Table of Contents

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

shortcut

Prentice Hall Open Source Software Development Series

Debugging Embedded Linux

Christopher Hallinan

KERNEL DEBUGGING TECHNIQUES

DEBUGGING EMBEDDED LINUX APPLICATIONS

PRENTICE HALL
PEARSON EDUCATION

PRENHALLPROFESSIONAL.COM

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

Visit us on the Web: www.prenhallprofessional.com

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458
United States of America
Fax: (201) 236-3290
ISBN 0-13-158013-2

First release, August 2006

No matter how you approach it, Linux debugging will always be complex. The first part of this Shortcut examines some of these complexities as they relate to kernel debugging and present ideas and methods to improve your debugging skills, inside the kernel itself and then inside device drivers. You begin by learning to use Kernel GNU Debugger (KGDB) to probe the kernel. But because you cannot debug very early kernel startup code with KGDB, we also show you how to work with a hardware debug probe. For the examples in this section, we use a unit manufactured by Abatron called the BDI-2000. Hardware debug probes are often called JTAG probes, because they use a low-level communications method first employed for boundary scan testing of integrated circuits defined by the Joint Test Action Group (JTAG).

In the second part of this Shortcut, we continue our coverage of GDB for debugging application code in user space. We extend our coverage of remote debugging and the tools and techniques used for this peculiar debugging environment.

This Shortcut includes Chapters 14 and 15 from the book *Embedded Linux Primer* by Christopher Hallinan (0-13-167984-8). As such, it contains references to earlier and later chapters that are not included in this Shortcut. Apart from these references, the Shortcut is fully self-contained and is an excellent choice for embedded Linux developers interested in learning both kernel-level and application-level debugging techniques.

SECTION 14.1

Challenges to Kernel Debugging

CHAPTER 14

Kernel Debugging Techniques

Often the pivotal factor in achieving development timetables comes down to one's efficiency in finding and fixing bugs. Debugging inside the Linux kernel can be quite challenging. No matter how you approach it, kernel debugging will always be complex. This chapter examines some of the complexities and presents ideas and methods to improve your debugging skills inside the kernel and device drivers.

14.1 Challenges to Kernel Debugging

Debugging a modern operating system involves many challenges. Virtual memory operating systems present their own unique challenges. Gone are the days when we could replace a processor with an in-circuit emulator. Processors have become far too fast and complex. Moreover, pipeline architectures hide important code-execution details, partly because memory accesses on the bus can be ordered differently from code execution, and particularly because of internal caching of instruction streams. It is not always possible to correlate external bus activity to internal processor instruction execution, except at a rather coarse level.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.1

Challenges to Kernel Debugging

Some of the challenges you will encounter while debugging Linux kernel code are:

- Linux kernel code is highly optimized for speed of execution in many areas.
- Compilers use optimization techniques that complicate the correlation of C source to actual machine instruction flow. Inline functions are a good example of this.
- Single-stepping through compiler optimized code often produces unusual and unexpected results.
- Virtual memory isolates user space memory from kernel memory and can make various debugging scenarios especially difficult.
- Some code cannot be stepped through with traditional debuggers.
- Startup code can be especially difficult because of its proximity to the hardware and the limited resources available (for example, no console, limited memory mapping, and so on).

The Linux kernel has matured into a very high-performance operating system capable of competing with the best commercial operating systems. Many areas within the kernel do not lend themselves to easy analysis by simply reading the source code. Knowledge of the architecture and detailed design are often necessary to understand the code flow in a particular area. Several good books are available that describe the kernel design in detail. Refer to Section 14.6.1, "Suggestions for Additional Reading," for recommendations.

GCC is an optimizing compiler. By default, the Linux kernel is compiled with the `-O2` compiler flag. This enables many optimization algorithms that can change the fundamental structure and order of your code.[1] For example, the Linux kernel makes heavy use of *inline* functions. Inline

[1] *See the GCC manual referenced at the end of this chapter in Section 14.6.1, "Suggestions for Additional Reading" for details on the optimization levels.*

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

functions are small functions declared with the `inline` keyword, which results in the function being included directly in the execution thread instead of generating a function call and the associated overhead.[2] `Inline` functions require a minimum of `-O1` optimization level. Therefore, you cannot turn off optimization, which would be desirable for easier debugging.

In many areas within the Linux kernel, single-stepping through code is difficult or impossible. The most obvious examples are code paths that modify the virtual memory settings. When your application makes a system call that results in entry into the kernel, this results in a change in address space as seen by the process. In fact, any transition that involves a processor exception changes the operational context and can be difficult or impossible to single-step through.

14.2 Using KGDB for Kernel Debugging

Two popular methods enable symbolic source-level debugging within the Linux kernel:

- Using KGDB as a remote `gdb` agent
- Using a hardware JTAG probe to control the processor

We cover JTAG debugging in Section 14.4, "Hardware-Assisted Debugging."

KGDB (Kernel GDB) is a set of Linux kernel patches that provide an interface to `gdb` via its remote serial protocol. KGDB implements a `gdb` stub that communicates to a `cross-gdb` running on your host development workstation. Until very recently, KGDB on the target required a serial connection to the development host. Some targets support KGDB connection via Ethernet, although this is relatively new. Complete support for KGDB is still not in the mainline `kernel.org` kernel. You need to port KGDB to your chosen target or obtain an embedded Linux distribution for your

[2] *Inline functions are like macros, but with the advantage of compile-time type checking.*

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2

Using KGDB for Kernel Debugging

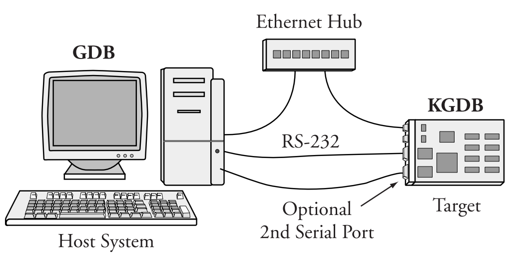
chosen architecture and platform that contains KGDB support. Most embedded Linux distributions available today support KGDB.

Figure 14-1 describes the KGDB debug setup. Up to three connections to the target board are used. Ethernet is used to enable NFS root mount and telnet sessions from the host. If your board has a ramdisk image in Flash that it mounts as a root file system, you can eliminate the Ethernet connection.

A serial port is dedicated for the connection between KGBD and `gdb` running on the development host system, and an optional second serial port serves as a console. Systems that have only one serial port make KGDB somewhat more cumbersome to use.

As you can see in Figure 14-1, the debugger (your cross-version of `gdb`) runs on your development host system. KGDB is part of the kernel running on your target system. KGDB implements the hooks required to interface `gdb` with your target board to enable features such as setting breakpoints, examining memory, and enabling single-step program execution.

FIGURE 14-1
KGDB debug setup



Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2
Using KGDB for Kernel Debugging

14.2.1 KGDB Kernel Configuration

KGDB is a kernel feature and must be enabled in your kernel. KGDB is selected from the Kernel Hacking menu, as shown in Figure 14-2. As part of the configuration, you must select the serial port for KGDB to use. Notice also from Figure 14-2 that we enabled the option to compile the kernel with debug information. This adds the `-g` compiler flag to the build process to enable symbolic debugging.

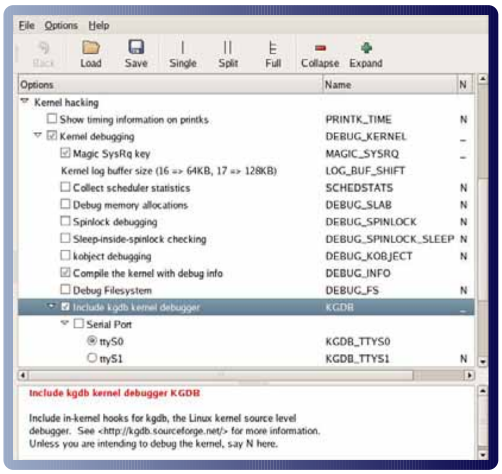


FIGURE 14-2
Kernel configuration for KGDB

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2
Using KGDB for Kernel Debugging

14.2.2 Target Boot with KGDB Support

After your kernel is built with KGDB support, it must be enabled. Unfortunately, the method to enable it is not yet uniform across all architectures and implementations. In general, KGDB is enabled by passing a command-line switch to the kernel via the kernel command line. If KGDB support is compiled into the kernel but not enabled via a command-line switch, it does nothing. When KGDB is enabled, the kernel stops at a KGDB-enabled breakpoint very early in the boot cycle to allow you to connect to the target using `gdb`. Figure 14-3 shows the logic for generating an initial breakpoint when KGDB is enabled.

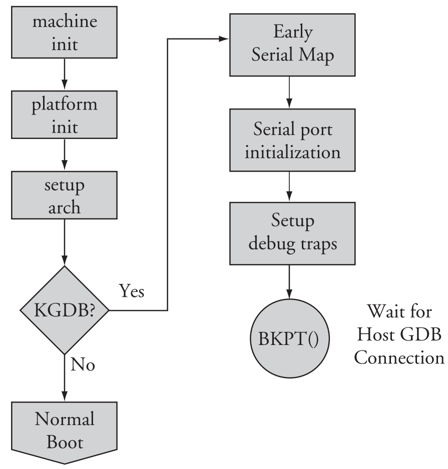


FIGURE 14-3
KGDB logic

KGDB requires a serial port for connection to the host.[3] The first step in setting up KGDB is to enable a serial port very early in the boot process. In many architectures, the hardware UART must be mapped into kernel memory before access. After the address range is

[3] *Notwithstanding the comments made earlier about KGDB over Ethernet.*

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2

Using KGDB for Kernel Debugging

mapped, the serial port is initialized. Debug trap handlers are installed to allow processor exceptions to trap into the debugger.

Listing 14-1 displays the terminal output when booting with KGDB enabled. This example is based on the AMCC 440EP Evaluation Kit (Yosemite board), which ships with the U-Boot bootloader.

LISTING 14-1 Booting with KGDB Enabled Using U-Boot

```
=> sete bootargs console=ttyS1,115200 root=/dev/nfs rw ip=dhcp gdb
=> bootm 200000
## Booting image at 00200000 ...
   Image Name:   Linux-2.6.13
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1064790 Bytes =  1 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
$T0440:c000ae5c;01:c0205fa0;#d9    <<< See text
```

Most of the boot sequence is familiar from our coverage of U-Boot in Chapter 7, "Bootloaders." This kernel boot sequence has two unique features: the command-line parameter to enable KGDB and the odd-looking text string after the kernel is uncompressed.

Recall from Chapter 7 that the kernel command line is defined by the U-Boot `bootargs` environment variable. Notice that we have added the `gdb` parameter, which instructs the kernel to force an early breakpoint and wait for the host debugger (your `cross-gdb`) to connect.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2

Using KGDB for Kernel Debugging

As diagrammed in Figure 14-3, the kernel detects the presence of the `gdb` parameter and attempts to pass control to the remote (host-based) debugger. This is evidenced by the sequence of ASCII characters dumped to the serial port in Listing 14-1. If you are curious about this `gdb` remote serial protocol, it is documented in the `gdb` manual cited at the end of this chapter. In this example, `KGDB` is sending a *Stop Reply* packet reporting the breakpoint trap to the remote `gdb` session on the host. The two 32-bit parameters indicate the location of the program and the stack frame.

Now that the kernel is set up and waiting for the host debugger, we can begin our debugging session. We invoke `cross-gdb` from our host development workstation and connect to the target via `gdb`'s remote protocol. In this example, we are sharing the serial port, so we must disconnect the terminal emulator from the target before trying to connect with `gdb`. Listing 14-2 highlights the `gdb` connection process. This assumes that we have already exited our terminal emulator and freed the serial port for `gdb` to use.

LISTING 14-2 Connecting to KGDB

```
$ ppc_4xx-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }
(gdb) l
820                     return;
821             }
822
823             asm("   .globl breakinst        \n\
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION **14.2**

Using KGDB for Kernel Debugging

```
824                 breakinst: .long 0x7d821008");
825     }
826
827     #ifdef CONFIG_KGDB_CONSOLE
828     /* Output string in GDB O-packet format if GDB has connected.
If nothing
829        output, returns 0 (caller must then handle output). */
(gdb)
```

Here we have performed three actions:

- Invoked `gdb`, passing it the kernel ELF file `vmlinux`
- Connected to the target using the `target remote` command within `gdb`
- Issued the `list` command, using its abbreviated form to display our location in the source code

At the risk of pointing out the obvious, the `vmlinux` image that we pass to `gdb` must be from the same kernel build that produced the target kernel binary. It also must have been compiled with the `-g` compiler flag to contain debug information.

When we issued the `target remote` command, `gdb` responded by displaying the location of the program counter (PC). In this example, the kernel is stopped at the breakpoint defined by the inline assembler statement at line 823 in file `.../arch/ppc/kernel/ppc-stub.c`. When we issue the `continue (c)` command, execution resumes starting at line 825, as indicated.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.2
Using KGDB for Kernel Debugging

14.2.3 Useful Kernel Breakpoints

We have now established a debug connection with the kernel on our target board. When we issue the `gdb continue` (`c`) command, the kernel proceeds to boot, and if there are no problems, the boot process completes. There is one minor limitation of using KGDB on many architectures and processors. An engineering trade-off was made between the need to support very early kernel debugging (for example, before a full-blown interrupt-driven serial port driver is installed) and the desire to keep the complexity of the KGDB debug engine itself very simple and, therefore, robust and portable. KGDB uses a simple polled serial driver that has zero overhead when the kernel is running. As a drawback to this implementation, the traditional `Ctl-C` or Break sequence on the serial port will have no effect. Therefore, it will be impossible to stop execution of the running kernel unless a breakpoint or other fault is encountered.

For this reason, it has become common practice to define some system-wide breakpoints, which provide the capability to halt the current thread of execution. Two of the most common are highlighted in Listing 14-3.

LISTING 14-3 Common Kernel Breakpoints

```
(gdb) b panic
Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
(gdb) b sys_sync
Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
(gdb)
```

Using the `gdb breakpoint` command, again using its abbreviated version, we enter two breakpoints. One is at `panic()` and the other is at the `sync` system call entry `sys_sync()`. The former allows the debugger to be invoked if a later event generates a panic. This enables examination of the system state at the time of the panic. The second is a useful way to halt the kernel and trap into the debugger from user space by entering the `sync` command from a terminal running on your target hardware.

We are now ready to proceed with our debugging session. We have a KGDB-enabled kernel running on our target, paused at a KGDB-defined early breakpoint. We established a connection to the target with our host-based cross debugger—in this case, invoked as `ppc_4xx-gdb`—and we have entered a pair of useful system breakpoints. Now we can direct our debugging activities to the task at hand.

One caveat: By definition, we cannot use KGDB for stepping through code before the `breakpoint()` function in `.../arch/ppc/setup.c` used to establish the connection between a KGDB-enabled kernel and `cross-gdb` on our host. Figure 14-3 is a rough guide to the code that executes before KGDB gains control. Debugging this early code requires the use of a hardware-assisted debug probe. We cover this topic shortly in Section 14.4, "Hardware-Assisted Debugging."

14.3 Debugging the Linux Kernel

One of the more common reasons you might find yourself stepping through kernel code is to modify or customize the platform-specific code for your custom board. Let's see how this might be done using the AMCC Yosemite board. We place a breakpoint at the platform-specific architecture setup function and then continue until that breakpoint is encountered. Listing 14-4 shows the sequence.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

LISTING 14-4 Debugging Architecture-Setup Code

```
(gdb) b yosemite_setup_arch
    Breakpoint 3 at 0xc021a488:
        file arch/ppc/platforms/4xx/yosemite.c, line 308.
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGILL not sent.

