```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="kgdbOnLinux">
 <bookinfo>
  <title>Using kgdb, kdb and the kernel debugger internals</title>

  <authorgroup>
   <author>
    <firstname>Jason</firstname>
    <surname>Wessel</surname>
    <affiliation>
     <address>
      <email>jason.wessel@windriver.com</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>
  <copyright>
   <year>2008,2010</year>
   <holder>Wind River Systems, Inc.</holder>
  </copyright>
  <copyright>
   <year>2004-2005</year>
   <holder>MontaVista Software, Inc.</holder>
  </copyright>
  <copyright>
   <year>2004</year>
   <holder>Amit S. Kale</holder>
  </copyright>

  <legalnotice>
   <para>
   This file is licensed under the terms of the GNU General Public License
   version 2. This program is licensed "as is" without any warranty of any
   kind, whether express or implied.
   </para>

  </legalnotice>
 </bookinfo>

<toc></toc>
  <chapter id="Introduction">
    <title>Introduction</title>
    <para>
    The kernel has two different debugger front ends (kdb and kgdb)
    which interface to the debug core.  It is possible to use either
    of the debugger front ends and dynamically transition between them
    if you configure the kernel properly at compile and runtime.
    </para>
    <para>
    Kdb is simplistic shell-style interface which you can use on a
    system console with a keyboard or serial console.  You can use it
    to inspect memory, registers, process lists, dmesg, and even set
    breakpoints to stop in a certain location.  Kdb is not a source
```

level debugger, although you can set breakpoints and execute some
basic kernel run control. Kdb is mainly aimed at doing some
analysis to aid in development or diagnosing kernel problems. You
can access some symbols by name in kernel built-ins or in kernel
modules if the code was built
with <symbol>CONFIG_KALLSYMS</symbol>.
</para>
<para>
Kgdb is intended to be used as a source level debugger for the
Linux kernel. It is used along with gdb to debug a Linux kernel.
The expectation is that gdb can be used to "break in" to the
kernel to inspect memory, variables and look through call stack
information similar to the way an application developer would use
gdb to debug an application. It is possible to place breakpoints
in kernel code and perform some limited execution stepping.
</para>
<para>
Two machines are required for using kgdb. One of these machines is
a development machine and the other is the target machine. The
kernel to be debugged runs on the target machine. The development
machine runs an instance of gdb against the vmlinux file which
contains the symbols (not boot image such as bzImage, zImage,
uImage...). In gdb the developer specifies the connection
parameters and connects to kgdb. The type of connection a
developer makes with gdb depends on the availability of kgdb I/O
modules compiled as built-ins or loadable kernel modules in the test
machine's kernel.
</para>
</chapter>
<chapter id="CompilingAKernel">
<title>Compiling a kernel</title>
<para>
<itemizedlist>
<listitem><para>In order to enable compilation of kdb, you must first enable
kgdb.</para></listitem>
<listitem><para>The kgdb test compile options are described in the kgdb test
suite chapter.</para></listitem>
</itemizedlist>
</para>
<sect1 id="CompileKGDB">
<title>Kernel config options for kgdb</title>
<para>
To enable <symbol>CONFIG_KGDB</symbol> you should first turn on
"Prompt for development and/or incomplete code/drivers"
(CONFIG_EXPERIMENTAL) in "General setup", then under the
"Kernel debugging" select "KGDB: kernel debugger".
</para>
<para>
While it is not a hard requirement that you have symbols in your
vmlinux file, gdb tends not to be very useful without the symbolic
data, so you will want to turn
on <symbol>CONFIG_DEBUG_INFO</symbol> which is called "Compile the
kernel with debug info" in the config menu.
</para>
<para>
It is advised, but not required that you turn on the

<symbol>CONFIG_FRAME_POINTER</symbol> kernel option which is called "Compile the
kernel with frame pointers" in the config menu. This option
inserts code to into the compiled executable which saves the frame
information in registers or on the stack at different points which
allows a debugger such as gdb to more accurately construct
stack back traces while debugging the kernel.
</para>
<para>
If the architecture that you are using supports the kernel option
CONFIG_DEBUG_RODATA, you should consider turning it off. This
option will prevent the use of software breakpoints because it
marks certain regions of the kernel's memory space as read-only.
If kgdb supports it for the architecture you are using, you can
use hardware breakpoints if you desire to run with the
CONFIG_DEBUG_RODATA option turned on, else you need to turn off
this option.
