XAPP1036 (v1.0) February 7, 2008

Introduction to Software Debugging on Xilinx PowerPC 405 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 PowerPC™ (PPC) 405 ML403 reference system:

www.xilinx.com/support/documentation/application_notes/xapp1036.zip

Introduction

This application note offers an introduction to software debugging of Xilinx PPC405 embedded processing platforms using XMD (Xilinx Processor Debugger) and GDB.

XMD is used to download executables to the system, control running these applications with breakpoints, and examine/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/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 ML403 Development Board
- Xilinx Platform USB Cable or Parallel IV Cable
- RS232 Cable
- Serial Communications Utility Program (e.g. HyperTerminal)
- Xilinx Platform Studio 9.2.01i
- Xilinx Platform Studio SDK 9.2.01i
- Xilinx Integrated Software Environment (ISE™) 9.2.03i

PPC405 Reference System Specifics

The included ML403 system was created with Base System Builder. It includes a PPC405, XPS UART 16550, XPS Interrupt Controller, XPS LL TEMAC Ethernet controller, Multi-Port Memory Controller (MPMC), XPS GPIO connected to on-board LEDs, XPS Multi-Channel Memory Controller (MCH EMC) connected to on-board flash, 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	0xFF000000	0xFF7FFFFF
DDR_SDRAM SDMA	mpmc	0x84600000	0x8460FFFF
RS232_Uart	xps_uart16550	0x83E00000	0x83E0FFFF
TriMode_MAC_GMII	xps_ll_temac	0x81C00000	0x81C0FFFF
xps_intc_0	xps_intc	0x81800000	0x8180FFFF
LEDs_4Bit	xps_gpio	0x81400000	0x8140FFFF
DDR_SDRAM	mpmc	0x00000000	0x03FFFFFF

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 source for these applications is provided within the `SDK_projects` directory.

The TestApp_Crash application will *crash* by intentionally causing the processor to execute invalid instructions. How to find the cause of application crashes is discussed using this application as a model.

TestApp_StackOverflow is used to again examine software crashes, this time providing a more realistic example of likely software errors. 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.

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/` 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 xapp1036.cmd
   ```

3. Invoke XMD and connect to the processor by the following command:

   ```
   $ xmd -opt xapp1036.opt
   ```

4. Download the executables by 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` inside XPS.
2. Use **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 ML403. 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 becomes evident when scripting is combined with XMDs abilities to access CPU registers and memory. XMD can access CPU registers interactively, as shown below:

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

This reads the PPC405 **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 PPC405 MSR Register and examine the EE bit.
# Print in plain English if External Exceptions(interrupts) are presently
# enabled.
proc ppc405_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 example script is provided in the xmd_tcl_scripts directory as ppc405_intenable_print.tcl. If presently in the ready_for_download directory, it would be loaded as shown below:

```
XMD% source ../../xmd_tcl_scripts/ppc405_intenable_print.tcl
```

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

```
XMD% ppc405_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, decoding many register fields. To utilize these scripts, copy ***dotxmdrc*** from the xmd_tcl_scripts directory to **$HOME/.xmdrc**.

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:
ppc405_print
mb_print
emaclite_print
lltemac_print
lltemac_read_phy
lldma_mm_print
uartlite_print
uartns550_print
xps_intc_print
XMD%
```

TestApp_Crash

Using XMD, download and run TestApp_crash (assumes bitstream is already downloaded):

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

If working properly, (in the present form it is not), the expected output would be:

```
-- Entering main() --
-- Exiting main() --
```

Identifying the Problem

When run in its present form, only the first line is printed correctly, which indicates that the application has *crashed*, thereby resulting in the appearance of only half 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 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 was determined shall be demonstrated below), which generates a PROGRAM exception on the Power PC405, which is vector `0x700`.

This means that the processor will execute instructions at offset `0x700` 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 EVPR Special Purpose Register. **The test application has not 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. Read the contents of the EVPR to see the base address of the Exception Vector table. Adding the PROGRAM vector `0x700` to this base will provide the address where the breakpoint must be set. In the below example, it is observed that the processor will execute instructions at `0x1E800700` when a PROGRAM exception occurs.

Note: The application has not initialized EVPR. The value used in this example, `0x1E800000`, may not reflect the value seen any other instance the same program is executed.

