Day 6-1

Assignment:

- HW6 due Thursday
- Project Proposals

Today's Topics:

- IFTTT
- Google Compute Engine
- Linux Kernel Module

07-2 Device Driver Basics

Using kernel modules

Free Electrons

Loadable kernel modules

- Modules: add a given functionality to the kernel (drivers, filesystem support, and many others)
- Can be loaded and unloaded at any time, only when their functionality is needed
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs)
- Also useful to reduce boot time: you don't spent time initializing devices and kernel features that you only need later
- Caution: once loaded, have full access to the whole kernel address space. No particular protection

Minimal Device Driver

```
/**
 * @file    hello.c
 * @author  Derek Molloy
 * @date    4 April 2015
 * @version 0.1
 * @brief  An introductory "Hello World!" loadable kernel module (LKM) that can display a messag
 * in the /var/log/kern.log file when the module is loaded and removed. The module can accept an
 * argument when it is loaded -- the name, which appears in the kernel log files.
 * @see http://www.derekmolloy.ie/ for a full description and follow-up descriptions.
*/

#include <linux/init.h>             // Macros used to mark up functions e.g., __init __exit
#include <linux/module.h>           // Core header for loading LKMs into the kernel
#include <linux/kernel.h>           // Contains types, macros, functions for the kernel

MODULE_LICENSE("GPL");              ///< The license type -- this affects runtime behavior
MODULE_AUTHOR("Derek Molloy");      ///< The author -- visible when you use modinfo
MODULE_DESCRIPTION("A simple Linux driver for the BBB.");  ///< The description -- see modinfo
MODULE_VERSION("0.1");              ///< The version of the module
```

Minimal Device Driver

```
static char *name = "world";        ///< An example LKM argument -- default value is "world"
module_param(name, charp, S_IRUGO); ///< Param desc. charp = char ptr, S_IRUGO can be read/not
changed
MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");  ///< parameter description

/** @brief The LKM initialization function
 *  The static keyword restricts the visibility of the function to within this C file. The __init
 *  macro means that for a built-in driver (not a LKM) the function is only used at initializatio
 *  time and that it can be discarded and its memory freed up after that point.
 *  @return returns 0 if successful
 */
static int __init helloBBB_init(void){
   printk(KERN_INFO "EBB: Hello %s from the BBB LKM!\n", name);
   return 0;
}
```

Minimal Device Driver

```
/** @brief The LKM cleanup function
 *  Similar to the initialization function, it is static. The __exit macro notifies that if this
 *  code is used for a built-in driver (not a LKM) that this function is not required.
 */
static void __exit helloBBB_exit(void){
   printk(KERN_INFO "EBB: Goodbye %s from the BBB LKM!\n", name);
}

/** @brief A module must use the module_init() module_exit() macros from linux/init.h, which
 *  identify the initialization function at insertion time and the cleanup function (as
 *  listed above)
 */
module_init(helloBBB_init);
module_exit(helloBBB_exit);
```

Module Build Output – Out-of-tree

- Load headers for current version of kernel

```
bone$ apt update
bone$ apt install linux-headers-`uname –r`
```

- Clone Molloy's examples

```
bone$ git clone https://github.com/derekmolloy/exploringBB.git
```

- Find hello world example

```
bone$ cd exploringBB/extras/kernel/hello
bone$ cat Makefile
obj-m+=hello.o
all:
        make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

- **Compile with**

```
bone$ make
```

Loading and Unloading a Module

```
bone$ insmod hello.ko
bone$ dmesg | tail -4
[    9.106206] snd-usb-audio 1-1:1.0: usb_probe_interface
[    9.106244] snd-usb-audio 1-1:1.0: usb_probe_interface - got id
[    9.813239] usbcore: registered new interface driver snd-usb-
  audio
[Oct 7 14:20] EBB: Hello world from the BBB LKM!
bone$ rmmod hello
bone$ dmesg | tail -4
[    9.106244] snd-usb-audio 1-1:1.0: usb_probe_interface - got id
[    9.813239] usbcore: registered new interface driver snd-usb-
  audio
[Oct 7 14:20] EBB: Hello world from the BBB LKM!
[ +20.535832] EBB: Goodbye world from the BBB LKM!
```

Example Driver with Parameter

```
///< An example LKM argument -- default value is "world"
static char *name = "world";

