Tech Newsletter

Golden Bits® Software, Inc.

Golden Bits Software, Inc.
3525 Del Mar Heights Rd, Suite 158
San Diego, CA 92130
858.259.3870 phone
858.259.7655 fax

Volume 1, Issue 3, Fall 2005

In this issue:

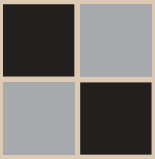
Golden Bits® is a software engineering firm providing consulting services in a wide range of diverse technologies:

- Windows, Linux
- Device Drivers, Embedded Systems
- Database, TCP/IP, GUIs

See page 12 for past projects www.goldenbits.com.

Technologies

- **Fibre channel**
- **Device drivers**
- **COM/DCOM**
- **Database, SQL**
- **C/C++**
- **SCSI**
- **GUIs/MFC/.NET**
- **Windows**
- **Linux**



KDB - Linux Kernel Debugging

How to use KDB in your next driver

Download sample code: www.goldenbits.com/newsletter/issue3/kdb_sample.zip

This article was featured in the Nov 2005 issue of Dr. Dobbs.

Introduction

Developing Linux kernel level software can be a difficult task. You code has to work well within the kernel (can't hog resources or hold spin locks for long periods of time), be multi-threaded, and must run smoothly on several Linux distributions.

Depending on the product and driver, approximately 40-50% of your development time will be spent testing your driver on different Linux distributions, with different system loads, and under a range of error conditions such as a hardware failure or out of memory cases. No matter how the hardware fails or if the OS is out of resources, your driver should never cause a panic. The bottom line is: ***test test test*** and ***debug debug debug***. It is not glamorous, but this is how solid products are created.

Unfortunately there isn't a lot of kernel level debugging support in Linux. The only available stock tool (one that is always available) is the good old debug printk() function. Remember debugging your first program, "Hello World", with printf()? Ugh!!! Print statements alone are completely inadequate when debugging any kernel code of moderate complexity. Fortunately for kernel developers, there is the Kernel Debugger or KDB. KDB is not part of the official kernel release from kernel.org, but it has been around for a while and has become the de facto kernel debugger of choice.

continued on page 2

Linux USB Driver

Download sample code: www.goldenbits.com/newsletter/issue3/linuxusb_sample.zip

Introduction:

USB has become a very popular technology used to connect all sorts of devices to host systems. Almost every computer system sold today, from low cost PCs to high-end server systems, comes with USB built into the system. What does this mean from a software viewpoint? If you're a software developer, it means you'll have to understand how to write a device driver for your whiz bang USB device.

This article presents a primer on how to develop a USB device driver for the Linux kernel, version 2.6.8. It includes a description of how USB is implemented in the Linux kernel, some of the key design aspects, and a sample USB driver.

The sample USB device presented is an encryptor/decyprtor device where the actual en(de)cryptor engine runs on the USB device itself. You can download the sample code from

continued on page 8

KDB (continued)

This article explains how to debug your driver with KDB, add hooks into KDB from your driver, and create a KDB module. It also includes several KDB tips and a sample driver (with source code) you can download from www.goldenbits.com/newsletter/issue3/kdb_sample.zip.

KDB Overview

KDB is an assembly level debugger; yes, you should be familiar with assembly to write and debug kernel code. It's very handy if you can decode a stack trace and the assembly instructions that worked on the stack. There is also a good source-level kernel debugger called kgdb that supports the 2.4 and 2.6 kernels (see http://sourceforge.net/projects/kgdb) and is worth investigating. If possible you might want to setup two development systems or two different bootable kernels, one with using KDB and the other with kgdb.

KDB provides a command line interface from which you can run various debug commands to dump memory, view registers, look at stack traces, and more. Typing the "help"command or "?" at the kdb prompt will display all of the commands. The nice thing about kdb is that it's "lightweight", meaning it does not interfere or change the normal flow and timing of the kernel code during runtime. Unlike a source-level debugger, you do not have to compile your driver with debug information. Sometimes this extra debug information does change the timing of your driver, and this subtle difference can mask a bug.

KDB is invoked when a breakpoint is hit, system panic, or by the break key. The main entry point for KDB is kdb() in kdb/kdbmain.c; when invoked, kdb disables interrupts for the local processor, and will stops the other processors and disables their interrupts. The key point is that all interrupts are disabled (with the exception of non-maskable interrupts) when KDB is invoked. If your hardware generates an interrupt request while you are in KDB, the request is ignored until you leave KDB.

Setup, getting what you need

You'll often hear that KDB is the Linux Kernel's "built-in" debugger. Now the words "built-in" in this case doesn't always mean what you would usually think. In fact, when you download the kernel source from www.kernel.org, look hard. You won't find any KDB source, so you'll have to apply a KDB patch your kernel.

Kernel patches for KDB are available from ftp://oss.sgi.com/projects/kdb/download, the latest being version 4.4; but beware of these patches, since they are usually written for a generic kernel from kernel.org and not for a specific distribution. As a result, sometimes these patches do not work for a specific distribution such as Red Hat, Mandrake, and it's usually the distribution you need to test against!! You might have to tweak the patch itself or hopefully the distribution contains a version of KDB that compiles and runs. For this article, I used the Mandrake 10.0 distribution which does come with KDB already bundled with the kernel source.

