XAPP1137 (v1.0) June 9, 2009

Linux Operating System Software Debugging Techniques with Xilinx Embedded Development Platforms

Author: Brian Hill

Abstract

This application note discusses Linux Operating System debugging techniques. Debugging boot issues, kernel panics, software and hardware debuggers, driver <-> application interaction, and various other tools are discussed.

Included Systems

Included with this application note is one reference system built for the Xilinx ML507 Rev A board. The reference system is available in the following ZIP file available at:

https://secure.xilinx.com/webreg/clickthrough.do?cid=125262

Introduction

This application note discusses Linux Operating System debugging techniques. Software applications and drivers are included to present debugging material.

General Topics

The application note discusses debugging Linux boot issues, and the use of various Linux debugging facilities such as *strace*.

Included Software

buggy_core

The buggy_core application is used to demonstrate techniques for debugging application crashes. The use of gdb, gdbserver, and coredump analysis are discussed.

buggy_memleak

The buggy_memleak application allocates memory with malloc() and never frees it. This application is used to discuss memory leak detection and debugging techniques with the mtrace facility and the Linux `/proc` filesystem.

buggy_drv

A kernel driver 'buggy_drv' is provided as an example of debugging kernel bugs.

buggy_allstrings

The buggy_allstrings application operates with the buggy_drv driver.

buggy_choosestring

The buggy_choosestring application operates with the buggy_drv driver.

Intended Audience

This application note is best suited to users who are comfortable configuring, building, and booting Linux on a Xilinx embedded platform.

Hardware and Software Requirements

The hardware requirements for this reference system are:

- Xilinx ML507 Rev A board
- Xilinx Platform USB or Parallel IV programming cable
- RS232 serial cable and serial communication utility (HyperTerminal)
- Xilinx Platform Studio 11.1
- Xilinx Integrated Software Environment (ISE®) 11.1
- MontaVista Linux 5.0 (Linux kernel 2.6.24)

Reference System Specifics

See Table 1 for the address map of the system.

Address Map

Table 1: **Reference System Address Map**

Peripheral	Instance	Base Address	High Address
ppc440mc_ddr2	DDR2_SDRAM	0x00000000	0x0FFFFFFF
xps_gpio	Push_Buttons_5Bit	0x81400000	0x8140FFFF
xps_iic	IIC_EEPROM	0x81600000	0x8160FFFF
xps_intc	xps_intc_0	0x81800000	0x8180FFFF
xps_ll_temac	Hard_Ethernet_MAC	0x81C00000	0x81C0FFFF
xps_uart16550	RS232_Uart_1	0x83E00000	0x83E0FFFF
xps_bram_if_cntlr	xps_bram_if_cntlr_1	0xFFFFC000	0xFFFFFFFF

Executing the Reference System

Using HyperTerminal or a similar serial communications utility, map the operation of the utility to the physical COM port to be used. Then connect the UART of the board to this COM port. Set the HyperTerminal to the Bits per second to **9600**, Data Bits to **8**, Parity to **None,** and Flow Control to **None**. The settings are shown in Figure 1. This is necessary to see the results from the software application.

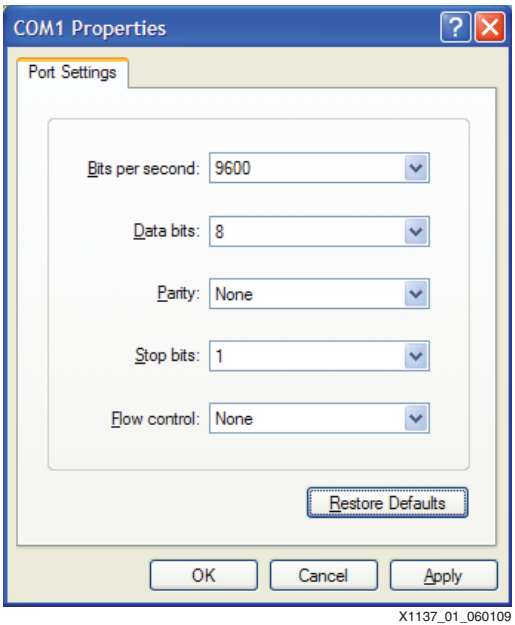

Figure 1: **HyperTerminal Settings**

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

To execute the system using files in 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 command:

   ```
   impact -batch xapp1137.cmd
   ```

3. Invoke XMD and connect to the PowerPC® 440 processor by using the following command:

   ```
   xmd -opt xapp1137.opt
   ```

4. Download the executable by using the following command depending on the software application:
   - **`dow zImage.initrd`**
5. Enter in the `run` command to run the software application.

Executing the Reference System from XPS for Hardware

To execute the system for hardware using XPS, follow these steps:

1. Open `system.xmp` in XPS.
2. Select **Hardware→Generate Bitstream** to generate a bitstream for the system.
3. Select **Device Configuration→Download Bitstream** to download the bitstream.
4. Select **Debug→Launch XMD to i**nvoke XMD.
5. Download the executable file by using the following command depending on the software application:
   - **`dow zImage.initrd`**
6. Enter in the `run` command to run the software application.

XMD and TCL Scripting

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
hexdump
ppc405_print
ppc405_rm_sa_print
ppc405_read_tlb
ppc405_dcache_print
ppc440_print
ppc440_read_tlb
ppc440_dcache_print
ppc440_dcache_match
ppc_bt
ppc_dis
mb_print
mb_bt
emaclite_print
lltemac_print
lltemac_read_phy
lltemac_stats
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.

Patch the kernel

The 2.6.24 Linux kernel does not initialize the PowerPC processor `DBCR0` register. The value placed here by XMD is not suitable for use with Linux. The Linux kernel is patched for proper operation. Edit `<Linux> /arch/powerpc/kernel/head_44x.S` and make the additions shown in **red**:

```
SET_IVOR(15, Debug);

/* Establish the interrupt vector base */
lis r4,interrupt_base@h /* IVPR only uses the high 16-bits */
mtspr   SPRN_IVPR,r4
```

```
#if !defined(CONFIG_BDI_SWITCH)
    /*
     * The Abatron BDI JTAG debugger does not tolerate others
     * mucking with the debug registers.
     */
    lis     r2,DBCR0_IDM@h
    mtspr   SPRN_DBCR0,r2
    isync
    /* clear any residual debug events */
    li      r2,-1
    mtspr   SPRN_DBSR,r2
#endif

    /*
     * This is where the main kernel code starts.
     */

    /* ptr to current */
    lis r2,init_task@h
    ori r2,r2,init_task@l
```

The system provided with this application note include the XPS Local Link Tri-Mode Ethernet core version 2.00.a. The Linux 2.6.24 kernel must be patched for proper operation with this version of the core. Edit `<Linux>/drivers/net/xilinx_lltemac/xlltemac_main.c` and make the modifications shown in **red**:

```
#define BdGetRxLen(BdPtr) \
    (XLlDma_mBdRead((BdPtr), XLLDMA_BD_USR4_OFFSET) & 0x3FFF)
```

Build a kernel image

The user will build a `zImage.initrd` executable using the steps shown. This image will **not** successfully boot. The cause of the problem will be analyzed in "Debugging Kernel Boot Issues".

