB00143E0` is used:

```
XMD% mrd 0xB00143E0 8
B00143E0:   DEADBEEF
B00143E4:   DEADBEEF
B00143E8:   DEADBEEF
B00143EC:   DEADBEEF
B00143F0:   DEADBEEF
B00143F4:   DEADBEEF
B00143F8:   DEADBEEF
B00143FC:   DEADBEEF
```

As expected, the contents originally written to memory are seen. This technique may be useful to determine if memory + cache contents correspond in cases where an actual coherency bug is expected.

Conclusion

This application note has provided several debugging tools such as XMD TCL scripts to debug Xilinx peripherals, software to handle exceptions and display useful debugging information, and software to gather memory allocation statistics. Standard tools (generally referred to as *binutils*) such as **nm** and **objdump** were introduced to the user. The Xilinx Microprocessor Debugger (XMD) and the GNU Debugger (GDB) were used to debug several software errors.

References

1. UG111, *Embedded System Tools Reference Manual*
2. *IBM PPC 440x5 CPU Core User's Manual*
3. Stallman, Richard, Roland Pesch, Stan Schebs. Debugging with GDB. Boston: The Free Software Foundation, 2007
4. Ousterhout, John. *Tcl and the TK Toolkit*. Reading: Addison-Wesley Publishing Company, 1994.

Notice of Disclaimer

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
8/21/08	1.0	Initial Xilinx release.