After the KDB patch has been applied, you need to turn on debugging via the kernel configuration menu and rebuild the kernel. The configuration menu selection is under "Kernel Hacking / Built-in Kernel Debugger support". Two other menu selections of note are "KDB Modules" and "KDB off by default". KDB modules enable you to extend the functionality of KDB itself, a very powerful feature and something I'll talk about later in detail. "KDB off by default" controls whether KDB is enabled at boot time or manually enabled by the user at a later time (off by default). If KDB is disabled, and the kernel panics, KDB will not be invoked. To enable KDB after boot use the command "**echo "1" > /proc/sys/kernel/kdb**" to enable, to disable use "**echo "0" > /proc/sys/kernel/kdb**".

You can run KDB from the console or via a serial cable. Using a serial connection is preferable because if the kernel panics

continued on page 3

A mathematician, a physicist, and an engineer were all given a red rubber ball and told to find the volume.
The mathematician carefully measured the diameter and evaluated a triple integral.
The physicist filled a beaker with water, put the ball in the water, and measured the total displacement.
The engineer looked up the model and serial numbers in his red-rubber-ball table.

KDB (continued)

while the console is running X-Windows (rendering your keyboard useless) you can still use KDB via the serial cable. Note also that terminal programs, such as Minicom, provide a history of commands and KDB output to which you can refer while debugging. To set up a serial connection you will need to configure the kernel to boot using the serial port as a console and a NULL modem cable; the details are well documented in linux/Documentation/serial-console.txt. Even if you are not using KDB, it's very handy to have a serial console to save any kernel messages while debugging. For example, if an overnight test is running and in the morning the system is hung, hopefully there will be some useful debug information on the serial console.

You can also configure KDB to your personal preferences by using the "kdb_cmds" text file. This file can contain environment variables such as number of lines to display and KDB commands such as breakpoints. However since this file is it is compiled into KDB itself, you must recompile KDB whenever you make any changes.

Common Commands

When you drop into KDB, the KDB prompt displays on the console terminal (or minicom if connected via serial line). From the KDB command prompt, you can start entering debugging commands, type the "?" for a help listing of commands. See **Figure 1** for a sample output of the KDB help command. Some useful debugging commands are:

id – Disassembles instructions at a specific address. You can use the instruction pointer address to find the name of a function.

Here's where you need to understand some assembly code. When the kernel panics, the instruction pointer (IP) at the time of the panic is saved; from this IP, you can use the **id** command to disassemble the code. See **Figure 2** for sample output from the id command. From this assembly, you can then figure out where in the source code the panic occurred. To help determine the exact location of the panic, you can compile your source code into just the assembly part by using the "-S" option with **gcc**. The result is assembly intermixed with source code line numbers.

ss – Single steps an instruction.

bp – Sets a breakpoint.

continued on page 4

```
Entering kdb (current=0xc0355f80, pid 0) on processor 0 due to Keyboard Entry
[0]kdb> ?
Command        Usage             Description
_____________________________
md               <vaddr>           Display Memory Contents, also mdWcN, e.g. md8c1
mdr              <vaddr> <bytes>   Display Raw Memory

<– snip more commands than shown here –>

be             <bpnum>           Enable Breakpoint
bd             <bpnum>           Disable Breakpoint
ss                               Single Step
ssb                              Single step to branch/call
pt_regs         address           Format struct pt_regs

tkdata_drv     [num lines]       Dumps tank datum array (from drv)
tkthrd_drv                       Display tank thread state (from drv)
tankdata       [num lines]       Dumps tank datum array
tankthrd                         Display tank thread state
[0]kdb>
```

KDB output showing all of the available comands. Notice the last four commands, these are the KDB commands that our sample driver hooks into KDB. You can run these custom commands directly from the KDB debugger.

Figure 1 - Help Command

KDB (continued)

mm – Looks at memory. You can also modify memory contents.

dmesg – Prints out messages in the kernel message buffer. This enables you to view the most recent system messages before KDB was invoked.

This highlights just a few of the KDB commands. Refer to the KDB documentation for more information regarding all of the KDB commands.

In addition to the standard KDB commands, you can extend KDB and add your own commands!!

KDB Modules, Extending KDB, Custom KDB Commands

A powerful feature of KDB is the ability to extend KDB's functionality by adding your own debugging module, referred to as a KDB module. This enables you to add debugging commands, specific to your driver, directly into KDB itself. A KDB module is a kernel module, similar to a device driver; that registers itself with KDB when it is loaded. Like a driver, a KDB module contains **module_init()** and **module_exit()** functions and is loaded into the kernel using the **insmod** command.

When the module is loaded, the function, kdb_register(), is called for each command you wish to register with KDB. In the sample KDB module that accompanies this article, the function, tank_kdbm_init(), in the source file tank_kdbm.c contains the code used to register a debug command with KDB. The prototype of the kdb_register() function is:

```
kdb_register(char *cmd, kdb_func_t func,
             char *usage,char *help,
             short minlen);
```

Where:

cmd – Pointer to command string, this the actual command used at the KDB prompt.
func – Pointer to the function KDB will call when this command is entered.
usage – How this command issued, any arguments for this command.
help – Help text.
minlen - Minimum length of command, enables uses to abbreviate command.

