Use Case for KGDB Kernel Debugging Wind River Linux 3.0

Version 3.0 rev 1, 4/2/2008,
History
Version 2.0 rev 1, 5/27/2007, by David Reyna
Update by Jackie huang for 3.0

Table of Contents

1 Introduction

Wind River Workbench provides source-level debugging for Linux 2.6 kernels that support the Kernel GNU Debugger (KGDB).

The term KGDB refers to the kernel **gdb** debugger technology.

With KGDB installed, you can debug the kernel in the same way that you debug applications, including stepping through code, setting breakpoints, and examining variables. Kernel mode debugging between Workbench and the target takes place over a serial protocol connection, which may be over a serial line or Ethernet.

Wind River is a contributor to the KGDB open source project. Basic information on KGDB can be found at http://sourceforge.net/projects/kgdb/.

Wind River has added several features to the mainline KGDB, including:

- Functionality that is tested on all supported boards
- Support for CPU auto-detection
- Additional flags and test features added to the KGDB agent

1.1 Setup for kgdboe between a Target and a Development Host

KGDB over Ethernet is designed to work within single subnets, since it is based on unicast UDP packets.

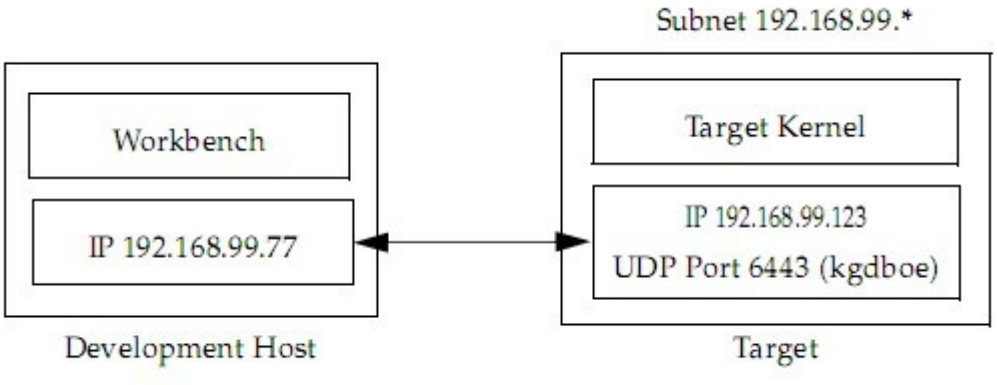

Figure 1.1: Connecting from the host to the target for KGDB

The connection is from the host over UDP to the target, using the standard port 6443 for KGDB.

1.2 Using Agent-Proxy to cross sub-net boundaries for kgdboe

Wind River Linux provides a utility that can convert TCP connections on an intranet to UDP connections in a subnet. If there is an available host that sits on both the intranet and the subnet, the utility agent-proxy can provide this required access. agent-proxy is in the host-tools/bin directory, which you would copy to the bridging host.

Example instructions for using agent-proxy:

1. Log into the bridging host. In this example, the bridging host has one Ethernet connection on the intranet (208.77.190.11), and a second Ethernet connection on the target's subnet (192.168.29.11). (See Figure 1.2.)

2. Start a copy of agent-proxy. This will connect the TCP port 3333 to the UDP port 6443 from 192.168.99.123

   ```
   $ ./agent-proxy 3333 192.168.99.123 udp:6443
   ```

3. Log into the target (192.168.99.123), and entered this command. This will start the kgdboe agent, connected to the bridging host.

   ```
   root@target:/> modprobe kgdboe kgdboe=@192.168.99.123/,@192.168.29.11/
   ```

4. Log into the development host on the intranet.

5. Start Workbench, and create a kgdboe connection with these parameters. This has Workbench connect to the TCP port 3333 on 147.11.200.11, and treat it like a normal KGDB connection.

   TRANSPORT=TCP, CPU=default, IP=208.77.190.11**, Port=3333**

   Workbench connects to the TCP port 3333 on 208.77.190.11, and treats it like a normal KGDB connection.

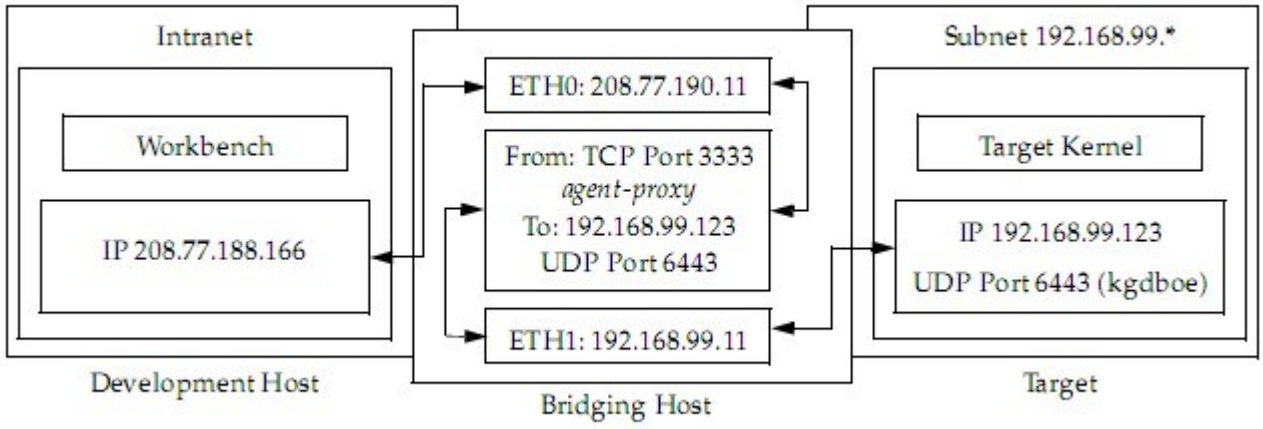

Figure 1.2: Using **agent-proxy** to bridge sub-nets

Notes:

- You can have any number of agent-proxy connections running on the bridging host, one for each target within the sub-net. Just assign a different TCP port, for example 3334, 3335. 3336, and so forth.

