**Introduction to linux device driver**

Of 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.

Further, a “pilot” could be a person or even an automatic system monitored by a person (an auto-pilot system in airliners, for example). Similarly, 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.

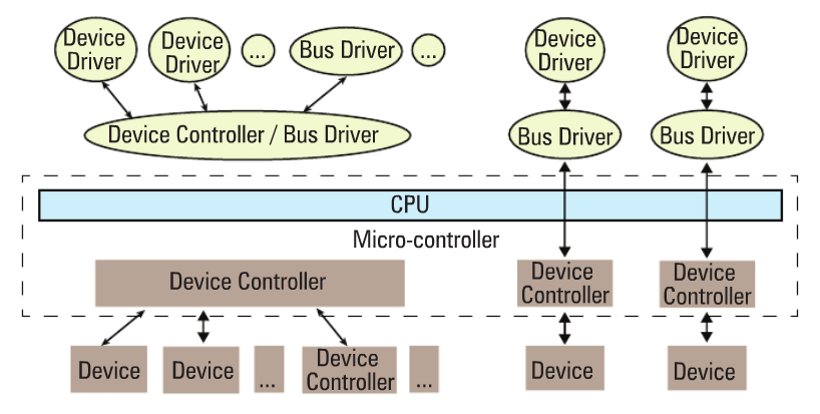


Figure 1: Device and driver interaction

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.

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.

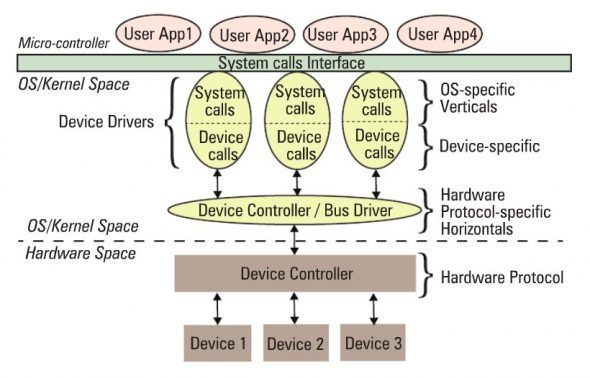
[](http://www.opensourceforu.com/wp-content/uploads/2010/11/ldd2.jpg)

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.

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.

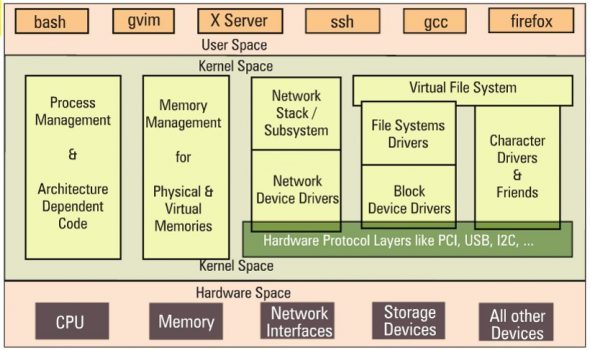
[](http://www.opensourceforu.com/wp-content/uploads/2010/11/ldd3.jpg)

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.

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.

Summing up

So, to conclude, a device driver is a piece of software that drives a device, though there are so many classifications. In case it drives only another piece of software, we call it just a driver. Examples are file-system drivers, usbcore, etc. Hence, all device drivers are drivers, but all drivers are not device drivers.

“Hey, Pugs, hold on; we’re getting late for class, and you know what kind of trouble we can get into. Let’s continue from here, later,” exclaimed Shweta.

Jumping up, Pugs finished his explanation: “Okay. This is the basic theory about device drivers. If you’re interested, later, I can show you the code, and all that we have been doing for the various kinds of drivers.” And they hurried towards their classroom.

# Writing Your First Linux Driver in the Classroom

This article, which is part of the [series on Linux device drivers](http://www.opensourceforu.com/tag/linux-device-drivers-series/), deals with the concept of dynamically loading drivers, first writing a Linux driver, before building and then loading it.

Shweta and Pugs reached their classroom late, to find their professor already in the middle of a lecture. Shweta sheepishly asked for his permission to enter. An annoyed Professor Gopi responded, “Come on! You guys are late again; what is your excuse, today?”

Pugs hurriedly replied that they had been discussing the very topic for that day’s class — device drivers in Linux. Pugs was more than happy when the professor said, “Good! Then explain about dynamic loading in Linux. If you get it right, the two of you are excused!” Pugs knew that one way to make his professor happy was to criticise Windows.

He explained, “As we know, a typical driver installation on Windows needs a reboot for it to get activated. That is really not acceptable; suppose we need to do it on a server? That’s where Linux wins. In Linux, we can load or unload a driver on the fly, and it is active for use instantly after loading. Also, it is instantly disabled when unloaded. This is called dynamic loading and unloading of drivers in Linux.”

This impressed the professor. “Okay! Take your seats, but make sure you are not late again.” The professor continued to the class, “Now you already know what is meant by dynamic loading and unloading of drivers, so I’ll show you how to do it, before we move on to write our first Linux driver.”

## 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/<kernel\_version>/kernel, where <kernel\_version> would be the output of the command uname -ron the system, as shown in Figure 1.

