./

Learning Report –

Debugging Techniques & Advanced Drivers



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **Approved By** | **Remarks/Revision Details** |
| 1 | 17-11-2020 | DEEKSHA P  (99002664) | - |  | Update on Debug techniques and Advanced drivers |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents

[1. DEBUGGING TECHNIQUES AND VALIDATION 5](#_Toc56697735)

[1.1 printk() 5](#_Toc56697736)

[1.2 Kernel Log Levels 6](#_Toc56697737)

[To determine current console level: 9](#_Toc56697738)

[To change the current console log level: 9](#_Toc56697739)

[1.3 pr\_xxxx APIs 10](#_Toc56697740)

[1.4 OOPS 10](#_Toc56697741)

[1.4.1 Objdump 10](#_Toc56697742)

[1.4.2 Debugging with OOPS 11](#_Toc56697743)

[1.5 Ftrace - Function Tracer 14](#_Toc56697745)

[1.5.1 Setting up the Ftrace 15](#_Toc56697746)

[1.5.2 Ftrace mounting 15](#_Toc56697747)

[1.5.3 Function Tracing 16](#_Toc56697748)

[1.5.4 Using Trace\_printk 20](#_Toc56697749)

[1.6 BUG\_ON 21](#_Toc56697750)

[1.7 Magic SysRQ 22](#_Toc56697751)

[1.8 GDB 23](#_Toc56697752)

[2. Advanced Drivers 25](#_Toc56697753)

[2.1 ioctl() – control device 25](#_Toc56697754)

[2.2 Driver Model 26](#_Toc56697755)

[2.2.1 Platform Devices 27](#_Toc56697756)

[2.2.2 Platform Drivers 27](#_Toc56697757)

[2.2.3 Device Tree Mapping 29](#_Toc56697758)

[2.3 Interrupts 30](#_Toc56697759)

[2.4 Top Halves and Bottom Halves 31](#_Toc56697760)

[References 33](#_Toc56697761)

**List of Figures**

[Figure 1: printk() statements 6](#_Toc56698324)

[Figure 2: To check the printk debugging in Qemu 6](#_Toc56698325)

[Figure 3: Log levels 7](#_Toc56698326)

[Figure 4:changing default log level 10](#_Toc56698327)

[Figure 5: Use of objdump 12](#_Toc56698328)

[Figure 6 Figure displaying the error logs 12](#_Toc56698329)

[Figure 7: Figure displaying the address with the user code where the file is crashing 13](#_Toc56698330)

[Figure 8: Error line in .c file 14](#_Toc56698331)

[Figure 9: Use of current\_tracer 16](#_Toc56698332)

[Figure 10: Use of available\_tracer 17](#_Toc56698333)

[Figure 11: Use of tracing\_on 17](#_Toc56698334)

[Figure 12: Various tracer selection in menuconfig 18](#_Toc56698335)

[Figure 13: ftrace in Qemu 19](#_Toc56698336)

[Figure 14: page fault tracing 20](#_Toc56698337)

[Figure 15: Use of trace\_printk 21](#_Toc56698338)

[Figure 16:SysRq values 23](#_Toc56698339)

[Figure 17: Commands used in GDB debugger 24](#_Toc56698340)

[Figure 18: ioctl commands defined in scull 26](#_Toc56698341)

[Figure 19: Platform Devices 27](#_Toc56698342)

[Figure 20: Platform drivers and device in same .c file 28](#_Toc56698343)

[Figure 21: Platform drivers and devices in separate .c files 29](#_Toc56698344)

[Figure 22: Linux Device Tree for Hardware mapping 30](#_Toc56698345)

[Figure 23: Dummy IRQ 31](#_Toc56698346)

[Figure 24: Difference between top and bottom halves 32](#_Toc56698347)

**List of Tables**

[Table 1: Linux Kernel log levels 8](#_Toc56697383)

# 1. DEBUGGING TECHNIQUES AND VALIDATION

## 1.1 printk()

1. It’s the standard tool we have for printing messages and usually the most basic way of tracing and debugging. printk() is one of the most widely known functions in the Linux kernel.
2. printk works more or less the same way as printf in userspace, so if you ever debugged your userspace program using printf, you are ready to do the same with your kernel code.

printk("My Debugger is Printk\n");

1. Some of important aspects of printk() listed below.
   1. printk() messages can specify a log level.
   2. the format string, while largely compatible with C99, doesn’t follow the exact same specification. It has some extensions and a few limitations (no %n or floating point conversion specifiers). See How to get printk format specifiers right.
2. All printk() messages are printed to the kernel log buffer, which is a ring buffer exported to user space through /dev/kmsg. The usual way to read it is using dmesg. We can customize the kernel, by enabling kernel hacking ->printk & dmesg opearation
3. printk() is typically used like this:
   1. printk(KERN\_INFO "Message: %s\n", arg);

**Procedure:**

1. In the kernel-modules folder use the logdemo folder use the logdemo.c and run the make command to get the .ko file
2. Mount the rootfs copy the .ko file and then unmount the file and then launch qemu.
3. To check the current log level of printk use the command cat /proc/sys/kernel/printk
4. Use the following commands to see the output

* Insmod lodemo.ko
* cat /proc/sys/kernel/printk
* echo “5” > /proc/sys/kernel/printk
* rmmod logdemo.ko
* Insmod lodemo.ko
* rmmod logdemo.ko

1. In order to change the current log level use the command echo “5” > /proc/sys/kernel/printk
2. After the change in the current log level remove the module and add the module again to check the changes.

