# Using kgdb and the kgdb Internals


## Jason Wessel

<jason.wessel@windriver.com>

## Tom Rini

<trini@kernel.crashing.org>

## Amit S. Kale

<amitkale@linsyssoft.com>




---

**Table of Contents**


1. Introduction
2. Compiling a kernel
3. Enable kgdb for debugging
    - Kernel parameter: kgdbwait
    - Kernel parameter: kgdboc
        - Using kgdboc
    - Kernel parameter: kgdbcon
4. Connecting gdb
5. kgdb Test Suite
6. KGDB Internals
    - Architecture Specifics
    - kgdboc internals

7. Credits

# Chapter 1. Introduction

kgdb is a source level debugger for 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 what an application developer would use gdb for. 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 a test machine. The kernel to be debugged runs on the test 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 builtin's or kernel modules in the test machine's kernel.

# Chapter 2. Compiling a kernel

To enable CONFIG_KGDB 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 debugging with remote gdb".

It is advised, but not required that you turn on the CONFIG_FRAME_POINTER kernel option. This option inserts code to into the compiled executable which saves the frame information in registers or on the stack at different points which will allow a debugger such as gdb to more accurately construct stack back traces while debugging the kernel.

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.

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, see following chapter.

The kgdb test compile options are described in the kgdb test suite chapter.

# Chapter 3. Enable kgdb for debugging

**Table of Contents**

Kernel parameter: kgdbwait
Kernel parameter: kgdboc

Using kgdboc

Kernel parameter: kgdbcon

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.

All drivers can be reconfigured at run time, if CONFIG_SYSFS and CONFIG_MODULES are enabled, by echo'ing a new config string to `/sys/module/<driver>/parameter/<option>`. 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 `detach` command prior to trying unconfigure a kgdb I/O driver.

# Kernel parameter: kgdbwait

The Kernel command line option `kgdbwait` 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.

The kernel will stop and wait as early as the I/O driver and architecture will allow when you use this option. If you build the kgdb I/O driver as a kernel module kgdbwait will not do anything.

# Kernel parameter: kgdboc

The kgdboc driver was originally an abbreviation meant to stand for "kgdb over console". Kgdboc is designed to work with a single serial port. It was meant to cover the circumstance where you wanted to use a serial console as your primary console as well as using it to perform kernel debugging. Of course you can also use kgdboc without assigning a console to the same port.

## Using kgdboc

You can configure kgdboc via sysfs or a module or kernel boot line parameter depending on if you build with CONFIG_KGDBOC as a module or built-in.

1. From the module load or build-in

   ```
   kgdboc=<tty-device>,[baud]
   ```

   The example here would be if your console port was typically ttyS0, you would use something like `kgdboc=ttyS0,115200` or on the ARM Versatile AB you would likely use `kgdboc=ttyAMA0,115200`

2. From sysfs

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

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 problem and has a separate port for the debugger to connect to that sends the sysrq-g for you.

When using kgdboc with no debugger proxy, you can end up connecting the debugger for 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 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 `g`. 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.

# Kernel parameter: kgdbcon

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.

1. Activate with the kernel command line option:

   ```
   kgdbcon
   ```

2. Use sysfs before configuring an io driver

   ```
   echo 1 > /sys/module/kgdb/parameters/kgdb_use_con
   ```

   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.

IMPORTANT NOTE: Using this option with kgdb over the console (kgdboc) is not supported.

# Chapter 4. Connecting gdb

If you are using kgdboc, you need to have used kgdbwait as a boot argument, issued a sysrq-g, or the system you are going to debug has already taken an exception and is waiting for the debugger to attach before you can connect gdb.

If you are not using different kgdb I/O driver other than kgdboc, you should be able to connect and the target will automatically respond.