Build zImage.initrd

Copy the provided ramdisk, kernel configuration, and device tree to the kernel tree.

1. cp <edk project>/ready_for_download/dotconfig <Linux> /.config
2. cp <edk project>/ready_for_download/ml507.dts <Linux>/arch/powerpc/boot/dts/ml507.dts
3. cp <edk project>/ready_for_download/ramdisk.image.gz <Linux>/arch/powerpc/boot/ramdisk.image.gz
4. cp -a <work area>/buggy_drv <Linux>/drivers/char/
5. Add the following to <Linux>/drivers/char/Kconfig:

```
config BUGGY_DRV_EXAMPLE
        bool "Buggy Driver Example"
        help
          Example driver to demonstrate kernel debugging.
```

6. Add the following to <Linux>/drivers/char/Makefile

```
obj-$(CONFIG_BUGGY_DRV_EXAMPLE) += buggy_drv/
```

7. cd <Linux>
8. make ARCH=powerpc menuconfig
9. Enable Device Drivers -> Character devices -> Buggy Driver Example
10. make ARCH=powerpc zImage.initrd

Note: A previously generated image, zImage.initrd-noboot is provided in the `ready_for_download` area.

Debugging Kernel Boot Issues

Users frequently encounter issues booting the Linux kernel. This can be due to an improper kernel configuration, inconsistencies in the hardware description, an improperly created Root File System, and various other causes. Boot issues can be difficult to debug because many of the tools generally available to debug Linux issues can not be used during the boot process.

This application note utilizes a simple Linux image with a compressed kernel, ramdisk, and device tree all bundled in a single ELF file. In MontaVista 5.0, this file is named `zImage.initrd`.

Note: The name of this bootable image varies with different Linux distributions.

To successfully debug boot issues, the user should understand how a zImage is booted. Refer to Figure 2 for this topic.

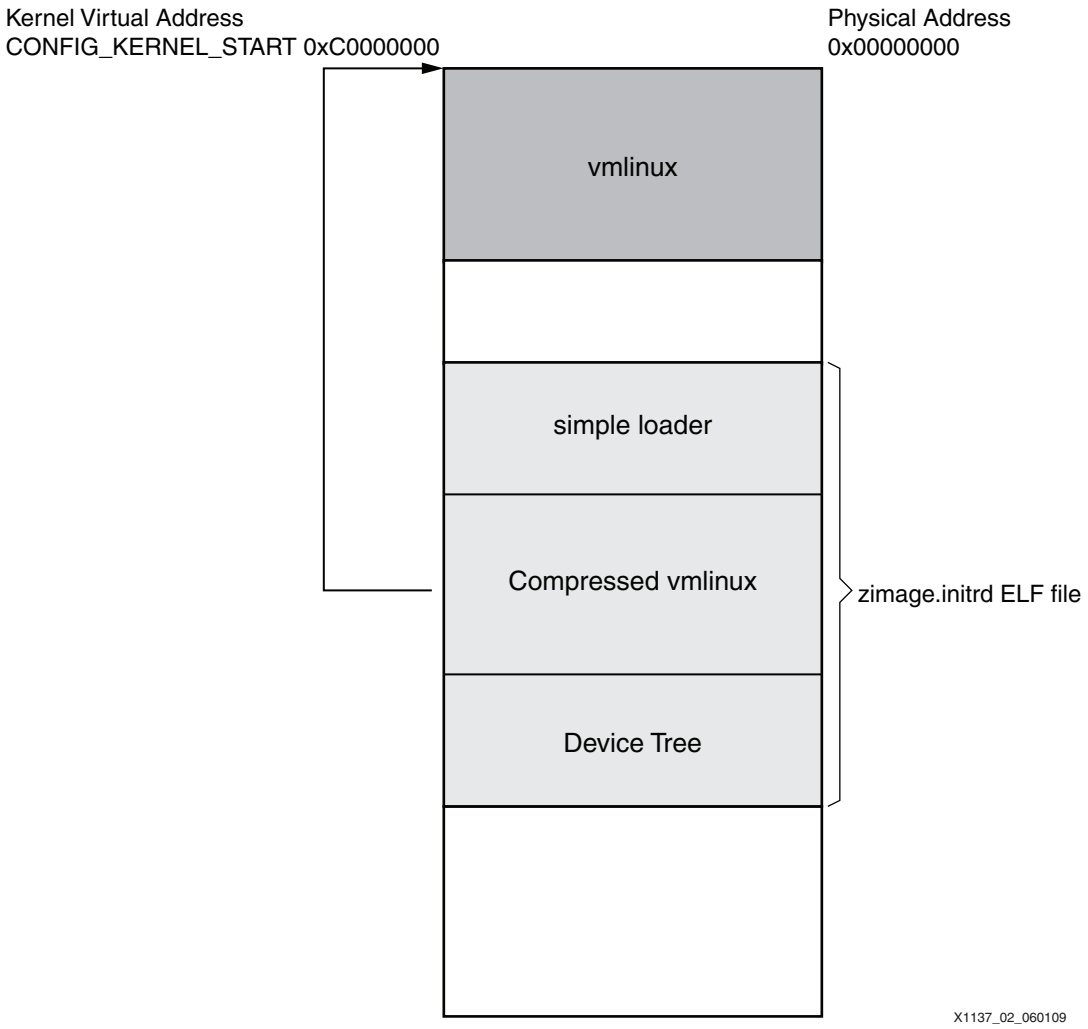

Figure 2: **Linux PowerPC Processor zImage Boot Process**

Booting a zImage

Note: This document describes a Xilinx ML507 system using a PowerPC 440 processor with MontaVista Linux 5.0 (Linux 2.6.24). Other processors, Linux distributions, and kernel versions will be similar but may vary.

The `zImage.initrd` executable is a simple Linux loader which contains the compressed kernel and device tree images embedded within it. The loader does not perform any TLB configuration; a 1:1 TLB mapping must already have been performed elsewhere (XMD performs this function). The function of the loader is to perform some minimal hardware setup

and uncompress the vmlinux (kernel) image to the start of physical memory -- 0x00000000 and launch it.

The absolute entry point of the loader is _start, found in arch/powerpc/boot/crt0.S. Where the zImage.initrd executable is linked depends upon the size of the kernel image embedded within it. A typical link location for the start of the image would be 0x04000000, but addresses such as 0x08000000, etc... can be seen. The address is aligned to the 4MB boundary beyond the uncompressed kernel size used.

Experiencing a problem with booting Linux may not be a problem with the kernel at all; the error could have occurred in the loader, before the kernel was even launched. Control is given to the kernel proper at the end of start() in arch/powerpc/boot/main.c, where the loader branches to 0x00000000. The first instructions of the kernel proper are in _start, located in arch/powerpc/kernel/head_44x.S. The kernel is actually linked to 0xC0000000 (CONFIG_KERNEL_START), not 0x00000000. The kernel will create a mapping of 0xC0000000 -> 0x00000000 very early in its initialization. The first 'C' function found in the kernel is start_kernel() in init/main.c

It is vital to consider that there are two ELF executables involved, with two separate symbol tables. The zImage.initrd bootable image found in <Linux>/arch/powerpc/boot/ and the kernel itself <Linux>/vmlinux. This point is discussed when symbolic kernel debugging is presented.

Download and run the zImage

Download and run the generated zImage.initrd using XMD, following the steps outlined in "Executing the Reference System".

```
XMD% dow zImage.initrd
XMD% run
```

Note: If the user has not completed the steps in "Build a kernel image" a prebuilt image zImage.initrd-noboot has been provided in the ready_for_download area.

The console displays the output shown, and nothing more. Linux has failed to boot properly.

```
booting virtex
memstart=0x10
memsize=0xf

zImage starting: loaded at 0x00400000 (sp: 0x00899eb8)
Allocating 0x308428 bytes for kernel ...
gunzipping (0x00000000 <- 0x0040d000:0x0056b234)...done 0x2e63d4 bytes
Attached initrd image at 0x0056c000-0x00898e18
initrd head: 0x1f8b0808

Linux/PowerPC load: console=ttyS0,9600 ip=192.168.0.10 root=/dev/ram rw
Finalizing device tree... flat tree at 0x8a6300
```

None of the messages generated were produced by the Linux kernel. The last message seen, "Finalizing device tree..." was printed by the loader in start(), which is found in <Linux>/arch/powerpc/boot/main.c. Nothing seen so far indicates that the loader has even launched the Linux kernel.

The Program Counter register is examined.

```
XMD% rrd pc
    pc: c00098e0
```

The last instruction executed was at memory location 0xC00098E0. The **nm** command is used to display all symbols in an object file with the addresses chosen by the linker.

```
$ cd <Linux>/arch/powerpc/boot/
$ ppc_440-nm --numeric-sort zImage.initrd > zImage.initrd.sym
```

The file `zImage.initrd.sym` is created. It is seen that symbols in this executable start at `0x00400000` and extend approximately 4MB beyond this start. Kernel virtual address space begins at `0xC0000000`. It is very likely that the loader has completed its task and that the failure occurred while executing kernel code, not the loader. The symbols from `vmlinux` must be examined, not those in `zImage.initrd`. A symbol listing is automatically generated when the kernel is built and can be found in `<Linux>/System.map`.

Note: `System.map-noboot` is provided in the `ready_for_download` area.

Examining `System.map`, it is seen that `0xC00098E0` is somewhere within the body of the `__delay()` procedure.

```
c00098b4 T __plb_disable_wrp
c00098c8 T __delay
c00098f8 T udelay
```

This function does not indicate what the problem is. Use the **`ppc_bt`** script to display a function called *backtrace* by examining what is on the stack.

Note: The script should already be available if the user has completed the steps in "XMD and TCL Scripting".

```
XMD% ppc_bt
PC:   0xc00098e0
LR:   0xc001f030
R1:   0xc02e3d20 0xc001f000
R1:   0xc02e3d70 0xc0023b08
R1:   0xc02e3db0 0xc000b09c
R1:   0xc02e3de0 0xc000b34c
R1:   0xc02e3ed0 0xc000d7e4
R1:   0xc02e3f90 0xc02a7224
R1:   0xc02e3fb0 0xc02a27c0
```

The user could manually use the **`nm`** command as already described to match these addresses to source code. The **`ppc_bt`** script will automatically attempt to match addresses to symbols in the most recently downloaded elf executable, in this case `zImage.initrd`. Since kernel symbols are not present in this elf, the user must specify that `vmlinux` is to be used for symbols. If the user is presently in the `<Linux>/arch/powerpc/boot` directory, the following command is issued:

```
XMD% set elf_file ../../../vmlinux
```

Obtain a symbolic back trace

```
XMD% ppc_bt
PC:   0xc00098e0             __delay           time.h:97
LR:   0xc001f030             panic             panic.c:145
R1:   0xc02e3d20 0xc001f000 panic             panic.c:145
R1:   0xc02e3d70 0xc0023b08 do_exit           exit.c:987
R1:   0xc02e3db0 0xc000b09c kernel_bad_stack  traps.c:1142
R1:   0xc02e3de0 0xc000b34c _exception        traps.c:187
R1:   0xc02e3ed0 0xc000d7e4 ret_from_except_full entry_32.S:644
R1:   0xc02e3f90 0xc02a7224 xilinx_intc_init_tree xilinx_intc.c:143
R1:   0xc02e3fb0 0xc02a27c0 init_IRQ          irq.c:343
XMD%
```

A serious error has caused the kernel to panic(). The approximate location of the error appears to be in the function xilinx_intc_init_tree() located in `xilinx_intc.c` line 143. Note that this is an *approximate* location for several reasons. The kernel can not be compiled without optimization. The minimum optimization level which can be used is -O1. The higher the compiler optimization used, the more difficult it is to match an assembly instruction to a line of source code.

When a kernel panic occurs, Linux is expected to display some debugging information on the console. No message has appeared. This can occur when critical errors happen before Linux is sufficiently initialized to send text to the console. Even though no message has appeared, a message should still have been placed in the kernel syslog buffer.

Using GDB to debug kernel boot problems

XMD provides a GDB Server. This server is not suitable for general purpose Linux debugging for a variety of reasons, but it is still a useful tool to debug Linux boot issues.

Start GDB. GDB is used in textual mode as indicated by the -nw switch. The kernel elf `vmlinux` is specified, not `zImage.initrd`.

```
$ powerpc-eabi-gdb -nw vmlinux
```

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

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

Note: If a third party version of GDB which does not explicitly support the PowerPC 440 processor is used, the user should choose powerpc:common instead.

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 localhost:1234
```

GDB displays the present program status:

```
Connected to a PPC440 target.
__delay (loops=100000) at include/asm/time.h:97
97              return mftbl();
(gdb)
```

It is immediately seen that execution has halted in the __delay() function, as was observed with XMD.

GDB Macros

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 `log.txt`, will display the present Linux syslog:

```
# display Linux syslog
define syslog
   set $size = sizeof(__log_buf)
   # Go through the buffer a byte at a time, until a NULL is encountered.
   set $byte = 0
   while ( $byte < $size )
     printf "%c", log_buf[$byte++]
```

```
      # End of string?
      if (log_buf[$byte] == 0)
        # Exit loop
        set $byte = $size
      end
    end
end
```

The macro is loaded as follows:

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

The Linux syslog is displayed:

```
(gdb) syslog
```

Note: This operation will not complete quickly.

```
<0>------------[ cut here ]------------
<2>kernel BUG at arch/powerpc/sysdev/xilinx_intc.c:150!
<4>stopped custom tracer.
<4>Oops: Exception in kernel mode, sig: 5 [#1]
<4>PREEMPT Xilinx Virtex
<4>Modules linked in:
<4>NIP: c02a71c4 LR: c02a7224 CTR: c02a7140
<4>REGS: c02e3ee0 TRAP: 0700   Not tainted  (2.6.24_pro5024-ml507)
<4>MSR: 00021000 <ME>  CR: 22002242  XER: 20000000
<4>TASK = c02c04b0[0] 'swapper' THREAD: c02e2000
<6>GPR00: 00000001 c02e3f90 c02c04b0 00000000 c08a7230 c08a72a9 c02f6894
00000000
<6>GPR08: c02d31d0 00000000 c08a722b 00000051 22002242 100ac364 0ffcfa00
0ffde000
<6>GPR16: c026056c c0260554 cf81fd88 c0263868 0fe9d429 00000002 0ffc981c
00000000
<6>GPR24: 00000000 0056c000 00000de0 c02eda78 c02f0000 c025fcbc 00000000
00000000
<4>NIP [c02a71c4] xilinx_intc_init_tree+0x84/0x134
<4>LR [c02a7224] xilinx_intc_init_tree+0xe4/0x134
<4>Call Trace:
<4>[c02e3f90] [c02a7224] xilinx_intc_init_tree+0xe4/0x134 (unreliable)
<4>[c02e3fb0] [c02a27c0] init_IRQ+0x24/0x34
<4>[c02e3fc0] [c029c964] start_kernel+0x190/0x2c0
<4>[c02e3ff0] [c0000254] skpinv+0x1fc/0x238
<4>Instruction dump:
<4>41920078 7fc4f378 38a00000 7fe3fb78 4beee855 38800000 2f830000 7fa5eb78
<4>7fe3fb78 409effd0 7c000026 54009ffe <0f000000> 7fe3fb78 4bfffe89
3d20c02f
<4>---[ end trace 8640abe69a316dee ]---
```

The syslog shows an assertion failure in `xilinx_intc.c` line 150 because no compatible interrupt controller was found in the device tree:

```
void __init xilinx_intc_init_tree(void)
{
    struct device_node *np;

    /* find top level interrupt controller */
    for_each_compatible_node(np, NULL, "xlnx,opb-intc-1.00.c") {
        if (!of_get_property(np, "interrupts", NULL))
            break;
```

```
        }
            if(!np) {
            for_each_compatible_node(np, NULL, "xlnx,xps-intc-1.00.a") {
                if (!of_get_property(np, "interrupts", NULL))
                    break;
            }
            }

        /* xilinx interrupt controller needs to be top level */
        BUG_ON(!np);

        master_irqhost = xilinx_intc_init(np);
        BUG_ON(!master_irqhost);

        irq_set_default_host(master_irqhost);
        of_node_put(np);
    }
```

View the syslog with XMD

The Linux syslog buffer can also be viewed with XMD. The script linux-syslog.tcl provided with this application note performs this task. If the user has completed the steps in "XMD and TCL Scripting" this script will already be available for use.

Note: The output shown assumes that the user is presently in the `<Linux>/arch/powerpc/boot` directory.

```
XMD% set elf_file ../../../vmlinux
XMD% stop
XMD% linux_syslog
Displaying Linux syslog buffer of 0x00004000 length at 0xc02f005c
<6>Using Xilinx Virtex machine description
...
```

Correcting the problem

A specific version of the XPS intc peripheral was expected to be found in the device tree system description, and it was not present. The BUG_ON() macro causes the system to panic(); a fatal error. It accomplishes this using the PowerPC processor trap instruction (twnei).

Edit `<Linux>/arch/powerpc/boot/dts/ml507.dts`, making the modifications shown in **red**:

```
            xps_intc_0: interrupt-controller@81800000 {
                #interrupt-cells = <0x2>;
                compatible = "xlnx,xps-intc-2.00.a", "xlnx,xps-intc-1.00.a";
                interrupt-controller ;
                reg = < 0x81800000 0x10000 >;
                xlnx,num-intr-inputs = <0x5>;
            } ;
```

The compatible line is a list of conforming devices. The system provided with this application note contains version 2.00.a of the XPS Interrupt Controller. The driver has no direct support for this version of the device, so it is necessary for the device tree to also list version 1.00.a as a conforming device for the driver to recognize this core in the system. The system will not function without an interrupt controller present.

Rebuild `zImage.initrd` and boot the new image. The login prompt should appear. Log in as 'root'. There is no password.

Using *strace*

Identifying the Problem

When debugging, it is often advantageous to have more than one console available. To achieve this, the user will telnet into the ML507 board and login over the network. The system has been statically assigned the IP address of `192.168.0.10` in the device tree:

```
chosen {
        bootargs = "console=ttyS0,9600 ip=192.168.0.10 root=/dev/ram rw";
        linux,stdout-path = "/plb@0/serial@83e00000";
    } ;
```

The host has been assigned **`192.168.0.1`** in the examples shown in this application note.

Note: The user may wish for the embedded board to use DHCP or assign it a static address suitable to a locally established network. IP addressing is beyond the scope of this application note.

Verify that the network is operational by pinging the embedded board from the host PC:

```
$ ping 192.168.0.10

Pinging 192.168.0.10 with 32 bytes of data:

Reply from 192.168.0.10: bytes=32 time=4ms TTL=64
Reply from 192.168.0.10: bytes=32 time=2ms TTL=64
Reply from 192.168.0.10: bytes=32 time=2ms TTL=64
Reply from 192.168.0.10: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.0.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 4ms, Average = 2ms
```

Attempt to telnet into the embedded board:

```
$ telnet 192.168.0.10
```

The attempt will fail. No error is shown. The network has been shown to be operational with a successful ping. The root file system on the embedded board must be incorrect in some way.

When successful, the telnet daemon telnetd is started by inetd unpon a new incoming telnet connection. Verify that inetd is running:

```
# ps | grep inetd
  452 root     /usr/sbin/inetd
```

Verify that inetd is configured to start telnetd when a new connection arrives:

```
# grep ^telnet /etc/inetd.conf
telnet  stream  tcp     nowait  telnetd    /usr/sbin/telnetd  telnetd -l
/bin/sh
```

Verify that the telnet service has been assigned the expected TCP port, 23:

```
# grep ^telnet /etc/services
telnet          23/tcp
```

The configuration is correctly specified, but telnet connections still fail. Additional visibility is needed to determine why it has failed.

STRACE

The strace utility is used to trace system calls and signals. All user applications, such as inetd and telnetd, request services from the kernel using system calls. Common system calls include open (open a file), close (close a file), read (read a file), close (close a file), and so forth.

Kill the presently running inetd. The previous ps command determined that in this particular instance this was process 452:

```
# kill -9 452
```

Start inetd with strace. The -f (follow fork) option is used because we wish to see the system calls made by child processes spawned by inetd (telnetd).