- If you run the agent-proxy without any parameters, you will get a list of its commands and abilities, as shown here:

```
$  wrlinux/host-tools/bin/agent-proxy
agent-proxy version 1.6
Usage:
 agentproxy <[udp:][virt IP:]local port> <remote host> <[udp:]remote port>
   For udp wdb agent proxy example for default ports
    agentproxy udp:0x1001 10.0.0.2 udp:0x4321
    agentproxy udp:0x1002 remotehost2 udp:0x4321
   For KGDB udp target to udp host with default g
    agentproxy udp:3331 10.0.0.2 udp:6443
   For KGDB udp target to tcp host with default g
    agentproxy 3332 10.0.0.3 udp:6443
   Also you can bind to a local vitual interface IE:
      agentproxy udp:47.1.1.2:0x4321 10.0.0.2 udp:0x4321
      agentproxy udp:47.1.1.3:0x4321 10.0.0.3 udp:0x4321
      agentproxy udp:47.1.1.3:6443 10.0.0.3 udp:6443
      agentproxy 47.1.1.3:44444 10.0.0.3 44444
```

- The KGDB over serial connection does not have this limitation. You will need either the physical serial cable that can make the connection. Alternatively, you can set up a telnet port repeater that connects the serial connection to a telnet port on the intranet.

2 Notes, Hints, and FAQ's

This section is intended to provide general setup and usage advice, and can also be used as a guide for trouble-shooting.

2.1 Kernels: Enabling KGDB versus Debug Builds

The KGDB feature is a Linux kernel configuration option, as opposed to compiling the kernel using the debug flags. By default, KGDB support is enabled in the pre-built and configured Wind River Linux kernels.

You can configure a Platform Project with **`--enable-debug`**, but this has no effect on KGDB and only applies to the package builds. The pre-built or generated **vmlinux** file contains all the required debug information required for the kernel.

See the Appendix for more details on the KGDB kernel configuration support.

2.2 KGDB and safe areas when debugging the Kernel

Due to the nature of KGDB debugging, not all sections of the kernel can be safely or reliably debugged. In kgdboe for example, it is impossible to debug the ethernet driver itself or any of its related services. The kgdb8560 serial interface option in general has fewer dependencies, but is still has sections where it cannot go, and it runs slower that ther Ethernet connection. The On Chip Debugger (OCD) has the fewest restrictions on where is can debug, since it does not rely on kernel services at all, but does require additional hardware to support it.

Also, there will be different behavior in parts of the kernel depending on the specific architecture or configuration. If a common breakpoint location (for example do_fork) does not seem to work on a particular installation, note it and find an alternative break location.

While these restrictions may discourage some people from employing interactive kernel debugging, this feature does provide tremendous advantages over simple **printk'**s in spite of these restrictions.

Here are some of the sections of the kernel code to specific avoid:

- **Spin locks**. The Ethernet driver has several spin locks, and you attempt to step over one that is needed, then the connection will lock up.
- **Printk**. This routine also has spin locks that will freeze the KGDB connection.
- Simultaneous **usermode-agent** connections. The two debugging connections indirectly and unavoidably share some spin locks.

2.3 Hints using Workbench

Wind River Linux Platform Projects are very large, and can present a challenge to Workbench performance.

If you follow the standard method or using the pre-build packages and even the kernel, this may not be a problem. If you perform a "make all", or build several kernels, then these steps can create up to 28,000 files each project for Workbench to manage and track.

Here is some advice for managing this for Workbench.

2.3.1 Starting Workbench: having enough memory

You should start Workbench with more reserved heap space memory. More is better, if you have the available physical memory.

```
./startWorkbench  -vmargs -Xmx512m
```

You should have at least 512 Mbytes of physical RAM, but 1 Gbyte to 2 Gbytes is explicitly recommended.

2.3.2 Module Optimization Levels and Jumping Program Counters

The kernel will be built with various optimization flags set. If you step through kernel code, you will see the program counter marker jump around as you step through the optimized and re-ordered instructions. This is normal, and proves that optimization was applied.

This will also occur for your kernel modules if you compile without overriding the optimization level, because by default all kernel modules built against a kernel will inherit its optimization. This can be overcome by building your kernel modules without optimization.

1. **Right-click** on the Kernel Module Project, and select **Properties > Build Properties**

2. Change the Build command to:

   make COPTIMIZE=”-O0” *(that is a dash, a capital letter “O”, and a zero)*

When you next build the module, it will have no (or to be exact, minimum) optimization.

2.3.3 Module Optimization Levels and Skipped Breakpoints

Sometimes you may find behavior like a step-next that does not break on the next step. In this example, the compiler when optimizing the code may in fact create two in-line copies, where the break point gets set in the first copy, but the code may branch into the second copy. This is a limitation of some ELF formats, and a resulting optimization and context awareness issue for the debugger.

You can try switching to mixed assembly and source stepping, where you step the actual assembly instructions, but the best choice is again to remove the optimization for the module and retest.

2.3.4 Manual Refresh of the “build” linked directory in Workbench

To help the workflow, Workbench has disabled the automatic project refresh after builds so that the user can immediately chose the next task. The consequence is that you need to manually refresh the “build” directory whenever you wish to “see” any new or revised file within, or else the file may not be visible, or the exusting file ‘handle’ in the Project Navigator might be stale.

2.3.5 Disabling Workspace Refresh on Startup

You may choose to disable the automatic workspace refresh when starting Workbench, particularly when you have one or more Platform Projects.

This is done by selecting **Window > Preferences > General > Startup and Shutdown**, and un-checking **Refresh workspace on startup**.

When you need to browse the ‘build’ tree, you can always manually refresh that directory by **right-clicking** on the project (or specifically the **build** entry) and selecting **Refresh**.

2.3.6 Workbench “freezes” for a period, and will not respond the mouse clicks.

Sometimes Workbench will not immediately fresh its screen nor respond to mouse or keyboard actions. This is often due to (a) a garbage collection timeout, or (b) running low on real and virtual memory. In Wind River Linux Platform Projects, the heap allocation amount can rise and fall 40Mbytes at a time, and will often rise to 500 Mbytes in size.

