Debugging the Linux kernel using the GDB

Contents

1 Configuring the Linux kernel

To get debug symbols and add the GDB script (i.e. Linux® awareness), the Linux kernel configuration must activate **CONFIG_DEBUG_INFO** and **CONFIG_GDB_SCRIPTS** using the Linux kernel Menuconfig tool (Menuconfig or how to configure kernel):

```
Symbol: DEBUG_INFO [=y]
Location:
  Kernel Hacking --->
    Tracers -->
      [*] -> Compile time checks and compiler options

Symbol: GDB_SCRIPTS [=y]
Location:
  Kernel Hacking --->
    Tracers -->
      [*] -> Compile-time checks and compiler options
        [*] Provide GDB scripts for kernel debugging
```

CONFIG_DEBUG_INFO and CONFIG_GDB_SCRIPTS are enabled by default in all STM32MPU software Packages, so you have nothing to do.

2 Kernel boot debugging issue

Please refer to GDB page for Linux kernel boot base.

If the default setting of setup.gdb file is used, the kernel stops in *stext* function.

```
...
define break_boot_linuxkernel
        symadd_bl32 $bl32_load_addr
        thbreak stext
        c
end
...
```

If you prefer to break in another function from the Linux kernel boot, just update the function name in the above break point definition #04572F.

You can then proceed with program execution or set new break point(s). A description of all GDB commands can be found here .

To monitor and debug source code execution, you can also use GDB UI as Gdbgui.

3 Debugging the kernel after boot

Please refer to GDB page for Linux kernel execution phase.

In this case, the GDB attaches the target on the current state.

You can then proceed with program execution or set new break point(s). A description of all GDB commands can be found here .

To monitor and debug source code execution, you can also use GDB UI as Gdbgui.

4 Debugging an already-probed kernel module

Debugging an external Linux kernel module requires some specific actions, especially because the symbols for this module are not part of the main vmlinux symbol file.

4.1 Adding symbols manually and debugging

4.1.1 Checking the module various ELF section addresses

Information on where the module various ELF sections were loaded into the kernel space can be obtained by looking into the */sys/module/<module_name>/sections* directory.

Below an example for the GPU driver:

```
Board $> ls -al /sys/module/galcore/sections/
 total 0
 drwxr-xr-x 2 root root    0 Jun 30 22:14 .
 drwxr-xr-x 7 root root    0 Jun 30 22:04 ..
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .ARM.exidx
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .ARM.exidx.exit.text
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .ARM.exidx.init.text
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .ARM.exidx.text.unlikely
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .ARM.extab
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .alt.smp.init
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .bss                    (where the BSS section
was loaded)
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .data                   (where the data
section was loaded)
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .exit.text
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .gnu.linkonce.this_module
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .init.text
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .note.gnu.build-id
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .pv_table
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .rodata
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .rodata.str
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .rodata.str1.4
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .strtab
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .symtab
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .text                   (where the text
section was loaded)
 -r--r--r-- 1 root root 4096 Jun 30 22:14 .text.unlikely
 -r--r--r-- 1 root root 4096 Jun 30 22:14 __bug_table
 -r--r--r-- 1 root root 4096 Jun 30 22:14 __param
```

You can then retrieve the addresses where the required sections have been loaded, mainly .bss (optional), .data and .text.

```
Board $> cat .text .data .bss
```

Below an example for the GPU driver:

```
Board $> cat .text .data .bss
 0xbf0ba000 --> .text
 0xbf0ee000 --> .data
 0xbf0fae40 --> .bss
```

4.1.2 Adding the module symbol file to the GDB environment

This is done by using the *add-symbol-file* GDB command:

```
(gdb) add-symbol-file <module.ko_path> <address of module's text section> \
  -s .data <address of module's data section> \
  -s .bss <address of module's bss section if available>
```

Below an example for the GPU driver:

If the symbol file is not available on your side, you can get it from the remote target:

```
PC $> scp root@ip_of_board>/lib/modules/4.19.4/extra/galcore.ko <your_path_to>/galcore.ko
```

When the symbol file is available, add it to the GDB environment:

```
(gdb) add-symbol-file <your_path_to>/galcore.ko 0xbf0ba000 \
  -s .data 0xbf0ee000 \
  -s .bss 0xbf0fae40
```

You can then search for a symbol belonging to the new module:

```
(gdb) p gpu_probe
 $3 = {int (struct platform_device *)} 0xbf0c4298 <gpu_probe>
```

and set a debug breakpoint.

