An introduction to SCSI drivers

Alan Cox

alan@redhat.com

1. Copyright and Licensing

2. An Introduction to SCSI Drivers

For this months article I'm going to attempt to cover writing a driver for a simple SCSI controller under Linux. The Linux kernel does most of the work for SCSI devices so a 'dumb' SCSI interface can actually be relatively painless to write. For more advanced devices however the SCSI layer is actually too clever. There are plans afoot to streamline it because of this.

The job of the SCSI driver is different to that of a block driver. The upper layers of the SCSI code handle the CD-ROM, Disk and other devices. Requests are turned into SCSI command blocks before they are fed to your driver. This means your SCSI driver need only worry about SCSI and not about other aspects of the kernel device structure.

In order to illustrate the SCSI drivers I'm going to invent a SCSI controller that has a simple and easy to use command interface. They sadly don't tend to exist.

It does however make it a lot easier to follow the example. Even so within the limits of a magazine I can only cover the basics and hopefully enough to get people started.

A Linux SCSI driver contains seven main functions

The detect function is called first by the SCSI layer. It has the job of scanning for the controllers and registering those it finds with the SCSI layer. Once they are registered the SCSI layer will issue commands to the controller to probe the SCSI devices on the chain.

The command function issues a command synchronously and waits for it to complete. Most drivers implement this by calling their own queue command function.

The queue command function issues a command and does not wait for it to finish. This is used by almost all operations. When the command completes the driver calls back into the SCSI layer to inform the SCSI layer of the completion and passes back any error information.

Abort and Reset are used to handle error situations or cases where the SCSI layer thinks a command has gone missing. The SCSI layer will first attempt to abort the command then if need be start using a larger hammer on the problem until it gets to the point of trying to reset the entire controller. Hopefully this will never happen.

The info function returns a description of the controller itself. This is generally a very short piece of code indeed.

Finally the mapping of SCSI to PC disk geometry has never been exactly a standard. The bios_param function is called by the SCSI layer to ask the controller to either query its BIOS for the faked disk geometry or to compute a geometry (hopefully using the same algorithm as the controller BIOS itself).

The first function to look at in detail is the probe function. This is called at boot time or when a SCSI module is loaded. For our example we assume there can only be one card and that it behaves sanely.

```
int myscsi_detect(Scsi_Host_Template *tpnt)
```

```
{
struct Scsi_Host *shpnt;
int io=0x320, irq=11; /* Assume fixed for example */
```

For our example we will use a fixed IO and IRQ. A real controller would either read PCI space or would probe a list of common addresses. We will also hide the probe logic in a function.

```
if(myscsi_probe(io, irq)==0)
{
/* Found - create an instance of this controller */
shpnt = scsi_register(tpnt, 0);
if(shpnt==NULL)
return 0;
```

The first thing we do is to ask scsi_register to make us a device. The tpnt is the template passed into this function and whose definition we will describe later in the article. It basically defines this type of card. The returned point is an instance of the card. We pass 0 for the second argument as we need no private data area attaching. Passing a size arranges for a private block to be allocated as shpnt->hostdata

```
shpnt->unique_id = io;
shpnt->io_port = io;
shpnt->n_io_port = MY_PORT_RANGE;
shpnt->irq = irq;
shpnt->this_id = MY_SCSI_ID;
```

Now we start to fill in the structure. The unique_id is for telling cards apart. In our case the I/O port is a convenient choice for this. 'this_id' holds the ID of the controller itself. Each SCSI device has an identity including the controller. The SCSI layer needs to know the controllers ID. We assume for this case it is fixed.

```
my_hardware_init(shpnt);
```

Initialize my hardware. You get to write all of this bit.

```
if(request_irq(irq, my_irq_handler, 0, "myscsi", shpnt))
{
scsi_unregister(shpnt);
printk("my_scsi: IRQ %d is busy.\n", irq);
return 0;
}
```