```
$ xmd
XMD% connect ppc hw
XMD% dow TestApp_Crash.elf
XMD% srrd evpr
  evpr: 1e800000
XMD% bps 0x1e800700 hw
Setting breakpoint at 0x1e800700
XMD% run
Info:Processor started. Type "stop" to stop processor
RUNNING> Info:Hardware Breakpoint 0 Hit, Processor Stopped at 0x1e800700
XMD%
```

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

The ppc405.tcl script, provided in the `xmd_tcl_scripts` directory, can provide some information about the cause of this crash. The commands it contains should already be available if the instructions in the "XMD and TCL Scripting" section of this application note have been completed.

Execute the "**`ppc405_print`**" procedure after the application hits the breakpoint as shown below:

```
XMD% ppc405_print
MSR:  0x00000000 PPriv
TCR:  0x00000000
TSR:  0xcc000000 ENW WIS
CCR0: 0x50700000 DPP1
ESR:  0x08000000 PIL
```

This script prints the MSR, TCR, TSR, CCR0, and ESR PowerPC405 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). PIL corresponds to *Program - Illegal Instruction*. The text (in this case, *PIL*) corresponds with the names used in the PowerPC 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 `SDK_projects/TestApp_Crash/src/TestApp_Crash.c` file:

```
unsigned crash_instructions[10];

/*
 * crash_function:
 * This function will generate an Illegal Instruction Program Exception
 * (PPC405 vector 0x700) 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
crash_function (void) {
   crash_instructions[0] = 0;
   crash_instructions[1] = 1;
   crash_instructions[2] = 2;
   crash_instructions[3] = 3;
   /*
    * Branch Unconditionally with no return to address of the array
    * crash_instructions[] (which contains only data, not valid instructions)
    * with the help of some inline PPC assembly code (the "b" instruction):
    */
   __asm__ __volatile__("b crash_instructions\n");
}
```

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

```
XMD% srrd srr0
  srr0: 000008e8
```

This is the address of the instruction (0x000008E8) which, when executed, generated the exception. To display the addresses of all symbols in the executable, use the command:

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

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

```
...
000008cc ? __tdata_start
000008d0 b object.3143
000008e8 B crash_instructions
00000910 T _boot0
00000910 T __boot0_start
...
```

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: 00000278
```

The symbol listing shows that the last function call was made from somewhere within main() -- the return address of `0x00000278` is greater than the start of main() at `0x00000230` and less than the end main() at `0x000002A4`:

```
...
000001d0 T crash_function
00000230 T main
000002a4 T __cpu_init
...
```

It could be helpful to see what function was called from within main(). To do this, disassemble the executable and examine the listing:

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

```
---
00000230 <main>:
 230:   94 21 ff f0     stwu    r1,-16(r1)
 234:   7c 08 02 a6     mflr    r0
 238:   93 e1 00 0c     stw     r31,12(r1)
 23c:   90 01 00 14     stw     r0,20(r1)
 240:   7c 3f 0b 78     mr      r31,r1
 244:   3c 60 83 e0     lis     r3,-31776
 248:   3c 80 02 fa     lis     r4,762
 24c:   60 84 f0 80     ori     r4,r4,61568
 250:   38 a0 25 80     li      r5,9600
 254:   48 00 00 f1     bl      344 <XUartNs550_SetBaud>
 258:   3c 60 83 e0     lis     r3,-31776
 25c:   60 63 10 0f     ori     r3,r3,4111
 260:   38 80 00 03     li      r4,3
 264:   48 00 02 29     bl      48c <XIo_Out8>
 268:   3d 20 00 00     lis     r9,0
 26c:   38 69 07 60     addi    r3,r9,1888
 270:   48 00 00 45     bl      2b4 <print>
 274:   4b ff ff 5d     bl      1d0 <crash_function>
 278:   3d 20 00 00     lis     r9,0
---
```

The last function call was crash_function() made from main() - the instruction immediately preceding the return address (the contents of the Link Register) `0x00000278` is "bl <offset to address of crash_function>".

The last function call displayed is not crash_instructions but a function call to crash_function. The PPC assembly instruction used in crash_function() to jump to the illegal instructions "b" is an unconditional branch with no return. It is analogous to a "goto" rather than a function call (like the `bl` or `branch with link` seen above). It is noteworthy that there is no history of this jump.