Breakpoint 3, yosemite_setup_arch () at arch/ppc/platforms/4xx/yosemite.c:308
308             yosemite_set_emacdata();
(gdb) l
303     }
304
305     static void __init
306     yosemite_setup_arch(void)
307     {
308             yosemite_set_emacdata();
309
310             ibm440gx_get_clocks(&clocks, YOSEMITE_SYSCLK, 6 * 1843200);
311             ocp_sys_info.opb_bus_freq = clocks.opb;
312
(gdb)
```

When the breakpoint at `yosemite_setup_arch()` is encountered, control passes to `gdb` at line 308 of `yosemite.c`. The `list` (`l`) command displays the source listing centered on the breakpoint at line 308. The warning message displayed by `gdb` after the `continue` (`c`) command can be safely ignored.

SECTION 14.3

Debugging the Linux Kernel

It is part of `gdb`'s way of testing the capabilities of the remote system. It first sends a remote `continue_with_signal` command to the target. The KGDB implementation for this target board does not support this command; therefore, it is `NAK`'d by the target. `gdb` responds by displaying this informational message and issuing the standard remote `continue` command instead.

14.3.1 gdb Remote Serial Protocol

`gdb` includes a debug switch that enables us to observe the remote protocol being used between `gdb` on your development host and the target. This can be very useful for understanding the underlying protocol, as well as troubleshooting targets that exhibit unusual or errant behavior. To enable this debug mode, issue the following command:

```
(gdb) set debug remote 1
```

With remote debugging enabled, it is instructive to observe the `continue` command in action and the steps taken by `gdb`. Listing 14-5 illustrates the use of the `continue` command with remote debugging enabled.

LISTING 14-5 **`continue` Remote Protocol Example**

```
(gdb) c
Continuing.
Sending packet: $mc0000000,4#80...Ack
Packet received: c022d200
Sending packet: $Mc0000000,4:7d821008#68...Ack
Packet received: OK
Sending packet: $mc0016de8,4#f8...Ack
Packet received: 38600001
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
Sending packet: $Mc0016de8,4:7d821008#e0...Ack
Packet received: OK
Sending packet: $mc005bd5c,4#23...Ack
Packet received: 38600001
Sending packet: $Mc005bd5c,4:7d821008#0b...Ack
Packet received: OK
Sending packet: $mc021a488,4#c8...Ack
Packet received: 4bfffbad
Sending packet: $Mc021a488,4:7d821008#b0...Ack
Packet received: OK
Sending packet: $c#63...Ack
    <<< program running, gdb waiting for event
```

Although it might look daunting at first, what is happening here is easily understood. In summary, `gdb` is restoring all its breakpoints on the target. Recall from Listing 14-3 that we entered two breakpoints, one at `panic()` and one at `sys_sync()`. Later in Listing 14-4, we added a third breakpoint at `yosemite_setup_arch()`. Thus, there are three active user-specified breakpoints. These can be displayed by issuing the `gdb info breakpoints` command. As usual, we use the abbreviated version.

```
(gdb) i b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0xc0016de8 in panic at kernel/panic.c:74
2   breakpoint     keep y   0xc005bd5c in sys_sync at fs/buffer.c:296
3   breakpoint     keep y   0xc021a488 in yosemite_setup_arch at
arch/ppc/platforms/4xx/yosemite.c:308
        breakpoint already hit 1 time
(gdb)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

Now compare the previous breakpoint addresses with the addresses in the `gdb` remote `$m` packet in Listing 14-5. The `$m` packet is a "read target memory" command, and the `$M` packet is a "write target memory" command. Once for each breakpoint, the address of the breakpoint is read from target memory, stored away locally on the host by `gdb` (so it can be restored later), and replaced with the PowerPC `trap` instruction `twge r2, r2` (`0x7d821008`), which results in control passing back to the debugger. Figure 14-4 illustrates this action.

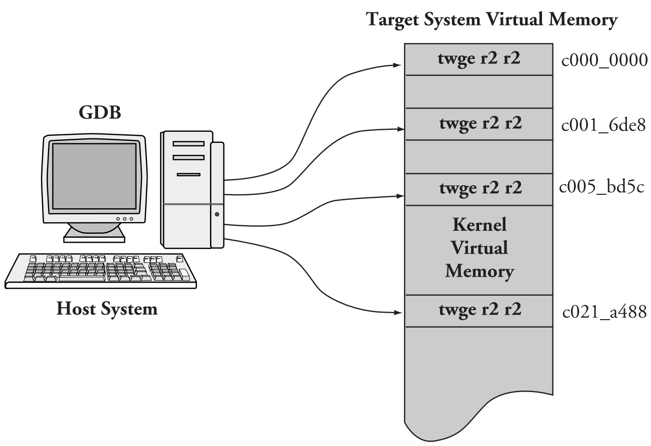


FIGURE 14-4
`gdb` inserting target memory breakpoints

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

You might have noticed that `gdb` is updating four breakpoints, whereas we entered only three. The first one at target memory location `0xc000_0000` is put there by `gdb` automatically upon startup. This location is the base address of the linked kernel image from the ELF file—essentially, `_start`. It is equivalent to a breakpoint at `main()` for user space debugging and is done by `gdb` automatically. The other three breakpoints are the ones we entered earlier.

The same thing happens in reverse when an event occurs that returns control to `gdb`. Listing 14-6 details the action when our breakpoint at `yosemite_setup_arch()` is encountered.

LISTING 14-6 Remote Protocol: Breakpoint Hit

```
Packet received: T0440:c021a488;01:c020ff90;
Sending packet: $mc0000000,4#80...Ack  <<< Read memory @c0000000
Packet received: 7d821008
Sending packet: $Mc0000000,4:c022d200#87...Ack  <<< Write memory
Packet received: OK
Sending packet: $mc0016de8,4#f8...Ack
Packet received: 7d821008
Sending packet: $Mc0016de8,4:38600001#a4...Ack
Packet received: OK
Sending packet: $mc005bd5c,4#23...Ack
Packet received: 7d821008
Sending packet: $Mc005bd5c,4:38600001#cf...Ack
Packet received: OK
Sending packet: $mc021a488,4#c8...Ack
Packet received: 7d821008
Sending packet: $Mc021a488,4:4bfffbad#d1...Ack
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3
Debugging the Linux Kernel

```
Packet received: OK

Sending packet: $mc021a484,c#f3...Ack
Packet received: 900100244bfffbad3fa0c022
Breakpoint 3, yosemite_setup_arch () at arch/ppc/platforms/4xx/yosemite.c:308
308            yosemite_set_emacdata();
(gdb)
```

The `$T` packet is a `gdb` Stop Reply packet. It is sent by the target to `gdb` when a breakpoint is encountered. In our example, the `$T` packet returned the value of the program counter and register `r1`.[4] The rest of the activity is the reverse of that in Listing 14-5. The PowerPC trap breakpoint instructions are removed, and `gdb` restores the original instructions to their respective memory locations.

14.3.2 Debugging Optimized Kernel Code

At the start of this chapter, we said that one of the challenges identified in debugging kernel code results from compiler optimization. We noted that the Linux kernel is compiled by default with optimization level `-O2`. In the examples up to this point, we used `-O1` optimization to simplify the debugging task. Here we illustrate one of the many ways optimization can complicate debugging.

The related Internet mail lists are strewn with questions related to what appear to be broken tools. Sometimes the poster reports that his debugger is single-stepping backward or that his line numbers do not line up with his source code. Here we present an example to illustrate the complexities that optimizing compilers bring to source-level debugging. In this example, the line

[4] *As pointed out earlier, the* `gdb` *remote protocol is detailed in the* `gdb` *manual cited at the end of this chapter in Section 14.6.1, "Suggestions for Additional Reading."*

SECTION 14.3

Debugging the Linux Kernel

numbers that gdb reports when a breakpoint is hit do not match up with the line numbers in our source file due to *function inlining*.

For this demonstration, we use the same debug code snippet as shown in Listing 14-4. However, for this example, we have compiled the kernel with the compiler optimization flag -O2. This is the default for the Linux kernel. Listing 14-7 shows the results of this debugging session.

LISTING 14-7 Optimized Architecture-Setup Code

```
$ ppc_44x-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }
(gdb) b panic
Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
(gdb) b sys_sync
Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
(gdb) b yosemite_setup_arch
Breakpoint 3 at 0xc020f438: file arch/ppc/platforms/4xx/yosemite.c, line 116.
(gdb) c
Continuing.

Breakpoint 3, yosemite_setup_arch ()
    at arch/ppc/platforms/4xx/yosemite.c:116
116             def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 0);
(gdb) l
```

SECTION 14.3

Debugging the Linux Kernel

```
111             struct ocp_def *def;
112             struct ocp_func_emac_data *emacdata;
113
114             /* Set mac_addr and phy mode for each EMAC */
115
116             def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 0);
117             emacdata = def->additions;
118             memcpy(emacdata->mac_addr, __res.bi_enetaddr, 6);
119             emacdata->phy_mode = PHY_MODE_RMII;
120
(gdb) p yosemite_setup_arch
$1 = {void (void)} 0xc020f41c <yosemite_setup_arch>
```

Referring back to Listing 14-4, notice that the function `yosemite_setup_arch()` actually falls on line 306 of the file `yosemite.c`. Compare that with Listing 14-7. We hit the breakpoint, but `gdb` reports the breakpoint at file `yosemite.c` line 116. It appears at first glance to be a mismatch of line numbers between the debugger and the corresponding source code. Is this a `gdb` bug? First let's confirm what the compiler produced for debug information. Using the `readelf`[5] tool described in Chapter 13, "Development Tools," we can examine the debug information for this function produced by the compiler.

```
$ ppc_44x-readelf --debug-dump=info vmlinux ¦ grep -u6 \
  yosemite_setup_arch ¦ tail -n 7
    DW_AT_name        : (indirect string, offset: 0x9c04): yosemite_setup_arch
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 307
```

[5] *Remember to use your cross-version of readelf—for example, ppc_44x-readelf for the PowerPC 44x architecture.*

SECTION 14.3

Debugging the Linux Kernel

```
DW_AT_prototyped  : 1
DW_AT_low_pc      : 0xc020f41c
DW_AT_high_pc     : 0xc020f794
DW_AT_frame_base  : 1 byte block: 51       (DW_OP_reg1)
```

We don't have to be experts at reading DWARF2 debug records[6] to recognize that the function in question is reported at line 307 in our source file. We can confirm this using the `addr2line` utility, also introduced in Chapter 13. Using the address derived from `gdb` in Listing 14-7:

```
$ ppc_44x-addr2line  -e vmlinux 0xc020f41c
    arch/ppc/platforms/4xx/yosemite.c:307
```

At this point, `gdb` is reporting our breakpoint at line 116 of the `yosemite.c` file. To understand what is happening, we need to look at the assembler output of the function as reported by `gdb`. Listing 14-8 is the output from `gdb` after issuing the `disassemble` command on the `yosemite_setup_arch()` function.

[6] A reference for the Dwarf debug specification appears at the end of this chapter in Section 14.6.1, "Suggestions for Additional Reading."

LISTING 14-8 Disassemble Function **`yosemite_setup_arch`**

```
(gdb) disassemble yosemite_setup_arch
0xc020f41c <yosemite_setup_arch+0>:     mflr    r0
0xc020f420 <yosemite_setup_arch+4>:     stwu    r1,-48(r1)
0xc020f424 <yosemite_setup_arch+8>:     li      r4,512
0xc020f428 <yosemite_setup_arch+12>:    li      r5,0
0xc020f42c <yosemite_setup_arch+16>:    li      r3,4116
0xc020f430 <yosemite_setup_arch+20>:    stmw    r25,20(r1)
0xc020f434 <yosemite_setup_arch+24>:    stw     r0,52(r1)
0xc020f438 <yosemite_setup_arch+28>:    bl      0xc000d344
```

SECTION 14.3

Debugging the Linux Kernel

```
<ocp_get_one_device>
0xc020f43c <yosemite_setup_arch+32>:    lwz     r31,32(r3)
0xc020f440 <yosemite_setup_arch+36>:    lis     r4,-16350
0xc020f444 <yosemite_setup_arch+40>:    li      r28,2
0xc020f448 <yosemite_setup_arch+44>:    addi    r4,r4,21460
0xc020f44c <yosemite_setup_arch+48>:    li      r5,6
0xc020f450 <yosemite_setup_arch+52>:    lis     r29,-16350
0xc020f454 <yosemite_setup_arch+56>:    addi    r3,r31,48
0xc020f458 <yosemite_setup_arch+60>:    lis     r25,-16350
0xc020f45c <yosemite_setup_arch+64>:    bl      0xc000c708
<memcpy>
0xc020f460 <yosemite_setup_arch+68>:    stw     r28,44(r31)
0xc020f464 <yosemite_setup_arch+72>:    li      r4,512
0xc020f468 <yosemite_setup_arch+76>:    li      r5,1
0xc020f46c <yosemite_setup_arch+80>:    li      r3,4116
0xc020f470 <yosemite_setup_arch+84>:    addi    r26,r25,15104
0xc020f474 <yosemite_setup_arch+88>:    bl      0xc000d344
<ocp_get_one_device>
0xc020f478 <yosemite_setup_arch+92>:    lis     r4,-16350
0xc020f47c <yosemite_setup_arch+96>:    lwz     r31,32(r3)
0xc020f480 <yosemite_setup_arch+100>:   addi    r4,r4,21534
0xc020f484 <yosemite_setup_arch+104>:   li      r5,6
0xc020f488 <yosemite_setup_arch+108>:   addi    r3,r31,48
0xc020f48c <yosemite_setup_arch+112>:   bl      0xc000c708
<memcpy>
0xc020f490 <yosemite_setup_arch+116>:   lis     r4,1017
0xc020f494 <yosemite_setup_arch+120>:   lis     r5,168
```

SECTION 14.3

Debugging the Linux Kernel

```
0xc020f498 <yosemite_setup_arch+124>:   stw     r28,44(r31)
0xc020f49c <yosemite_setup_arch+128>:   ori     r4,r4,16554
0xc020f4a0 <yosemite_setup_arch+132>:   ori     r5,r5,49152
0xc020f4a4 <yosemite_setup_arch+136>:   addi    r3,r29,-15380
0xc020f4a8 <yosemite_setup_arch+140>:   addi    r29,r29,-15380
0xc020f4ac <yosemite_setup_arch+144>:   bl      0xc020e338
<ibm440gx_get_clocks>
0xc020f4b0 <yosemite_setup_arch+148>:   li      r0,0
0xc020f4b4 <yosemite_setup_arch+152>:   lis     r11,-16352
0xc020f4b8 <yosemite_setup_arch+156>:   ori     r0,r0,50000
0xc020f4bc <yosemite_setup_arch+160>:   lwz     r10,12(r29)
0xc020f4c0 <yosemite_setup_arch+164>:   lis     r9,-16352
0xc020f4c4 <yosemite_setup_arch+168>:   stw     r0,8068(r11)
0xc020f4c8 <yosemite_setup_arch+172>:   lwz     r0,84(r26)
0xc020f4cc <yosemite_setup_arch+176>:   stw     r10,8136(r9)
0xc020f4d0 <yosemite_setup_arch+180>:   mtctr   r0
0xc020f4d4 <yosemite_setup_arch+184>:   bctrl
0xc020f4d8 <yosemite_setup_arch+188>:   li      r5,64
0xc020f4dc <yosemite_setup_arch+192>:   mr      r31,r3
0xc020f4e0 <yosemite_setup_arch+196>:   lis     r4,-4288
0xc020f4e4 <yosemite_setup_arch+200>:   li      r3,0
0xc020f4e8 <yosemite_setup_arch+204>:   bl      0xc000c0f8
<ioremap64>
End of assembler dump.
(gdb)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

Once again, we need not be PowerPC assembly language experts to understand what is happening here. Notice the labels associated with the PowerPC `bl` instruction. This is a function call in PowerPC mnemonics. The symbolic function labels are the important data points. After a cursory analysis, we see several function calls near the start of this assembler listing:

Address	Function
0xc020f438	ocp_get_one_device()
0xc020f45c	memcpy()
0xc020f474	ocp_get_one_device()
0xc020f48c	memcpy()
0xc020f4ac	ibm440gx_get_clocks()

Listing 14-9 reproduces portions of the source file `yosemite.c`. Correlating the functions we found in the `gdb disassemble` output, we see those labels occurring in the function `yosemite_set_emacdata()`, around the line numbers reported by `gdb` when the breakpoint at `yosemite_setup_arch()` was encountered. The key to understanding the anomaly is to notice the subroutine call at the very start of `yosemite_setup_arch()`. The compiler has inlined the call to `yosemite_set_emacdata()`instead of generating a function call, as would be expected by simple inspection of the source code. This inlining produced the mismatch in the line numbers when `gdb` hit the breakpoint. Even though the `yosemite_set_emacdata()` function was not declared using the inline keyword, GCC inlined the function as a performance optimization.

