Linux Device Driver Foundation

Topic 1: Introduction to Device Drivers

This topic covers what is device driver and bus driver? Device and driver interaction, device driver partition, device driver verticals.

1. Drivers and buses

A driver drives, manages, controls, directs and monitors the entity under its command. What a bus driver does with a bus, a device driver does with a computer device (any piece of hardware connected to a computer) like a mouse, keyboard, monitor, hard disk, Web-camera, clock, and more.

A specific piece of hardware could be controlled by a piece of software (a device driver), or could be controlled by another hardware device, which in turn could be managed by a software device driver. In the latter case, such a controlling device is commonly called a device controller. This, being a device itself, often also needs a driver, which is commonly referred to as a bus driver.

General examples of device controllers include hard disk controllers, display controllers, and audio controllers that in turn manage devices connected to them. More technical examples would be an IDE controller, PCI controller, USB controller, SPI controller, I2C controller, etc. Pictorially, this whole concept can be depicted as in Figure 1.

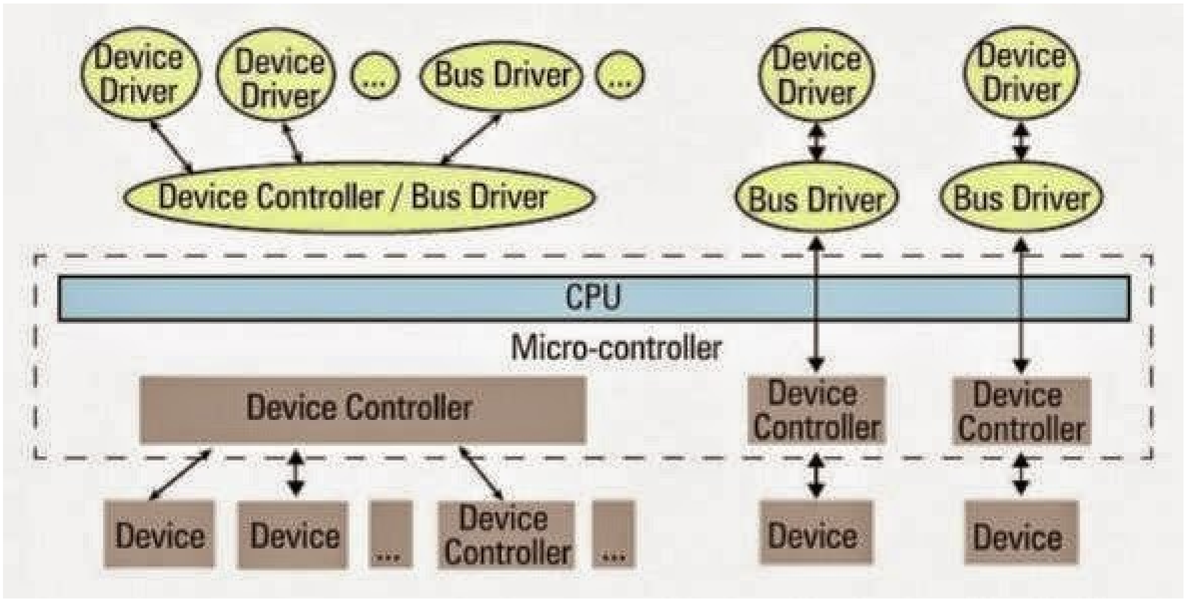


Figure 1: Device and driver interaction

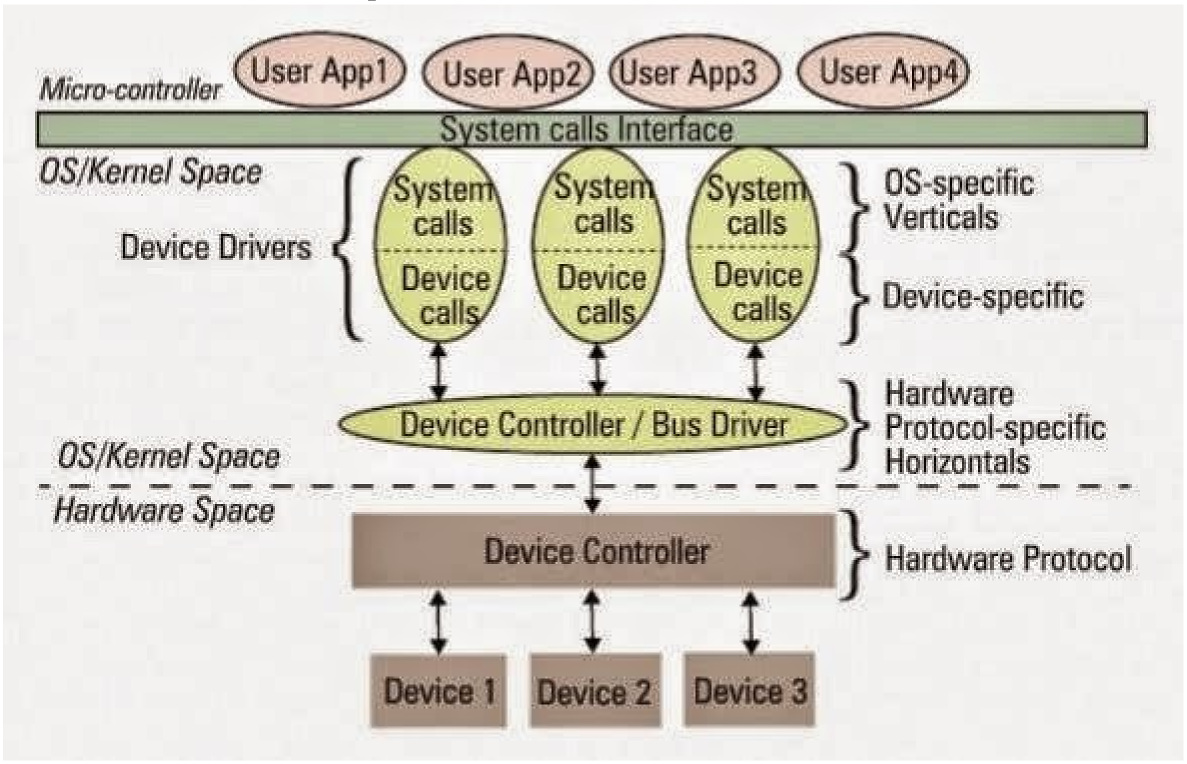
Device controllers are typically connected to the CPU through their respectively named buses (collection of physical lines) — for example, the PCI bus, the IDE bus, etc. In today's embedded world, we encounter more micro-controllers than CPUs; these are the CPU plus various device controllers built onto a single chip. This effective embedding of device controllers primarily reduces cost and space, making it suitable for embedded systems. In such cases, the buses are integrated into the chip itself. Does this change anything for the drivers, or more generically, on the software front?

The answer is, not much — except that the bus drivers corresponding to the embedded device controllers are now developed under the architecture-specific umbrella.

1. Drivers have two parts

Bus drivers provide hardware-specific interfaces for the corresponding hardware protocols, and are the bottom-most horizontal software layers of an operating system (OS). Over these sit the actual device drivers. These operate on the underlying devices using the horizontal layer interfaces, and hence are device-specific. However, the whole idea of writing these drivers is to provide an abstraction to the user, and so, at the other "end", these do provide an interface (which varies from OS to OS). In short, a device driver has two parts, which are: a) device-specific, and b) OS-specific. Refer to Figure 2.

