Linux *Kernel Hacking* Free Course 2003

UNIVERSITA' DEGLI STUDI DI ROMA TorVergata Università

D. P. Bovet, M. Cesati, R. Spennato

Implementing an I/O driver for the USB sensor device

The USB software layers (1)

- The code executed by the host controller handles the traffic across the USB bus (see previous lecture)
- The host controller interfaces the PC through a set of I/O ports
- A first software layer (UHDCI or OHDCI) manages the I/O ports of the host controller and allows the upper level software layer to issue requests to USB devices by means of USB Request Blocks (URB): see `drivers/usb/uhci.c`, `drivers/usb/usb-uhci.c`, and `drivers/usb/usb-ohdci.c`

The USB software layers (2)

- A second software layer (USB core) includes a collection of helper routines that should be used when implementing a new USB I/O driver: see `drivers/usb/usb.c`

UHCID or OHCID?

- Both software layers implement basically the same functions
- Depending on the type of host controller interface (UHCI or OHCI) should be used
- Use `cat /proc/pci` to check whether the host controller of your computer makes use of the UHCI or OHCI interface
- Configure the kernel accordingly

The main descriptors

- Once an USB device has been configured, several types of descriptors have been initialized (see `include/linux/usb.h`):
  - Device descriptor
  - Configuration descriptor
  - Interface descriptor
  - Endpoint descriptor
  - String descriptor

The HID descriptors

- A USB device that requires human interaction is called Human Interface Device or HID (our infrared device is a HID device)
- The microcontroller of a HID device sends additional packets containing two additional descriptors:
  - HID descriptor
  - Report descriptor

The I/O driver for the USB infrared device

The main data structure

- The main data structure used by our driver:

```
struct iosp_context {
   struct usb_device *usbdev;
   int open;
   unsigned char data_buf[SIZE];
   wait_queue_head_t  wait;
   int wakeup;
   struct urb irq;
   } *piosp_context;
```

- *usbdev: pointer to the data structure that contains the usb device descriptors
- open: equal to 1 if the device driver is open
- data_buf[SIZE]: the code of the key pushed
- wait: pointer to the I/O driver waitqueue
- irq: the URB structure used to setup a data transfer

The file operations

- We make use of the following file operations:

```
static struct file_operations
iosp_fops = {
   owner:    THIS_MODULE,
   read:     iosp_read,
   open:     iosp_open,
   release:  iosp_close,
   };
```

- Our device is an input device, we don't need a write operation

The device descriptor

```
static struct usb_device_id iosp_id [] = {
   { USB_DEVICE(0x4242, 0x0001) },
   { }
   };

MODULE_DEVICE_TABLE (usb, iosp_id);
```

- Thanks to this structure, the USB core is able to select the proper I/O driver for our device
- We have used as VendorId: 0x4242 (thanks to John Hyde) and as ProductId: 0x0001
- The `MODULE_DEVICE_TABLE` macro exports the data structure defined inside the module

The `usb_driver` structure

```
static struct usb_driver iosp_driver = {
   name:      "iosp",
   probe:     iosp_probe,
   disconnect:  iosp_disconnect,
   id_table: iosp_id,
   fops:      &iosp_fops,
   minor:     USB_IOSP_MINOR,
   };
```

- All USB I/O drivers must define such data structure
- `USB_IOSP_MINOR` (80) is the first minor number assigned to our I/O device
- The 15 minor numbers following `USB_IOSP_MINOR` will also be assigned to our device
- All USB char devices have the same major number: 180

The two basic module functions

- The I/O driver is compiled as a module, it will thus require:

```
static int __init usb_iosp_init(void)
{
   if (usb_register(&iosp_driver) < 0)
      return -1;
   return 0;
}

static void __exit usb_iosp_cleanup(void)
{
   usb_deregister(&iosp_driver);
}
```

- `usb_register( )` and `usb_deregister( )` are USB core functions related to device registration

The `probe()` function (1)

```
static void *iosp_probe(struct usb_device *dev,
unsigned int ifnum, const struct usb_device_id *id)
   {
   struct iosp_context *iosp;
   struct usb_endpoint_descriptor *endpoint;
   int pipe,maxp;

   if(!(iosp = kmalloc(sizeof(struct iosp_context),
                       GFP_KERNEL)))
       return NULL;
   memset(iosp, 0, sizeof(struct iosp_context));

   iosp->usbdev = dev;
    endpoint=dev-> config[0].interface[ifnum].
             altsetting[0].endpoint;

   if((endpoint->bmAttributes & 3) != 3)
       return NULL;
```

