



kaiwan**TECH**

# ***KERNEL-SPACE CODE-BASED DEBUGGING TECHNIQUES***

## ***USING THE PRINTK, DEBUGFS AND IOCTL FOR DEBUGGING***

## Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the [permissive MIT license](#).

Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

**VERY IMPORTANT ::** Before using this source(s) in your project(s), you **\*MUST\*** check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are **\*not\*** under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Kaiwan N Billimoria  
kaiwanTECH, Bangalore, India.

<b><i>kaiwanTECH Linux OS Corporate Training Programs</i></b>
<i>Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs <a href="#">here</a>.</i>

<< Source: “Linux Device Drivers” by J Corbet, A Rubini & GK Hartman, 3<sup>rd</sup> Ed., O'Reilly >>

## Instrumentation - Using printk for Debugging

The most common debugging technique is monitoring, which in applications programming is done by calling printf at suitable points. When you are debugging kernel code, you can accomplish the same goal with **printk**.

We used the printk function in earlier chapters with the simplifying assumption that it works like printf. Now it's time to introduce some of the differences. One of the differences is that **printk lets you classify messages according to their severity by associating different loglevels**, or priorities, with the messages. You usually indicate the loglevel with a macro. For example, KERN\_INFO, which we saw prepended to some of the earlier print statements, is one of the possible loglevels of the message. The loglevel macro expands to a string, which is concatenated to the message text at compile time; that's why there is no comma between the priority and the format string in the following examples. Here are two examples of printk commands, a debug message and a critical message:

```
#define pr_fmt(fmt) "%s:%s(): " fmt, KBUILD_MODNAME, __func__
...
pr_debug("Here I am: %s:%s:%i\n", __FILE__, __func__, __LINE__);
pr_crit("I'm trashed; giving up on %p\n", ptr);
```

There are **eight possible loglevel strings**, defined in the header `<linux/kernel.h>`; we list them in order of **decreasing severity**:

### KERN\_EMERG

Used for emergency messages, usually those that precede a crash.

### KERN\_ALERT

A situation requiring immediate action.

### KERN\_CRIT

Critical conditions, often related to serious hardware or software failures.

### KERN\_ERR

Used to report error conditions; device drivers often use KERN\_ERR to report hardware difficulties.

### KERN\_WARNING

Warnings about problematic situations that do not, in themselves, create serious problems with the system.

### KERN\_NOTICE

Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

### KERN\_INFO

Informational messages. Many drivers print information about the hardware they find at startup time at this level.

### KERN\_DEBUG

Used for debugging messages.

Each string (in the macro expansion) represents an integer in angle brackets. Integers range from 0 to 7, with smaller values representing higher priorities.

&lt;&lt;

From [include/linux/kern-levels.h](#)*--snip--*

```
#define KERN_SOH      "\001"      /* ASCII Start Of Header */
#define KERN_SOH_ASCII '\001'

#define KERN_EMERG    KERN_SOH "0" /* system is unusable */
#define KERN_ALERT    KERN_SOH "1" /* action must be taken immediately */
#define KERN_CRIT     KERN_SOH "2" /* critical conditions */
#define KERN_ERR      KERN_SOH "3" /* error conditions */
#define KERN_WARNING  KERN_SOH "4" /* warning conditions */
#define KERN_NOTICE   KERN_SOH "5" /* normal but significant condition */
#define KERN_INFO     KERN_SOH "6" /* informational */
#define KERN_DEBUG    KERN_SOH "7" /* debug-level messages */
```

*--snip--*

**Convenience macros: use `thr pr_foo()` in place of `printk(KERN_F00 ...)`**  
[include/linux/printk.h](#)

```
...
/*
 * These can be used to print at the various log levels.
 * All of these will print unconditionally, although note that pr_debug()
 * and other debug macros are compiled out unless either DEBUG is defined
 * or CONFIG_DYNAMIC_DEBUG is set.
 */
#define pr_emerg(fmt, ...) \
    printk(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)
#define pr_alert(fmt, ...) \
    printk(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_crit(fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_err(fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warning(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn pr_warning
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
...

/* If you are writing a driver, please use dev_dbg instead */
...
```

Drivers:

dev\_foo() macros to be used !

&gt;&gt;

A `printk` statement with no specified priority defaults to `DEFAULT_MESSAGE_LOGLEVEL`, specified in `kernel/printk.c` as an integer. In the 2.6.10 kernel, `DEFAULT_MESSAGE_LOGLEVEL` is `KERN_WARNING`, but that has been known to change in the past.

Based on the loglevel, the kernel may print the message to the **current console**, be it a text-mode terminal, a serial port, or a parallel printer. If the priority is less than the integer variable `console_loglevel`, the message is delivered to the console one line at a time (nothing is sent unless a trailing newline is provided). If both `klogd` and `syslogd` are running on the system (the default), kernel messages are **appended to `/var/log/messages`** (or otherwise treated depending on your syslogd configuration; << often it's `/var/log/syslog` >>), independent of `console_loglevel`. If `klogd` is not running, the message won't reach user space unless you read `/proc/kmsg` (which is often most easily done with the `dmesg` command). When using `klogd`, you should remember that it doesn't save consecutive identical lines; it only saves the first such line and, at a later time, the number of repetitions it received.

The variable `console_loglevel` is initialized to `DEFAULT_CONSOLE_LOGLEVEL` and can be modified through the `sys_syslog` system call. One way to change it is by specifying the `-c` switch when invoking `klogd`, as specified in the `klogd` manpage. Note that to change the current value, you must first kill `klogd` and then restart it with the `-c` option. Alternatively, you can write a program to change the console loglevel. You'll find a version of such a program in `misc-progs/setlevel.c` in the source files provided on O'Reilly's FTP site. The new level is specified as an integer value between 1 and 8, inclusive. If it is set to 1, only messages of level 0 (`KERN_EMERG`) reach the console; if it is set to 8, all messages, including debugging ones, are displayed.

It is also possible to read and modify the console loglevel using the text file `/proc/sys/kernel/printk`.

The file hosts four integer values: the current loglevel, the default level for messages that lack an explicit loglevel, the minimum allowed loglevel, and the boot-time default loglevel.

```
# cat /proc/sys/kernel/printk
4      4      1      7
```

Writing a single value to this file changes the *current loglevel* to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

```
$ sudo sh -c "echo \"8 4 1 7\" > /proc/sys/kernel/printk "
```

It should now be apparent why the `hello.c` sample had the `pr_alert()`; they are there to make sure that the messages appear on the console.

### Additional Notes

1. The `printk` function operates by trying to grab the console semaphore, place the output into the console's log buffer, and then call the console driver to flush the buffer. If `printk` cannot grab

the console semaphore, it places the output into the log buffer and relies on the process that has the console semaphore to flush the buffer. The log-buffer lock is taken before *printk* places any data into the log buffer, so concurrent calls to *printk* do not trample each other. If the console semaphore is being held, numerous calls to *printk* can occur before the log buffer is flushed. **So, do not rely on the *printk* statements to indicate any program timing.**

## 2. Kernel command-line parameter 'ignore\_loglevel' (see [Documentation/kernel-parameters.txt](#))

`ignore_loglevel [KNL]`

Ignore loglevel setting - this will print /all/  
kernel messages to the console. Useful for debugging.