Figure 2: Linux device driver partition



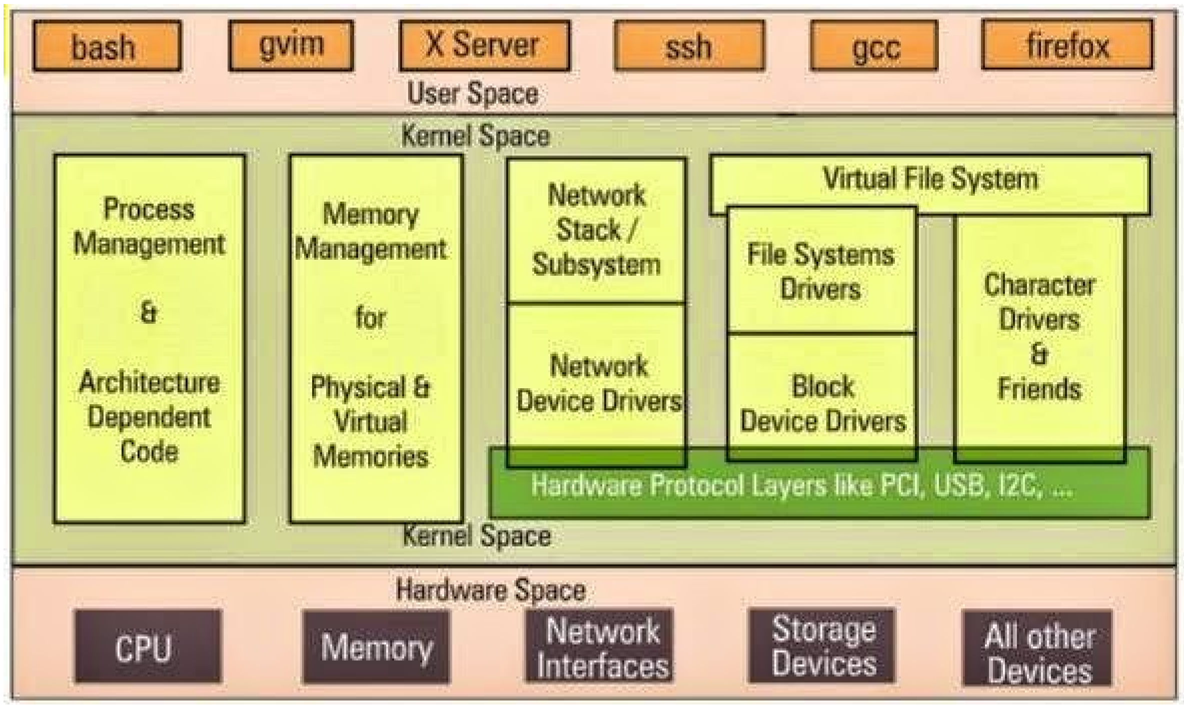
The device-specific portion of a device driver remains the same across all operating systems, and is more about understanding and decoding the device data sheets than software programming. A data sheet for a device is a document with technical details of the device, including its operation, performance, programming, etc. — in short a device user manual.

Later, I shall show some examples of decoding data sheets as well. However, the OS-specific portion is the one that is tightly coupled with the OS mechanisms of user interfaces, and thus differentiates a Linux device driver from a Windows device driver and from a MacOS device driver.

1. Verticals

In Linux, a device driver provides a "system call" interface to the user; this is the boundary line between the so-called kernel space and user-space of Linux, as shown in Figure 2. Figure 3 provides further classification.

Figure 3: Linux kernel overview



Based on the OS-specific interface of a driver, in Linux, a driver is broadly classified into three verticals:

- Packet-oriented or the network vertical
- Block-oriented or the storage vertical
- Byte-oriented or the character vertical

The CPU vertical and memory vertical, taken together with the other three verticals, give the complete overview of the Linux kernel, like any textbook definition of an OS: "An OS performs 5 management functions: CPU/process, memory, network, storage, device I/O." Though these two verticals could be classified as device drivers, where CPU and memory are the respective devices, they are treated differently, for many reasons.

These are the core functionalities of any OS, be it a micro-kernel or a monolithic kernel. More often than not, adding code in these areas is mainly a Linux porting effort, which is typically done for a new CPU or architecture. Moreover, the code in these two verticals cannot be loaded or unloaded on the fly, unlike the other three verticals. Henceforth, when we talk about Linux device drivers, we mean to talk only about the latter three verticals in Figure 3.

Let's get a little deeper into these three verticals. The network vertical consists of two parts: a) the network protocol stack, and b)the network interface card (NIC) device drivers, or simply network device drivers, which could be for Ethernet, Wi-Fi, or any other network horizontals. Storage, again, consists of two parts: a) File-system drivers, to decode the various formats on different partitions, and b) Block device drivers for various storage (hardware) protocols, i.e., horizontals like IDE, SCSI, MTD, etc.

With this, you may wonder if that is the only set of devices for which you need drivers (or for which Linux has drivers). Hold on a moment; you certainly need drivers for the whole lot of devices that interface with the system, and Linux does have drivers for them. However, their byte-oriented cessibility puts all of them under the character vertical — this is, in reality, the majority bucket. In fact, because of the vast number of drivers in this vertical, character drivers have been further sub-classified — so you have tty drivers, input drivers, console drivers, frame-buffer drivers, sound drivers, etc. The typical horizontals here would be RS232, PS/2, VGA, I2C, I2S, SPI, etc.

1. Multiple-vertical drivers

One final note on the complete picture (placement of all the drivers in the Linux driver ecosystem): the horizontals like USB, PCI, etc, span below multiple verticals. Why is that?

Simple — you already know that you can have a USB Wi-Fi dongle, a USB pen drive, and a USB-to-serial converter — all are USB, but come under three different verticals!

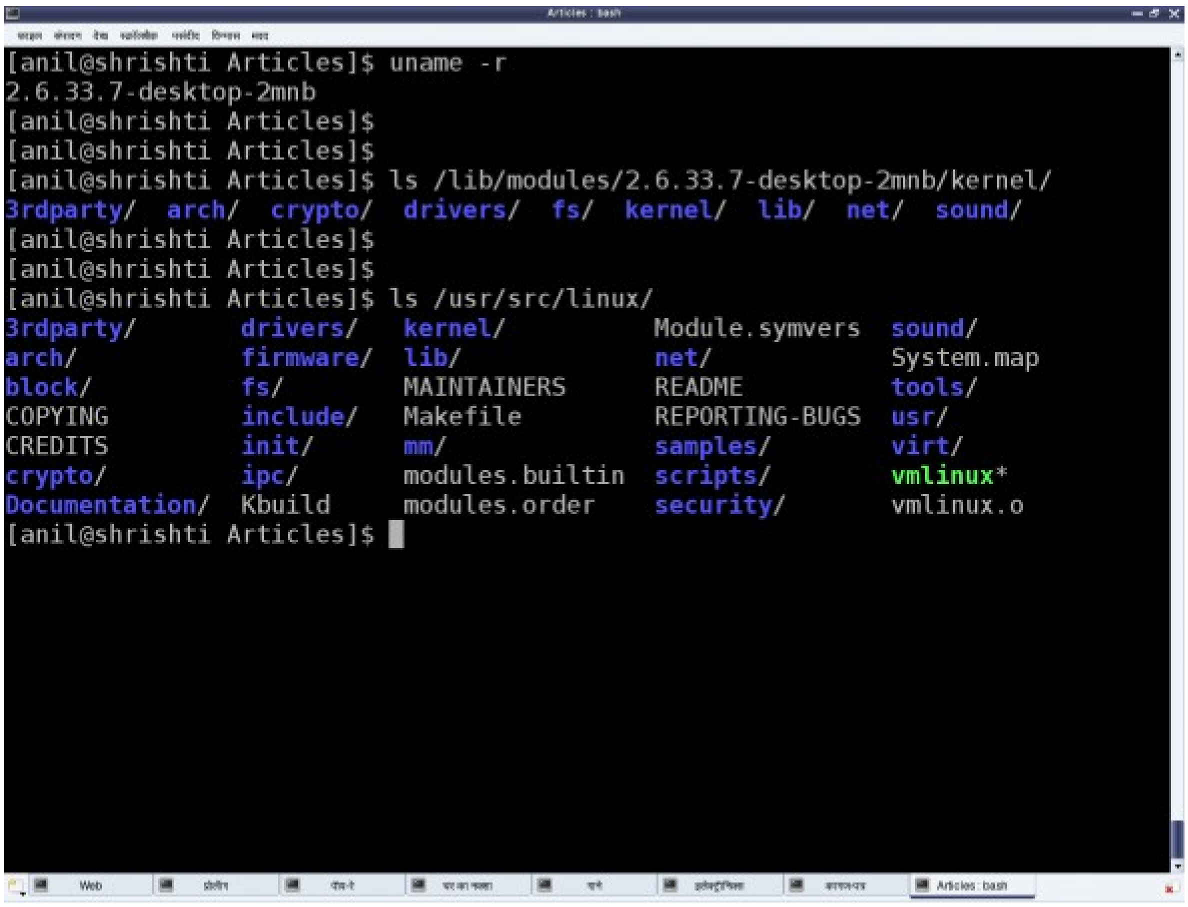
In Linux, bus drivers or the horizontals, are often split into two parts, or even two drivers: a) device controller-specific, and b) an abstraction layer over that for the verticals to interface, commonly called cores. A classic example would be the USB controller drivers ohci, ehci, etc., and the USB abstraction, usbcore.

Topic 2: Writing your first device driver

The drivers are called modules. This topic covers the handling of modules. How it is organized in Linux. Writing & build your first device driver.