The `probe()` function (2)

```
 pipe=usb_rcvintpipe(dev,endpoint->bEndpointAddress);
 maxp=usb_maxpacket(dev, pipe, usb_pipeout(pipe));

 FILL_INT_URB(&iosp->irq, dev, pipe, iosp->data_buf,
     maxp, usb_iosp_irq, iosp, endpoint->bInterval);

 piosp_context = iosp;
 return iosp;
 }
```

- `probe( )` is invoked by USB core when it has identified the I/O driver
- The `usb_driver` structure used by our driver is initialized, a pipe of type receive interrupt is defined and the `FILL_INT_URB` macro is used to set up an URB

The `disconnect()` function

```
   static void iosp_disconnect(struct usb_device
   *dev, void *ptr)
      {
      struct iosp_context *iosp = (struct
   iosp_context *) ptr;

      usb_unlink_urb(&iosp->irq);
      kfree(iosp);
      piosp_context = NULL;
      }
```

- This function is invoked when the I/O device is disconnected from the USB bus
- `usb_unlink_urb( )` removes pending URBs
- The `iosp` I/O driver main data structure is released

The `open()` function (1)

```
static int iosp_open (struct inode *inode,
struct file *file)
   {
   struct iosp_context *iosp;

   if(piosp_context==NULL)
      return -1;
   iosp = piosp_context;

   if(iosp->open)
      return -1;
   iosp->open++;

   iosp->irq.dev = iosp->usbdev;
   if(usb_submit_urb(&iosp->irq))
      return -1;

   init_waitqueue_head(&iosp->wait);
   return 0;
   }
```

The `open()` function (2)

- Checks whether the device is already opened, if so return an error code
- Starts the data transfer, that is the polling, by invoking `usb_submit_urb( )`
- Invokes `init_waitqueue_head( )` to initialize the I/O driver wait queue

The `close()` function

```
static int iosp_close (struct inode *inode,
struct file *file)
   {
   struct iosp_context *iosp;

   if(piosp_context==NULL)
      return -1;
   iosp = piosp_context;

   if(!iosp->open)
      return -1;
   iosp->open--;

   usb_unlink_urb(&iosp->irq);

   return 0;
   }
```

- Returns aan error code if the device file is already closed
- Invokes the usb core function `usb_unlink_urb()` to release the urb

The `read()` function (1)

```
static ssize_t iosp_read (struct file *file,
char *buf, size_t len, loff_t *ppos)
  {
  struct iosp_context *iosp;

  if(piosp_context==NULL)
      return -1;
  iosp = piosp_context;
  if(!iosp->open)
      return -1;
  wait_event(iosp->wait, iosp->wakeup);
  if(copy_to_user (buf, iosp->data_buf,
                   SIZE))
      return -1;
  iosp->wakeup = 0;
  return 0;
  }
```

The `read()` function (2)

- Checks whether `iosp`, the pointer to our I/O device descriptor, is not null
- Verifies that the device file is open
- Invokes `wait_event()` on the `iosp->wakeup` condition
- When the condition occurs, invokes `copy_to_user( )` to copy the data into User Mode address space
- Reset the `iosp->wakeup` condition

The `usb_iosp_irq()` function

```
static void usb_iosp_irq (struct urb *urb)
  {
  struct iosp_context *iosp;
  iosp = piosp_context;
  iosp->wakeup = 1;
  wake_up(&iosp->wait);
  }
```

- This function is invoked by the USB core at the end of a data transfer
- The function invokes `wake_up()` to wake up the process blocked by the `wait_event()`

How to use our I/O driver (1)

- We wrote a simple application program to test our USB driver: a special daemon wakes up each time we press a key of the remote control
- The action performed by the daemon is specifed in a configuration file
- Two kinds of actions:
  - <key name>:#<symbolic name of sequence of chars>
  - <key name>:<name executable program>

```
8:#prior
0:#next
ttx:nedit
4:xterm
green:#altf4
```

How to use our I/O driver (2)

- In order to fool up an application program and let it believe that we pushed a keyboard key, we must send to the application the scancode of the key
- We use the `/dev/console` device associated with the selected console (the active one), by issuing an `ioctl()`
- Function that sends the scancode to the console:

```
void send_key(char car)
{
int tty_cur, mode;
tty_cur = open("/dev/console", O_RDONLY);
ioctl(tty_cur, KDGKBMODE, &mode);
if(!mode)
   ioctl(tty_cur, TIOCSTI, &car);
close(tty_cur);
}
```