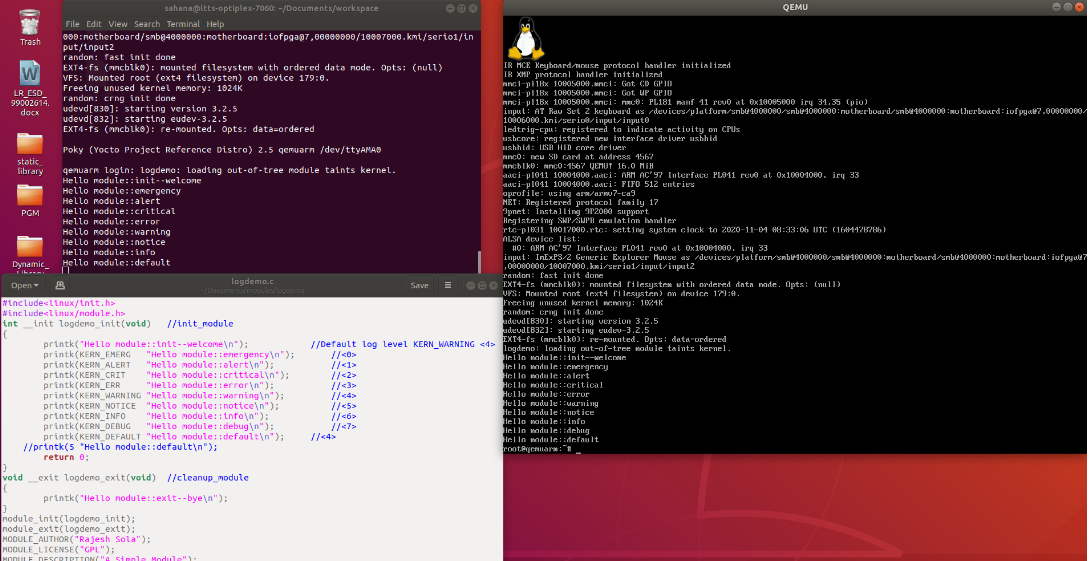


Figure : printk() statements

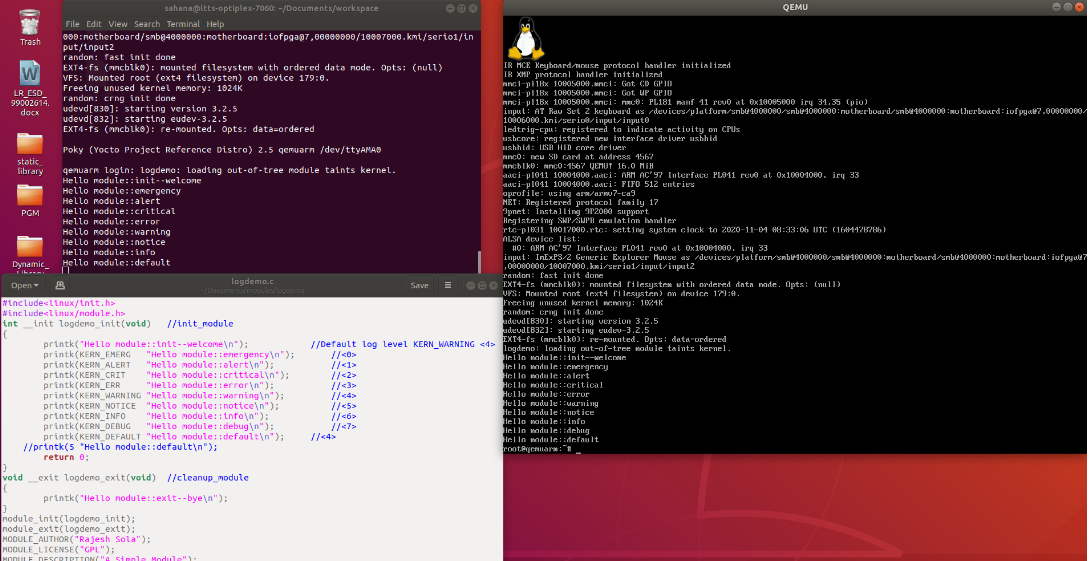


Figure : To check the printk debugging in Qemu

## 1.2 Kernel Log Levels

There are basically eight log levels which a message sent by the linux kernel can adopt, starting from level 0 and decreasing in severity 'till level 7: the lowest log level identifier, the most critical context.

printk(KERN\_ERR "something went wrong, return code: %d\n",ret);

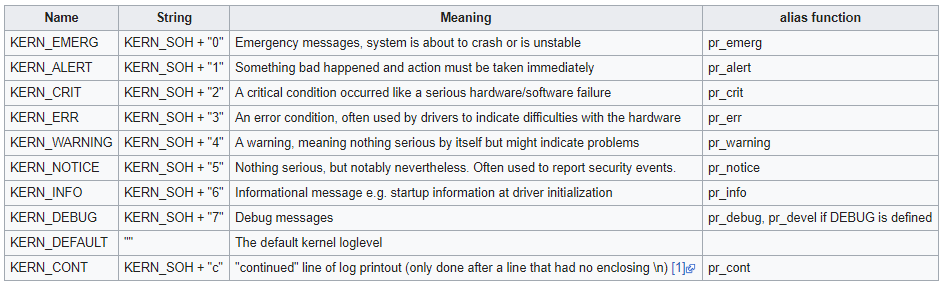


Figure : Log levels

When a log level is set as the default for the console, either persistently or temporarily, it acts as a filter, so that only messages with a log level lower than it, (therefore messages with a higher severity) are displayed [1].

1. The first log level is 0, identified by the KERN\_EMERG string. This is the highest level in order of severity: it's adopted by messages about system instability or imminent crashes.
2. Loglevel 1, or KERN\_ALERT it's what comes immediately after. This level is used in situations where the user attention is immediately required.
3. The next log level in order of severity is KERN\_CRIT, or loglevel 2. This level of severity is used to inform about critical errors, both hardware or software related.
4. Loglevel 3, also identified by the KERN\_ERR string, it's the next in the scale. Messages adopting this level are often used to notify the user about non-critical errors, as for example a failed or problematic device recognition, or more generally driver-related problems.
5. loglevel 4 or KERN\_WARNING it’s the log level usually used as the default in the most of Linux distributions. This level it's used to display warnings or messages about non-imminent errors.
6. Loglevel 5, it's KERN\_NOTICE. Messages which uses this level of severity are about events which may be worth noting.
7. Loglevel 6 it's KERN\_INFO: this is the log level used for informational messages about the action performed by the kernel.
8. Finally, we have KERN\_DEBUG, or loglevel 7, which is mainly used for debugging.

Table 1: Linux Kernel log levels

|  |  |  |  |
| --- | --- | --- | --- |
| Name | String | Meaning | alias function |
| KERN\_EMERG | KERN\_SOH + "0" | Emergency messages, system is about to crash or is unstable | pr\_emerg |
| KERN\_ALERT | KERN\_SOH + "1" | Something bad happened and action must be taken immediately | pr\_alert |
| KERN\_CRIT | KERN\_SOH + "2" | A critical condition occurred like a serious hardware/software failure | pr\_crit |
| KERN\_ERR | KERN\_SOH + "3" | An error condition, often used by drivers to indicate difficulties with the hardware | pr\_err |
| KERN\_WARNING | KERN\_SOH + "4" | A warning, meaning nothing serious by itself but might indicate problems | pr\_warning |
| KERN\_NOTICE | KERN\_SOH + "5" | Nothing serious, but notably nevertheless. Often used to report security events. | pr\_notice |
| KERN\_INFO | KERN\_SOH + "6" | Informational message e.g. startup information at driver initialization | pr\_info |
| KERN\_DEBUG | KERN\_SOH + "7" | Debug messages | pr\_debug, pr\_devel if DEBUG is defined |
| KERN\_DEFAULT | "" | The default kernel loglevel |  |
| KERN\_CONT | KERN\_SOH + "c" | "continued" line of log printout (only done after a line that had no enclosing \n) [1] | pr\_cont |

## To determine current console level:

Checking the default loglevel used on our system it's very easy. All we must do is to examine the content of the /proc/sys/kernel/printk file, run below commands

* + $ cat /proc/sys/kernel/printk

This is the typical output of the command:

* + 7 4 1 7

The first integer shows you your current console\_loglevel: the value, 7 in this case, represents the log level currently used. It tells that only messages adopting a severity level higher than it, will be displayed on the console.

The second value in the output represents the default\_message\_loglevel. This value is automatically used for messages without a specific log level: if a message is not associated with a log level, this one will be used for it.

The third value in the output reports the minimum\_console\_loglevel status. It indicates the minimum loglevel which can be used for console\_loglevel. The level here used it's 1, the highest.

Finally, the last value represents the default\_console\_loglevel, which is the default log level used for console\_loglevel at boot time.

## To change the current console log level:

To change your current console\_loglevel simply write to this file, so in order to get all messages printed to the console do a simple command as shown below and every kernel message will appear on your console.

/proc/sys/kernel/printk

* echo "4" > /proc/sys/kernel/printk
* cat /proc/sys/kernel/printk
  + 4 4 1 7

Another way to change the console log level is to use *dmesg* with the *-n* parameter

* #set console\_loglevel to print KERN\_WARNING (4) or more severe messages
* dmesg -n 5

Only messages with a value lower (**not** lower equal) than the console\_loglevel will be printed.

You can also specify the console\_loglevel at boot time using the *loglevel* boot parameter.

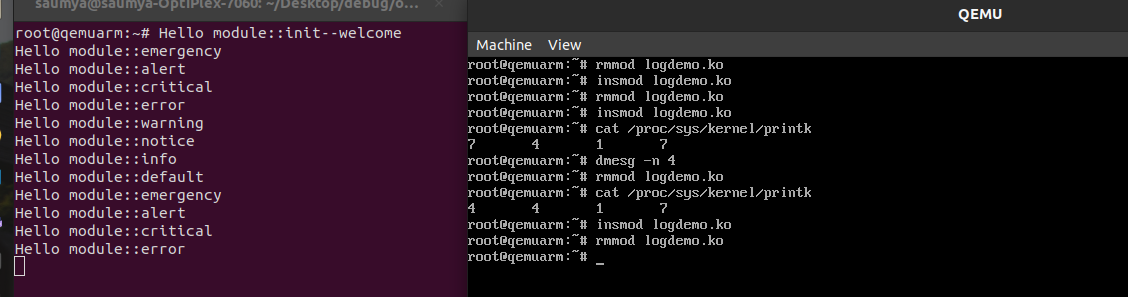


Figure 4:changing default log level

## 1.3 pr\_xxxx APIs

1. These are used when messages that aren’t associated with any particular device. [2]
2. A header called <linux/printk.h> is used which defines few functions like pr\_notice(), pr\_info(), pr\_warn(), pr\_err(), etc.
3. debug message printing is handled differently than printing other non-debug messages i.e pr\_debug() is handled differently compared to others.
4. Other pr\_xxxx API print unconditionally, it pr\_debug is not compiled by default.
5. If CONFIG\_DYNAMIC\_DEBUG is set, then only kernel will include while compiling.

## 1.4 OOPS

### 1.4.1 Objdump

To debug a kernel, use objdump and look for the hex offset from the crash output to find the valid line of code/assembler. Without debug symbols, you will see the assembler code for the routine shown, but if your kernel has debug symbols the C code will also be available. (Debug symbols can be enabled in the kernel hacking menu of the menu configuration.) For example:

* $ objdump -r -S -l --disassemble net/dccp/ipv4.o

### 1.4.2 Debugging with OOPS

To check the error debugging of OOPS, we didn’t allocate memory for local array and try to use in copy from user AP [3]. Here dereferencing the null pointer makes the file to be crashed. To find where the crash is happening will use a OOPs method.

1. Few lines of code where the crashing happen is shown below:

void noinline do\_write(void)

{  
int i,dummy=0;  
for(i=1;i<=10;i++)   
dummy+=i;  
\*ptr = 100;   
printk("end of do\_write\n");  
}

void noinline do\_read(void)

{  
int val,i,dummy;  
for(i=1;i<=10;i++)   
dummy+=i;  
val = \*ptr;   
printk("val is %d\n",\*ptr);  
}

static int \_\_init hello\_init(void)

{  
printk("Hello World..welcome\n");  
if(choice==1)  
do\_write();   
else  
do\_read();  
return 0;  
}

1. First, we compiled the code use objdump to see assembly and c code side by side as below.

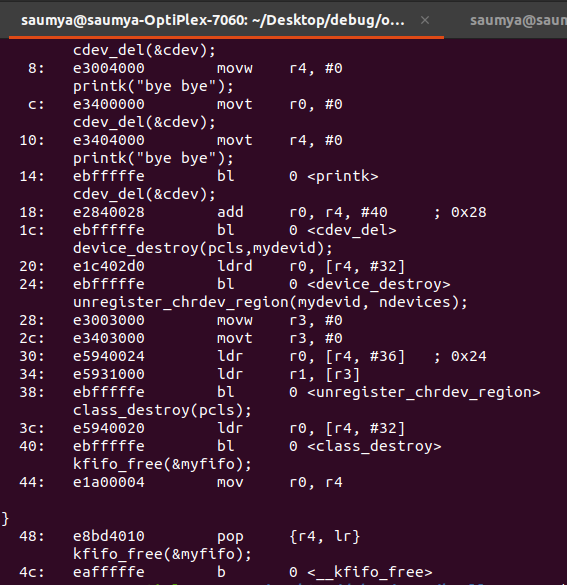


Figure : Use of objdump

1. Generate the .ko file by using the command make and mount the rootfs and then copy the .ko file and then unmount the file and launch qemu.
2. To make the do\_write to crash give the command insmod oops.ko
3. To make the do\_read to crash give the command insmod oops.ko choice=2.
4. A large log will be printed and the qemu shows a segmentation fault. In the large log file search PC and LR to check address 0x18.

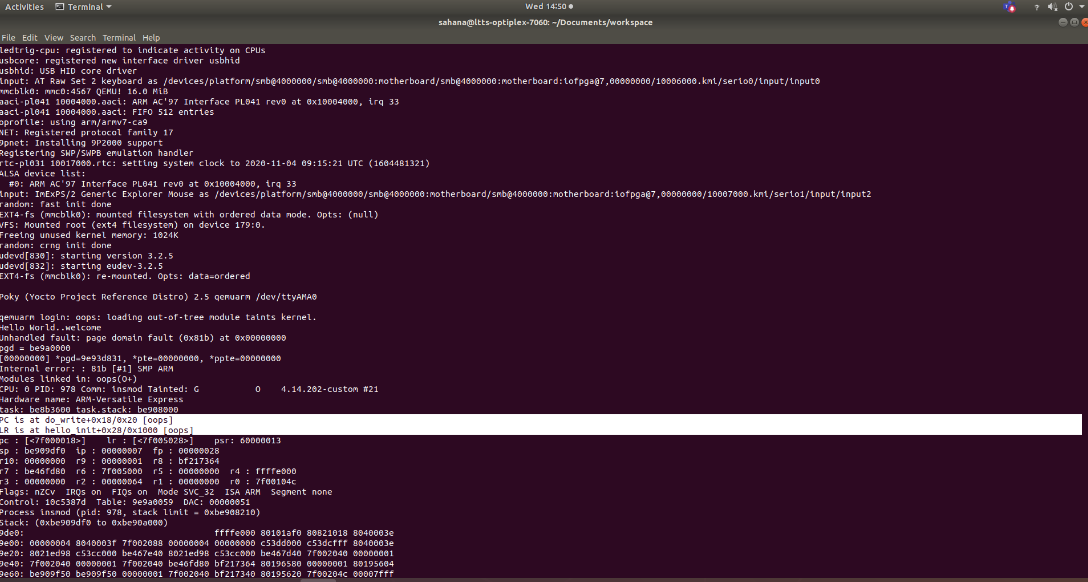


Figure Figure displaying the error logs

1. Then using addr2line we can find which line is giving error in our c code.
2. Then use the command arm-linux-gnueabi-objdump –d oops.ko to get the assembly code equivalent and find the same address.
3. Then use the command arm-linux-gnueabi-objdump –S oops.ko to get the combination of assembly and user code.

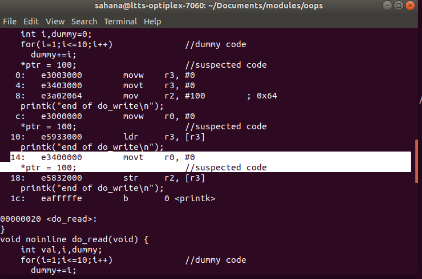


Figure : Figure displaying the address with the user code where the file is crashing

1. To find the line number in .c file use the command addr2line 0x18 –e oops.ko
2. We must add offset to F8 address which got from OOPS debug message which leads to

1DC

* In the other terminal use the command arm-linux-gnueabi-objdump –S stat\_error.ko and also there find the offset of copy to user which is 108.
* Add both the offsets to get the hexa value to be 19c.
* Now use the command addr2line 0x19c –e stat\_error.ko to find the line number equivalent.

# 

Figure : Error line in .c file

## 1.5 Ftrace - Function Tracer

Ftrace is a tracing utility built directly into the Linux kernel. Many distributions already have various configurations of Ftrace enabled in their most recent releases. One of the benefits that Ftrace brings to Linux is the ability to see what is happening inside the kernel. As such, this makes finding problem areas or simply tracking down that strange bug more manageable.

Ftrace's ability to show the events that led up to a crash gives a better chance of finding exactly what caused it and can help the developer in creating the correct solution. This article is a two-part series that will cover various methods of using Ftrace for debugging the Linux kernel. This first part will talk briefly about setting up Ftrace, using the function tracer, writing to the Ftrace buffer from within the kernel, and various ways to stop the tracer when a problem is detected.