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 inline assembly code inside crash_function() which branched to the array crash_instructions.

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
```

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:

```
(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 evpr
evpr           0x1e800000       511705088
(gdb) hbreak *0x1e800700
Hardware assisted breakpoint 1 at 0x1e800700
```

Now, start the application and watch it crash:

```
(gdb) continue
Continuing.
Breakpoint 1, 0x1e800700 in ?? ()
```

Execution has stopped 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  0x1e800700 in ?? ()
#1  0x00000278 in main ()
```

It is seen that from somewhere in main() or a function called within main() has caused the application to crash. The numbers (#0, #1) 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.

Examine stack frame 1 (examine what happened in the function main()):

```
(gdb) frame 1
#1  0x00000278 in main () at ../src/TestApp_Crash.c:91
91 crash_function();
```

This indicates that the last thing which happened within the function main() was a call to crash_function(). GDB can be told to display the pertinent lines of source code from within main() with the **list** command:

```
(gdb) list 91
86          XUartNs550_SetBaud(XPAR_RS232_UART_BASEADDR,
XPAR_XUARTNS550_CLOCK_HZ, 9600);
87          XUartNs550_mSetLineControlReg(XPAR_RS232_UART_BASEADDR,
XUN_LCR_8_DATA_BITS);
88          print("-- Entering main() --\r\n");
89
90          /* This function will perform actions causing the application to
"crash" */
91          crash_function();
92
93          print("-- Exiting main() --\r\n");
94          return 0;
95      }
```

And, in TestApp_Crash.c at line 77 the offending call to crash_function() is seen.

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

```
(gdb) info registers srr0
srr0           0x8e8       2280
```

GDB will look up an address in the symbol table:

```
(gdb) info symbol 0x8e8
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             0x278       632
(gdb) info symbol 0x278
main + 72 in section .text
```

GDB will also disassemble instructions in memory, so it can be seen exactly what is at the address stored in the Link Register:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x00000230 <main+0>:    stwu    r1,-16(r1)
0x00000234 <main+4>:    mflr    r0
0x00000238 <main+8>:    stw     r31,12(r1)
0x0000023c <main+12>:   stw     r0,20(r1)
0x00000240 <main+16>:   mr      r31,r1
0x00000244 <main+20>:   lis     r3,-31776
0x00000248 <main+24>:   lis     r4,762
0x0000024c <main+28>:   ori     r4,r4,61568
0x00000250 <main+32>:   li      r5,9600
0x00000254 <main+36>:   bl      0x344 <XUartNs550_SetBaud>
0x00000258 <main+40>:   lis     r3,-31776
0x0000025c <main+44>:   ori     r3,r3,4111
0x00000260 <main+48>:   li      r4,3
0x00000264 <main+52>:   bl      0x48c <XIo_Out8>
0x00000268 <main+56>:   lis     r9,0
0x0000026c <main+60>:   addi    r3,r9,1888
```

```
0x00000270 <main+64>:   bl      0x2b4 <print>
0x00000274 <main+68>:   bl      0x1d0 <crash_function>
0x00000278 <main+72>:   lis     r9,0
```

And again, it is observed that the instruction preceding `0x00000278` (main + 72) is a call to crash_function()

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 crash_function has been identified. To prevent TestApp_Crash from causing an exception, remove the inline assembly branch instruction 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, 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 additional detail later in this section.

Identifying the Problem

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

Lets examine the application TestApp_StackOverflow. The skeleton of the application is something like this:

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

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

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

Connect to the XMD GDB server:

```
(gdb) target remote 149.199.109.40:1234
Remote debugging using 149.199.109.40:1234
0xfffffffc in ?? ()
```

Have GDB download the executable to memory:

```
(gdb) load
```

Set a hardware breakpoint at the location observed in EVPR + `0x700` :

```
(gdb) hbreak *0x1e800700
Hardware assisted breakpoint 1 at 0x1e800700
```

Continue program execution from the present PC:

```
(gdb) c
Continuing.
Breakpoint 1, 0x1e800700 in ?? ()
```

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  _vector0700 () at xvectors.S:432
#1  0x000022b4 in crash_function ()
#2  0x00002294 in frame_dummy ()
#3  0x000023ec in myfunction1 ()
#4  0x00002458 in main ()
```

