Linux Device Driver Design Tradeoffs

Presented by

David W. Hawkins
dwh@ovro.caltech.edu

California Institute of Technology
Owens Valley Radio Observatory
Tutorial coverage:

1. Where do you start?

2. The basics
   2.1. Kernel modules
   2.2. Device drivers
   2.3. Hotplug, sysfs, and udev
   2.4. Kernel timers
   2.5. Interrupts
   2.6. Data buffering

3. PCI drivers

4. A 'real-world' PCI driver

5. Cross-development
Where do you start?

The big picture:

• Linux Device Drivers, 3rd Ed, 2005, Corbet, Rubini, Kroah-Hartman

• Linux Kernel Development, 2nd Ed, 2005, Love

• Understanding the Linux Kernel, 3rd Ed, 2005, Bovet and Cesati

Understanding:

• "Use the source, Luke"

• Write some code
Kernel modules

• Add new functionality to the kernel
• simple_module.c

    static __init int simple_init(void)
    {
        LOG_ERROR("Module loaded.\n");
        LOG_DEBUG("This is a debug message.\n");
        return 0;
    }

    static __exit void simple_exit(void)
    {
        LOG_ERROR("Module unloaded\n");
    }

    module_init(simple_init);
    module_exit(simple_exit);
Device drivers

- For controlling devices (of course!)
- Device types; character, block, network
- This tutorial will focus on character devices
- The kernel implementation of the user-space calls: open(), close(), read(), write(), ioctl(), lseek(), select(), and mmap().
- Implement call-backs: struct file_operations
Example driver

- simple_driver.c
  - module load/unload
  - install-time parameter passing, ioctl modification and interrogation
  - open, release, read, write, ioctl, llseek, poll, mmap, fasync
  - dynamic creation of /dev nodes

- simple_driver_test.c
  - make sure it works!
Device nodes; major and minor numbers

- Load the simple driver

  ```
  # insmod simple_driver.ko simple_device_count=3
  simple_minor_count=2
  # ls -al /dev/simple*
  crw------- 1 root root 253, 0 Jan 16 11:34 /dev/simple_a0
  crw------- 1 root root 253, 1 Jan 16 11:34 /dev/simple_a1
  crw------- 1 root root 253, 2 Jan 16 11:34 /dev/simple_b0
  crw------- 1 root root 253, 3 Jan 16 11:34 /dev/simple_b1
  crw------- 1 root root 253, 4 Jan 16 11:34 /dev/simple_c0
  crw------- 1 root root 253, 5 Jan 16 11:34 /dev/simple_c1
  ```

- 3 devices; a, b, c, each with 2 functions: 0, 1

- Split the minor numbers among devices and device functions
Hotplug, sysfs, udev

- Hotplug:
  - kernel event generation
  - user-space call-back via /sbin/hotplug

- sysfs:
  - /sys filesystem kernel-to-user-space interface

- udev:
  - user-space scripts

- Dynamic loading of drivers

- Dynamic creation of /dev nodes
  - /dev only contains installed devices

- Details at: man hotplug, and man udev
Automatic creation of /dev nodes

- (LDD3, p388) class_simple
- create /sys/class/some-widget
- main purpose is for device node creation
- API underwent changes; see simple_driver.c
- What does it create?

```
# tree /sys/class/simple_driver/
/sys/class/simple_driver/
|-- simple_a0
| `-- dev
|-- simple_a1
| `-- dev
`-- dev
|-- simple_c1
| `-- dev
`-- dev
```

Device names and numbers needed by udev

253:0
253:1
253:5
Device permissions

- User-space script sets the permissions

```sh
# /etc/udev/permissions.d/20-simple.permissions
#
# Change the permissions on the simple device nodes to
# owner dwh, group mm, with read-write permissions.
#
simple_*:dwh:mm:0660
```

- Reload the driver and list the device nodes

```sh
# ls -al /dev/simple*
crw-rw---- 1 dwh mm 253, 0 Jan 16 12:01 /dev/simple_a0
crw-rw---- 1 dwh mm 253, 1 Jan 16 12:01 /dev/simple_a1
crw-rw---- 1 dwh mm 253, 2 Jan 16 12:01 /dev/simple_b0
crw-rw---- 1 dwh mm 253, 3 Jan 16 12:01 /dev/simple_b1
crw-rw---- 1 dwh mm 253, 4 Jan 16 12:01 /dev/simple_c0
crw-rw---- 1 dwh mm 253, 5 Jan 16 12:01 /dev/simple_c1
```
Kernel timers

- `simple_timer.c` and `simple_timer_test.c`

- Device timeouts, device polling, delays

```c
/* Timer setup */
dev->fired = 0;
dev->timer.function = &simple_timer_handler;
dev->timer.data = (unsigned long)dev;
mod_timer(&dev->timer, jiffies + HZ);

/* Wait for timer */
wait_event_interruptible(dev->wait, dev->fired != 0)) ... 

/* Timer handler */
dev->fired = 1;
wake_up_interruptible(&dev->wait);
```
Interrupts

- simple_irq.c and simple_irq_test.c

- Example uses the parallel port IRQ

- Driver init claims resources, exit releases them (resource flags are used to simplify the logic)

- The driver creates a 1s timer and the timer handler triggers an IRQ

- User-space read()/write()/select() block until the IRQ occurs, or a 2s timeout occurs

- read() fills the user-buffer with 0, 1, 2, 3, ...