If we can't register our interrupt handler we are a bit stuck. If so we unregister our scsi controller and report no controllers found. We also let the user know so as to avoid confusion.

```
}
return 1;
}
```

And if it worked we report that we found 1 controller. The SCSI layer will now go off and scan all our devices. Time to write the command functions.

```
int myscsi_queuecommand(Scsi_Cmnd *SCpnt, void (*done)(Scsi_Cmnd *))
{
int io, i;
unsigned long flags;

io = SCpnt->host->io_port; /* Dig out our I/O port */

current_command = Scpnt;
```

For this example we will assume that the controller handles only one command at a time. Typical for a cheap ISA controller, not for decent hardware. If we supported many commands we couldn't keep a global current_command but would need to keep some kind of list and match replies from the card to the list entries.

```
current_command->scsi_done = done;
current_command->SCp.Status = 0;
```

We need to remember what to call when the command completes. Next we set up the command. Our card is hypothetical and rather over-smart for a basic ISA device. You won't be so lucky...

```
save_flags(flags);
outb(SCpnt->target, io+TARGET_PORT);
for(i=0;i<SCpnt->cmd_len;i++)
outb(SCpnt->cmnd[i], io+BUF+i);
outb(COMMAND_BEGIN, io+COMMAND);
```

Firstly we load the target device into the card, then the command. SCSI commands are blocks of up to 16 bytes including length information. After shoving it onto the card we can let the card begin operation, and also we can allow interrupts as we are ready to handle the result of the command.

```
restore_flags(flags);
return 0;
```

and we return back to the SCSI layer, the command is queued and hopefully something will happen. If not then the SCSI layer will bother us after a timeout.

When the SCSI layer does want to bother us about commands that have gone walkies then it will call our abort function then if that fails our reset function. Many simpler controllers cannot support the abort function. If so the abort function is nice and simple

```
int myscsi_abort(Scsi_Cmnd *SCpnt)
{
return SCSI_ABORT_SNOOZE;
}
```

We ask the kernel to wait a bit longer and hope. In the end the kernel will get bored of waiting and call our reset handler. We can also report SCSI_ABORT_PENDING to indicate the command is being aborted but that it has not yet aborted - for example if an interrupt must occur from the card confirming the abort, and we can return SCSI_ABORT_SUCCESS if we aborted the command. Finally we can report SCSI_ABORT_BUSY if we are busy or there is some other reason we would like to abort but cannot do so right now.

After trying to abort and reissue failing commands the SCSI layer will try to reset things. It tries to reset first the device in case that has become confused, then to reset the SCSI bus in case the bus itself has locked up. Finally it tries to reset the controller in case the hardware has choked.

How you handle this depends on the ability of the controller itself.

```
int myscsi_reset(Scsi_Cmnd *SCpnt, unsigned int flags)
{
myhardware_reset(SCpnt->host);
return SCSI_RESET_PENDING;
}
```

For our example we assume that the controller is fairly dumb. We ignore the flag hints and we reset the device. The SCSI_RESET_PENDING return indicates that the bus has been reset but that commands will be returned with a failure status later. If the controller reset returned the commands immediately we could reissue the commands and return SCSI_RESET_SUCCESS. If we do not think this type of reset is appropriate we can return SCSI_RESET_PUNT. You should at least support resetting the bus.

The flags field is a set of four flags designed to provide hints as to what to reset and how. The important flags are SCSI_RESET_SUGGEST_BUS_RESET when the SCSI layer thinks the entire bus should be reset and SCSI_RESET_SUGGEST_HOST_RESET which is the last resort hint to the driver that things are bad and that it might be appropriate to completely restart the board itself.

