Linux Kernel Modules

An Overview for Embedded Systems

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- What is a “kernel module”?
- Kernel modules are not device drivers!
- Entry and exit codes
- Error handling
- Building and testing

What is a “kernel module”?

Linux can load object files into a running kernel:

- These object files are called “kernel modules”

Yes, ordinary object files:

- Code, data
- ‘Exportable” symbols
- Unresolved references to external symbols

What is a “kernel module”?

Once loaded, a kernel module becomes kernel code:

- Loaded into kernel memory space
- Can access kernel services
- Can implement new kernel services
- Can offer interfaces for user applications

What is a “kernel module”?

And, like all kernel code:

- Cannot use an FPU, if present
- Cannot touch user memory directly

What is a “kernel module”?

“Kernel modules are device drivers, right?”

- No!

Device drivers are code:

- ... and a kernel module might contain such code
- ... but might not
- (Linux device drivers aren’t what you think they are)

What is a “kernel module”?

It’s more accurate to say:

- Device drivers are often implemented in kernel modules
- A kernel module *might contain* a device driver, but might not

Minimal Example

```
 1    #include <linux/module.h>
 2
 3    #define MODULE_NAME "skeleton"
 4
 5    int __init example_init (void)
 6    {
 7       printk(KERN_ERR "%s: %s()\n", MODULE_NAME, __FUNCTION__);
 8       return 0;
 9    }
10
11    void __exit example_exit (void)
12    {
13       printk(KERN_ERR "%s: %s()\n", MODULE_NAME, __FUNCTION__);
14    }
15
16    module_init(example_init);
17    module_exit(example_exit);
```

Entry and Exit Functions

`module_init()`

- Macro that refers to module entry code
- Invoked when module is loaded
- Module “expunged” on nonzero return value
- Limit one per module

Entry and Exit Functions

`module_exit()`

- Macro that refers to module exit code
- Invoked when module is unloaded
- No return value
- Module “expunged” from memory at exit
- Limit one per module

Entry and Exit Functions

Things to do on entry:

- Allocate module-global memory
- Register device drivers
- Register interfaces
- ...

On exit:

- Undo all the above!

The `__init` and `__exit` Macros

`__init`

- Code gets placed into the `init.text` section
- After module initialization, `init.text` memory is discarded
- After initialization, the code no longer exists!

```
Freeing unused kernel memory:  68k freed
```

The `__init` and `__exit` Macros

`__exit`

- Marks code needed only during exit
- Kernel knows it can set this code aside until later

Handling Errors

```
int __init my_init_function(void)
{
  int err;
  /* registration takes a pointer and a name */
  err = register_this(ptr1, "skull");
  if (err) goto fail_this;
  err = register_that(ptr2, "skull");
  if (err) goto fail_that;
  err = register_those(ptr3, "skull");
  if (err) goto fail_those;
  return 0; /* success */
  fail_those: unregister_that(ptr2, "skull");
  fail_that: unregister_this(ptr1, "skull");
  fail_this: return err; /* propagate the error */
}
```

Source: Chapter 2, Linux Device Drivers, 3rd. ed.

Handling Errors

The general idea:

- Break initialization down into atomic steps
- Provide labels for error-recovery code
- Stack up recovery steps in reverse order

On an error:

- Enter at the right place via `goto`
- Fall through recovery steps to completion

Handling Errors

Initialization:

```
int __init my_init_function(void)
{
  int err;

  /* step 1 */
  err = init_step_1(...);
  if (err)
      goto step_1_failed;

  /* step 2 */
  err = init_step_2(...);
  if (err)
      goto step_2_failed;

  ...
```

Handling Errors

... and the recovery:

```
    ...

    /* success! */
    return 0;

step_3_failed:
    /* undo step 2 */
    ...;

step_2_failed:
    /* undo step 1 */
    ...;

step_1_failed:
    /* (nothing to do) */

    return err;
}
```

Handling Errors

```
int __init my_init_function(void)
{
  struct a *p_a;
  struct b *p_b;

  /* step 1 */
  p_a = kmalloc(sizeof(struct a), GFP_KERNEL);
  if (!p_a)
      goto a_alloc_failed;

  /* step 2 */
  p_b = kmalloc(sizeof(struct b), GFP_KERNEL);
  if (!p_b)
      goto b_alloc_failed;

  ...
```

Handling Errors

```
  ...

  /* success! */
  return 0;

b_alloc_failed:
  kfree(p_a);

a_alloc_failed:
  /* (nothing to kfree() */

  return err;
}
```

Handling Errors

Each step must be atomic:

- It must completely succeed, or completely fail
- Subdivide into smaller steps as necessary
- Refactor code into subroutines

Module Information Macros

`MODULE_LICENSE`

- Communicates the module's distribution license to the kernel
- Proper use prevents "tainting" the kernel
- Some kernel symbols are only exported to GPL-licensed code

```
MODULE_LICENSE(‘‘GPL’’);
MODULE_LICENSE(‘‘Copyright (c) 2007...’’);
```

Module Information Macros

Recognized license descriptions:

```
"GPL"
"GPL v2"
"GPL and additional rights"
"Dual BSD/GPL"
"Proprietary"
```

The `tainted` flag

"*Tainting* the kernel?!"