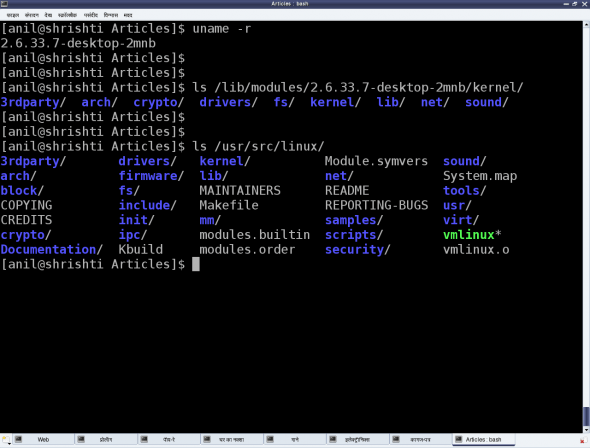
[](http://www.opensourceforu.com/wp-content/uploads/2010/12/figure_4_linux_modules.png)

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 <module\_file> — inserts/loads the specified module file
* modprobe <module> — inserts/loads the module, along with any dependencies
* rmmod <module> — 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.

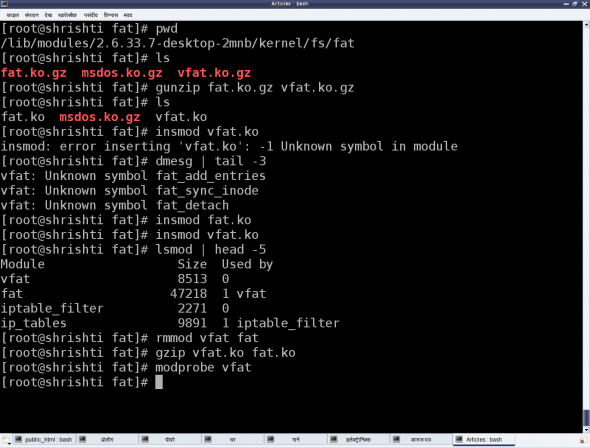
[](http://www.opensourceforu.com/wp-content/uploads/2010/12/figure_5_linux_module_operations.png)

Figure 2: Linux module operations

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.

## 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 exitfunctions, respectively, by the macros module\_init() and module\_exit(), which are defined in the kernel header module.h.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | /\* ofd.c – Our First Driver code \*/  #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>    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 <email\_at\_sarika-pugs\_dot\_com>");  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”.

## 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # 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' |

## Summing up

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.

While the students were trying their first module, the bell rang, marking the end of the session. Professor Gopi concluded, “Currently, you may not be able to observe anything other than the lsmod listing showing the driver has loaded. Where’s the printk output gone? Find that out for yourselves, in the lab session, and update me with your findings. Also note that our first driver is a template for any driver you would write in Linux. Writing a specialised driver is just a matter of what gets filled into its constructor and destructor. So, our further learning will be to enhance this driver to achieve specific driver functionalities.”

# Kernel C Extras in a Linux Driver

This article in the [series on Linux device drivers](http://www.opensourceforu.com/tag/linux-device-drivers-series/) deals with the kernel’s message logging, and kernel-specific GCC extensions.

Enthused by how Pugs impressed their professor in the last class, Shweta wanted to do so too. And there was soon an opportunity: finding out where the output of printk had gone. So, as soon as she entered the lab, she grabbed the best system, logged in, and began work. Knowing her professor well, she realised that he would have dropped a hint about the possible solution in the previous class itself. Going over what had been taught, she remembered the error output demonstration from insmod vfat.ko — running dmesg | tail. She immediately tried that, and found the printk output there.

But how did it come to be here? A tap on her shoulder roused her from her thoughts. “Shall we go for a coffee?” proposed Pugs.

“But I need to –“.

“I know what you’re thinking about,” interrupted Pugs. “Let’s go, I’ll explain you all about dmesg.”

## Kernel message logging

Over coffee, Pugs began his explanation.

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, in the [last article](http://www.opensourceforu.com/2010/12/writing-your-first-linux-driver/). 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.

Figure 1: Kernel’s message logging

## Kernel-specific GCC extensions

Shweta, frustrated since she could no longer show off as having discovered all these on her own, retorted, “Since you have explained all about printing in the kernel, why don’t you also tell me about the weird C in the driver as well — the special keywords \_\_init, \_\_exit, etc.”

These are not special keywords. 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.

## The kernel function’s return guidelines

While returning from coffee, Pugs kept praising OSS and the community that’s grown around it. Do you know why different individuals are able to come together and contribute excellently without any conflicts, and in a project as huge as Linux, at that? There are many reasons, but most important amongst them is that they all follow and abide by inherent coding guidelines.

Take, for example, the kernel programming guideline for returning values from a function. 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.

## Kernel C = pure C

Once back in the lab, Shweta remembered their professor mentioning that no /usr/include headers can be used for kernel programming. But Pugs had said that kernel C is just standard C with some GCC extensions. Why this conflict?

Actually this is not a conflict. 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.

“Oh yes! That is why we need to have the kernel source to build a driver,” agreed Shweta.

“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,” added Pugs.

“In the lab, all the sources are set up. But if I want to try out drivers on my Linux system in my hostel room, how do I go about it?” asked Shweta.

“Our lab has Fedora, where the kernel sources are typically installed under /usr/src/kernels/<kernel-version>, unlike the standard /usr/src/linux. Lab administrators must have installed it using the command-line yum install kernel-devel. I use Mandriva, and installed the kernel sources using urpmi kernel-source,” replied Pugs.

“But I have Ubuntu,” Shweta said.

“Okay! For that, just use apt-get utility to fetch the source — possibly apt-get install linux-source,” replied Pugs.

## Summing up

The lab session was almost over when Shweta suddenly asked, out of curiosity, “Hey Pugs, what’s the next topic we are going to learn in our Linux device drivers class?”

“Hmm… most probably character drivers,” threw back Pugs.

With this information, Shweta hurriedly packed her bag and headed towards her room to set up the kernel sources, and try out the next driver on her own. “In case you get stuck, just give me a call,” smiled Pugs

# Module Interactions

As Shweta and Pugs gear up for their final semester’s project on Linux drivers, they’re closing in on some final titbits of technical romancing. This mainly includes the various communications with a Linux module (dynamically loadable and unloadable driver) like accessing its variables, calling its functions, and passing parameters to it.

## Global variables and functions

One might wonder what the big deal is about accessing a module’s variables and functions from outside it. Just make them global, declare them extern in a header, include the header and access, right? In the general application development paradigm, it’s this simple — but in kernel development, it isn’t despite of recommendations to make everything static, by default there always have been cases where non-static globals may be needed.

A simple example could be a driver spanning multiple files, with function(s) from one file needing to be called in the other. Now, to avoid any kernel name-space collisions even with such cases, every module is embodied in its own namespace. And we know that two modules with the same name cannot be loaded at the same time. Thus, by default, zero collision is achieved. However, this also implies that, by default, nothing from a module can be made really global throughout the kernel, even if we want to. And so, for exactly such scenarios, the <linux/module.h>header defines the  
following macros:

* EXPORT\_SYMBOL(sym)
* EXPORT\_SYMBOL\_GPL(sym)
* EXPORT\_SYMBOL\_GPL\_FUTURE(sym)

Each of these exports the symbol passed as their parameter, additionally putting them in one of the default, \_gpl or \_gpl\_future sections, respectively. Hence, only one of them needs to be used for a particular symbol — though the symbol could be either a variable name or a function name. Here’s the complete code (our\_glob\_syms.c) to demonstrate this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include <linux/module.h>  #include <linux/device.h>    static struct class \*cool\_cl;  static struct class \*get\_cool\_cl(void)  {      return cool\_cl;  }  EXPORT\_SYMBOL(cool\_cl);  EXPORT\_SYMBOL\_GPL(get\_cool\_cl);    static int \_\_init glob\_sym\_init(void)  {      if (IS\_ERR(cool\_cl = class\_create(THIS\_MODULE, "cool")))      /\* Creates /sys/class/cool/ \*/      {          return PTR\_ERR(cool\_cl);      }      return 0;  }    static void \_\_exit glob\_sym\_exit(void)  {      /\* Removes /sys/class/cool/ \*/      class\_destroy(cool\_cl);  }    module\_init(glob\_sym\_init);  module\_exit(glob\_sym\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs.com>");  MODULE\_DESCRIPTION("Global Symbols exporting Driver"); |

Each exported symbol also has a corresponding structure placed into (each of) the kernel symbol table (\_\_ksymtab), kernel string table (\_\_kstrtab), and kernel CRC table (\_\_kcrctab) sections, marking it to be globally accessible.

Figure 1 shows a filtered snippet of the /proc/kallsyms kernel window, before and after loading the module our\_glob\_syms.ko, which has been compiled using the usual driver Makefile.

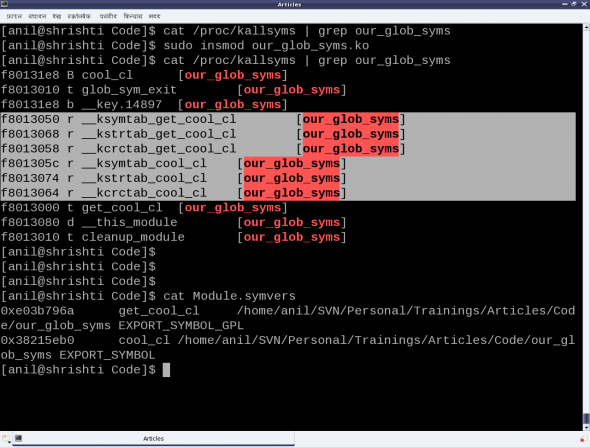
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_30_our_glob_syms.png)

Figure 1: Our global symbols module

The following code shows the supporting header file (our\_glob\_syms.h), to be included by modules using the exported symbols cool\_cl and get\_cool\_cl:

|  |
| --- |
| #ifndef OUR\_GLOB\_SYMS\_H  #define OUR\_GLOB\_SYMS\_H    #ifdef \_\_KERNEL\_\_  #include <linux/device.h>    extern struct class \*cool\_cl;  extern struct class \*get\_cool\_cl(void);  #endif    #endif |

Figure 1 also shows the file Module.symvers, generated by compiling the module our\_glob\_syms. This contains the various details of all the exported symbols in its directory. Apart from including the above header file, modules using the exported symbols should possibly have this file Module.symvers in their build directory.

Note that the <linux/device.h> header in the above examples is being included for the various class-related declarations and definitions, which have already been covered in the earlier discussion on character drivers.

## Module parameters

Being aware of passing command-line arguments to an application, it would be natural to ask if something similar can be done with a module — and the answer is, yes, it can. Parameters can be passed to a module while loading it, for instance, when using insmod. Interestingly enough, and in contrast to the command-line arguments to an application, these can be modified even later, through sysfs interactions.