## How Messages Get Logged

The *printk* function writes messages into a circular buffer that is `__LOG_BUF_LEN` bytes long: a value from 4 KB to 1 MB chosen while configuring the kernel. The function then wakes any process that is waiting for messages, that is, any process that is sleeping in the `syslog` system call or that is reading `/proc/kmsg`. These two interfaces to the logging engine are almost equivalent, but note that reading from `/proc/kmsg` consumes the data from the log buffer, whereas the `syslog` system call can optionally return log data while leaving it for other processes as well. In general, reading the `/proc` file is easier and is the default behavior for `klogd`. The `dmesg` command can be used to look at the content of the buffer without flushing it; actually, the command returns to `stdout` the whole content of the buffer, whether or not it has already been read.

If you happen to read the kernel messages by hand, after stopping `klogd`, you'll find that the `/proc` file looks like a FIFO, in that the reader blocks, waiting for more data. Obviously, you can't read messages this way if `klogd` or another process is already reading the same data, because you'll contend for it.

If the circular buffer **fills up, *printk* wraps around** and starts adding new data to the beginning of the buffer, overwriting the oldest data. **Therefore, the logging process loses the oldest data.** This problem is negligible compared with the advantages of using such a circular buffer. For example, a circular buffer allows the system to run even without a logging process, while minimizing memory waste by overwriting old data should nobody read it. Another feature of the Linux approach to messaging is that *printk* can be invoked from anywhere, even from an interrupt handler, with no limit on how much data can be printed. The only disadvantage is the possibility of losing some data.

If the `klogd` process is running, it retrieves kernel messages and dispatches them to `syslogd`, which in turn checks `/etc/syslog.conf` to find out how to deal with them. `syslogd` differentiates between messages according to a facility and a priority; allowable values for both the facility and the priority are defined in `<sys/syslog.h>`. Kernel messages are logged by the `LOG_KERN` facility at a priority corresponding to the one used in *printk* (for example, `LOG_ERR` is used for `KERN_ERR` messages). If `klogd` isn't running, data remains in the circular buffer until someone reads it or the buffer overflows.

If you want to avoid clobbering your system log with the monitoring messages from your driver, you can either specify the `-f` (file) option to `klogd` to instruct it to save messages to a specific file, or customize `/etc/syslog.conf` to suit your needs. Yet another possibility is to take the brute-force approach: kill `klogd` and



verbosely print messages on an unused virtual terminal,\* or issue the command `cat /proc/kmsg` from an unused xterm.

\* For example, use `setlevel 8`; `setconsole 10` to set up terminal 10 to display messages.

## Turning printk's On and Off

During the early stages of driver development, `printk` can help considerably in debugging and testing new code. When you officially release the driver, on the other hand, you should remove, or at least disable, such print statements. Unfortunately, you're likely to find that as soon as you think you no longer need the messages and remove them, you implement a new feature in the driver (or somebody finds a bug), and you want to turn at least one of the messages back on. There are several ways to solve both issues, to globally enable or disable your debug messages and to turn individual messages on or off.

Here we show one way to code `printk` calls so you can turn them on and off individually or globally; the technique depends on defining a macro that resolves to a `printk` (or `printf`) call when you want it to:

...

<<

### **NOTE! UPDATE-**

*With the modern kernel, these home-grown `printk`-based macros become mostly unnecessary. The kernel now has the powerful dynamic debug facility (soon to be discussed!) which makes stuff like this unnecessary.*

*Nevertheless, you might find some of them – like the `QP*()` ones – useful in some situations.*

From my convenient header file, named, (no prizes for guessing), "`convenient.h`" :

```
...
#ifdef DEBUG
#ifdef __KERNEL__
#define MSG(string, args...) \
    pr_info("%s:%d : " string, __func__, __LINE__, ##args)
#else
#define MSG(string, args...) \
    fprintf(stderr, "%s:%d : " string, __func__, __LINE__, ##args)
#endif

#ifdef __KERNEL__
#define MSG_SHORT(string, args...) \
    pr_info(string, ##args)
#else
#define MSG_SHORT(string, args...) \
    fprintf(stderr, string, ##args)
#endif
#endif
```

```

#define QP MSG("\n")

#ifdef __KERNEL__
#define QPDS do { \
    MSG("\n"); \
    dump_stack(); \
} while(0)
#define PRCS_CTX do { \
    if (in_task()) { \
        MSG("prcs ctx: %s(%d)\n", current->comm, current->pid); \
    } \
    else { \
        MSG("irq ctx\n"); \
        PRINT_IRQCTX(); \
    } \
} while(0)
#endif

#ifdef __KERNEL__
#define HexDump(from_addr, len) \
    print_hex_dump_bytes (" ", DUMP_PREFIX_ADDRESS, from_addr, len);
#endif
#else
#define MSG(string, args...)
#define MSG_SHORT(string, args...)
#define QP
#define QPDS
#endif

...

>>

<<

```

### ***Dumping the Kernel-Mode Stack***

The kernel supplies an architecture-independent function for viewing the current contents of the kernel-mode stack (of the task in context):

```
void dump_stack(void);
```

This can indeed prove very useful when debugging kernel code- it's the equivalent of the 'backtrace' (or bt) gdb command.

**But what if we see only hex addresses in the stack and no symbolic names?**

*Scenario:*

Attempting to debug on an ARM-based system - the kernel hangs at a point during boot; we put in some printk's as well as a



**dump\_stack();**  
call.

The output, however, looks like this:

```
...
smc95xx v1.0.4
smc95xx 1-1.1:1.0: eth0: register 'smc95xx' at usb-bcm2708_usb-1.1, smc95xx USB
2.0 Ethernet, b8:27:eb:86:e1:c5
PRINT_CTX:: in function usbnet_probe on cpu # 0
in process context: khubd:13
Backtrace:                                     <-- o/p from dump_stack()
Function entered at [<c0011b50>] from [<c036eb48>]
r6:db9f5ba0 r5:db9dcc00 r4:db9f5800 r3:c042582c
Function entered at [<c036eb30>] from [<c02509ec>]
Function entered at [<c0250458>] from [<c026050c>]
Function entered at [<c0260424>] from [<c021b734>]
Function entered at [<c021b6c0>] from [<c021b9a4>]
r7:db99d868 r6:c021b95c r5:db9dcc20 r4:c043bdc4
Function entered at [<c021b95c>] from [<c0219cd0>]
r5:db9dcc20 r4:00000000
Function entered at [<c0219c84>] from [<c021b688>]
r6:c043bffc r5:db9dcc54 r4:db9dcc20
Function entered at [<c021b60c>] from [<c021ac80>]
r6:c043bffc r5:db9dcc20 r4:db9dcc20 r3:db861520
Function entered at [<c021abf4>] from [<c0219364>]
r6:db9dcc28 r5:00000000 r4:db9dcc20 r3:00000000
Function entered at [<c0218e7c>] from [<c025f010>]
Function entered at [<c025eb14>] from [<c0267868>]
Function entered at [<c026781c>] from [<c025f528>]
r5:db99d868 r4:c04b6f14
Function entered at [<c025f508>] from [<c021b734>]
Function entered at [<c021b6c0>] from [<c021b9a4>]
r7:db966468 r6:c021b95c r5:db99d868 r4:c043c5a4
Function entered at [<c021b95c>] from [<c0219cd0>]
r5:db99d868 r4:00000000
Function entered at [<c0219c84>] from [<c021b688>]
r6:c043bffc r5:db99d89c r4:db99d868
Function entered at [<c021b60c>] from [<c021ac80>]
r6:c043bffc r5:db99d868 r4:db99d868 r3:db861520
Function entered at [<c021abf4>] from [<c0219364>]
r6:db99d870 r5:00000000 r4:db99d868 r3:00000000
Function entered at [<c0218e7c>] from [<c0257c1c>]
Function entered at [<c0257ae4>] from [<c0258714>]
r7:db88a000 r6:db966400 r5:00000000 r4:db99d800
Function entered at [<c0257c98>] from [<c003a9e8>]
Function entered at [<c003a958>] from [<c00222ec>]
r6:c00222ec r5:c003a958 r4:db82df0c
<hanging here>
```

How do we get a meaningful stack dump with symbolic names?