Ftrace is the Linux kernel internal tracer that was included in the Linux kernel in 2.6.27. Although Ftrace is named after the function tracer it also includes many more functionalities. But the function tracer is the part of Ftrace that makes it unique as you can trace almost any function in the kernel and with dynamic Ftrace, it has no overhead when not enabled.

### 1.5.1 Setting up the Ftrace

Currently the API to interface with Ftrace is located in the Debugfs file system. Typically, that is mounted at /sys/kernel/debug. For easier accessibility, I usually create a /debug directory and mount it there. Feel free to choose your own location for Debugfs.

When Ftrace is configured, it will create its own directory called tracing within the Debugfs file system. This article will reference those files in that directory as though the user first changed directory to the Debugfs tracing directory to avoid any confusion as to where the Debugfs file system has been mounted.

[~]# cd /sys/kernel/debug/tracing

[tracing]#

This article is focusing on using Ftrace as a debugging tool. Some configurations for Ftrace are used for other purposes, like finding latency or analyzing the system. For debugging, the kernel configuration parameters that should be enabled are:

CONFIG\_FUNCTION\_TRACER

CONFIG\_FUNCTION\_GRAPH\_TRACER

CONFIG\_STACK\_TRACER

CONFIG\_DYNAMIC\_FTRACE

### 1.5.2 Ftrace mounting

When tracefs is configured into the kernel (which selecting any ftrace option will do) the directory /sys/kernel/tracing will be created. To mount this directory, you can add to your /etc/fstab file:

* **Tracefs /sys/kernel/tracing tracefs defaults 0 0**

To mount it at the run-time:

* **mount -t tracefs nodev /sys/kernel/tracing**

For quicker access to that directory you may want to make a soft link to it:

* **ln -s /sys/kernel/tracing /tracing**

### 1.5.3 Function Tracing

After mounting tracefs you will have access to the control and output files of ftrace. Here is a list of some of the key files:

Note: all time values are in microseconds.

**current\_tracer**

This is used to set or display the current tracer that is configured. Changing the current tracer clears the ring buffer content as well as the “snapshot” buffer.

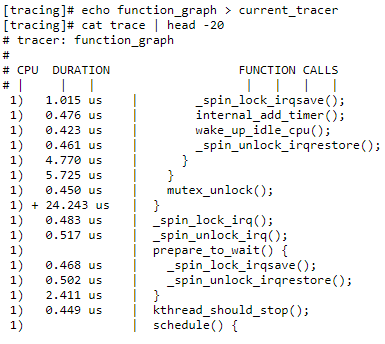


Figure : Use of current\_tracer

**available\_tracers**

This holds the different types of tracers that have been compiled into the kernel. The tracers listed here can be configured by echoing their name into current\_tracer.

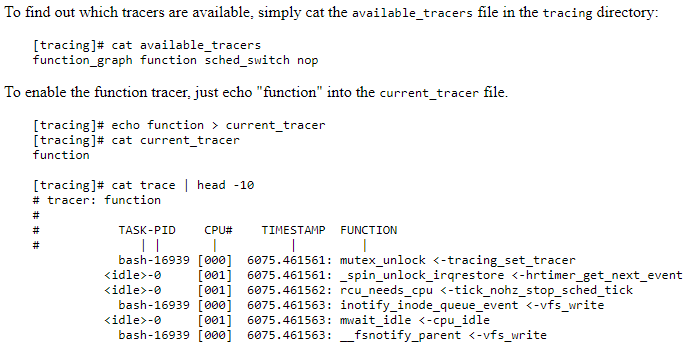


Figure : Use of available\_tracer

**tracing\_on**

This sets or displays whether writing to the trace ring buffer is enabled. Echo 0 into this file to disable the tracer or 1 to enable it. Note, this only disables writing to the ring buffer, the tracing overhead may still be occurring.

The kernel function tracing\_off () can be used within the kernel to disable writing to the ring buffer, which will set this file to “0”. User space can re-enable tracing by echoing “1” into the file.

* When available\_tracers command was executed it listed tracers we can use for debugging
* Current\_tracer shows which tracer is executing at that moment.
* Then we will disable tracing by putting ‘0’ in tracing\_on.
* Now we put function as current tracer.
* We will see whenever page fault occurs by putting do\_page\_fault in set\_ftrace\_filter.



Figure : Use of tracing\_on

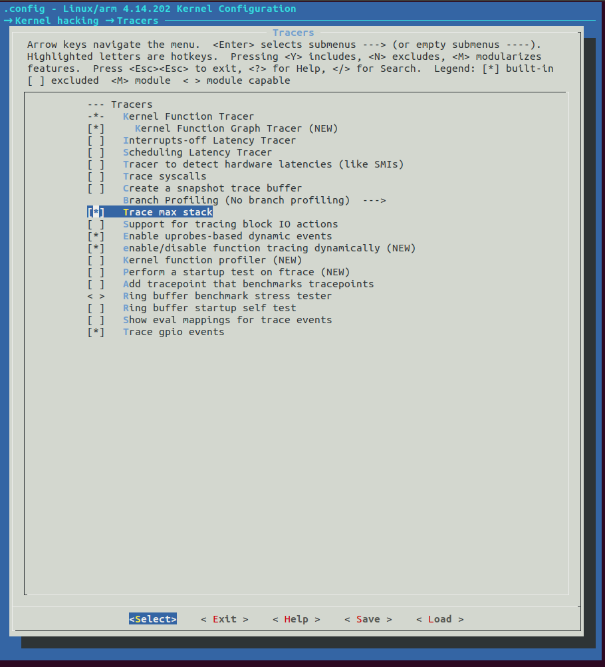


Figure : Various tracer selection in menuconfig

1. After selecting different tracer in the menuconfig. Now build the kernel again by using the command make **ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-zImage j6**
2. Now copy the new zImage and then launch qemu.
3. Check the important directory for ftrace by using the following command

**ls /sys/kernel/debug/tracing**

Use the command **cd /sys/kernel/debug/tracing** to move into the folder.

1. To see all the available and selected tracers in menuconfig use the command

**cat available\_tracers**

1. Then to see the current tracer use the command

**cat current\_tracer**

1. Use the command **echo nop > current\_trace** to stop the tracing.
2. Next use the command **echo 0 > tracing\_on** to stop and choose the tracing options
3. Next enable two tracers
   1. echo function > current\_tracer -> To enable the function tracer, just echo "function" into the current\_tracer file
   2. echo do\_page\_fault > set\_ftrace\_filter -> to trace all the page faults occurring by various processes.
4. Then we give **echo 1 > tracing\_on** to start the tracing again.
5. Now use the command cat trace to see all the tracing output for the different selected tracers.
6. We can also use the command **cat trace | tail –f trace** or **head** command to extract only a few traces.
7. Trace\_print k can be used to get all the printk traces and can be helpful in loading the number of times the printk was used.