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

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

```
(gdb) info frame 3
Stack frame at 0x7f68:
 pc = 0x23ec in myfunction1; saved pc 0x2458
 called by frame at 0x7f78, caller of frame at 0x7f58
 Arglist at 0x7f48, args:
 Locals at 0x7f48, Previous frame's sp is 0x7f68
 Saved registers:
  r31 at 0x7f64, pc at 0x7f6c, lr at 0x7f6c
```

This shows that myfunction1() left its frame at pc `0x23EC`. Disassemble the function to see what instruction is at `0x23EC`:

```
(gdb) disassemble myfunction1
Dump of assembler code for function myfunction1:
0x000023c8 <myfunction1+0>:     stwu    r1,-16(r1)
0x000023cc <myfunction1+4>:     mflr    r0
0x000023d0 <myfunction1+8>:     stw     r31,12(r1)
0x000023d4 <myfunction1+12>:    stw     r0,20(r1)
0x000023d8 <myfunction1+16>:    mr      r31,r1
0x000023dc <myfunction1+20>:    lis     r9,0
0x000023e0 <myfunction1+24>:    addi    r3,r9,25592
0x000023e4 <myfunction1+28>:    bl      0x327c <puts>
0x000023e8 <myfunction1+32>:    bl      0x2380 <myfunction2>
0x000023ec <myfunction1+36>:    lis     r9,0
```

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

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_StackOverflow.

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

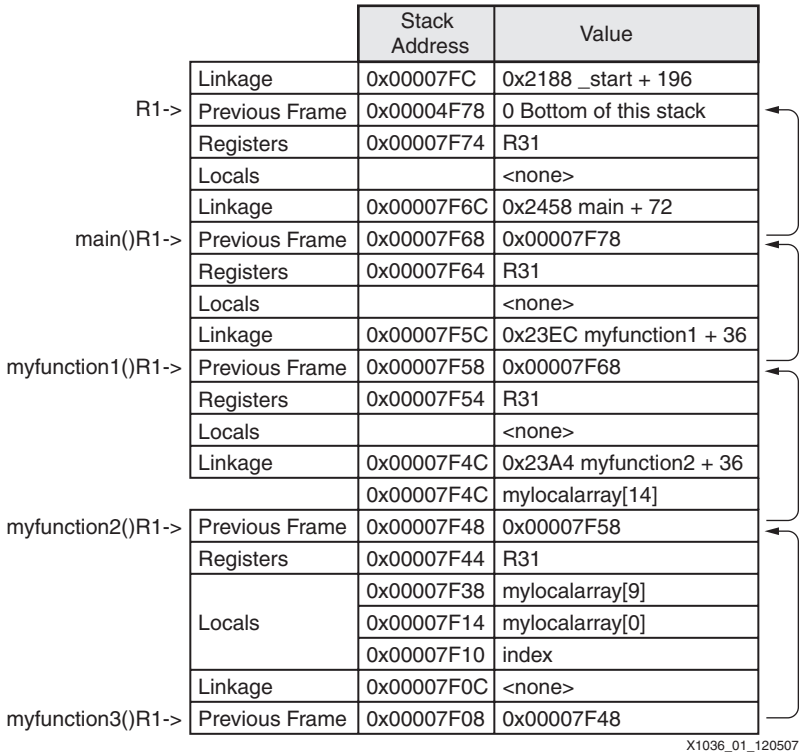

Figure 1: **TestApp_StackOverflow Stack**

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_StackOverflow.elf

void
myfunction3 (void) {
# At call time, R1(the stack pointer) = 0x00007F08 + 64
# store word and update (stwu) the stack pointer.
# *sp = 0x00007F48; sp = sp - 64bytes;
    2310:       94 21 ff c0     stwu    r1,-64(r1)
# Copy the contents of the link register to r0
    2314:       7c 08 02 a6     mflr    r0
# Store contents of R31 on the stack at 0x00007F08 + 60 (0x00007F44)
    2318:       93 e1 00 3c     stw     r31,60(r1)
# Store the contents of the LR at 0x00007F08 + 68 (0x00007F4c)
# This offset is within the caller's frame
    231c:       90 01 00 44     stw     r0,68(r1)
# copy stack pointer to R31
    2320:       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");
```