LISTING 14-9 Portions of Source File **yosemite.c**

```
static void __init yosemite_set_emacdata(void)
{
        struct ocp_def *def;
        struct ocp_func_emac_data *emacdata;

        /* Set mac_addr and phy mode for each EMAC */

        def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 0);
        emacdata = def->additions;
        memcpy(emacdata->mac_addr, __res.bi_enetaddr, 6);
        emacdata->phy_mode = PHY_MODE_RMII;

        def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 1);
        emacdata = def->additions;
        memcpy(emacdata->mac_addr, __res.bi_enet1addr, 6);
        emacdata->phy_mode = PHY_MODE_RMII;
}

...

static void __init
yosemite_setup_arch(void)
{
        yosemite_set_emacdata();

```


Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

```
        ibm440gx_get_clocks(&clocks, YOSEMITE_SYSCLK, 6 * 1843200);
        ocp_sys_info.opb_bus_freq = clocks.opb;

        /* init to some ~sane value until calibrate_delay() runs */
        loops_per_jiffy = 50000000/HZ;

        /* Setup PCI host bridge */
        yosemite_setup_hose();

#ifdef CONFIG_BLK_DEV_INITRD
        if (initrd_start)
                ROOT_DEV = Root_RAM0;
        else
#endif
#ifdef CONFIG_ROOT_NFS
                ROOT_DEV = Root_NFS;
#else
                ROOT_DEV = Root_HDA1;
#endif

        yosemite_early_serial_map();

        /* Identify the system */
        printk( "AMCC PowerPC " BOARDNAME " Platform\n" );
}
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

To summarize the previous discussion:

- We entered a breakpoint in `gdb` at `yosemite_setup_arch()`.
- When the breakpoint was hit, we found ourselves at line 116 of the source file, which was far removed from the function where we defined the breakpoint.
- We produced a disassembly listing of the code at `yosemite_setup_arch()` and discovered the labels to which this sequence of code was branching.
- Comparing the labels back to our source code, we discovered that the compiler had placed the `yosemite_set_emacdata()` subroutine inline with the function where we entered a breakpoint, causing potential confusion.

This explains the line numbers reported by `gdb` when the original breakpoint in `yosemite_setup_arch()` was hit.

Compilers employ many different kinds of optimization algorithms. This example presented but one: function inlining. Each can confuse a debugger (the human and the machine) in a different way. The challenge is to understand what is happening at the machine level and translate that into what we as developers had intended. You can see now the benefits of using the minimum possible optimization level for debugging.

14.3.3 gdb User-Defined Commands

You might already realize that `gdb` looks for an initialization file on startup, called `.gdbinit`. When first invoked, `gdb` loads this initialization file (usually found in the user's home directory) and acts on the commands within it. One of my favorite combinations is to connect to the target system and set initial breakpoints. In this case, the contents of `.gdbinit` would look like Listing 14-10.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

LISTING 14-10 Simple **gdb** Initialization File

```
$ cat ~/.gdbinit
set history save on
set history filename ~/.gdb_history
set output-radix 16

define connect
#   target remote bdi:2001
    target remote /dev/ttyS0
    b panic
    b sys_sync
end
```

This simple `.gdbinit` file enables the storing of command history in a user-specified file and sets the default output radix for printing of values. Then it defines a `gdb` *user-defined command* called `connect`. (User-defined commands are also often called macros.) When issued at the `gdb` command prompt, `gdb` connects to the target system via the desired method and sets the system breakpoints at `panic()` and `sys_sync()`. One method is commented out; we discuss this method shortly in Section 14.4.

There is no end to the creative use of `gdb` user-defined commands. When debugging in the kernel, it is often useful to examine global data structures such as task lists and memory maps. Here we present several useful `gdb` user-defined commands capable of displaying specific kernel data that you might need to access during your kernel debugging.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

14.3.4 Useful Kernel gdb Macros

During kernel debugging, it is often useful to view the processes that are running on the system, as well as some common attributes of those processes. The kernel maintains a linked list of tasks described by `struct task_struct`. The address of the first task in the list is contained in the kernel global variable `init_task`, which represents the initial task spawned by the kernel during startup. Each task contains a `struct list_head`, which links the tasks in a circular linked list. These two ubiquitous kernel structures are described in the following header files:

```
struct task_struct           .../include/linux/sched.h
struct list_head             .../include/linux/list.h
```

Using `gdb` macros, we can traverse the task list and display useful information about the tasks. It is easy to modify the macros to extract the data you might be interested in. It is also a very useful tool for learning the details of kernel internals.

The first macro we examine is a simple one that searches the kernel's linked list of `task_struct` structures until it finds the given task. If it is found, it displays the name of the task.

LISTING 14-11 **gdb `find_task` Macro**

```
# Helper function to find a task given a PID or the
# address of a task_struct.
# The result is set into $t
define find_task
  # Addresses greater than _end: kernel data...
  # ...user passed in an address
  if ((unsigned)$arg0 > (unsigned)&_end)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
    set $t=(struct task_struct *)$arg0
  else
    # User entered a numeric PID
    # Walk the task list to find it
    set $t=&init_task
    if (init_task.pid != (unsigned)$arg0)
      find_next_task $t
      while (&init_task!=$t && $t->pid != (unsigned)$arg0)
        find_next_task $t
      end
      if ($t == &init_task)
        printf "Couldn't find task; using init_task\n"
      end
    end
  end
  printf "Task \"%s\":\n", $t->comm
end
```

Place this text into your `.gdbinit` file and restart `gdb`, or *source*[7] it using `gdb's source` command. (We explain the `find_next_task` macro later in Listing 14-15.) Invoke it as follows:

```
(gdb) find_task 910
  Task "syslogd":
```

or

```
(gdb) find_task 0xCFFDE470
  Task "bash":
```

[7] *A helpful shortcut for macro development is the* `gdb source` *command. This command opens and reads a source file containing macro definitions.*

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

Line 4 defines the macro name. Line 7 decides whether the input argument is a PID (numeric entry starting at zero and limited to a few million) or a `task_struct` address that must be greater than the end of the Linux kernel image itself, defined by the symbol `_end`.[8] If it's an address, the only action required is to cast it to the proper type to enable dereferencing the associated `task_struct`. This is done at line 8. As the comment in line 3 states, this macro returns a `gdb` convenience variable typecasted to a pointer to a `struct task_struct`.

If the input argument is a numeric PID, the list is traversed to find the matching `task_struct`. Lines 12 and 13 initialize the loop variables (`gdb` does not have a `for` statement in its macro command language), and lines 15 through 17 define the search loop. The `find_next_task` macro is used to extract the pointer to the next `task_struct` in the linked list. Finally, if the search fails, a sane return value is set (the address of `init_task`) so that it can be safely used in other macros.

Building on the `find_task` macro in Listing 14-11, we can easily create a simple `ps` command that displays useful information about each process running on the system.

Listing 14-12 defines a `gdb` macro that displays interesting information from a running process, extracted from the `struct task_struct` for the given process. It is invoked like any other `gdb` command, by typing its name followed by any required input parameters. Notice that this user-defined command requires a single argument, either a PID or the address of a `task_struct`.

LISTING 14-12 **gdb Macro: Print Process Information**

```
define ps
  # Print column headers
  task_struct_header
  set $t=&init_task
```

[8] *The symbol _end is defined in the linker script file during the final link.*

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

```
 5   task_struct_show $t
 6   find_next_task $t
 7   # Walk the list
 8   while &init_task!=$t
 9     # Display useful info about each task
10     task_struct_show $t
11     find_next_task $t
12   end
13 end
14
15 document ps
16 Print points of interest for all tasks
17 end
```

This `ps` macro is similar to the `find_task` macro, except that it requires no input arguments and it adds a macro (`task_struct_show`) to display the useful information from each `task_struct`. Line 3 prints a banner line with column headings. Lines 4 through 6 set up the loop and display the first task. Lines 8 through 11 loop through each task, calling the `task_struct_show` macro for each.

Notice also the inclusion of the `gdb document` command. This allows the `gdb` user to get help by issuing the `help ps` command from the `gdb` command prompt as follows:

```
(gdb) help ps
   Print points of interest for all tasks
```

Listing 14-13 displays the output of this macro on a target board running only minimal services.

SECTION 14.3

Debugging the Linux Kernel

LISTING 14-13 gdb ps Macro Output

```
(gdb) ps
Address      PID State      User_NIP  Kernel-SP  device comm
0xC01D3750     0 Running              0xC0205E90 (none) swapper
0xC04ACB10     1 Sleeping  0x0FF6E85C 0xC04FFCE0 (none) init
0xC04AC770     2 Sleeping             0xC0501E90 (none) ksoftirqd/0
0xC04AC3D0     3 Sleeping             0xC0531E30 (none) events/0
0xC04AC030     4 Sleeping             0xC0533E30 (none) khelper
0xC04CDB30     5 Sleeping             0xC0535E30 (none) kthread
0xC04CD790    23 Sleeping             0xC06FBE30 (none) kblockd/0
0xC04CD3F0    45 Sleeping             0xC06FDE50 (none) pdflush
0xC04CD050    46 Sleeping             0xC06FFE50 (none) pdflush
0xC054B7B0    48 Sleeping             0xC0703E30 (none) aio/0
0xC054BB50    47 Sleeping             0xC0701E20 (none) kswapd0
0xC054B410   629 Sleeping             0xC0781E60 (none) kseriod
0xC054B070   663 Sleeping             0xCFC59E30 (none) rpciod/0
0xCFFDE0D0   675 Sleeping  0x0FF6E85C 0xCF86DCE0 (none) udevd
0xCF95B110   879 Sleeping  0x0FF0BE58 0xCF517D80 (none) portmap
0xCFC24090   910 Sleeping  0x0FF6E85C 0xCF61BCE0 (none) syslogd
0xCF804490   918 Sleeping  0x0FF66C7C 0xCF65DD70 (none) klogd
0xCFE350B0   948 Sleeping  0x0FF0E85C 0xCF67DCE0 (none) rpc.statd
0xCFFDE810   960 Sleeping  0x0FF6E85C 0xCF5C7CE0 (none) inetd
0xCFC24B70   964 Sleeping  0x0FEEBEAC 0xCF64FD80 (none) mvltd
0xCFE35B90   973 Sleeping  0x0FF66C7C 0xCFEF7CE0 ttyS1  getty
0xCFE357F0   974 Sleeping  0x0FF4B85C 0xCF6EBCE0 (none) in.telnetd
0xCFFDE470   979 Sleeping  0x0FEB6950 0xCF675DB0 ttyp0  bash
0xCFFDEBB0   982<Running   0x0FF6EB6C 0xCF7C3870 ttyp0  sync
(gdb)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION **14.3**

Debugging the Linux Kernel

The bulk of the work done by this `ps` macro is performed by the `task_struct_show` macro. As shown in Listing 14-13, the `task_struct_show` macro displays the following fields from each `task_struct`:

- `Address`—Address of the `task_struct` for the process
- `PID`—Process ID
- `State`—Current state of the process
- `User_NIP`—Userspace Next Instruction Pointer
- `Kernel_SP`—Kernel Stack Pointer
- `device`—Device associated with this process
- `comm`—Name of the process (or command)

It is relatively easy to modify the macro to show the items of interest for your particular kernel debugging task. The only complexity is in the simplicity of the macro language. Because function equivalents such as `strlen` do not exist in `gdb`'s user-defined command language, screen formatting must be done by hand.

Listing 14-14 reproduces the `task_struct_show` macro that produced the previous listing.

LISTING 14-14 **`gdb task_struct_show` Macro**

```
define task_struct_show
  # task_struct addr and PID
  printf "0x%08X %5d", $arg0, $arg0->pid

  # Place a '<' marker on the current task
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
#  if ($arg0 == current)
# For PowerPC, register r2 points to the "current" task
if ($arg0 == $r2)
  printf "<"
else
  printf " "
end

# State
if ($arg0->state == 0)
  printf "Running   "
else
  if ($arg0->state == 1)
    printf "Sleeping  "
  else
    if ($arg0->state == 2)
      printf "Disksleep "
    else
      if ($arg0->state == 4)
        printf "Zombie    "
      else
        if ($arg0->state == 8)
          printf "sTopped   "
        else
          if ($arg0->state == 16)
            printf "Wpaging   "
          else
            printf "%2d        ", $arg0->state
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
            end
          end
        end
      end
    end
  end

  # User NIP
  if ($arg0->thread.regs)
    printf "0x%08X ", $arg0->thread.regs->nip
  else
    printf "           "
  end

  # Display the kernel stack pointer
  printf "0x%08X ", $arg0->thread.ksp

  # device
  if ($arg0->signal->tty)
    printf "%s  ", $arg0->signal->tty->name
  else
    printf "(none) "
  end

  # comm
  printf "%s\n", $arg0->comm
end
```

SECTION 14.3

Debugging the Linux Kernel

Line 3 displays the address of the `task_struct`. Lines 8 through 12 display the process ID. If this is the current process (the process that was currently running on this CPU at the time the breakpoint was hit), it is marked with a < character.

Lines 14 through 39 decode and display the state of the process. This is followed by displaying the user process next instruction pointer (NIP) and the kernel stack pointer (SP). Finally, the device associated with the process is displayed, followed by the name of the process (stored in the `->comm` element of the `task_struct`.)

It is important to note that this macro is architecture dependent, as shown in lines 7 and 8. In general, macros such as these are highly architecture- and version-dependent. Any time a change in the underlying structure is made, macros such as these must be updated. However, if you spend a lot of time debugging the kernel using `gdb`, the payback is often worth the effort.

For completeness, we present the `find_next_task` macro. Its implementation is less than obvious and deserves explanation. (It is assumed that you can easily deduce the `task_struct_header` that completes the series necessary for the `ps` macro presented in this section. It is nothing more than a single line arranging the column headers with the correct amount of whitespace.) Listing 14-15 presents the `find_next_task` macro used in our `ps` and `find_task` macros.