- write() prints the user-buffer to the system log

- Data movement: copy_from_user(), copy_to_user()
Data buffering

- simple_buffer.c

- IRQ writes a timestamp to the driver buffer

- write() writes to the driver buffer

- read() reads from the driver buffer

- two producers, one consumer
  - since an IRQ handler is involved, a spinlock is required
  - use semaphores for process-to-process protection
simple_buffer.c testing

- read() test
  
  # insmod simple_buffer.ko
  insmod: error inserting 'simple_buffer.ko': -1
  Input/output error
  # modprobe -r lp
  # modprobe -r parport_pc
  # insmod simple_buffer.ko
  # cat /dev/simple
  cat: /dev/simple: No data available
  # cat /dev/simple
  1137452165.999000
  1137452166.998848
  1137452167.998695
  1137452168.998546
  1137452169.998390
simple_buffer.c testing

- read()/write() test

First terminal:

```bash
# rmmod simple_buffer
# insmod simple_buffer.ko simple_timer_enable=0 simple_timeout_enable=0
# cat /dev/simple
```

Second terminal:

```bash
# echo "Hello" > /dev/simple
```

Back on the first terminal:

```bash
# cat /dev/simple
Hello
```
simple_buffer.c testing

- mix both the IRQ test and the echo test

```c
# rmmod simple_buffer
# insmod simple_buffer.ko
# cat /dev/simple
1137454000.999038
1137454001.998886
1137454002.998733
Hello from another terminal
1137454003.998582
1137454004.998431
1137454005.998278
```

- buffer locking ensures each message is written in its entirety
Driver buffering

- a spinlock protects a resource shared by an IRQ and a process
- you can NOT sleep while holding a spinlock
- copy_to_user and copy_from_user can sleep
- kmalloc can sleep
- data can not be copied directly from the user-space buffer to the kernel buffer, or vice versa
- the simple_buffer.c solution is to use an intermediate buffer
- double buffering, and queues of buffers can eliminate the redundant copying
How long is a 1s sleep?

![Graph showing the error relative to absolute time (ms) over time (minutes).]
How long is a 1s sleep?
How good is NTP?
PCI drivers

- Test equipment
  - High-speed oscilloscope
  - Logic analyzer
  - Logic analyzer adapters and probes
  - Logic analyzer 'inverse assembler' software
- Development hosts; x86 (LE), PowerPC (BE)
- Development boards (peripherals)
- Processor JTAG debugger
Development host

Logic analyzer

'Yosemite' PowerPC 440EP
PCI-to-cPCI adapter
cPCI analyzer interface
cPCI peripheral
Finding PCI devices

- Earlier kernels found PCI devices at load-time;\n  pci_find.c

- Hotswap PCI devices can load/unload anytime

- struct pci_driver probe() and remove() calls handle the dynamic nature of hotswap PCI

- pci_probe.c
  - init calls pci_register_driver()
  - exit calls pci_unregister_driver()
  - probe() and remove() log a message
  - New PCI IDs can be added at run-time
    echo "10b5:9054" > /sys/bus/pci/drivers/pci_probe/new_id
  - its not a real 'device driver'
PCI resources

- NET2270 resources

```bash
# lspci -s 00:0c.0 -v
00:0c.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 02)
  Subsystem: PLX Technology, Inc. PCI <-> IOBus Bridge
  Flags: bus master, medium devsel, latency 64, IRQ 11
  Memory at e6a01000 (32-bit, non-prefetchable) [size=256]
  I/O ports at e800 [size=256]
  Memory at e6800000 (32-bit, non-prefetchable) [size=1M]
  Memory at e6900000 (32-bit, non-prefetchable) [size=1M]
  Capabilities: [40] Power Management version 1
  Capabilities: [48] #00 [0080]
```

- Kernel resource accessors:
  - `pci_resource_start()`, `pci_resource_len()`, `pci_resource_flags()
  - I/O: `request_region()`, `iopport_map()
  - Memory: `request_mem_region()`, `ioremap()`
Generic PCI I/O driver

- Testing new hardware
  - Read and write registers
  - Generate interrupts and poll status registers

- pci_io.c
  - Requests PCI I/O and Memory regions
  - Creates a character device per PCI device
  - Minor numbers distinguish PCI regions (BARs)
  - read()/write()/lseek() for I/O regions
  - read()/write()/lseek()/mmap() for memory regions
  - shows how to use pci_dev->dev.driver_data, and file->private_data
### Generic PCI I/O driver testing