Example (using a serial 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`

# Chapter 5. 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.

# Chapter 6. KGDB Internals

**Table of Contents**

Architecture Specifics
kgdboc internals

# Architecture Specifics

Kgdb is organized into three basic components:

1. kgdb core

   The kgdb core is found in kernel/kgdb.c. It contains:

   - All the logic to implement the gdb serial protocol
   - A generic OS exception handler which includes sync'ing the processors into a stopped state on an multi cpu system.
   - The API to talk to the kgdb I/O drivers
   - The API to make calls to the arch specific kgdb implementation
   - The logic to perform safe memory reads and writes to memory while using the debugger
   - A full implementation for software breakpoints unless overridden by the arch

2. kgdb arch specific implementation

   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:

- contains an arch specific trap catcher which invokes kgdb_handle_exception() to start kgdb about doing its work
- translation to and from gdb specific packet format to pt_regs
- Registration and unregistration of architecture specific trap hooks
- Any special exception handling and cleanup
- NMI exception handling and cleanup
- (optional)HW breakpoints

3. kgdb I/O driver

   Each kgdb I/O driver has to provide an implemenation for the following:

   - configuration via builtin or module
   - dynamic configuration and kgdb hook registration calls
   - read and write character interface
   - A cleanup handler for unconfiguring from the kgdb core
   - (optional) Early debug methodology

   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.

If you are intent on adding kgdb architecture specific support for a new architecture, the architecture should define `HAVE_ARCH_KGDB` 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.

There are a few flags which must be set on every architecture in their <asm/kgdb.h> file. These are:

- NUMREGBYTES: The size in bytes of all of the registers, so that we can ensure they will all fit into a packet.

  BUFMAX: The size in bytes of the buffer GDB will read into. This must be larger than NUMREGBYTES.

  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.

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 (exception,
                        regs);

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 for kgdb.

---

# Name

kgdb_post_primary_code — (optional) Save error vector/code numbers.

# Synopsis

```
void kgdb_post_primary_code (regs,
                             e_vector,
                             err_code);

struct pt_regs *  regs;
int               e_vector;
int               err_code;
```

# Arguments

*regs*

Original pt_regs.

*e_vector*

Original error vector.

*err_code*

Original error code.

# Description

This is usually needed on architectures which support SMP and KGDB. This function is called after all the secondary cpus have been put to a know spin state and the primary CPU has control over KGDB.

---

# Name

kgdb_disable_hw_debug — (optional) Disable hardware debugging hook

# Synopsis

```
void kgdb_disable_hw_debug (regs);

struct pt_regs *  regs;
```

# Arguments

*regs*

Current struct pt_regs.

# Description

This function will be called if the particular architecture must disable hardware debugging while it is processing gdb packets or handling exception.

---

# Name

kgdb_breakpoint — compiled in breakpoint

# Synopsis

```
void kgdb_breakpoint (void);

 void;
```

# Arguments

*void*

no arguments

# Description

This will be impelmented 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 initalization.

# Synopsis

```
int kgdb_arch_init (void);

 void;
```

# Arguments

*void*

no arguments

# Description

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

---

# Name

kgdb_arch_exit — Perform any architecture specific uninitalization.

# Synopsis

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

```
void;
```

# Arguments

*void*

no arguments

# Description

This function will handle the uninitalization 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 (gdb_regs,
                          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

sleeping_thread_to_gdb_regs — Convert ptrace regs to GDB regs

# Synopsis

```
void sleeping_thread_to_gdb_regs (gdb_regs,
                                  p);

unsigned long *       gdb_regs;
struct task_struct *  p;
```

# Arguments

*gdb_regs*

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

*p*

The struct task_struct of the desired process.

# Description

Convert the register values of the sleeping process in *p* to the format that GDB expects. This function is called when kgdb does not have access to the struct pt_regs and therefore it should fill the gdb registers *gdb_regs* with what has been saved in struct thread_struct thread field during switch_to.

---

# Name

gdb_regs_to_pt_regs — Convert GDB regs to ptrace regs.