```
    2324:       3d 20 00 00     lis     r9,0
    2328:       38 69 63 78     addi    r3,r9,25464
    232c:       48 00 0f 51     bl      327c <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.
    2330:       38 00 00 0e     li      r0,14
    2334:       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]      = 0x00007F4C
# the link address register was saved at 0x00007F4C
# the data saved in this array has overwritten the LR -- stack corruption.
# when this function returns, it will now branch to crash_function()
    mylocalarray[index] = (unsigned) crash_function;
    2338:       81 7f 00 08     lwz     r11,8(r31)
    233c:       3d 20 00 00     lis     r9,0
    2340:       38 09 22 94     addi    r0,r9,8852
    2344:       7c 0a 03 78     mr      r10,r0
    2348:       55 69 10 3a     rlwinm  r9,r11,2,0,29
    234c:       38 1f 00 08     addi    r0,r31,8
    2350:       7d 29 02 14     add     r9,r9,r0
    2354:       39 29 00 04     addi    r9,r9,4
    2358:       91 49 00 00     stw     r10,0(r9)

    printf("-- Exiting myfunction3() --\r\n");
    235c:       3d 20 00 00     lis     r9,0
    2360:       38 69 63 98     addi    r3,r9,25496
    2364:       48 00 0f 19     bl      327c <puts>
}
# Load R11 with previous stack pointer
    2368:       81 61 00 00     lwz     r11,0(r1)
# Get saved link register from the stack
    236c:       80 0b 00 04     lwz     r0,4(r11)
# move saved LR value to the link register
    2370:       7c 08 03 a6     mtlr    r0
# place saved R31 value into R31
    2374:       83 eb ff fc     lwz     r31,-4(r11)
# restore previous stack pointer
    2378:       7d 61 5b 78     mr      r1,r11
# branch to the link register, returning from this function.
# this will branch to crash_function()
    237c:       4e 80 00 20     blr
```

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

mylocalarray[index] = (unsigned) crash_function;

This appears to be a perfectly 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 past 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 and/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_StackOverflow application there is provided a **minimal** example of crash debugging software. The project file, `ppc405_crashtrace.c`, contains code to set up a handler for this exception and print out useful information when a crash occurs. The necessary setup is presently commented out:

```
int
main (void) {

#ifdef SETUP_CRASHTRACE
   setup_crashtrace();
#endif
```

The **#ifdef** is removed, ppc405_crashtrace.c is added to the project, and the application recompiled. Program output is now:

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

STACK MAXIMUM DEPTH: 0x1B0 bytes out of 0x1000 total.
============================================================
Illegal Instruction Program Exception at: 0x00006E38
FRAME: 0x00007E60 RETURN 0x00002650
FRAME: 0x00007E98 RETURN 0x00000778
FRAME: 0x00007F38 RETURN 0x000022B4
FRAME: 0x00007F48 RETURN 0x00002294
FRAME: 0x00007F58 RETURN 0x000023EC
FRAME: 0x00007F68 RETURN 0x0000245C
FRAME: 0x00007F78 RETURN 0x00002188
============================================================
```

This information should be saved to a text file 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_StackOverflow-crashtrace.elf >
TestApp_StackOverflow-crashtrace.sym
```

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

```
$ xmd -tcl ../../xmd_tcl_scripts/stackscan.tcl TestApp_StackOverflow-
crashtrace.crash TestApp_StackOverflow-crashtrace.sym
EXCEPTION ADDR: 0x00006E38 crash_instructions
FRAME:          0x00002650 ProgramExceptionHandler+324
FRAME:          0x00000778 _vector0700+120
FRAME:          0x000022B4 crash_function+32
FRAME:          0x00002294 crash_function
FRAME:          0x000023EC myfunction1+36
FRAME:          0x0000245C main+76
FRAME:          0x00002188 __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_StackOverflow-crashtrace.crash TestApp_StackOverflow-
crashtrace.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 crash_function, which seems to have been called by myfunction1(). It is known from examining the source that myfunction1() never calls crash_function(). 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 crash_function or the variable crash_instructions.

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.

SDK 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 SDK compiler options configuration to create a wrapper for malloc() and free(). These are used in the provided application TestApp_malloc.