You can track this behavior be enabling the Workbench heap status monitor, with the command **Windows > Preferences > General > Show Heap Status**. The heap monitor will appear on the lower right-hand corner.

2.3.7 Difficulty inserting breakpoints

Sometime it is hard to insert a breakpoint in the middle of a file, and have it actually planted.

One trick is to set a breakpoint on a global routine to bring in the expected source file to the debugger (Workbench may in fact prompt and guide you if you had not properly set up the proper source code mapping). Once that happens, you can then set the breakpoint that you desire. That global routine can be the same routine that contains the desired location, or it could be an entirely unrelated routine, as long as it is in the same C file.

2.4 Mounting the target's file system via NFS

This is handy for allowing easy kernel module transfers from the host to the target, especially if your target is an NFS mounted file system via **pxeboot**. If you already have the NFS image mounted to your target's file system, you can skip this section.

1. Open a console on your host. Change to super-user mode.

   ```
   $ su
   ```

2. Create a mount directory for the target rootfs.

   ```
   $ mkdir /target
   $ chmod a+rw /target
   ```

3. Mount this directory. In this example, the many lab boards get their root file systems from NFS directories on a shared host.

   $ mount –t nfs ala-tuxlab://export/pxeboot/*my_board***/rootfs /target**

3 Preparing the Target for KGDB

This section describes how to start the KGDB agent on the target. This is required before any connection from Workbench can be made.

This document describes two connection methods to connect to KGDB, the first over the ethernet via UDP, and the second using a serial connection either directly to a COMM port (or otherwise over Telnet). The Ethernet KGDB connections are typically much fast that serial connections.

3.1 Option 1: Using a KGDB Ethernet UDP Connection

Note: if you are using a PCD-type target, go to section 3.2.

On the target with the kgdb enabled kernel, find out the IP address of the target and the IP address of the host where you'll be running gdb/Workbench.

The **kgdboe** module usage syntax is here. It is very rare that you will need to enter the source port or the host mac address.

kgdboe=[*src-port*]@[*src-ip*]/[*dev*],[*hst-port*]@*hst-ip*/[*hst-macaddr*]

It is recommend to explicitly call out the Ethernet port since this guarantees that kgdboe will used the correct port (kgdboe does not know which port the device was booted with). Since more and more boards have multiple Ethernet ports, the default of "eth0" may not be correct.

```
target_# modprobe kgdboe kgdboe=@/eth2,@192.168.29.107/
```

*Note: to underline this point, if this board has multiple Ethernet ports and/or the port you want is not the default, you must explicitly select that alternate Ethernet port. You can use **ipconfig** to identify the proper port to use.*

If you have problems connecting to KGDB, double check which port is configured and connected to the network If for example the **modprobe** commands takes a long time to return, it may be accessing or defaulting to a port that is not connected, and the only other sign of error is that kgdboe connections do not succeed.

Here are some common short cuts for starting kgdboe:

- This version assumes the default ethernet port, for example **eth0**:

```
target_# modprobe kgdboe kgdboe=@/,@192.168.29.107/
```

- This version explicitly calls out the target's IP address:

```
target_# modprobe kgdboe kgdboe=@192.168.29.96/,@192.168.29.107/
```

You can now skip to Section 4.

3.2 Option 2: Using a KGDB Ethernet UDP Connection on the PCD product

In the PCD product, the kgdboe module is statically and automatically linked into the kernel.

There are two ways to configure the kgdboe:

3.2.1 Static configuration of KGDB-OE for PCD (uclibc_small+small)

The first method is to add the kgdboe configuration into the boot command. You must know the IP address of both the target and the host at boot time. You can immediately connect with kgdb anytime after the boot completes.

For example:

```
Target IP: 10.1.1.12
Host   IP: 10.1.1.111
```

On the boot line you would need

```
kgdboe=@10.1.1.12/,@10.1.1.111/
```

So the boot line would look like:

```
boot root=/dev/nfs rw console=ttyS0,115200 kgdboe=@10.1.1.12/,@10.1.1.111/
nfsroot=10.1.1.111:/rootfs ip=10.1.1.12:: 10.1.1.1:255.255.255.0:mainstone:eth0:off mem=128M
```

And during the boot you will see:

```
kgdboe: device eth0 not up yet, forcing it
eth0: link down
eth0: link up, 100Mbps, full-duplex, lpa 0x45E1
kgdboe: debugging over ethernet enabled
kgdb: Defering I/O setup to kernel module.
```

You can now skip to Section 4.

3.2.2 Dynamic configuration of KGDB-OE for PCD (uclibc_small+small)

The kernel now also supports the new dynamic kgdb-oe configuration when used as a kernel built-in. You do:

```
$ echo @/,@10.1.1.111/ > /proc/sys/kgdboe
```

To see the current configuration, do this:

```
$ cat /proc/sys/kgdboe
```

Note that since the device has booted, you do not need to specify its IP address.

You can now skip to Section 4.

3.3 Option 3: Using a KGDB Serial/Telnet Connection

On the target with the kgdb enabled kernel, find out the IP address of the target and the IP address of the host where you'll be running gdb or Workbench.

The **kgdb-8250** module usage syntax is:

8250_kgdb kgdb8250=<*com-port*>,<*baud-rate*>

This example is for a COM1 connection at 115200 baud:

```
target_# modprobe 8250_kgdb kgdb8250=0,115200
```

This example is for a COM2 connection at 9600 baud:

```
target_# modprobe 8250_kgdb kgdb8250=1,9600
```

Notes:

- Do not use a port that has a console attached to it. You may make a connection, but it will not otherwise work, because the console will attempt to interpret the KGDB data as typed commands.
- Also, for 9600 baud connections, you may want to reduce the kgdb watch options to achieve reasonable performance. See the kgdb documents for details.

You can now skip to Section 4.

4 Connecting with KGDB from Workbench

This section guides you in creating the Target Connection to the target for KGDB from Workbench. This requires that the KGDB agent is already running on the target, as described in Chapter 3.