*Kernel config:*

make menuconfig

General Setup /

- \*- Configure standard kernel features (expert users) ---&gt;

[\*] Load all symbols for debugging/ksymoops

&lt;- CONFIG\_KALLSYMS

[\*] Include all symbols in kallsyms

&lt;- CONFIG\_KALLSYMS\_ALL

**Turn on** these config options - viz.,**CONFIG\_KALLSYMS and CONFIG\_KALLSYMS\_ALL**

&lt;&lt;

Find the options here:

.config - Linux/x86 3.14.34 Kernel Configuration

> *General setup* > *Configure standard kernel features (expert users)*

Load all symbols for debugging/ksymoops

CONFIG\_KALLSYMS:

Say Y here to let the kernel print out symbolic crash information and symbolic stack backtraces. This

increases the size of the kernel somewhat, as all symbols have to be loaded into the kernel image.

And

CONFIG\_KALLSYMS\_ALL:

Normally kallsyms only contains the symbols of functions for nicer OOPS messages and backtraces (i.e., symbols from the text and inittext sections). This is sufficient for most cases. And only in very rare cases (e.g., when a debugger is used) all symbols are required (e.g., names of variables from the data sections, etc).

This option makes sure that all symbols are loaded into the kernel image (i.e., symbols from all sections) in cost of increased kernel size (depending on the kernel configuration, it may be 300KiB or something like this).

Say N unless you really need all symbols.

&gt;&gt;

Rebuild; now the kernel output looks like this; much better!

...

smc95xx v1.0.4

smc95xx 1-1.1:1.0: eth0: register 'smc95xx' at usb-bcm2708\_usb-1.1, smc95xx USB 2.0 Ethernet, b8:27:eb:86:e1:c5

**PRINT\_CTX:: in function usbnet\_probe on cpu # 0**

in process context: khubd:13

**Backtrace:**[<c0011b78>] (dump\_backtrace+0x0/0x10c) from [<c0370cb4>] (dump\_stack+0x18/0x1c)  
r6:db9f4ba0 r5:db9dec00 r4:db9f4800 r3:c04ad82c

[&lt;c0370c9c&gt;] (dump\_stack+0x0/0x1c) from [&lt;c0252b58&gt;] (usbnet\_probe+0x594/0x6b4)

[&lt;c02525c4&gt;] (usbnet\_probe+0x0/0x6b4) from [&lt;c0262678&gt;]

(usb\_probe\_interface+0xe8/0x1b8)

[&lt;c0262590&gt;] (usb\_probe\_interface+0x0/0x1b8) from [&lt;c021d8a0&gt;]

```

(driver_probe_device+0x74/0x204)
[<c021d82c>] (driver_probe_device+0x0/0x204) from [<c021db10>]
(__device_attach+0x48/0x4c)
r7:db99f868 r6:c021dac8 r5:db9dec20 r4:c04c3dc4
[<c021dac8>] (__device_attach+0x0/0x4c) from [<c021be3c>]
(bus_for_each_drv+0x4c/0x94)
r5:db9dec20 r4:00000000
[<c021bdf0>] (bus_for_each_drv+0x0/0x94) from [<c021d7f4>]
(device_attach+0x7c/0x88)
r6:c04c3ffc r5:db9dec54 r4:db9dec20
[<c021d778>] (device_attach+0x0/0x88) from [<c021cdec>]
(bus_probe_device+0x8c/0xb4)
r6:c04c3ffc r5:db9dec20 r4:db9dec20 r3:db861520
[<c021cd60>] (bus_probe_device+0x0/0xb4) from [<c021b4d0>] (device_add+0x4e8/0x5b4)
r6:db9dec28 r5:00000000 r4:db9dec20 r3:00000000
[<c021afe8>] (device_add+0x0/0x5b4) from [<c026117c>]
(usb_set_configuration+0x4fc/0x940)
[<c0260c80>] (usb_set_configuration+0x0/0x940) from [<c02699d4>]
(generic_probe+0x4c/0x90)
[<c0269988>] (generic_probe+0x0/0x90) from [<c0261694>]
(usb_probe_device+0x20/0x24)
r5:db99f868 r4:c053f0f4
[<c0261674>] (usb_probe_device+0x0/0x24) from [<c021d8a0>]
(driver_probe_device+0x74/0x204)
[<c021d82c>] (driver_probe_device+0x0/0x204) from [<c021db10>]
(__device_attach+0x48/0x4c)
r7:db964468 r6:c021dac8 r5:db99f868 r4:c04c45a4
[<c021dac8>] (__device_attach+0x0/0x4c) from [<c021be3c>]
(bus_for_each_drv+0x4c/0x94)
r5:db99f868 r4:00000000
[<c021bdf0>] (bus_for_each_drv+0x0/0x94) from [<c021d7f4>]
(device_attach+0x7c/0x88)
r6:c04c3ffc r5:db99f89c r4:db99f868
[<c021d778>] (device_attach+0x0/0x88) from [<c021cdec>]
(bus_probe_device+0x8c/0xb4)
r6:c04c3ffc r5:db99f868 r4:db99f868 r3:db861520
[<c021cd60>] (bus_probe_device+0x0/0xb4) from [<c021b4d0>] (device_add+0x4e8/0x5b4)
r6:db99f870 r5:00000000 r4:db99f868 r3:00000000
[<c021afe8>] (device_add+0x0/0x5b4) from [<c0259d88>] (usb_new_device+0x138/0x1b4)
[<c0259c50>] (usb_new_device+0x0/0x1b4) from [<c025a880>] (hub_thread+0xa7c/0x11b4)
r7:db88a000 r6:db964400 r5:00000000 r4:db99f800
[<c0259e04>] (hub_thread+0x0/0x11b4) from [<c003aa14>] (kthread+0x90/0x9c)
[<c003a984>] (kthread+0x0/0x9c) from [<c002230c>] (do_exit+0x0/0x73c)
r6:c002230c r5:c003a984 r4:db82df0c

```

>>

To simplify the process of adding these debug macros to your code further, add the following lines to your *Makefile*:

```

# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

```

```

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
    DBGFLAGS = -Og -g -DDEBUG    # "-O<n>" is needed to expand inlines
else
    DBGFLAGS = -O2
endif

CFLAGS += $(DBGFLAGS)

<<
More recent- from my LKD book:

...
MYDEBUG := n
ifeq (${MYDEBUG}, y)

# https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#compilation-flags
# EXTRA_CFLAGS deprecated; use ccflags-y
    ccflags-y += -DDEBUG -ggdb -gdwarf-4 -Og -Wall -fno-omit-frame-pointer -fvar-
tracking-assignments
    # man gcc: "...-Og may result in a better debugging experience"
else
    INSTALL_MOD_STRIP := 1
endif
...
>>

```

The macros shown in this section depend on a *gcc* extension to the ANSI C preprocessor that supports macros with a variable number of arguments. This *gcc* dependency shouldn't be a problem, because the kernel proper depends heavily on *gcc* features anyway. In addition, the makefile depends on GNU's version of *make*; once again, the kernel already depends on GNU *make*, so this dependency is not a problem. If you're familiar with the C preprocessor, you can expand on the given definitions to implement the concept of a "debug level," defining different levels and assigning an integer (or bit mask) value to each level to determine how verbose it should be.