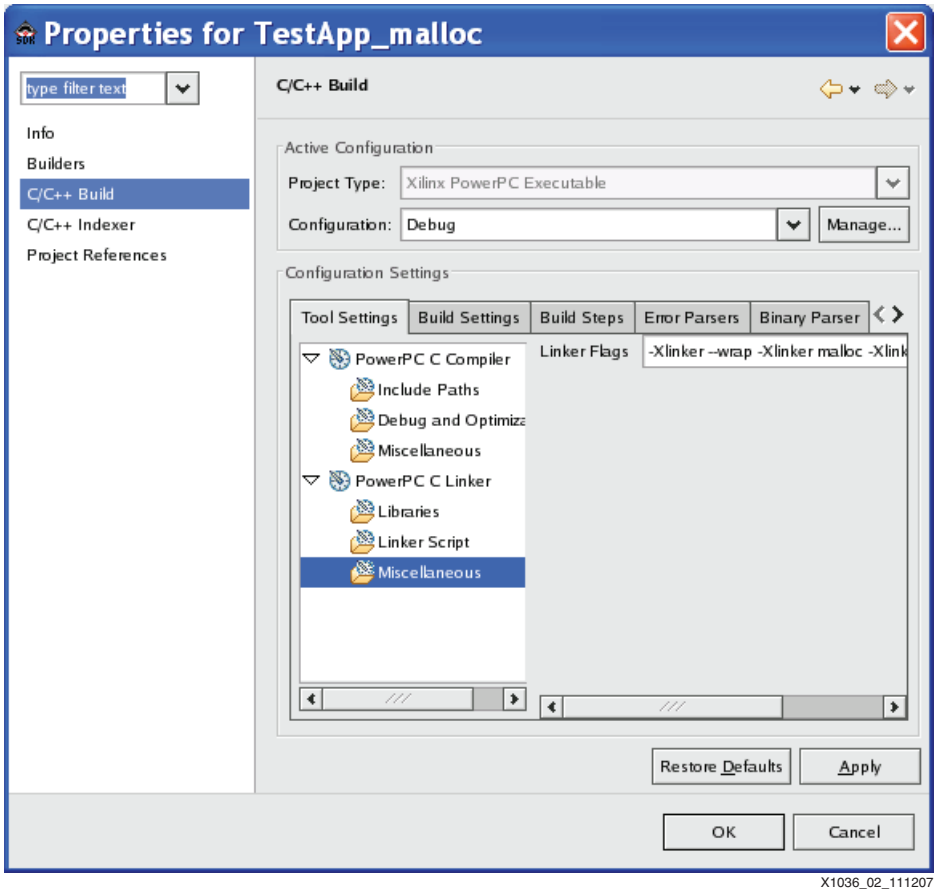

Figure 2: **Setting the Linker Wrapper Option**

TestApp_malloc

TestApp_malloc provides code useful in locating memory leaks. This is done by building upon lessons learned with TestApp_StackOverflow. 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.