4.1 Option 1: Making a KGDB Ethernet connection from Workbench

1. Start the Workbench on your host.

2. Create a new target connection.

3. Select the connection type "**`Linux KGDB`**", and click **Next**.

4. Select the connection option "**`Linux KGDB via Ethernet`**", and click **Next**.

5. Select **`UDP`** as the transport. *Note: if you are using the agent-proxy, make this a TCP connection, as describe in Chapter 2.*

6. Use the "**Default from Target**" CPU default.

   *Select the Arch only if necessary (for Pentium targets, use **Pentium4** instead of **ia32)**. Use the new architecture families when explicitly selection the target CPU type.*

7. Enter the target's IP address. Note that the IP address must match the setting from Chapter 3.1.

8. Use the port **`6443`**. This port is reserved for kgdb connections. *Note: if you are using the agent-proxy, select the respective agent-proxy port number, as describe in Chapter 2.*

9. Fill in the "Backend Communication Log File" with ***$logpath//kgdb/backend.log***

10. Click **Next**.

11. Select the Target OS **`Linux 2.6`** `if it is not already set.`

12. For the “Kernel Image”, point to your kernel’s **vmlinux.**. *Do not select the kernel image itself (for example the ‘bzImage’ file).*

    $mybuild/**export/*-vmlinux-symbols-***

    Note: if you have built the kernel, the physical location of the new symbol file is here:

    $mybuild/**build/Linux-2.6.14/vmlinux**

13. Click **Next**.

14. For the “Object Path Mappings” page, click **Add**, set to following values, and click **OK**:

    ```
    Target Path = (leave empty)
    Host Path   = /target/
    ```

15. Select **Next** through the following page(s), then select **Done**.

    ***Note**: It may take 20 to 40 seconds to load all the kernel symbols. When it completes, a red “S” appears on the vmlinux element in the target’s connection tree..*

16. Make sure that the target has started the module kgdboe before opening the connection in the Workbench, else you will get a timeout and an connection error summary.

17. When the connection is successful, the Target Manager displays the **connected** message, and the kernel is shown as **Stopped**.

18. You can now proceed to Section 5.

19. If the connection failed, go to Chapter 4.4

4.2 Option 2: Making a KGDB Telnet connection from Workbench

A port selector device can aggregate multiple serial port connections into telnet connections that can be made available across the network. The IP address for this telnet virtual connection would be the one for this port selector device, and the port number would be the one for the port of that target.

For example, if your multiplexer “digiswitch_1” was at 192.168.29.2 and the telnet serial port for the target was 2011, then:

```
IP   = 192.168.29.2
Port = 2011
```

To make a KGDB telnet connection:

1. Start the Workbench on your host.

2. From the Remote Systems view, click **New Target Connection**.
3. Select **Wind River Linux KGDB Connection** and then click **Next**.
4. Select the connection option **Linux KGDB via Terminal Server**, and click **Next**.
5. Select **TCP** as the transport.
6. Use the **Default from Target** CPU default.

   *Select the Arch only if necessary (for Pentium targets, use **Pentium4** instead of **ia32)**.*
7. Enter the target's IP address. Note that the IP address must match device proving the telnet connection.
8. Enter the telnet port.
9. Click **Next**.
10. Continue at step 10 in Section 4.2.
11. You can now proceed to Section 5.
12. If the connection failed, go to Chapter 4.4.

4.3 Option 3: Making a KGDB Serial Cable Connection from Workbench

1. Start the Workbench on your host.
2. From the Remote Systems view, click **New Target Connection**.
3. Select **Wind River Linux KGDB Connection** and then click **Next**.
4. Select the connection option **Linux KGDB via RS-232**, and click **Next**.
5. Select the Baud rate, Data Bits, Parity, Stop Bits, and Flow control appropriate to the connection. Note that the baud rate must match the setting from Chapter 3.2.
6. Click **Next**.
7. Continue at step 10 in Section 4.2.
8. You can now proceed to Section 5.
9. If the connection failed, go to Chapter 4.4.

4.4 When a KGDB Connection Fails

There are several reasons a connection can fail. Here are some things to try to overcome this problem.

- Make sure the KGDB agent on the target is running. You can use the **lsmod** command on the target to see if the module is loaded.
- Make sure that the KGDB agent is using a functional network driver. For example, run the command **ipconfig** on the target, and double confirm that the kgdboe agent is using the right Ethernet driver.
- Double check all the IP address for the target and host are correct, by running **ipconfig** on both the target and host.
- Try to ping the target from the host, and the host from the target, to test basic connectivity.
- The ping command uses TCP, and KGDB uses the more restrictive unicast USP packet, so you also need to check that you are on the same sub-net, or have setup an “intranet to sub-net” agent, as described in Chapter 1.
- Make sure that there are no firewalls on the host or target that may be blocking the connection.
- Make sure that there is only one copy of the Wind River WTX Target Registry running. You can try killing any running copy(s), and have Workbench start a fresh instance. You may need to restart Workbench.
- Make sure you have declared "**127.0.0.1 localhost**" within your “**/etc/hosts”** file, so that the Workbench Target Manager can find the Target Registry on the host.
- You can turn on the debug log for the KGDB connection by adding the following to the advanced options box of the connection wizard.
    - **Right click** on the Connection
    - Select **Properties**
    - Fill in a location for the log in the **Backend Log**.
    - Attempt the connect again
    - Select **Help > Collect Logfiles,** which will gather the connection log files.
    - Forward this to the Wind River Support team.

5 Attaching and Debugging the Kernel from Workbench

After you establish a KGDB connection, Workbench lets you access the target kernel and debug source code.

If the connection had failed, go to Chapter 4.4.

1. One the connection is established, Workbench shifts to the Embedded Debug perspective and the Editor opens to the stopped location in the **kgdb.c** source file.