The module parameters are set up using the following macro (defined in <linux/moduleparam.h>, included through <linux/module.h>):

|  |
| --- |
| module\_param(name, type, perm) |

Here, name is the parameter name, type is the type of the parameter, and perm refers to the permissions of the sysfs file corresponding to this parameter. The supported type values are: byte, short, ushort, int, uint, long, ulong, charp (character pointer), bool or invbool (inverted Boolean).

The following module code (module\_param.c) demonstrates a module parameter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <linux/module.h>  #include <linux/kernel.h>    static int cfg\_value = 3;  module\_param(cfg\_value, int, 0764);    static int \_\_init mod\_par\_init(void)  {      printk(KERN\_INFO "Loaded with %d\n", cfg\_value);      return 0;  }    static void \_\_exit mod\_par\_exit(void)  {      printk(KERN\_INFO "Unloaded cfg value: %d\n", cfg\_value);  }    module\_init(mod\_par\_init);  module\_exit(mod\_par\_exit);  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email@sarika-pugs.com>");  MODULE\_DESCRIPTION("Module Parameter demonstration Driver"); |

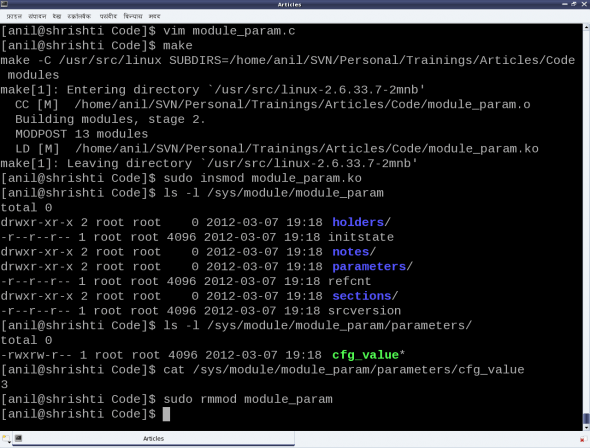
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_31_module_param.png)

Figure 2: Experiments with the module parameter

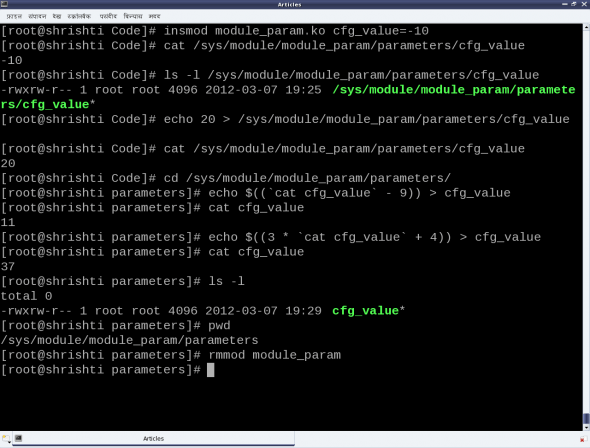
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_32_module_param_as_root.png)

Figure 3: Experiments with the module parameter (as root)

Note that before the parameter setup, a variable of the same name and compatible type needs to be defined. Subsequently, the following steps and experiments are shown in Figures 2 and 3:

* Building the driver (module\_param.ko file) using the usual driver Makefile
* Loading the driver using insmod (with and without parameters)
* Various experiments through the corresponding /sys entries
* And finally, unloading the driver using rmmod.

Note the following:

* Initial value (3) of cfg\_value becomes its default value when insmod is done without any parameters.
* Permission 0764 gives rwx to the user, rw- to the group, and r-- for the others on the file cfg\_valueunder the parameters of module\_param under /sys/module/.

Check for yourself:

* The output of dmesg/tail on every insmod and rmmod, for the printk outputs.
* Try writing into the /sys/module/module\_param/parameters/cfg\_value file as a normal (non-root) user.

## Summing up

With this, the duo have a fairly good understanding of Linux drivers, and are all set to start working on their final semester project. Any guesses what their project is about? Hint: They have picked up one of the most daunting Linux driver topics. Let us see how they fare with it next month.

# Kernel Window — Peeping through /proc

After many months, Shweta and Pugs got together for some peaceful technical romancing. All through, they had been using all kinds of kernel windows, especially through the /proc virtual filesystem (using cat), to help them decode various details of Linux device drivers. Here’s a non-exhaustive summary listing:

* /proc/modules — dynamically loaded modules
* /proc/devices — registered character and block major numbers
* /proc/iomem — on-system physical RAM and bus device addresses
* /proc/ioports — on-system I/O port addresses (especially for x86 systems)
* /proc/interrupts — registered interrupt request numbers
* /proc/softirqs — registered soft IRQs
* /proc/kallsyms — running kernel symbols, including from loaded modules
* /proc/partitions — currently connected block devices and their partitions
* /proc/filesystems — currently active filesystem drivers
* /proc/swaps — currently active swaps
* /proc/cpuinfo — information about the CPU(s) on the system
* /proc/meminfo — information about the memory on the system, viz., RAM, swap, …

## Custom kernel windows

“Yes, these have been really helpful in understanding and debugging Linux device drivers. But is it possible for us to also provide some help? Yes, I mean can we create one such kernel window through /proc?” asked Shweta.

“Why just one? You can have as many as you want. And it’s simple — just use the right set of APIs, and there you go.”