Note the ***func*** argument. This is a pointer to a function that will be called when the command, pointed to by the ***cmd** argument, is executed from the KDB prompt. This function is where you actually implement (via code) your debugging command. If you look at the KDB source file, kdbmain.c, you will notice that all of the debugging commands are registered with the function kdb_register(). Also, there is a kdb_unregister() function that must be called when your module or driver unloads. If you don't unregister your debug command, KDB has no knowledge that the command and function for that command is no longer loaded. When the command is executed from the KDB prompt, the old function pointer is called (which by now points to some random memory) and BOOM!! Panic.

So how do you create your own KDB module? The registration of the KDB command is straightforward -- simply call the kdb_register() -- but what about the function KDB calls? Whenever you add a KDB command, you will also have to supply a corresponding function for this command.

The function prototype is:

```
static int
debug_func(int argc, const char **argv,
           const char **envp,
           struct pt_regs *regs)
```

Where:

argc - Number of argments passed to debug command.
****argv** - Pointer to array of arguments. Identical to how arguments are passed to a main() function.
****envp** - Pointer to array of environment variables.
***regs** - Pointer to registers, can be NULL.

This debug function will be called by KDB directly and will be executed while KDB has control of the system. As such, when writing your debug function, keep in mind the execution context of your debug function: a) system is halted, b) interrupts are disabled, c) the kernel has possibly panic, d) memory might be corrupted, and e) who knows what else.

Because of this context, you will probably be limited in the amount and type of things you can do within your debug function. You can call other kernel functions and subsystems (such as the network stack), but since the kernel is probably in an unstable state (which is why you are in the debugger), these calls may

continued on page 5

KDB (continued)

not be successful and will further destabilize the kernel. It is best to keep your debug function simple; limit the work to the specific debug command. Usually you will want to dump some data structures or set/reset some values within your driver. After these warnings and suggestions, remember, it is your code and project so if you need to perform any complex debugging work (like sending debug output over a network connection) then go for it!!

KDB provides a set of functions you should use whenever possible, which are exported from kdbmain.c. Some of the commonly used KDB functions are:

- kdb_printf()
- kdb_getarea_size()
- kdb_putarea_size()
- kdbgetenv()

For more details check the kdb code itself.

Often it is necessary to pass arguments with your debug command, such as the number of lines to display. In the sample KDB module, the number of lines to display is passed as an argument for both debug commands, **tankdata** and **tankthrd**. Arguments are passed using the **argc** and ****argv** parameters, just like a main() function. The number of arguments is contained in **argc**, and a pointer for each argument is contained in the char array pointed to by ****argv**.

In the previous paragraphs, I discussed how to create a stand alone KDB module you can load (via insmod) to debug you driver. You can also register KDB commands, using the kdb_register() function, directly from your driver; a separate module isn't necessary. The kernel build define, CONFIG_KDB, enables you to conditionally define your KDB functions (using kdb_register(), kdb_unregister(), etc.).

In the sample code, the driver contains two KDB debug functions, **kdb_TankDataArray** and **kdb_TankThrdState**. Both of these functions are exported from the driver and are called directly by the functions **dump_tankdata()** and t**ank_threadstate()** in the sample KDB module, tank_kdbm.ko. This illustrates how you can hook into KDB from a module or directly from from your driver. The debug functions themselves, **kdb_TankDataArray** and **kdb_TankThrdState** (both located in the driver), dump the tank data array and thread state.

Debugging Tips and Tricks with KDB

Here are some ways you can use KDB for debugging.

continued on page 6

Sample output of the KDB id command (x86 instruction set)

```
[0]kdb> id kdb_TankDataArray   [You can use a symbolic name or an address.]
0xdc1eab02:    jno    0xdc1eab79
0xdc1eab04:    gs
0xdc1eab05:    jae    0xdc1eab7b
0xdc1eab07:    push   %ebx
0xdc1eab08:    insb   (%dx),%es:(%edi)
0xdc1eab09:    popa
0xdc1eab0a:    jbe    0xdc1eab71
0xdc1eab0c:    sub    %dl,0x53(%ecx)
0xdc1eab0f:    je     0xdc1eab83
0xdc1eab11:    imul   $0x7453512c,0x67(%esi),%ebp
0xdc1eab18:    jb     0xdc1eab83
0xdc1eab1a:    outsb  %ds:(%esi),(%dx)
0xdc1eab1b:    addr16 sub $0x51,%al
0xdc1eab1e:    push   %ebx
0xdc1eab1f:    je     0xdc1eab93
0xdc1eab21:    imul   $0x29,0x67(%esi),%ebp
[0]kdb>
```

Figure 2 - Sample Output

KDB (continued)

Global Debug Variable. Since KDB is an assembly debugger, it is difficult to figure out what's on the stack or in memory. However if you create a global variable in your driver, then you can refer to this global by its symbolic name when using the KDM display memory command (mm).

For example, declare the following in your driver:

```
ulong g_currentstate;
```

Update this variable in your driver as necessary (per your logic flow). When KDB is invoked, you can easily display or modify the contents using the command, **mm g_currentstate**.