</para>
<para>
Next you should choose one of more I/O drivers to interconnect
debugging host and debugged target. Early boot debugging requires
a KGDB I/O driver that supports early debugging and the driver
must be built into the kernel directly. Kgdb I/O driver
configuration takes place via kernel or module parameters which
you can learn more about in the in the section that describes the
parameter "kgdboc".
</para>
<para>Here is an example set of .config symbols to enable or
disable for kgdb:
<itemizedlist>
<listitem><para># CONFIG_DEBUG_RODATA is not set</para></listitem>
<listitem><para>CONFIG_FRAME_POINTER=y</para></listitem>
<listitem><para>CONFIG_KGDB=y</para></listitem>
<listitem><para>CONFIG_KGDB_SERIAL_CONSOLE=y</para></listitem>
</itemizedlist>
</para>
</sect1>
<sect1 id="CompileKDB">
<title>Kernel config options for kdb</title>
<para>Kdb is quite a bit more complex than the simple gdbstub
sitting on top of the kernel's debug core. Kdb must implement a
shell, and also adds some helper functions in other parts of the
kernel, responsible for printing out interesting data such as what
you would see if you ran "lsmod", or "ps". In order to build kdb
into the kernel you follow the same steps as you would for kgdb.
</para>
<para>The main config option for kdb
is <symbol>CONFIG_KGDB_KDB</symbol> which is called "KGDB_KDB:
include kdb frontend for kgdb" in the config menu. In theory you
would have already also selected an I/O driver such as the
CONFIG_KGDB_SERIAL_CONSOLE interface if you plan on using kdb on a
serial port, when you were configuring kgdb.
</para>
<para>If you want to use a PS/2-style keyboard with kdb, you would
select CONFIG_KDB_KEYBOARD which is called "KGDB_KDB: keyboard as
input device" in the config menu. The CONFIG_KDB_KEYBOARD option

```
    is not used for anything in the gdb interface to kgdb.  The
    CONFIG_KDB_KEYBOARD option only works with kdb.
    </para>
    <para>Here is an example set of .config symbols to enable/disable kdb:
    <itemizedlist>
    <listitem><para># CONFIG_DEBUG_RODATA is not set</para></listitem>
    <listitem><para>CONFIG_FRAME_POINTER=y</para></listitem>
    <listitem><para>CONFIG_KGDB=y</para></listitem>
    <listitem><para>CONFIG_KGDB_SERIAL_CONSOLE=y</para></listitem>
    <listitem><para>CONFIG_KGDB_KDB=y</para></listitem>
    <listitem><para>CONFIG_KDB_KEYBOARD=y</para></listitem>
    </itemizedlist>
    </para>
  </sect1>
  </chapter>
  <chapter id="kgdbKernelArgs">
  <title>Kernel Debugger Boot Arguments</title>
  <para>This section describes the various runtime kernel
  parameters that affect the configuration of the kernel debugger.
  The following chapter covers using kdb and kgdb as well as
  provides some examples of the configuration parameters.</para>
   <sect1 id="kgdboc">
   <title>Kernel parameter: kgdboc</title>
   <para>The kgdboc driver was originally an abbreviation meant to
   stand for "kgdb over console".  Today it is the primary mechanism
   to configure how to communicate from gdb to kgdb as well as the
   devices you want to use to interact with the kdb shell.
   </para>
   <para>For kgdb/gdb, kgdboc is designed to work with a single serial
   port. It is intended to cover the circumstance where you want to
   use a serial console as your primary console as well as using it to
   perform kernel debugging.  It is also possible to use kgdb on a
   serial port which is not designated as a system console.  Kgdboc
   may be configured as a kernel built-in or a kernel loadable module.
   You can only make use of <constant>kgdbwait</constant> and early
   debugging if you build kgdboc into the kernel as a built-in.
   </para>
   <sect2 id="kgdbocArgs">
   <title>kgdboc arguments</title>
   <para>Usage:
<constant>kgdboc=[kbd][[,]serial_device][,baud]</constant></para>
   <sect3 id="kgdbocArgs1">
   <title>Using loadable module or built-in</title>
   <para>
   <orderedlist>
   <listitem><para>As a kernel built-in:</para>
   <para>Use the kernel boot argument:
<constant>kgdboc=&amp;lt;tty-device&amp;gt;,[baud]</constant></para></listitem>
   <listitem>
   <para>As a kernel loadable module:</para>
   <para>Use the command: <constant>modprobe kgdboc
kgdboc=&amp;lt;tty-device&amp;gt;,[baud]</constant></para>
   <para>Here are two examples of how you might formate the kgdboc
   string. The first is for an x86 target using the first serial port.