“For you, everything is simple,” Shweta grumbled.

“No yaar, this is seriously simple,” smiled Pugs. “Just watch me creating one for you,” he added.  
And in a jiffy, Pugs created the proc\_window.c file below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/proc\_fs.h>  #include <linux/jiffies.h>    static struct proc\_dir\_entry \*parent, \*file, \*link;  static int state = 0;    int time\_read(char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data) {      int len, val;      unsigned long act\_jiffies;        len = sprintf(page, "state = %d\n", state);      act\_jiffies = jiffies - INITIAL\_JIFFIES;      val = jiffies\_to\_msecs(act\_jiffies);      switch (state) {          case 0:              len += sprintf(page + len, "time = %ld jiffies\n", act\_jiffies);              break;          case 1:              len += sprintf(page + len, "time = %d msecs\n", val);              break;          case 2:              len += sprintf(page + len, "time = %ds %dms\n",                      val / 1000, val % 1000);              break;          case 3:              val /= 1000;              len += sprintf(page + len, "time = %02d:%02d:%02d\n",                      val / 3600, (val / 60) % 60, val % 60);              break;          default:              len += sprintf(page + len, "<not implemented>\n");              break;      }      len += sprintf(page + len, "{offset = %ld; count = %d;}\n", off, count);        return len;  }  int time\_write(struct file \*file, const char \_\_user \*buffer, unsigned long count, void \*data) {      if (count > 2)          return count;      if ((count == 2) && (buffer[1] != '\n'))          return count;      if ((buffer[0] < '0') || ('9' < buffer[0]))          return count;      state = buffer[0] - '0';      return count;  }    static int \_\_init proc\_win\_init(void) {      if ((parent = proc\_mkdir("anil", NULL)) == NULL) {          return -1;      }      if ((file = create\_proc\_entry("rel\_time", 0666, parent)) == NULL) {          remove\_proc\_entry("anil", NULL);          return -1;      }      file->read\_proc = time\_read;      file->write\_proc = time\_write;      if ((link = proc\_symlink("rel\_time\_l", parent, "rel\_time")) == NULL) {          remove\_proc\_entry("rel\_time", parent);          remove\_proc\_entry("anil", NULL);          return -1;      }      link->uid = 0;      link->gid = 100;      return 0;  }    static void \_\_exit proc\_win\_exit(void) {      remove\_proc\_entry("rel\_time\_l", parent);      remove\_proc\_entry("rel\_time", parent);      remove\_proc\_entry("anil", NULL);  }    module\_init(proc\_win\_init);  module\_exit(proc\_win\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Kernel window /proc Demonstration Driver"); |

And then Pugs did the following:

* Built the driver file (proc\_window.ko) using the usual driver’s Makefile.
* Loaded the driver using insmod.
* Showed various experiments using the newly created proc windows. (Refer to Figure 1.)
* And finally, unloaded the driver using rmmod.