But every driver has its own features and monitoring needs. The art of good programming is in choosing the best trade-off between flexibility and efficiency, and we can't tell what is the best for you. Remember that preprocessor conditionals (as well as constant expressions in the code) are executed at compile time, so you must recompile to turn messages on or off. A possible alternative is to use C conditionals, which are executed at runtime and, therefore, permit you to turn messaging on and off during program execution. This is a nice feature, but it requires additional processing every time the code is executed, which can affect performance even when the messages are disabled. Sometimes this performance hit is unacceptable.

The macros shown in this section have proven themselves useful in a number of situations, with the only disadvantage being the requirement to recompile a module after any changes to its messages.

&lt;&lt;

**Useful: a standard template for all your printk's ?**

An extract from my book: [Linux Kernel Programming, Packt, Mar 2021 \[link\]](#)  
 Ch 4 'Writing your first kernel module - LKMs Part 1' section 'Understanding kernel logging and printk':

“ ...

Enter the **pr\_fmt** macro; defining this macro right at the beginning of your code (it must be even before the first #include), guarantees that every single subsequent printk in your code will be prefixed with the format specified by this macro. Lets take an example (we show a snippet of code from the next chapter; worry not, it's really very simple, and serves as a template for your future kernel modules):

```
// ch5/lkm_template/lkm_template.c
[ ... ]
*/
#define pr_fmt(fmt) "%s:%s(): " fmt, KBUILD_MODNAME, __func__

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
[ ... ]

static int __init lkm_template_init(void)
{
    pr_info("inserted\n");
    [ ... ]
}
```

The **pr\_fmt()** macro is highlighted in bold font; it uses the pre-defined **KBUILD\_MODNAME** macro to substitute the name of your kernel module, and the **\_\_func\_\_** specifier to display the name of the function we're currently running! (You can even add a **%d** matched by the corresponding **\_\_LINE\_\_** macro to display the line number). So, bottom line: the **pr\_info()** we emit in the init function of this LKM will display like this in the kernel log:

```
[381534.391966] lkm_template:lkm_template_init(): inserted
```

Notice how the LKM name and the function name are automatically prefixed. This is very useful and indeed very common; in the kernel, literally hundreds of source files begin with the **pr\_fmt()**. (A quick search on the 5.4 kernel code base revealed over 2,000 instances of this macro in the code base! ...)

Note- *Driver authors:*

The useful **pr\_fmt()** 'prefixing' doesn't work with the **dev\_\*()** macros; for them, substitute the **dev\_fmt()** (at the beginning of the source file)!

*Eg.*

```
#define dev_fmt(fmt) "%s(): " fmt, __func__
...
[ 6225.672886] dht2x_kdrv:dht2x_read_sensors(): str_crc=32
[ 6225.672904] dht2x 1-0038: dht2x_read_sensors(): crc obtd=0x32 crc=0x32
[ 6225.672920] dht2x 1-0038: dht2x_temp_show(): Temperature=24826 milliC
```

(BTW, using the `dev_*` macros are the reason why the useful *drivername bus#-device\_addr* prefix shows up!)

```
>>
```

## Rate Limiting

If you are not careful, you can find yourself generating thousands of messages with `printk`, overwhelming the console and, possibly, overflowing the system log file. When using a slow console device (e.g., a serial port), an excessive message rate can also slow down the system or just make it unresponsive. It can be very hard to get a handle on what is wrong with a system when the console is spewing out data nonstop. Therefore, *you should be very careful about what you print, especially in production versions* of drivers and especially once initialization is complete. In general, production code should never print anything during normal operation; printed output should be an indication of an exceptional situation requiring attention.

On the other hand, you may want to emit a log message if a device you are driving stops working. But you should be careful not to overdo things. An unintelligent process that continues forever in the face of failures can generate thousands of retries per second; if your driver prints a “my device is broken” message every time, it *could create vast amounts of output and possibly hog the CPU* if the console device is slow—no interrupts can be used to driver the console, even if it is a serial port or a line printer.

In many cases, the best behavior is to set a flag saying, “I have already complained about this,” and not print any further messages once the flag gets set. In others, though, there are reasons to emit an occasional “the device is still broken” notice. The kernel has provided a function that can be helpful in such cases:

```
include/linux/printk.h
...
/*
 * Please don't use printk_ratelimit(), because it shares ratelimiting state
 * with all other unrelated printk_ratelimit() callsites. Instead use
 * printk_ratelimited() or plain old __ratelimit().
 */
...
#define printk_ratelimited(fmt, ...) \
...
```



This function **should be called before you consider printing a message that could be repeated often**. If the function returns a nonzero value, go ahead and print your message, otherwise skip it. Thus, typical calls look like this:

```
printk_ratelimited(KERN_NOTICE "The printer is still on fire\n");
```

`printk_ratelimited` works by tracking how many messages are sent to the console. When the level of output exceeds a threshold, `printk_ratelimit` starts returning 0 and causing messages to be dropped.

The behavior of `printk_ratelimit` can be **customized** by modifying:

`/proc/sys/kernel/printk_ratelimit` : the number of seconds to wait before re-enabling messages and  
`/proc/sys/kernel/printk_ratelimit_burst` : the number of messages accepted before ratelimiting.

Eg.:

```
$ uname -r  
4.4.0-59-generic  
$ cat /proc/sys/kernel/printk_ratelimit /proc/sys/kernel/printk_ratelimit_burst  
5          << number of seconds to wait before re-enabling messages >>  
10         << number of messages accepted before ratelimiting >>  
$
```

<<

**Note:**

1. The kernel has some built-in ratelimited printk's :

```
include/linux/printk.h
```

```
...  
#define pr_emerg_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_alert_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_crit_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_err_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_warn_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_notice_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)  
#define pr_info_ratelimited(fmt, ...) \  
    printk_ratelimited(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)  
...
```

Example of rate limiting (a failing SD MMC card):

```
kern :err : [808993.600764] Buffer I/O error on dev sda, logical block 1027, lost async page write
kern :err : [808993.600765] Buffer I/O error on dev sda, logical block 1028, lost async page write
kern :err : [808993.600766] Buffer I/O error on dev sda, logical block 1029, lost async page write
kern :info : [809015.592922] sd 0:0:0:0: [sda] tag#0 FAILED Result: hostbyte=DID_OK driverbyte=DRIVER_OK cmd_age=21s
kern :info : [809015.592927] sd 0:0:0:0: [sda] tag#0 Sense Key : Medium Error [current]
kern :info : [809015.592929] sd 0:0:0:0: [sda] tag#0 Add. Sense: Incompatible medium installed
kern :info : [809015.592931] sd 0:0:0:0: [sda] tag#0 CDB: Write(10) 2a 00 00 00 20 d0 00 00 f0 00
kern :err : [809015.592932] I/O error, dev sda, sector 8400 op 0x1:(WRITE) flags 0x104000 phys_seg 30 prio class 0
kern :warn : [809015.592936] buffer_io_error: 20 callbacks suppressed
kern :err : [809015.592936] Buffer I/O error on dev sda, logical block 1050, lost async page write
kern :err : [809015.592944] Buffer I/O error on dev sda, logical block 1051, lost async page write
kern :err : [809015.592946] Buffer I/O error on dev sda, logical block 1052, lost async page write
kern :err : [809015.592947] Buffer I/O error on dev sda, logical block 1053, lost async page write
kern :err : [809015.592948] Buffer I/O error on dev sda, logical block 1054, lost async page write
kern :err : [809015.592950] Buffer I/O error on dev sda, logical block 1055, lost async page write
kern :err : [809015.592951] Buffer I/O error on dev sda, logical block 1056, lost async page write
kern :err : [809015.592952] Buffer I/O error on dev sda, logical block 1057, lost async page write
kern :err : [809015.592954] Buffer I/O error on dev sda, logical block 1058, lost async page write
kern :err : [809015.592955] Buffer I/O error on dev sda, logical block 1059, lost async page write
```