   The second example is for the ARM Versatile AB using the second
   serial port.
```

```
   <orderedlist>
   <listitem><para><constant>kgdboc=ttyS0,115200</constant></para></listitem>
   <listitem><para><constant>kgdboc=ttyAMA1,115200</constant></para></listitem>
   </orderedlist>
   </para>
   </listitem>
   </orderedlist></para>
   </sect3>
   <sect3 id="kgdbocArgs2">
   <title>Configure kgdboc at runtime with sysfs</title>
   <para>At run time you can enable or disable kgdboc by echoing a
   parameters into the sysfs.  Here are two examples:</para>
   <orderedlist>
   <listitem><para>Enable kgdboc on ttyS0</para>
   <para><constant>echo ttyS0 &gt;
/sys/module/kgdboc/parameters/kgdboc</constant></para></listitem>
   <listitem><para>Disable kgdboc</para>
   <para><constant>echo "" &gt;
/sys/module/kgdboc/parameters/kgdboc</constant></para></listitem>
   </orderedlist>
   <para>NOTE: You do not need to specify the baud if you are
   configuring the console on tty which is already configured or
   open.</para>
   </sect3>
   <sect3 id="kgdbocArgs3">
   <title>More examples</title>
   <para>You can configure kgdboc to use the keyboard, and or a serial device
   depending on if you are using kdb and or kgdb, in one of the
   following scenarios.
   <orderedlist>
   <listitem><para>kdb and kgdb over only a serial port</para>
   <para><constant>kgdboc=&lt;serial_device&gt;[,baud]</constant></para>
   <para>Example: <constant>kgdboc=ttyS0,115200</constant></para>
   </listitem>
   <listitem><para>kdb and kgdb with keyboard and a serial port</para>
   <para><constant>kgdboc=kbd,&lt;serial_device&gt;[,baud]</constant></para>
   <para>Example: <constant>kgdboc=kbd,ttyS0,115200</constant></para>
   </listitem>
   <listitem><para>kdb with a keyboard</para>
   <para><constant>kgdboc=kbd</constant></para>
   </listitem>
   </orderedlist>
   </para>
   </sect3>
   <para>NOTE: Kgdboc does not support interrupting the target via the
   gdb remote protocol.  You must manually send a sysrq-g unless you
   have a proxy that splits console output to a terminal program.
   A console proxy has a separate TCP port for the debugger and a separate
   TCP port for the "human" console.  The proxy can take care of sending
   the sysrq-g for you.
   </para>
   <para>When using kgdboc with no debugger proxy, you can end up
    connecting the debugger at one of two entry points.  If an
    exception occurs after you have loaded kgdboc, a message should
    print on the console stating it is waiting for the debugger.  In
    this case you disconnect your terminal program and then connect the
```

debugger in its place. If you want to interrupt the target system
and forcibly enter a debug session you have to issue a Sysrq
sequence and then type the letter <constant>g</constant>. Then
you disconnect the terminal session and connect gdb. Your options
if you don't like this are to hack gdb to send the sysrq-g for you
as well as on the initial connect, or to use a debugger proxy that
allows an unmodified gdb to do the debugging.
</para>
</sect2>
</sect1>
<sect1 id="kgdbwait">
<title>Kernel parameter: kgdbwait</title>
<para>
The Kernel command line option <constant>kgdbwait</constant> makes
kgdb wait for a debugger connection during booting of a kernel. You
can only use this option you compiled a kgdb I/O driver into the
kernel and you specified the I/O driver configuration as a kernel
command line option. The kgdbwait parameter should always follow the
configuration parameter for the kgdb I/O driver in the kernel
command line else the I/O driver will not be configured prior to
asking the kernel to use it to wait.
</para>
<para>
The kernel will stop and wait as early as the I/O driver and
architecture allows when you use this option. If you build the
kgdb I/O driver as a loadable kernel module kgdbwait will not do
anything.
</para>
</sect1>
<sect1 id="kgdbcon">
<title>Kernel parameter: kgdbcon</title>
<para> The kgdbcon feature allows you to see printk() messages
inside gdb while gdb is connected to the kernel. Kdb does not make
 use of the kgdbcon feature.
</para>
<para>Kgdb supports using the gdb serial protocol to send console
messages to the debugger when the debugger is connected and running.
There are two ways to activate this feature.