///< Param desc. charp = char ptr, S_IRUGO can be read/not changed
module_param(name, charp, S_IRUGO);

///< parameter description
MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");
```

Passing Parameters to a Module

```
bone$ insmod hello.ko name=Mark
[Oct 7 14:23] EBB: Hello Mark from the BBB LKM!
bone$ rmmod hello
[Oct 7 15:23] EBB: Goodbye Mark from the BBB LKM!
bone$ insmod hello.ko
[Oct 7 15:24] EBB: Hello world from the BBB LKM!
```

Other module commands

```
bone$ lsmod
bone$ modinfo.ko hello
bone$ depmod
```

(creates modules.dep.bin)

- Go play with them

Day 6-2

Assignment:

- HW6 due today
- Project Proposals

Today's Topics:

- Linux Kernel Modules – file operations

Adding File System Ops to Hello.c

- http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/ has a long example about adding file system operations to **`hello.c`**
- Look it over
- Creates a new device (**`/dev/ebbchar`**)
- You can read and write it
- Do it

Major and Minor Number

- Every device has a major and minor number

```
$ ls -ls /dev/console
0 crw------- 1 yoder root 5, 1 2011-02-06 17:57 /dev/console
```

- Used by the kernel to identify the correct device driver when the device is accessed
- Device numbers *used* to be statically assigned
- See …**`/Documentation/devices.txt`**

```
 5 char   Alternate TTY devices
                  0 = /dev/tty            Current TTY device
                  1 = /dev/console  System console
                  2 = /dev/ptmx           PTY master multiplex
                 64 = /dev/cua0           Callout device for ttyS0
```

- The text uses static assignment

```
234-239        UNASSIGNED
240-254 char   LOCAL/EXPERIMENTAL USE
```

Character Drivers

- Character devices are identified by a 'c'
- Block devices a 'b'

```
bone$ ls –l /dev
crw-rw---- 1 root i2c      89,   0 Oct 12 11:10 i2c-0
crw-rw---- 1 root i2c      89,   1 Oct 12 11:11 i2c-1
crw-rw---- 1 root i2c      89,   2 Oct 12 11:11 i2c-2
drwxr-xr-x 3 root root         100 Oct 12 11:11 input
crw-r----- 1 root kmem      1,   2 Oct 12 11:11 kmem
crw-r--r-- 1 root root      1,  11 Oct 12 11:11 kmsg
drwxr-xr-x 2 root root          60 Dec 31  1969 lightnvm
crw-rw---- 1 root disk     10, 237 Oct 12 11:10 loop-control
drwxr-xr-x 2 root root          60 Oct 12 11:10 mapper
crw-r----- 1 root kmem      1,   1 Oct 12 11:11 mem
crw------- 1 root root     10,  57 Oct 12 11:11 memory_bandwidth
brw-rw---- 1 root disk    179,   0 Oct 12 11:11 mmcblk0
brw-rw---- 1 root disk    179,   1 Oct 12 11:11 mmcblk0p1
brw-rw---- 1 root disk    179,   8 Oct 12 11:11 mmcblk1
brw-rw---- 1 root disk    179,  16 Oct 12 11:11 mmcblk1boot0
```

Assigning Device Numbers

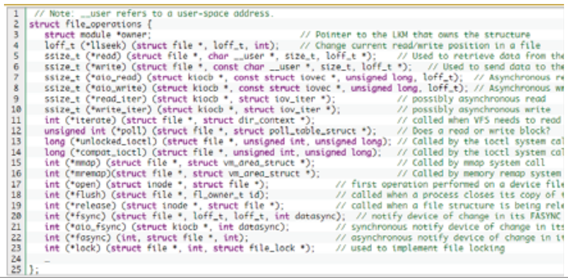
- You an manually create a device file and associate it with your device

```
bone$ mknod /dev/test c 92 1
```

- You have to make sure the device (92) isn't in use.
- Look in `/usr/src/`uname –r`/include/uapi/linux/major.h`
- But there is a better way…

File Operations Data Structure

- The `file_operations` data structure holds pointers to functions within a driver that allows you to define the behavior of certain file operations
- It is defined in …`/include/linux/fs.h`