We've issued commands and we can start an abort. At this point we can't get any further without considering the interrupt handler. The needs of the interrupt handler can vary a lot between cards. For our example driver I'm going to assume that it will interrupt us once when it wants the data to send/receive and once on command completion.

```
int my_irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
struct Scsi_Host *shpnt = dev_id;
int io = shpnt->io_port;
u16 data;
```

When we requested the interrupt we used the host pointer as the 'dev_id' - a device specific field that is passed to the handler by the kernel. This makes it very easy for us to find which card we are handling in a driver that is supporting multiple interface cards. We then dig out our I/O port as we will probably need this a lot in a moment.

```
data = inw(io+READ_STATUS);
if(data&RESET_DONE)
{
current_command->result = DID_RESET«16;
current_command->scsi_done(current_command);
return;
}
```

Firstly we check if the bus has been reset (either by us or other devices). If so we report the command was reset. This will also tell the SCSI layer that the reset we reported as pending in our reset handler has now completed.

```
if(data&PARITY_ERROR)
{
current_command->result = DID_PARITY«16;
current_command->scsi_done(current_command);
return;
}
```

We check for parity errors. We would check for as many errors as we can identify cleanly on a real card. For an error with no exact detail we

```
if(data&GENERAL_ERROR)
{
current_command->result = DID_ERROR «16;
```

```
current_command->scsi_done(current_command);
return;
}
```

The SCSI mid layer will handle doing the right things to recover from an error situation. Next we look to see if this is a SCSI phase change (SCSI commands pass through a set of phases. A smart controller handles all of this a dumb one less. In our case we will assume that the only phases that need help are 'data in' and 'data out' - where we copy bytes to or from the SCSI device we issued a command.

```
if(data&DATA_OUT)
{
outsw(port+DATA_FIFO,
current_command->request_buffer,
current_command->request_bufflen);
}
```

To send data we blast the buffer to the controller. This may well be done by DMA in a real controller. Our example we keep simple. On input we check how many bytes were received and copy them to the request buffer - which is probably a page of disk cache most of the time. We don't have to worry where it goes however, just that it fits.

```
if(data&DATA_IN)
{
int len = inw(port+DATA_LEN);
if(len>current_command->request_bufflen)
len=current_command->request_bufflen;
insw(port+DATA_FIFO, current_command->request_buffer,
current_command->request_bufflen);
}
```

Finally check if a command finished. If so put the device SCSI status in the low byte of the response and tell the SCSI layer the command has completed. The top 16bits hold the kernel info, the bottom the SCSI info. The top 16bits for no error are 0 precisely to make this simple.

```
if(data&COMMAND_DONE)
{
current_command->status = inb(port+CMD_STATUS);
current_command->scsi_done(current_command);
}
}
```

and we exit our interrupt.

SCSI commands can be issued synchronously although this is now basically dead and we do things properly. Supporting the synchronous commands is best done in terms of the queuecommand function and the code below is basically boilerplate used by almost every driver.

```
static void it_finished(Scsi_Cmnd *SCpnt)
{
SCpnt->SCp.Status++;
}

int myscsi_command(Scsi_Cmnd *SCpnt)
{
myscsi_queuecommand(SCpnt, it_finished);
while(!SCpnt->SCp.Status)
barrier();
return SCpnt->result;
}
```

We queue a command and tell the queue function that the 'completion' handler (scsi_done) is to increment the status. Having issued the command we spin in a loop until the command finishes. The barrier() statement is important here. Gcc might otherwise optimize

```
while(variable)
```

to

```
if(variable)
while(1);
```

Barrier tells it that it cannot cache values from variables across the barrier() function call. This ensures that the status, which is changed by an interrupt will be seen by the looping code.

This completes the SCSI command handlers for our simple card. They are not optimized and our card is a little simplistic. We still need to fill in the geometry function and the info function. The info function returns a text description for our controller.

```
const char *myscsi_info(struct Scsi_Host *SChost)
{
return("My SCSI device");
}
```

it could (perhaps should in fact) return the I/O and IRQ information, driver version and other valuable information too.

The bios_param function maps our SCSI disk to a PC BIOS faked geometry. Real disks don't have the simple geometry the PC has, but everyone has carried on faking it rather than fixing all the operating systems. Thus we have to continue this fiction. We need to use the same algorithm as the BIOS or life will be messy.