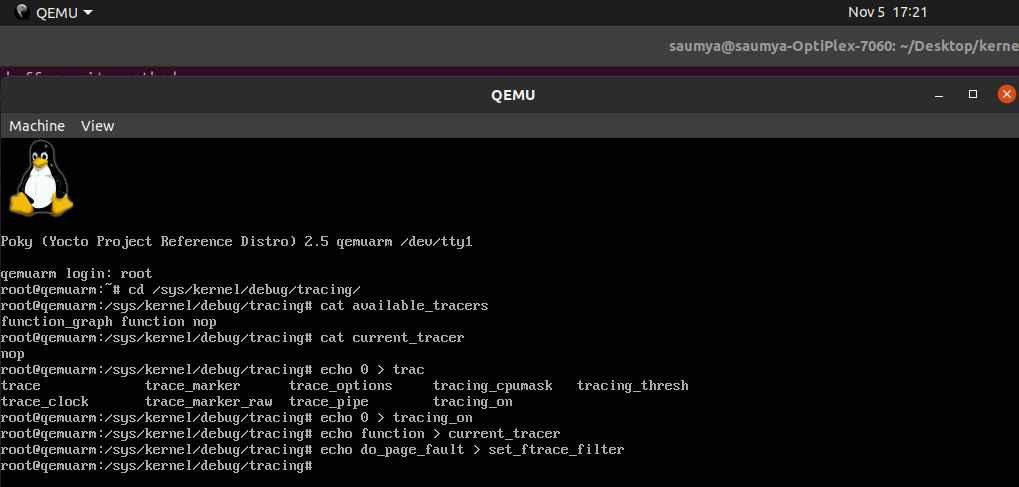


Figure 13: ftrace in Qemu

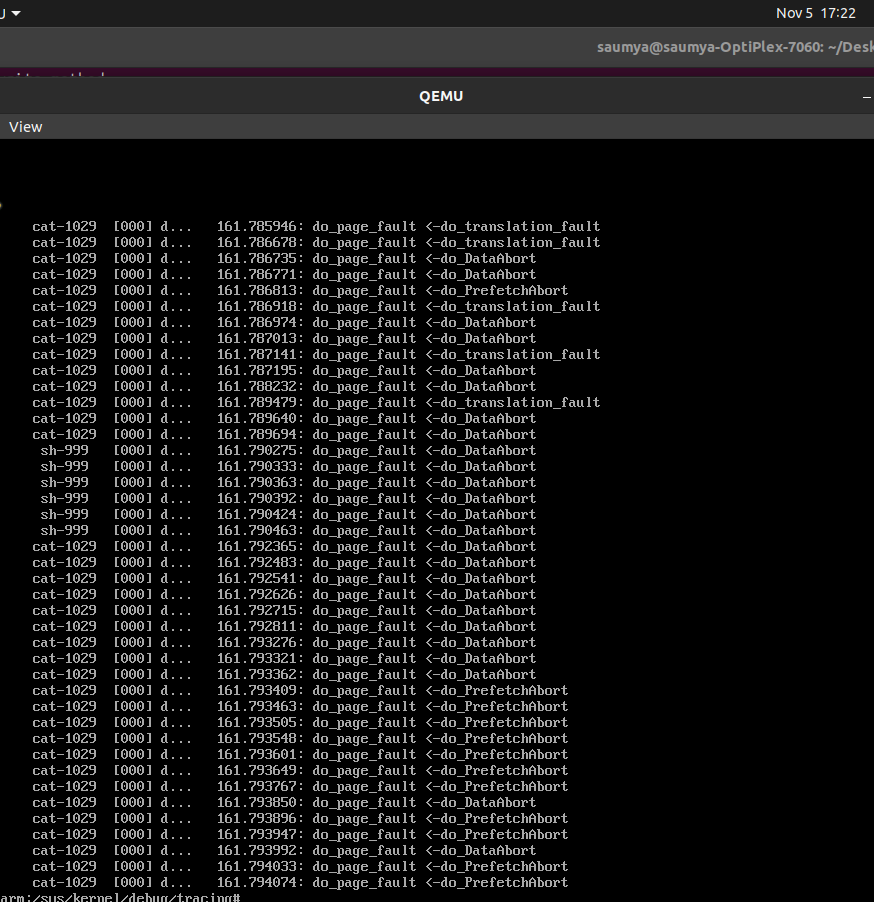


Figure 14: page fault tracing

### 1.5.4 Using Trace\_printk

printk() is the king of all debuggers, but it has a problem. If you are debugging a high volume area such as the timer interrupt, the scheduler, or the network, printk() can lead to bogging down the system or can even create a live lock. It is also quite common to see a bug "disappear" when adding a few printk()s. This is due to the sheer overhead that printk()introduces.

Ftrace introduces a new form of printk() called trace\_printk(). It can be used just like printk(), and can also be used in any context (interrupt code, NMI code, and scheduler code). What is nice about trace\_printk()is that it does not output to the console. Instead it writes to the Ftrace ring buffer and can be read via the trace file.

Writing into the ring buffer with trace\_printk() only takes around a tenth of a microsecond or so. But using printk(), especially when writing to the serial console, may take several milliseconds per write. The performance advantage of trace\_printk() lets you record the most sensitive areas of the kernel with very little impact.

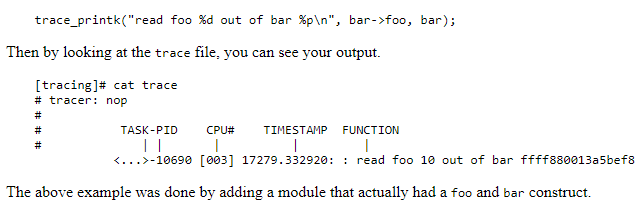


Figure 15: Use of trace\_printk

## 1.6 BUG\_ON

* BUG() and BUG\_ON(condition) are used as a debugging help when something in the kernel goes terribly wrong. When a BUG\_ON() assertion fails, or the code takes a branch with BUG() in it, the kernel will print out the contents of the registers and a stack trace. After that the current process will die.
* The following are examples of how BUG() and BUG\_ON() are used, from a piece of code that is not supposed to run in interrupt context. The explicit ifwith BUG() is the coding style used in older kernels. In the 2.6 kernel, generally BUG\_ON() is preferred.
* It is frequently found macro in Linux device drivers [5].
* Syntax for BUG\_ON is- **BUG\_ON(condition)**
* Functions of BUG\_ON are
  + It prints the contents of the registers.
  + It prints the Stack Trace.
  + The current process dies.

#include <linux/kernel.h>

#include <linux/module.h>

MODULE\_LICENSE("GPL");

static int test\_bug\_init(void)

{

printk(KERN\_INFO"%s: In init\n",\_\_func\_\_);

BUG(); //Bug is called here

return 0;

}

static void test\_bug\_exit(void)

{

printk(KERN\_INFO”%s:In exit\n”,\_\_func\_\_);

}

In the above example, we cannot rmmod module, it will say module in use.

## 1.7 Magic SysRQ

* Magic SysRq (Magic System Request) is a kernel hack that enables the kernel to listen to specific key presses and respond by calling a specific kernel function. Magic SysRq is activated via input from the keyboard or a serial line.
* It is a key combination understood by the Linux kernel, which allows the user to perform various low-level commands regardless of the system's state.
* The magic SysRq key is implemented as part of Linux’s keyboard driver – it will work if the Linux kernel is still running. Only a kernel panic should disable this key combination.
* It is often used to recover from freezes, or to reboot a computer without corrupting the filesystem.
* This key combination provides access to powerful features for software development and disaster recovery. In this sense, it can be considered a form of [escape sequence](https://en.wikipedia.org/wiki/Escape_sequence).
* To use this key combination, your Linux kernel must have been compiled with the CONFIG\_MAGIC\_SYSRQ compile option – most Linux distributions will have this enabled by default. Assuming it’s compiled into your kernel, it can be enabled or disabled on a running system by changing the value of **/proc/sys/kernel/sysrq**. To check if it’s enabled, run the following command:

We can set by **echo number > /proc/sys/kernel/sysrq**.

* **cat /proc/sys/kernel/sysrq**: If you see a “1”, all functions of the magic SysRq key are enabled. A greater number indicates only certain functions are enabled

Here is the list of possible values in /proc/sys/kernel/sysrq:

* + 0 - disable sysrq completely
  + 1 - enable all functions of sysrq
  + >1 - bitmask of allowed sysrq functions (see below for detailed function description):

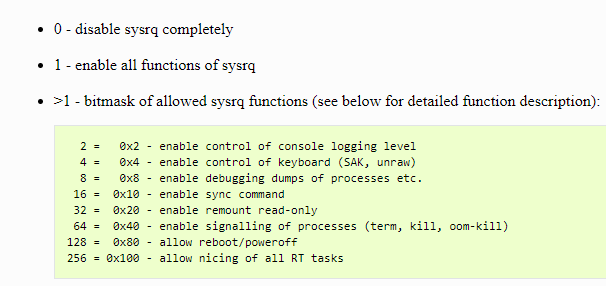


Figure 16:SysRq values

You can set the value in the file by the following command:

* **echo "number" >/proc/sys/kernel/sysrq**

## 1.8 GDB

* It is a debugger where we can load program and set the breakpoint to desired line number and start executing the code line by line.
* The first step of learning how to use GDB for C++ debugging is to compile the C++ code with the -g flag

Command- $ g++ - g filename.cpp

* The next step is calling the GDB to start the debugging process for the program you wish to analyze:

Command- $ gdb program\_name

* There are several ways of running the program you have opened. First, you can execute it with run command:

Command- $ (gdb) run

* You can pass arguments if the program needs some command-line arguments to be passed ti it:

Command- $ (gdb) run arg1 arg2

* Debugging with GDB lets you investigate the core file as well. The core file contains information about when a program crashed.