Circular Buffer. Create a circular buffer containing debug statements and save the pointer to the buffer in a global variable along with the current index (last entry made in buffer). When the kernel panics or another error condition occurs you can use KDB to dump the contents of the buffer by using the global pointer. This will give you a snapshot of activity and any recent errors that your driver has logged. This is really an old technique, but one that is very helpful and is usually implemented in most driver or embedded projects.

Table, Key Data Structs. If your driver uses any tables or key data structures, such as a structure containing device information, then you can create a KDB command to display the contents of the tables or any data structure in a nice readable format. This is where a custom KDB command is ideal, since you can extend KDB to display or modify driver specific information.

Invoking KDB Directly. How often have you seen the comment, "This cannot happen" in a source file? Nope, no way the code should be executing here. Right? I'm guilty of writing this comment and have been proven wrong when ***Yes*** the code does actually execute these particular lines of code. In situations like this, you can invoke KDB directly from your driver by simply using the macro, KDB_ENTER() (defined in include/asm/kdb.h). This will freeze the execution of your driver and enable you to debug your code and hopefully answer the question: Why is the code executing in the wrong place?

The sample driver contains an example of invoking KDB directly by using the sample user application. The user application uses the IOCTL_FORCE_KDB ioctl() call. Invoking KDB directly is great when your code detects a corrupted link list, invalid state, or an unknown input. You can freeze the driver state when the error occurred. This is especially helpful if you are running an overnight test and this error condition occurs during the night. In the morning, you can examine the state of your driver.

Modify Kernel Panic. Since you have access to the kernel source, you can directly modify the kernel panic function to display debug information specific to your driver. To do this, create a function pointer as a global variable within the kernel itself. When your driver loads, set this pointer to a function within your driver, when the system panics, check if the function pointer is NULL. If not, then call the function (which is in your driver) using the function pointer directly from the panic code. When your driver unloads, set the global function pointer to NULL. Unless you are building your driver directly into the kernel, you need to create a global variable since your exported debug function is not known to the kernel at boot time. The kernel panic function is called panic() and is in the kernel source file, kernel/panic.c. This is very handy when the system panics, since you can display debug information for your driver. While this technique is not KDB specific, it is good to know.

KDB – Sample Driver

The sample driver included with this article is a char driver that monitors the air pressure of a scuba tank during testing. Because of the high pressure used, scuba tanks have to be hydrostatically tested every five years. This test consists of filling the tank with water, submerging the filled tank within a larger test tank also filled with water, and then pressurizing the water within the tank. As the water within the scuba tank is pressurized, the tank itself expands; the amount of tank expansion is measured and must fall within a specific range. This procedure tests the strength of the tank itself. An interesting web page that describes this procedure in more detail can be found at: http://www.deep-six.com/page37.htm.

The sample driver monitors the pressure within the tank and the amount of expansion the tank experiences during the testing cycle. A real-world monitoring system would use some type of external hardware (like a PCI card or USB device) connected to the specialized test system to provide the real numbers; for our purposes the sample driver will generate sample data internally.

There are two data types: tank pressure and tank expansion. The sample driver maintains a circular queue of tank pressure and expansion using an array of structures of **tank_test_data_t** (defined in tank_ioctl.h). When the driver loads, the function InitMonitor() initializes this queue; the pointer to the queue buffer is saved in the driver structure **tank_mon_device_t**. A

continued on page 7

KDB (continued)

kernel thread, TestDataThread, is used to simulate a real device that would measure tank pressure and expansion. The sample application, tankapp, uses a set of ioctl commands to start, stop, and reset the thread. The sample application is also used to read the actual tank data and invoke KDB directly.

KDB Debug

Given this sample driver, how can we use KDB to help debug? The sample code presents two KDB commands, one displays the thread state and the other dumps the contents of the circular buffer. In our sample, the circular queue contains tank pressure and stretch, but queues are very common structures used by device drivers. Disk IO, events, network packets, cell packets, and pending actions are examples of things that get queued by different drivers. In addition to KDB commands the driver will invoke KDB directly when the user application calls the IOCTL_FORCE_KDB ioctl command(by running the command "tankapp –k").

The KDB module source code is located in tank_kdbm.c and registers the KDB commands **tankdata** and **tankthrd** when loaded. The sample driver registers the KDBM commands, **tkdata_drv** and **tkthrd_drv**, when it loads. The KDB module and driver both ultimately call the driver functions **kdb_TankDataArray** and **kdb_TankThrdState** functions implemented in the driver. The KDB 'help' command displays the commands registered by the driver and standalone KDB module.

To run the debug commands from the KDB prompt, simply enter the following commands:

tkdata_drv 10 - Displays 10 lines of tank test data.
tankthrd - Displays thread state.

Figure 3 shows sample output from our KDB commands.

Summary

KDB is a powerful tool and you should use it whenever you are developing a driver of any complexity. After reading this article and studying the sample driver, you should be able to incorporate KDB into your next driver.