*Eg. partial screenshot of cscope on 5.4 Linux to see where `pr_crit_ratelimited()` is called:*

Functions calling this function: `pr_crit_ratelimited`

File	Function	Line	Code
0 md-multipath.c	multipath_map	47	<code>pr_crit_ratelimited("multipath_map(): no more operational IO paths?\n");</code>
1 raid1.c	raid1_read_request	1266	<code>pr_crit_ratelimited("md/raid1:%s: %s: unrecoverable I/O read error for block %llu\n",</code>
2 raid1.c	fix_sync_read_error	2042	<code>pr_crit_ratelimited("md/raid1:%s: %s: unrecoverable I/O read error for block %llu\n",</code>
3 raid10.c	raid10_read_request	1180	<code>pr_crit_ratelimited("md/raid10:%s: %s: unrecoverable I/O read error for block %llu\n",</code>
4 ima_crypto.c	ahash_wait	199	<code>pr_crit_ratelimited("ahash calculation failed: err: %d\n", err);</code>

...

*/\* If you are writing a driver, please use `dev_dbg` instead \*/*

`dev_dbg` becomes `dev_printk`

```
dev_printk(KERN_DEBUG, dev, format, ##arg)
```

and takes effect ONLY when `DEBUG` is defined:

```
include/linux/device.h
```

```
...
#if defined(CONFIG_DYNAMIC_DEBUG)
#define dev_dbg(dev, format, ...) \
do { \
    dynamic_dev_dbg(dev, format, ##__VA_ARGS__); \
} while (0)
#elif defined(DEBUG)
```

```

#define dev_dbg(dev, format, arg...) \
    dev_printk(KERN_DEBUG, dev, format, ##arg)
#else
#define dev_dbg(dev, format, arg...) \
({
    if (0) \
        dev_printk(KERN_DEBUG, dev, format, ##arg); \
    0; \
})
#endif
...

drivers/base/core.c
...
int dev_printk(const char *level, const struct device *dev,
               const char *fmt, ...)
{
    ...
}>>

```

## Where you Can and Cannot Use printk

<< Source: “Linux Kernel Development” by Robert M Love, 2<sup>nd</sup> Ed., Novell Press >>

### The Robustness of printk()

One property of printk() quickly taken for granted is its robustness. The printk() function is callable from just about anywhere in the kernel at any time. It can be called from interrupt or process context. It can be called while a lock is held. It can be called simultaneously on multiple processors, yet it does not require the caller to hold a lock.

It is a resilient function. This is important because the usefulness of printk() rests on the fact that it is always there and always works.

### The Nonrobustness of printk()

#### Scenario I

A chink in the armor of printk()'s robustness does exist. It is unusable before a certain point in the kernel boot process, prior to console initialization. Indeed, if the console is not initialized, where is the output supposed to go?

This is normally not an issue, unless you are debugging issues very early in the boot process (for example, in `setup_arch()`, which performs architecture-specific initialization). Such debugging is a challenge to begin with, and the absence of any sort of print method only compounds the problem.

There is some hope, but not a lot. Hardcore architecture hackers use the hardware that does work (say, a serial port) to communicate with the outside world. Trust me—this is not fun for most people. Some

supported architectures do implement a sane solution, however—and others (i386 included) have patches available that also save the day.

The solution is a *printk()* variant that can output to the console very early in the boot process: **early\_printk** (or *early\_print()*). The behavior is the same as *printk()*, only the name and its capability to work earlier are changed. This is not a portable solution, however, because not all supported architectures have such a method implemented. It might become your best friend, though, if it does.

Unless you need to write to the console very early in the boot process, you can rely on *printk()* to always work.

```
<<
```

For using *early\_printk* and friends, must turn on

#### **CONFIG\_DEBUG\_LL : Kernel Hacking**

```
[*] Kernel low-level debugging functions
[ ] Kernel low-level debugging via EmbeddedICE DCC channel
```

```
[...]
```

CONFIG\_DEBUG\_LL:

```
Say Y here to include definitions of printascii, printch, printhex
in the kernel. This is helpful if you are debugging code that
executes before the console is initialized.
```

```
Symbol: DEBUG_LL [=n]
Prompt: Kernel low-level debugging functions
Defined at arch/arm/Kconfig.debug:54
Depends on: DEBUG_KERNEL
Location:
-> Kernel hacking
```

Insert some *printascii()* calls to follow early init code.

(note- above snippet is on an ARM Linux kernel).

```
>>
```

```
<<
```

*kernel/printk/printk.c*

```
asmlinkage __visible void early_printk(const char *fmt, ...)
{
    va_list ap;
    char buf[512];
    int n;

    if (!early_console)
        return;

    va_start(ap, fmt);
    n = vsnprintf(buf, sizeof(buf), fmt, ap);
```

```

        va_end(ap);

        early_console->write(early_console, buf, n);
}

```

In it, the `write` method of the `early_console` structure points to an **arch-specific wrapper** around the routine `write`; this in turn invokes an appropriate API to write each character to the serial console device (typically in a loop).

F.e. for the x86:

`arch/x86/kernel/early_printk.c`

```

static struct console early_serial_console = {
    .name = "earlyser",
    .write = early_serial_write,
    .flags = CON_PRINTBUFFER,
    .index = -1,
};
...

static void early_serial_write(struct console *con, const char *s, unsigned n)
{
    while (*s && n-- > 0) {
        if (*s == '\n')
            early_serial_putc('\r');
        early_serial_putc(*s);
        s++;
    }
}

```

The `early_serial_putc()` becomes `serial_out()` whose implementation is arch-specific:

```

> 1 F C v    serial_out      arch/x86/kernel/early_printk.c
      static void (*serial_out)(unsigned long addr, int offset, int value)
= io_serial_out;
  2 F  f    serial_out      drivers/fsi/fsi-master-gpio.c
      signature:(struct fsi_master_gpio *master, const struct fsi_gpio_msg
*cmd)
      static void serial_out(struct fsi_master_gpio *master,
  3 F  f    serial_out      drivers/tty/serial/8250/8250.h
      signature:(struct uart_8250_port *up, int offset, int value)
      static inline void serial_out(struct uart_8250_port *up, int offset,
int value)
  4 F  f    serial_out      drivers/tty/serial/ma35d1_serial.c
      signature:(struct uart_ma35d1_port *p, u32 offset, u32 value)
      static void serial_out(struct uart_ma35d1_port *p, u32 offset, u32
value)
  5 F  f    serial_out      drivers/tty/serial/omap-serial.c
      signature:(struct uart_omap_port *up, int offset, int value)
      static inline void serial_out(struct uart_omap_port *up, int offset,
int value)
...
...