<orderedlist>
<listitem><para>Activate with the kernel command line option:</para>
<para><constant>kgdbcon</constant></para>
</listitem>
<listitem><para>Use sysfs before configuring an I/O driver</para>
<para>
<constant>echo 1 > /sys/module/kgdb/parameters/kgdb_use_con</constant>
</para>
<para>
NOTE: If you do this after you configure the kgdb I/O driver, the
setting will not take effect until the next point the I/O is
reconfigured.
</para>
</listitem>
</orderedlist>
<para>IMPORTANT NOTE: You cannot use kgdboc + kgdbcon on a tty that is an
active system console. An example incorrect usage is
<constant>console=ttyS0,115200 kgdboc=ttyS0 kgdbcon</constant>

```
   </para>
   <para>It is possible to use this option with kgdboc on a tty that is not a
system console.
   </para>
  </para>
  </sect1>
  </chapter>
  <chapter id="usingKDB">
  <title>Using kdb</title>
  <para>
  </para>
  <sect1 id="quickKDBserial">
  <title>Quick start for kdb on a serial port</title>
  <para>This is a quick example of how to use kdb.</para>
  <para><orderedlist>
  <listitem><para>Boot kernel with arguments:
  <itemizedlist>
  <listitem><para><constant>console=ttyS0,115200
kgdboc=ttyS0,115200</constant></para></listitem>
  </itemizedlist></para>
  <para>OR</para>
  <para>Configure kgdboc after the kernel booted; assuming you are using a
serial port console:
  <itemizedlist>
  <listitem><para><constant>echo ttyS0 &gt;
/sys/module/kgdboc/parameters/kgdboc</constant></para></listitem>
  </itemizedlist>
  </para>
  </listitem>
  <listitem><para>Enter the kernel debugger manually or by waiting for an oops
or fault.  There are several ways you can enter the kernel debugger manually;
all involve using the sysrq-g, which means you must have enabled
CONFIG_MAGIC_SYSRQ=y in your kernel config.</para>
  <itemizedlist>
  <listitem><para>When logged in as root or with a super user session you can
run:</para>
   <para><constant>echo g &gt; /proc/sysrq-trigger</constant></para></listitem>
  <listitem><para>Example using minicom 2.2</para>
  <para>Press: <constant>Control-a</constant></para>
  <para>Press: <constant>f</constant></para>
  <para>Press: <constant>g</constant></para>
  </listitem>
  <listitem><para>When you have telneted to a terminal server that supports
sending a remote break</para>
  <para>Press: <constant>Control-]</constant></para>
  <para>Type in:<constant>send break</constant></para>
  <para>Press: <constant>Enter</constant></para>
  <para>Press: <constant>g</constant></para>
  </listitem>
  </itemizedlist>
  </listitem>
  <listitem><para>From the kdb prompt you can run the "help" command to see a
complete list of the commands that are available.</para>
  <para>Some useful commands in kdb include:
  <itemizedlist>
  <listitem><para>lsmod  -- Shows where kernel modules are
```

```
loaded</para></listitem>
  <listitem><para>ps -- Displays only the active processes</para></listitem>
  <listitem><para>ps A -- Shows all the processes</para></listitem>
  <listitem><para>summary -- Shows kernel version info and memory
usage</para></listitem>
  <listitem><para>bt -- Get a backtrace of the current process using
dump_stack()</para></listitem>
  <listitem><para>dmesg -- View the kernel syslog buffer</para></listitem>
  <listitem><para>go -- Continue the system</para></listitem>
  </itemizedlist>
  </para>
  </listitem>
  <listitem>
  <para>When you are done using kdb you need to consider rebooting the
  system or using the "go" command to resuming normal kernel
  execution.  If you have paused the kernel for a lengthy period of
  time, applications that rely on timely networking or anything to do
  with real wall clock time could be adversely affected, so you
  should take this into consideration when using the kernel
  debugger.</para>
  </listitem>
  </orderedlist></para>
  </sect1>
  <sect1 id="quickKDBkeyboard">
  <title>Quick start for kdb using a keyboard connected console</title>
  <para>This is a quick example of how to use kdb with a keyboard.</para>
  <para><orderedlist>
  <listitem><para>Boot kernel with arguments:
  <itemizedlist>
  <listitem><para><constant>kgdboc=kbd</constant></para></listitem>
  </itemizedlist></para>
  <para>OR</para>
  <para>Configure kgdboc after the kernel booted:
  <itemizedlist>
  <listitem><para><constant>echo kbd &gt;
/sys/module/kgdboc/parameters/kgdboc</constant></para></listitem>
  </itemizedlist>
  </para>
  </listitem>
  <listitem><para>Enter the kernel debugger manually or by waiting for an oops
or fault.  There are several ways you can enter the kernel debugger manually;