1. Dynamically loading drivers

These dynamically loadable drivers are more commonly called modules and built into individual files with a .ko (kernel object) extension. Every Linux system has a standard place under the root of the file system (/) for all the pre-built modules. They are organised similar to the kernel source tree structure, under /lib/modules/, where would be the output of the command uname -ron the system, as shown in Figure 1.



```
[anil@shrishti Articles]$ uname -r
2.6.33.7-desktop-2mnb
[anil@shrishti Articles]$
[anil@shrishti Articles]$
[anil@shrishti Articles]$ ls /lib/modules/2.6.33.7-desktop-2mnb/kernel/
3rdparty/  arch/  crypto/  drivers/  fs/  kernel/  lib/  net/  sound/
[anil@shrishti Articles]$
[anil@shrishti Articles]$
[anil@shrishti Articles]$ ls /usr/src/linux/
3rdparty/       drivers/   kernel/          Module.symvers  sound/
arch/           firmware/  lib/             net/            System.map
block/          fs/        MAINTAINERS      README          tools/
COPYING         include/   Makefile         REPORTING-BUGS  usr/
CREDITS         init/      mm/              samples/        virt/
crypto/         ipc/       modules.builtin  scripts/        vmlinux*
Documentation/  Kbuild     modules.order    security/       vmlinux.o
[anil@shrishti Articles]$
```

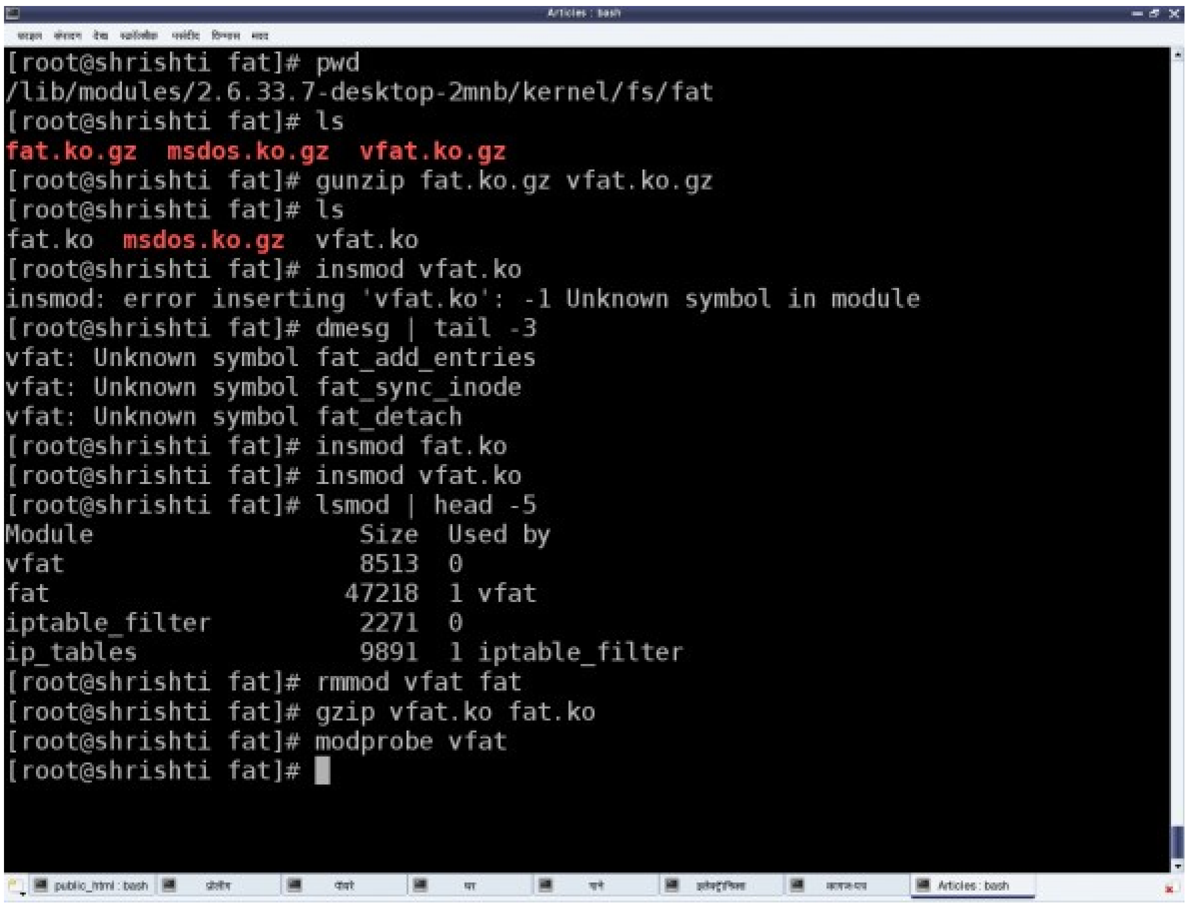
Figure 1: Linux pre-built modules

To dynamically load or unload a driver, use these commands, which reside in the /sbin directory, and must be executed with root privileges:

- lsmod — lists currently loaded modules
- insmod — inserts/loads the specified module file
- modprobe — inserts/loads the module, along with any dependencies
- rmmod — removes/unloads the module

Let’s look at the FAT filesystem-related drivers as an example. Figure 2 demonstrates this complete process of experimentation. The module files would be fat.ko, vfat.ko, etc., in the fat (vfat for older kernels) directory under /lib/modules/`uname -r`/kernel/fs. If they are in compressed .gz format, you need to uncompress them with gunzip, before you can insmodthem.

Figure 2: Linux module operations



```
[root@shrishti fat]# pwd
/lib/modules/2.6.33.7-desktop-2mnb/kernel/fs/fat
[root@shrishti fat]# ls
fat.ko.gz  msdos.ko.gz  vfat.ko.gz
[root@shrishti fat]# gunzip fat.ko.gz vfat.ko.gz
[root@shrishti fat]# ls
fat.ko  msdos.ko.gz  vfat.ko
[root@shrishti fat]# insmod vfat.ko
insmod: error inserting 'vfat.ko': -1 Unknown symbol in module
[root@shrishti fat]# dmesg | tail -3
vfat: Unknown symbol fat_add_entries
vfat: Unknown symbol fat_sync_inode
vfat: Unknown symbol fat_detach
[root@shrishti fat]# insmod fat.ko
[root@shrishti fat]# insmod vfat.ko
[root@shrishti fat]# lsmod | head -5
Module                  Size  Used by
vfat                    8513  0
fat                    47218  1 vfat
iptable_filter          2271  0
ip_tables               9891  1 iptable_filter
[root@shrishti fat]# rmmod vfat fat
[root@shrishti fat]# gzip vfat.ko fat.ko
[root@shrishti fat]# modprobe vfat
[root@shrishti fat]# 
```

The vfat module depends on the fat module, so fat.ko needs to be loaded first. To automatically perform decompression and dependency loading, use modprobe instead. Note that you shouldn't specify the .ko extension to the module's name, when using the modprobe command. rmmod is used to unload the modules.

1. Our first Linux driver

Before we write our first driver, let's go over some concepts. A driver never runs by itself. It is similar to a library that is loaded for its functions to be invoked by a running application. It is written in C, but lacks a main() function. Moreover, it will be loaded/linked with the kernel, so it needs to be compiled in a similar way to the kernel, and the header files you can use are only those from the kernel sources, not from the standard /usr/include.

One interesting fact about the kernel is that it is an object-oriented implementation in C, as we will observe even with our first driver. Any Linux driver has a constructor and a destructor. The module's constructor is called when the module is successfully loaded into the kernel, and the destructor when rmmod succeeds in unloading the module. These two are like normal functions in the driver, except that they are specified as the *init* and *exit* functions, respectively, by the macros module_init() and module_exit(), which are defined in the kernel header module.h.

```
1 /* ofd.c – Our First Driver code */
2 #include
3 #include
4 #include
5 static int __init ofd_init(void) /* Constructor */
```

```
{
printk(KERN_INFO "Namaskar: ofd registered");
return 0;
}
static void __exit ofd_exit(void) /* Destructor */
{
printk(KERN_INFO "Alvida: ofd unregistered");
}
module_init(ofd_init);
module_exit(ofd_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anil Kumar Pugalia
MODULE_DESCRIPTION("Our First Driver");
```

Given above is the complete code for our first driver; let's call it ofd.c. Note that there is no stdio.h (a user-space header); instead, we use the analogous kernel.h (a kernel space header). printk() is the equivalent of printf(). Additionally, version.h is included for the module version to be compatible with the kernel into which it is going to be loaded. The MODULE_* macros populate module-related information, which acts like the module's "signature".

1. Building our first Linux driver

Once we have the C code, it is time to compile it and create the module file ofd.ko. We use the kernel build system to do this. The following Makefile invokes the kernel's build system from the kernel source, and the kernel's Makefile will, in turn, invoke our first driver's Makefile to build our first driver.