```

&gt;&gt;

## Scenario II

<< From the LWN article “Debugging the kernel using Ftrace - part 1” [here](#). >>

--snip--

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** (especially on a slow serial line – which is typically the case in embedded development environments).

Ftrace introduces a new form of printk() called **trace\_printk()** (details follow later).

---

## Dynamic Debug

Kernel config: CONFIG\_DYNAMIC\_DEBUG needs to be enabled.

From *lib/Kconfig.debug*:

```
...
config DYNAMIC_DEBUG
    bool "Enable dynamic printk() support"
    default n
    depends on PRINTK
    depends on DEBUG_FS
    help
```

Compiles debug level messages into the kernel, which would not otherwise be available at runtime. These messages can then be enabled/disabled based on various levels of scope - per source file, function, module, format string, and line number. This mechanism implicitly compiles in all `pr_debug()` and `dev_dbg()` calls, which enlarges the kernel text size by about 2%.

If a source file is compiled with DEBUG flag set, any `pr_debug()` calls in it are enabled by default, but can be disabled at runtime as below. Note that DEBUG flag is turned on by many CONFIG\_\*DEBUG\* options.

Usage:

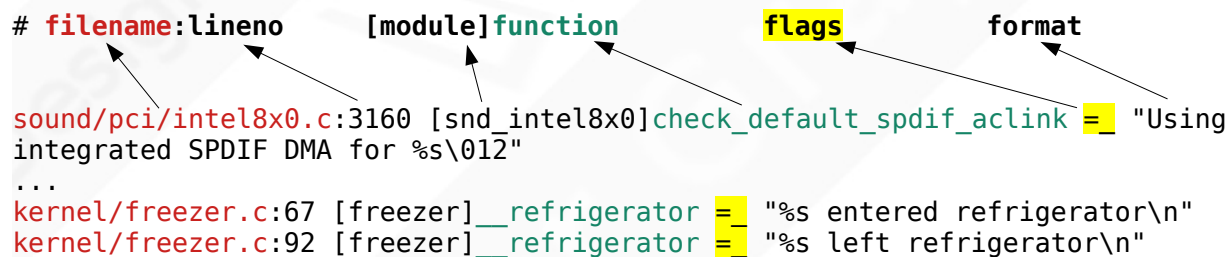
Dynamic debugging is controlled via the 'dynamic\_debug/control' file, which is contained in the 'debugfs' filesystem. Thus, the debugfs filesystem must first be mounted before making use of this feature. We refer the control file as: `<debugfs>/dynamic_debug/control`. This file contains a list of the debug statements that can be enabled. The



format for each line of the file is: ...

>>

*Dynamic debug control file format specification (with a few sample lines from the actual runtime control pseudo file: `/sys/kernel/debug/dynamic_debug/control` (or) `/proc/dynamic_debug/control` :*



```
# filename:lineno [module]function flags format
sound/pci/intel8x0.c:3160 [snd_intel8x0]check_default_spdif_aclink == "Using
integrated SPDIF DMA for %s\012"
...
kernel/freezer.c:67 [freezer]__refrigerator == "%s entered refrigerator\n"
kernel/freezer.c:92 [freezer]__refrigerator == "%s left refrigerator\n"
```

**flags** specifier: =

\_ : OFF

p : ON

Ok, let's turn it ON for this file `sound/pci/intel8x0.c` ! (Of course, this requires **root** access):

```
# echo "file sound/pci/intel8x0.c +p" > /sys/kernel/debug/dynamic_debug/control
```

### TIP

If dynamic debug is On but debugfs is Off (or made 'invisible'), the dynamic debug 'control' file's available within procfs here:

`/proc/dynamic_debug/control`

Also see:

**Official [Kernel doc](#) on dynamic debug**

[Details](#).

Dynamic debug allows one to see all debug printk's (pr\_debug() callsites) from within a file specification / module spec / etc !

## Examples

A couple of examples here will help (run as root of course):

- Enable all debug messages in all files where the pathname includes the string "usb":  
`echo -n 'file *usb* +p' > /sys/kernel/debug/dynamic_debug/control`
- enable *all* debug printk's (pr\_debug()|dev\_dbg() callsites) for all kernel modules:  
`echo -n 'module * +pflmt' > /sys/kernel/debug/dynamic_debug/control`

*WARNING: can lead to very voluminous output!*

Turn it Off with:

```
echo -n 'module * -pflmt' > /sys/kernel/debug/dynamic_debug/control
```

Keep an eye on the kernel log output with (something like) `journalctl -f`.

You can lookup and **try out this example** from the *Linux Kernel Debugging* book (Ch 4, *Debug via Instrumentation – Kprobes*):

[https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch4/kprobes/3\\_kprobe](https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch4/kprobes/3_kprobe). The `test.sh` Bash script within here enables dynamic debug (in order to see all printk's being emitted from the module).

*Interpret the **flags** (like +p):*

p	Enables the <code>pr_debug()</code> callsite.
f	Include the <code>function</code> name in the printed message
l	Include <code>line</code> number in the printed message
m	Include <code>module</code> name in the printed message
t	Include <code>thread</code> ID in messages not generated from interrupt context
_	No flags are set. (OR'd with others on input)

---

**‘Poor man’s printf()’:**

- using an LED (via GPIO/port)
- The Raspberry Pi uses the LED in a blink sequence to denote different types of errors [[link](#)] (see next table):
- using a scope: see the ‘[The Embedded Muse #364](#)’, section ‘No printf()? No Problem’

**Saving the kernel log in non-volatile storage**

- the default kernel logs are in RAM
  - in a circular ring buffer (in `__log_buf` in RAM)
- on PCs and servers, the log is backed up to non-volatile disk (typically into the `/var/log/[messages|syslog]` file, or with `systemd`, via the *journal*)
- but on some/many embedded systems, we don’t write to flash as far as is possible (as flash chips have a limited # of erase-write cycles, and thus lifetime)
- So how to save the kernel log?
  - One approach: with a system that supports flash chips via the MTD subsystem (Memory Technology Devices): configure the kernel with `CONFIG_MTD_OOPS` and append the kernel parameter ‘`console=ttyMTDn`’ to the kernel command line. This essentially writes out the kernel log on any Oops or panic to an MTD device ‘n’.

From the *Kconfig*:

```
...
CONFIG_MTD_OOPS:
| This enables panic and oops messages to be logged to a circular
| buffer in a flash partition where it can be read back at some
| later point.
|
| Symbol: MTD_OOPS [=y]
| Type : tristate
| Prompt: Log panic/oops to an MTD buffer
| Location:
|   -> Device Drivers
|   -> Memory Technology Device (MTD) support (MTD [=y])
...
```

Similarly, `CONFIG_MMC_OOPS` for SDMMC cards; but *not* available (yet) in mainline.

DOCUMENTATION > CONFIGURATION > LED\_BLINK\_WARNINGS

**LED warning flash codes**

If a Pi fails to boot for some reason, or has to shut down, in many cases an LED will be flashed a specific number of times to indicate what happened. The LED will blink for a number of long flashes (0 or more), then short flashes, to indicate the exact status. In most cases, the pattern will repeat after a 2 second gap.