# Synopsis

```
void gdb_regs_to_pt_regs (gdb_regs,
                          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_arch_handle_exception — Handle architecture specific GDB packets.

# Synopsis

```
int kgdb_arch_handle_exception (vector,
                                signo,
                                err_code,
                                remcom_in_buffer,
                                remcom_out_buffer,
                                regs);

int               vector;
int               signo;
int               err_code;
char *            remcom_in_buffer;
char *            remcom_out_buffer;
struct pt_regs *  regs;
```

# Arguments

*vector*

The error vector of the exception that happened.

*signo*

The signal number of the exception that happened.

*err_code*

The error code of the exception that happened.

*remcom_in_buffer*

The buffer of the packet we have read.

*remcom_out_buffer*

The buffer of BUFMAX bytes to write a packet into.

*regs*

The struct pt_regs of the current process.

# Description

This function MUST handle the 'c' and 's' command packets, as well packets to set / remove a hardware

breakpoint, if used. If there are additional packets which the hardware needs to handle, they are handled here. The code should return -1 if it wants to process more packets, and a `0` or `1` if it wants to exit from the kgdb callback.

---

# Name

kgdb_roundup_cpus — Get other CPUs into a holding pattern

# Synopsis

`void` **`kgdb_roundup_cpus`** (*flags*);

`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 be in 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

struct kgdb_arch — Describe architecture specific values.

# Synopsis

```
struct kgdb_arch {
  unsigned char gdb_bpt_instr[BREAK_INSTR_SIZE];
  unsigned long flags;
  int (* set_breakpoint) (unsigned long, char *);
  int (* remove_breakpoint) (unsigned long, char *);
  int (* set_hw_breakpoint) (unsigned long, int, enum kgdb_bptype);
  int (* remove_hw_breakpoint) (unsigned long, int, enum kgdb_bptype);
  void (* remove_all_hw_break) (void);
  void (* correct_hw_break) (void);
};
```

# Members

gdb_bpt_instr[BREAK_INSTR_SIZE]

The instruction to trigger a breakpoint.

flags

Flags for the breakpoint, currently just `KGDB_HW_BREAKPOINT`.

set_breakpoint

Allow an architecture to specify how to set a software breakpoint.

remove_breakpoint

Allow an architecture to specify how to remove a software breakpoint.

set_hw_breakpoint

Allow an architecture to specify how to set a hardware breakpoint.

remove_hw_breakpoint

Allow an architecture to specify how to remove a hardware breakpoint.

remove_all_hw_break

Allow an architecture to specify how to remove all hardware breakpoints.

correct_hw_break

Allow an architecture to specify how to correct the hardware debug registers.

---

# Name

struct kgdb_io — Describe the interface for an I/O driver to talk with KGDB.

# Synopsis

```
struct kgdb_io {
  const char * name;
  int (* read_char) (void);
  void (* write_char) (u8);
  void (* flush) (void);
  int (* init) (void);
  void (* pre_exception) (void);
  void (* post_exception) (void);
};
```

# Members

name

Name of the I/O driver.

read_char

Pointer to a function that will return one char.

write_char

Pointer to a function that will write one char.

flush

Pointer to a function that will flush any pending writes.

init

Pointer to a function that will initialize the device.

pre_exception

Pointer to a function that will do any prep work for the I/O driver.

post_exception

Pointer to a function that will do any cleanup work for the I/O driver.

# kgdboc internals

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.

When using kgdboc with a uart, the uart driver must implement two callbacks in the `struct uart_ops`. Example from drivers/8250.c:

```
#ifdef CONFIG_CONSOLE_POLL
        .poll_get_char = serial8250_get_poll_char,
        .poll_put_char = serial8250_put_poll_char,
#endif
```

Any implementation specifics around creating a polling driver use the `#ifdef CONFIG_CONSOLE_POLL`, 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.

# Chapter 7. 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>