   Note: If you use the pre-built vmlinux file that comes with the pre-built kernel (in wrlinux-1.x/boards),

*the path to the source files will not be immediately known to Workbench. To resolve this. right-click the entry in the Debug view and then select **Edit Source Lookup**.*

2. If the Compilation Unit Source Is Not Found

If the Editor where kgdb.c should appear instead displays a message that says Source not found for compilation unit, your vmlinux file is either missing or, more likely, does not point to the current location of your Linux kernel source.

To f ix this:

1) Click the Browse button in this error message and navigate to your Linux kernel's source directory. If you have performed a "***make fs***", "***make –C build linux***", or a "***make build-all***", the linux source folder will have been expanded in your project's build folder.For Example, */home/user/workdir/myBoard/***build/linux-2.6.14-**<type>

2) Click **Next** until you can click Finish.
Workbench resolves the location of **kgdb.c** and any other kernel source.You can view the mappings between the object files and the respective source files by right-clicking the entry in the Debug view and selecting **Edit Source Lookup**.

3. **If the Target's Terminal Is Stopped**

If you have a terminal open to the target, you may see that it is stopped. For example, you may not be able to enter commands in the terminal. To resume execution of the kernel, click Resume in the Debug view.

4. Setting and Removing a Sample Breakpoint

With the kernel running again, set a breakpoint as follows:

1) Select **Run > Add Expression Breakpoint**.

2) In the **Breakpoint Location and General Attributes** dialog, enter **do_fork** in the **Location Expression** box.

3) Click OK.

4) Enter a command, for example **ls** (press **ENTER** only once,) on the target.

It does not return because execution has stopped at the fork. The Editor opens the **fork.c** source file. Click **Resume** in the Debug view to allow the command to complete; the next fork, for example another command, again causes a break.

5) To complete this example, remove the breakpoint and click Resume in the Debug view to continue execution of the kernel.

6) To end the session, right-click the target connection in the Remote Systems view and select **Disconnect**.

***Note**: if the do_fork location as a breakpoint does not work on this installation, note it and choose another kernel entry point. This is related to the comments in Chapter 2 on 'safe' areas of the kernel.*

6 Debugging User-Supplied Kernel Modules from Workbench

Workbench includes a sample project to demonstrate kernel module debugging as discussed in this section. This example requires that you have access to the kernel source you used to build the kernel and that you have terminal or console window access to the target.

6.1 Debugging a running User Kernel Module

1. In the terminal view or console on the target, get the new file to the target, and use the **insmod** command to install the module.

   ```
   kgdb_target_$ ftp <your build  host>
   kgdb_target_$ cd ${Project_path}/project/moduledebug_2_6
   kgdb_target_$ get moduleDebugSample.ko
   kgdb_target_$ quit
   kgdb_target_$ insmod moduleDebugSample.ko
   ```

   To see the periodic output from this user module, enter the following command to re-direct the kernel print statements to the console. You will see a message display every second or so. You may want to open a second telnet or a ssh session to remove this module when you are done.

   ```
   kgdb_target_$ klogd –c 8
   ```

   If the klogd is already running and there is no output, you can find the PID, kill the current instance, and restart the deamon.

   ```
   kgdb_target_$ cat /var/run/klogd.pid
                 1084
   kgdb_target_$ kill 1084
   kgdb_target_$ klogd –c 8
   ```

   If you still do not see output, you can do the following:

   ```
   kgdb_target_$ cat /proc/kmsg
   ```

   Note: there are two messages being internally printed, a 'global message' and a 'local' message. Depending on the display level, you may see both messages interspersed.

2. If you are connected to the target, disconnect. Once the workbench is again ready, reconnect to the target with the kgdb connection.

   Note: If you have placed ***moduleDebugSample.ko*** *in a location other that the root, adjust the target connection's object mapping list to include that location, and then connect to the target. WARNING: make sure that the host path as a forward slash at the end.*

3. Find the **moduleDebugSample.ko** entry in the Target Manager view, below the entry for vmlinux. If there is not a red 'S' mark on its icon, right-click on it and select "Load Symbols". The red 'S' should appear after a moment indicating that symbols were indeed loaded.

   Note: if the symbols do not load, or if moduleDebugSample.ko does not appear in the Target

Connection view, go back to step 9 and adjust the Object Mappings.

4. Then right-click the *CPU*:*Kernel* entry in the Target Manager and select **Attach to Core**. Click the **Resume** icon in the Debug view so that the kernel is Running.

5. Click the **System Comtext** in **Debug** window, and Click **Suspend** in Debug window menu.

6. Select **Run >Add Expression Breakpoint** and in the **Breakpoint Location and General Attributes** dialog, enter **putABreakPointHere** in the **Location Expression** box. Click **OK**. *(see note below).*

7. *Click the **System Comtext** in **Debug** window, and* Click **Resume** *in Debug window menu.*

8. Once it breaks, try **single steps** in and then out of this procedure. Then click the **Run** button.

9. When it breaks again, disable the break point and click the **Resume** button.

10. You can remove the module from the kernel with the **rmmod** command:

    ```
    kgdb_target_$ rmmod moduleSampleDebug.ko
    ```

6.2 Stepping into User-Supplied Kernel Module init from module.c

The section describes how you can place a breakpoint in the kernel's module load routine, and step directly into a loading module's init routine.

This technique can be used in many ways, beyond this specific example. You can also see the next section for fast ways to break within specific kernel module init routines, loaded or not.

1. In Workbench, open the **module.c** file in the kernel source tree. Select **File > Open** and browse to the **kernel** subdirectory of your build's Linux-2.6.14 source tree root. Double-click on the **module.c** file to open it in the Editor.

   For example, module.c might be found at:

   ```
   /home/user/workdir/myboard/build/linux-2.6.14-<type>/kernel/module.c
   ```

2. Go to the rountine **sys_init_module** and find the below line start calls the modules init procedure (about line 1852 depending on your particular source file. Right-click in the gutter to insert a **System Context** breakpoint at that line.

   ```
         /* Start the Module */
   =>    if (mod->init != NULL)
               ret = mod->init();
   ```

   Warning: you may find that the break point cannot be planted. In this case make sure you added the Object Mapping as described in Chapter 4.1, step 13. Also, do not single step from the break at **sys_init_module** to this location – you will hit spin locks that will

block your KGDB connection.