To build a Linux driver, you need to have the kernel source (or, at least, the kernel headers) installed on your system. The kernel source is assumed to be installed at /usr/src/linux. If it's at any other location on your system, specify the location in the KERNEL_SOURCE variable in this Makefile.

```
# Makefile – makefile of our first driver
# if KERNELRELEASE is defined, we've
been invoked from the
# kernel build system and can use its language.
ifneq (${KERNELRELEASE},)
obj-m := ofd.o
# Otherwise we were called directly from the
command line.
# Invoke the kernel build system.
else
KERNEL_SOURCE := /usr/src/linux
PWD := $(shell pwd)
default:
${MAKE} -C ${KERNEL_SOURCE}
SUBDIRS=${PWD} modules
clean:
${MAKE} -C ${KERNEL_SOURCE}
SUBDIRS=${PWD} clean
endif
```

With the C code (ofd.c) and Makefile ready, all we need to do is invoke make to build our first driver (ofd.ko).

```
$ make
make -C /usr/src/linux
SUBDIRS=... modules
make[1]: Entering directory
`/usr/src/linux'
CC [M] .../ofd.o
Building modules, stage 2.
MODPOST 1 modules
CC .../ofd.mod.o
LD [M] .../ofd.ko
make[1]: Leaving directory
`/usr/src/linux'
```

Once we have the ofd.ko file, perform the usual steps as the root user, or with sudo.

```
# su
# insmod
ofd.ko
# lsmod | head
-10
```

lsmod should show you the ofd driver loaded.

Topic 3: Kernel C extras in a Linux driver

This topic cover the message logging, message priority, why one need to build the driver along with kernel code?, Kernel specific GCC extension and kernel functions return guidelines.

1. Kernel message logging

As far as parameters are concerned, printf and printk are the same, except that when programming for the kernel, we don't bother about the float formats %f, %lf and the like. However, unlike printf, printk is not designed to dump its output to some console.

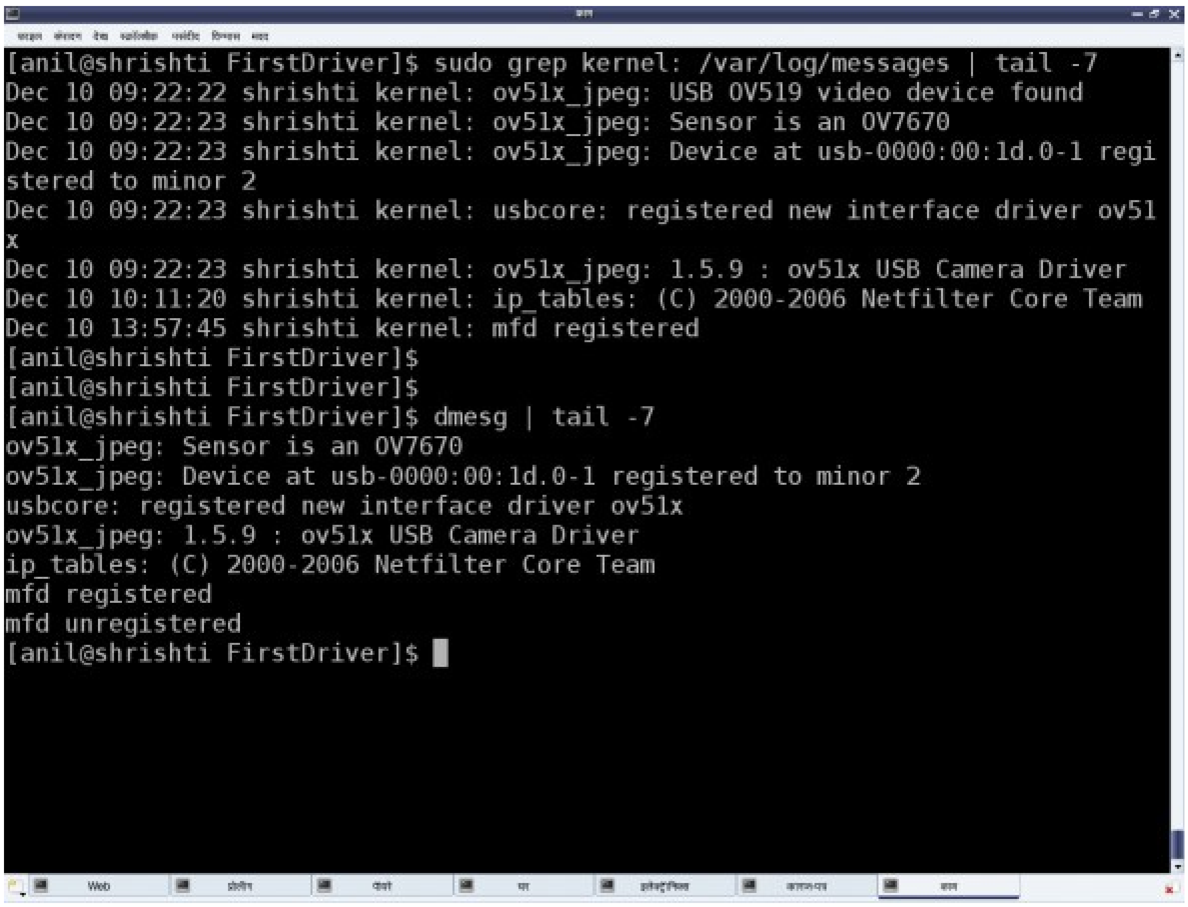
In fact, it cannot do so; it is something in the background, and executes like a library, only when triggered either from hardware-space or user-space. All printk calls put their output into the (log) ring buffer of the kernel. Then, the syslog daemon running in user-space picks them up for final processing and redirection to various devices, as configured in the configuration file /etc/syslog.conf.

You must have observed the out-of-place macro KERN_INFO, in the printk calls. That is actually a constant string, which gets concatenated with the format string after it, into a single string. Note that there is no comma (,) between them; they are not two separate arguments. There are eight such macros defined in linux/kernel.h in the kernel source, namely:

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

Now depending on these log levels (i.e., the first three characters in the format string), the syslog user-space daemon redirects the corresponding messages to their configured locations. A typical destination is the log file /var/log/messages, for all log levels. Hence, all the printk outputs are, by default, in that file. However, they can be configured differently — to a serial port (like /dev/ttyS0), for instance, or to all consoles, like what typically happens for KERN_EMERG.

Now, /var/log/messages is buffered, and contains messages not only from the kernel, but also from various daemons running in user-space. Moreover, this file is often not readable by a normal user. Hence, a user-space utility, dmesg, is provided to directly parse the kernel ring buffer, and dump it to standard output. Figure 1 shows snippets from the two.



```
[anil@shrishti FirstDriver]$ sudo grep kernel: /var/log/messages | tail -7
Dec 10 09:22:22 shrishti kernel: ov51x_jpeg: USB OV519 video device found
Dec 10 09:22:23 shrishti kernel: ov51x_jpeg: Sensor is an OV7670
Dec 10 09:22:23 shrishti kernel: ov51x_jpeg: Device at usb-0000:00:1d.0-1 regi
stered to minor 2
Dec 10 09:22:23 shrishti kernel: usbcore: registered new interface driver ov51
x
Dec 10 09:22:23 shrishti kernel: ov51x_jpeg: 1.5.9 : ov51x USB Camera Driver
Dec 10 10:11:20 shrishti kernel: ip_tables: (C) 2000-2006 Netfilter Core Team
Dec 10 13:57:45 shrishti kernel: mfd registered
[anil@shrishti FirstDriver]$
[anil@shrishti FirstDriver]$
[anil@shrishti FirstDriver]$ dmesg | tail -7
ov51x_jpeg: Sensor is an OV7670
ov51x_jpeg: Device at usb-0000:00:1d.0-1 registered to minor 2
usbcore: registered new interface driver ov51x
ov51x_jpeg: 1.5.9 : ov51x USB Camera Driver
ip_tables: (C) 2000-2006 Netfilter Core Team
mfd registered
mfd unregistered
[anil@shrishti FirstDriver]$ 
```

Figure 1: Kernel's message logging

1. Kernel-specific GCC extensions

Kernel C is not "weird C", but just standard C with some additional extensions from the C compiler, GCC. Macros __init and __exit are just two of these extensions. However, these do not have any relevance in case we are using them for a dynamically loadable driver, but only when the same code gets built into the kernel. All functions marked with __init get placed inside the init section of the kernel image automatically, by GCC, during kernel compilation; and all functions marked with __exit are placed in the exit section of the kernel image.

What is the benefit of this? All functions with __init are supposed to be executed only once during bootup (and not executed again till the next bootup). So, once they are executed during bootup, the kernel frees up RAM by removing them (by freeing the init section). Similarly, all functions in the exit section are supposed to be called during system shutdown.

Now, if the system is shutting down anyway, why do you need to do any cleaning up? Hence, the exit section is not even loaded into the kernel — another cool optimisation. This is a beautiful example of how the kernel and GCC work hand-in-hand to achieve a lot of optimisation, and many other tricks that we will see as we go along. And that is why the Linux kernel can only be compiled using GCC-based compilers — a closely knit bond.

1. The kernel function’s return guidelines

Any kernel function needing error handling, typically returns an integer-like type — and the return value again follows a guideline. For an error, we return a negative number: a minus sign appended with a macro that is available through the kernel header linux/errno.h, that includes the various error number headers under the kernel sources — namely, asm/errno.h, asm-generic/errno.h, asm-generic/errno-base.h.