Long flashes	Short flashes	Status
0	3	Generic failure to boot
0	4	start*.elf not found
0	7	Kernel image not found
0	8	SDRAM failure
0	9	Insufficient SDRAM
0	10	In HALT state
2	1	Partition not FAT
2	2	Failed to read from partition
2	3	Extended partition not FAT
2	4	File signature/hash mismatch - Pi 4
3	1	SPI EEPROM error - Pi 4
3	2	SPI EEPROM is write protected - Pi 4
4	4	Unsupported board type
4	5	Fatal firmware error
4	6	Power failure type A
4	7	Power failure type B

### ***Filesystem Pstore (Persistent RAM) and the RamOops Functionality***

Similarly, Linux has a RAMOOPS Oops/panic logger facility. Source: *fs/pstore/ram.c*  
 CONFIG\_PSTORE\_RAM : from Linux 2.6.39.

Doc: <https://www.kernel.org/doc/html/v4.15/admin-guide/ramoops.html>

“Ramoops is an oops/panic logger that writes its logs to RAM before the system crashes. It works by logging oopses and panics in a circular buffer. Ramoops needs a system with persistent RAM so that the content of that area can survive after a restart.

[...]

Setting the ramoops parameters can be done in several different manners:

A. Use the module parameters (which have the names of the variables described as before). For quick debugging, you can also reserve parts of memory during boot and then use the reserved memory for ramoops. For example, assuming a machine with > 128 MB of memory, the following kernel command line will tell the kernel to use only the first 128 MB of memory, and place ECC-protected ramoops region at 128 MB boundary:

```
mem=128M ramoops.mem_address=0x8000000 ramoops.ecc=1
```

...

<< Features like Pstore tend to be arch and even device-dependant; check if it's (or even can be) enabled for your system >>

*fs/pstore/Kconfig*

```
config PSTORE
    tristate "Persistent store support"
    select CRYPTO if PSTORE_COMPRESS
    default n
    help
        This option enables generic access to platform level
        persistent storage via "pstore" filesystem that can
        be mounted as /dev/pstore. Only useful if you have
        a platform level driver that registers with pstore to
        provide the data, so you probably should just go say "Y"
        (or "M") to a platform specific persistent store driver
        (e.g. ACPI_APEI on X86) which will select this for you.
        If you don't have a platform persistent store driver,
        say N.
```

[...]

```
config PSTORE_CONSOLE
    bool "Log kernel console messages"
    depends on PSTORE
    help
        When the option is enabled, pstore will log all kernel
        messages, even if no oops or panic happened.
```

...

A screenshot

```
.config - Linux/arm 4.14.52 Kernel Configuration
- File systems -> Miscellaneous filesystems

Miscellaneous filesystems
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus
----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M>
modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search.
Legend: [*] built-in [ ] excluded <M> module <> module capable

[ ] Include support for LZ4 compressed file systems
[*] Include support for LZ0 compressed file systems
[ ] Include support for XZ compressed file systems
[ ] Include support for ZSTD compressed file systems
[ ] Use 4K device block size?
[ ] Additional option for memory-constrained systems
<> FreeVxFS file system support (VERITAS VxFS(TM) compatible)
<> Minix file system support
<> SonicBlue Optimized MPEG File System support
<> OS/2 HPFS file system support
<> QNX4 file system support (read only)
<> QNX6 file system support (read only)
<> ROM file system support
<M> Persistent store support
    Choose compression algorithm (ZLIB) --->
[*] Log kernel console messages
[*] Log user space messages
[*] Persistent function tracer
<M> Log panic/oops to a RAM buffer
<> System V/Xenix/V7/Coherent file system support
<> UFS file system support (read only)

<Select> < Exit > < Help > < Save > < Load >
```

Perhaps useful:

[Using pstore to collect crash dumps – kernel-hardening mailing list, Kees Cook \[03Oct2019\]](#)

[Resource: Logging Kernel Oops and Panic](#)

## Debugging Early Boot Issues

Even if there is a serial console, but we receive no (printk) output, perhaps because an issue has arisen well before the serial device (console\_init() code) has even run! Or, the system simply freezes... Then how do we debug this?

An interesting and at times a very valuable approach: realize that once the printk has occurred, the data has been buffered by the kernel **into the printk log buffer** (a software RAM buffer). If we can ascertain the address of the printk log buffer in RAM, and somehow dump it's contents, we can see the intended printk output!

But how? Typical approach:

*Example below is with the (old) [TI Pandaboard](#) OMAP4430 SoC shown below*

1. Figure out the kernel virtual address of the printk log buffer before-hand:

The global variable holding the pointer to the printk log buffer is `__log_buf`.

```
# grep __log_buf /proc/kallsyms
c061c7cc b __log_buf
#
```

So, as seen, the kernel virtual address (kva) of the printk log buffer (in this case) is 0xc061c7cc.

2. Then *warm-boot* the board and stop at the bootloader monitor.  
(Press the RESET button on the board; in this case the bootloader is U-Boot):

```
...
#
```

```
Texas Instruments X-Loader 1.41 (Sep  1 2010 - 13:43:00)
mmc read: Invalid size
Starting OS Bootloader from MMC/SD1 ...
```

```
U-Boot 2011.12 (Jan 06 2012 - 11:54:05)
```

```
CPU   : OMAP4430 ES2.2
Board: OMAP4 Panda
I2C:   ready
DRAM:  512 MiB
MMC:   OMAP SD/MMC: 0
Using default environment
```



```
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  1
u-boot_Panda #
```

← We're at the U-boot prompt

3. Figure out the mapping between board physical addresses and the high-level OS (Linux) virtual addresses.

Use the board TRM (Technical Reference Manual) or the [OMAP 4430 SoC TRM](#) to glean this information.

In the “Memory Mapping” chapter (pg 275) we see a table that shows:

Table 2-1. Global Memory Space Mapping

Quarter Description	Module Name	Start Address (hex)	End Address (hex)	Size
Q2 (1 GB) :	DRAM address space	0x8000 0000	0xBFFF FFFF	1GB
		DDR-SDRAM CS0 address space		

From this we infer that RAM address 0x0 is the board physical address 0x8000 0000 .

Now, we know that on a 32-bit system running the Linux OS with a **3 GB:1 GB :: user:kernel VM split** (CONFIG\_VM\_SPLIT), kernel virtual address space starts at 0xC000 0000. This corresponds to RAM address 0x0.

Hence:

**board physical address 0x8000 0000 == kernel virtual address 0xC000 0000 == RAM physical address 0x0**

Thus, kernel va for `__log_buf` (from Step 1) **0xc061c7cc** == board physical address **0x8061c7cc** .