4.2 Adding symbols using the Linux awareness and debug

Please note that the vmlinux symbol file must be aligned with the running OpenSTLinux kernel on the target board.

If you modify the Linux kernel sources, make sure the vmlinux path is updated in the Path_env.gdb script file.

- Enable Linux awareness (refer to Enabling Linux awareness paragraph).
- Load the symbol file.

```
# Go to your linux kernel build directory path.
(gdb) cd <your_linux_kernel_build_dir>
# Load the kernel symbol vmlinux and all the symbols of the loaded module(s)
(gdb) lx-symbols
```

You can search for a a symbol belonging to the new module (videobuf2-core.ko module in the example below):

```
(gdb) p vb2_core_queue_init
 $1 = {intt (struct vb2_queue *)} 0xbf089380 <vb2_core_queue_init>
```

and set a debug breakpoint.

- Specific configuration for external module driver

You can also use Linux awareness to debug an external module (source code out of the Linux kernel tree).

In that case, add an argument to *lx-symbols* command to provide the path to the external module .ko file (only path, not full path).

For example:

```
(gdb) cd <kernel_build_directory_path>
(gdb) lx-symbols /local/views/hands-on/openstlinux-distribution/openstlinux-hands-driver
```

5 Debugging a kernel module when probed

You can debug an external module when it is inserted (probed) at Linux kernel runtime or during Linux kernel boot phase.

- The Linux awareness environment for GDB is required in this context. Please refer to Enabling Linux awareness paragraph.
- Load the symbol file. You have to specify the directory path for your module symbol file if it is not part of the standard Linux kernel build tree.

```
# Go to your linux kernel build directory path:
(gdb) cd <your_linux_kernel_build_dir>
# Load the kernel symbol vmlinux and all the symbols of the loaded module(s):
(gdb) lx-symbols <your_symbol_file_path_if_required>
```

- Load source file paths

If your external module is not part of the standard Linux kernel tree, specify your source file path:

```
(gdb) directory <your_external_module_source_path>
```

- Add a breakpoint in the *load_module* function from the *kernel/module.c* Linux kernel source file

  This Linux kernel function loads the external module.

  Breakpoints have to be set at a specific line in the function. At this step, the module is prepared by the kernel but not yet started:

```
/* Now the module is in its final location, and you can initialize the linked lists. */
       err = module_unload_init(mod);
       if (err) --> please check the line number in the kernel/module.c file, i.e. line
3678
              goto unlink_mod;
       ...
```

```
# Insert a breakpoint in the load_module function at line 3678
(gdb) b module.c:3678
(gdb) continue
```

- Check that the module is the one that you inserted, by printing the module name from the module structure:

```
(gdb) p mod->name
```

  If the module is not the expected one, proceed with software execution until the expected module is available.

- When the breakpoint defined in the *load_module* function is reached, set your breakpoint in your external module

  For example:

```
(gdb) b goodix_ts_probe
```

- Proceed with software execution until you reach the breakpoint in your external module driver:

```
(gdb) continue
```

6 Debugging Linux kernel decompression

- Configure the GDB and the OpenOCD to be attached on the running target in U-Boot phase.
- Add a breakpoint in the *boot_jump_linux* function from U-Boot source file, *arch/arm/lib/bootm.c*.

  The U-Boot function jumps to the Linux kernel module previously loaded.