For success, zero is the most common return value, unless there is some additional information to be provided. In that case, a positive value is returned, the value indicating the information, such as the number of bytes transferred by the function.

1. Kernel C = pure C

Standard C is pure C — just the language. The headers are not part of it. Those are part of the standard libraries built in for C programmers, based on the concept of reusing code.

Does that mean that all standard libraries, and hence, all ANSI standard functions, are not part of “pure” C? Yes, that’s right. Then, was it really tough coding the kernel?

Well, not for this reason. In reality,kernel developers have evolved their own set of required functions, which are all part of the kernel code. The printk function is just one of them. Similarly, many string functions, memory functions, and more, are all part of the kernel source, under various directories like kernel, ipc, lib, and so on, along with the corresponding headers under the include/linux directory.

That is why we need to have the kernel source to build a driver.

“If not the complete source, at least the headers are a must. And that is why we have separate packages to install the complete kernel source, or just the kernel headers.

use apt-get utility to fetch the source — possibly apt-get install linux-source.

Linux Device Driver Interview Question

Linux Device Model (LDM)

Explain about the Linux Device Model (LDM)?

Explain about about ksets, kobjects and ktypes. How are they related?

Questions about sysfs.

Linux Boot Sequence

Explain about the Linux boot sequence in case of ARM architecture?

How are the command line arguments passed to Linux kernel by the u-boot (bootloader)?

Explain about ATAGS?

Explain about command line arguments that are passed to linux kernel and how/where they are
parsed in kernel code?

Explain about device tree.

Interrupts in Linux

Explain about the interrupt mechanims in linux?

What are the APIs that are used to register an interrupt handler?
How do you register an interrupt handler on a shared IRQ line?

Explain about the flags that are passed to request_irq().

Explain about the internals of Interrupt handling in case of Linux running on ARM.
Solution

What are the precautions to be taken while writing an interrupt handler?

Explain interrupt sequence in detail starting from ARM to registered interrupt handler.

What is bottom half and top half.

What is request_threaded_irq()

If same interrupts occurs in two cpu how are they handled?

How to synchronize data between 'two interrupts' and 'interrupts and process'.

How are nested interrupts handled?

How is task context saved during interrupt.

Bottom-half Mechanisms in Linux

What are the different bottom-half mechanisms in Linux?
Softirq, Tasklet and Workqueues

What are the differences between Softirq/Tasklet and Workqueue? Given an example what you
prefer to use?

What are the differences between softirqs and tasklets?

- Softirq is guaranteed to run on the CPU it was scheduled on, where as tasklets don’t have that guarantee.
- The same tasklet can't run on two separate CPUs at the same time, where as a softirq can.

When are these bottom halfs executed?

Explain about the internal implementation of softirqs?
http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-1-softirqs.html

Explain about the internal implementation of tasklets?
http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-2-tasklets.html

Explain about the internal implementation of workqueues?
http://linuxblore.blogspot.in/2013/01/workqueues-in-linux.html

Explain about the concurrent work queues.

Linux Memory Management

What are the differences between vmalloc and kmalloc? Which is preferred to use in device drivers?

What are the differences between slab allocator and slub allocator?

What is boot memory allocator?

How do you reserve block of memory?
What is virtual memory and what are the advanatages of using virtual memory?

What's paging and swapping?

Is it better to enable swapping in embedded systems? and why?

What is the page size in Linux kernel in case of 32-bit ARM architecture?

What is page frame?

What are the different memory zones and why does different zones exist?

What is high memory and when is it needed?

Why is high memory zone not needed in case of 64-bit machine?

How to allocate a page frame from high memory?

In ARM, an abort exception if generated, if the page table doesn't contain a virtual to physical map for a particular page. How exactly does the MMU know that a virtual to physical map is present in the pagetable or not?

A Level-1 page table entry can be one of four possible types. The 1st type is given below:

- A fault entry that generates an abort exception. This can be either a prefetch or data abort, depending on the type of access. This effectively indicates virtual addresses that are unmapped.

In this case the bit [0] and [1] are set to 0. This is how the MMU identifies that it's a fault entry.

Same is the case with Level-2 page table entry.

Does the Translation Table Base Address (TTBR) register, Level 1 page table and Level 2 page table contain Physical addresses or Virtual addresses?

TTBR: Contain physical address of the pgd base
Level 1 page table (pgd): Physical address pointing to the pte base
Level 2 page table (pte): Physical address pointing to the physical page frame

Since page tables are in kernel space and kernel virtual memory is mapped directly to RAM. Using just an easy macro like __virt_to_phys(), we can get the physical address for the pgd base or pte base or pte entry.

Kernel Synchronization

Why do we need synchronization mechanisms in Linux kernel?
What are the different synchonization mechanisms present in Linux kernel?
What are the differences between spinlock and mutex?
What is lockdep?
Which synchronization mechanism is safe to use in interrupt context and why?
Explain about the implementation of spinlock in case of ARM architecture.
Solution
Explain about the implementation of mutex in case of ARM architecture.
Solution

Explain about the notifier chains.
Solution
Explain about RCU locks and when are they used?

Explain about RW spinlocks locks and when are they used?

Which are the synchronization technoques you use 'between processes', 'between processe and interrupt' and 'between interrupts'; why and how ?

What are the differences between semaphores and spinlocks?

Process Management and Process Scheduling

What are the different schedulers class present in the linux kernel?

How to create a new process?

What is the difference between fork( ) and vfork( )?

Which is the first task what is spawned in linux kernel?

What are the processes with PID 0 and PID 1?
PID 0 - idle task
PID 1 - init

How to extract task_struct of a particular process if the stack pointer is given?

How does scheduler picks particular task?

When does scheduler picks a task?

How is timeout managed?

How does load balancing happens?

Explain about any scheduler class?

Explain about wait queues and how they implemented? Where and how are they used?

What is process kernel stack and process user stack? What is the size of each and how are they allocated?

Why do we need seperate kernel stack for each process?

What all happens during context switch?

What is thread_info? Why is it stored at the end of kernel stack?

What is the use of preempt_count variable?

What is the difference between interruptible and uninterruptible task states?

How processes and threads are created? (from user level till kernel level)

How is virtual run time (vruntime) calculated?
Solution

Timers and Time Management

What are jiffies and HZ?
What is the initial value of jiffies when the system has started?

Explain about HR timers and normal timers?

On what hardware timers, does the HR timers are based on?

How to declare that a specific hardware timers is used for kernel periodic timer interrupt used by the scheduler?

How software timers are implemented?

Power Management in Linux

Explain about cpuidle framework.

Explain about cpufreq framework.

Explain about clock framework.

Explain about regulator framework.

Explain about suspened and resume framwork.

Explain about early suspend and late resume.

Explain about wakelocks.

Linux Kernel Modules

How to make a module as loadable module?

How to make a module as in-built module?

Explain about Kconfig build system?

Explain about the init call mechanism.
Solution

What is the difference between early init and late init?
Early init:

- Early init functions are called when only the boot processor is online.
- Run before initializing SMP.
- Only for built-in code, not modules.

Late init:

- Late init functions are called _after_ all the CPUs are online.

Linux Kernel Debugging

What is Oops and kernel panic?
Does all Oops result in kernel panic?
What are the tools that you have used for debugging the Linux kernel?
What are the log levels in printk?
Can printk's be used in interrupt context?
How to print a stack trace from a particular function?
What's the use of early_printk( )?
Explan about the various gdb commands.

Miscellaneous

How are the atomic functions implemented in case of ARM architecture?

Solution

How is container_of( ) macro implemented?

Explain about system call flow in case of ARM Linux.

What 's the use of __init and __exit macros?

How to ensure that init function of a partiuclar driver was called before our driver's init function is called (assume that both these drivers are built into the kenrel image)?

What's a segementation fault and what are the scenarios in which segmentation fault is triggered?

If the scenarios which triggers the segmentation fault has occured, how the kernel identifies it and what are the actions that the kernel takes?

Difference Between Processes and Threads

A process is an executing instance of an application. What does that mean? Well, for example, when you double-click the Microsoft Word icon, you start a process that runs Word. A thread is a path of execution *within* a process. Also, a process can contain multiple threads. When you start Word, the operating system creates a process and begins executing the primary thread of that process.

It's important to note that a thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a 'lightweight' process. Thus, the essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications.

Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication – is quite difficult and resource-intensive.

MultiThreading

Threads, of course, allow for multi-threading. A common example of the advantage of multithreading is the fact that you can have a word processor that prints a document using a background thread, but at the same time another thread is running that accepts user input, so that you can type up a new document.

If we were dealing with an application that uses only one thread, then the application would only be able to do one thing at a time – so printing and responding to user input at the same time would not be possible in a single threaded application.

