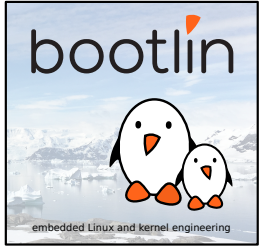
Kernel debugging



ReDS
Reconfigurable & embedded Digital Systems

Debugging using messages (1/3)

Three APIs are available

- The old `printk()`, no longer recommended for new debugging messages
- The `pr_*()` family of functions: `pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_err()`, `pr_warn()`, `pr_notice()`, `pr_info()`, `pr_cont()`
  and the special `pr_debug()` (see next pages)
  - Defined in `include/linux/printk.h`
  - They take a classic format string with arguments
  - Example:

    ```
    pr_info("Booting CPU %d\n", cpu);
    ```

  - Here's what you get in the kernel log:

    ```
    [  202.350064] Booting CPU 1
    ```

- `print_hex_dump_debug()`: useful to dump a buffer with `hexdump` like display

Debugging using messages (2/3)

- The `dev_*()` family of functions: `dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_err()`, `dev_warn()`, `dev_notice()`, `dev_info()`
  and the special `dev_dbg()` (see next page)
  - They take a pointer to `struct device` as first argument, and then a format string with arguments
  - Defined in `include/linux/dev_printk.h`
  - To be used in drivers integrated with the Linux device model
  - Example:

    ```
    dev_info(&pdev->dev, "in probe\n");
    ```

  - Here's what you get in the kernel log:

    ```
    [   25.878382] serial 48024000.serial: in probe
    [   25.884873] serial 481a8000.serial: in probe
    ```

- `*_ratelimited()` version exists which limits the amount of print if called too much based on `/proc/sys/kernel/printk_ratelimit{_burst}` values

Debugging using messages (3/3)

- The kernel defines many more format specifiers than the standard `printf()` existing ones.
  - `%p`: Display the hashed value of pointer by default.
  - `%px`: Always display the address of a pointer (use carefully on non-sensitive addresses).
  - `%pK`: Display hashed pointer value, zeros or the pointer address depending on `kptr_restrict` sysctl value.
  - `%pOF`: Device-tree node format specifier.
  - `%pr`: Resource structure format specifier.
  - `%pa`: Physical address display (work on all architectures 32/64 bits)
  - `%pe`: Error pointer (displays the string corresponding to the error number)
- `/proc/sys/kernel/kptr_restrict` should be set to `1` in order to display pointers which uses `%pK`
- See `core-api/printk-formats` for an exhaustive list of supported format specifiers

pr_debug() and dev_dbg()

- When the driver is compiled with `DEBUG` defined, all these messages are compiled and printed at the debug level. `DEBUG` can be defined by `#define DEBUG` at the beginning of the driver, or using `ccflags-$(CONFIG_DRIVER) += -DDEBUG` in the `Makefile`
- When the kernel is compiled with `CONFIG_DYNAMIC_DEBUG`, then these messages can dynamically be enabled on a per-file, per-module or per-message basis, by writing commands to `/proc/dynamic_debug/control`. Note that messages are not enabled by default.
  - Details in `admin-guide/dynamic-debug-howto`
  - Very powerful feature to only get the debug messages you're interested in.
- When neither `DEBUG` nor `CONFIG_DYNAMIC_DEBUG` are used, these messages are not compiled in.

Configuring the priority

- Each message is associated to a priority, ranging from `0` for emergency to `7` for debug, as specified in `include/linux/kern_levels.h`.
- All the messages, regardless of their priority, are stored in the kernel log ring buffer
  - Typically accessed using the `dmesg` command
- Some of the messages may appear on the console, depending on their priority and the configuration of
  - The `loglevel` kernel parameter, which defines the priority number below which messages are displayed on the console. Details in `admin-guide/kernel-parameters`. Examples: `loglevel=0`: no message, `loglevel=8`: all messages
  - The value of `/proc/sys/kernel/printk`, which allows to change at runtime the priority above which messages are displayed on the console. Details in `admin-guide/sysctl/kernel`

DebugFS

A virtual filesystem to export debugging information to user space.

- Kernel configuration: `CONFIG_DEBUG_FS`
  - `Kernel hacking -> Debug Filesystem`
- The debugging interface disappears when Debugfs is configured out.
- You can mount it as follows:
  - `sudo mount -t debugfs none /sys/kernel/debug`
- First described on https://lwn.net/Articles/115405/
- API documented in the Linux Kernel Filesystem API: `filesystems/debugfs` The debugfs filesystem

DebugFS API

- Create a sub-directory for your driver:
  - `struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);`
- Expose an integer as a file in DebugFS. Example:
  - `struct dentry *debugfs_create_u8 (const char *name, mode_t mode, struct dentry *parent, u8 *value);`
    - `u8`, `u16`, `u32`, `u64` for decimal representation
    - `x8`, `x16`, `x32`, `x64` for hexadecimal representation
- Expose a binary blob as a file in DebugFS:
  - `struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob);`
- Also possible to support writable DebugFS files or customize the output using the more generic `debugfs_create_file()` function.

Deprecated debugging mechanisms

Some additional debugging mechanisms, whose usage is now considered deprecated

- Adding special `ioctl()` commands for debugging purposes. DebugFS is preferred.
- Adding special entries in the `proc` filesystem. DebugFS is preferred.
- Adding special entries in the `sysfs` filesystem. DebugFS is preferred.
- Using `printk()`. The `pr_*()` and `dev_*()` functions are preferred.