- **NET2270 PLX-9054 registers (bytes)**

```plaintext
CMD> db 0 100
00: 00 00 F0 FF 01 00 00 00 00 00 20 01 00 05 30 00
10: 00 00 00 00 00 00 00 00 40 01 03 42 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40: 78 56 34 12 EF CD AB 89 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: B5 10 54 90 0A 00 00 00 78 56 34 12 EF CD AB 89
80: 43 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 43 00 00 00 00 00 00 00 00 00 00 00
A0: 00 00 00 00 00 00 00 00 00 10 10 00 00 00 00 20
B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0: 00 00 F0 FF 01 00 10 00 C3 03 00 00 00 00 00 00
```
Generic PCI I/O driver testing

- **NET2270 PLX-9054 registers (x86, LE, 32-bit words)**

```
CMD> d 0 100
00: FFF00000 00000001 01200000 00300500
10: 00000000 00000000 42030140 00000000
20: 00000000 00000000 00000000 00000000
30: 00000000 00000000 00000008 00000000
40: 12345678 89ABCDEF 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 00F010100 180F767E
70: 905410B5 0000000A 12345678 89ABCDEF
80: 00000043 00000000 00000000 00000000
90: 00000000 00000043 00000000 00000000
A0: 00000000 00000000 00001010 01200000
B0: 00000000 00000000 00000000 00000000
C0: 00000002 00000000 00000000 00000000
D0: 00000000 00000000 00000000 00000000
E0: 00000000 00000000 00000050 00000000
F0: FFF00000 00100001 000003C3 00000000
```
Generic PCI I/O driver testing

- NET2270 PLX-9054 registers (PPC, BE, 32-bit words)

```
CMD> d 0 100
00: 0000F0FF 01000000 00002001 00053000
10: 00000000 00000000 40010342 00000000
20: 00000000 00000000 00000000 00000000
30: 00000000 08000000 00000000 00000000
40: 78563412 EFCDAB89 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 001010F 7E760F18
70: B5105490 0A000000 78563412 EFCDAB89
80: 43000000 00000000 00000000 00000000
90: 00000000 43000000 00000000 00000000
A0: 00000000 00000000 10100000 00002001
B0: 00000000 00000000 00000000 00000000
C0: 02000000 00000000 00000000 00000000
D0: 00000000 00000000 00000000 00000000
E0: 00000000 00000000 50000000 00000000
F0: 0000F0FF 01001000 C3030000 00000000
```
PCI Transactions

- PCI single read (FRAME# asserted for one clock)
PCI Transactions

- PCI burst read (FRAME# asserted for multiple clocks)
CPU PCI read of 8-bytes

Bus bandwidth: 3.3MB/s
Board-to-board DMA PCI write of 1024-bytes

Bus bandwidth: 125MB/s
Board-to-board DMA PCI write of 256-bytes (through a PCI-to-PCI bridge)

Bridge starts accepting data

Bridge disconnects

Bridge issues retry

Transfer continues

Bus bandwidth: 65MB/s
Board-to-board DMA PCI read of 256-bytes (through a PCI-to-PCI bridge)

Bus bandwidth: 24MB/s
A 'real-world' PCI driver

- COBRA Correlator System
- Compatible with command-line tools
- Device functions:
  - `stdio` and `stderr` for DSP terminal interface
  - `control` for commands (binary interface)
  - `data` for correlation data
  - `monitor` for monitor data
- Separation of functionality at the driver-level simplifies user-space application design
The COBRA driver
The CARMA Array
The SZA Correlator System
A COBRA Correlator Board
Cross-development

- 'Cheap' hardware:
  - Linksys WRT54G wireless router, $50, MIPs CPU (www.linksys.com/support, www.openwrt.org)
  - Hauppauge MediaMVP, $100, PowerPC 405 CPU (www.hauppauge.com, mvpmc.sourceforge.net)
  - eBay;
    PowerMac G3/G4 $100
    Artesyn PowerPC 750 PrPMC $50 (1/16/06, 125pcs)
  - free; your co-worker's Mac + Ubuntu Linux Live CD
Cross-development

- Specific hardware: ~$1k
  - Yosemite 440EP PowerPC kit; $800
  - Freescale Coldfire M5485EVB; $850
  - Altera NIOS II development kit; $995

- Processor JTAG debugger
  - Abatron BDI2000 $3k + $1k per processor type
Building a kernel and drivers

- **Denx Embedded Linux Development Kit (ELDK)**
  (www.denx.de)

- **Kernel-build environment variables (PowerPC 440EP)**

  ```
  $ export PATH=/opt/eldk-3.1.1/bin:/opt/eldk-3.1.1/usr/bin:/$PATH
  $ export ARCH=ppc
  $ export CROSS_COMPILE=ppc_4xxFP-
  $ export KERNEL_DIR=/home/dwh/yosemite/linux-2.6.13
  $ export BUILD_DIR=/home/dwh/yosemite/linux-2.6.13-ppc
  $ mkdir $BUILD_DIR
  $ cd $KERNEL_DIR
  $ make O=$BUILD_DIR
  $ xconfig
  $ make O=$BUILD_DIR
  $ uImage
  $ modules
  ```

- **Driver-build environment variables**

  ```
  $ export KERNEL_DIR=$BUILD_DIR
  $ cd /home/dwh/drivers
  $ make
  ```