all involve using the sysrq-g, which means you must have enabled
CONFIG_MAGIC_SYSRQ=y in your kernel config.</para>
  <itemizedlist>
  <listitem><para>When logged in as root or with a super user session you can
run:</para>
   <para><constant>echo g &gt; /proc/sysrq-trigger</constant></para></listitem>
  <listitem><para>Example using a laptop keyboard</para>
  <para>Press and hold down: <constant>Alt</constant></para>
  <para>Press and hold down: <constant>Fn</constant></para>
  <para>Press and release the key with the label:
<constant>SysRq</constant></para>
  <para>Release: <constant>Fn</constant></para>
  <para>Press and release: <constant>g</constant></para>
  <para>Release: <constant>Alt</constant></para>
  </listitem>
```

```
  <listitem><para>Example using a PS/2 101-key keyboard</para>
  <para>Press and hold down: <constant>Alt</constant></para>
  <para>Press and release the key with the label:
<constant>SysRq</constant></para>
  <para>Press and release: <constant>g</constant></para>
  <para>Release: <constant>Alt</constant></para>
  </listitem>
  </itemizedlist>
  </listitem>
  <listitem>
  <para>Now type in a kdb command such as "help", "dmesg", "bt" or "go" to
continue kernel execution.</para>
  </listitem>
  </orderedlist></para>
  </sect1>
  </chapter>
  <chapter id="EnableKGDB">
   <title>Using kgdb / gdb</title>
   <para>In order to use kgdb you must activate it by passing
   configuration information to one of the kgdb I/O drivers.  If you
   do not pass any configuration information kgdb will not do anything
   at all.  Kgdb will only actively hook up to the kernel trap hooks
   if a kgdb I/O driver is loaded and configured.  If you unconfigure
   a kgdb I/O driver, kgdb will unregister all the kernel hook points.
   </para>
   <para> All kgdb I/O drivers can be reconfigured at run time, if
   <symbol>CONFIG_SYSFS</symbol> and <symbol>CONFIG_MODULES</symbol>
   are enabled, by echo'ing a new config string to
   <constant>/sys/module/&lt;driver&gt;/parameter/&lt;option&gt;</constant>.
   The driver can be unconfigured by passing an empty string.  You cannot
   change the configuration while the debugger is attached.  Make sure
   to detach the debugger with the <constant>detach</constant> command
   prior to trying to unconfigure a kgdb I/O driver.
   </para>
  <sect1 id="ConnectingGDB">
  <title>Connecting with gdb to a serial port</title>
  <orderedlist>
  <listitem><para>Configure kgdboc</para>
   <para>Boot kernel with arguments:
   <itemizedlist>
    <listitem><para><constant>kgdboc=ttyS0,115200</constant></para></listitem>
   </itemizedlist></para>
   <para>OR</para>
   <para>Configure kgdboc after the kernel booted:
   <itemizedlist>
    <listitem><para><constant>echo ttyS0 &gt;
/sys/module/kgdboc/parameters/kgdboc</constant></para></listitem>
   </itemizedlist></para>
  </listitem>
  <listitem>
  <para>Stop kernel execution (break into the debugger)</para>
  <para>In order to connect to gdb via kgdboc, the kernel must
  first be stopped.  There are several ways to stop the kernel which
  include using kgdbwait as a boot argument, via a sysrq-g, or running
  the kernel until it takes an exception where it waits for the
  debugger to attach.
```

```
  <itemizedlist>
  <listitem><para>When logged in as root or with a super user session you can
run:</para>
   <para><constant>echo g &gt; /proc/sysrq-trigger</constant></para></listitem>
  <listitem><para>Example using minicom 2.2</para>
  <para>Press: <constant>Control-a</constant></para>
  <para>Press: <constant>f</constant></para>
  <para>Press: <constant>g</constant></para>
  </listitem>
  <listitem><para>When you have telneted to a terminal server that supports
sending a remote break</para>
  <para>Press: <constant>Control-]</constant></para>
  <para>Type in:<constant>send break</constant></para>
  <para>Press: <constant>Enter</constant></para>
  <para>Press: <constant>g</constant></para>
  </listitem>
  </itemizedlist>
  </para>
  </listitem>
  <listitem>
    <para>Connect from from gdb</para>
    <para>
    Example (using a directly connected port):
    </para>
    <programlisting>
    % gdb ./vmlinux
    (gdb) set remotebaud 115200
    (gdb) target remote /dev/ttyS0
    </programlisting>
    <para>
    Example (kgdb to a terminal server on TCP port 2012):
    </para>
    <programlisting>
    % gdb ./vmlinux
    (gdb) target remote 192.168.2.2:2012
    </programlisting>
    <para>
    Once connected, you can debug a kernel the way you would debug an
    application program.