LISTING 14-15 gdb `find_next_task` Macro

```
define find_next_task
  # Given a task address, find the next task in the linked list
  set $t = (struct task_struct *)$arg0
  set $offset=( (char *)&$t->tasks - (char *)$t)
  set $t=(struct task_struct *)( (char *)$t->tasks.next- (char *)$offset)
end
```

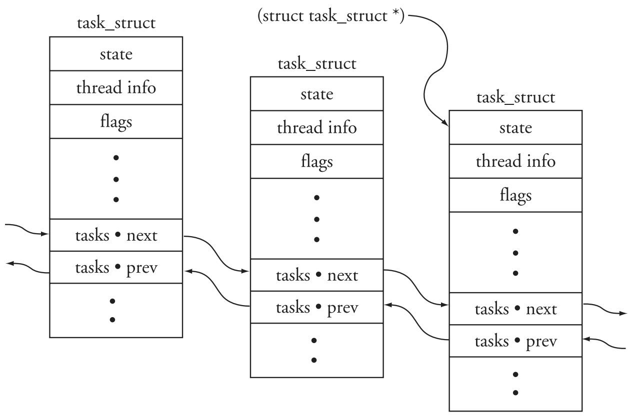

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

The function performed by this macro is simple. The implementation is slightly less than straightforward. The goal is to return the `->next` pointer, which points to the next `task_struct` on the linked list. However, the `task_struct` structures are linked by the address of the `struct list_head` member called `tasks`, as opposed to the common practice of being linked by the starting address of the `task_struct` itself. Because the `->next` pointer points to the address of the `task` structure element in the next `task_struct` on the list, we must subtract to get the address of the top of the `task_struct` itself. The value we subtract from the `->next` pointer is the offset from that pointer's address to the top of `task_struct`. First we calculate the offset and then we use that offset to adjust the `->next` pointer to point to the top of `task_struct`. Figure 14-5 should make this clear.

FIGURE 14-5
Task structure list linking



Now we present one final macro that will be useful in the next section when we discuss debugging loadable modules. Listing 14-16 is a simple macro that displays the kernel's list of currently installed loadable modules.

LISTING 14-16 gdb List Modules Macro

```
define lsmod
  printf "Address\t\tModule\n"
  set $m=(struct list_head *)&modules
  set $done=0
  while ( !$done )
    # list_head is 4-bytes into struct module
    set $mp=(struct module *)((char *)$m->next - (char *)4)
    printf "0x%08X\t%s\n", $mp, $mp->name
    if ( $mp->list->next == &modules)
      set $done=1
    end
    set $m=$m->next
  end
end

document lsmod
List the loaded kernel modules and their start addresses
end
```

This simple loop starts with the kernel's global variable `module`. This variable is a `struct list_head` that marks the start of the linked list of loadable modules. The only complexity is the same as that described in Listing 14-15. We must subtract an offset from the `struct list_head` pointer to point to the top of the `struct module`. This is performed in line 7. This macro produces a simple listing of modules containing the address of the `struct module` and the module's name. Here is an example of its use:

```
(gdb) lsmod
Address         Module
0xD1012A80      ip_conntrack_tftp
0xD10105A0      ip_conntrack
0xD102F9A0      loop
(gdb) help lsmod
List the loaded kernel modules and their start addresses
(gdb)
```

Macros such as the ones presented here are very powerful debugging aids. You can create macros in a similar fashion to display anything in the kernel that lends itself to easy access, especially the major data structures maintained as linked lists. Examples include process memory map information, module information, file system information, and timer lists and so on. The information presented here should get you started.

14.3.5 Debugging Loadable Modules

The most common reason for using KGDB is to debug loadable kernel modules, that is, device drivers. One of the more convenient features of loadable modules is that, under most circumstances, it is not necessary to reboot the kernel for each new debugging session. You can start a debugging session, make some changes, recompile, and reload the module without the hassle and delay of a complete kernel reboot.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

The complication associated with debugging loadable modules is in gaining access to the symbolic debug information contained in the module's object file. Because loadable modules are dynamically linked when they are loaded into the kernel, the symbolic information contained in the object file is useless until the symbol table is adjusted.

Recall from our earlier examples how we invoke `gdb` for a kernel debugging session:

```
$ ppc_4xx-gdb vmlinux
```

This launches a `gdb` debugging session on your host, and reads the symbol information from the Linux kernel ELF file `vmlinux`. Of course, you will not find symbols for any loadable modules in this file. Loadable modules are separate compilation units and are linked as individual standalone ELF objects. Therefore, if we intend to perform any source-level debugging on a loadable module, we need to load its debug symbols from the ELF file. `gdb` provides this capability in its `add-symbol-file` command.

The `add-symbol-file` command loads symbols from the specified object file, assuming that the module itself has already been loaded. However, we are faced with the chicken-and-egg syndrome. We don't have any symbol information until the loadable module has been loaded into the kernel and the `add-symbol-file` command is issued to read in the module's symbol information. However, after the module has been loaded, it is too late to set breakpoints and debug the module's `*_init` and related functions because they have already executed.

The solution to this dilemma is to place a breakpoint in the kernel code that is responsible for loading the module, after it has been linked but before its initialization function has been called. This work is done by `.../kernel/module.c`. Listing 14-17 reproduces the relevant portions of `module.c`.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

LISTING 14-17 **module.c: Module Initialization**

```
...
1901        down(&notify_mutex);
1902       notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod);
1903        up(&notify_mutex);
1904
1905        /* Start the module */
1906        if (mod->init != NULL)
1907                ret = mod->init();
1908        if (ret < 0) {
1909                /* Init routine failed: abort.  Try to protect us from
1910                   buggy refcounters. */
1911                mod->state = MODULE_STATE_GOING;
...
```

We load the module using the `modprobe` utility, which was demonstrated in Listing 8-5 in Chapter 8, "Device Driver Basics," and looks like this:

```
$ modprobe loop
```

This command issues a special system call that directs the kernel to load the module. The module loading begins at `sys_init_module()` in `module.c`. After the module has been loaded into kernel memory and dynamically linked, control is passed to the module's `_init` function. This is shown in lines 1906 and 1907 of Listing 14-17. We place our breakpoint here. This enables us to add the symbol file to `gdb` and subsequently set breakpoints in the module. We demonstrate this process using the Linux kernel's loopback driver called `loop.ko`. This module has no dependencies on other modules and is reasonably easy to demonstrate.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

Listing 14-18 shows the `gdb` commands to initiate this debugging session on `loop.ko`.

LISTING 14-18 Initiate Module Debug Session: **`loop.ko`**

```
$ ppc-linux-gdb --silent vmlinux
(gdb) connect
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }
Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
(gdb) b module.c:1907
Breakpoint 3 at 0xc003430c: file kernel/module.c, line 1907.
(gdb) c
Continuing.
>>>> Here we let the kernel finish booting
     and then load the loop.ko module on the target

Breakpoint 3, sys_init_module (umod=0x30029000, len=0x2473e,
    uargs=0x10016338 "") at kernel/module.c:1907
1907                    ret = mod->init();
(gdb) lsmod
Address         Module
0xD102F9A0      loop
(gdb) set $m=(struct module *)0xD102F9A0.
(gdb) p $m->module_core
$1 = (void *) 0xd102c000
(gdb) add-symbol-file ./drivers/block/loop.ko 0xd102c000
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

```
24 add symbol table from file "./drivers/block/loop.ko" at
25         .text_addr = 0xd102c000
26 (y or n) y
27 Reading symbols from /home/chris/sandbox/linux-2.6.13-amcc/
drivers/block        /loop.ko...done.
```

Starting with line 2, we use the `gdb` user-defined macro `connect` created earlier in Listing 14-10 to connect to the target board and set our initial breakpoints. We then add the breakpoint in `module.c`, as shown in line 7, and we issue the `continue` command `(c)`. Now the kernel completes the boot process and we establish a `telnet` session into the target and load the `loop.ko` module (not shown). When the loopback module is loaded, we immediately hit breakpoint #3. `gdb` then displays the information shown in lines 14 through 16.

At this point, we need to discover the address where the Linux kernel linked our module's `.text` section. Linux stores this address in the module information structure `struct module` in the `module_core` element. Using the `lsmod` macro we defined in Listing 14-16, we obtain the address of the `struct module` associated with our `loop.ko` module. This is shown in lines 17 through 19. Now we use this structure address to obtain the module's `.text` address from the `module_core` structure member. We pass this address to the `gdb` `add-symbol-file` command, and `gdb` uses this address to adjust its internal symbol table to match the actual addresses where the module was linked into the kernel. From there, we can proceed in the usual manner to set breakpoints in the module, step through code, examine data, and so on.

We conclude this section with a demonstration of placing a breakpoint in the loopback module's initialization function so that we can step through the module's initialization code. The complication here is that the kernel loads the module's initialization code into a separately allocated

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

portion of memory so that it can be freed after use. Recall from Chapter 5, "Kernel Initialization," our discussion of the `__init` macro. This macro expands into a compiler attribute that directs the linker to place the marked portion of code into a specially named ELF section. In essence, any function defined with this attribute is placed in a separate ELF section named `.init.text`. Its use is similar to the following:

```
static int __init loop_init(void){...}
```

This invocation would place the compiled `loop_init()` function into the `.init.text` section of the `loop.ko` object module. When the module is loaded, the kernel allocates a chunk of memory for the main body of the module, which is pointed to by the `struct module` member named `module_core`. It then allocates a separate chunk of memory to hold the `.init.text` section. After the initialization function is called, the kernel frees the memory that contained the initialization function. Because the object module is split like this, we need to inform `gdb` of this addressing scheme to be able to use symbolic data for debugging the initialization function.[9] Listing 14-19 demonstrates these steps.

LISTING 14-19 Debugging Module `init` Code

```
$ ppc_4xx-gdb -slient vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }

<< Place a breakpoint before calling module init >>
(gdb) b module.c:1907
```

[9] *As of this writing, there is a bug in* `gdb` *that prevents this technique from working properly. Hopefully, by the time you read this, it will be fixed.*

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
Breakpoint 1 at 0xc0036418: file kernel/module.c, line 1907.
(gdb) c
Continuing.

Breakpoint 1, sys_init_module (umod=0xd102ef40, len=0x23cb3, uargs=0x10016338 "") at kernel/
module.c:1907
1907                    ret = mod->init();

<< Discover init addressing from struct module >>
(gdb) lsmod
Address         Module
0xD102EF40      loop
(gdb) set $m=(struct module *)0xD102EF40
(gdb) p $m->module_core
$1 = (void *) 0xd102b000
(gdb) p $m->module_init
$2 = (void *) 0xd1031000

<< Now load a symbol file using the core and init addrs >>
(gdb) add-symbol-file ./drivers/block/loop.ko 0xd102b000 -s .init.text 0xd1031000
add symbol table from file "./drivers/block/loop.ko" at
        .text_addr = 0xd102b000
        .init.text_addr = 0xd1031000
(y or n) y
Reading symbols from /home/chris/sandbox/linux-2.6.13-amcc/drivers/block/loop.ko...done.
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
(gdb) b loop_init
Breakpoint 3 at 0xd1031000: file drivers/block/loop.c, line 1244.
(gdb) c
Continuing.

<< Breakpoint hit, proceed to debug module init function >>
Breakpoint 3, 0xd1031000 in loop_init () file drivers/block/loop.c, line 1244
1244        if (max_loop < 1 ¦¦ max_loop > 256) {
(gdb)
```

14.3.6 printk Debugging

Debugging kernel and device driver code using `printk` is a popular technique, mostly because `printk` has evolved into a very robust method. You can call `printk` from almost any context, including from interrupt handlers. `printk` is the kernel's version of the familiar `printf()` C library function. `printk` is defined in `.../kernel/printk.c`.

It is important to understand the limitations of using `printk` for debugging. First, `printk` requires a console device. Moreover, although the console device is configured as early as possible during kernel initialization, there are many calls to `printk` before the console device has been initialized. We present a method to cope with this limitation later, in Section 14.5, "When It Doesn't Boot."

The `printk` function allows the addition of a string marker that identifies the level of severity of a given message. The header file `.../include/linux/kernel.h` defines eight levels:

```
#define     KERN_EMERG    "<0>" /* system is unusable */
#define     KERN_ALERT    "<1>" /* action must be taken immediately */
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

```
#define     KERN_CRIT     "<2>" /* critical conditions */
#define     KERN_ERR      "<3>" /* error conditions */
#define     KERN_WARNING  "<4>" /* warning conditions */
#define     KERN_NOTICE   "<5>" /* normal but significant condition */
#define     KERN_INFO     "<6>" /* informational */
#define     KERN_DEBUG    "<7>" /* debug-level messages */
```

A simple `printk` message might look like this:

```
printk("foo() entered w/ %s\n", arg);
```

If the severity string is omitted, the kernel assigns a default severity level, which is defined in `printk.c`. In recent kernels, this is set at severity level 4, `KERN_WARNING`. Specifying `printk` with a severity level might look something like this:

```
printk(KERN_CRIT "vmalloc failed in foo()\n");
```

By default, all `printk` messages below a predefined *loglevel* are displayed on the system console device. The default loglevel is defined in `printk.c`. In recent Linux kernels, it has the value `7`. This means that any `printk` message that is greater in importance than `KERN_DEBUG` will be displayed on the console.

You can set the default kernel loglevel in a variety of ways. At boot time, you can specify the default loglevel on your target board by passing the appropriate kernel command-line parameters to the kernel at boot time. Three kernel command line options defined in `main.c` affect the default loglevel:

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

- `debug`—Sets the console loglevel to 10
- `quiet`—Sets the console loglevel to 4
- `loglevel=`—Sets the console loglevel to your choice of value

Using `debug` effectively displays every `printk` message. Using `quiet` displays all `printk` messages of severity `KERN_ERR` or higher.

`printk` messages can be logged to files on your target or via the network. Use `klogd` (kernel log daemon) and `syslogd` (system log daemon) to control the logging behavior of `printk` messages. These popular utilities are described in man pages and many Linux references, and are not described here.

14.3.7 Magic SysReq Key

This useful debugging aid is invoked through a series of special predefined key sequences that send messages directly to the kernel. For many target architectures and boards, you use a simple terminal emulator on a serial port as a system console. For these architectures, the Magic SysReq key is defined as a break character followed by a command character. Consult the documentation on the terminal emulator you use for how to send a break character. Many Linux developers use the `minicom` terminal emulator. For `minicom`, the break character is sent by typing `Ctl-A F`. After sending the break in this manner, you have 5 seconds to enter the command character before the command times out.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.3

Debugging the Linux Kernel

This useful kernel tool can be very helpful for development and debugging, but it can also cause data loss and system corruption. Indeed, the `b` command immediately reboots your system without any notification or preparation. Open files are not closed, disks are not synced, and file systems are not unmounted. When the reboot (`b`) command is issued, control is immediately passed to the reset vector of your architecture in a most abrupt and stunning manner. Use this powerful tool at your own peril!

This feature is well documented in the Linux kernel documentation subdirectory in a file called `sysrq.txt`. There you find the details for many architectures and the description of available commands.

For example, another way to set the kernel loglevel just discussed is to use the Magic SysReq key. The command is a number from 0 through 9, which results in the default loglevel being set to the number of the command. From minicom, press `Ctl-A F` followed by a number, such as 9. Here is how it looks on the terminal:

```
$ SysRq : Changing Loglevel
   Loglevel set to 9
```

Commands can be used to dump registers, shut down your system, reboot your system, dump a list of processes, dump current memory information to your console, and more. See the documentation file in any recent Linux kernel for the details.

This feature is most commonly used when something causes your system to lock up. Often the Magic SysReq key provides a way to learn something from an otherwise dead system.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4

Hardware-Assisted Debugging

14.4 Hardware-Assisted Debugging

By now you've probably realized that you cannot debug very early kernel-startup code with KGDB. This is because KGDB is not initialized until after most of the low-level hardware-initialization code has executed. Furthermore, if you are assigned the task of bringing up a brand-new board design and porting a bootloader and the Linux kernel, having a hardware-debug probe is without a doubt the most efficient means of debugging problems in these early stages of board porting.

You can choose from a wide variety of hardware-debug probes. For the examples in this section, we use a unit manufactured by Abatron called the BDI-2000 (see www.abatron.ch). These units are often called JTAG probes because they use a low-level communications method that was first employed for boundary scan testing of integrated circuits defined by the Joint Test Action Group (JTAG).

A JTAG probe contains a small connector designed for connection to your target board. It is often a simple square-pin header and ribbon cable arrangement. Most modern high-performance CPUs contain a JTAG interface that is designed to provide this software debugging capability. The JTAG probe connects to this CPU JTAG interface. The other side of the JTAG probe connects to your host development system usually via Ethernet, USB, or a parallel port. Figure 14-6 details the setup for the Abatron unit.

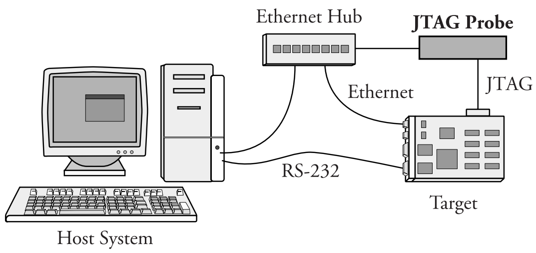
JTAG probes can be complicated to set up. This is a direct result of the complexity of the CPU to which it is connected. When power is applied to a target board and its CPU comes out of reset, almost nothing is initialized. In fact, many processors need at least a small amount of initialization before they can do anything. Many methods are available for getting this initial configuration into the CPU. Some CPUs read a hardware-configuration word or initial values of specific pins to learn their power-on configuration. Others rely on reading a default location in a simple nonvolatile

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4
Hardware-Assisted Debugging

storage device such as Flash. When using a JTAG probe, especially for bringing up a new board design, a minimum level of CPU and board initialization must be performed before anything else can be done. Many JTAG probes rely on a configuration file for this initialization.

FIGURE 14-6
Hardware JTAG probe debugging



The Abatron unit uses a configuration file to initialize the target hardware it is connected to, as well as to define other operational parameters of the debugger. This configuration file contains directives that initialize the CPU, memory system, and other necessary board-level hardware. It is the developer's responsibility to customize this configuration file with the proper directives for his own board. The details on the configuration command syntax can be found in the JTAG probe's documentation. However, only the embedded developer can create the unique configuration file required for a given board design. This requires detailed knowledge of the CPU and board-level design features. Much like creating a custom Linux port for a new board, there is no shortcut or substitute for this task.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4

Hardware-Assisted Debugging

Appendix F, "Sample BDI-2000 Configuration File," contains a sample Abatron configuration file for a custom board based on the Freescale Semiconductor MPC5200 embedded controller. In that appendix, you can see the necessary setup for a custom board. Notice the liberal use of comments describing various registers and initialization details. This makes it easier to update and maintain over time, and it can help you to get it right the first time.

Hardware probes are generally used in two ways. Most have a user interface of some type that enables the developer to use features of the probe. Examples of this are to program Flash or download binary images. The second usage is as a front end to `gdb` or other source-level debuggers. We demonstrate both usage scenarios.

14.4.1 Programming Flash Using a JTAG Probe

Many hardware probes include the capability to program a wide variety of Flash memory chips. The Abatron BDI-2000 is no exception. The BDI-2000 configuration file includes a `[FLASH]` section to define the characteristics of the target Flash. Refer to Appendix F for a sample. The `[FLASH]` section defines attributes of the Flash chip as used in a particular design, such as the chip type, the size of the device, and its data bus width. Also defined are the location in memory and some way to describe the chip's storage organization.

When updating one portion of the Flash, you often want to preserve the contents of other portions of the same Flash. In this case, your hardware probe must have some way to limit the sectors that are erased. In the case of the Abatron unit, this is done by adding a line starting with the keyword `ERASE` for each sector to be erased. When the `erase` command is issued to the Abatron unit via its `telnet` user interface, all sectors defined with an `ERASE` specification are erased. Listing

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4

Hardware-Assisted Debugging

14-20 demonstrates erasing a portion of Flash on a target board and subsequently programming a new U-Boot bootloader image.

LISTING 14-20 Erase and Program Flash

```
$ telnet bdi
Trying 192.168.1.129...
Connected to bdi (192.168.1.129).
Escape character is '^]'.
BDI Debugger for Embedded PowerPC
=================================
...  (large volume of help text)

uei> erase
Erasing flash at 0xfff00000
Erasing flash at 0xfff10000
Erasing flash at 0xfff20000
Erasing flash at 0xfff30000
Erasing flash at 0xfff40000
Erasing flash passed
uei> prog 0xfff00000 u-boot.bin BIN
Programming u-boot.bin , please wait ....
Programming flash passed
uei>
```

First we establish a `telnet` session to the Abatron BDI-2000. After some initialization, we are presented with a command prompt. When the `erase` command is issued, the Abatron displays a line of output for each section defined in the configuration file. With the configuration shown in Appendix F, we defined five erase sectors. This reserves up to 256KB of space for the U-Boot bootloader binary.

The `prog` command is shown with all three of its optional parameters. These specify the location in memory where the new image is to be loaded, the name of the image file, and the format of the file—in this case, a binary file. You can specify these parameters in the BDI-2000 configuration file. In this case, the command reduces to simply `prog` without parameters.

This example only scratches the surface of these two BDI-2000 commands. Many more combinations of usage and capabilities are supported. Each hardware JTAG probe has its own way to specify Flash erasure and programming capabilities. Consult the documentation for your particular device for the specifics.

14.4.2 Debugging with a JTAG Probe

Instead of interfacing directly with a JTAG probe via its user interface, many JTAG probes can interface with your source-level debugger. By far the most popular debugger supported by hardware probes is the `gdb` debugger. In this usage scenario, `gdb` is instructed to begin a debug session with the target via an external connection, usually an Ethernet connection. Rather than communicate directly with the JTAG probe via a user interface, the debugger passes commands back and forth between itself and the JTAG probe. In this model, the JTAG probe uses the `gdb` remote protocol to control the hardware on behalf of the debugger. Refer again to Figure 14-6 for connection details.

SECTION 14.4

Hardware-Assisted Debugging

JTAG probes are especially useful for source-level debugging of bootloader and early startup code. In this example, we demonstrate the use of `gdb` and an Abatron BDI-2000 for debugging portions of the U-Boot bootloader on a PowerPC target board.

Many processors contain debugging registers that include the capability to set traditional address breakpoints (stop when the program reaches a specific address) as well as data breakpoints (stop on conditional access of a specified memory address). When debugging code resident in read-only memory such as Flash, this is the only way to set a breakpoint. However, these registers are typically limited. Many processors contain only one or two such registers. This limitation must be understood before using hardware breakpoints. The following example demonstrates this.

Using a setup such as that shown in Figure 14-6, assume that our target board has U-Boot stored in Flash. When we presented bootloaders in Chapter 7, you learned that U-Boot and other bootloaders typically copy themselves into RAM as soon as possible after startup. This is because hardware read (and write) cycles from RAM are orders of magnitude faster than typical read-only memory devices such as Flash. This presents two specific debugging challenges. First, we cannot modify the contents of read-only memory (to insert a software breakpoint), so we must rely on processor-supported breakpoint registers for this purpose.

The second challenge comes from the fact that only one of the execution contexts (Flash or RAM) can be represented by the ELF executable file from which `gdb` reads its symbolic debugging information. In the case of U-Boot, it is linked for the Flash environment where it is initially stored. The early code relocates itself and performs any necessary address adjustments. This means that we need to work with `gdb` within both of these execution contexts. Listing 14-21 shows an example of such a debug session.

SECTION 14.4

Hardware-Assisted Debugging

LISTING 14-21 U-Boot Debugging Using JTAG Probe

```
$ ppc-linux-gdb --silent u-boot
(gdb) target remote bdi:2001
Remote debugging using bdi:2001
_start () at /home/chris/sandbox/u-boot-1.1.4/cpu/mpc5xxx/start.S:91
91        li      r21, BOOTFLAG_COLD   /* Normal Power-On */
Current language:  auto; currently asm

<< Debug a flash resident code snippet >>
(gdb) mon break hard
(gdb) b board_init_f
Breakpoint 1 at 0xfff0457c: file board.c, line 366.
(gdb) c
Continuing.

Breakpoint 1, board_init_f (bootflag=0x7fc3afc) at board.c:366
366             gd = (gd_t *) (CFG_INIT_RAM_ADDR + CFG_GBL_DATA_OFFSET);
Current language:  auto; currently c
(gdb) bt
#0  board_init_f (bootflag=0x1) at board.c:366
#1  0xfff0456c in board_init_f (bootflag=0x1) at board.c:353
(gdb) i frame
Stack level 0, frame at 0xf000bf50:
 pc = 0xfff0457c in board_init_f (board.c:366); saved pc 0xfff0456c
 called by frame at 0xf000bf78
 source language c.
 Arglist at 0xf000bf50, args: bootflag=0x1
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4

Hardware-Assisted Debugging

```
 Locals at 0xf000bf50, Previous frame's sp is 0x0

<< Now debug a memory resident code snippet after relocation >>
(gdb) del 1
(gdb) symbol-file
Discard symbol table from '/home/chris/sandbox/u-boot-1.1.4-powerdna/u-boot'? (y or n) y
No symbol file now.
(gdb) add-symbol-file u-boot 0x7fa8000
add symbol table from file "u-boot" at
        .text_addr = 0x7fa8000
(y or n) y
Reading symbols from u-boot...done.
(gdb) b board_init_r
Breakpoint 2 at 0x7fac6c0: file board.c, line 608.
(gdb) c
Continuing.

Breakpoint 2, board_init_r (id=0x7f85f84, dest_addr=0x7f85f84) at board.c:608
608             gd = id;      /* initialize RAM version of global data */
(gdb) i frame
Stack level 0, frame at 0x7f85f38:
 pc = 0x7fac6c0 in board_init_r (board.c:608); saved pc 0x7fac6b0
 called by frame at 0x7f85f68
 source language c.
 Arglist at 0x7f85f38, args: id=0x7f85f84, dest_addr=0x7f85f84
 Locals at 0x7f85f38, Previous frame's sp is 0x0
(gdb) mon break soft
(gdb)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.4

Hardware-Assisted Debugging

Study this example carefully. Some subtleties are definitely worth taking the time to understand. First, we connect to the Abatron BDI-2000 using the target remote command. The IP address in this case is that of the Abatron unit, represented by the symbolic name `bdi`.[10] The Abatron BDI-2000 uses port 2001 for its remote `gdb` protocol connection.

Next we issue a command to the BDI-2000 using the `gdb mon` command. The `mon` command tells `gdb` to pass the rest of the command directly to the remote hardware device. Therefore, `mon break hard` sets the BDI-2000 into hardware breakpoint mode.

We then set a hardware breakpoint at `board_init_f`. This is a routine that executes while still running out of Flash memory at address `0xfff0457c`. After the breakpoint is defined, we issue the continue `c` command to resume execution. Immediately, the breakpoint at `board_init_f` is encountered, and we are free to do the usual debugging activities, including stepping through code and examining data. You can see that we have issued the `bt` command to examine the stack backtrace and the `i frame` command to examine the details of the current stack frame.

Now we continue execution again, but this time we know that U-Boot copies itself to RAM and resumes execution from its copy in RAM. So we need to change the debugging context while keeping the debugging session alive. To accomplish this, we discard the current symbol table (`symbol-file` command with no arguments) and load in the same symbol file again using the `add-symbol-file` command. This time, we instruct `gdb` to offset the symbol table to match where U-Boot has relocated itself to memory. This ensures that our source code and symbolic debugging information match the actual memory resident image.

After the new symbol table is loaded, we can add a breakpoint to a location that we know will reside in RAM when it is executed. This is where one of the subtle complications is exposed.

[10] *An entry in the host system's* `/etc/hosts` *file enables the symbolic IP address reference.*

SECTION 14.4

Hardware-Assisted Debugging

Because we know that U-Boot is currently running in Flash but is about to move itself to RAM and jump to its RAM-based copy, we must still use a hardware breakpoint. Consider what happens at this point if we use a software breakpoint. `gdb` dutifully writes the breakpoint opcode into the specified memory location, but U-Boot overwrites it when it copies itself to RAM. The net result is that the breakpoint is never hit, and we begin to suspect that our tools are broken. After U-Boot has entered the RAM copy and our symbol table has been updated to reflect the RAM-based addresses, we are free to use RAM-based breakpoints. This is reflected by the last command in Listing 14-21 setting the Abatron unit back to soft breakpoint mode.

Why do we care about using hardware versus software breakpoints? If we had unlimited hardware breakpoint registers, we wouldn't. But this is never the case. Here is what it looks like when you run out of processor-supported hardware breakpoint registers during a debug session:

```
(gdb) b flash_init
Breakpoint 3 at 0x7fbebe0: file flash.c, line 70.
(gdb) c
Continuing.
warning: Cannot insert breakpoint 3:
Error accessing memory address 0x7fbebe0: Unknown error 4294967295.
```

Because we are debugging remotely, we aren't told about the resource constraint until we try to resume after entering additional breakpoints. This is because of the way `gdb` handles breakpoints. When a breakpoint is hit, `gdb` restores all the breakpoints with the original opcodes for that particular memory location. When it resumes execution, it restores the breakpoint opcodes at the specified locations. You can observe this behavior by enabling `gdb`'s remote debug mode:

```
(gdb) set debug remote 1
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.5
When It Doesn't Boot

14.5 When It Doesn't Boot

One of the most frequently asked questions on the various mailing lists that serve embedded Linux goes something like this:

> *I am trying to boot Linux on my board, and I get stuck after this message prints to my console:*
>
> *"Uncompressing Kernel Image . . . OK."*

Thus starts the long and sometimes frustrating learning curve of embedded Linux! Many things that can go wrong could lead to this common failure. With some knowledge and a JTAG debugger, there are ways to determine what went awry.

14.5.1 Early Serial Debug Output

The first tool you might have available is `CONFIG_SERIAL_TEXT_DEBUG`. This Linux kernel-configuration option adds support for debug messages very early in the boot process. At the present time, this feature is limited to the PowerPC architecture, but nothing prevents you from duplicating the functionality in other architectures. Listing 14-22 provides an example of this feature in use on a PowerPC target using the U-Boot bootloader.

