Linux “Miscdev” Devices

Simple Character Device Interfaces

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- What is a *device driver*?
- What is a *device node*?
- What is an *interface*?

Overview

Roadmap:

- `struct file_operations`
- `struct miscdevice`
- Examples

Device Drivers

Device driver:

> “Distinct, [software] ’black boxes’ that make a particular piece of hardware respond to a well-defined internal programming interface...”
>
> Alessandro Rubini. Linux Device Drivers. O’Reilly, 2005.

Interface:

> “1. *n*. The meeting point of two entities.”
>
> Barr, Ganssle. Embedded Systems Dictionary. CMP Books, 2003.

Interfaces

An interface is *not* a device driver!

- But a lot of code tries to believe otherwise
- ... to their own detriment

The two concepts are distinct!

- ... and Linux keeps them that way
- ... and if you do too, you'll write better code

Interfaces

Kernel-side interfaces:

- SPI, I2C, USB
- IDE, SCSI
- ...

User-visible interfaces:

- `char`
- `block`

Device Nodes

Byte-oriented interface:

- `open()`, `close()`
- `read()`, `write()`, `ioctl()`
- Exposed to users via *device nodes*

```
int fd = open(‘‘/dev/ttyS0’’, O_RDWR);
int ret = write(fd, ‘‘hello, world!\n’’, 14);
```

Device Nodes

File-like abstraction:

- Maps user requests to interface methods
- Major, minor numbers
- Looks like a file to user applications

```
$ ls -l /dev/console
crw------- 1 bgat root 5,1 Mar 7 08:30 /dev/console
```

Device Nodes

Major, *minor* numbers:

- Uniquely identifies the device node
- Maps to a specific set of operations
- The “identity” of the device node

Typically:

- Major numbers specify device groups
- Minor numbers enumerate grouped devices

Device Nodes

```
$ ls -l /dev/ttyS?
crw-rw---- 1 root uucp 4, 64 Mar 6 11:58 /dev/ttyS0
crw-rw---- 1 root uucp 4, 65 Mar 6 11:58 /dev/ttyS1
crw-rw---- 1 root uucp 4, 66 Mar 6 11:58 /dev/ttyS2
crw-rw---- 1 root uucp 4, 67 Mar 6 11:58 /dev/ttyS3
crw-rw---- 1 root uucp 4, 68 Mar 6 11:58 /dev/ttyS4
crw-rw---- 1 root uucp 4, 69 Mar 6 11:58 /dev/ttyS5
crw-rw---- 1 root uucp 4, 70 Mar 6 11:58 /dev/ttyS6
crw-rw---- 1 root uucp 4, 71 Mar 6 11:58 /dev/ttyS7
crw-rw---- 1 root uucp 4, 72 Mar 6 11:58 /dev/ttyS8
crw-rw---- 1 root uucp 4, 73 Mar 6 11:58 /dev/ttyS9
```

Device Nodes

Creating device nodes:

- Requires root privileges

```
# mknod /dev/ttyS0 c 4 64
# ls -l /dev/ttyS0
crw-rw---- 1 root uucp 4,64 Mar 6 11:58 /dev/ttyS0
```

Device Nodes

Operations fail without a handler:

- ... even if the device node exists!

`struct file_operations`

Defines handlers for device nodes:

- `open()`, `close()`
- `ioctl()`, ...
- Typically one per each major number

struct file_operations

```
struct file_operations {
  int     (*open   )(struct inode *, ...
  int     (*flush  )(struct file  *, ...
  int     (*release)(struct inode *, ...
  ssize_t (*read   )(struct file  *, ...
  ssize_t (*write  )(struct file  *, ...
  int     (*ioctl  )(struct inode *, ...
  int     (*mmap   )(struct file  *, ...
  ...
};
```

struct miscdevice

"Miscellaneous" devices:

- A single char interface
- Single major, minor number
- Wrapper around struct device
- Device-plus-driver implementations
- Ideal for straightforward situations

struct miscdevice

```
struct miscdevice  {
  int minor;
  const char       *name;
  const struct file_operations *fops;
  struct list_head  list;
  struct device    *parent;
  struct device    *this_device;
};
```

struct miscdevice

int misc_register()

- Registers a `struct miscdevice`
- Allocates a minor number
- Creates a device node (*)

```
#include <linux/miscdevice.h>

int misc_register(struct miscdevice *misc);
```

struct miscdevice

Creates a device node (*):

- Publishes the necessary information
- Tools like *udev* produce the node itself