  Breakpoints have to be set at a specific line in the function:

```
...
   kernel_entry(0, machid, r2); -> please check the line number in the arch/arm/lib/bootm.
c file, i.e. line 400
...
```

```
# Insert a breakpoint in the load_module function:
(gdb) b arch/arm/lib/bootm.c:400
Breakpoint 1 at 0xc0100c70: arch/arm/lib/bootm.c:400. (2 locations)
(gdb) continue
```

- From the UART console, run the command to boot the Linux kernel image:

```
Board $> run bootcmd
```

- In the GDB, check that the software execution stopped at the expected breakpoint. You can then check the kernel_entry address (address of the Linux kernel compressed image that will be executed, i.e. zImage)

```
(gdb) p kernel_entry
$1 = (void (*)(int, int, uint)) 0xc2000040
```

- Load the compressed Linux kernel symbol file (see symadd_vmlinux_compressed variable define in the path_env.gdb configuration file used in the GDB).

```
(gdb) symadd_vmlinux_compressed
```

- Set a breakpoint in the *decompress_kernel* function from Linux kernel:

```
(gdb) b decompress_kernel
Breakpoint 2 at 0xc2000ae4: file <your_linux_kernel_source_dir>/arch/arm/boot/compressed
/misc.c, line 150
```

- Proceed with software execution. It then stops at the breakpoint set in the *decompress_kernel* function:

```
(gdb) continue
```

- From this point, you can start a debugging session.

7 Debugging using Linux awareness environment (Python plugin)

To enhance debugging experience, Linux awareness provides the debugger with additional knowledge on the underlying operating system (e.g. location of the task list in memory or kernel log buffer).

7.1 Enabling Linux awareness

It is highly recommended to use a Linux kernel compiled to get the Linux awareness environment, based on a Developer Package.

Refer to Install Linux kernel software package with STM32MP1 Developer Package for detail.

- Add Linux kernel build directories for Python path:

```
(gdb) add-auto-load-safe-path <your_linux_kernel_build_dir>
```

- Launch the Linux awareness Python script:

```
(gdb) source <your_linux_kernel_build_dir>/vmlinux-gdb.py
```

7.2 Displaying dmesg

```
(gdb) lx-dmesg
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.48 (oe-user@oe-host) (gcc version 7.3.0 (GCC)) #1 SMP
PREEMPT Thu Jul 5 07:37:20 UTC 2018
[    0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
[    0.000000] CPU: div instructions available: patching division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] OF: fdt: Machine model: STMicroelectronics STM32MP157C-EV1 pmic eval
daughter on eval mother
[    0.000000] bootconsole [earlycon0] enabled
...
```

7.3 Displaying the module list

```
(gdb) lx-lsmod
 Address    Module                  Size  Used by
 0xbf0ba000 galcore               294912  0
 0xbf0ad000 stm32_dcmi             28672  0
 0xbf0a4000 videobuf2_dma_contig    20480  1 stm32_dcmi
 0xbf09d000 videobuf2_memops       16384  1 videobuf2_dma_contig
 0xbf094000 videobuf2_v4l2         20480  1 stm32_dcmi
 0xbf086000 videobuf2_core         36864  2 stm32_dcmi,videobuf2_v4l2
 0xbf078000 ov5640                 36864  0
 0xbf071000 v4l2_fwnode            16384  2 stm32_dcmi,ov5640
 0xbf06a000 v4l2_common            16384  2 stm32_dcmi,ov5640
 0xbf038000 videodev              139264  5 stm32_dcmi,videobuf2_v4l2,videobuf2_core,
```

```
ov5640,v4l2_common
 0xbf030000 stm32_cec              16384  0
 0xbf022000 media                  32768  2 ov5640,videodev
 0xbf013000 cec                    36864  1 stm32_cec
 0xbf009000 snd_soc_stm32_spdifrx    20480  2
 0xbf000000 goodix                 16384  0
```

7.4 Displaying the current kernel processes

```
(gdb) lx-ps
 0xc1007580 <init_task> 0 swapper/0
 0xee8b8000 1 systemd
 0xee8b8680 2 kthreadd
 0xee8b9380 4 kworker/0:0H
 0xee8b9a00 5 kworker/u4:0
 0xee8ba080 6 mm_percpu_wq
 0xee8ba700 7 ksoftirqd/0
 0xee8bad80 8 rcu_preempt
 0xee8bb400 9 rcu_sched
 0xee8bba80 10 rcu_bh
 ...
```

- Print task_struct information:

```
(gdb) p *(struct task_struct *)<addr_of_process, see above>
```

For example:

```
(gdb) p *(struct task_struct *)0xee8ba080
 $1 = {state = 1026, stack = 0xee8ca000, usage = {counter = 2}, flags = 69238880, ptrace
= 0, wake_entry = {next = 0x0}, on_cpu = 0, wakee_flips = 0, wakee_flip_decay_ts = 0,
...
```