Using Magic SysRq

Functionnality provided by serial drivers

- Allows to run multiple debug / rescue commands even when the kernel seems to be in deep trouble
  - On PC: press `[Alt]` + `[Prnt Scrn]` + `<character>` simultaneously (`[SysRq]` = `[Alt]` + `[Prnt Scrn]`)
  - On embedded: in the console, send a break character (Picocom: press `[Ctrl]` + `a` followed by `[Ctrl]` + `\`), then press `<character>`
- Example commands:
  - `h`: show available commands
  - `s`: sync all mounted filesystems
  - `b`: reboot the system
  - `n`: makes RT processes nice-able.
  - `w`: shows the kernel stack of all sleeping processes
  - `t`: shows the kernel stack of all running processes
  - You can even register your own!
- Detailed in `admin-guide/sysrq`

kgdb - A kernel debugger

- CONFIG_KGDB in *Kernel hacking*.
- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Feature supported for the most popular CPU architectures
- CONFIG_GDB_SCRIPTS allows to build GDB python scripts that are provided by the kernel.
  - See dev-tools/gdb-kernel-debugging for more information

Using kgdb (1/2)

- Details available in the kernel documentation: `dev-tools/kgdb`
- You must include a kgdb I/O driver. One of them is `kgdb` over serial console (`kgdboc`: `kgdb` over console, enabled by `CONFIG_KGDB_SERIAL_CONSOLE`)
- Configure `kgdboc` at boot time by passing to the kernel:
  - `kgdboc=<tty-device>,<bauds>`.
  - For example: `kgdboc=ttyS0,115200`
- Or at runtime using sysfs:
  - `echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc`
  - If the console does not have polling support, this command will yield an error.

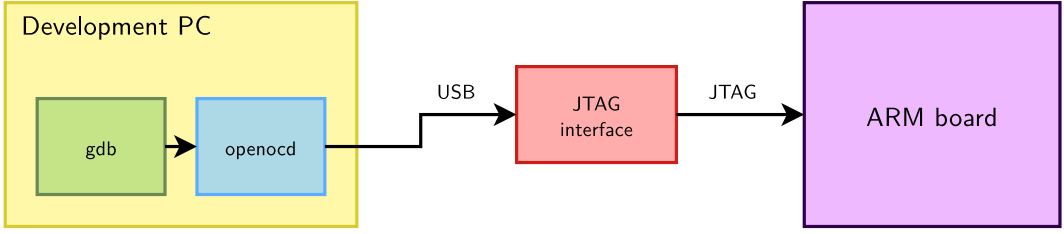
Using kgdb (2/2)

- Then also pass `kgdbwait` to the kernel: it makes `kgdb` wait for a debugger connection.
- Boot your kernel, and when the console is initialized, interrupt the kernel with a break character and then `g` in the serial console (see our *Magic SysRq* explanations).
- On your workstation, start `gdb` as follows:
  - `arm-linux-gdb ./vmlinux`
  - `(gdb) set remotebaud 115200`
  - `(gdb) target remote /dev/ttyS0`
- Once connected, you can debug a kernel the way you would debug an application program.
- On GDB side, the first threads represent the CPU context (ShadowCPU<x>), then all the other threads represents a task.

Debugging with a JTAG interface

Two types of JTAG dongles

- The ones offering a gdb compatible interface, over a serial port or an Ethernet connection. gdb can directly connect to them.
- The ones not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger): http://openocd.sourceforge.net/
  - OpenOCD is the bridge between the gdb debugging language and the JTAG interface of the target CPU.
  - See the very complete documentation: https://openocd.org/pages/documentation.html
  - For each board, you'll need an OpenOCD configuration file (ask your supplier)



Early traces

- If something breaks before the `tty` layer, serial driver and serial console are properly registered, you might just have nothing else after "`Starting kernel...`"
- On ARM, if your platform implements it, you can activate (`CONFIG_DEBUG_LL` and `CONFIG_EARLYPRINTK`), and add `earlyprintk` to the kernel command line
  - Assembly routines to just push a character and wait for it to be sent
  - Extremely basic, but is part of the uncompressed section, so available even if the kernel does not uncompress correctly!
- On other platforms, hoping that your serial driver implements `OF_EARLYCON_DECLARE()`, you can enable `SERIAL_EARLYCON`
  - The kernel will try to hook an appropriate `earlycon` UART driver using the `stdout-path` of the device-tree.

More kernel debugging tips

- Make sure `CONFIG_KALLSYMS_ALL` is enabled
  - To get oops messages with symbol names instead of raw addresses
  - Turned on by default
- Make sure `CONFIG_DEBUG_INFO` is also enabled
  - This way, the kernel is compiled with `$(CROSSCOMPILE)gcc -g`, which keeps the source code inside the binaries.
- If your device is not probed, try enabling `CONFIG_DEBUG_DRIVER`
  - Extremely verbose!
  - Will enable all the debug logs in the device-driver core section

Getting help and reporting bugs

- If you are using a custom kernel from a hardware vendor, contact that company. The community will have less interest supporting a custom kernel.
- Otherwise, or if this doesn't work, try to reproduce the issue on the latest version of the kernel.
- Make sure you investigate the issue as much as you can: see `admin-guide/bug-bisect`
- Check for previous bugs reports. Use web search engines, accessing public mailing list archives.
- If you're the first to face the issue, it's very useful for others to report it, even if you cannot investigate it further.
- If the subsystem you report a bug on has a mailing list, use it. Otherwise, contact the official maintainer (see the `MAINTAINERS` file). Always give as many useful details as possible.