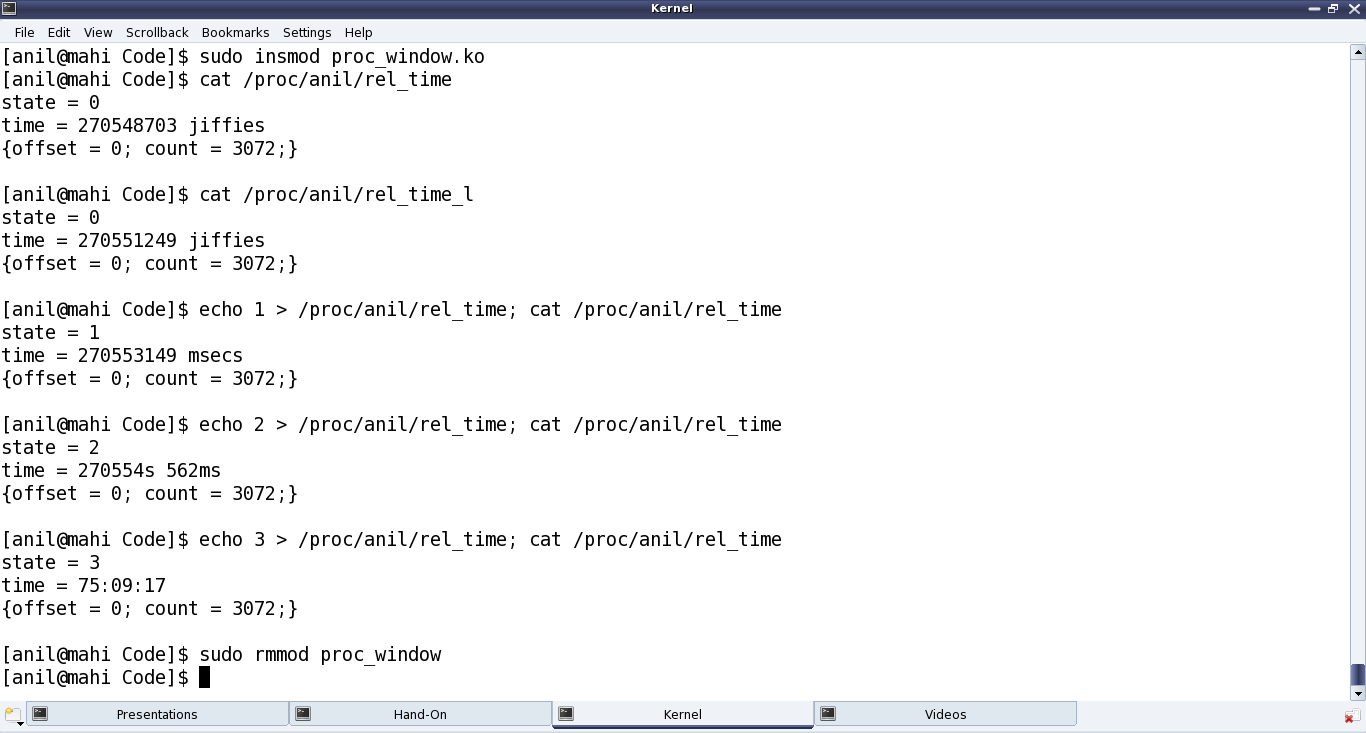


Figure 1: Peeping through /proc

## Demystifying the details

Starting from the constructor proc\_win\_init(), three proc entries have been created:

* Directory anil under /proc (i.e., NULL parent) with default permissions 0755, using proc\_mkdir()
* Regular file rel\_time in the above directory, with permissions 0666, using create\_proc\_entry()
* Soft link rel\_time\_l to the file rel\_time, in the same directory, using proc\_symlink()

The corresponding removal of these is done with remove\_proc\_entry() in the destructor, proc\_win\_exit(), in chronological reverse order.

For every entry created under /proc, a corresponding struct proc\_dir\_entry is created. For each, many of its fields could be further updated as needed:

* mode — Permissions of the file
* uid — User ID of the file
* gid — Group ID of the file

Additionally, for a regular file, the following two function pointers for reading and writing over the file could be provided, respectively:

* int (\*read\_proc)(char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data)
* int (\*write\_proc)(struct file \*file, const char \_\_user \*buffer, unsigned long count, void \*data)

write\_proc() is very similar to the character driver’s file operation write(). The above implementation lets the user write a digit from 0 to 9, and accordingly sets the internal state. read\_proc() in the above implementation provides the current state, and the time since the system has been booted up — in different units, based on the current state. These are jiffies in state 0; milliseconds in state 1; seconds and milliseconds in state 2; hours, minutes and seconds in state 3; and <not implemented> in other states.

And to check the computation accuracy, Figure 2 highlights the system uptime in the output of top. read\_proc‘s page parameter is a page-sized buffer, typically to be filled up with count bytes from offset off. But more often than not (because of less content), just the page is filled up, ignoring all other parameters.

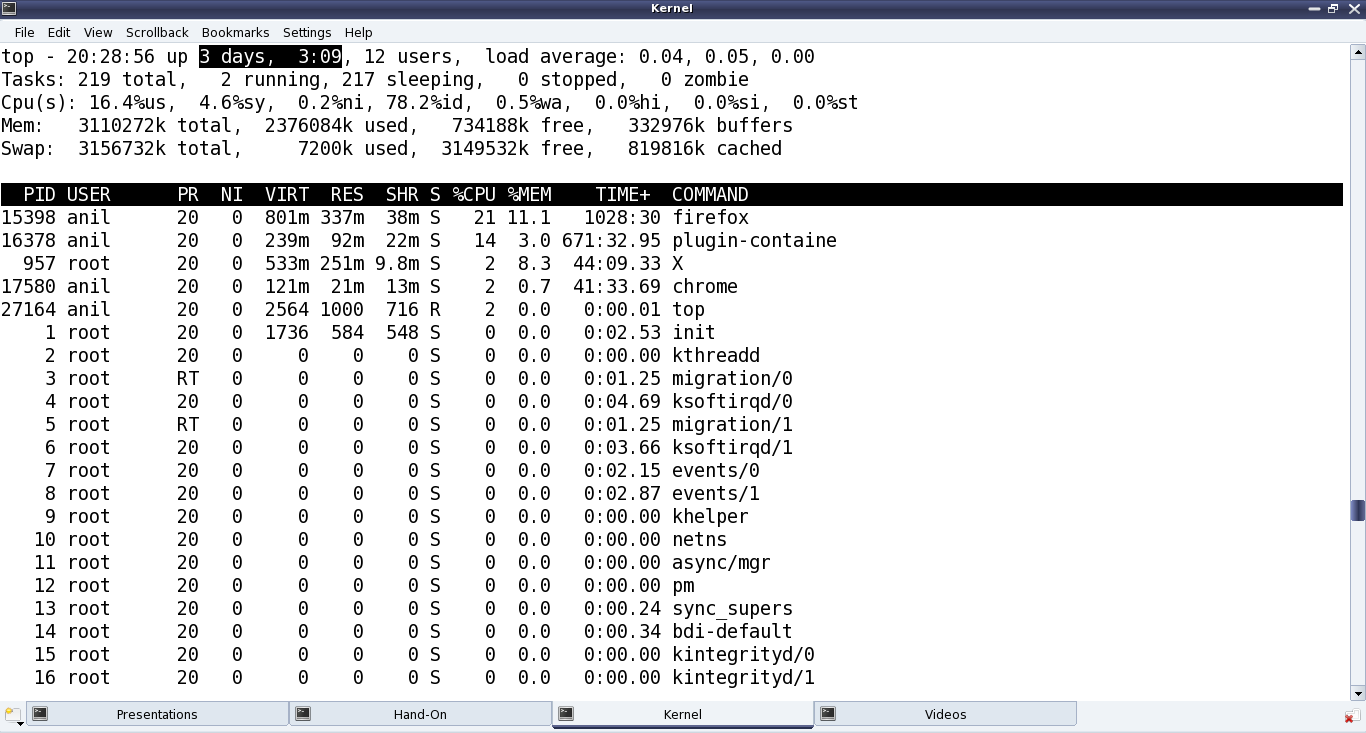


Figure 2: Comparison with top’s output

All the /proc-related structure definitions and function declarations are available through <linux/proc\_fs.h>. The jiffies-related function declarations and macro definitions are in <linux/jiffies.h>. As a special note, the actual jiffies are calculated by subtracting INITIAL\_JIFFIES, since on boot-up, jiffies is initialised to INITIAL\_JIFFIES instead of zero.