Minimal

A do-nothing example:

- Just exercises the interface
- “Skeleton” code

`minimal-miscdevice.c`

minimal-miscdevice.c

```c
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>

#define INTERFACE_NAME "m"

struct m {
  struct file_operations fops;
  struct miscdevice miscdevice;
};
```

minimal-miscdevice.c

```
static int
m_open (struct inode *inode, struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}

static int
m_release (struct inode *inode, struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}
```

minimal-miscdevice.c

```c
static ssize_t
m_read (struct file *file,
        char __user *buf,
        size_t count, loff_t *offp)
{
  printk(KERN_INFO "%s: count = %d, returns 0\n",
         __FUNCTION__, count);
  return 0;
}
```

minimal-miscdevice.c

```
static ssize_t
m_write (struct file *file,
         const char __user *buf,
         size_t count, loff_t *offp)
{
  printk(KERN_INFO "%s: count = %d, returns %d\n",
         __FUNCTION__, count, count);
  return count;
}
```

minimal-miscdevice.c

```
static struct m m = {
  .fops = {
    .owner   = THIS_MODULE,
    .read    = m_read,
    .write   = m_write,
    .open    = m_open,
    .release = m_release,
  }
};
```

minimal-miscdevice.c

```c
static int __init m_init (void)
{
  int ret;

  m.miscdevice.minor = MISC_DYNAMIC_MINOR;
  m.miscdevice.name = INTERFACE_NAME;
  m.miscdevice.fops = &m.fops;
  ret = misc_register(&m.miscdevice);

  return ret;
}
```

minimal-miscdevice.c

```
static void __exit m_exit (void)
{
  misc_deregister(&m.miscdevice);
}

module_init(m_init);
module_exit(m_exit);

MODULE_LICENSE("GPL");
```

Constant Read

A more interesting `read()`:

- Returns the contents of a constant text string
- Returns the contents of a constant text string
- Returns the contents of a constant text string
- ...

`miscdevice-const-read.c`

miscdevice-const-read.c

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>

#define INTERFACE_NAME "m-const-read"

struct m {
  struct file_operations fops;
  struct miscdevice miscdevice;
};
```

miscdevice-const-read.c

```
static int
m_open (struct inode *inode,
        struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}

static int
m_release (struct inode *inode,
           struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}
```

miscdevice-const-read.c

```
static ssize_t
m_read (struct file *file,
        char __user *buf,
        size_t count, loff_t *offp)
{
  ssize_t ret = 0;
  static char const *pstuff = stuff;
```

miscdevice-const-read.c

```
  if (!*pstuff) {
    ret = 0;
    pstuff = stuff;
  }
  else if (!(copy_to_user(buf, pstuff++, 1)))
    ret = 1;
  else
    ret = -EFAULT;

  return ret;
}
```

`miscdevice-const-read.c`

The `copy_to_user()` function:

- Safely moves data from kernel to user
- Roughly equivalent to `memcpy()`
- Always check the return value!

Linux implements *protected memory*:

- User, kernel memory can’t see each other directly
- The `buf` pointer cannot be dereferenced!

miscdevice-const-read.c

Conventions for returning from `read()`:

- Number of bytes read
- Negative number to indicate error
- Zero to indicate end-of-data

See `<include/linux/errno.h>` for error codes:

```
return -EAGAIN;
```

miscdevice-const-read.c

```
static ssize_t
m_write (struct file *file,
         const char __user *buf,
         size_t count, loff_t *offp)
{
  printk(KERN_INFO "%s: count = %d, returns %d\n",
         __FUNCTION__, count, count);
  return count;
}
```

miscdevice-const-read.c

```
static struct m m = {
  fops: {
    read    : m_read,
    write   : m_write,
    open    : m_open,
    release : m_release,
  }
};
```

miscdevice-const-read.c

```
static int __init m_init (void)
{
  int ret;

  m.miscdevice.minor = MISC_DYNAMIC_MINOR;
  m.miscdevice.name = INTERFACE_NAME;
  m.miscdevice.fops = &m.fops;
  ret = misc_register(&m.miscdevice);

  return ret;
}
```

miscdevice-const-read.c

```
static void __exit m_exit (void)
{
  misc_deregister(&m.miscdevice);
}

module_init(m_init);
module_exit(m_exit);

MODULE_LICENSE("GPL");
```

miscdevice-const-read.c

Test:

```
# cat /dev/m-const-read
0123456789abcdef

# echo ‘‘foo’’ > /dev/m-const-read
# cat /dev/m-const-read
0123456789abcdef
```

miscdevice-const-read.c

“What if `read()` never returns zero?”

- Answer depends on the calling program
- `cat(1)` will run forever

```
$ head -c 12 /dev/m-const-read | hexcat
```

`miscdevice-const-read.c`

"What if we don't wrap `pstuff`?"

- You start returning kernel memory!

"Isn't returning one byte at a time really inefficient?"

- Yes!

`miscdevice-const-read.c`

“What if the `module_init()` fails?”

- Return an error code
- Linux will “expunge” the module from memory
- (Don’t forget to un-allocate resources!)

Read, Write a Buffer

A more interesting `write()`:

- Stores input to a buffer
- Sends back what you write to it

miscdevice-read-write.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>

#define INTERFACE_NAME "m-read-write"

struct m {
  struct file_operations fops;
  struct miscdevice miscdevice;
};
```

miscdevice-read-write.c

```
static char stuff[32];
static char *pstuff = stuff;
```

miscdevice-read-write.c

```
static ssize_t
m_write (struct file *file,
         const char __user *buf,
         size_t count, loff_t *offp)
{
  if (count >= (sizeof(stuff) - 1))
    return -ENOMEM;
  if (copy_from_user(stuff, buf, count))
    return -EFAULT;

  stuff[count] = 0;
  pstuff = stuff;
  return count;
}
```

miscdevice-read-write.c

```
static ssize_t
m_read (struct file *file,
        char __user *buf,
        size_t count, loff_t *offp)
{
  ssize_t ret = 0;
```

miscdevice-read-write.c

```
  if (!*pstuff) {
    ret = 0;
    pstuff = stuff;
  }
  else if (!(copy_to_user(buf, pstuff++, 1)))
    ret = 1;
  else
    ret = -EFAULT;

  return ret;
}
```

miscdevice-read-write.c

```
static int
m_open (struct inode *inode,
        struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}

static int
m_release (struct inode *inode,
           struct file *pfile)
{
  printk(KERN_INFO "%s: returns 0\n", __FUNCTION__);
  return 0;
}
```

miscdevice-read-write.c

```c
static int __init m_init (void)
{
  int ret;

  strncpy(stuff, "0123456789abcdef\n",
          sizeof(stuff) - 1);

  m.miscdevice.minor = MISC_DYNAMIC_MINOR;
  m.miscdevice.name = INTERFACE_NAME;
  m.miscdevice.fops = &m.fops;
  ret = misc_register(&m.miscdevice);

  return ret;
}
```

miscdevice-read-write.c

```c
static void __exit m_exit (void)
{
  misc_deregister(&m.miscdevice);
}

module_init(m_init);
module_exit(m_exit);

MODULE_LICENSE("GPL");
```

miscdevice-read-write.c

Test:

```
# cat /dev/m-read-write
0123456789abcdef

# echo ‘‘foo’’ > /dev/m-read-write
# cat /dev/m-read-write
foo
```

`miscdevice-read-write.c`

“What if someone is reading while we’re writing?”

- Pay careful attention to `stuff[]` and `pstuff`
- (Can you spot the problem?)

We need *concurrent access protection*:

- A.k.a. “mutual exclusion”, “mutex”

miscdevice-read-write.c

```
  if (!*pstuff) {
    ret = 0;
    pstuff = stuff;
  }
  else if (!(copy_to_user(buf, pstuff++, 1)))
    ret = 1;
  else
    ret = -EFAULT;

  return ret;
}
```

And finally...

“What about the major number?”

- Usually only one `struct file_operations` for each

Structure trickery:

- Changing handlers on-the-fly!
- See `drivers/char/misc.c` for details

And finally...

```
static const struct file_operations
misc_fops = {
  .owner          = THIS_MODULE,
  .open           = misc_open,
};
```

And finally...

```
static int
misc_open(struct inode * inode,
          struct file  * file)
{
  ...
  old_fops = file->f_op;
  file->f_op = new_fops;
  if (file->f_op->open) {
    err=file->f_op->open(inode,file);
    ...
  }
  ...
}
```

Recap

Device driver:

- A thing that controls a device
- Not an interface!

Interface:

- Abstraction for user programs
- Not a device driver!

Recap

`struct miscdevice`

- Simple interface framework
- Uses a `struct file_operations`

Linux “Miscdev” Devices

Simple Character Device Interfaces

Bill Gatliff
`bgat@billgatliff.com`

Freelance Embedded Systems Developer