References and resources:

http://linuxdevices.com/articles/AT3761062961.html
Good article on KDB commands.

http://oss.sgi.com/projects/kdb/
KDB patches, open source project.

http://www-128.ibm.com/developerworks/linux/library/l-kdbug/

Sample output of our custom KDB commands.

```
[0]kdb> tkdata_drv 10

Dumping tank monitor data (numlines: 10)
Index   Pressure    Stretch
0000    38          19
0001    40          20
0002    42          21
0003    44          22
0004    46          23
0005    48          24
0006    50          25
0007    52          26
0008    54          27
0009    56          28

[0]kdb> tankthrd
Tank Monitor thread is: STOPPED
           DatumCount: 48
         CurrentIndex: 48
```

Figure 3 - Custom Commands

Linux USB (continued)

www.goldenbits.com/newsletter/issue3/linuxusb_sample.zip

It is very helpful to have an understanding of USB technology itself when developing a USB driver. For more information about USB, you can read my previous newsletter (www.goldenbits.com/newsletter/issue2/gbtechnews_issue2.pdf), which goes into more detail about USB itself. Also, there is a lot of good information about USB technology in references such as: www.usb.org, *USB Complete*, Second Edition by Jan Axelson, www.intel.com/technology/usb, and sourceforge.net/projects/linux-usb.

Linux USB

The USB driver code is organized into the separate directories under drivers/usb: class, core, gadget, host, image, input, media, misc, net, serial, and storage. The core and host directories contain the key kernel source for managing the different types of USB drivers. These two directories along with the usb-skeleton.c sample driver are a great place to start learning about the details of writing a Linux USB driver.

Core USB support in Linux is provide by the kernel drivers:

usbcore - Core USB routines to send/receive urb packets, interfaces with individual USB devices.

ehci-hcd - The Enhanced Host Controller Interface (EHCI) is standard for USB 2.0 "high speed" (480 Mbit/sec, 60 Mbyte/sec) host controller hardware.

ohci-hcd - The Open Host Controller Interface (OHCI) is a standard for accessing USB 1.1 host controller hardware.

uhci-hcd - The Universal Host Controller Interface is a standard by Intel for accessing the USB hardware in the PC (which is also called the USB host controller).

continued on page 9

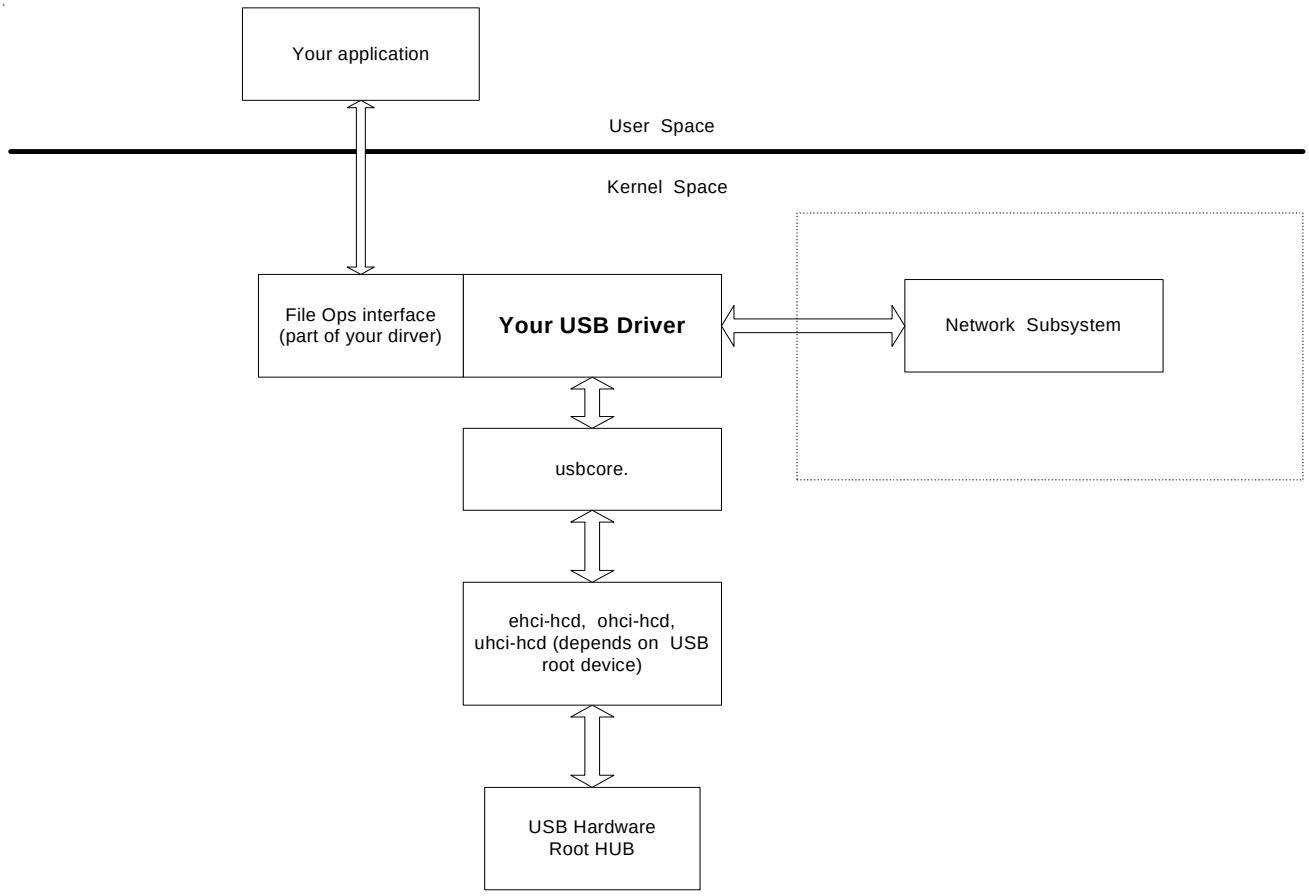


Figure 4 - USB Stack

Linux USB (continued)