Each process has it’s own address space, but the threads within the same process share that address space. Threads also share any other resources within that process. This means that it’s very easy to share data amongst threads, but it’s also easy for the threads to step on each other, which can lead to bad things.

Multithreaded programs must be carefully programmed to prevent those bad things from happening. Sections of code that modify data structures shared by multiple threads are called critical sections. When a critical section is running in one thread it’s extremely important that no other thread be allowed into that critical section. This is called synchronization, which we wont get into any further over here. But, the point is that multithreading requires careful programming.

Also, context switching between threads is generally less expensive than in processes. And finally, the overhead (the cost of communication) between threads is very low relative to processes.

Here’s a summary of the differences between threads and processes:

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.

3. Threads are considered lightweight because they use far less resources than processes.

4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other. This is really another way of stating #2 above.

5. A process can consist of multiple threads

Spin Lock In Linux Kernel

Why are spin locks good choices in Linux Kernel Design instead of something more common in userland code, such as semaphore or mutex?

1. When spinlock is used ?

Ans: In the following situations.

1. The thread that holds the lock is not allowed to sleep.
2. The thread that is waiting for a lock does not sleep, but spins in a tight loop.

When properly used, spinlock can give higher performance than semaphore. Ex: Intrrrupt handler.

2. What are the rules to use spinlocks?

Ans:

Rule - 1: Any code that holds the spinlock, can not relinquish the processor for any reason except to service interrupts (sometimes not even then). So code holding spinlock can not sleep.

Reason: suppose your driver holding spinlock goes to sleep. Ex: calls function `copy_from_user()` or `copy_to_user()`, or kernel preemption kicks in so higher priority process pushed your code aside. Effectively the process relinquishes the CPU holding spinlock. Now we do not know when the code will release the lock. If some other thread tries to obtain the same lock, it would spin for very long time. In the worst case it would result in deedlock.
Kernel preemption case is handled by the spinlock code itself. Anytime kernel code holds a spinlock, preemption is disabled on the relevant processor. Even uniprocessor system must disable the preemption in this way.

Rule - 2: Disable interrupts on the local CPU, while the spinlock is held.

Reason: Support your driver take a spinlock that control access to the device and then issues an interrupt. This causes the interrupt handler to run. Now the interrupt handler also needs the lock to access the device. If the interrupt handler runs on the same processor, it will start spinning. The driver code also can not run to release the lock. SO the processor will spin for ever.

Rule - 3: Spinlocks must be held for the minimum time possible.

Reason: Long lock hold times also keeps the current processor from scheduling, meaning a higher priority process may have to wait to get the CPU.
So it impacts kernel latency (time a process may have to wait to be scheduled). Typically spinlocks should be held for the time duration, less than that CPU takes to do a contex switch between threads.

Rule -4: if you have semaphores and spinlocks both to be taken. Then take semaphore first and then spinlock.

How do function pointers in C work

Function Pointers and Callbacks in C

A pointer is a special kind of variable that holds the address of another variable. The same concept applies to function pointers, except that instead of pointing to variables, they point to functions. If you declare an array, say, int a[10]; then the array name a will in most contexts (in an expression or passed as a function parameter) “decay” to a non-modifiable pointer to its first element (even though pointers and arrays are not equivalent while declaring/defining them, or when used as operands of the sizeof operator). In the same way, for int func();, func decays to a non-modifiable pointer to a function. You can think of func as a const pointer for the time being.
But can we declare a non-constant pointer to a function? Yes, we can — just like we declare a non-constant pointer to a variable:

```
int (*ptrFunc) ();
```

Here, ptrFunc is a pointer to a function that takes no arguments and returns an integer. DO NOT forget to put in the parenthesis, otherwise the compiler will assume that ptrFunc is a normal function name, which takes nothing and returns a pointer to an integer.
Let’s try some code. Check out the following simple program:

```

#include
/* function prototype */
int func(int, int);
int main(void)
{
    int result;
    /* calling a function named func */
    result = func(10,20);
    printf("result = %d\n",result);
    return 0;
}
/* func definition goes here */
int func(int x, int y)
{
return x+y;
}

```

As expected, when we compile it with gcc -g -o example1 example1.c and invoke it with ./example1, the output is as follows:

```
result = 30
```

The above program calls func() the simple way. Let’s modify the program to call using a pointer to a function. Here’s the changed main() function:

```


#include
int func(int, int);
int main(void)
{
    int result1,result2;
    /* declaring a pointer to a function which takes
       two int arguments and returns an integer as result */
    int (*ptrFunc)(int,int);
    /* assigning ptrFunc to func's address */
    ptrFunc=func;
    /* calling func() through explicit dereference */
    result1 = (*ptrFunc)(10,20);
    /* calling func() through implicit dereference */
    result2 = ptrFunc(10,20);
    printf("result1 = %d result2 = %d\n",result1,result2);
    return 0;
}
int func(int x, int y)
{
    return x+y;
}


```

The output has no surprises:

```
result1 = 30 result2 = 30
```

A Simple Call Back Function

At this stage, we have enough knowledge to deal with function callbacks. According to Wikipedia, “In computer programming, a callback is a reference to executable code, or a piece of executable code, that is passed as an argument to other code. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer.” Let’s try one simple program to demonstrate this. The complete program has three files: callback.c, reg_callback.h and reg_callback.c.

```

/* callback.c */
#include
#include"reg_callback.h"
/* callback function definition goes here */
void my_callback(void)
{
   printf("inside my_callback\n");
}
int main(void)
{
   /* initialize function pointer to
   my_callback */
   callback ptr_my_callback=my_callback;
   printf("This is a program demonstrating function callback\n");
   /* register our callback function */
   register_callback(ptr_my_callback);
   printf("back inside main program\n");
   return 0;
}

```

```
/* reg_callback.h */
typedef void (*callback)(void);
void register_callback(callback ptr_reg_callback);
```

```
/* reg_callback.c */
#include
#include"reg_callback.h"
/* registration goes here */
void register_callback(callback ptr_reg_callback)
{
   printf("inside register_callback\n");
   /* calling our callback function my_callback */
   (*ptr_reg_callback)();
}
```

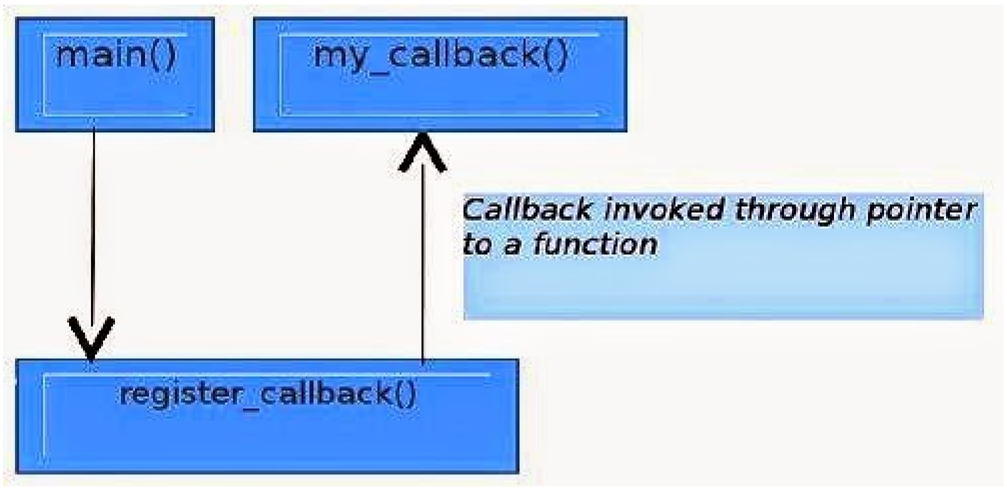
Compile, link and run the program with gcc -Wall -o callback callback.c reg_callback.c and ./callback:

```
This is a program demonstrating function callback
inside register_callback
inside my_callback
back inside main program
```

The code needs a little explanation. Assume that we have to call a callback function that does some useful work (error handling, last-minute clean-up before exiting, etc.), after an event occurs in another part of the program. The first step is to register the callback function, which is just passing a function pointer as an argument to some other function (e.g., register_callback) where the callback function needs to be called.

We could have written the above code in a single file, but have put the definition of the callback function in a separate file to simulate real-life cases, where the callback function is in the top layer and the function that will invoke it is in a different file layer. So the program flow is like what can be seen in Figure 1.

Figure 1: Program flow



The higher layer function calls a lower layer function as a normal call and the callback mechanism allows the lower layer function to call the higher layer function through a pointer to a callback function.
This is exactly what the Wikipedia definition states.

