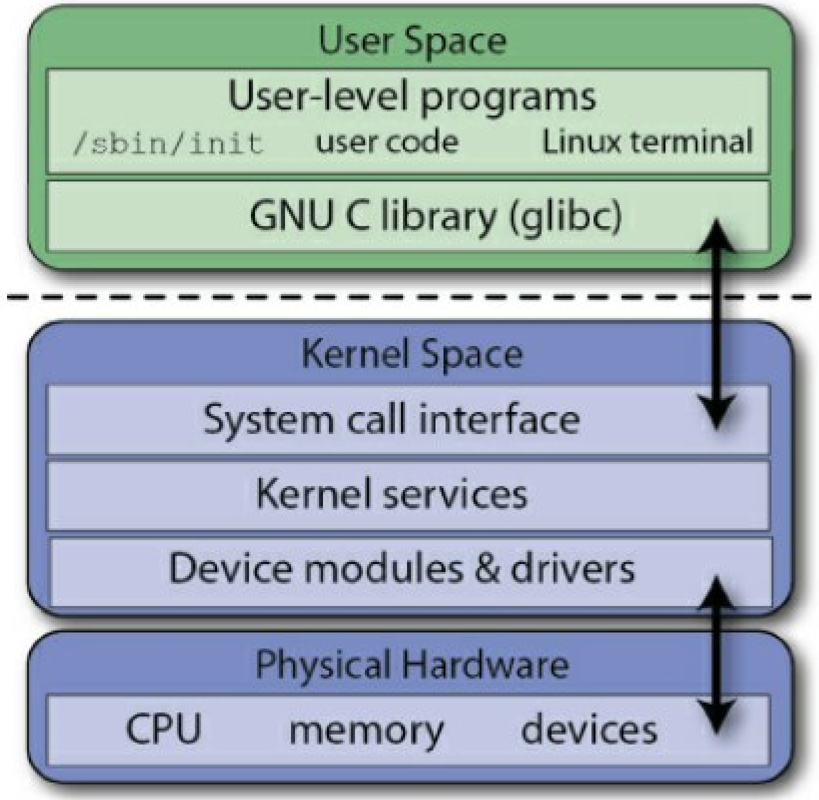
Building an LKM

Building a loadable kernel module in Linux using the beaglebone black.

Loadable Kernel Modules

- Loadable Kernel Modules (LKM) can be used to control peripherals on the beaglebone black.
- These Kernel Modules run in kernel space, and care must be taken when coding them.
- Kernel code has support for interrupts!
- Kernel code is difficult to write and debug ☹ recommend user space unless you absolutely have to!



Loadable Kernel Modules Overview

- Written in C - but they are not programs, there is no main function!
- Do not execute sequentially – registers to handle requests!
- Do not clean up automatically – you must release resources!
- Do not have printf() statements – cannot access user mode libraries!
- Can be interrupted – can be used by multiple processes at the same time!
- High level of execution priviledge – be careful of using excessive CPU time!
- No floating point support!