```
# strace -f inetd
```

Wait until inetd reaches a steady state. The many system calls made will scroll until inetd waits in its select() loop:

```
select(8, [4 5 6 7], NULL, NULL, NULL)
```

At this time, telnet to the ML507 from the host and observe the system calls made. Among the many system calls, it is seen that the child process could not open `/dev/ptmx`. This device is used to create a new pseudo terminal.

```
[pid   494] open("/dev/ptmx", O_RDWR|O_LARGEFILE) = -1 EACCES (Permission
denied)
```

Kill the strace process with a ^C. Examine the file permissions on `/dev/ptmx`:

```
# ls -l /dev/ptmx
crw-rw----    1 root     root       5,   2 Jan  1 00:00 /dev/ptmx
```

The telnetd process is run as user telnetd, not user root. It does not have access to `/dev/ptmx`.

How to Solve the Problem

Make `/dev/ptmx` world accessible:

```
# chmod a+rw /dev/ptmx
# ls -l /dev/ptmx
crw-rw-rw-    1 root     root       5,   2 Jan  1 00:00 /dev/ptmx
```

Telnet into the ML507 from the host:

```
MontaVista(R) Linux(R) Professional Edition 5.0.24 (0802884)
Linux/ppc 2.6.24_pro5024-ml507

/ $
/ $ whoami
telnetd
/ $
```

Modifying the Root File System

The proper file permissions on `/dev/ptmx` must be made permanently to the root filesystem image bundled in the `zImage.initrd` executable. **The user must have root permission to perform this task.**

1. cd `<Linux>/arch/powerpc/boot`
2. Uncompress and mount the ramdisk image
   a. gunzip `ramdisk.image.gz`
   b. mkdir `ramdisk_mnt`
   c. mount -o loop `ramdisk.image ramdisk_mnt`

3. Perform any desired modifications to the root filesystem. This system has been configured to use **udev**. The appropriate udev configuration file is edited to enable incoming telnet connections.
   a. cd `ramdisk_mnt/etc/udev/rules.d`
   b. Edit `devfs.rules` to include the configuration shown in **red**:

```
KERNEL=="ptmx", MODE="0666", GROUP="tty"
```

4. Unmount and compress the ramdisk image
   a. umount `ramdisk_mnt`
   b. gzip `ramdisk.image`
5. Create a new `zImage.initrd` executable.
   a. cd <Linux>
   b. make ARCH=powerpc zImage.initrd

Debugging user applications

The application buggy_core is designed to “crash”. It will attempt to access memory outside of its address space, causing a segmentation violation.

```
# buggy_core
buggy_core:
Segmentation fault
#
```

Using gdbserver

To debug application software bugs the gdbserver application, a remote server for GDB, is used. GDB running on the host is used to debug an application run on the embedded board. GDB can communicate to a remote server through a serial port or over the network, as was seen in “Using GDB to debug kernel boot problems” where GDB communicated with the GDB server present within XMD.

Note: XMD is not suitable for debugging Linux user applications. It is unaware of Linux processes and can not natively access or modify Linux page tables upon demand.

Stripped binaries

The buggy_core application found on the ramdisk on the embedded target has been stripped. The strip utility removes all symbols and debugging information from an executable. This makes the executable much smaller. Stripped executables by themselves are virtually useless to a debugger.

Examine the executable on the ramdisk:

```
# ls -l buggy_core
-rwxr-xr-x    1 root root 4076 Apr 20 13:45 buggy_core
#
```

Compare this to the unstripped executable:

```
$ cd <EDK Project>/buggy_core
$ ls -l buggy_core-unstripped
-rwxr-xr-x  1 root root 13184 Apr 20 13:45 buggy_core-unstripped
```

The unstripped executable is approximately three times the size of the stripped executable.

Launch the application with gdbserver:

```
# gdbserver localhost:1234 buggy_core
Process buggy_core created; pid = 480
Listening on port 1234
```

The “localhost:1234” option indicates to gdbserver that the remote debugger will connect over the network. The GDB server will listen for connections on port 1234.

Connect to the gdbserver

Start GDB on the host using the **unstripped** binary:

```
$ powerpc-eabi-gdb -nw buggy_core-unstripped
```

Connect to the gdbserver

```
(gdb) target remote 192.168.0.10:1234
Remote debugging using 192.168.0.10:1234
Using default architecture powerpc:405
[New thread 480]
0x48016084 in ?? ()
(gdb)
```

Continue program execution and await the crash:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xdeadbeec in ?? ()
```

The program has attempted to execute code at `0xDEADBEEC`, which is out of the address space of the application. Direct GDB to display a function call backtrace:

```
(gdb) bt
#0  0xdeadbeec in ?? ()
#1  0x1000042c in crashfunc ()
#2  0x1000047c in main ()
```

The branch to the illegal address occurred in the function crashfunc(). Examining the source, the location is shown in **red**:

```
void crashfunc()
{
    void (*fp)() = (void*)0xDEADBEEF;

    /*
     * Attempt to call function at 0xDEADBEEF. This will cause a SEGV
     */
    (*fp)();
}
```

Debugging Problems in the Field

It will not always be possible to debug an application interactively with gdbserver. In these situations, a core file from the application which has crashed is generated and retrieved from the embedded board for later analysis.

When buggy_core was run previously no coredump was generated. This is because the default behavior is to limit the coredump size (in this case to 0 bytes). Limits are viewed and set with the ulimit command:

```
# ulimit -a
time(seconds)        unlimited
file(blocks)         unlimited
data(kb)             unlimited
stack(kb)            8192
coredump(blocks)     0
memory(kb)           unlimited
locked memory(kb)    32
process              2048
```

```
nofiles              1024
vmemory(kb)          unlimited
locks                unlimited
```

Change the coredump size to unlimited. In this way, coredump size is limited only by space available in the file system.

```
# ulimit -c unlimited
```

Run buggy_core

```
# buggy_core
buggy_core:
Segmentation fault (core dumped)
```

Unlike previously, this time Linux indicates that a core file has been generated.

```
# ls -l core
-rw-------    1 root     root       159744 Jan  1 16:50 core
```

There are various ways to fetch this file from the embedded board. The user will find NFS a convenient tool when debugging. Configuring and using NFS are beyond the scope of this application note. To copy the core file to the host for remote debugging TFTP is used.

Configure the TFTP server

All material provided in this section pertaining to configuration of the TFTP server is intended as a quick reference only. Users which are unfamiliar with these concepts may need additional reference material.

Linux

Typical Linux server installations will provide an already configured TFTP server. Files placed in the `/tftpboot` directory are available to TFTP clients. By default, clients are not allowed to upload files. This configuration must be changed if TFTP is to be used to upload core files from the embedded board.

Note: Obtaining, installing, and configuring a Linux TFTP server is beyond the scope of this application note. The steps shown are a **general guideline** only. If the user wishes to use a Linux TFTP server they should consult the pertinent documentation.

