Using kgdb, kdb and the kernel debugger internals

kernel.org/doc/html/v4.15/dev-tools/kgdb.html

Author: Jason Wessel

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.

Compiling a kernel

- In order to enable compilation of kdb, you must first enable kgdb.
- The kgdb test compile options are described in the kgdb test suite chapter.

Kernel config options for kgdb

To enable `CONFIG_KGDB` you should look under Kernel hacking ▸ Kernel debugging and select KGDB: kernel debugger.

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 `CONFIG_DEBUG_INFO` which is called Compile the kernel with debug info in the config menu.

It is advised, but not required, that you turn on the `CONFIG_FRAME_POINTER` 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.

If the architecture that you are using supports the kernel option `CONFIG_STRICT_KERNEL_RWX` , 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_STRICT_KERNEL_RWX` 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 which you can learn more about in the in the section that describes the parameter kgdboc.

Here is an example set of `.config` symbols to enable or disable for kgdb:

```
# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
```

Kernel config options for kdb

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.

The main config option for kdb is `CONFIG_KGDB_KDB` 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.

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 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_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

Kernel Debugger Boot Arguments

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 providing some examples of the configuration parameters.

Kernel parameter: kgdboc

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.

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 `kgdbwait` and early debugging if you build kgdboc into the kernel as a built-in.

Optionally you can elect to activate kms (Kernel Mode Setting) integration. When you use kms with kgdboc and you have a video driver that has atomic mode setting hooks, it is possible to enter the debugger on the graphics console. When the kernel execution is resumed, the previous graphics mode will be restored. This integration can serve as a useful tool to aid in diagnosing crashes or doing analysis of memory with kdb while allowing the full graphics console applications to run.

kgdboc arguments

Usage:

```
kgdboc=[kms][[,]kbd][[,]serial_device][,baud]
```

The order listed above must be observed if you use any of the optional configurations together.

Abbreviations:

- kms = Kernel Mode Setting
- kbd = Keyboard

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. The order listed above must be observed if you use any of the optional configurations together. Using kms + only gdb is generally not a useful combination.

Using loadable module or built-in

1. As a kernel built-in:

   Use the kernel boot argument:

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

2. As a kernel loadable module:

   Use the command:

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

   Here are two examples of how you might format 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.

   1. `kgdboc=ttyS0,115200`
   2. `kgdboc=ttyAMA1,115200`

Configure kgdboc at runtime with sysfs

At run time you can enable or disable kgdboc by echoing a parameters into the sysfs. Here are two examples:

1. Enable kgdboc on ttyS0:

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

2. Disable kgdboc:

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

Note

You do not need to specify the baud if you are configuring the console on tty which is already configured or open.

More examples

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.

1. kdb and kgdb over only a serial port:

   ```
   kgdboc=<serial_device>[,baud]
   ```

   Example:

   ```
   kgdboc=ttyS0,115200
   ```

2. kdb and kgdb with keyboard and a serial port:

   ```
   kgdboc=kbd,<serial_device>[,baud]
   ```

   Example:

```
    kgdboc=kbd,ttyS0,115200
    ```

3. kdb with a keyboard:

    ```
    kgdboc=kbd
    ```

4. kdb with kernel mode setting:

    ```
    kgdboc=kms,kbd
    ```

5. kdb with kernel mode setting and kgdb over a serial port:

    ```
    kgdboc=kms,kbd,ttyS0,115200
    ```

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.

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 `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: `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 if 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 allows when you use this option. If you build the kgdb I/O driver as a loadable kernel module kgdbwait will not do anything.

Kernel parameter: `kgdbcon`

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.

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 I/O 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

You cannot use kgdboc + kgdbcon on a tty that is an active system console. An example of incorrect usage is:

```
console=ttyS0,115200 kgdboc=ttyS0 kgdbcon
```

It is possible to use this option with kgdboc on a tty that is not a system console.

Run time parameter: `kgdbreboot`

The kgdbreboot feature allows you to change how the debugger deals with the reboot notification. You have 3 choices for the behavior. The default behavior is always set to 0.

1	`echo -1 > /sys/module/debug_core/parameters/kgdbreboot`	Ignore the reboot notification entirely.
2	`echo 0 > /sys/module/debug_core/parameters/kgdbreboot`	Send the detach message to any attached debugger client.
3	`echo 1 > /sys/module/debug_core/parameters/kgdbreboot`	Enter the debugger on reboot notify.

Kernel parameter: `nokaslr`

If the architecture that you are using enable KASLR by default, you should consider turning it off. KASLR randomizes the virtual address where the kernel image is mapped and confuse gdb which resolve kernel symbol address from symbol table of vmlinux.

Using kdb

Quick start for kdb on a serial port

This is a quick example of how to use kdb.

1. Configure kgdboc at boot using kernel parameters:

   ```
   console=ttyS0,115200 kgdboc=ttyS0,115200 nokaslr
   ```

   OR

   Configure kgdboc after the kernel has booted; assuming you are using a serial port console:

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

2. 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.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: `CTRL-A` `f` `g`

   - When you have telneted to a terminal server that supports sending a remote break

     Press: `CTRL-]`

     Type in: `send break`

     Press: `Enter` `g`

3. From the kdb prompt you can run the `help` command to see a complete list of the commands that are available.

   Some useful commands in kdb include:

`lsmod`	Shows where kernel modules are loaded
`ps`	Displays only the active processes
`ps A`	Shows all the processes
`summary`	Shows kernel version info and memory usage
`bt`	Get a backtrace of the current process using `dump_stack()`
`dmesg`	View the kernel syslog buffer
`go`	Continue the system

4. 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.

Quick start for kdb using a keyboard connected console

This is a quick example of how to use kdb with a keyboard.

1. Configure kgdboc at boot using kernel parameters:

   ```
   kgdboc=kbd
   ```

   OR

   Configure kgdboc after the kernel has booted:

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

2. 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.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using a laptop keyboard:

     Press and hold down: `Alt`

     Press and hold down: `Fn`

     Press and release the key with the label: `SysRq`

     Release: `Fn`

     Press and release: `g`

     Release: `Alt`

   - Example using a PS/2 101-key keyboard

     Press and hold down: `Alt`

     Press and release the key with the label: `SysRq`

     Press and release: `g`

     Release: `Alt`

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

Using kgdb / gdb

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 kgdb I/O 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 to unconfigure a kgdb I/O driver.

Connecting with gdb to a serial port

1. Configure kgdboc

   Configure kgdboc at boot using kernel parameters:

   ```
   kgdboc=ttyS0,115200
   ```

   OR

   Configure kgdboc after the kernel has booted:

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

2. Stop kernel execution (break into the debugger)

   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.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: `CTRL-A` `f` `g`

   - When you have telneted to a terminal server that supports sending a remote break

     Press: `CTRL-]`

     Type in: `send break`

     Press: `Enter` `g`

3. Connect from gdb

   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.

kgdb and kdb interoperability

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.

Switching between kdb and kgdb

Switching from kgdb to kdb

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 the kernel debugger stops in kgdb mode it will print the message `KGDB or $3#33 for KDB`. 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.

1. Change from kgdb to kdb by blindly typing:

   ```
   $3#33
   ```

2. Change from kgdb to kdb with gdb:

   ```
   maintenance packet 3
   ```

   Note

   Now you must kill gdb. Typically you press `CTRL-Z` and issue the command:

   ```
   kill -9 %
   ```

Change from kdb to kgdb

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.

1. From kdb issue the command:

   ```
   kgdb
   ```

   Now disconnect your terminal program and connect gdb in its place

2. At the kdb prompt, disconnect the terminal program and connect gdb in its place.

Running kdb commands from gdb

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 `monitor help`.

Example:

```
(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)
```

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.

Kernel Debugger Internals

Architecture Specifics

The kernel debugger is organized into a number of components:

1. The debug core

   The debug core is found in `kernel/debugger/debug_core.c`. It contains:

   - 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
   - The API to invoke either the kdb or kgdb frontend to the debug core.
   - The structures and callback API for atomic kernel mode setting.

     Note

     kgdboc is where the kms callbacks are invoked.

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. gdbstub frontend (aka kgdb)

   The gdbstub is located in `kernel/debug/gdbstub.c`. It contains:

   > All the logic to implement the gdb serial protocol
4. kdb frontend

   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.

    - A simple shell
    - The kdb core command set
    - A registration API to register additional kdb shell commands.
        - A good example of a self-contained kdb module is the `ftdump` command for dumping the ftrace buffer. See: `kernel/trace/trace_kdb.c`
        - For an example of how to dynamically register a new kdb command you can build the kdb_hello.ko kernel module from `samples/kdb/kdb_hello.c`. To build this example you can set `CONFIG_SAMPLES=y` and `CONFIG_SAMPLE_KDB=m` in your kernel config. Later run `modprobe kdb_hello` and the next time you enter the kdb shell, you can run the `hello` command.
    - The implementation for `kdb_printf()` which emits messages directly to I/O drivers, bypassing the kernel log.
    - SW / HW breakpoint management for the kdb shell

5. kgdb I/O driver

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

    - configuration via built-in 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 watchdog 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.

int `kgdb_skipexception` (int *exception*, struct pt_regs * *regs*)
(optional) exit kgdb_handle_exception early

Parameters

`int exception`
Exception vector number

`struct pt_regs * 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.

void `kgdb_breakpoint` (void)
compiled in breakpoint

Parameters

`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.

int `kgdb_arch_init` (void)
Perform any architecture specific initalization.

Parameters

`void`
no arguments

Description

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

void `kgdb_arch_exit` (void)
Perform any architecture specific uninitalization.

Parameters

`void`
no arguments

Description

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

void `pt_regs_to_gdb_regs` (unsigned long * *gdb_regs*, struct pt_regs * *regs*)
Convert ptrace regs to GDB regs

Parameters

`unsigned long * gdb_regs`
A pointer to hold the registers in the order GDB wants.

`struct pt_regs * 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**.

void `sleeping_thread_to_gdb_regs` (unsigned long * *gdb_regs*, struct task_struct * *p*)
Convert ptrace regs to GDB regs

Parameters

`unsigned long * gdb_regs`
A pointer to hold the registers in the order GDB wants.

`struct task_struct * 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.

void `gdb_regs_to_pt_regs` (unsigned long * *gdb_regs*, struct pt_regs * *regs*)
Convert GDB regs to ptrace regs.

Parameters

`unsigned long * gdb_regs`
A pointer to hold the registers we’ve received from GDB.

`struct pt_regs * 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**.

int `kgdb_arch_handle_exception` (int *vector*, int *signo*, int *err_code*, char * *remcom_in_buffer*, char * *remcom_out_buffer*, struct pt_regs * *regs*)
Handle architecture specific GDB packets.

Parameters

`int vector`
The error vector of the exception that happened.

`int signo`
The signal number of the exception that happened.

`int err_code`
The error code of the exception that happened.

`char * remcom_in_buffer`
The buffer of the packet we have read.

`char * remcom_out_buffer`
The buffer of `BUFMAX` bytes to write a packet into.

`struct pt_regs * 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.

void `kgdb_roundup_cpus` (unsigned long *flags*)
Get other CPUs into a holding pattern

Parameters

`unsigned long 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.

void `kgdb_arch_set_pc` (struct pt_regs * *regs*, unsigned long *pc*)
Generic call back to the program counter

Parameters

`struct pt_regs * regs`
Current `struct pt_regs`.

`unsigned long pc`
The new value for the program counter

Description

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

void `kgdb_arch_late` (void)

Perform any architecture specific initalization.

Parameters

`void`
no arguments

Description

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

struct `kgdb_arch`

Describe architecture specific values.

Definition