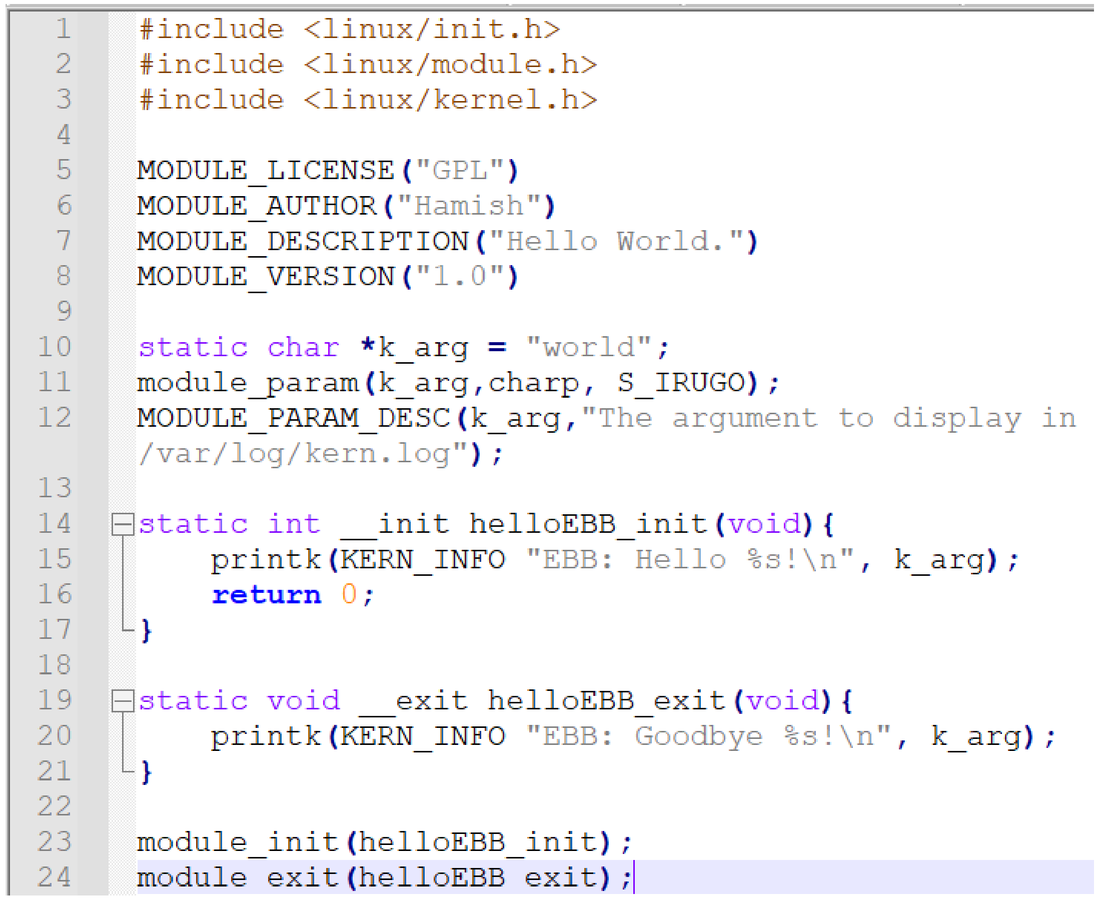
Hello World LKM driver example

This example provides a code for an example LKM.

The kernel module prints “Hello World” when no kernel argument is given, otherwise it prints “Hello <argument>” where argument is the kernel argument given.

Beware, back-up your system before you begin. It is easy to accidentally cause a system crash when writing and testing LKMs.

- We include the linux kernel module headers. (lines 1-3).
- We include some module information in lines 5-8 that can be obtained using the modinfo command.
- We define a variable in lines 10-12.
- The LKM has a module entry and a module exit.



```
 1  #include <linux/init.h>
 2  #include <linux/module.h>
 3  #include <linux/kernel.h>
 4
 5  MODULE_LICENSE("GPL")
 6  MODULE_AUTHOR("Hamish")
 7  MODULE_DESCRIPTION("Hello World.")
 8  MODULE_VERSION("1.0")
 9
10  static char *k_arg = "world";
11  module_param(k_arg,charp, S_IRUGO);
12  MODULE_PARAM_DESC(k_arg,"The argument to display in
    /var/log/kern.log");
13
14  static int __init helloEBB_init(void){
15      printk(KERN_INFO "EBB: Hello %s!\n", k_arg);
16      return 0;
17  }
18
19  static void __exit helloEBB_exit(void){
20      printk(KERN_INFO "EBB: Goodbye %s!\n", k_arg);
21  }
22
23  module_init(helloEBB_init);
24  module_exit(helloEBB_exit);
```

Parts of the module

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

- The first header file contains macro definitions for our LKM.
- The second header module.h contains definitions for creating our LKM.
- The third header file kernel.h contains the definitions of kernel types, macros and functions.