Command- $ (gdb) core filename

* Setting Breakpoint to stop the execution of the program.
* The following function sets a breakpoint at the start of the main:

Command- $ (gdb) b main

* This example sets a breakpoint at a specific line (20):

Command- $ (gdb) b 20

* Printing values of variables and expressions.

The possible syntax for using print:

* print <exp>: to get values of **expressions**.
* print /x <exp>: to get the value of expressions in **hexadecimal**.
* print /t <exp>: to get the value of expressions in **binary**.
* print /d <exp>: to get the value of expressions as **unsigned int format**.
* print /c <exp>: to get the value of expressions as **signed int format**.

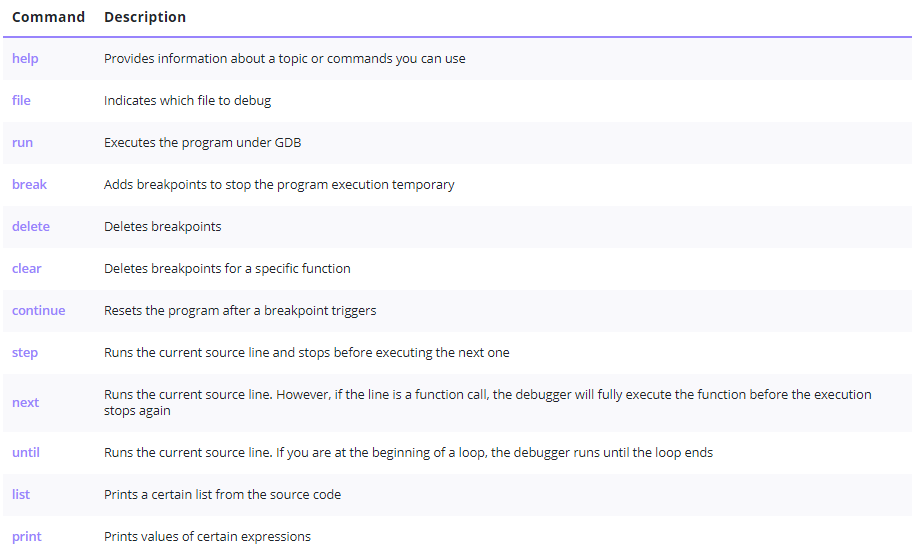


Figure : Commands used in GDB debugger

# 2. Advanced Drivers

## 2.1 ioctl() – control device

* The header file used is #include <sys/ioctl.h> for ioctl.
* Fuction prototype is - int ioctl (int fd, unsigned long request, ...);

**Description:**

ioctl() system call is the most common way for applications to interface with device drivers. It is flexible and easily extended by adding new commands and can be passed through character devices, block devices as well as sockets and other special file descriptors.

The **ioctl**() function works for the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() requests. The argument *d* must be an open file descriptor.

The second argument is a device-dependent request code.

* + A Magic number - 8 bits
  + A sequence number - 8 bits
  + Argument type (typically 14 bits), if any.
  + Direction of data transfer (2 bits).

The third argument is an untyped pointer to memory.

An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl() request are located in the file <sys/ioctl.h>.

**Return Value:**

Usually, on success zero is returned. A few ioctl() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and errno is set apropriately.

**Errors:**

**EBADF**

*d* is not a valid descriptor.

**EFAULT**

*argp* references an inaccessible memory area.

**EINVAL**

*Request* or *argp* is not valid.

**ENOTTY**

*d* is not associated with a character special device.

**ENOTTY**

The specified request does not apply to the kind of object that the descriptor *d* references.

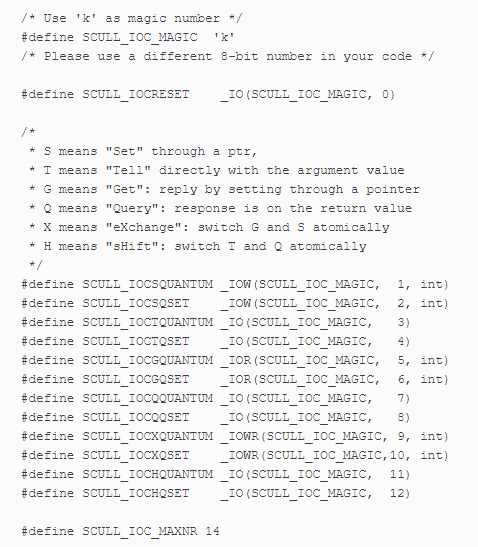


Figure : ioctl commands defined in scull

## 2.2 Driver Model

The Linux Kernel Driver Model is a unification of all the disparate driver models that were previously used in the kernel. It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures.

Traditional driver models implemented some sort of tree-like structure (sometimes just a list) for the devices they control. There wasn’t any uniformity across the different bus types.

The current driver model provides a common, uniform data model for describing a bus and the devices that can appear under the bus. The unified bus model includes a set of common attributes which all busses carry, and a set of common callbacks, such as device discovery during bus probing, bus shutdown, bus power management, etc.

### 2.2.1 Platform Devices

* Platform devices are devices that typically appear as autonomous entities in the system.
* port-based devices and host bridges to peripheral buses, and most controllers integrated into system-on-chip platforms.
* Example platform device is shown below

struct platform\_device

{

const char \*name;

u32 id;

struct device dev;

u32 num\_resources;

struct resource \*resource;

};