These core drivers handle the USB details, but how does a USB device present itself in a useful manner to the user? The device driver for a particular USB device is responsible for attaching/registering itself to the appropriate kernel subsystem. For example, if your USB device is a network device, then the driver must register itself with the network subsystem using register_netdev(). It's your driver's responsibility to translate network requests into the correct USB requests. For some device types (classes), there is additional support, in terms of code, that your driver should use. For example, storage devices will need to use the usb-storage driver.

Figure 4 illustrates the organization of USB drivers within the kernel.

USB Descriptors

Arguably one of the most confusing aspects of developing a USB driver is understanding the USB descriptors and how they are used. The key descriptors are device, configuration, interface, and endpoint. A descriptor is nothing more than a data structure. When the designers of the USB standard tried to figure out a flexible and extensible way of configuring a USB device, they defined a set of data structures (the descriptors) that a USB device would present to describe itself and its capabilities. All of the USB descriptors are defined in include/linux/usb_ch9.h.

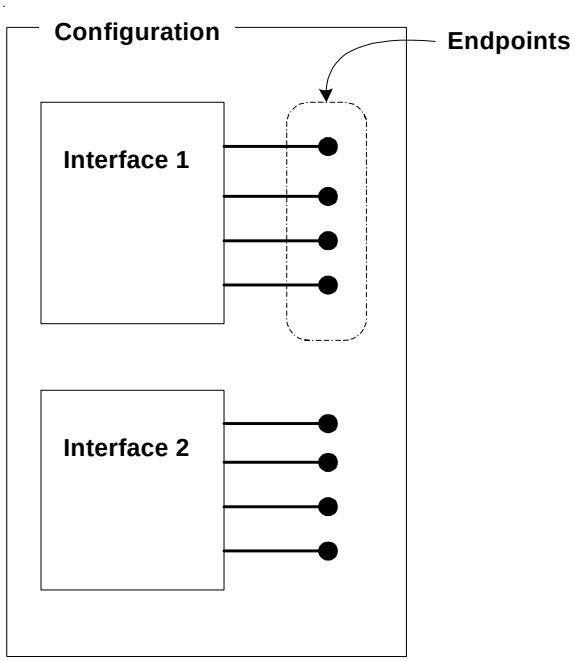


Figure 5 - Endpoints

A USB device has one device descriptor that describes the device as a whole. This descriptor contains the vendor and product ID and the number of configurations. Each configuration is described by a configuration descriptor. The USB standard allows for multiple configurations, but in practice, only one is almost always used. Each configuration contains a set of interface descriptors where each interface defines a set of endpoints. It is these endpoints that your driver communicates with. Your driver code ultimately uses a pipe handle to send/receive data with a USB device, the pipe handle actually represents an endpoint. Figure 5 (below) shows the logical organization of a USB interface and endpoints.

Device Driver Structure

Like all kernel drivers, a USB driver needs to call a set of functions to "register" itself with the correct kernel system. When a USB driver loads, in its driver init function it should call **usb_register()** with a pointer to a USB device structure - **struct usb_driver** , defined in include/linux/usb.h. This call essentially registers the driver with the usbcore module. The usbcore module will subsequently call two key functions, probe() and disconnect(), exported by the **usb_driver** struct when the USB device is connected or disconnected. In the sample driver, the probe function is called usb_crypt_probe().

The interesting stuff occurs when the **probe()** function is called, since here is where the driver will setup the necessary structures to send/receive data with your whiz-bang USB device. This is also the place where your driver should connect to the necessary kernel subsystem (network, storage, etc.). For example, if your USB device is a wireless network card, then you'll need to call register_netdev(). **probe()** is called with a pointer to a **struct usb_interface** and **struct usb_device_id**. From this, you can start building your private driver structures and setting up your interface endpoints. Note that **probe()** is called once for each interface your USB device has defined. If you create any global data structures in the **probe()** function, you need to check if these globals have been previously created, otherwise your code will accidentally create a second global structure. In the sample driver, a global structure called usb_crypt_devinfo is created only once; you will see a check to avoid a second instance of usb_crypt_devinfo from being created.

Once your driver has initialized the necessary data structures, it should call **usb_set_intfdata()**, which associates your private data (usb_crypt_devinfo in the sample driver) with this interface. You can later retrieve this data from the interface structure using the function, **usb_set_intfdata()**. This is very handy because it enables you to store a pointer to your private data with the USB interface structure.

continued on page 10

Linux USB (continued)

The **probe()** function is also the place you should register the driver with the other kernel subsystems, such as the network, storage, and character subsystems. If you want to be able to communicate directly with your USB driver from a user application, you'll need to register with the character device system; the function **usb_register_dev()** does this. The second argument to **usb_register_dev(),** struct **usb_class_driver**, contains the name of the /dev device entry to create, the device minor base number, permissions, and a pointer to the driver's file struct file_operations table. This call will create the /dev entries and return the device minor number in the interface structure.

usb_register_dev() should be called for each interface; if you have multiple interfaces then this function is called for each interface. For each USB interface, a separate character device is created in /dev. This makes sense because your USB device should be designed such that each interface implements different functionality. For example, in the sample USB device, there are two interfaces, one for encryption and a second for decryption. The sample driver registers these two interfaces using **usb_register_dev()**, which in turn creates two /dev entries, /dev/usb/crypt0 (for encryption) and /dev/usb/crypt1 (decryption). Depending on what the application needs, it can simply open the correct /dev entry.