The TFTP server must be configured to permit the creation of files. This is done by starting the server with the -c command line option. If the server in question uses xinetd this is accomplished with a `/etc/xinetd.d/tftp` config file like shown:

```
service tftp
{
        socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -c -s /tftpboot
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

If the server in question uses inetd rather than xinetd the configuration found in `/etc/inetd.conf` should resemble this:

```
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
tftp dgram udp wait nobody /usr/sbin/in.tftpd in.tftpd -c -s /tftpboot
```

Inetd or xinetd must be restarted after this modification to their configuration files.

The `/tftpboot` directory is typically owned by 'root'. It must be world writable or clients will not be able to create files there. The TFTP server runs as user 'nobody'.

```
# chmod 777 /tftpboot
# ls -ld /tftpboot
drwxrwxrwx  3 root root 4096 Apr 15 09:45 /tftpboot/
```

Microsoft Windows

Windows has no traditional TFTP server. TFTPD32, which is freely available from http://tftpd32.jounin.net is recommended for use with this application note. It requires no installation and can be run directly from the directory in which it has been downloaded.

Transfer the core file

Transfer the core file generated in "Debugging Problems in the Field" from the embedded board to the tftp server.

```
# tftp -pr core 192.168.0.1
```

Note: The tftp client implementation on the embedded board (busybox) does not use the standard Linux tftp command line arguments.

Analyze the core file

Start gdb, specifying the core file to use.

```
$ ppc_440-gdb -nw buggy_core-unstripped /tftpboot/core
```

Note: The GDB shipped with EDK does not support core files; it is necessary to use the GDB supplied by MontaVista.

Obtain a backtrace:

```
(gdb) bt
#0  0xdeadbeec in ?? ()
#1  0x1000042c in crashfunc () at buggy_core.c:21
#2  0x1000047c in main (argc=1, argv=0xbfde2e44) at buggy_core.c:28
```

As observed previously, the branch to the invalid address occurred in the procedure crashfunc().

Debugging Memory Leaks

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**. 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 the process is terminated. Eventually no memory will be left in the pool, and all calls to **malloc**() will fail. If the memory utilization of a process has not been limited with the ulimit command, one process can use all system resources, to the detriment of the entire system.

The application buggy_memleak performs several memory allocations in different functions. The functions mtrace() and muntrace() are used to gather memory allocation statistics. Any calls to malloc() and free() are recorded when tracing is enabled.

```
mtrace();

... calls to malloc() and free() logged...

muntrace();
```

Calls are recorded to the file indicated in the environment variable **MALLOC_TRACE**. No memory allocation statistics are gathered if this environment variable has not been set.

Application ‘buggy_memleak’

Run the buggy_memleak application:

```
# export MALLOC_TRACE=malloc-log.txt
# buggy_memleak
buggy_memleak:
Press <enter> to perform memory allocations:
```

The application will wait for a keypress before proceeding. Do not press any key at this time. Telnet into the embedded board from the host PC. The embedded board has been assigned a static IP address of 192.168.0.10. The user must have successfully completed the steps in “STRACE” to be able to successfully telnet to the board.

From the telnet window, run the ‘free’ command. This will display the amount of physical memory in use at this time (before buggy_memleak has performed it’s memory allocations):

```
/ $ su
/ #
/ # free
              total         used         free       shared      buffers
  Mem:        256884        17760       239124            0         7876
 Swap:            0            0            0
Total:        256884        17760       239124
```

Determine the PID of buggy_memleak:

```
/ # ps | grep buggy_memleak
  582 root     buggy_memleak
```

The /proc filesystem provides information about all presently running processes, and is the source of information for user applications such as ‘ps’. Examine the files associated with process 582:

Note: The PID number seen by the user will vary according to which shell commands have been run prior to beginning this procedure.

Examine the address space for this process:

```
/ # cd /proc/582
/proc/582 # cat maps
00100000-00103000 r-xp 00100000 00:00 0          [vdso]
0fe85000-0ffd8000 r-xp 00000000 01:00 464        /lib/libc-2.5.90.so
0ffd8000-0ffe8000 ---p 00153000 01:00 464        /lib/libc-2.5.90.so
0ffe8000-0ffe9000 r--p 00153000 01:00 464        /lib/libc-2.5.90.so
0ffe9000-0ffed000 rwxp 00154000 01:00 464        /lib/libc-2.5.90.so
0ffed000-0fff0000 rwxp 0ffed000 00:00 0
10000000-10001000 r-xp 00000000 01:00 387        /root/buggy_memleak
10010000-10011000 rwxp 00000000 01:00 387        /root/buggy_memleak
48000000-4801e000 r-xp 00000000 01:00 459        /lib/ld-2.5.90.so
4801e000-48022000 rw-p 4801e000 00:00 0
4802d000-4802f000 rwxp 0001d000 01:00 459        /lib/ld-2.5.90.so
bfbd9000-bfbee000 rw-p bffeb000 00:00 0          [stack]
```

Examine the current memory usage of this process:

```
/proc/582 # cat status
Name:   buggy_memleak
State:  S (sleeping)
Tgid:   582
Pid:    582
PPid:   453
TracerPid:      0
```

```
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0 1 2 4
VmPeak:     1700 kB
VmSize:     1700 kB
VmLck:         0 kB
VmHWM:       436 kB
VmRSS:       436 kB
VmData:       40 kB
VmStk:        84 kB
VmExe:        68 kB
VmLib:      1412 kB
VmPTE:        20 kB
...
```

In the console window where 'buggy_memleak' is run, press <enter> so that the application will perform it's memory allocations. The application will prompt the user to press <enter> to write to the allocated memory. Do not press <enter> at this time.

```
Press <enter> to perform memory allocations:

Press <enter> to write to allocated memory:
```

In the telnet window, view the free physical memory again:

```
/proc/582 # free
             total         used         free       shared      buffers
  Mem:       256884        18616       238268            0         7876
 Swap:            0            0            0
Total:      256884        18616       238268
```

Compared to the previous output of 'free' (before the application allocated memory with malloc(), 18616 - 17760 = 856k of system memory has been consumed.

View the process address space again:

```
/proc/582 # cat maps
00100000-00103000 r-xp 00100000 00:00 0          [vdso]
0fe85000-0ffd8000 r-xp 00000000 01:00 464        /lib/libc-2.5.90.so
0ffd8000-0ffe8000 ---p 00153000 01:00 464        /lib/libc-2.5.90.so
0ffe8000-0ffe9000 r--p 00153000 01:00 464        /lib/libc-2.5.90.so
0ffe9000-0ffed000 rwxp 00154000 01:00 464        /lib/libc-2.5.90.so
0ffed000-0fff0000 rwxp 0ffed000 00:00 0
10000000-10001000 r-xp 00000000 01:00 387        /root/buggy_memleak
10010000-10011000 rwxp 00000000 01:00 387        /root/buggy_memleak
10011000-10316000 rwxp 10011000 00:00 0          [heap]
48000000-4801e000 r-xp 00000000 01:00 459        /lib/ld-2.5.90.so
4801e000-48024000 rw-p 4801e000 00:00 0
4802d000-4802f000 rwxp 0001d000 01:00 459        /lib/ld-2.5.90.so
4802f000-48231000 rw-p 4802f000 00:00 0
bfbd9000-bfbee000 rw-p bffeb000 00:00 0          [stack]
```

New valid addresses have been added to the process address space. The address ranges at `0x10011000` and `0x4802F000` are where memory returned by calls to malloc() have been mapped. There are two ranges rather than one due to the way malloc() operates with glibc. Memory allocations under 128k come from the segment noted as [heap]. These are allocated through use of brk() or sbrk(). Larger allocations are obtained with a call to mmap() for a new private segment.

View the amount of memory presently allocated by this process:

```
/proc/582 # cat status
Name:   buggy_memleak
State:  S (sleeping)
Tgid:   582
Pid:    582
PPid:   453
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0 1 2 4
VmPeak:     6856 kB
VmSize:     6856 kB
VmLck:         0 kB
VmHWM:      1244 kB
VmRSS:      1244 kB
VmData:     5196 kB
VmStk:        84 kB
VmExe:        68 kB
VmLib:      1412 kB
VmPTE:        24 kB
```

Compared to the previous memory usage of the application, (5196 - 40) = 5156k. This is significantly more memory usage than seen in the difference of system memory reported by 'free', 856k. This is because the memory allocated to this process has mostly just increased the size of the process address space. Actual storage is not consumed until the process accesses this memory. The amount of physical resources actually used is indicated by VmRSS.

In the console window, press <enter> so that buggy_memleak will write to one of the large blocks of memory just allocated. The user will be prompted to press <enter> to exit the application. Do not do so at this time.

```
Press <enter> to write to allocated memory:

Press <enter> to exit:
```

In the telnet window, view the used physical memory again:

```
/proc/582 # free
             total         used         free       shared      buffers
  Mem:       256884        19636       237248            0         7876
 Swap:            0            0            0
Total:      256884        19636       237248
```

Additional physical memory is now in use. View the amount of memory allocated to the process:

```
/proc/582 # cat status
Name:   buggy_memleak
State:  S (sleeping)
Tgid:   582
Pid:    582
PPid:   453
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0 1 2 4
VmPeak:     6856 kB
VmSize:     6856 kB
VmLck:         0 kB
```

```
VmHWM:       2264 kB
VmRSS:       2264 kB
VmData:      5196 kB
VmStk:         84 kB
VmExe:         68 kB
VmLib:       1412 kB
VmPTE:         24 kB
```

It is seen that the quantity of memory requested by the process (VmData) is unchanged, but the actual memory consumed (VmRSS, the Resident Segment Size) has increased.

Press <enter> to exit the application. The memory trace file is now available for debugging.

```
Press <enter> to exit:

# ls -l malloc-log.txt
-rw-r--r--    1 root     root          9300 Jan  1 04:28 malloc-log.txt
```

Transfer the log file to the host with TFTP. The user must have completed the steps in "Configure the TFTP server" to complete this step.

```
# tftp -pr malloc-log.txt 192.168.0.1
```

The log file contains a record of all calls to malloc() and free(). The output is not intended to be human readable. The standard Linux utility 'mtrace' is used to analyze the log:

```
$ mtrace malloc-log.txt
```

All memory which was allocated and not freed is represented in this output. The size allocated and address from where malloc() was called are listed.

EXCERPT:

```
Memory not freed:
-----------------
           Address     Size     Caller
0x0000000010011390   0x1000  at 0x10000524
0x00000000100123a0   0x1000  at 0x10000524
0x00000000100133b0   0x1000  at 0x10000524
0x00000000100143c0   0x1000  at 0x10000524
0x00000000100153d0   0x1000  at 0x10000524
...
```

The output of `mtrace` is suitable for further processing. It would be useful to see all of the allocations by a single caller to be added together and represented as a single entity. The script `mtrace_analyze.pl` provided with this application note is used.

```
$ cd <project area>
$ cp <tftp directory>/malloc-log.txt <project area>
$ mtrace malloc-log.txt | buggy_memleak/mtrace_analyze.pl -
buggy_memleak-unstripped
Total: 1048576 Caller: 0x1000056c memwaster1() buggy_memleak.c:28
Total: 1048576 Caller: 0x10000558 memwaster1() buggy_memleak.c:26
Total: 2621440 Caller: 0x100004f0 memwaster3() buggy_memleak.c:16
Total:  409600 Caller: 0x10000524 memwaster2() buggy_memleak.c:21
```

The totals and locations are seen to match the source code in `buggy_memleak.c`.

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 buggy_memleak. 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.

Debugging Driver Application Interaction

'Buggy' Driver Operation

A driver has been supplied with this application note to demonstrate debugging kernel drivers and interaction of user applications with a driver. The user integrated this driver into their kernel build with the steps in "Build a kernel image".

The buggy_drv contains several static strings. The application selects a string by writing the desired string index to the device. When the application reads from the device, the presently selected string is read.

The defined strings are shown below:

```
const char *buggy_strings[] = {
 "0:Hello world",
 "1:That's life",
 "2:Goodbye, cruel world",
 "3:The Swift Brown Fox Jumps Over The Lazy Dog",
 "Invalid Selection"
};
```

If the application writes the binary integer value '1' to `/dev/buggy` string index 1 will be selected. The following read from the device will read the string at index 1 on the buggy_strings array. The string "1:That's life" is read.

Application 'buggy_allstrings'

The buggy_allstrings application selects and displays the first three string indices. However, when the application is run, this in not the behavior observed:

```
# buggy_allstrings
Display the first three strings provided by /dev/buggy.
Selected string 0
String 1677721600 is 14 bytes :0:Hello world:
#
```

Only the first string has printed. The driver only contains a few strings, so clearly index 1677721600 is invalid. When the application source is examined, it is seen that *indx* is only incremented once per loop:

```
int main()
{
    int devfd;
    ssize_t bytes;
    char buffer[10];
    int indx;

    printf("Display the first three strings provided by /dev/buggy.\n");

    devfd = open(BUGGY_DEV, O_RDWR);
    if (devfd < 0) {
        printf("Unable to open device file %s.\n", BUGGY_DEV);
        exit(1);
    }

    /* Print the first 3 strings */
    for (indx = 0; indx < 3; indx++) {
        /* Select a string */
        write(devfd, &indx, sizeof(int));
        printf("Selected string %d\n", indx);
```

```
        /* Display selected string */
        bytes = read(devfd, buffer, sizeof(buffer));
        printf("String %d is %d bytes :%s:\n", indx, bytes, buffer);
    }
}
```

No error manipulating indx is apparent.

Driver /proc filesystem entry

The driver creates an entry for itself in the `/proc` filesystem. This filesystem in entirely virtual - when files in this filesystem are read, the data read are composed on the fly by software. This is a useful debugging tool for providing information on whatever internal driver data the programmer whishes to make available to user space.

The drivers `/proc` entry is read. The below output is produced by the buggy_proc_seq_show() procedure.

```
# cat /proc/driver/buggy_drvr

Reads:      1
Bytes Read: 14
Writes:     1
Opens:      1
Closes:     1

String Index: 0
```

It is seen that there has been only a single read request on the device, and only a single write request. These correspond to selecting string index 0 and displaying this string. 14 bytes have been read.

Examining the source for buggy_allstrings it is seen that a read from the device will fill a 10 byte buffer. Yet the return from read() and the driver statistics in `/proc/driver/buggy_drvr` indicated that 14 bytes have been read. The read() function (and the read system call) stipulate a buffer size. The available byte count has been correctly specified in the application.

```
int main()
{
    int devfd;
    ssize_t bytes;
    char buffer[10];
    int indx;

    printf("Display the first three strings provided by /dev/buggy.\n");

    devfd = open(BUGGY_DEV, O_RDWR);
    if (devfd < 0) {
        printf("Unable to open device file %s.\n", BUGGY_DEV);
        exit(1);
    }

    /* Print the first 3 strings */
    for (indx = 0; indx < 3; indx++) {
        /* Select a string */
        write(devfd, &indx, sizeof(int));
        printf("Selected string %d\n", indx);
```

```
            /* Display selected string */
            bytes = read(devfd, buffer, sizeof(buffer));
            printf("String %d is %d bytes :%s:\n", indx, bytes, buffer);
        }
    }
```

A read of the device is handled by the driver function buggy_read(). When the number of bytes to be copied **copybytes** is calculated, the driver does not honor the maximum value **count** provided by the application. This results in the driver writing past the end of the buffer in the user application.

```
/*
 * buggy_read:
 * A userspace read of the device file.
 */
ssize_t buggy_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos)
{
...
    /*
     * How many bytes to copy
     */
    copybytes = strlen(buggy_strings[dev->string_index]) + 1;

    /*
     * Copy this buffer to user space.
     */
    err = copy_to_user(buf, buggy_strings[dev->string_index], copybytes);
```

The next item in memory after the buffer is the variable indx. When the driver writes past the end of the buffer the value of indx is corrupted. The value of indx, shown in the program output, is actually the characters of the string displayed.

```
# ./buggy_allstrings
Display the first three strings provided by /dev/buggy.
Selected string 0
String 1677721600 is 14 bytes :0:Hello world:
```

Refer to Table 2.

Table 2: **String Buffer**

String	0	:	H	e	l	l	o		w	o	r	l	d	NUL		
ASCII	30	3A	48	65	6C	6C	6F	20	77	6F	72	6C	64	00	00	00
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Decimal													1677721600			

How to solve the problem

The driver must not exceed the maximum available buffer space available. The buggy_read() procedure is modified:

```
copybytes = min(strlen(buggy_strings[dev->string_index]) + 1, count);
```

Application 'buggy_choosestring'

The buggy_choosestring application will select and display the string present at the index specified by the user on the command line:

```
# buggy_choosestring 0
Selected string 0
String 0 is 14 bytes :0:Hello world:
```

Display string 1000:

```
# buggy_choosestring 1000
Unable to handle kernel paging request for data at address 0x00000000
Faulting instruction address: 0xc0011354
stopped custom tracer.
Oops: Kernel access of bad area, sig: 11 [#1]
PREEMPT Xilinx Virtex
Modules linked in:
NIP: c0011354 LR: c0164c48 CTR: c0164be0
REGS: cf877e10 TRAP: 0300   Not tainted  (2.6.24_pro5024-ml507)
MSR: 00029000 <EE,ME>  CR: 24000482  XER: 20000000
DEAR: 00000000, ESR: 00000000
TASK = cf019410[510] 'buggy_choosestr' THREAD: cf876000
GPR00: 00000000 cf877ec0 cf019410 00000000 ffffffff 00000064 cf877f20
00000000
GPR08: 00000004 c02d7068 00000004 c02d8008 00000000 10018ae8 10087954
1007d988
GPR16: 1007d998 1007d944 00000000 1008579c 100910d3 100a825c bf895768
c02f0000
GPR24: cf0284e4 00000064 cf877f20 bfaf4b20 00000000 ce407aa0 cf0284e0
00000064
NIP [c0011354] strlen+0x4/0x18
LR [c0164c48] buggy_read+0x68/0x1f4
Call Trace:
[cf877ec0] [bfaf4978] 0xbfaf4978 (unreliable)
[cf877ef0] [c007dacc] vfs_read+0xb4/0x16c
[cf877f10] [c007e038] sys_read+0x64/0xd8
[cf877f40] [c000d1d0] ret_from_syscall+0x0/0x3c
Instruction dump:
4082fff4 4e800020 38a3ffff 3884ffff 8c650001 2c830000 8c040001 7c601851
4d860020 4182ffec 4e800020 3883ffff <8c040001> 2c000000 4082fff8 7c632050
---[ end trace 8712654a561d6dc2 ]---
```

A kernel Oops indicates a serious software error in the kernel. Kernel code has attempted to access a virtual address (0x00000000) for which there was no virtual->physical mapping. The address was accessed from strlen(), which was called by the buggy_read() routine.

It is seen that buggy_read() calls strlen() in several places, the first of which is where it determines how many bytes to copy:

```
copybytes = strlen(buggy_strings[dev->string_index]) + 1;
```

If the string_index is greater than the total number of strings defined in buggy_strings[] then this access would dereference a pointer beyond the buggy_strings[] pointer. Examining the buggy_write() procedure it is seen that the driver accepts any input from user space, regardless of whether or not it is valid:

```
dev->string_index = *(int*)buffer;
```

How to Solve the Problem

The buggy_write() procedure must not allow the driver to select an invalid string index.

```
dev->string_index = *(int*)buffer;
if (dev->string_index >= NUM_BUGGY_STRINGS) {
     dev->string_index = NUM_BUGGY_STRINGS;
}
```

References

1. UG111 Embedded System Tools Reference Manual
2. XAPP1117 Software Debugging Techniques for PowerPC 440 Processor Embedded Systems.
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-Weslet Publishing Complany, 1994.
5. Christopher Hallinan. Embedded Linux Primer. Prentice Hall, 2007.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
6/9/09	1.0	Initial Xilinx release.

Notice of Disclaimer

Xilinx is disclosing this Application Note to you “AS-IS” with no warranty of any kind. This Application Note is one possible implementation of this feature, application, or standard, and is subject to change without further notice from Xilinx. You are responsible for obtaining any rights you may require in connection with your use or implementation of this Application Note. XILINX MAKES NO REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL XILINX BE LIABLE FOR ANY LOSS OF DATA, LOST PROFITS, OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM YOUR USE OF THIS APPLICATION NOTE.