## Summing up

“Hey Pugs! Why did you set the folder name to anil? Who is this Anil? You could have used my name, or maybe yours,” suggested Shweta. “Ha! That’s a surprise. My real name is Anil; it’s just that everyone in college knows me as Pugs,” smiled Pugs.

# Kernel-Space Debuggers in Linux

Shweta, back from hospital, was relaxing in the library, reading various books. Ever since she learned of the ioctlway of debugging, she was impatient to find out more about debugging in kernel-space. She was curious about how and where to run the kernel-space debugger, if there was any. This was in contrast with application/user-space debugging, where we have the OS running underneath, and a shell or a GUI over it to run the debugger (like gdb, and the data display debugger, ddd). Then she came across this interesting kernel-space debugging mechanism using kgdb, provided as part of the kernel itself, since kernel 2.6.26.

## The debugger challenge in kernel-space

As we need some interface to be up to run a debugger to debug anything, a kernel debugger could be visualised in two possible ways:

* Put the debugger into the kernel itself, accessible via the usual console. For example, in the case of kdb, which was not official until kernel 2.6.35, one had to download source code (two sets of patches — one architecture-dependent, one architecture-independent) from [this FTP address](ftp://oss.sgi.com/projects/kdb/download/) and then patch these into the kernel source. However, since kernel 2.6.35, the majority of it is in the officially released kernel source. In either case, kdb support needs to be enabled in kernel source, with the kernel compiled, installed and booted with. The boot screen itself would give the kdb debugging interface.
* Put a minimal debugging server into the kernel; a client would connect to it from a remote host or local user-space over some interface (say serial or Ethernet). This is kgdb, the kernel’s gdb server, to be used with gdbas its client. Since kernel 2.6.26, its serial interface is part of the official kernel release. However, if you’re interested in a network interface, you still need to patch with one of the releases from the [kgdb project page](http://sourceforge.net/projects/kgdb/). In either case, you need to enable kgdb support in the kernel, recompile, install and boot the new kernel.

Please note that in both the above cases, the complete kernel source for the kernel to be debugged is needed, unlike for building modules, where just headers are sufficient. Here is how to play around with kgdb over the serial interface.

## Setting up the Linux kernel with kgdb

Here are the prerequisites: Either the kernel source package for the running kernel should be installed on your system, or a corresponding kernel source release should have been downloaded from [kernel.org](http://kernel.org/).

First of all, the kernel to be debugged needs to have kgdb enabled and built into it. To achieve that, the kernel source has to be configured with CONFIG\_KGDB=y. Additionally, for kgdb over serial, CONFIG\_KGDB\_SERIAL\_CONSOLE=y needs to be configured. And CONFIG\_DEBUG\_INFO is preferred for symbolic data to be built into the kernel, to make debugging with gdb more meaningful. CONFIG\_FRAME\_POINTER=y enables frame pointers in the kernel, allowing gdb to construct more accurate stack back-traces. All these options are available under “Kernel hacking” in the menu obtained in the kernel source directory (preferably as root, or using sudo), by issuing the following command:

|  |
| --- |
| $ make mrproper      # To clean up properly  $ make oldconfig     # Configure the kernel same as the current running one  $ make menuconfig    # Start the ncurses based menu for further configuration |