- Proprietary modules, when loaded, set the `tainted` flag
- Flag shows up in OOPS messages and elsewhere
- Requires reboot to reset
- `/proc/sys/kernel/tainted`

The `tainted` flag

```
Unable to handle kernel NULL pointer dereference at virtual address 0
pgd = c0004000
[00000000] *pgd=00000000
Internal error: Oops: 8f5
Modules linked in:
CPU: 0
PC is at do_initcalls+0x28/0xe0
LR is at init+0x34/0xf0
pc : [<c0008a0c>]    lr : [<c002409c>]    Not tainted
sp : c037ffc4  ip : c037ffe4  fp : c037ffe0
r10: 00000000  r9 : 00000000  r8 : 00000000
r7 : c001e8c4  r6 : 00000000  r5 : c037e000  r4 : c001e684
r3 : 00000000  r2 : c036dee4  r1 : 00000000  r0 : c036a4e0
Flags: Nzcv  IRQs on  FIQs on  Mode SVC\_32  Segment kernel
Control: 397F  Table: A0004000  DAC: 00000017
Process swapper (pid: 1, stack limit = 0xc037e194)
...
```

Module Information Macros

Introduced in 2.4.10 by Alan Cox:

> “I get so many bug reports caused by the nvidia modules...”
>
> [http://lwn.net/2001/0906/a/ac-license.php3]

- Some developers ignore posts of tainted OOPS messages

Module Information Macros

"So why don't I just take that flag out?"

- You could, but the kernel would still be "tainted"

> "... since its arguably digital rights management [you] might face five years in jail in the USA for doing so 8)"
>
> [http://lwn.net/2001/0906/a/ac-tainted.php3]

Module Information Macros

`MODULE_AUTHOR`

- Identifies the module's author

```
MODULE_AUTHOR(‘‘Bill Gatliff <bgat@billgatliff.com>’’);
```

Module Information Macros

`MODULE_DESCRIPTION`

- Describes what the module is or does

```
MODULE_DESCRIPTION(‘‘TSC2003 touch screen driver...’’);
```

Module Information Macros

`MODULE_VERSION`

- Module version information

```
MODULE_VERSION(‘‘1.23-rc4’’);
```

skeleton_module.c

```
 1  #include <linux/module.h>
 2
 3  #define MODULE_NAME "skeleton"
 4
 5  int __init example_init (void)
 6  {
 7     printk(KERN_ERR "%s: %s()\n", MODULE_NAME, __FUNCTION__);
 8     return 0;
 9  }
10
11  void __exit example_exit (void)
12  {
13     printk(KERN_ERR "%s: %s()\n", MODULE_NAME, __FUNCTION__);
14  }
15
16  module_init(example_init);
17  module_exit(example_exit);
18
19  MODULE_LICENSE("GPL");
20  MODULE_VERSION("1.2-rc3");
21  MODULE_AUTHOR("Bill Gatliff <bgat@billgatliff.com>");
22  MODULE_DESCRIPTION("A do-nothing example");
```

Building and Loading a Module

The kernel's build machinery is called "Kbuild"

- Makefiles, configurators, configuration files
- Distributed with the Linux kernel's source code
- Also used by other projects, e.g. Busybox

```
$ make ARCH=arm menuconfig
```

Building and Loading a Module

Even proprietary modules need Kbuild:

- Selects the right compiler, flags, command scripts
- Matches up kernel, module version information

... but proprietary modules:

- Cannot mix with GPL kernel code
- Must maintain related kernel sources separately

Building and Loading a Module

Create a `Makefile` in your working directory:

```
1    obj-m += example_module.o
2
3    all:
4            make -C $(KERNELSRC) M=$(PWD) modules
5
6    clean:
7            make -C $(KERNELSRC) M=$(PWD) clean
```

Building and Loading a Module

Build!

```
$ export KERNELSRC=.../path/to/kernel/source
$ make

$ modinfo example_module.ko
filename: example_module.ko
license:  Copyright (c) 2007...
author: Bill Gatliff <bgat@billgatliff.com>

$ make clean
```

Building and Loading a Module

Test!

```
$ insmod example_module.ko
$ dmesg
  ...
  example_module: example_init called
$ rmmod example_module
$ dmesg
  ...
  example_module: example_init called
  example_module: example_exit called
```

Building and Loading a Module

Actually:

- ... including cross-specific stuff
- ... omitting redundancy in template `Makefile`
- (Yes, you could eliminate the `Makefile` altogether!)

```
$ make KERNELRC=/path/to/kernel/source
     obj-m=skeleton.o ARCH=arm CROSS_COMPILE=${TARGET}-
```

Linux Kernel Modules

An Overview for Embedded Systems

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer