Reporting the bug

Once you find where the bug happened, by inspecting its location, you could either try to fix it yourself or report it upstream.

In order to report it upstream, you should identify the mailing list used for the development of the affected code. This can be done by using the `get_maintainer.pl` script.

For example, if you find a bug at the gspca's sonixj.c file, you can get their maintainers with:

```
$ ./scripts/get_maintainer.pl -f drivers/media/usb/gspca/sonixj.c
Hans Verkuil <hverkuil@xs4all.nl> (odd fixer:GSPCA USB WEBCAM
DRIVER,commit_signer:1/1=100%)
Mauro Carvalho Chehab <mchehab@kernel.org> (maintainer:MEDIA INPUT
INFRASTRUCTURE (V4L/DVB),commit_signer:1/1=100%)
Tejun Heo <tj@kernel.org> (commit_signer:1/1=100%)
Bhaktipriya Shridhar <bhaktipriya96@gmail.com>
(commit_signer:1/1=100%,authored:1/1=100%,added_lines:4/4=100%,removed_lines:9/9
=100%)
linux-media@vger.kernel.org (open list:GSPCA USB WEBCAM DRIVER)
linux-kernel@vger.kernel.org (open list)
```

Please notice that it will point to:

- The last developers that touched on the source code. On the above example, Tejun and Bhaktipriya (in this specific case, none really envolved on the development of this file);
- The driver maintainer (Hans Verkuil);
- The subsystem maintainer (Mauro Carvalho Chehab);
- The driver and/or subsystem mailing list (linux-media@vger.kernel.org);
- the Linux Kernel mailing list (linux-kernel@vger.kernel.org).

Usually, the fastest way to have your bug fixed is to report it to mailing list used for the development of the code (linux-media ML) copying the driver maintainer (Hans).

If you are totally stumped as to whom to send the report, and `get_maintainer.pl` didn't provide you anything useful, send it to linux-kernel@vger.kernel.org.

Thanks for your help in making Linux as stable as humanly possible.

Fixing the bug

If you know programming, you could help us by not only reporting the bug, but also providing us with a solution. After all, open source is about sharing what you do and don't you want to be recognised for your genius?

If you decide to take this way, once you have worked out a fix please submit it upstream.

Please do read Documentation/process/submitting-patches.rst though to help your code get accepted.

Finding the bug’s location

Reporting a bug works best if you point the location of the bug at the Kernel source file. There are two methods for doing that. Usually, using `gdb` is easier, but the Kernel should be pre-compiled with debug info.

gdb

The GNU debug (`gdb`) is the best way to figure out the exact file and line number of the OOPS from the `vmlinux` file.

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
```

Kernel panic logs

Kernel panic logs are useful to figure out what happened during an unsuccessful boot. If you’re trying to run a custom ROM but your phone is stuck at the boot loop, you can collect kernel panic logs to help the ROM developer find out what went wrong.

A majority of Android manufacturers use upstream ‘pstore’ and ‘ramoops’ drivers to store kernel logs after a panic. Ramoops writes its logs to the RAM before the system crashes. With root access, these logs can be retrieved from:

```
/sys/fs/pstore/console-ramoops
```

In kernel module programming, developers typically use a variety of functions to output information for debugging, logging, or monitoring purposes. The most commonly used functions for printing messages from kernel modules include:

1. **printk()**: This is the primary function used for printing messages in the Linux kernel. It allows developers to print messages with various log levels (e.g., KERN_INFO, KERN_ERR) and format specifiers similar to printf() in userspace.
2. **pr_info(), pr_err(), pr_warn(), pr_debug()**: These are macros provided by the Linux kernel for printing messages with specific log levels. They are wrappers around printk() with predefined log levels (INFO, ERR, WARN, DEBUG).
3. **dev_info(), dev_err(), dev_warn(), dev_dbg()**: These functions are used specifically for printing messages related to device drivers. They are similar to pr_info(), pr_err(), etc., but include the device context information.

4. **trace_printk()**: This function is used for tracing purposes and is often used for lightweight tracing within the kernel. It behaves similarly to printk() but is optimized for low overhead tracing.
5. **seq_printf()**: This function is used to format and print messages to a sequence file within the kernel. It is commonly used when implementing the procfs or sysfs interfaces for kernel modules.
6. **tracepoint()**: This is not exactly a print function, but it's worth mentioning. Tracepoints allow developers to instrument specific locations in the kernel code and collect tracing information dynamically. Tracepoints can be used for various debugging and performance analysis tasks.

In kernel module programming, developers primarily use the `printk()` function for printing messages to the kernel log. Here's the syntax for `printk()`:

```
c

#include <linux/kernel.h>

void printk(const char *fmt, ...);
```

The `printk()` function is similar to `printf()` in userspace C programming but is used specifically for kernel-space code. It takes a format string (`fmt`) and optional arguments (like `printf()`) and prints formatted output to the kernel log.

Here's an example of how to use `printk()`:

```
c

#include <linux/module.h> // Needed for kernel modules
#include <linux/kernel.h> // Needed for printk

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, kernel world!\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, kernel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple example kernel module");
```

In this example, `printk()` is used to print "Hello, kernel world!" when the module is initialized and "Goodbye, kernel world!" when it's exited.

Additionally, there are several macros provided in the Linux kernel for printing messages with specific log levels. For example:

- `pr_info()` for informational messages
- `pr_err()` for error messages
- `pr_warn()` for warning messages
- `pr_debug()` for debug messages

These macros have similar syntax to `printk()` but automatically include the appropriate log level. For instance:

c

```
pr_info("This is an informational message\n");
pr_err("This is an error message\n");
pr_warn("This is a warning message\n");
pr_debug("This is a debug message\n");
```

In kernel module programming, the `printk()` function, which is commonly used for printing messages to the kernel log, does not have a return type as it is a `void` function. It is used purely for outputting information rather than returning a value.

However, there are functions available in the Linux kernel that can be used for printing messages with a return type, such as `sprintf()`, `snprintf()`, and `vsnprintf()`, which are similar to their counterparts in userspace C programming.

Here are the syntaxes for these functions:

1. **sprintf()**:

c

```
#include <linux/kernel.h>

int sprintf(char *buf, const char *fmt, ...);
```

`sprintf()` formats a series of strings and values and stores them in a character array (`buf`). It returns the number of characters written to the buffer (excluding the null terminating character).

2. **snprintf()**:

c

```
#include <linux/kernel.h>

int snprintf(char *buf, size_t size, const char *fmt, ...);
```

`snprintf()` is similar to `sprintf()`, but it limits the number of characters written to the buffer (`buf`) to at most `size` characters (including the null terminating character). It returns the number of characters that would have been written if the buffer was large enough (excluding the null terminating character).

3. **vsnprintf()**:

c

```
#include <linux/kernel.h>
#include <stdarg.h>
```

```
int vsnprintf(char *buf, size_t size, const char *fmt, va_list args);
```

`vsnprintf()` is similar to `snprintf()`, but it takes a `va_list` instead of a variable number of arguments. This allows it to be used with functions that use variable arguments (`...`). It formats a series of strings and values according to the format specifier `fmt` and stores them in the buffer (`buf`). It returns the number of characters that would have been written if the buffer was large enough (excluding the null terminating character).

The log levels are:

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

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init my_module_init(void) {
    pr_emerg("Hello from my kernel module!\n");
    return 0; // Initialization succeeded
}

static void __exit my_module_exit(void) {
    pr_emerg("Exiting from my kernel module!\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple kernel module");

#include <linux/init.h>
```

```
#include <linux/module.h>
#include <linux/kernel.h>

static int __init my_module_init(void) {
    pr_alert("Hello from my kernel module!\n");
    return 0; // Initialization succeeded
}

static void __exit my_module_exit(void) {
    pr_alert("Exiting from my kernel module!\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple kernel module");
```

In the context of Linux kernel programming, `printk` and `pr_debug` serve different purposes and are used in different scenarios.

1. **printk**:
    - `printk` is a kernel logging function used to output messages to the system console, kernel log buffer, or other logging targets.
    - It's commonly used for debugging purposes, as well as for general logging and information output from the kernel.
    - Messages logged with `printk` can be seen using tools like `dmesg`.
    - `printk` messages can be enabled or disabled dynamically at runtime using the `syslog` or `klogctl` system calls, or via `/proc/sys/kernel/printk`.
    - Example usage: `printk(KERN_INFO "This is an informational message\n");`
2. **pr_debug**:
    - `pr_debug` is a macro provided by the Linux kernel for conditional debugging.
    - Messages logged with `pr_debug` are only compiled into the kernel if the `CONFIG_DYNAMIC_DEBUG` configuration option is enabled and if dynamic debug is enabled at runtime for the specific code paths.
    - This means that `pr_debug` statements are generally not compiled into production kernels but can be selectively enabled for debugging without recompiling the kernel.
    - `pr_debug` statements are removed entirely from the kernel image if dynamic debug is disabled, minimizing the impact on performance and code size in production.
    - Example usage: `pr_debug("This is a debug message\n");`

In summary, while both `printk` and `pr_debug` can be used for debugging purposes in the Linux kernel, `printk` is more general-purpose and provides output that can be seen in the system logs, whereas `pr_debug` is more suited for conditional debugging and is only compiled into the kernel when dynamic debug is enabled.