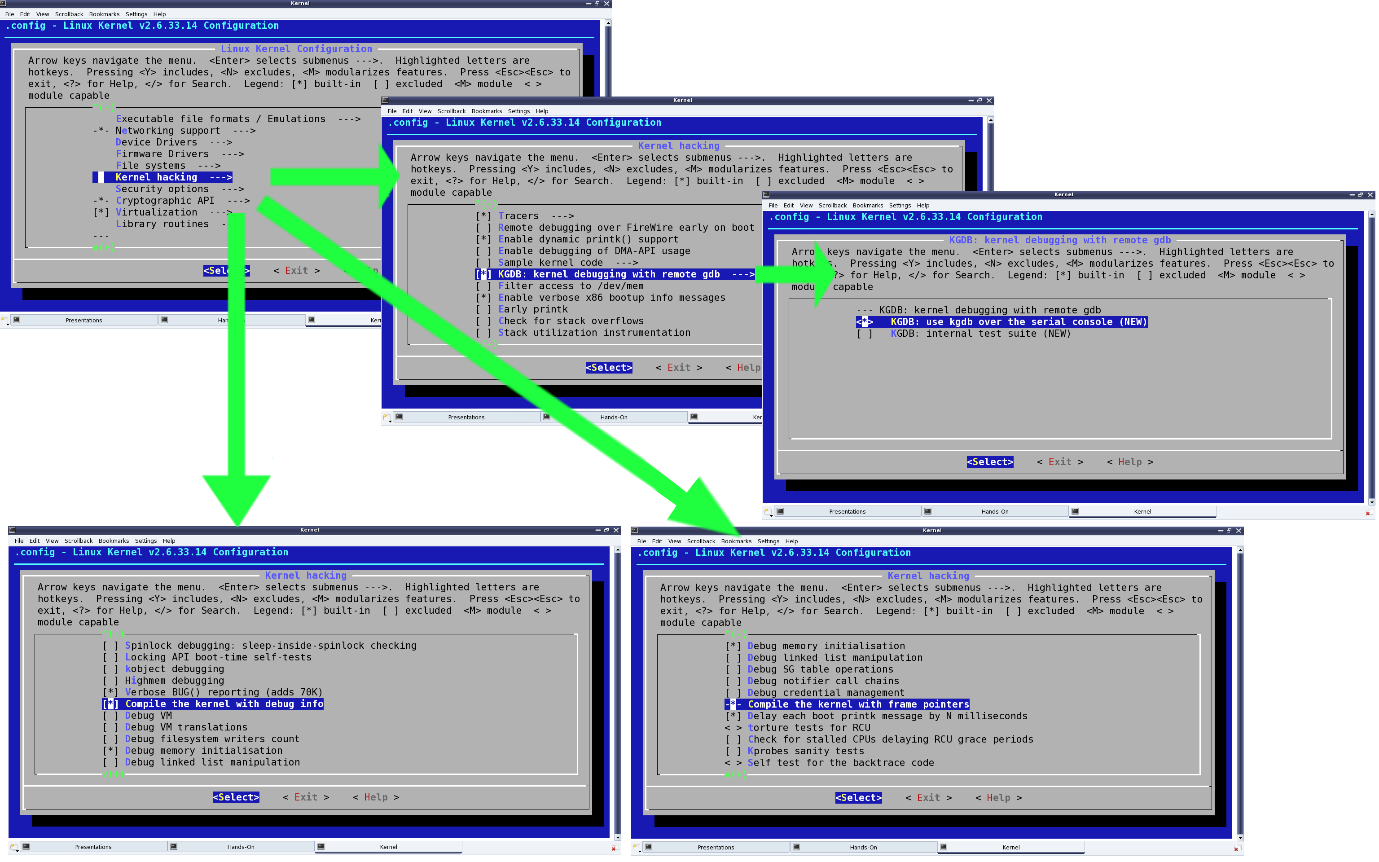


Figure 1: Configuring kernel options for kgdb

See the highlighted selections in Figure 1, for how and where these options would be:

* “KGDB: kernel debugging with remote gdb” –> CONFIG\_KGDB
* “KGDB: use kgdb over the serial console” –> CONFIG\_KGDB\_SERIAL\_CONSOLE
* “Compile the kernel with debug info” –> CONFIG\_DEBUG\_INFO
* “Compile the kernel with frame pointers” –> CONFIG\_FRAME\_POINTER

Once configuration is saved, build the kernel (run make), and then a make install to install it, along with adding an entry for the installed kernel in the GRUB configuration file. Depending on the distribution, the GRUB configuration file may be /boot/grub/menu.lst, /etc/grub.cfg, or something similar. Once installed, the kgdb-related kernel boot parameters need to be added to this new entry, as shown in the highlighted text in Figure 2.



Figure 2: GRUB configuration for kgdb

kgdboc is for gdb connecting over the console, and the basic format is kgdboc= <serial\_device>, <baud-rate>where:

* <serial\_device> is the serial device file (port) on the system running the kernel to be debugged
* <baud-rate> is the baud rate of this serial port

kgdbwait tells the kernel to delay booting till a gdb client connects to it; this parameter should be given only after kgdboc.

With this, we’re ready to begin. Make a copy of the vmlinux kernel image for use on the gdb client system. Reboot, and at the GRUB menu, choose the new kernel, and then it will wait for gdb to connect over the serial port.

All the above snapshots are with kernel version 2.6.33.14. The same should work for any 2.6.3x release of the kernel source. Also, the snapshots for kgdb are captured over the serial device file /dev/ttyS0, i.e., the first serial port.

## Setting up gdb on another system

Following are the prerequisites:

* Serial ports of the system to be debugged, and the other system to run gdb, should be connected using a null modem (i.e., a cross-over serial) cable.
* The vmlinux kernel image built, with kgdb enabled, needs to be copied from the system to be debugged, into the working directory on the system where gdb is going to be run.

To get gdb to connect to the waiting kernel, launch gdb from the shell and run these commands:

|  |
| --- |
| (gdb) file vmlinux  (gdb) set remote interrupt-sequence Ctrl-C  (gdb) set remotebaud 115200  (gdb) target remote /dev/ttyS0  (gdb) continue |

In the above commands, vmlinux is the kernel image copied from the system to be debugged.

## Debugging using gdb with kgdb

After this, it is all like debugging an application from gdb. One may stop execution using Ctrl+C, add break points using b[reak], stop execution using s[tep] or n[ext] … — the usual gdb way. There are enough GDB tutorials available online, if you need them. In fact, if you are not comfortable with text-based GDB, use any of the standard GUI tools over gdb, like ddd, Eclipse, etc.

## Summing up

By now, Shweta was excited about wanting to try out kgdb. Since she needed two systems to try it out, she went to the Linux device drivers’ lab. There, she set up the systems and ran gdb as described above.