- Print the executable name, path excluded:

```
(gdb) p (*(struct task_struct *)0xee8ba080)->comm
 $7 = "mm_percpu_wq\000\000\000"
```

- Print the PID:

```
(gdb) p (*(struct task_struct *)0xee8ba080)->pid
 $8 = 6
```

- Print stack and thread information (ti)

```
(gdb) p  *(struct thread_info *)(*(struct task_struct *)0xee8ba080)->stack
 $9 = {flags = 0, preempt_count = 2, addr_limit = 0, task = 0xee8ba080, cpu = 0,
cpu_domain = 0, cpu_context = {r4 = 4017867840, r5 = 4002128000, r6 = 4002119680, r7 =
3238137856, r8 = 0, r9 = 0, sl = 3238767332, fp = 4002201396, sp = 4002201320, pc =
3232110636, extra = {0, 0}}, syscall = 0, used_cp = '\000' <repeats 15 times>, tp_value =
{0, 4115157644}, fpstate = {hard = {save = {0 <repeats 35 times>}}, soft = {save = {0
<repeats 35 times>}}}, vfpstate = {hard = {fpregs = {0 <repeats 32 times>}, fpexc = 0,
fpscr = 0, fpinst = 0, fpinst2 = 0, cpu = 0}}, thumbee_state = 0}
```

7.5 Container_of

Each task contains a pointer towards the previous and next task. These pointers are actually pointing to the list itself and not the object.

To get the exact object address, kernel developers generally use container_of. The same macro is implemented in the plugin to quickly browse data structures.

Below an example to find the **next task_struct**:

```
(gdb) p $container_of(init_task.tasks.next, "struct task_struct", "tasks")
 $10 = (struct task_struct *) 0xee8b8000
```

7.6 Checking list consistency

```
(gdb) p init_task.tasks
 $11 = {next = 0xee8b8278, prev = 0xedd495f8}
(gdb) lx-list-check init_task.tasks
 Starting with: {next = 0xee8b8278, prev = 0xedd495f8}
 list is consistent: 92 node(s)
```

7.7 Loading symbols

The *lx-symbols* command (re)loads the symbols of the Linux kernel and the currently loaded modules.

The kernel (vmlinux) is directly taken from the current working. Modules (.ko) are scanned recursively, starting in the same directory.

Optionally, the module search path can be extended by a list of paths separated by a space and passed to the lx-symbols command:

```
(gdb) cd <kernel_build_directory_path>
(gdb) lx-symbols [specific_module_symbol_path1] [specific_module_symbol_path2]...
```

7.8 Other commands

You can get the list of commands for Linux awareness:

```
(gdb) apropos lx
 function lx_current -- Return current task
 function lx_module -- Find module by name and return the module variable
```

```
  function lx_per_cpu -- Return per-cpu variable
  function lx_task_by_pid -- Find Linux task by PID and return the task_struct variable
  function lx_thread_info -- Calculate Linux thread_info from task variable
  function lx_thread_info_by_pid -- Calculate Linux thread_info from task variable found
by pid
  lx-cmdline --  Report the Linux Commandline used in the current kernel
  lx-cpus -- List CPU status arrays
  lx-dmesg -- Print Linux kernel log buffer
  lx-fdtdump -- Output Flattened Device Tree header and dump FDT blob to the filename
  lx-iomem -- Identify the IO memory resource locations defined by the kernel
  lx-ioports -- Identify the IO port resource locations defined by the kernel
  lx-list-check -- Verify a list consistency
  lx-lsmod -- List currently loaded modules
  lx-mounts -- Report the VFS mounts of the current process namespace
  lx-ps -- Dump Linux tasks
  lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modules
  lx-version --  Report the Linux Version of the current kernel
```

8 Reference

- Useful external links

Document link	Document Type	Description
Gdb Kernel debugging	User guide	Documentation from kernel.org

GNU dedugger, a portable debugger that runs on many Unix-like systems

User Interface

Executable Linkable Format - NEW

Graphics Processing Units

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Universal Asynchronous Receiver/Transmitter

Central processing unit

symetric multiprocessing

input/output

Virtual File System