

Using kgdb, kdb and the kernel debugger internals

Jason Wessel <jason.wessel@windriver.com>

Using kgdb, kdb and the kernel debugger internals

by Jason Wessel

Copyright © 2008,2010 Wind River Systems, Inc.

Copyright © 2004-2005 MontaVista Software, Inc.

Copyright © 2004 Amit S. Kale

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.

Chapter 1. Introduction

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.

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 CONFIG_KALLSYMS.

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.

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 a 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.

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.

Here is an example set of .config symbols to enable/disable kdb:

- # CONFIG_DEBUG_RODATA is not set
- CONFIG_FRAME_POINTER=y
- CONFIG_KGDB=y
- CONFIG_KGDB_SERIAL_CONSOLE=y
- CONFIG_KGDB_KDB=y
- CONFIG_KDB_KEYBOARD=y

Enter the debugger on reboot notify.

Release: `Alt`

3. Now type in a kdb command such as "help", "dmesg", "bt" or "go" to continue kernel execution.

Example (using a directly connected port):

```
% gdb ./vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
```

Example (kgdb to a terminal server on TCP port 2012):

```
% gdb ./vmlinux
(gdb) target remote 192.168.2.2:2012
```

Once connected, you can debug a kernel the way you would debug an application program.

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 `target remote` command by typing in: `set debug remote 1`

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 `sys_sync` and then you can run "sync" from a shell or script to break into the debugger.


```
(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)
```

Chapter 7. kgdb Test Suite

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.

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.

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.

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.

Name

kgdb_skipexception — (optional) exit kgdb_handle_exception early

Synopsis

```
int kgdb_skipexception (int exception, struct pt_regs * regs);
```

Arguments

exception Exception vector number

regs Current struct pt_regs.

Description

On some architectures it is required to skip a breakpoint exception when it occurs after a breakpoint has been removed. This can be implemented in the architecture specific portion of kgdb.

Name

kgdb_breakpoint — compiled in breakpoint

Synopsis

```
void kgdb_breakpoint ( void );
```

Arguments

void no arguments

Description

This will be implemented as a static inline per architecture. This function is called by the kgdb core to execute an architecture specific trap to cause kgdb to enter the exception processing.

Name

`kgdb_arch_init` — Perform any architecture specific initialization.

Synopsis

```
int kgdb_arch_init ( void );
```

Arguments

void no arguments

Description

This function will handle the initialization of any architecture specific callbacks.

Name

`kgdb_arch_exit` — Perform any architecture specific uninitialization.

Synopsis

```
void kgdb_arch_exit ( void );
```

Arguments

void no arguments

Description

This function will handle the uninitialization of any architecture specific callbacks, for dynamic registration and unregistration.

Name

`pt_regs_to_gdb_regs` — Convert ptrace regs to GDB regs

Synopsis

```
void pt_regs_to_gdb_regs (unsigned long * gdb_regs, struct pt_regs *  
regs);
```

Arguments

gdb_regs A pointer to hold the registers in the order GDB wants.

regs The struct `pt_regs` of the current process.

Description

Convert the `pt_regs` in *regs* into the format for registers that GDB expects, stored in *gdb_regs*.

Name

`gdb_regs_to_pt_regs` — Convert GDB regs to ptrace regs.

Synopsis

```
void gdb_regs_to_pt_regs (unsigned long * gdb_regs, struct pt_regs *  
regs);
```

Arguments

gdb_regs A pointer to hold the registers we've received from GDB.

regs A pointer to a struct `pt_regs` to hold these values in.

Description

Convert the GDB regs in *gdb_regs* into the `pt_regs`, and store them in *regs*.

Name

`kgdb_roundup_cpus` — Get other CPUs into a holding pattern

Synopsis

```
void kgdb_roundup_cpus (unsigned long flags);
```

Arguments

flags Current IRQ state

Description

On SMP systems, we need to get the attention of the other CPUs and get them into a known state. This should do what is needed to get the other CPUs to call `kgdb_wait`. Note that on some arches, the NMI approach is not used for rounding up all the CPUs. For example, in case of MIPS, `smp_call_function` is used to roundup CPUs. In this case, we have to make sure that interrupts are enabled before calling `smp_call_function`. The argument to this function is the flags that will be used when restoring the interrupts. There is `local_irq_save` call before `kgdb_roundup_cpus`.

On non-SMP systems, this is not called.

Name

`kgdb_arch_set_pc` — Generic call back to the program counter

Synopsis

```
void kgdb_arch_set_pc (struct pt_regs * regs, unsigned long pc);
```

Arguments

regs Current struct pt_regs.

pc The new value for the program counter

Description

This function handles updating the program counter and requires an architecture specific implementation.

Name

`kgdb_arch_late` — Perform any architecture specific initialization.

Synopsis

```
void kgdb_arch_late ( void );
```

Arguments

void no arguments

Description

This function will handle the late initialization of any architecture specific callbacks. This is an optional function for handling things like late initialization of hw breakpoints. The default implementation does nothing.

Chapter 9. Credits

The following people have contributed to this document:

1. Amit Kale<amitkale@linsyssoft.com>
2. Tom Rini<trini@kernel.crashing.org>

In March 2008 this document was completely rewritten by:

- Jason Wessel<jason.wessel@windriver.com>

In Jan 2010 this document was updated to include kdb.

- Jason Wessel<jason.wessel@windriver.com>