Use of callback functions

One use of callback mechanisms can be seen here:

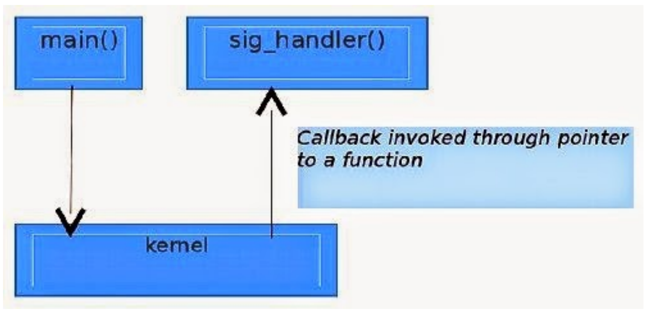
```c


/ * This code catches the alarm signal generated from the kernel
   Asynchronously */
#include
#include
#include
struct sigaction act;
/* signal handler definition goes here */
void sig_handler(int signo, siginfo_t *si, void *ucontext)
{
  printf("Got alarm signal %d\n",signo);
  /* do the required stuff here */
}
int main(void)
{
   act.sa_sigaction = sig_handler;
   act.sa_flags = SA_SIGINFO;
   /* register signal handler */
   sigaction(SIGALRM, &act, NULL);
   /* set the alarm for 10 sec */
   alarm(10);
   /* wait for any signal from kernel */
   pause();
   /* after signal handler execution */
   printf("back to main\n");
   return 0;
}


```

Signals are types of interrupts that are generated from the kernel, and are very useful for handling asynchronous events. A signal-handling function is registered with the kernel, and can be invoked asynchronously from the rest of the program when the signal is delivered to the user process. Figure 2 represents this flow.

Figure 2: Kernel callback



Callback functions can also be used to create a library that will be called from an upper-layer program, and in turn, the library will call user-defined code on the occurrence of some event. The following source code (insertion_main.c, insertion_sort.c and insertion_sort.h), shows this mechanism used to implement a trivial insertion sort library. The flexibility lets users call any comparison function they want.

```
/* insertion_sort.h */
typedef int (*callback)(int, int);
void insertion_sort(int *array, int n, callback comparison);
```

```
/* insertion_main.c */
#include
#include
#include"insertion_sort.h"
int ascending(int a, int b)
{
    return a > b;
}
int descending(int a, int b)
{
    return a < b;
}
int even_first(int a, int b)
{
    /* code goes here */
}
int odd_first(int a, int b)
{
    /* code goes here */
}
int main(void)
{
    int i;
    int choice;
    int array[10] = {22,66,55,11,99,33,44,77,88,0};
    printf("ascending 1: descending 2: even_first 3: odd_first 4: quit 5\n");
    printf("enter your choice = ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            insertion_sort(array,10, ascending);
            break;
        case 2:
            insertion_sort(array,10, descending);
         case 3:
            insertion_sort(array,10, even_first);
            break;
        case 4:
            insertion_sort(array,10, odd_first);
            break;
        case 5:
            exit(0);
        default:
            printf("no such option\n");
    }
```

```
/* insertion_sort.c */
#include"insertion_sort.h"
void insertion_sort(int *array, int n, callback comparison)
{
   int i, j, key;
   for(j=1; j<=n-1;j++)
   {
      key=array[j];
      i=j-1;
      while(i >=0 && comparison(array[i], key))
      {
         array[i+1]=array[i];
         i=i-1;
      }
      array[i+1]=key;
   }
```

How to add your linux driver module in a kernel

Use below steps to add your driver module in linux kernel. by adding your driver as a module you can load/unload it at your convenience and will not be a part of your kernel image. I have used hello driver to explain it.

Steps to follow
===========

1). Create your module directory in /kernel/drivers

Eg. mkdir hellodriver

2). Create your file inside /kernel/drivers/hellodriver/ and add below functions and save it.

Write your driver init and exit function in hello.c file.

```
#include
#include
static int __init hello_module_init(void)
{
printk ("hello test app module init");
return 0;
}
static int __exit hello_module_cleanup(void)
{
```

```
printk("hello test app module cleanup ");
return 0;
}
module_init(hello_module_init);
module_exit(hello_module_cleanup);
MODULE_LICENSE("GPL");
```

———-

3). Create empty Kconfig file and Makefile in /kernel/drivers/hellodriver/

4). Add below entries in Kconfig

```
config HELLOTEST_APP
tristate "HelloTest App"
depends on ARM
default m
help
hellotest app
```

5). Add below entries in Makefile

```
obj-$(CONFIG_HELLOTEST_APP) +=hello.o
```

6). Modify the /kernel/drivers Kconfig and Makefile to support your module

7). Add below entry in /kernel/drivers/Kconfig file

```
source "drivers/hellodriver/Kconfig"
```

8). Add below entry in /kernel/drivers/Makefile file

```
obj-$(CONFIG_HELLOTEST_APP) +=hellodriver/
```

9). Now go to kernel directory and give

```
make menuconfig ARCH=arm
```

Verify that your driver module entry is visible under Device Drivers —>
For module entry it shows
Now recompile the kernel with your requirement and give

```
sudo make ARCH=arm CROSS_COMPILE=your toolchain path-
```

10). Check the hello.o and hello.ko files are generated at /kernel/drivers/hellodriver/

If you want to make your module as a part of kernel image then you only need to change <*> HelloTest App the option in menuconfig and recompile the kernel.
If you don't want to make your module as a part of kernel image then you only need to change <> HelloTest App the option in menuconfig and recompile the kernel.
This is a very simple example of adding a module in a kernel.

<u>How to Load/Unload module/s from the user space:</u>

In user space, you can load the module as root by typing the following into the command line. insmod load the module into kernel space.

```
# insmod hello.ko
```

To see the module loaded you can do following:

```
# lsmod
```

To remove your module from the kernel space you can do followig:

```
#rmmod hello.ko
```

Writing I2c Client Document

This is a small guide for those who want to write kernel drivers for I2C or SMBus devices, using Linux as the protocol host/master (not slave).

To set up a driver, you need to do several things. Some are optional, and some things can be done slightly or completely different. Use this as a guide, not as a rule book!

General remarks
===============

Try to keep the kernel namespace as clean as possible. The best way to do this is to use a unique prefix for all global symbols. This is especially important for exported symbols, but it is a good idea to do it for non-exported symbols too. We will use the prefix `foo_' in this tutorial.

The driver structure
====================

Usually, you will implement a single driver structure, and instantiate all clients from it. Remember, a driver structure contains general access routines, and should be zero-initialized except for fields with data you provide. A client structure holds device-specific information like the driver model device node, and its I2C address.

```
static struct i2c_device_id foo_idtable[] = {
 { "foo", my_id_for_foo },
 { "bar", my_id_for_bar },
 { }
};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static struct i2c_driver foo_driver = {
 .driver = {
  .name = "foo",
 },
 .id_table = foo_idtable,
 .probe  = foo_probe,
 .remove  = foo_remove,
 /* if device autodetection is needed: */
 .class  = I2C_CLASS_SOMETHING,
 .detect  = foo_detect,
 .address_list = normal_i2c,

 .shutdown = foo_shutdown, /* optional */
 .suspend = foo_suspend, /* optional */
 .resume  = foo_resume, /* optional */
 .command = foo_command, /* optional, deprecated */
}
```

The name field is the driver name, and must not contain spaces. It should match the module name (if the driver can be compiled as a module), although you can use MODULE_ALIAS (passing "foo" in this example) to add another name for the module. If the driver name doesn't match the module name, the module won't be automatically loaded (hotplug/coldplug).

All other fields are for call-back functions which will be explained below.

Extra client data

Each client structure has a special `data' field that can point to any structure at all. You should use this to keep device-specific data.

```
 /* store the value */
 void i2c_set_clientdata(struct i2c_client *client, void *data);

 /* retrieve the value */
 void *i2c_get_clientdata(const struct i2c_client *client);
```

Note that starting with kernel 2.6.34, you don't have to set the `data' field to NULL in remove() or if probe() failed anymore. The i2c-core does this automatically on these occasions. Those are also the only times the core will touch this field.

Accessing the client

Let's say we have a valid client structure. At some time, we will need to gather information from the client, or write new information to the client.

I have found it useful to define foo_read and foo_write functions for this.
For some cases, it will be easier to call the i2c functions directly,
but many chips have some kind of register-value idea that can easily
be encapsulated.

The below functions are simple examples, and should not be copied
literally.

```
int foo_read_value(struct i2c_client *client, u8 reg)
{
 if (reg < 0x10) /* byte-sized register */
  return i2c_smbus_read_byte_data(client, reg);
 else  /* word-sized register */
  return i2c_smbus_read_word_data(client, reg);
}

int foo_write_value(struct i2c_client *client, u8 reg, u16 value)
{
 if (reg == 0x10) /* Impossible to write - driver error! */
  return -EINVAL;
 else if (reg < 0x10) /* byte-sized register */
  return i2c_smbus_write_byte_data(client, reg, value);
 else   /* word-sized register */
  return i2c_smbus_write_word_data(client, reg, value);
}
```

Probing and attaching

The Linux I2C stack was originally written to support access to hardware
monitoring chips on PC motherboards, and thus used to embed some assumptions
that were more appropriate to SMBus (and PCs) than to I2C. One of these
assumptions was that most adapters and devices drivers support the SMBUS_QUICK
protocol to probe device presence. Another was that devices and their drivers
can be sufficiently configured using only such probe primitives.