This example is taken from the Symbios 53c416 driver and is quite typical

```
int sym53c416_bios_param(Disk *disk, kdev_t dev, int *ip)
{
int size;

size = disk->capacity;
ip[0] = 64;                    /* heads       */
ip[1] = 32;                    /* sectors     */
if((ip[2] = size >> 11) > 1024)
    /* cylinders, test for
big disk */
```

```
{
ip[0] = 255;            /* heads      */
ip[1] = 63;             /* sectors    */
ip[2] = size / (255 * 63);  /* cylinders  */
}
return 0;
}
```

Given the disk size we fill in an array of integers for the heads, sectors and cylinders of our disk. We actually want to be sure that these are right. Getting the mapping wrong will give people who use mixed Linux/DOS disks corrupted file systems and generate unhappy mail.

All is now fine except that to unload the module we need to clean up our resources. We provide a release function for this.

```
int myscsi_release(struct Scsi_Host *SChost)
{
free_irq(SChost->irq, SChost);
return 0;
}
```

A real driver should of course have allocated and freed the I/O ports it used too.

To make our driver a SCSI module we have to include some magic at the end of the file

```
#ifdef MODULE

Scsi_Host_Template driver_template = MYSCSI;

#include "scsi_module.c"
#endif
```

This generates the init_module and cleanup_module code needed for a SCSI device, rather than the author having to replicate it each time. The MYSCSI object is a define we need to create in a header file we also include. It is a define in a separate file as for a compiled in driver we will need it again.

Our myscsi.h file looks like

```
extern int myscsi_detect(Scsi_Host_Template *);
extern const char *myscsi_info(struct Scsi_Host *)
...
```

to declare the routines we provide. Then we defined the MYSCSI template

```
#define MYSCSI { \
name: "My SCSI Demo", \
detect: myscsi_detect, \
info: myscsi_info, \
command: myscsi_command, \
queuecommand: myscsi_queuecommand, \
abort: myscsi_abort, \
reset: myscsi_reset, \
bios_param: myscsi_bios_param, \
```

This part defines the SCSI functions we use. The "field: value" format is a gcc extension which sets a given field in a structure rather than listing all the fields in order.

```
can_queue: 1, \
```

To tell the kernel we can queue commands and return

```
this_id: MY_SCSI_ID, \
```

Our host SCSI id

```
sg_tablesize: SG_NONE, \
```

Scatter gather is a very useful extension for performance. For this simple driver we don't support it.

```
cmd_per_lun: 1, \
```

We can have at most one command outstanding per LUN (logical unit).

```
unchecked_isa_dma: 1, \
```

If you set this to zero the kernel will do the hard work of ensuring all the disk buffers are copied into ISA bus accessible memory when needed. This only matters to ISA bus controllers that do DMA.

```
use_clustering: ENABLE_CLUSTERING, \
```

We turn on clustering. Clustering tells the SCSI layer that it is worth trying to merge multiple disk read or write requests into a single SCSI command. A very intelligent controller may well not set this.

```
proc_dir: &myscsi_proc \
}
```

Lastly we define our directory for /proc/scsi. We haven't put this into the driver yet so we add

```
struct proc_dir myscsi_proc =
{
PROC_SCSI_MYSCSI,
"myscsi",
6, /* Length of name */
S_IFDIR|S_IRUGO|S_IXUGO,
2
};
```

which will be used to install our directory in /proc/scsi. The PROC_SCSI_MYSCSI needs to be added to include/linux//proc_fs.h to get a unique inode number for this directory in /proc/scsi. The scsi_directory_inos enumeration is simply a list of all the possible devices. We drop our entry in before the debugging driver

```
PROC_SCSI_FCAL,
PROC_SCSI_I2O,
PROC_SCSI_MYSCSI, /* here */
PROC_SCSI_SCSI_DEBUG,
```

Hopefully this article has provided enough grounding that those interested in writing SCSI drivers can now follow through existing drivers - especially simple ones like the symbios 53c416 driver and see how to implement a new one.