Parts of the module

```
MODULE_LICENSE("GPL")
MODULE_AUTHOR("Hamish")
MODULE_DESCRIPTION("Hello World.")
MODULE_VERSION("1.0")
```

These lines of code provide information about our kernel module. They allow our module to report information about the module to the user using the modinfo command.

Note: the license type alters the behaviour of the module. If the license differs from GPL it can ‘taint’ the kernel.

Parts of the module

```
static char *k_arg = "world";
module_param(k_arg,charp, S_IRUGO);
MODULE_PARAM_DESC(k_arg,"The argument to display in /var/log/kern.log");
```

We use the static keyword when defining k_arg to limit the scope to this *.c file. Global variables can be accessed anywhere in the kernel, and we do not wish to cause conflicts with other modules.

The module_param() macro contains three arguments, the parameter name used in the module, the type of the parameter, in this case a character pointer, and the access permissions to sysfs

Parts of the module

```
static int __init helloEBB_init(void){
        printk(KERN_INFO "EBB: Hello %s!\n", k_arg);
        return 0;
}
```

- A static keyword indicates this function is only visible in this *.c file.
- The __init macro states this is only used once for a built-in driver, and can be thrown away and memory freed afterwards (does not apply to a LKM).
- The printk command prints Hello to a kernel message, kernel messages are accessible using the dmesg command.

Parts of the module

```
static void __exit helloEBB_exit(void){
        printk(KERN_INFO "EBB: Goodbye %s!\n", k_arg);
}
```

- A static keyword indicates this function is only visible in this *.c file.
- The exit macro indicates that this function is not required for a built-in driver, but is required for a LKM.
- Once again we print a kernel message this time saying Goodbye modified by the kernel argument k_arg.

Parts of the module

```
module_init(helloEBB_init);
module_exit(helloEBB_exit);
```

The kernel module must use the module_init() and module_exit() macros from linux/init.h

These functions identify the initialisation function at insertion time and the cleanup function at exit.

In this example, these call the helloEBB_init function on initialisation and the helloEBB_exit function on exit.

Building the LKM using make

```
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
```

- To build a kernel module we can use make on the beaglebone itself.
- The first line is the goal definition and specifies the module to be built.
- The rest of the build is a standard makefile
- The call $(shell uname –r) specifies the kernel version used, the –C option changes the directory before building. M=$(PWD) specifies that the module resides in the current working directory.

Building the module on the beaglebone

```
sudo apt-cache search linux-headers-$(uname -r) # Find the kernel headers
sudo apt install linux-headers-$(uname -r) # install the kernel headers
# In the directory of your module
make # build the system
ls -l # show the output *.ko file
```

- To build a kernel module we need to have the linux headers installed.
- The linux headers are kernel version specific so we can use $(uname –r) to obtain the correct version.
- We then navigate to the directory containing your module, and run the make command. The ls command will then show the output.

Loading the module

```
insmod hello.ko # install the kernel module
lsmod | grep hello # observe the loaded kernel module
modinfo hello.ko # display the module info we inserted in our code
rmmod hello.ko # remove the kernel module
```

- Install module (insmod) will load the LKM into the kernel.
- The list modules command (lsmod) will display the installed kernel modules. If we pipe the output with the global regular expression print (grep) we can limit the output to our LKM.
- The module information we placed in our LKM can be displayed by using the modinfo command.
- Remove module (rmmod) will unload the module from the kernel.

Interacting with the module

```
insmod hello.ko k_arg=Hamish # this replaces world with Hamish when run
tail -f /var/log/kern.log # we can use tail to observe the log output from our LKM
# The custom parameter can be viewed in our module directory by running
cat k_arg # this will return Hamish as previously defined
```

- The module parameter can be modified when the LKM is loaded.
- We can view the output of the LKM from our printk commands by interrogating the kernel log files.
- We also can navigate into our module directory and interrogate k_arg directly using the concatenate command (cat).

Hello World Summary

- In this lecture we have covered creating a loadable kernel module (LKM).

- The structure of the LKM

- How to build an LKM

- How to interact with a LKM