3. On the target, start the module.

```
kgdb_target_$ insmod moduleDebugSample.ko
```

4. In a moment, the Debug view in Workbench will show the System Context as stopped and **load_module** highlighted, and the Editor will open **module.c** file to the location of the breakpoint. Note that the target window is hung at the **insmod** command.

5. Insure that the **moduleSampleDebug.ko** is registered by the Target Manager. Examine the Target Connection window for your target, and look for **moduleSampleDebug.ko** in the list on installed kernel modules, under "Processes". If is not present, do the following steps to synchronize the target manager:

   a. Un-expand the icon for the target labeled "Processes".
   b. Click the **Refresh** button in the Target Manager's tool bar.
   c. Re-expand the icon for the target labeled "Processes".
   d. It the module does not yet appear, repeat steps 'a' to 'd'.

6. Insure that the symbols for **moduleSampleDebug.ko** are loaded. Examine the Target Connection window for your target, and look for **moduleSampleDebug.ko** in the list on installed kernel modules, under **Processes**. The icon for this module should be decorated with a red **S**. If it is not, right click on the module and select **Load Symbols**.

7. You can now single step into the initialization routine of the kernel module.

8. To resume processing, remove the breakpoint and click the **Resume** icon in the Debug view. The **insmod** command completes. You can remove the module from the kernel with the **rmmod** command:

```
kgdb_target_$ rmmod moduleSampleDebug.ko
```

6.3 Placing Breakpoints in modules not yet loaded

When a kernel module is loaded, it tries to load the symbol table and place any pending breakpoints. You can make use of this to break in the init routine of a kernel module, as follows:

1. Right-click the GDB target connection entry in the Remote Systems view.

2. Select **Properties** > **Object Path Mapping**

3. Click **Add**

4. Leave the **Target Path** empty.

5. Click **Browse** for the **Host Path,** and select the <**Project_path**>/**moduledebug_2_6** directory, where the its generated **moduleDebugExample.ko** file is found.

6. Click **OK** for the path, and **OK** for the Target Connection's properties dialog.

7. Make the connection to the target. If that connection is already active, disconnect and reconnect to assert the new mappings.

8. Select **Run >Add Expression Breakpoint** and in the **Breakpoint Location and General Attributes** dialog, enter **putABreakPointHere** in the **Location Expression** box. Click **OK**. *(see note below).*

   Note: you may get a dialog stating that the location for the breakpoint is not available. This is expected, in that the module is not currently loaded. Click **OK** to continue.

9. Load the module on the target. Observe the breakpoint being hit, and the source being opened to that location.

   ```
   $ insmod moduleDebugExample.ko
   ```

   Observe that the breakpoint is hit.

10. You can debug the init routine, or **Resume** the execution.

11. If you remove the module, you may get another dialog stating that the location for the breakpoint is not available. This is again expected, in that the module is unloaded. Click **OK** to continue.

    ```
    $ rmmod moduleDebugExample.ko
    ```

6.4 Using the testmod Enhanced Sample Kernel Module

Workbench includes a sample kernel module that demonstrates the following features: (a) simple kernel module, (b) parameter passing, (c) a kernel timer, and (d) using systcl to change the timeout.

The file **main.c** initializes and removes the module. The file **timer.c** manages the timers. The file **settings.c** manages the sysctl bindings.

1. In the terminal view or console on the target, get the new file to the target, and use the **insmod** command to install the module.

   ```
   kgdb_target_$ ftp <your host>
   kgdb_target_$ cd ${Project_path}/project/testmod
   kgdb_target_$ get testmod.ko
   kgdb_target_$ quit

   kgdb_target_$ insmod testmod.ko timeout=2340
   ```

2. Look at console output:

   ```
   kgdb_target_$ dmesg | tail
   ```

   You should see:

   ```
   Loading testmod
   timeout is 2340
   ```

3. Here are some additional sysctl supported commands for this module:

   - **sysctl -a** will show all variables
   - **sysctl dev.testmod.timeout** will query that variable
   - **sysctl dev.testmod.timeout=5000** will set that variable
   - Can be queried and set also by looking in **/proc/sys/dev/testmod/timeout**

4. The module can be removed with:

   ```
   kgdb_target_$ rmmod testmod.ko
   ```

7 Debugging with KGDB from the Command Line

You can make a KGDB connection from the command line using **gdb**.

This is useful if (a) you are more familiar with gdb for particular types of debugging, (b) you wish to automate some KGDB tests, or (c) if you are having problems with your KGDB connection from Workbench.

1. Go to your project build directory (this makes it easier to provide the path to the vmlinux symbol file):

   $ cd *prjbuildDir*

2. Locate your cross-compiled version of gdb.You can find one in the project's host-cross directory, for example this one for a powerpc project where the gdb binary name has a prefix for its cross compile architecture, for example:

   ```
   ./host-cross/arm-wrs-linux-gnueabi/x86-linux2/arm-wrs-linux-gnueabi-gdb
   ```

   Note: If the host and target architectures are the same, you can use the host's gdb.

3. Run the cross-compiled version of gdb on your vmlinux. You will see various banners when it starts.

   ```
   $ ./host-cross/arm-wrs-linux-gnueabi/x86-linux2/arm-wrs-linux-gnueabi-gdb \
   export/*vmlinux-symbols*

   GNU gdb Red Hat Linux (5.3.90-0.20030710.40rh)
   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 "i386-redhat-linux-gnu"...Using host libthread_db library
   "/lib/tls/libthread_db.so.1".
   ```

4. For some boards, you need to assert the architecture for gdb.

   For the 8560, it is necessary to specify:

   (gdb) set architecture powerpc:common

   For a MIPS-64 CPU board with a 32-bit kernel, it is necessary to specify:

   (gdb) set architecture mips

   Note: without this setting, gdb will continually respond with errors like "Program received signal SIGTRAP, Trace/breakpoint trap. 0x00000000 in ?? ()", or "".