As Linux and its I2C stack became more widely used in embedded systems
and complex components such as DVB adapters, those assumptions became more
problematic. Drivers for I2C devices that issue interrupts need more (and
different) configuration information, as do drivers handling chip variants
that can't be distinguished by protocol probing, or which need some board
specific information to operate correctly.

Device/Driver Binding

System infrastructure, typically board-specific initialization code or
boot firmware, reports what I2C devices exist. For example, there may be
a table, in the kernel or from the boot loader, identifying I2C devices
and linking them to board-specific configuration information about IRQs
and other wiring artifacts, chip type, and so on. That could be used to
create i2c_client objects for each I2C device.

I2C device drivers using this binding model work just like any other
kind of driver in Linux: they provide a probe() method to bind to
those devices, and a remove() method to unbind.

```
 static int foo_probe(struct i2c_client *client,
 	const struct i2c_device_id *id);
 static int foo_remove(struct i2c_client *client);
```

Remember that the i2c_driver does not create those client handles. The
handle may be used during foo_probe(). If foo_probe() reports success
(zero not a negative status code) it may save the handle and use it until
foo_remove() returns. That binding model is used by most Linux drivers.

The probe function is called when an entry in the id_table name field
matches the device's name. It is passed the entry that was matched so
the driver knows which one in the table matched.

Device Creation

If you know for a fact that an I2C device is connected to a given I2C bus,
you can instantiate that device by simply filling an i2c_board_info
structure with the device address and driver name, and calling
i2c_new_device(). This will create the device, then the driver core will
take care of finding the right driver and will call its probe() method.
If a driver supports different device types, you can specify the type you
want using the type field. You can also specify an IRQ and platform data
if needed.

Sometimes you know that a device is connected to a given I2C bus, but you
don't know the exact address it uses. This happens on TV adapters for
example, where the same driver supports dozens of slightly different
models, and I2C device addresses change from one model to the next. In
that case, you can use the i2c_new_probed_device() variant, which is
similar to i2c_new_device(), except that it takes an additional list of
possible I2C addresses to probe. A device is created for the first
responsive address in the list. If you expect more than one device to be
present in the address range, simply call i2c_new_probed_device() that

many times.

The call to i2c_new_device() or i2c_new_probed_device() typically happens
in the I2C bus driver. You may want to save the returned i2c_client
reference for later use.

Device Detection

Sometimes you do not know in advance which I2C devices are connected to
a given I2C bus. This is for example the case of hardware monitoring
devices on a PC's SMBus. In that case, you may want to let your driver
detect supported devices automatically. This is how the legacy model
was working, and is now available as an extension to the standard
driver model.

You simply have to define a detect callback which will attempt to
identify supported devices (returning 0 for supported ones and -ENODEV
for unsupported ones), a list of addresses to probe, and a device type
(or class) so that only I2C buses which may have that type of device
connected (and not otherwise enumerated) will be probed. For example,
a driver for a hardware monitoring chip for which auto-detection is
needed would set its class to I2C_CLASS_HWMON, and only I2C adapters
with a class including I2C_CLASS_HWMON would be probed by this driver.
Note that the absence of matching classes does not prevent the use of
a device of that type on the given I2C adapter. All it prevents is
auto-detection; explicit instantiation of devices is still possible.

Note that this mechanism is purely optional and not suitable for all
devices. You need some reliable way to identify the supported devices
(typically using device-specific, dedicated identification registers),
otherwise misdetections are likely to occur and things can get wrong
quickly. Keep in mind that the I2C protocol doesn't include any
standard way to detect the presence of a chip at a given address, let
alone a standard way to identify devices. Even worse is the lack of
semantics associated to bus transfers, which means that the same
transfer can be seen as a read operation by a chip and as a write
operation by another chip. For these reasons, explicit device
instantiation should always be preferred to auto-detection where
possible.

Device Deletion

Each I2C device which has been created using i2c_new_device() or
i2c_new_probed_device() can be unregistered by calling

i2c_unregister_device(). If you don't call it explicitly, it will be
called automatically before the underlying I2C bus itself is removed, as a
device can't survive its parent in the device driver model.

Initializing the driver
=======================

When the kernel is booted, or when your foo driver module is inserted,
you have to do some initializing. Fortunately, just registering the
driver module is usually enough.

```
static int __init foo_init(void)
{
 return i2c_add_driver(&foo_driver);
}
module_init(foo_init);

static void __exit foo_cleanup(void)
{
 i2c_del_driver(&foo_driver);
}
module_exit(foo_cleanup);
```

The module_i2c_driver() macro can be used to reduce above code.

```
module_i2c_driver(foo_driver);
```

Note that some functions are marked by `__init'. These functions can
be removed after kernel booting (or module loading) is completed.
Likewise, functions marked by `__exit' are dropped by the compiler when
the code is built into the kernel, as they would never be called.

Driver Information
==================

```
/* Substitute your own name and email address */
MODULE_AUTHOR("Frodo Looijaard "
MODULE_DESCRIPTION("Driver for Barf Inc. Foo I2C devices");

/* a few non-GPL license types are also allowed */
MODULE_LICENSE("GPL");
```

Power Management
================

If your I2C device needs special handling when entering a system low
power state -- like putting a transceiver into a low power mode, or
activating a system wakeup mechanism -- do that in the suspend() method.
The resume() method should reverse what the suspend() method does.

These are standard driver model calls, and they work just like they
would for any other driver stack. The calls can sleep, and can use
I2C messaging to the device being suspended or resumed (since their
parent I2C adapter is active when these calls are issued, and IRQs
are still enabled).

System Shutdown
===============

If your I2C device needs special handling when the system shuts down
or reboots (including kexec) -- like turning something off -- use a
shutdown() method.

Again, this is a standard driver model call, working just like it
would for any other driver stack: the calls can sleep, and can use
I2C messaging.

Command function
================

A generic ioctl-like function call back is supported. You will seldom
need this, and its use is deprecated anyway, so newer design should not
use it.

Sending and receiving
=====================

If you want to communicate with your device, there are several functions
to do this. You can find all of them in .

If you can choose between plain I2C communication and SMBus level
communication, please use the latter. All adapters understand SMBus level
commands, but only some of them understand plain I2C!

Plain I2C communication

```
 int i2c_master_send(struct i2c_client *client, const char *buf,
       int count);
 int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

These routines read and write some bytes from/to a client. The client contains the i2c address, so you do not have to include it. The second parameter contains the bytes to read/write, the third the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.) Returned is the actual number of bytes read/written.

```
 int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg,
    int num);
```

This sends a series of messages. Each message can be a read or write, and they can be mixed in any way. The transactions are combined: no stop bit is sent between transaction. The i2c_msg structure contains for each message the client address, the number of bytes of the message and the message data itself.

You can read the file `i2c-protocol' for more information about the actual I2C protocol.

SMBus communication

```
 s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
      unsigned short flags, char read_write, u8 command,
      int size, union i2c_smbus_data *data);
```

This is the generic SMBus function. All functions below are implemented in terms of it. Never use this function directly!

```
 s32 i2c_smbus_read_byte(struct i2c_client *client);
 s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
 s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
 s32 i2c_smbus_write_byte_data(struct i2c_client *client,
         u8 command, u8 value);
 s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
 s32 i2c_smbus_write_word_data(struct i2c_client *client,
         u8 command, u16 value);
 s32 i2c_smbus_read_block_data(struct i2c_client *client,
         u8 command, u8 *values);
 s32 i2c_smbus_write_block_data(struct i2c_client *client,
          u8 command, u8 length, const u8 *values);
 s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
       u8 command, u8 length, u8 *values);
 s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,
       u8 command, u8 length,
        const u8 *values);
```

These ones were removed from i2c-core because they had no users, but could be added back later if needed:

```
 s32 i2c_smbus_write_quick(struct i2c_client *client, u8 value);
 s32 i2c_smbus_process_call(struct i2c_client *client,
       u8 command, u16 value);
 s32 i2c_smbus_block_process_call(struct i2c_client *client,
      u8 command, u8 length, u8 *values);
```

All these transactions return a negative errno value on failure. The 'write' transactions return 0 on success; the 'read' transactions return the read value, except for block transactions, which return the number of values read. The block buffers need not be longer than 32 bytes.

You can read the file `smbus-protocol' for more information about the actual SMBus protocol.

General purpose routines

Below all general purpose routines are listed, that were not mentioned before.

```
 /* Return the adapter number for a specific adapter */
 int i2c_adapter_id(struct i2c_adapter *adap);
```