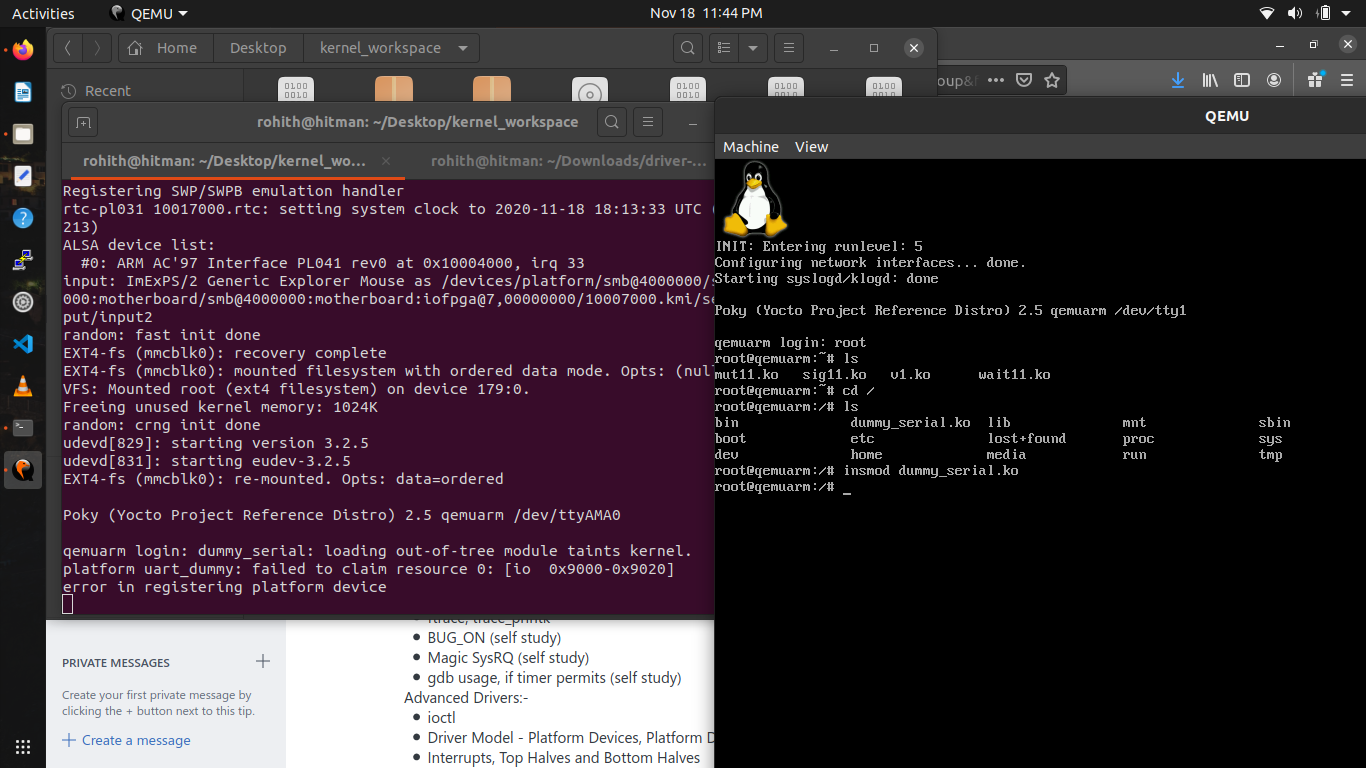


Figure : Platform Devices

### 2.2.2 Platform Drivers

* Platform drivers follow the standard driver model convention, but handled outside the drivers.
* These drivers provide probe() and remove() methods.
* Example for platform driver is shown below
* These drivers support power management and power off the system.

struct platform\_driver

{

int (\*probe)(struct platform\_device \*);

int (\*remove)(struct platform\_device \*);

void (\*shutdown)(struct platform\_device \*);

int (\*suspend)(struct platform\_device \*, pm\_message\_t state);

int (\*suspend\_late)(struct platform\_device \*, pm\_message\_t state);

int (\*resume\_early)(struct platform\_device \*);

int (\*resume)(struct platform\_device \*);

struct device\_driver driver;

};