5. In the gdb session, connect to the target. Port 6443 is reserved for kgdb communication. You will see various warnings and exceptions that you can ignore. If, however, gdb informs you that the connection was not made, review your configuration, command syntax's, and the IP addresses used.

   (gdb) target remote udp:<target IP>:6443

   ```
   warning: The remote protocol may be unreliable over UDP.
   warning: Some events may be lost, rendering further debugging impossible.
   Remote debugging using udp:192.168.29.96:6443
   0xc013db8e in kgdb_handle_exception (exVector=1282, signo=9,
       err_code=-1072564100, linux_regs=0x0) at kernel/kgdb.c:1008
   1008                                    sleeping_thread_to_gdb_regs(gdb_regs, thread);
   warning: shared library handler failed to enable breakpoint
   ```

6. Enter the "where" command, and note the output. You should see a backtrace stack of some depth. If you see only one or two entries, or a "??", then you are observing an error .

   (gdb) where

7. Enter the "info registers" command, and note the output. You should see the list of registers. Exmanine this list. If for example the Program Counter (the last entry) is zero or otherwise un-reasonable, then you are observing an error .

   (gdb) info registers

8. Enter a breakpoint command for do_fork.

   (gdb) break do_fork

9. On the target, enter a "ls" command and *one* return key.

   target_# ls<*enter*>

10. On the host, see that the breakpoint was hit.

    Breakpoint 1 at 0xc011a863: file kernel/fork.c, line 1120.

11. Continue the target execution. Note that the target resumes normal operation.

    (gdb) c

12. Now hit CTRL-C to get the gdb prompt back, and have it wait for the next breakpoint.

```
  <CTRL-C>
(gdb)
```

13. From here, you can ^C to send a break, set breakpoints, view the stack, view variables, etc. etc. etc. You may wish to build the kernel with CONFIG_DEBUG_INFO=y if you want more debugging info.

14. Release the kgdb connection.

```
(gdb) quit
```

8 KGDB Debugging Using the Serial Console (KGDBOC)

KGDBOC permits KGDB debugging operations using the serial console. The serial console operates in two modes—the usual mode in which you use the serial console to login and so on, and a mode that allows you to enter the KGDB debugger.

KGDBOC requires a serial polling driver which is available with the following drivers on Wind River Linux Platform targets:

- 8250 (most common targets)
- plb011 (ARM versatile)
- CPM (various 82xx, 83xx, 85xx)
- MPSC (ppmc280 ATCAf101)

8.1 Target Preparation

To use KGDBOC you must specify the device assigned to the console. You can find this in the console= argument in your target's boot line. You can also view the boot line at runtime with the command **cat /proc/cmdline.** For example, on an ARM Versatile 926EJS target, console=**ttyAMA0**. On a common PC target, console=**ttyS0**.

Load the kernel module, supplying the appropriate port, for example:

 # modprobe kgdboc kgdboc=ttyS0

8.2 Host Preparation

On your development host, run the agent-proxy from your project build directory:

$ ./host-cross/bin/agent-proxy arguments

For example, if you are using a terminal server (128.224.50.30 on port 2011), the command would be:

$ agent-proxy 2222^2223 128.224.50.30 2011

If you have the target directly connected to your Linux host:

$ agent-proxy 2222^2223 0 /dev/ttyS0,115200

Replace /dev/ttyS0,115200 with your serial port device and baud rate to the target.

NOTE: This program turns your host into a mini terminal server.

After agent-proxy has properly connected to the target, the console port is now multiplexed into a pass-through console and a debug port, which will automatically send the SYSRQ sequence. You can use the Workbench terminal view or a Telnet program to connect to the target console as follows:

$ telnet localhost 2222

For the KGDB connection with Workbench, specify a terminal server connection to TCP port 2223.

If you use gdb to connect to KGDB, use the following command to connect:

$ target remote localhost:2223

NOTE: When the KGDB connection is active you will see the raw KGDB data appear on the pass-through console connection.

9 KGDB test suite (kgdbts)

kgdbts is a test suite for kgdb for the sole purpose of validating that key pieces of the kgdb internals are working properly such as HW/SW breakpoints, single stepping, and NMI.

9.1 Steps

1. Enable the kernel option: CONFIG_KGDB_TESTS=y

2. Rebuild the kernel.

3. The kgdb suite can be invoked from the kernel command line arguments system or executed dynamically at run time.

    1) Executed at run time:

        a) After the target boot run the basic test.
        * echo kgdbts=V1 > /sys/module/kgdbts/parameters/kgdbts

        b) Run the concurrency tests. It is best to use n+1 while loops where n is the number of cpus you have in your system.

        The example below uses only two loops:
        * ## This tests break points on sys_open
        * while [1] ; do find / > /dev/null 2>&1 ; done &
        * while [1] ; do find / > /dev/null 2>&1 ; done &
        * echo kgdbts=V1S10000 > /sys/module/kgdbts/parameters/kgdbts

* fg # and hit control-c
* fg # and hit control-c
* ## This tests break points on do_fork
* while [1] ; do date > /dev/null ; done &
* while [1] ; do date > /dev/null ; done &
* echo kgdbts=V1F1000 > /sys/module/kgdbts/parameters/kgdbts
* fg # and hit control-c

2) Invoked from boot(optional):

Boot with the test suite enabled by using the kernel arguments "kgdbts=V1F100 kgdbwait"

If kgdb arch specific implementation has NMI use "kgdbts=V1N6F100

or

Enable the kernel option: CONFIG_KGDB_TESTS_ON_BOOT=y;
CONFIG_KGDB_TESTS_BOOT_STRING="V1F100"

Rebuild the kernel, with which bootup the target, the kgdbts test will be invoked during bootup.

Please refer to the Notes below or see the PRJ_DIR/build/ linux/drivers/misc/kgdbts.c for details about the tests.