When your driver exits the **probe()** function, it should be ready to send and receive data. The structure used to transmit data is the **struct urb** (defined in include/linux/usb.h).

Some Data Structures, Endpoints, and Pipes

One thing about USB drivers, there's no lack of structures. If you're going to write a USB driver, you need to familiarize yourself with the various structures used to send, receive, and configure. All of the USB related structures are contained in two header files, include/linux/usb.h, and include/linux/usb_ch9.h. Some of the key data structures are:

usb_device. Allocated and used to define your particular USB device, includes your vendor and product ID.

usb_host_config – Contains usb device configuration and interfaces.

usb_interface – Contains endpoints which are used to send/receive data.

usb_host_endpoint – Contains information for one endpont.

urb – Actual struct used to generate a USB request (read, write, or control). Functions used to allocate and initialize include: **usb_init_urb()** to initialize, then depending on the type of USB request you can use **usb_fill_int_urb()** to fill an interrupt URB, **usb_fill_bulk_urb()** to fill a bulk URB, **usb_fill_control_urb()** to fill a control URB.

USB devices send and receive data through a set of endpoints. An endpoint is a logical address that the USB device defines in the set of descriptors it sends to the USB host controller. Endpoints are unidirectional, meaning two separate endpoints are used to send and receive data. The direction of the endpoint is relative to the USB device, not the host. For example, an OUT endpoint means data is sent OUT from the USB device to the host. Endpoints are grouped into an interface, which is intended to represent logical functionality. I know this statement is a bit fuzzy, but the design goal is to organize your USB device functionality into individual interfaces, where each interface contains the necessary number of endpoints to accomplish the task. For example, if you are designing an interface for an MP3 player, then one interface may be used to transfer the MP3 bit streams and second interface to transfer song titles or licensing information.

In the Linux USB driver environment, endpoints are represented by pipes; a pipe is an **unsigned int** which contains various bitfields that define the endpoint and its characteristics (see usb.h for details). Your USB driver communicates to an endpoint using the appropriate pipe. You can think of a pipe as a handle of sorts which represents a specific endpoint. A set of macros used to manage pipes are defined in usb.h. Each macro starts with usb_pipe* along with a set of macros to create a specific type of pipe (such as usb_sndctrlpipe(), usb_rcvctrlpipe(), usb_sndisopipe(), and usb_rcviospipe()).

SGL and Larger Buffers

Copying to/from user space works with smaller buffers or when timing isn't an issue. However, another approach is to use a zero copy method. Using this method you lock down the user space buffers into kernel space and use one of the usb_map….() functions to build your scatter gather list and send it to the device. For the 2.6 kernel, you can lock down and map user pages into kernel space by using the get_user_pages() (mm/memory.c), function.

continued on page 11

Linux USB (continued)

In the sample driver, the endpoints and pipes are saved off in the usb_crypt_devinfo structure in the usb_crypt_probe() function. Once these pipes are created, they're ready for use.

Sending & Receiving, USB Host Functions

Now that we have our interfaces and pipes, how do you actually communicate with a USB device through the USB host controller? The answer: **urbs.** Your must get to know urbs, an urb (**U**sb **R**equest **B**lock) is ***the*** data structure used to send/receive data and contains all of the information the USB host driver needs. The specific urb fields of interest are the pipe handle, buffer, buffer length, completion function, and context (a very handy field which I'll talk about more). You can allocate urbs using the **usb_alloc_urb()** or **kmalloc()**, if you allocate the urb memory directly don't forget to initialize each urb using the function **usb_init_urb().**

To transfer data with your USB device, your driver needs to create the necessary kernel buffer and set up the urb using one of the helper functions, **usb_fill_bulk_urb()**, **usb_fill_init_urb()**, or **usb_fill_control_urb()**. In the sample driver, a pool of urbs are created by the CreateIOPool() function, where each urb is contained within a crypt_io_entry structure. When the sample application sends data to the device, the user space buffer is copied to the temporary kernel buffer using the copy_from_user() function; for reads the copy_to_user() function is used.

Another approach to transferring data is the use of scatter gather lists. Instead of using a single buffer, your driver can build a scatter gather list that describes a set of buffers you want to transfer with your USB device. Each entry in the scatter gather list would point to an individual packet to transfer, effectively building a queue of buffers. The function, usb_sg_init(), is used to initialize the scatter gather list, and then the actual transfer is done using usb_sg_wait().