    </para>
    <para>
    If you are having problems connecting or something is going
    seriously wrong while debugging, it will most often be the case
    that you want to enable gdb to be verbose about its target
    communications.  You do this prior to issuing the <constant>target
    remote</constant> command by typing in: <constant>set debug remote
1</constant>
    </para>
  </listitem>
  </orderedlist>
  <para>Remember if you continue in gdb, and need to "break in" again,
  you need to issue an other sysrq-g.  It is easy to create a simple
  entry point by putting a breakpoint at <constant>sys_sync</constant>
  and then you can run "sync" from a shell or script to break into the
  debugger.</para>
  </sect1>
```

```
  </chapter>
  <chapter id="switchKdbKgdb">
  <title>kgdb and kdb interoperability</title>
  <para>It is possible to transition between kdb and kgdb dynamically.
  The debug core will remember which you used the last time and
  automatically start in the same mode.</para>
  <sect1>
  <title>Switching between kdb and kgdb</title>
  <sect2>
  <title>Switching from kgdb to kdb</title>
  <para>
  There are two ways to switch from kgdb to kdb: you can use gdb to
  issue a maintenance packet, or you can blindly type the command $3#33.
  Whenever kernel debugger stops in kgdb mode it will print the
  message <constant>KGDB or $3#33 for KDB</constant>.  It is important
  to note that you have to type the sequence correctly in one pass.
  You cannot type a backspace or delete because kgdb will interpret
  that as part of the debug stream.
  <orderedlist>
  <listitem><para>Change from kgdb to kdb by blindly typing:</para>
  <para><constant>$3#33</constant></para></listitem>
  <listitem><para>Change from kgdb to kdb with gdb</para>
  <para><constant>maintenance packet 3</constant></para>
  <para>NOTE: Now you must kill gdb. Typically you press control-z and
  issue the command: kill -9 %</para></listitem>
  </orderedlist>
  </para>
  </sect2>
  <sect2>
  <title>Change from kdb to kgdb</title>
  <para>There are two ways you can change from kdb to kgdb.  You can
  manually enter kgdb mode by issuing the kgdb command from the kdb
  shell prompt, or you can connect gdb while the kdb shell prompt is
  active.  The kdb shell looks for the typical first commands that gdb
  would issue with the gdb remote protocol and if it sees one of those
  commands it automatically changes into kgdb mode.</para>
  <orderedlist>
  <listitem><para>From kdb issue the command:</para>
  <para><constant>kgdb</constant></para>
  <para>Now disconnect your terminal program and connect gdb in its
place</para></listitem>
  <listitem><para>At the kdb prompt, disconnect the terminal program and connect
gdb in its place.</para></listitem>
  </orderedlist>
  </sect2>
  </sect1>
  <sect1>
  <title>Running kdb commands from gdb</title>
  <para>It is possible to run a limited set of kdb commands from gdb,
  using the gdb monitor command.  You don't want to execute any of the
  run control or breakpoint operations, because it can disrupt the
  state of the kernel debugger.  You should be using gdb for
  breakpoints and run control operations if you have gdb connected.
  The more useful commands to run are things like lsmod, dmesg, ps or
  possibly some of the memory information commands.  To see all the kdb
  commands you can run <constant>monitor help</constant>.</para>
```

```
  <para>Example:
  <informalexample><programlisting>
(gdb) monitor ps
1 idle process (state I) and
27 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command

0xc78291d0        1        0  0    0   S  0xc7829404  init
0xc7954150      942        1  0    0   S  0xc7954384  dropbear
0xc78789c0      944        1  0    0   S  0xc7878bf4  sh
(gdb)
  </programlisting></informalexample>
  </para>
  </sect1>
  </chapter>
  <chapter id="KGDBTestSuite">
    <title>kgdb Test Suite</title>
    <para>
    When kgdb is enabled in the kernel config you can also elect to
    enable the config parameter KGDB_TESTS.  Turning this on will
    enable a special kgdb I/O module which is designed to test the
    kgdb internal functions.