9.2 Results example

Here is an example of the results:

```
root@localhost:/root> echo kgdbts=V1 > /sys/module/kgdbts/parameters/kgdbts
kgdb: Registered I/O driver kgdbts.
kgdbts:RUN plant and detach test
kgdbts:RUN sw breakpoint test
kgdbts:RUN bad memory access test
kgdbts:RUN singlestep test 1000 iterations
kgdbts:RUN singlestep [0/1000]
kgdbts:RUN singlestep [100/1000]
kgdbts:RUN singlestep [200/1000]
kgdbts:RUN singlestep [300/1000]
kgdbts:RUN singlestep [400/1000]
kgdbts:RUN singlestep [500/1000]
kgdbts:RUN singlestep [600/1000]
kgdbts:RUN singlestep [700/1000]
kgdbts:RUN singlestep [800/1000]
kgdbts:RUN singlestep [900/1000]
kgdb: Unregistered I/O driver kgdbts, debugger disabled.
root@localhost:/root> while [ 1 ] ; do find / > /dev/null 2>&1 ; done &
[1] 4141
root@localhost:/root> while [ 1 ] ; do find / > /dev/null 2>&1 ; done &
[2] 4145
```

```
soot@localhost:/root> echo kgdbts=V1S10000 > /sys/module/kgdbts/parameters/kgdbt
kgdb: Registered I/O driver kgdbts.
kgdbts:RUN plant and detach test
kgdbts:RUN sw breakpoint test
kgdbts:RUN bad memory access test
kgdbts:RUN singlestep test 1000 iterations
kgdbts:RUN singlestep [0/1000]
kgdbts:RUN singlestep [100/1000]
kgdbts:RUN singlestep [200/1000]
kgdbts:RUN singlestep [300/1000]
kgdbts:RUN singlestep [400/1000]
kgdbts:RUN singlestep [500/1000]
kgdbts:RUN singlestep [600/1000]
kgdbts:RUN singlestep [700/1000]
kgdbts:RUN singlestep [800/1000]
kgdbts:RUN singlestep [900/1000]
kgdbts:RUN sys_open for 10000 breakpoints
root@localhost:/root> fg
while [ 1 ]; do
    find / > /dev/null 2>&1;
done
^C
root@localhost:/root> fg
while [ 1 ]; do
    find / > /dev/null 2>&1;
done
^C
root@localhost:/root> while [ 1 ] ; do date > /dev/null ; done &
[1] 4148
root@localhost:/root> while [ 1 ] ; do date > /dev/null ; done &
[2] 4394
root@localhost:/root> echo kgdbts=V1F1000 > /sys/module/kgdbts/parameters/kgdbts
kgdb: Registered I/O driver kgdbts.gdbts/parameters/kgdbts
kgdbts:RUN plant and detach test
kgdbts:RUN sw breakpoint test
kgdbts:RUN bad memory access test
kgdbts:RUN singlestep test 1000 iterations
kgdbts:RUN singlestep [0/1000]
kgdbts:RUN singlestep [100/1000]
kgdbts:RUN singlestep [200/1000]
kgdbts:RUN singlestep [300/1000]
kgdbts:RUN singlestep [400/1000]
kgdbts:RUN singlestep [500/1000]
kgdbts:RUN singlestep [600/1000]
kgdbts:RUN singlestep [700/1000]
kgdbts:RUN singlestep [800/1000]
kgdbts:RUN singlestep [900/1000]
kgdbts:RUN do_fork for 1000 breakpoints
root@localhost:/root> fg
while [ 1 ]; do
```

```
   date > /dev/null;
done
^C
root@localhost:/root> fg
while [ 1 ]; do
   date > /dev/null;
done
^C
root@localhost:/root>
```

9.3 Detail information about the KGDB test suite

Information about the kgdb test suite:

The kgdb test suite is designed as a KGDB I/O module which simulates the communications that a debugger would have with kgdb. The tests are broken up in to a line by line and referenced here as a "get" which is kgdb requesting input and "put" which is kgdb sending a response.

The kgdb suite can be invoked from the kernel command line arguments system or executed dynamically at run time. The test suite uses the variable "kgdbts" to obtain the information about which tests to run and to configure the verbosity level. The following are the various characters you can use with the kgdbts=line:

When using the "kgdbts=" you only choose one of the following core test types:
A = Run all the core tests silently
V1 = Run all the core tests with minimal output
V2 = Run all the core tests in debug mode

You can also specify optional tests:
N## = Go to sleep with interrupts of for ## seconds
     to test the HW NMI watchdog
F## = Break at do_fork for ## iterations
S## = Break at sys_open for ## iterations
I## = Run the single step test ## iterations

NOTE: that the do_fork and sys_open tests are mutually exclusive.

To invoke the kgdb test suite from boot you use a kernel start argument as follows:
   kgdbts=V1 kgdbwait
Or if you wanted to perform the NMI test for 6 seconds and do_fork test for 100 forks, you could use:
   kgdbts=V1N6F100 kgdbwait

The test suite can also be invoked at run time with:
   echo kgdbts=V1N6F100 > /sys/module/kgdbts/parameters/kgdbts
Or as another example:

```
    echo kgdbts=V2 > /sys/module/kgdbts/parameters/kgdbts
```

When developing a new kgdb arch specific implementation or using these tests for the purpose of regression testing,
several invocations are required.

1) Boot with the test suite enabled by using the kernel arguments
   "kgdbts=V1F100 kgdbwait"
  ## If kgdb arch specific implementation has NMI use
   "kgdbts=V1N6F100

2) After the system boot run the basic test.
echo kgdbts=V1 > /sys/module/kgdbts/parameters/kgdbts

3) Run the concurrency tests. It is best to use n+1
  while loops where n is the number of cpus you have
  in your system. The example below uses only two
  loops.

```
## This tests break points on sys_open
while [ 1 ] ; do find / > /dev/null 2>&1 ; done &
while [ 1 ] ; do find / > /dev/null 2>&1 ; done &
echo kgdbts=V1S10000 > /sys/module/kgdbts/parameters/kgdbts
fg # and hit control-c
fg # and hit control-c
## This tests break points on do_fork
while [ 1 ] ; do date > /dev/null ; done &
while [ 1 ] ; do date > /dev/null ; done &
echo kgdbts=V1F1000 > /sys/module/kgdbts/parameters/kgdbts
fg # and hit control-c
```