4. Use appropriate bootloader commands to dump the printk log buffer contents:

```
u-boot_Panda # help md
md - memory display
```

```
Usage:
md [.b, .w, .l] address [# of objects]
```

```
u-boot_Panda # md 8061c7cc
8061c7cc: 5b3e353c 20202020 30302e30 30303030    <5>[    0.000000
8061c7dc: 694c205d 2078756e 73726576 206e6f69    ] Linux version
8061c7ec: 2e312e33 6b282035 61776961 616b406e    3.1.5 (kaiwan@ka
8061c7fc: 6e617769 6968542d 61506b6e 32582d64    iwan-ThinkPad-X2
8061c80c: 20293032 63636728 72657620 6e6f6973    20) (gcc version
```

```

8061c81c: 352e3420 2820342e 6c697542 6f6f7264
8061c82c: 30322074 302e3231 29202938 20382320
8061c83c: 20504d53 20756854 20636544 31203732
8061c84c: 31343a33 2031313a 20545349 32313032
8061c85c: 3e343c0a 2020205b 302e3020 30303030
8061c86c: 43205d30 203a5550 764d5241 72502037
8061c87c: 7365636f 20726f73 3131345b 39306366
8061c88c: 72205d32 73697665 206e6f69 41282032
8061c89c: 37764d52 63202c29 30313d72 38333563
8061c8ac: 3c0a6437 205b3e34 30202020 3030302e
8061c8bc: 5d303030 55504320 4956203a 6e205450
u-boot_Panda # << Enter key >>
8061c8cc: 6c616e6f 69736169 6420676e 20617461
8061c8dc: 68636163 56202c65 20545049 61696c61
8061c8ec: 676e6973 736e6920 63757274 6e6f6974
8061c8fc: 63616320 3c0a6568 205b3e34 30202020
8061c90c: 3030302e 5d303030 63614d20 656e6968
8061c91c: 4d4f203a 20345041 646e6150 6f622061
8061c92c: 0a647261 5b3e363c 20202020 30302e30
8061c93c: 30303030 6552205d 76726573 20676e69
8061c94c: 37373631 36313237 74796220 53207365
8061c95c: 4d415244 726f6620 41525620 343c0a4d
8061c96c: 20205b3e 2e302020 30303030 205d3030
8061c97c: 6f6d654d 70207972 63696c6f 45203a79
8061c98c: 64204343 62617369 2c64656c 74614420
8061c99c: 61632061 20656863 74697277 6c6c6165
8061c9ac: 3c0a636f 205b3e36 30202020 3030302e
8061c9bc: 5d303030 414d4f20 33343450 53452030
u-boot_Panda #
...
...
8062247c: 37202020 3238362e 5d323433 62737520
8062248c: 312d3120 203a312e 6b6e696c 38687120
8062249c: 3030302d 65642f31 62616630 73203032
806224ac: 74726174 5b203220 20302f31 0a5d7375
806224bc: 00000000 00000000 00000000 00000000
...
<< End section >>

```

4.5.4 (Buildroot 2012.08) ) #8  
SMP Thu Dec 27 13:41:11 IST 2012  
.<4>[ 0.000000  
0] CPU: ARMv7 Processor [411fc092] revision 2 (ARMv7), cr=10c5387d.<4>[ 0.000000] CPU: VIPT non-aliasing data cache, VIPT aliasing instruction cache.<4>[ 0.000000] Machine : OMAP4 Panda board.<6>[ 0.000000] Reserving 16777216 bytes SDRAM for VRAM.<4>[ 0.000000] Memory policy: ECC disabled, Data cache writeall.<6>[ 0.000000] OMAP4430 ES7.682342] usb 1-1.1: link qh8 -0001/de0fab20 start 2 [1/0 us].  
.....

## Useful Resources

### [Linux Kernel Debugging by Printing](#)

(includes a section on 'Debugging Early Boot Problems')

See [Debugging the Linux kernel using Eclipse/CDT and Qemu](#) for a great article on using Eclipse (with the CDT plugin) to debug the Linux kernel.

### [Kernel Debugging Tips](#)

[A KDB / KGDB session on the popular Raspberry Pi embedded Linux board](#)

&lt;&lt;

**Modern!**

## **Interfacing Kernel and Userspace**

For theory and code, please refer my LKP-2 book:

*Linux Kernel Programming, Part 2 - Char Device Drivers and Kernel Synchronization*

- Free e-book!.
- GitHub repo: <https://github.com/PacktPublishing/Linux-Kernel-Programming-Part-2>
  - User-Kernel Interfacing code : <https://github.com/PacktPublishing/Linux-Kernel-Programming-Part-2/tree/main/ch2>

&gt;&gt;

<< **OLDER !** >>

&lt;&lt; Source: “Linux Device Drivers” by J Corbet, A Rubini &amp; GK Hartman, 3rd Ed., O'Reilly &gt;&gt;

## **Using the /proc Filesystem**

&lt;&lt;

**Note!Note!Note!**

**From 2.6 Linux, non-core users (module/driver authors) are **NOT** meant to use procfs. The Procfs API is not fully supported on modern Linux kernels and is meant for only in-kernel (core) use.**

**Please use **debugfs** instead.**

&gt;&gt;

&lt;&lt;

Having said that, it is possible of course to use procfs for your own purposes. It's just that such code will never be allowed within the mainline kernel.

```
include/linux/proc_fs.h
```

```
static inline struct proc_dir_entry *proc_create(
    const char *name, umode_t mode, struct proc_dir_entry *parent,
    const struct file_operations *proc_fops);

extern struct proc_dir_entry *proc_symlink(const char *,
    struct proc_dir_entry *, const char *);

extern struct proc_dir_entry *proc_mkdir(const char *, struct proc_dir_entry *);

extern int remove_proc_subtree(const char *, struct proc_dir_entry *);
>>
```

## debugfs

A very useful code-based debug facility that is kind of similar to procfs is the so-called “**debugfs**” filesystem. It's available in the mainline kernel from 2.6.10-rc3.

Very useful to quickly learn all basic uses/APIs of debugfs:

[Documentation/filesystems/debugfs.txt](#)

(Also see and try the 'debugfs\_eg' kernel module provided on the participant CD).

Additionally, this tutorial also quickly covers the meat of it:

<https://bitbucket.org/chadversary/debugfs-tutorial>

(Participant's should read and follow the kernel source tree documentation on debugfs ; this will give you the necessary information and knowledge to try some useful hands-on experience using 'debugfs' from the kernel developer / debugger viewpoint.

The Instructor will guide you with some sample code.)

## The ioctl Method

<<

Refer the LDD3 book : “*LINUX Device Drivers*” by J Corbet, A Rubini and GK-Hartman, 3<sup>rd</sup> Ed, O'Reilly and Associates, Ch 6 “Advanced Char Driver Operations” section “ioctl” page 135 onward.

Then read the sources of the demo ioctl driver (provided on [my L5 github repo](#)), and try it out.

>>

*ioctl*, which we show you how to use in Chapter 1 (userspace-wise), is a system call that acts on a file descriptor; **it receives a number that identifies a command to be performed and (optionally) another argument, usually a pointer**. As an alternative to using the */proc* filesystem, **you can implement a few *ioctl* commands tailored for debugging**. These commands **can copy relevant data structures from the**

driver to user space where you can examine them.

Using *ioctl* this way to get information is **somewhat more difficult than using */proc***, because you need another program to issue the *ioctl* and display the results. This program must be written, compiled, and kept in sync with the module you're testing. On the other hand, the driver-side code can be easier than what is needed to implement a */proc* file.

There are times when *ioctl* is the best way to get information, because it **runs faster** than reading */proc*. If some work must be performed on the data before it's written to the screen, retrieving the data in binary form is more efficient than reading a text file. In addition, *ioctl* **doesn't require splitting data** into fragments smaller than a page.

Another interesting advantage of the *ioctl* approach is that information-retrieval commands **can be left in the driver** even when debugging would otherwise be disabled. Unlike a */proc* file, which is visible to anyone who looks in the directory (and too many people are likely to wonder "what that strange file is"), undocumented *ioctl* commands are likely to remain unnoticed. In addition, they will still be there should something weird happen to the driver. The only drawback is that the module will be slightly bigger.

<<

In effect, one can leave "undocumented" debugging hooks enabled in the production system.

Also, remember that the 'magic number' is embedded into the *ioctl* command (see the header file); this means that even if a user (or customer) attempts an *ioctl* command on the device, it will be rejected (will fail - unless (s)he knows the exact magic and command number).

>>

### **kaiwanTECH Linux OS Corporate Training Programs**

Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs [here](#).