    </para>
    <para>
    The kgdb tests are mainly intended for developers to test the kgdb
    internals as well as a tool for developing a new kgdb architecture
    specific implementation.  These tests are not really for end users
    of the Linux kernel.  The primary source of documentation would be
    to look in the drivers/misc/kgdbts.c file.
    </para>
    <para>
    The kgdb test suite can also be configured at compile time to run
    the core set of tests by setting the kernel config parameter
    KGDB_TESTS_ON_BOOT.  This particular option is aimed at automated
    regression testing and does not require modifying the kernel boot
    config arguments.  If this is turned on, the kgdb test suite can
    be disabled by specifying "kgdbts=" as a kernel boot argument.
    </para>
  </chapter>
  <chapter id="CommonBackEndReq">
  <title>Kernel Debugger Internals</title>
  <sect1 id="kgdbArchitecture">
    <title>Architecture Specifics</title>
      <para>
      The kernel debugger is organized into a number of components:
      <orderedlist>
      <listitem><para>The debug core</para>
      <para>
      The debug core is found in kernel/debugger/debug_core.c.  It contains:
      <itemizedlist>
      <listitem><para>A generic OS exception handler which includes
      sync'ing the processors into a stopped state on an multi-CPU
      system.</para></listitem>
      <listitem><para>The API to talk to the kgdb I/O drivers</para></listitem>
      <listitem><para>The API to make calls to the arch-specific kgdb
```

```
implementation</para></listitem>
      <listitem><para>The logic to perform safe memory reads and writes to
memory while using the debugger</para></listitem>
      <listitem><para>A full implementation for software breakpoints unless
overridden by the arch</para></listitem>
      <listitem><para>The API to invoke either the kdb or kgdb frontend to the
debug core.</para></listitem>
      </itemizedlist>
      </para>
      </listitem>
      <listitem><para>kgdb arch-specific implementation</para>
      <para>
      This implementation is generally found in arch/*/kernel/kgdb.c.
      As an example, arch/x86/kernel/kgdb.c contains the specifics to
      implement HW breakpoint as well as the initialization to
      dynamically register and unregister for the trap handlers on
      this architecture.  The arch-specific portion implements:
      <itemizedlist>
      <listitem><para>contains an arch-specific trap catcher which
      invokes kgdb_handle_exception() to start kgdb about doing its
      work</para></listitem>
      <listitem><para>translation to and from gdb specific packet format to
pt_regs</para></listitem>
      <listitem><para>Registration and unregistration of architecture specific
trap hooks</para></listitem>
      <listitem><para>Any special exception handling and
cleanup</para></listitem>
      <listitem><para>NMI exception handling and cleanup</para></listitem>
      <listitem><para>(optional)HW breakpoints</para></listitem>
      </itemizedlist>
      </para>
      </listitem>
      <listitem><para>gdbstub frontend (aka kgdb)</para>
      <para>The gdbstub is located in kernel/debug/gdbstub.c. It
contains:</para>
      <itemizedlist>
        <listitem><para>All the logic to implement the gdb serial
protocol</para></listitem>
      </itemizedlist>
      </listitem>
      <listitem><para>kdb frontend</para>
      <para>The kdb debugger shell is broken down into a number of
      components.  The kdb core is located in kernel/debug/kdb.  There
      are a number of helper functions in some of the other kernel
      components to make it possible for kdb to examine and report
      information about the kernel without taking locks that could
      cause a kernel deadlock.  The kdb core contains implements the following
functionality.</para>
      <itemizedlist>
        <listitem><para>A simple shell</para></listitem>
        <listitem><para>The kdb core command set</para></listitem>
        <listitem><para>A registration API to register additional kdb shell
commands.</para>
        <para>A good example of a self-contained kdb module is the "ftdump"
command for dumping the ftrace buffer.  See:
kernel/trace/trace_kdb.c</para></listitem>
```

```
        <listitem><para>The implementation for kdb_printf() which
        emits messages directly to I/O drivers, bypassing the kernel
        log.</para></listitem>
        <listitem><para>SW / HW breakpoint management for the kdb
shell</para></listitem>
      </itemizedlist>
      </listitem>
      <listitem><para>kgdb I/O driver</para>
      <para>
      Each kgdb I/O driver has to provide an implementation for the following:
      <itemizedlist>
      <listitem><para>configuration via built-in or module</para></listitem>
      <listitem><para>dynamic configuration and kgdb hook registration
calls</para></listitem>
      <listitem><para>read and write character interface</para></listitem>
      <listitem><para>A cleanup handler for unconfiguring from the kgdb
core</para></listitem>
      <listitem><para>(optional) Early debug methodology</para></listitem>
      </itemizedlist>
      Any given kgdb I/O driver has to operate very closely with the
      hardware and must do it in such a way that does not enable
      interrupts or change other parts of the system context without
      completely restoring them. The kgdb core will repeatedly "poll"
      a kgdb I/O driver for characters when it needs input.  The I/O
      driver is expected to return immediately if there is no data
      available.  Doing so allows for the future possibility to touch
      watch dog hardware in such a way as to have a target system not
      reset when these are enabled.