Once you have your urb set up, you can send a urb synchronously or asynchronously. A little clarification here: you really "submit" the urb to the USB host driver, which in turn transmits the data or (if a read) generates a read request to the USB device. This enables you to structure your driver code to handle multiple urbs concurrently. In the sample driver code, usb_submit_urb() is used for both reads and writes. The function, usb_submit_urb(), is used to send a urb asynchronously. When the urb is sent or received, the host driver calls the callback function set in the complete field of the urb itself, with the value the urb context field as an argument. A pointer to the crypt_io_entry structure that contains the urb is used as the context, providing all of the context for the urb request while the host driver is processing our submit request. One advantage of using usb_smit_urb() is that it can be called from an interrupt routine.

However, this asynchronous approach does have one drawback: how do you handle the situation where a request never completes? For example, if the USB device fails or is suddenly disconnected, our driver needs a method for timing out the request. The sample driver uses a wait event to handle this situation. When a request is submitted, the code waits, with a timeout of 3 seconds, on an event embedded in the crypt_io_entry structure (the wait_queue_head_t IOWaitEvent field). If a request does time-out, the request is returned to the free pool and an error is returned to the user application. The wait code is defined by IO_TIMEOUT_WAITEVENT.

The other method of sending a request is **usb_bulk_msg()**, which sends the request synchronously; you provide a buffer, length, and timeout value. This function is a helper function of sorts, since it allocates and fills in the urb and then submits the urb to the host driver. This function is in core/message.c and is interesting to examine; eventually **usb_bulk_msg()** calls the asynchronous function **usb_submit_urb()** with a wait event to timeout the request. This is the same basic approach used in the sample driver, look for the IO_TIMEOUT_WAITEVENT define. **usb_bulk_msg()** cannot be called from within an interrupt and it is intended for bulk transfers only. For control messages, you can use **usb_control_msg()**.

Sample Driver

The sample driver manages our simple USB encryption device. The idea is to embed the actual encryption engine in the USB device itself. This enables a user to easily disconnect the device and secure it as necessary (for example in a vault). Our sample decryptor/encryptor device has two interfaces, one to encrypt and another to decrypt. The encrypting interface has two endpoints, one to send data to be encrypted and the second to read the encrypted data back. A TUSB3210KDBPDK development kit from Texas Instrument (see www.ti.com/usb) is used as our USB device. A sample application is also included and crypt_app.c contains the application code.

All of the driver code is contained in crypt_usb.c and the associated header file, crypt_usb.h. The code is small and is intended for demonstration purposes; a real production level driver would be more complicated and handle all of the nasty conditions when the USB device is disconnected while application IO requests are being processed.

continued on page 12

Linux USB (continued)

Like all standard kernel modules, the driver's entry point is defined by module_init(). When the driver module is loaded, the usb_crypt_init function is called. This is where the driver registers itself with the USB host driver via the **usb_register()** function. After registering, the USB host driver will call our probe function when our USB device is plugged into the system. The probe function, **usb_crypt_probe()**, is where the sample driver sets up the interfaces, this function and the **DumpUSBDevInfo()** function are good places to start viewing the code. The device interfaces, endpoints, and pipes are saved in the crypt_interface_info structure -- which is part of our private device structure (usb_crypt_devinfo) -- and are indexed by the defines *_INDEX.

A common function, crypt_xfer_func(), is used to send and receive data with the USB device. The crypt_io_entry structure contains the necessary urb and is used to track the urb after it is submitted to the host driver. A pool of these structures are created on startup by the CreateIOPool() function. All of the requests are submitted asynchronously using the **usb_submit_urb()** function. On completion, the callback function, USB_IO_Complete_cb(), is called with a pointer to the crypt_io_entry structure for that specific urb. The crypt_io_entry structure provides us with the necessary context to track the state of the urb**.**

When an urb is submitted, the code will block until the urb is completed by host driver. However, sometimes it might be desirable to timeout a request, rather than hang the calling application. The define, IO_TIMEOUT_WAITEVENT, handles this situation. Instead of blocking forever, the code will wait 3 seconds and then timeout and return an error to the calling application.

Summary

So there you have it, a primer on how to write a USB device driver for Linux. Please feel free to email me with any questions or comments: deang@goldenbits.com.

■

References and resources:

USB Complete, Second Edition by Jan Axelson. Lakeview Research.

www.usb.org & www.linux-usb.org

Project Experience - Several projects Golden Bits has successfuly completed.

SCSI Port Driver for Fibre Channel. Designed the operating system layer for a SCSI storage driver (XP, Win2K, NT, Linux) for a Fibre Channel HBA (host bus adapter – PCI/SBUS card).

Embedded Network Appliance. Developed an embedded monitoring device for web sites and/or other data center systems. The device uses the uC/OS real time kernel running on a Motorola ColdFire processor (MCF5206e).

WDM, NDIS Device Drivers . Developed a WDM and NDIS device driver for a prototype wireless system.

Parallel Search Engine. Developed a search engine that distributes database queries to other systems; the search runs in parallel on the supporting systems and the results are written (through bulk inserts) back into the database.

Satellite Set Top Box. Developed a script language and compiler used to code the television UI (guide menus, channel select).

Camera Control. Wrote highly customized Windows user interface with special graphics and custom controls. The application presents the user with camera images with graphic information overlaid (in near real time), camera configuration information, and product inspection information.

Embedded TCP/IP Protocol Stack. Wrote an NT packet driver using NDIS driver subsystem to simulate a mobile network for a military application. The embedded stack executed under NT, and the packet driver simulated network device IO.