LISTING 14-22 Early Serial Text Debug

```
## Booting image at 00200000 ...
   Image Name:   Linux-2.6.14
   Created:      2005-12-19  22:24:03 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    607149 Bytes = 592.9 kB
```

SECTION 14.5

When It Doesn't Boot

```
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
id mach(): done      <== Start of messages enabled by
MMU:enter            <== CONFIG_SERIAL_TEXT_DEBUG
MMU:hw init
MMU:mapin
MMU:setio
MMU:exit
setup_arch: enter
setup_arch: bootmem
arch: exit
arch: real exit
```

Using this feature, you can often tell where your board is getting stuck during the boot process. Of course, you can add your own early debug messages in other places in the kernel. Here is an example of its usage found in `.../arch/ppc/mm/init.c`:

```
        /* Map in all of RAM starting at KERNELBASE */
        if (ppc_md.progress)
                ppc_md.progress("MMU:mapin", 0x301);
        mapin_ram();
```

[11] *All these filenames are unique, so they can be found without full path-name references.*

The AMCC Yosemite platform is an excellent example of this infrastructure. Consult the following files in the Linux source tree[11] for details of how this debugging system is implemented:

SECTION 14.5

When It Doesn't Boot

File	Function	Purpose
gen550_dbg.c	gen550_init	Serial port setup, called by yosemite.c platform-initialization file
gen550_dbg.c	gen550_progress	Low-level serial output routine
ibm44x_common.c	ibm44x_platform_init	Binds platform-specific progress routine to generic ppc machine-dependent infrastructure

14.5.2 Dumping the printk Log Buffer

When we discussed printk debugging in Section 14.3.6, we pointed out some of the limitations of this method. printk itself is a very robust implementation. One of its shortcomings is that you can't see any printk messages until later in the boot sequence when the console device has been initialized. Very often, when your board hangs on boot, quite a few messages are stuck in the printk buffer. If you know where to find them, you can often pinpoint the exact problem that is causing the boot to hang. Indeed, many times you will discover that the kernel has encountered an error that led to a call to panic(). The output from panic() has likely been dumped into the printk buffer, and you can often pinpoint the exact line of offending code.

This is best accomplished with a JTAG debugger, but it is still possible to use a bootloader and its memory dump capability to display the contents of the printk buffer after a reset. Some corruption of memory contents might occur as a result of the reset, but log buffer text is usually very readable.

The actual buffer where printk stores its message text is declared in the printk source file .../kernel/printk.c.

```
static char __log_buf[__LOG_BUF_LEN];
```


Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.5
When It Doesn't Boot

We can easily determine the linked location of this buffer from the Linux kernel map file `System.map`.

```
$ grep __log_buf System.map
   c022e5a4 b __log_buf
```

Now if the system happens to hang upon booting, right after displaying the `"Uncompressing Kernel Image ... OK"` message, reboot and use the bootloader to examine the buffer. Because the relationship between kernel virtual memory and physical memory is fixed and constant on a given architecture, we can do a simple conversion. The address of `__log_buf` shown earlier is a kernel virtual address; we must convert it to a physical address. On this particular PowerPC architecture, that conversion is a simple subtraction of the constant `KERNELBASE` address, `0xc0000000`. This is where we probe in memory to read the contents, if any, of the `printk` log buffer.

Listing 14-23 is an example of the listing as displayed by the U-Boot memory dump command.

LISTING 14-23 Dump of Raw `printk` Log Buffer

```
=> md 22e5a4
0022e5a4: 3c353e4c 696e7578 20766572 73696f6e    <5>Linux version
0022e5b4: 20322e36 2e313320 28636872 6973406a     2.6.13 (chris@
0022e5c4: 756e696f 72292028 67636320 76657273    junior) (gcc
0022e5d4: 696f6e20 332e342e 3320284d 6f6e7461    version 3.4.3 (Monta
0022e5e4: 56697374 6120332e 342e332d 32352e30    Vista 3.4.3-25.0
0022e5f4: 2e37302e 30353031 39363120 32303035    .70.0501961 2005
0022e604: 2d31322d 31382929 20233131 20547565    -12-18)) #11 Tue
0022e614: 20466562 20313420 32313a30 353a3036     Feb 14 21:05:06
```

SECTION 14.5
When It Doesn't Boot

```
0022e624: 20455354 20323030 360a3c34 3e414d43     EST 2006.<4>AMC
0022e634: 4320506f 77657250 43203434 30455020    C PowerPC 440EP
0022e644: 596f7365 6d697465 20506c61 74666f72    Yosemite Platform.
0022e654: 6d0a3c37 3e4f6e20 6e6f6465 20302074    <7>On node 0
0022e664: 6f74616c 70616765 733a2036 35353336    totalpages: 65536
0022e674: 0a3c373e 2020444d 41207a6f 6e653a20    .<7>  DMA zone:
0022e684: 36353533 36207061 6765732c 204c4946    65536 pages, LIF
0022e694: 4f206261 7463683a 33310a3c 373e2020    O batch:31.<7>
=>
0022e6a4: 4e6f726d 616c207a 6f6e653a 20302070    Normal zone: 0
0022e6b4: 61676573 2c204c49 464f2062 61746368    pages, LIFO batch
0022e6c4: 3a310a3c 373e2020 48696768 4d656d20    :1.<7>  HighMemzone:
0022e6d4: 7a6f6e65 3a203020 70616765 732c204c    0 pages,
0022e6e4: 49464f20 62617463 683a310a 3c343e42    LIFO batch:1.<4>
0022e6f4: 75696c74 2031207a 6f6e656c 69737473    Built 1 zonelists
0022e704: 0a3c353e 4b65726e 656c2063 6f6d6d61    .<5>Kernel command
0022e714: 6e64206c 696e653a 20636f6e 736f6c65    line: console
0022e724: 3d747479 53302c31 31353230 3020726f    =ttyS0,115200
0022e734: 6f743d2f 6465762f 6e667320 72772069    root=/dev/nfs rw
0022e744: 703d6468 63700a3c 343e5049 44206861    ip=dhcp.<4>PID
0022e754: 73682074 61626c65 20656e74 72696573    hash table entries
0022e764: 3a203230 34382028 6f726465 723a2031    : 2048 (order:
0022e774: 312c2033 32373638 20627974 6573290a    11, 32768 bytes).
0022e784: 00000000 00000000 00000000 00000000    ................
0022e794: 00000000 00000000 00000000 00000000    ................
=>
```

It's not very pretty to read, but the data is there. We can see in this particular example that the kernel crashed someplace after initializing the PID hash table entries. With some additional use of `printk` messages, we can begin to close in on the actual source of the crash.

As shown in this example, this is a technique that can be used with no additional tools. You can see the importance of some kind of early serial port output during boot if you are working on a new board port.

14.5.3 KGDB on Panic

If KGDB is enabled, the kernel attempts to pass control back to KGDB upon error exceptions. In some cases, the error itself will be readily apparent. To use this feature, a connection must already be established between KGDB and `gdb`. When the exception condition occurs, KGDB emits a Stop Reply packet to `gdb`, indicating the reason for the trap into the debug handler, as well as the address where the trap condition occurred. Listing 14-24 illustrates the sequence.

LISTING 14-24 Trapping Crash on Panic Using KGDB

```
$ ppc-_4xx-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
Malformed response to offset query, qOffsets
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }
(gdb) c
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 14.6

Chapter Summary

```
Continuing.

<< KGDB gains control from panic() on crash >>
Program received signal SIGSEGV, Segmentation fault.
0xc0215d6c in pcibios_init () at arch/ppc/kernel/pci.c:1263
1263            *(int *)-1 = 0;
(gdb) bt
#0  0xc0215d6c in pcibios_init () at arch/ppc/kernel/pci.c:1263
#1  0xc020e728 in do_initcalls () at init/main.c:563
#2  0xc020e7c4 in do_basic_setup () at init/main.c:605
#3  0xc0001374 in init (unused=0x20) at init/main.c:677
#4  0xc00049d0 in kernel_thread ()
Previous frame inner to this frame (corrupt stack?)
(gdb)
```

The crash in this example was contrived by a simple write to an invalid memory location (all ones). We first establish a connection from `gdb` to KGDB and allow the kernel to continue to boot. Notice that we didn't even bother to set breakpoints. When the crash occurs, we see the line of offending code and get a nice backtrace to help us determine its cause.

14.6 Chapter Summary

- Linux kernel debugging presents many complexities, especially in a cross-development environment. Understanding how to navigate these complexities is the key to successful kernel debugging.

SECTION 14.6
Chapter Summary

- KGDB is a very useful kernel-level `gdb stub` that enables direct symbolic source-level debugging inside the Linux kernel and device drivers. It uses the `gdb` remote protocol to communicate to your host-based cross-gdb.
- Understanding (and minimizing) compiler optimizations helps make sense of seemingly strange debugger behavior when stepping through compiler-optimized code.
- `gdb` supports user-defined commands, which can be very useful for automating tedious debugging tasks such as iterating kernel linked lists and accessing complex variables.
- Kernel-loadable modules present their own challenges to source-level debugging. The module's initialization routine can be debugged by placing a breakpoint in `module.c` at the call to `module->init()`.
- `printk` and the Magic SysReq key provide additional tools to help isolate problems during kernel development and debugging.
- Hardware-assisted debugging via a JTAG probe enables debugging Flash or ROM resident code where other debugging methods can be cumbersome or otherwise impossible.
- Enabling `CONFIG_SERIAL_TEXT_DEBUG` on architectures where this feature is supported is a powerful tool for debugging a new kernel port.
- Examining the `printk log_buf` often leads to the cause of a silent kernel crash on boot.
- `KGDB` passes control to `gdb` on a kernel panic, enabling you to examine a backtrace and isolate the cause of the kernel panic.

SECTION 14.6

Chapter Summary

14.6.1 Suggestions for Additional Reading

Linux Kernel Development, 2nd Edition
Robert Love
Novell Press, 2005

The Linux Kernel Primer
Claudia Salzberg Rodriguez et al.
Prentice Hall, 2005

"Using the GNU Compiler Collection"
Richard M. Stallman and the GCC Developer Community
GNU Press, a division of Free Software Foundation
http://gcc.gnu.org/onlinedocs/

KGDB Sourceforge home page
http://sourceforge.net/projects/KGDB

Debugging with GDB
Richard Stallman, Roland Pesch, Stan Shebs, et al.
Free Software Foundation
www.gnu.org/software/gdb/documentation/

Tool Interface Standards
DWARF Debugging Information Format Specification
Version 2.0
TIS Committee, May 1995

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.1

Target Debugging

CHAPTER 15

Debugging Embedded Linux Applications

In the previous chapter, we explored the use of GDB for debugging kernel code and code resident in Flash, such as bootloader code. In this chapter, we continue our coverage of GDB for debugging application code in user space. We extend our coverage of remote debugging and the tools and techniques used for this peculiar debugging environment.

15.1 Target Debugging

We already explored several important debugging tools in Chapter 13, "Development Tools." `strace` and `ltrace` can be used to observe and characterize a process's behavior and often isolate problems. `dmalloc` can help isolate memory leaks and profile memory usage. `ps` and `top` are both useful for examining the state of processes. These relatively small tools are designed to run directly on the target hardware.

Debugging Linux application code on an embedded system has its own unique challenges. Resources on your embedded target are often limited. RAM and nonvolatile storage limitations might prevent you from running target-based development tools. You might not have an Ethernet port or other high-speed connection. Your target embedded system might not have a graphical display, keyboard, or mouse.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.2

Remote (Cross) Debugging

This is where your cross-development tools and an NFS root mount environment can yield large dividends. Many tools, especially GDB, have been architected to execute on your development host while actually debugging code on a remote target. GDB can be used to interactively debug your target code or to perform a postmortem analysis of a core file generated by an application crash. We covered the details of application core dump analysis in Chapter 13.

15.2 Remote (Cross) Debugging

Cross-development tools were developed primarily to overcome the resource limitations of embedded platforms. A modest-size application compiled with symbolic debug information can easily exceed several megabytes. With cross-debugging, the heavy lifting can be done on your development host. When you invoke your cross-version of GDB on your development host, you pass it an ELF file compiled with symbolic debug information. On your target, there is no reason you can't strip[1] the ELF file of all unnecessary debugging info to keep the resulting image to its minimum size.

We introduced the `readelf` utility in Chapter 13. In Chapter 14, "Kernel Debugging Techniques," we used it to examine the debug information in an ELF file compiled with symbolic debugging information. Listing 15-1 contains the output of `readelf` for a relatively small web server application compiled for the ARM architecture.

[1] *Remember to use your cross-version of strip, for example* `ppc_82xx-strip`.

LISTING 15-1 ELF File Debug Info for Example Program

```
$ xscale_be-readelf -S websdemo
There are 39 section headers, starting at offset 0x3dfd0:
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.2

Remote (Cross) Debugging

```
Section Headers:
[Nr] Name              Type        Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL        00000000 000000 000000 00      0   0  0
[ 1] .interp           PROGBITS    00008154 000154 000013 00   A  0   0  1
[ 2] .note.ABI-tag     NOTE        00008168 000168 000020 00   A  0   0  4
[ 3] .note.numapolicy  NOTE        00008188 000188 000074 00   A  0   0  4
[ 4] .hash             HASH        000081fc 0001fc 00022c 04   A  5   0  4
[ 5] .dynsym           DYNSYM      00008428 000428 000460 10   A  6   1  4
[ 6] .dynstr           STRTAB      00008888 000888 000211 00   A  0   0  1
[ 7] .gnu.version      VERSYM      00008a9a 000a9a 00008c 02   A  5   0  2
[ 8] .gnu.version_r    VERNEED     00008b28 000b28 000020 00   A  6   1  4
[ 9] .rel.plt          REL         00008b48 000b48 000218 08   A  5  11  4
[10] .init             PROGBITS    00008d60 000d60 000018 00  AX  0   0  4
[11] .plt              PROGBITS    00008d78 000d78 000338 04  AX  0   0  4
[12] .text             PROGBITS    000090b0 0010b0 019fe4 00  AX  0   0  4
[13] .fini             PROGBITS    00023094 01b094 000018 00  AX  0   0  4
[14] .rodata           PROGBITS    000230b0 01b0b0 0023d0 00   A  0   0  8
[15] .ARM.extab        PROGBITS    00025480 01d480 000000 00   A  0   0  1
[16] .ARM.exidx        ARM_EXIDX   00025480 01d480 000008 00  AL 12   0  4
[17] .eh_frame_hdr     PROGBITS    00025488 01d488 00002c 00   A  0   0  4
[18] .eh_frame         PROGBITS    000254b4 01d4b4 00007c 00   A  0   0  4
[19] .init_array       INIT_ARRAY  0002d530 01d530 000004 00  WA  0   0  4
[20] .fini_array       FINI_ARRAY  0002d534 01d534 000004 00  WA  0   0  4
[21] .jcr              PROGBITS    0002d538 01d538 000004 00  WA  0   0  4
[22] .dynamic          DYNAMIC     0002d53c 01d53c 0000d0 08  WA  6   0  4
[23] .got              PROGBITS    0002d60c 01d60c 000118 04  WA  0   0  4
[24] .data             PROGBITS    0002d728 01d728 0003c0 00  WA  0   0  8
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.2

Remote (Cross) Debugging

```
[25] .bss             NOBITS      0002dae8 01dae8 0001c8 00  WA  0  0  4
[26] .comment         PROGBITS    00000000 01dae8 000940 00      0  0  1
[27] .debug_aranges   PROGBITS    00000000 01e428 0004a0 00      0  0  8
[28] .debug_pubnames  PROGBITS    00000000 01e8c8 001aae 00      0  0  1
[29] .debug_info      PROGBITS    00000000 020376 013d27 00      0  0  1
[30] .debug_abbrev    PROGBITS    00000000 03409d 002ede 00      0  0  1
[31] .debug_line      PROGBITS    00000000 036f7b 0034a2 00      0  0  1
[32] .debug_frame     PROGBITS    00000000 03a420 003380 00      0  0  4
[33] .debug_str       PROGBITS    00000000 03d7a0 000679 00      0  0  1
[34] .note.gnu.arm.ide NOTE       00000000 03de19 00001c 00      0  0  1
[35] .debug_ranges    PROGBITS    00000000 03de35 000018 00      0  0  1
[36] .shstrtab        STRTAB      00000000 03de4d 000183 00      0  0  1
[37] .symtab          SYMTAB      00000000 03e5e8 004bd0 10     38 773 4
[38] .strtab          STRTAB      00000000 0431b8 0021bf 00      0  0  1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
$
```

You can see from Listing 15-1 that there are many sections containing debug information. There is also a `.comment` section that contains more than 2KB (0x940) of information that is not necessary for the application to function. The size of this example file, including debug information, is more than 275KB.