      </para>
      </listitem>
      </orderedlist>
      </para>
      <para>
      If you are intent on adding kgdb architecture specific support
      for a new architecture, the architecture should define
      <constant>HAVE_ARCH_KGDB</constant> in the architecture specific
      Kconfig file.  This will enable kgdb for the architecture, and
      at that point you must create an architecture specific kgdb
      implementation.
      </para>
      <para>
      There are a few flags which must be set on every architecture in
      their &lt;asm/kgdb.h&gt; file.  These are:
      <itemizedlist>
        <listitem>
          <para>
          NUMREGBYTES: The size in bytes of all of the registers, so
          that we can ensure they will all fit into a packet.
          </para>
          <para>
          BUFMAX: The size in bytes of the buffer GDB will read into.
          This must be larger than NUMREGBYTES.
          </para>
          <para>
          CACHE_FLUSH_IS_SAFE: Set to 1 if it is always safe to call
          flush_cache_range or flush_icache_range.  On some architectures,
```

```
          these functions may not be safe to call on SMP since we keep other
          CPUs in a holding pattern.
          </para>
        </listitem>
      </itemizedlist>
      </para>
      <para>
      There are also the following functions for the common backend,
      found in kernel/kgdb.c, that must be supplied by the
      architecture-specific backend unless marked as (optional), in
      which case a default function maybe used if the architecture
      does not need to provide a specific implementation.
      </para>
!Iinclude/linux/kgdb.h
  </sect1>
  <sect1 id="kgdbocDesign">
  <title>kgdboc internals</title>
  <para>
  The kgdboc driver is actually a very thin driver that relies on the
  underlying low level to the hardware driver having "polling hooks"
  which the to which the tty driver is attached.  In the initial
  implementation of kgdboc it the serial_core was changed to expose a
  low level UART hook for doing polled mode reading and writing of a
  single character while in an atomic context.  When kgdb makes an I/O
  request to the debugger, kgdboc invokes a call back in the serial
  core which in turn uses the call back in the UART driver.  It is
  certainly possible to extend kgdboc to work with non-UART based
  consoles in the future.
  </para>
  <para>
  When using kgdboc with a UART, the UART driver must implement two callbacks in
the <constant>struct uart_ops</constant>. Example from
drivers/8250.c:<programlisting>
#ifdef CONFIG_CONSOLE_POLL
        .poll_get_char = serial8250_get_poll_char,
        .poll_put_char = serial8250_put_poll_char,
#endif
  </programlisting>
  Any implementation specifics around creating a polling driver use the
  <constant>#ifdef CONFIG_CONSOLE_POLL</constant>, as shown above.
  Keep in mind that polling hooks have to be implemented in such a way
  that they can be called from an atomic context and have to restore
  the state of the UART chip on return such that the system can return
  to normal when the debugger detaches.  You need to be very careful
  with any kind of lock you consider, because failing here is most
  going to mean pressing the reset button.
  </para>
  </sect1>
  </chapter>
  <chapter id="credits">
     <title>Credits</title>
        <para>
                The following people have contributed to this document:
                <orderedlist>
                        <listitem><para>Amit
Kale<email>amitkale@linsyssoft.com</email></para></listitem>
```

```
                        <listitem><para>Tom
Rini<email>trini@kernel.crashing.org</email></para></listitem>
                </orderedlist>
                In March 2008 this document was completely rewritten by:
                <itemizedlist>
                <listitem><para>Jason
Wessel<email>jason.wessel@windriver.com</email></para></listitem>
                </itemizedlist>
                In Jan 2010 this document was updated to include kdb.
                <itemizedlist>
                <listitem><para>Jason
Wessel<email>jason.wessel@windriver.com</email></para></listitem>
                </itemizedlist>
        </para>
  </chapter>
</book>
```