```
 // Note: __user refers to a user-space address.
struct file_operations {
   struct module *owner;                             // Pointer to the LKM that owns the structure
   loff_t (*llseek) (struct file *, loff_t, int);    // Change current read/write position in a file
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);    // Used to retrieve data from the
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);   // Used to send data to the
   ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);  // Asynchronous re
   ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t); // Asynchronous wr
   ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);            // possibly asynchronous read
   ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);           // possibly asynchronous write
   int (*iterate) (struct file *, struct dir_context *);                // called when VFS needs to read
   unsigned int (*poll) (struct file *, struct poll_table_struct *);    // Does a read or write block?
   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long); // Called by the ioctl system cal
   long (*compat_ioctl) (struct file *, unsigned int, unsigned long);   // Called by the ioctl system cal
   int (*mmap) (struct file *, struct vm_area_struct *);                // Called by mmap system call
   int (*mremap)(struct file *, struct vm_area_struct *);               // Called by memory remap system
   int (*open) (struct inode *, struct file *);             // first operation performed on a device file
   int (*flush) (struct file *, fl_owner_t id);             // called when a process closes its copy of
   int (*release) (struct inode *, struct file *);          // called when a file structure is being rele
   int (*fsync) (struct file *, loff_t, loff_t, int datasync);  // notify device of change in its FASYNC
   int (*aio_fsync) (struct kiocb *, int datasync);         // synchronous notify device of change in its
   int (*fasync) (int, struct file *, int);                 // asynchronous notify device of change in it
   int (*lock) (struct file *, int, struct file_lock *);    // used to implement file locking
   …
};
```

Driver File System Operations

- Once a device driver is loaded into the live kernel…
  - **`open()`** is called each time the device is opened from user space
  - **`read()`** is called when data is sent from the device to user space
  - **`write()`** is called when data is sent from user space to the device
  - **`release()`** is called when the device is closed in user space
- Think in terms of reading and writing a file…

```
fd = open("file", …
read(fd, …
write(fd, …
close(fd)
```

open/release additions to hello.c

```
static int    majorNumber;        ///< Stores the device number -- determined automatically
// The prototype functions for the character driver -- must come before the struct definition
static int     dev_open(struct inode *, struct file *);
static int     dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

/** Devices are represented as file structure in the kernel. The file_operations structure from
*  /linux/fs.h lists the callback functions that you wish to associated with your file operations
*  using a C99 syntax structure. char devices usually implement open, read, write and release calls
*/
static struct file_operations fops =
{
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .release = dev_release,
};
```

From: exploringBB/extras/kernel/ebbchar

ebbchar_init

```
#define  DEVICE_NAME "ebbchar"  ///< The device will appear at /dev/ebbchar using this value
#define  CLASS_NAME  "ebb"        ///< The device class -- this is a character device driver

static int __init ebbchar_init(void){
   printk(KERN_INFO "EBBChar: Initializing the EBBChar LKM\n");

   // Try to dynamically allocate a major number for the device -- more difficult but worth it
   majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
   if (majorNumber<0){
      printk(KERN_ALERT "EBBChar failed to register a major number\n");
      return majorNumber;
   }
   printk(KERN_INFO "EBBChar: registered correctly with major number %d\n", majorNumber);

   // Register the device class
   ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);
   if (IS_ERR(ebbcharClass)){     // Check for error and clean up if there is
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to register device class\n");
      return PTR_ERR(ebbcharClass);   // Correct way to return an error on a pointer
   }
   printk(KERN_INFO "EBBChar: device class registered correctly\n");
```

ebbchar_init

```
   // Register the device driver
   ebbcharDevice = device_create(ebbcharClass, NULL,
                 MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
   // Clean up if there is an error
   if(IS_ERR(ebbcharDevice)){
      class_destroy(ebbcharClass);
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to create the device\n");
      return PTR_ERR(ebbcharDevice);
   }
   printk(KERN_INFO "EBBChar: device class created
correctly\n"); // Made it! device was initialized
   return 0;
}
```

ebbchar_exit

```
static void __exit ebbchar_exit(void){
   // remove the device
  device_destroy(ebbcharClass, MKDEV(majorNumber, 0));
  // unregister the device class
  class_unregister(ebbcharClass);
  // remove the device class
  class_destroy(ebbcharClass);
  // unregister the major number
  unregister_chrdev(majorNumber, DEVICE_NAME);
  printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
}
```

dev_open/dev_release

```
static int dev_open(struct inode *inodep, struct file *filep){
   numberOpens++;
   printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n", numberOpens);
   return 0;
}

static int dev_release(struct inode *inodep, struct file *filep){
   printk(KERN_INFO "EBBChar: Device successfully closed\n");
   return 0;
}
```

dev_write