```
$ ls -l websdemo
-rwxrwxr-x  1 chris chris 283511 Nov 8 18:48 websdemo
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.2

Remote (Cross) Debugging

If we strip this file using the `strip` utility, we can minimize its size to preserve resources on our target system. Listing 15-2 shows the results.

LISTING 15-2 Strip Target Application

```
$ xscale_be-strip -s -R .comment -o websdemo-stripped websdemo
$ ls -l websdemo*
-rwxrwxr-x  1 chris chris 283491 Apr  9 09:19 websdemo
-rwxrwxr-x  1 chris chris 123156 Apr  9 09:21 websdemo-stripped
$
```

Here we strip both the symbolic debug information and the `.comment` section from the executable file. We specify the name of the stripped binary using the `-o` command-line switch. You can see that the resulting size of the stripped binary is less than half of its original size. Of course, for larger applications, this space savings can be even more significant. A recent Linux kernel compiled with debug information was larger than 18MB. After stripping as in Listing 15-2, the resulting binary was slightly larger than 2MB!

For debugging in this fashion, you place the stripped version of the binary on your target system and keep a local unstripped copy on your development workstation containing symbolic information needed for debugging. You use `gdbserver` on your target board to provide an interface back to your development host where you run the full-blown version of GDB on your nonstripped binary.

15.2.1 gdbserver

Using `gdbserver` allows you to run GDB from a development workstation rather than on the target embedded Linux platform. This configuration has obvious benefits. For starters, it is common that

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

your development workstation has far more CPU power, memory, and hard-drive storage than the embedded platform. In addition, it is common for the source code for your application under debug to exist on the development workstation and not on the embedded platform.

`gdbserver` is a small program that runs on the target board and allows remote debugging of a process on the board. It is invoked on the target board specifying the program to be debugged, as well as an IP address and port number on which it will listen for connection requests from GDB. Listing 15-3 shows the startup sequence on the target board.

LISTING 15-3 Starting gdbserver on Target Board

```
$ gdbserver localhost:2001 websdemo-stripped
Process websdemo-stripped created; pid = 197
Listening on port 2001
```

This particular example starts `gdbserver` configured to listen for an Ethernet TCP/IP connection on port 2001, ready to debug our stripped binary program called `websdemo-stripped`.

From our development workstation, we launch GDB, passing it the name of the binary executable containing symbolic debug information that we want to debug as an argument. After GDB starts up, we issue a command to connect to the remote target board. Listing 15-4 shows this sequence.

LISTING 15-4 Starting Remote GDB Session

```
$ xscale_be-gdb -q websdemo
(gdb) target remote 192.168.1.141:2001
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
```


Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.2

Remote (Cross) Debugging

```
(gdb) p main       <<<< display address of main function
$1 = {int (int, char **)} 0x12b68 <main>
(gdb) b main       <<<< Place breakpoint at main()
Breakpoint 1 at 0x12b80: file main.c, line 72.
(gdb)
```

The sequence in Listing 15-4 invokes `cross-gdb` on your development host. When GDB is running, we issue the `gdb target remote` command. This command causes GDB to initiate a TCP/IP connection from your development workstation to your target board, with the indicated IP address on port 2001. When `gdbserver` accepts the connection request, it prints a line similar to this:

```
Remote debugging from host 192.168.0.10
```

Now GDB is connected to the target board's `gdbserver` process, ready to accept commands from GDB. The rest of the session is exactly the same as if you were debugging an application locally. This is a powerful tool, allowing you to use the power of your development workstation for the debug session, leaving only a small, relatively unobtrusive GDB stub and your program being debugged on the target board. In case you were wondering, `gdbserver` for this particular ARM target is only 54KB.

```
root@coyote:~# ls -l /usr/bin/gdbserver
-rwxr-xr-x  1 root root 54344 Jul 23  2005 /usr/bin/gdbserver
```

There is one caveat, and it is the subject of a frequently asked question (FAQ) on many mailing lists. You must be using a GDB on your development host that was configured as a *cross-debugger*. It is a binary program that runs on your development workstation but understands binary executable images compiled for another architecture. This is an important and frequently

SECTION 15.2

Remote (Cross) Debugging

overlooked fact. You cannot debug a PowerPC target with a native GDB such as that found in a typical Red Hat Linux installation. You must have a GDB configured for your host and target combination.

When GDB is invoked, it displays a banner consisting of several lines of information and then displays its compiled configuration. Listing 15-5 is an example of the GDB used for some examples in this book, which is part of an embedded Linux distribution provided by MontaVista Software configured for PowerPC cross-development.

LISTING 15-5 Invocation of cross-gdb

```
$ ppc_82xx-gdb
GNU gdb 6.0 (MontaVista 6.0-8.0.4.0300532 2003-12-24)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.  Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "--host=i686-pc-linux-gnu
--target=powerpc-hardhat-linux".
(gdb)
```

Notice the last lines of this GDB startup message. This is the configuration compiled into this version of GDB. It was compiled to execute on a Pentium (i686) PC host running GNU/Linux and to debug binary programs compiled for a PowerPC processor running GNU/Linux. This is specified by the `--host` and `--target` variables displayed by the banner text, and is also a part of the configuration string passed to `./configure` when building GDB.

15.3 Debugging with Shared Libraries

Now that you understand how to invoke a remote debug session using GDB on the host and `gdbserver` on the target, we turn our attention to the complexities of shared libraries and debug symbols. Unless your application is a statically linked executable (linked with the `-static` linker command-line switch), many symbols in your application will reference code outside your application. Obvious examples include the use of standard C library routines such as `fopen`, `printf`, `malloc`, and `memcpy`. Less obvious examples might include calls to application-specific functions, such as `jack_transport_locate()` (a routine from the JACK low-latency audio server), which calls a library function outside the standard C libraries.

To have symbols from these routines available, you must satisfy two requirements for GDB:

- You must have debug versions of the libraries available.
- GDB must know where to find them.

If you don't have debug versions of the libraries available, you can still debug your application; you just won't have any debug information available for library routines called by your application. Often this is perfectly acceptable, unless, of course, you are developing a shared library object as part of your embedded project.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.3

Debugging with Shared Libraries

Look back at Listing 15-4, where we invoked GDB on a remote target. After GDB connected via the `target remote` command, GDB issued a two-line response:

```
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
```

This confirms that GDB connected to our target at the indicated IP address and port. GDB then reports the location of the program counter as 0x40000790. Why do we get question marks instead of a symbolic location? Because this is the Linux dynamic loader (`ld-x.y.z.so`), and on this particular platform, we do not have debug symbols available for this shared library. How do we know this?

Recall our introduction of the `/proc` file system from Chapter 9, “File Systems.” One of the more useful entries was the `maps` entry (see Listing 9-16, in Chapter 9) in the per-process directory structure. We know the process ID (PID) of our target application from the `gdbserver` output in Listing 15-3. Our process was assigned PID 197. Given that, we can see the memory segments in use right after process startup, as shown in Listing 15-6.

LISTING 15-6 Initial Target Memory Segment Mapping

```
root@coyote:~# cat /proc/197/maps
00008000-00026000 r-xp 00000000 00:0e 4852444    ./websdemo-stripped
0002d000-0002e000 rw-p 0001d000 00:0e 4852444    ./websdemo-stripped
40000000-40017000 r-xp 00000000 00:0a 4982583    /lib/ld-2.3.3.so
4001e000-40020000 rw-p 00016000 00:0a 4982583    /lib/ld-2.3.3.so
bedf9000-bee0e000 rwxp bedf9000 00:00 0          [stack]
root@coyote:~#
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.3

Debugging with Shared Libraries

Here we see the target `websdemo-stripped` application occupying two memory segments. The first is the read-only executable segment at 0x8000, and the second is a read-write data segment at 0x2d000. The third memory segment is the one of interest. It is the Linux dynamic linker's executable code segment. Notice that it starts at address 0x40000000. If we investigate further, we can confirm that GDB is actually sitting at the first line of code for the dynamic linker, before any code from our own application has been executed. Using our cross version of `readelf`, we can confirm the starting address of the linker as follows:

```
# xscale_be-readelf -S ld-2.3.3.so ¦ grep \.text
[ 9] .text    PROGBITS    00000790 000790 012c6c 00  AX  0   0 16
```

From this data, we conclude that the address GDB reports on startup is the first instruction from `ld-2.3.3.so`, the Linux dynamic linker/loader. You can use this technique to get rough ideas of where your code is if you don't have symbolic debug information for a process or shared library.

Remember that we are executing this cross `readelf` command on our development host. Therefore, the `ld-2.3.3.so` file, itself an XScale binary object, must be accessible to your development host. Most typically, this file resides on your development host, and is a component of your embedded Linux distribution installed on your host.

15.3.1 Shared Library Events in GDB

GDB can alert you to shared library events. This can be useful for understanding your application's behavior or the behavior of the Linux loader, or for setting breakpoints in shared library routines you want to debug or step through. Listing 15-7 illustrates this technique. Normally, the complete path to the library is displayed. This listing has been edited for better readability.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.3
Debugging with Shared Libraries

LISTING 15-7 Stopping GDB on Shared Library Events

```
$ xscale_be-gdb -q websdemo
(gdb) target remote 192.168.1.141:2001
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) i shared       <<< Display loaded shared libs
No shared libraries loaded at this time.
(gdb) b main         <<< Break at main
Breakpoint 1 at 0x12b80: file main.c, line 72.
(gdb) c
Continuing.

Breakpoint 1, main (argc=0x1, argv=0xbec7fdc4) at main.c:72
72              int localvar = 9;
(gdb) i shared
From        To          Syms Read    Shared Object Library
0x40033300  0x4010260c  Yes         /opt/mvl/.../lib/tls/libc.so.6
0x40000790  0x400133fc  Yes         /opt/mvl/.../lib/ld-linux.so.3
(gdb) set stop-on-solib-events 1
(gdb) c
Continuing.
Stopped due to shared library event
(gdb) i shared
From        To          Syms Read    Shared Object Library
0x40033300  0x4010260c  Yes         /opt/mvl/.../lib/tls/libc.so.6
0x40000790  0x400133fc  Yes         /opt/mvl/.../lib/ld-linux.so.3
0x4012bad8  0x40132104  Yes         /opt/mvl/.../libnss_files.so.2
(gdb)
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.3

Debugging with Shared Libraries

When the debug session is first started, of course, no shared libraries are loaded. You can see this with the first `i shared` command. This command displays the shared libraries that are currently loaded. Setting a breakpoint at our application's `main()` function, we see that two shared libraries are now loaded. These are the Linux dynamic linker/loader and the standard C library component `libc`.

From here, we issue the `set stop-on-solib-event` command and continue program execution. When the application tries to execute a function from another shared library, that library is loaded. In case you are wondering, the `gethostbyname()` function is encountered and causes the next shared object load.

This example illustrates an important cross-development concept. The binary application (ELF image) running on the target contains information on the libraries it needs to resolve its external references. We can view this information easily using the `ldd` command introduced in Chapter 11, "BusyBox," and detailed in Chapter 13. Listing 15-8 shows the output of `ldd` invoked from the target board.

LISTING 15-8 ldd Executed on Target Board

```
root@coyote:/workspace# ldd websdemo
        libc.so.6 => /lib/tls/libc.so.6 (0x40020000)
        /lib/ld-linux.so.3 (0x40000000)
root@coyote:/workspace#
```

Notice that the paths to the shared libraries on the target are absolute paths starting at `/lib` on the root file system. But GDB running on your host development workstation cannot use these paths to find the libraries. You should realize that to do so would result in your host GDB loading

SECTION 15.3

Debugging with Shared Libraries

libraries from the wrong architecture. Your host is likely x86, whereas, in this example, the target is ARM XScale.

If you invoke your cross version of `ldd`, you will see the paths that were preconfigured into your toolchain. Your toolchain must have knowledge of where these files exist on your host development system.[2] Listing 15-9 illustrates this. Again, we have edited the listing for readability; long paths have been abbreviated.

LISTING 15-9 `ldd` **Executed on Development Host**

```
$ xscale_be-ldd websdemo
   libc.so.6 => /opt/mvl/.../xscale_be/target/lib/libc.so.6 (0xdead1000)
   ld-linux.so.3 => /opt/mvl/.../xscale_be/target/lib/ld-linux.so.3 (0xdead2000)
$
```

Your cross toolchain should be preconfigured with these library locations. Not only does your host GDB need to know where they are located, but, of course, your compiler and linker also need this knowledge.[3] GDB can tell you where it is configured to look for these libraries using the show `solib-absolute-prefix` command:

```
(gdb) show solib-absolute-prefix
Prefix for loading absolute shared library symbol files is
"/opt/mvl/pro/devkit/arm/xscale_be/target".
(gdb)
```

You can set or change where GDB searches for shared libraries using the GDB commands `set solib-absolute-prefix` and `set solib-search-path`. If you are developing your own shared library

[2] *It is certainly possible to pass these locations to your compiler, linker, and debugger for every invocation, but any good embedded Linux distribution will configure these defaults into the toolchain as a convenience to the developer.*

[3] *Of course, your compiler also needs to know the location of target files such as architecture-specific system and library header files.*

SECTION 15.3

Debugging with Shared Libraries

modules or have custom library locations on your system, you can use `solib-search-path` to instruct GDB where to look for your libraries. For more details about these and other GDB commands, consult the online GDB manual referenced at the end of this chapter in Section 15.6.1, "Suggestions for Additional Reading."

One final note about `ldd`. You might have noticed the addresses from Listing 15-8 and 15-9 associated with the libraries. `ldd` displays the load address for the start of these code segments as they would be if the program were loaded by the Linux dynamic linker/loader. Executed on the target, the addresses in Listing 15-5 make perfect sense, and we can correlate these with the `/proc/<pid>/maps` listing of the running process on the target. Listing 15-10 displays the memory segments for this target process after it is completely loaded and running.

LISTING 15-10 Memory Segments from `/proc/<pid>/maps` on Target

```
root@coyote:~# cat /proc/197/maps
00008000-00026000 r-xp 00000000 00:0e 4852444    /workspace/websdemo-stripped
0002d000-0002e000 rw-p 0001d000 00:0e 4852444    /workspace/websdemo-stripped
0002e000-0005e000 rwxp 0002e000 00:00 0         [heap]
40000000-40017000 r-xp 00000000 00:0a 4982583    /lib/ld-2.3.3.so
40017000-40019000 rw-p 40017000 00:00 0
4001e000-4001f000 r--p 00016000 00:0a 4982583    /lib/ld-2.3.3.so
4001f000-40020000 rw-p 00017000 00:0a 4982583    /lib/ld-2.3.3.so
40020000-4011d000 r-xp 00000000 00:0a 4982651    /lib/tls/libc-2.3.3.so
4011d000-40120000 ---p 000fd000 00:0a 4982651    /lib/tls/libc-2.3.3.so
40120000-40124000 rw-p 000f8000 00:0a 4982651    /lib/tls/libc-2.3.3.so
40124000-40126000 r--p 000fc000 00:0a 4982651    /lib/tls/libc-2.3.3.so
40126000-40128000 rw-p 000fe000 00:0a 4982651    /lib/tls/libc-2.3.3.so
```

```
40128000-4012a000 rw-p 40128000 00:00 0
4012a000-40133000 r-xp 00000000 00:0a 4982652    /lib/tls/libnss_files-2.3.3.so
40133000-4013a000 ---p 00009000 00:0a 4982652    /lib/tls/libnss_files-2.3.3.so
4013a000-4013b000 r--p 00008000 00:0a 4982652    /lib/tls/libnss_files-2.3.3.so
4013b000-4013c000 rw-p 00009000 00:0a 4982652    /lib/tls/libnss_files-2.3.3.so
becaa000-becbf000 rwxp becaa000 00:00 0         [stack]
root@coyote:~#
```

Notice the correlation of the target `ldd` output from Listing 15-8 to the memory segments displayed in the `/proc` file system for this process. The start (beginning of `.text` segment) of the Linux loader is `0x40000000` and the start of `libc` is at `0x40020000`. These are the virtual addresses where these portions of the application have been loaded, and are reported by the target invocation of `ldd`. However, the load addresses reported by the cross version of `ldd` in Listing 15-9 (`0xdead1000` and `0xdead2000`) are there to remind you that these libraries cannot be loaded on your host system (they are ARM architecture binaries), and the load addresses are simply placeholders.

15.4 Debugging Multiple Tasks

Generally the developer is presented with two different debugging scenarios when dealing with multiple threads of execution. Processes can exist in their own address space or can share an address space (and other system resources) with other threads of execution. The former (independent processes not sharing common address space) must be debugged using separate independent debug sessions. Nothing prevents you from using `gdbserver` on multiple processes on your target system, and using a separate invocation of GDB on your development host to coordinate a debug session for multiple cooperating but independent processes.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.4

Debugging Multiple Tasks

15.4.1 Debugging Multiple Processes

When a process being debugged under GDB uses the `fork()` system call[4] to spawn a new process, GDB can take two courses of action. It can continue to control and debug the parent process, or it can stop debugging the parent process and attach to the newly formed child process. You can control this behavior using the `set follow-fork-mode command`. The two modes are follow parent and follow child. The default behavior is for GDB to follow the parent. In this case, the child process executes immediately upon a successful fork.

Listing 15-11 reproduces a snippet of a simple program that forks multiple processes from its `main()` routine.

LISTING 15-11 Using **`fork()`** to Spawn a Child Process

```
...
  for( i=0; i<MAX_PROCESSES; i++ ) {
    /* Creating child process */
    pid[i] = fork();            /* Parent gets non-zero PID */
    if ( pid[i] == -1 ) {
      perror("fork failed");
      exit(1);
    }

    if ( pid[i] == 0 ) {      /* Indicates child's code path */
      worker_process();       /* The forked process calls this */
    }
  }
```

[4] *We will use the term system call, but* `fork()` *in this context is actually the C library function which in turn calls the Linux* `sys_fork()` *system call.*

SECTION 15.4

Debugging Multiple Tasks

```
  /* Parent's main control loop */
  while ( 1 ) {
...
  }
```

This simple loop creates `MAX_THREADS` new processes using the `fork()` system call. Each newly spawned process executes a body of code defined by the function `worker_process()`. When run under GDB in the default mode, GDB detects the creation of the new threads of execution (processes) but remains *attached* to the parent's thread of execution. Listing 15-12 illustrates this GDB session.

LISTING 15-12 GDB in `follow-fork-mode = parent`

```
(gdb) target remote 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) b main
Breakpoint 1 at 0x8888: file forker.c, line 104.
(gdb) c
Continuing.
[New Thread 356]
[Switching to Thread 356]

Breakpoint 1, main (argc=0x1, argv=0xbe807dd4) at forker.c:104
104       time(&start_time);
(gdb) b worker_process
Breakpoint 2 at 0x8784: file forker.c, line 45.
(gdb) c
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

```
Continuing.
Detaching after fork from child process 357.
Detaching after fork from child process 358.
Detaching after fork from child process 359.
Detaching after fork from child process 360.
Detaching after fork from child process 361.
Detaching after fork from child process 362.
Detaching after fork from child process 363.
Detaching after fork from child process 364.
```

Notice that eight child processes were spawned, with PID values from 357 to 364. The parent process was instantiated with PID 356. When the breakpoint in `main()` was hit, we entered a breakpoint at the `worker_process()` routine, which each child process executes upon `fork()`. Letting the program continue from main, we see each of the new processes spawned and detached by the debugger. They never hit the breakpoint because GDB is attached to the main process, which never executes the `worker_process()` routine.

If you need to debug each process, you must execute a separate independent GDB session and attach to the child process after it is `forked()`. The GDB documentation referenced at the end of this chapter outlines a useful technique to place a call to `sleep()` in the child process, giving you time to attach a debugger to the new process. Attaching to a new process is explained in Section 15.5.2, “Attaching to a Running Process.”

If you simply need to follow the child process, set the `follow-fork-mode` to follow child before your parent reaches the `fork()` system call. Listing 15-13 shows this.

SECTION 15.4

Debugging Multiple Tasks

LISTING 15-13 GDB in `follow-fork-mode = child`

```
(gdb) target remote 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) set follow-fork-mode child
(gdb) b worker_process
Breakpoint 1 at 0x8784: file forker.c, line 45.
(gdb) c
Continuing.
[New Thread 401]
Attaching after fork to child process 402.
[New Thread 402]
[Switching to Thread 402]

Breakpoint 1, worker_process () at forker.c:45
45        int my_pid = getpid();
(gdb) c
Continuing.
```

Here we see the parent process being instantiated as PID 401. When the first child is spawned by the `fork()` system call, GDB detaches silently from the parent thread of execution and attaches to the newly spawned child process having PID 402. GDB is now in control of the first child process and honors the breakpoint set at `worker_process()`. Notice, however, that the other child processes spawned by the code snippet from Listing 15-11 are not debugged and continue to run to their own completion.

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.4

Debugging Multiple Tasks

In summary, using GDB in this fashion, you are limited to debugging a single process at a time. You can debug through the `fork()` system call, but you have to decide which thread of execution to follow through the `fork()` call, either the parent or the child. As mentioned in the introduction to this section, you can use multiple independent GDB sessions if you must debug more than one cooperating process at a time.

15.4.2 Debugging Multithreaded Applications

If your application uses the POSIX thread library for its threading functions, GDB has additional capabilities to handle concurrent debugging of a multithreaded application. The Native Posix Thread Library (NPTL) has become the de facto standard thread library in use on Linux systems, including embedded Linux systems. The rest of this discussion assumes that you are using this thread library.

For this section, we use a demonstration program that spawns a number of threads using the `pthread_create()` library function in a simple loop. After the threads are spawned, the `main()` routine simply waits for keyboard input to terminate the application. Each thread displays a short message on the screen and sleeps for a predetermined time. Listing 15-14 shows the startup sequence on the target board.

LISTING 15-14 Target Threads Demo Startup

```
root@coyote:/workspace # gdbserver localhost:2001 ./tdemo
Process ./tdemo created; pid = 671
Listening on port 2001
Remote debugging from host 192.168.1.10
    ^^^^^  Previous three lines displayed by gdbserver
```

SECTION 15.4

Debugging Multiple Tasks

```
tdemo main() entered: My pid is 671
Starting worker thread 0
Starting worker thread 1
Starting worker thread 2
Starting worker thread 3
```

As in our previous examples, `gdbserver` prepares the application for running and waits for a connection from our host-based cross-gdb. When GDB connects, `gdbserver` reports the connection with the `Remote debugging...` message. Now we start GDB on the host and connect. Listing 15-15 reproduces this half of the session.

LISTING 15-15 Host GDB Connecting to Target Threads Demo

```
$ xscale_be-gdb -q tdemo
(gdb) target remote 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) b tdemo.c:97
Breakpoint 1 at 0x88ec: file tdemo.c, line 97.
(gdb) c
Continuing.
[New Thread 1059]
[New Thread 1060]
[New Thread 1061]
[New Thread 1062]
[New Thread 1063]
[Switching to Thread 1059]
```

SECTION 15.4

Debugging Multiple Tasks

```
Breakpoint 1, main (argc=0x1, argv=0xbefffdd4) at tdemo.c:98
98              int c = getchar();
(gdb)
```

Here we connect to the target (resulting in the "`Remote debugging...`" message in Listing 15-14), set a breakpoint just past the loop where we spawned the new threads, and continue. When the new thread is created, GDB displays a notice along with the thread ID. Thread 1059 is the `tdemo` application, doing its work directly from the `main()` function. Threads 1060 through 1063 are the new threads created from the call to `pthread_create()`.

When GDB hits the breakpoint, it displays the message `[Switching to Thread 1059]`, indicating that this was the thread of execution that encountered the breakpoint. It is the active thread for the debugging session, referred to as the *current thread* in the GDB documentation.

GDB enables us to switch between threads and perform the usual debugging operations such as setting additional breakpoints, examining data, displaying a backtrace, and working with the individual stack frames within the current thread. Listing 15-16 provides examples of these operations, continuing directly with our debugging session started in Listing 15-15.

LISTING 15-16 GDB Operations on Threads

```
...
(gdb) c
Continuing.
                 <<< Ctl-C to interrupt program execution
Program received signal SIGINT, Interrupt.
0x400db9c0 in read () from /opt/mvl/.../lib/tls/libc.so.6
```

SECTION 15.4

Debugging Multiple Tasks

```
(gdb) i threads
  5 Thread 1063  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
  4 Thread 1062  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
  3 Thread 1061  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
  2 Thread 1060  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
* 1 Thread 1059  0x400db9c0 in read ()
   from /opt/mvl/.../lib/tls/libc.so.6
(gdb) thread 4               <<< Make Thread 4 the current thread
[Switching to thread 4 (Thread 1062)]
#0  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
(gdb) bt
#0  0x400bc714 in nanosleep ()
   from /opt/mvl/.../lib/tls/libc.so.6
#1  0x400bc4a4 in __sleep (seconds=0x0) at sleep.c:137
#2  0x00008678 in go_to_sleep (duration=0x5) at tdemo.c:18
#3  0x00008710 in worker_2_job (random=0x5) at tdemo.c:36
#4  0x00008814 in worker_thread (threadargs=0x2) at tdemo.c:67
#5  0x40025244 in start_thread (arg=0xfffffdfc) at pthread_create.c:261
#6  0x400e8fa0 in clone () at../sysdeps/unix/sysv/linux/arm/clone.S:82
#7  0x400e8fa0 in clone () at../sysdeps/unix/sysv/linux/arm/clone.S:82
(gdb) frame 3
```

SECTION 15.4

Debugging Multiple Tasks

```
#3  0x00008710 in worker_2_job (random=0x5) at tdemo.c:36
36          go_to_sleep(random);
(gdb) l                   <<< Generate listing of where we are
31      }
32
33      static void worker_2_job(int random)
34      {
35          printf("t2 sleeping for %d\n", random);
36          go_to_sleep(random);
37      }
38
39      static void worker_3_job(int random)
40      {
(gdb)
```

A few points are worth mentioning. GDB assigns its own integer value to each thread and uses these values to reference the individual threads. When a breakpoint is hit in a thread, all threads within the process are halted for examination. GDB marks the current thread with an asterisk (*). You can set unique breakpoints within each thread—assuming, of course, that they exist in a unique context. If you set a breakpoint in a common portion of code where all threads execute, the thread that hits the breakpoint first is arbitrary.

The GDB user documentation referenced at the end of this chapter contains more useful information related to debugging in a multithreaded environment.

15.4.3 Debugging Bootloader/Flash Code

Debugging Flash resident code presents its own unique challenges. The most obvious limitation is the way in which GDB and `gdbserver` cooperate in setting target breakpoints. When we discussed the GDB remote serial protocol in Chapter 14, you learned how breakpoints are inserted into an application.[5] GDB replaces the opcode at the breakpoint location with an architecture-specific opcode that passes control to the debugger. However, in ROM or Flash, GDB cannot overwrite the opcode, so this method of setting breakpoints is useless.

Most modern processors contain some number of debug registers that can be used to get around this limitation. These capabilities must be supported by architecture- and processor-specific hardware probes or stubs. The most common technique for debugging Flash and ROM resident code is to use JTAG hardware probes. These probes support the setting of processor-specific hardware breakpoints. This topic was covered in detail in Chapter 14. Refer back to Section 14.4.2, “Debugging with a JTAG Probe,” for details.

15.5 Additional Remote Debug Options

Sometimes you might want to use a serial port for remote debugging. For other tasks, you might find it useful to attach the debugger to a process that is already running. These simple but useful operations are detailed here.

15.5.1 Debugging via Serial Port

Debugging via serial port is quite straightforward. Of course, you must have a serial port available on your target that is not being used by another process, such as a serial console. The same

[5] *Refer back to Listing 14-5 in Chapter 14.*

SECTION 15.5
Additional Remote Debug Options

limitation applies to your host. A serial port must be available. If both of these conditions can be met, simply replace the `IP:Port` specification passed to `gdbserver` with a serial port specification. Use the same technique when connecting to your target from your host-based GDB.

On your target:

```
root@coyote:/workspace # gdbserver /dev/ttyS0 ./tdemo
Process ./tdemo created; pid = 698
Remote debugging using /dev/ttyS0
```

From your host:

```
$ xscale_be-gdb -q tdemo
(gdb) target remote /dev/ttyS1
Remote debugging using /dev/ttyS1
0x40000790 in ?? ()
```

15.5.2 Attaching to a Running Process

It is often advantageous to connect to a process to examine its state while it is running instead of killing the process and starting it again. With `gdbserver`, it is trivial:

```
root@coyote:/workspace # ps ax ¦ grep tdemo
 1030 pts/0    Sl+    0:00 ./tdemo
root@coyote:/workspace # gdbserver localhost:2001 --attach 1030
Attached; pid = 1030
Listening on port 2001
```


Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

When you are finished examining the process under debug, you can issue the `gdb detach` command. This detaches the `gdbserver` from the application on the target and terminates the debug session. The application continues where it left off. This is a very useful technique for examining a running program. Be aware, though, that when you attach to the process, it halts, waiting for instructions from you. It will not resume execution until instructed to do so, using either the `continue` command or the `detach` command. Also note that you can use the `detach` command at almost any time to end the debug session and leave the application running on the target.

15.6 Chapter Summary

- Remote (cross) debugging enables symbolic debugging using host development workstation resources for the heavy lifting, preserving often scarce target resources.
- `gdbserver` runs on the target system and acts as the glue between the cross-gdb running on a development host and the process being debugged on the target.
- GDB on the host typically uses IP connections via Ethernet to send and receive commands to `gdbserver` running on the target. The GDB remote serial protocol is used between GDB and `gdbserver`.
- GDB can halt on shared library events and can automatically load shared library symbols when available. Your toolchain should be configured for the default paths on your cross-development system. Alternatively, you can use GDB commands to set the search paths for shared library objects.

Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30

SECTION 15.6

Chapter Summary

- GDB can be used to debug multiple independent processes via multiple concurrent GDB sessions.
- GDB can be configured to follow a forked process on a `fork()` system call. Its default mode is to continue to debug the parent—that is, the caller of `fork()`.
- GDB has features to facilitate debugging multithreaded applications written to POSIX thread APIs. The current default Linux thread library is the Native Posix Threads Library (NPTL).
- GDB supports attaching to and detaching from an already running process.

15.6.1 Suggestions for Additional Reading

GDB: The GNU Project Debugger
Online Documentation
http://sourceware.org/gdb/onlinedocs/

GDB Pocket Reference
Arnold Robbins
O'Reilly Media, 2005

Debugging Embedded Linux
Debugging Embedded Linux By Christopher Hallinan ISBN: 0131580132 Publisher: Prentice Hall
Print Publication Date: 2006/08/30