```
struct kgdb_arch {
  unsigned char gdb_bpt_instr;
  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 (* disable_hw_break) (struct pt_regs *regs);
  void (* remove_all_hw_break) (void);
  void (* correct_hw_break) (void);
  void (* enable_nmi) (bool on);
};
```

Members

`gdb_bpt_instr`
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.

`disable_hw_break`
Allow an architecture to specify how to disable hardware breakpoints for a single cpu.

`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.

`enable_nmi`
Manage NMI-triggered entry to KGDB

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

Definition

```
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);
  int is_console;
};
```

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.

`is_console`
1 if the end device is a console 0 if the I/O device is not a console

kgdboc internals

kgdboc and uarts

The kgdboc driver is actually a very thin driver that relies on the underlying low level to the hardware driver having “polling hooks” to which the tty driver is attached. In the initial implementation of kgdboc 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 callback in the serial core which in turn uses the callback in the UART driver.

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 likely going to mean pressing the reset button.

kgdboc and keyboards

The kgdboc driver contains logic to configure communications with an attached keyboard. The keyboard infrastructure is only compiled into the kernel when `CONFIG_KDB_KEYBOARD=y` is set in the kernel configuration.

The core polled keyboard driver driver for PS/2 type keyboards is in `drivers/char/kdb_keyboard.c`. This driver is hooked into the debug core when kgdboc populates the callback in the array called `kdb_poll_funcs[]`. The `kdb_get_kbd_char()` is the top-level function which polls hardware for single character input.

kgdboc and kms

The kgdboc driver contains logic to request the graphics display to switch to a text context when you are using `kgdboc=kms,kbd`, provided that you have a video driver which has a frame buffer console and atomic kernel mode setting support.

Every time the kernel debugger is entered it calls `kgdboc_pre_exp_handler()` which in turn calls `con_debug_enter()` in the virtual console layer. On resuming kernel execution, the kernel debugger calls `kgdboc_post_exp_handler()` which in turn calls `con_debug_leave()`.

Any video driver that wants to be compatible with the kernel debugger and the atomic kms callbacks must implement the `mode_set_base_atomic`, `fb_debug_enter` and `fb_debug_leave operations`. For the `fb_debug_enter` and `fb_debug_leave` the option exists to use the generic drm fb helper functions or implement something custom for the hardware. The following example shows the initialization of the .mode_set_base_atomic operation in drivers/gpu/drm/i915/intel_display.c:

```
static const struct drm_crtc_helper_funcs intel_helper_funcs = {
[...]
        .mode_set_base_atomic = intel_pipe_set_base_atomic,
[...]
};
```

Here is an example of how the i915 driver initializes the fb_debug_enter and fb_debug_leave functions to use the generic drm helpers in `drivers/gpu/drm/i915/intel_fb.c`:

```
static struct fb_ops intelfb_ops = {
[...]
       .fb_debug_enter = drm_fb_helper_debug_enter,
       .fb_debug_leave = drm_fb_helper_debug_leave,
[...]
};
```

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>