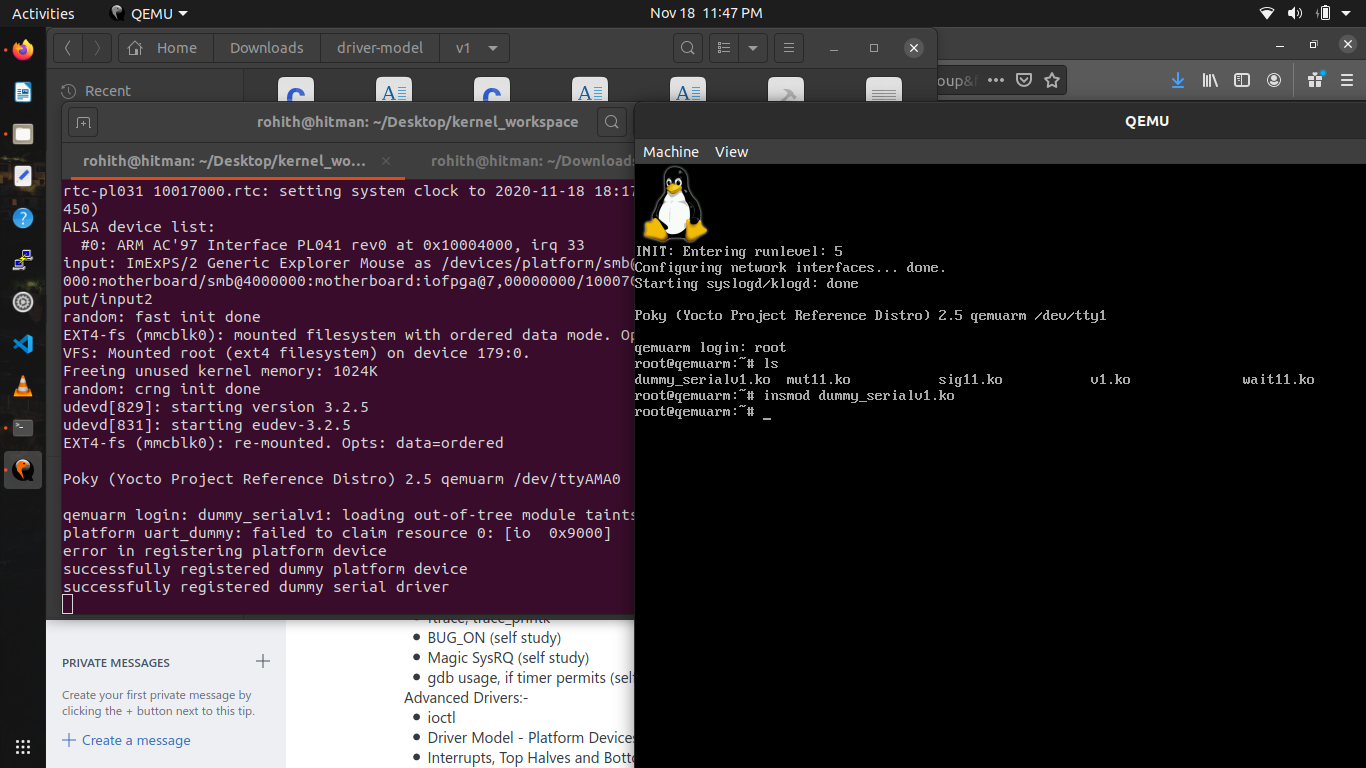


Figure : Platform drivers and device in same .c file

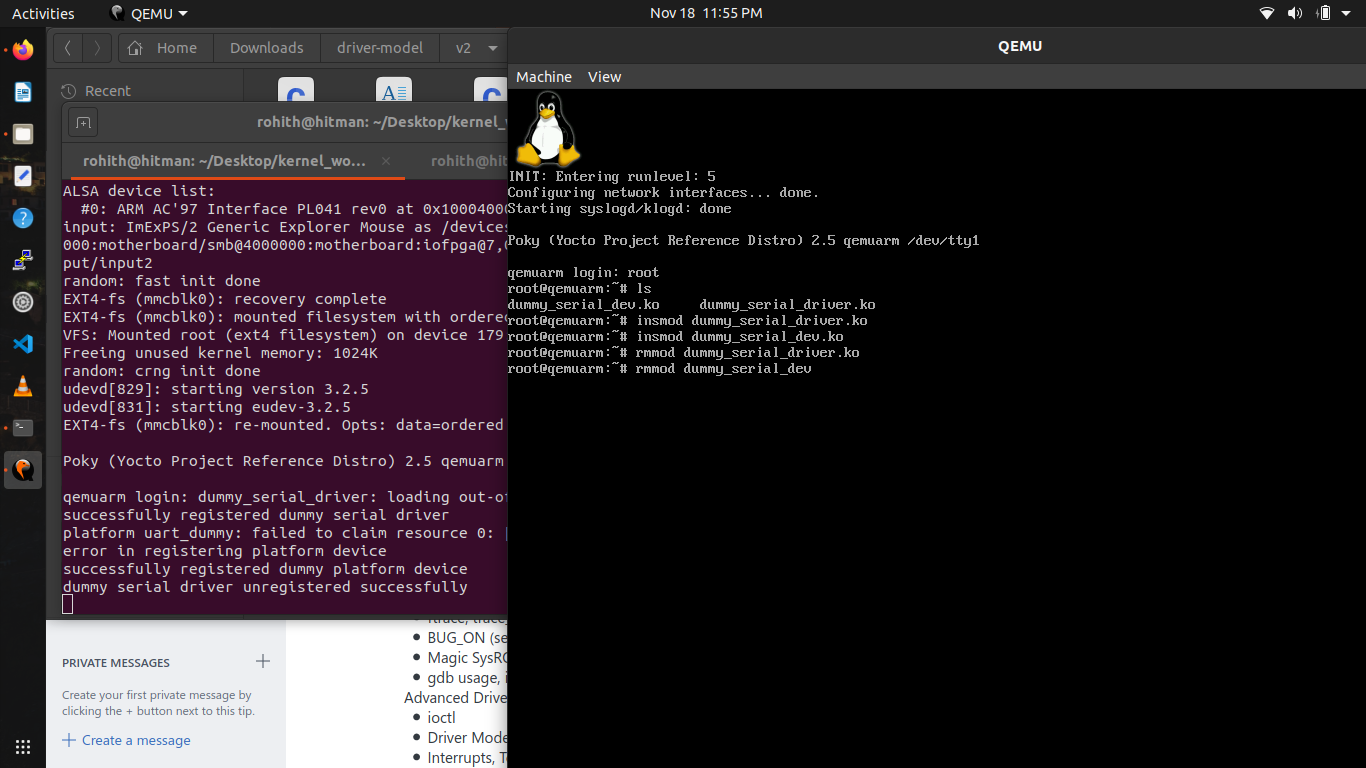


Figure : Platform drivers and devices in separate .c files

### 2.2.3 Device Tree Mapping

* Here we add our own dummy device in dts file located in folder arch/arm/boot/dts and choose vexpress-v2p-ca9.dtb file.
* Below code snippet must be added to vexpress dtb file.

dummyserial@9000

{

compatible = "arm,myuart", "arm,dummyserial";

reg = <0x9000, 0x20>

interrupts = <5>

};

* Then compile the kernel with updated dtb file using following command
  + Make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- dtbs
* Copy the newly generated vexpress dtb file to workspace where you are running QEMU.
* Now copy ko file generated from version 3 of dummy device code to QEMU.
* Now launch QEMU and insert this dummy\_serial module.

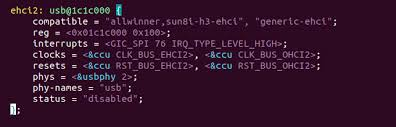


Figure : Linux Device Tree for Hardware mapping

## 2.3 Interrupts

* An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself.
* Interrupts can be grouped into two categories based on the source of the interrupt:
  + synchronous, generated by executing an instruction
  + asynchronous, generated by an external event
  + maskable
    - can be ignored.
    - signalled via INT pin.
  + non-maskable
    - cannot be ignored.
    - signalled via NMI pin.
* In case of a process context, there is a stack hence we can preempt current process and run higher priority tasks.
* Preemption or blocking of a process is allowed.
* In case of interrupts contexts there is no stack save the current contents of the process, hence we cannot preempt interrupt context.
* Interrupt context is time critical because the interrupt handler interrupts other code, it should be quick and simple and try to avoid busy looping.
* Interrupt handler does not have stack but they will share stack of process they interrupted.

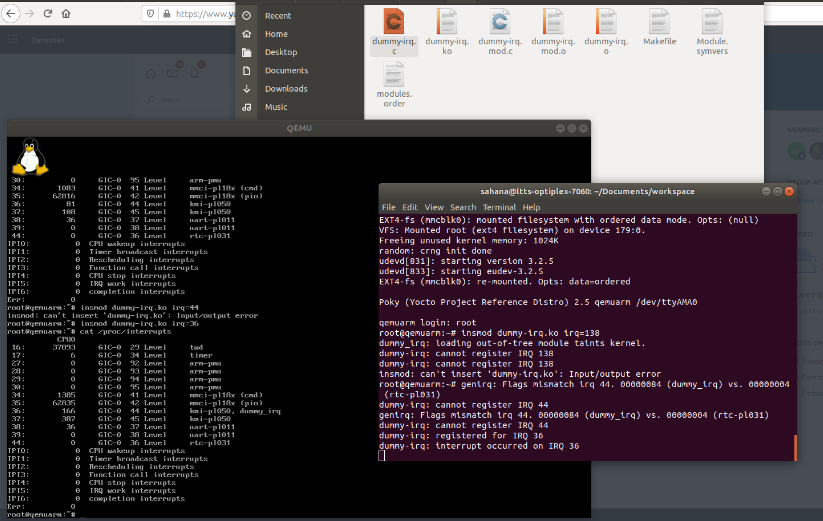


Figure : Dummy IRQ

## 2.4 Top Halves and Bottom Halves

**TOP HALVES**

Top halves executes as soon as CPU receives the interrupt. In the Top Halves Context Interrupt and Scheduler are disabled. This part of the code only contain Critical Code. Execution Time of this code should be as short as possible because now Interrupt are disabled, we don't want to miss out other interrupt by generated by the devices.

* Workflow of top halves –
  + Acknowledgement of receiving the interrupt.
  + copy the data received.
  + if the work is sensitive needs to perform in top halves.
  + If the work is related to hardware needs to perform in top halves.
  + If the work needs to be ensure that another interrupt does not interrupt it, should be perform in interrupt handler.

**BOTTOM HALVES**

The Job of the Bottom half is used to run left over (deferred) work by the top halves. When this piece of code is being Executed interrupt is Enabled and Scheduler is Disabled. Bottom Halves are scheduled by Softirqs & Tasklets to run deferred work.

* There are different types of bottom halves, they are
  + softirq
* runs in interrupt context
* statically allocated
* same handler may run in parallel on multiple cores

**Need for Top and Bottom Halves:**

* Interrupt handler runs asynchronously by interrupting the other code.
* All interrupt on the current processor disabled.
* Interrupts are often time critical as they deal with hardware.
* We cannot block interrupt handler as they run in interrupt context.
* Interrupt handling is divided into two parts
  + Top Half: - It is executed as immediate response to interrupt.
  + Bottom Half: - It is executed some time later when CPU get free time.

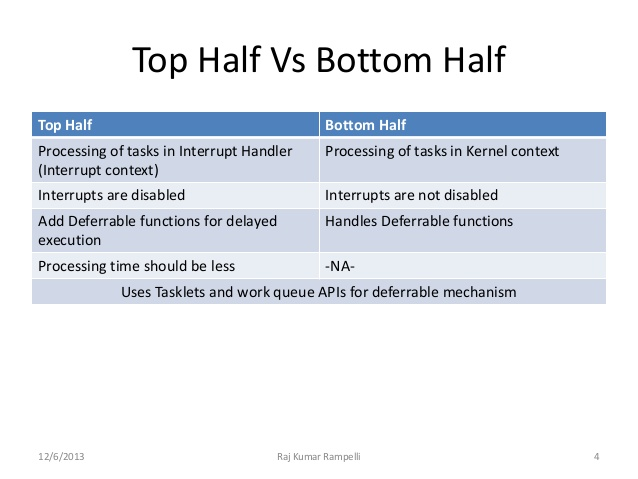


Figure : Difference between top and bottom halves

# References

|  |  |
| --- | --- |
|  | * https://linuxconfig.org/introduction-to-the-linux-kernel-log-levels. |
|  | * https://www.kernel.org/doc/html/v4.10/process/coding-style.html. |
|  | * https://www.linuxjournal.com/content/oops-debugging-kernel-panics-0. |
|  | * https://lwn.net/Articles/365835/. |
|  | * http://embeddedguruji.blogspot.com/2018/12/bugon-vs-warnon-macros-in-linux-kernel.html. |
|  | * https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html. |
|  | * https://www.tutorialspoint.com/gnu\_debugger/. * https://linux.die.net/man/2/ioctl * http://www.bo-yang.net/2015/01/12/linux-system-log * https://elinux.org/Ftrace * https://www.kernel.org/doc/html/latest/driver-api/driver-model/platform.html |