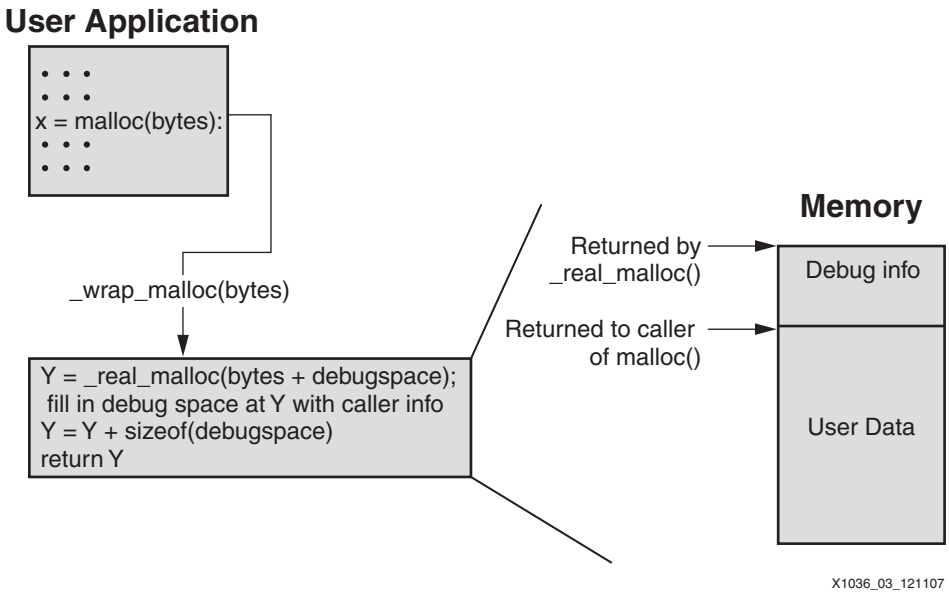

Figure 3: **Malloc Wrapper Flow**

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/TestApp_malloc.c:118 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: 0x000002e8 TOTAL BYTES:     2610 ALLOCS:    45 FREES     0
MALLOC CALLER: 0x000003d4 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:

```
$ powerpc-eabi-nm --numeric-sort TestApp_malloc.elf > TestApp_malloc.sym
Contents of generated TestApp_malloc.sym file:
...
000001d0 T MemWaster1
000002c0 T MemWaster2
00000368 T MemWaster3
0000043c T MemWaster4
000004c8 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+40 presently has 2610 bytes allocated
MALLOC CALLER 0x000003d4 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

Now 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 is *hung*.

Explanation of the Problem

To investigate, run the application with GDB.

```
$ powerpc-eabi-gdb -nw TestApp_temac.elf
(gdb) target remote 149.199.109.40:1234
(gdb) load
(gdb) c
Continuing.
```

Now, wait for the application to progress to the point where it *hangs*. At this point, enter `^C` to interrupt the program. Program execution will stop, and GDB will display a prompt again:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x000005b0 in _vector0500 () at xvectors.S:426
426             non_critical_interrupt 0500, 5
```

The application seems to have been forever stuck in an interrupt handler. Remember that vector `0x500` is the PPC405 External Interrupt handler.

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.

Examine what this interrupt has interrupted:

```
(gdb) bt
#0  0x000005b0 in _vector0500 () at xvectors.S:426
#1  0x00002d04 in TemacSgDmaIntrSingleFrameExample ()
#2  0x00002730 in TemacSgDmaIntrExample ()
#3  0x00002304 in main ()
```

So, execution of the application was interrupted somewhere inside of TemacSgDmaIntrSingleFrameExample(). 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).

Excerpt from TestApp_temac.dis:

```
        /*
         * Examine the TxBDs.
         *
        * There isn't much to do. The only thing to check would be DMA exception
         * bits. But this would also be caught in the error handler. So we just
         * return these BDs to the free list
         */
        Status = XLlDma_BdRingFree(TxRingPtr, 2, Bd1Ptr);
    2cf0:       80 1f 00 28     lwz     r0,40(r31)
    2cf4:       80 7f 00 14     lwz     r3,20(r31)
    2cf8:       38 80 00 02     li      r4,2
    2cfc:       7c 05 03 78     mr      r5,r0
    2d00:       48 00 1c b1     bl      49b0 <XLlDma_BdRingFree>
    2d04:       7c 60 1b 78     mr      r0,r3
    2d08:       90 1f 00 24     stw     r0,36(r31)
        if (Status != XST_SUCCESS) {
    2d0c:       80 1f 00 24     lwz     r0,36(r31)
    2d10:       2f 80 00 00     cmpwi   cr7,r0,0
    2d14:       41 9e 00 1c     beq-    cr7,2d30
<TemacSgDmaIntrSingleFrameExample+0x504>
                TemacUtilErrorTrap("Error freeing up TxBDs");
    2d18:       3d 20 00 01     lis     r9,1
    2d1c:       38 69 2f b0     addi    r3,r9,12208
    2d20:       48 00 1b 11     bl      4830 <TemacUtilErrorTrap>
                return XST_FAILURE;
    2d24:       38 00 00 01     li      r0,1
    2d28:       90 1f 00 44     stw     r0,68(r31)
    2d2c:       48 00 00 f4     b       2e20
<TemacSgDmaIntrSingleFrameExample+0x5f4>
        }
```

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

```
        /*
         * Examine the TxBDs.
         *
        * There isn't much to do. The only thing to check would be DMA exception
         * bits. But this would also be caught in the error handler. So we just
         * return these BDs to the free list
         */
        Status = XLlDma_BdRingFree(TxRingPtr, 2, Bd1Ptr);
        if (Status != XST_SUCCESS) {
                TemacUtilErrorTrap("Error freeing up TxBDs");
                return XST_FAILURE;
        }
```