```
static ssize_t dev_write(struct file *filep,
       const char *buffer, size_t len, loff_t *offset){
   // appending received string with its length
   sprintf(message, "%s(%d letters)", buffer, len);
   // store the length of the stored message
   size_of_message = strlen(message);
   printk(KERN_INFO "EBBChar: Received %d characters from the
user\n", len);
   return len;
}
```

dev_read

```
static ssize_t dev_read(struct file *filep, char *buffer, size_t len,
                    loff_t *offset){
   int error_count = 0;
   // copy_to_user has the format ( * to, *from, size) and returns 0 on success
   error_count = copy_to_user(buffer, message, size_of_message);

   if (error_count==0){            // if true then have success
      printk(KERN_INFO "EBBChar: Sent %d characters to the user\n",
                 size_of_message);
      return (size_of_message=0);  // clear the position to the start and return 0
   }
   else {
      printk(KERN_INFO "EBBChar: Failed to send %d characters to the user\n",
                    error_count);
      return -EFAULT;      // Failed -- return a bad address message (i.e. -14)
   }
}
```

/proc/devices

```
Character
devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
212 DVB
226 drm
245 ebbchar
246 uio
247 ttyGS
248 hidraw
249 bsg
250 watchdog
251 ptp
252 pps
253 media
254 rtc

Block devices:
259 blkext
  8 sd
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
```

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first
- Example: the usb-storage module depends on the scsi_mod, libusual and usbcore modules
- Dependencies are described in /lib/modules/<kernel-version>/modules.dep

/lib/modules/4.4.21-ti-r47/modules.dep

```
kernel/arch/arm/crypto/aes-arm.ko:
kernel/arch/arm/crypto/aes-arm-bs.ko:
      kernel/arch/arm/crypto/aes-arm.ko
      kernel/crypto/ablk_helper.ko
      kernel/crypto/cryptd.ko
kernel/arch/arm/crypto/sha1-arm.ko:
kernel/arch/arm/crypto/sha1-arm-neon.ko:
kernel/arch/arm/crypto/sha1-arm.ko
kernel/arch/arm/crypto/sha256-arm.ko:
kernel/arch/arm/crypto/sha512-arm.ko:
```

Kernel log

When a new module is loaded,
related information is available in the kernel log

- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the **`dmesg`** command
 (“**d**iagnostic **mes**sa**g**e”)
- Kernel log messages are also displayed in the system console (messages can be filtered by level using **`/proc/sys/kernel/printk`**)

printk

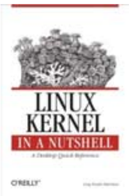
- **`/proc/sys/kernel/printk`**
- The four values in this file are
  - *console_loglevel*,
  - *default_message_loglevel*,
  - *minimum_console_level* and
  - *default_console_loglevel*.
- These values influence **`printk()`** behavior when printing or logging error messages
- Messages with a higher priority than *console_loglevel* will be printed to the console
- Messages without an explicit priority will be printed with priority *default_message_level*

http://www.tin.org/bin/man.cgi?section=5&topic=proc

Kernel log levels

0 (KERN_EMERG)	The system is unusable
1 (KERN_ALERT)	Actions that must be taken care of immediately
2 (KERN_CRIT)	Critical conditions
3 (KERN_ERR)	Noncritical error conditions
4 (KERN_WARNING)	Warning conditions that should be taken care of
5 (KERN_NOTICE)	Normal, but significant events
6 (KERN_INFO)	Informational messages that require no action
7 (KERN_DEBUG)	Kernel debugging messages, output by the

Useful reading



Linux Kernel in a Nutshell, Dec 2006

- By Greg Kroah-Hartman, O'Reilly
  http://www.kroah.com/lkn/
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book
  for easy electronic searches!
  Available as single PDF file on
  http://free-electrons.com/community/kernel/lkn/
- In **`exercises/pptx`**

Useful reading too

Linux Device Drivers, Third Edition, February 2005

- By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
  http://lwn.net/Kernel/LDD3/
- Freely available on-line!
  Great companion to the printed book
  for easy electronic searches!
  Available as single PDF file
- LDD3 is current as of the 2.6.10 kernel (Old?)
- In **`exercises/pptx`**