At xlltemac_example_intr_sgdma.c:633, a branch to XLlDma_BdRingFree() was taken. This information is useful because it tells us exactly how far our application has progressed — what has happened and 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:614

```
        while (!FramesTx);
```

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

```
(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 like 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. Run the lldma_mm_print procedure, providing the base address of the SDMA device as indicated in the system memory map:

```
XMD% lldma_mm_print 0x84600000
TX Ring
0x84600000 NDESC 0x00014c40
0x84600004 BUFA  0x00014396
0x84600008 BUFL  0x00000000
0x8460000c CDESC 0x00014c40
0x84600010 TDESC 0x00014c00
0x84600014 CR    0x01010000 IRQ_TMO_0x1 IRQ_THR_0x1 MSB_ADDR_0x0
0x84600018 IRQ   0x00010000
0x8460001c SR    0x00000014 COMPLETED EOP
RX Ring
0x84600020 NDESC 0x00016c00
0x84600024 BUFA  0x00014986
0x84600028 BUFL  0x000001f8
0x8460002c CDESC 0x00016c00
0x84600030 TDESC 0x00016bc0
0x84600034 CR    0x01010087 IRQ_TMO_0x1 IRQ_THR_0x1 IRQ_EN IRQ_ERROR_EN
IRQ_DELAY_EN IRQ_COALESCE_EN
0x84600038 IRQ   0x00010401 COALESCE
0x8460003c SR    0x0000001c COMPLETED SOP EOP

0x84600040 DMACR 0x0000001c RX_OVF_ERR_DIS TX_OVF_ERR_DIS TAIL_PTR_EN
```

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. Software has not processed this waiting packet (shown above, FramesRx = 0). Refer to the MPMC 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 ppc405_0/libsrc/standalone_v1_00_a/src/xvectors.S) has been installed at `0x500` (the external interrupt vector) because that is where the debugger stopped.

First, instruct GDB to print in a human readable form:

```
(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
$4 = {{
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x7ca4 <XIntc_InterruptHandler>,
    DataPtr = 0x14b80,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  }, {
    Handler = 0x59dc <NullHandler>,
    DataPtr = 0x0,
    ReadOnlySDA = 0x1b930,
    ReadWriteSDA = 0x1bf08
  } <repeats 11 times>}

(gdb)
```

At array index 5, which corresponds to the external interrupt vector `0x500`, 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 `0x500`. 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
$5 = {
  DeviceId = 0x0,
  BaseAddress = 0x81800000,
  AckBeforeService = 0x0,
  Options = 0x1,
  HandlerTable = {{
      Handler = 0x3b3c <TxIntrHandler>,
      CallBackRef = 0x18bc4
    }, {
      Handler = 0x7f20 <StubHandler>,
      CallBackRef = 0x14b80
    }, {
      Handler = 0x3d4c <TemacErrorHandler>,
      CallBackRef = 0x18c5c
    }, {
      Handler = 0x7f20 <StubHandler>,
      CallBackRef = 0x14b80
    }}
}
```

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_DDR_SDRAM_SDMA2_TX_INTOUT_INTR 0
#define XPAR_XPS_INTC_0_DDR_SDRAM_SDMA2_RX_INTOUT_INTR 1
#define XPAR_XPS_INTC_0_TRIMODE_MAC_GMII_TEMACINTC0_IRPT_INTR 2
#define XPAR_XPS_INTC_0_RS232_UART_IP2INTC_IRPT_INTR 3
```

An entry in HandlerTable[1] is expected for receive interrupts. Examine the callback data for handler that is there, StubHandler. This is of type XIntc:

```
(gdb) p *(XIntc*) XIntc_ConfigTable.HandlerTable[1].CallBackRef
$10 = {
  BaseAddress = 0x81800000,
  IsReady = 0x11111111,
  IsStarted = 0x22222222,
  UnhandledInterrupts = 0x87b8b,
  CfgPtr = 0x13950
}
```

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 is *hung* 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.

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. UG011, *PowerPC Processor Reference Guide*
2. UG111, *Embedded System Tools Reference 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.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
2/7/08	1.0	Initial Xilinx release.