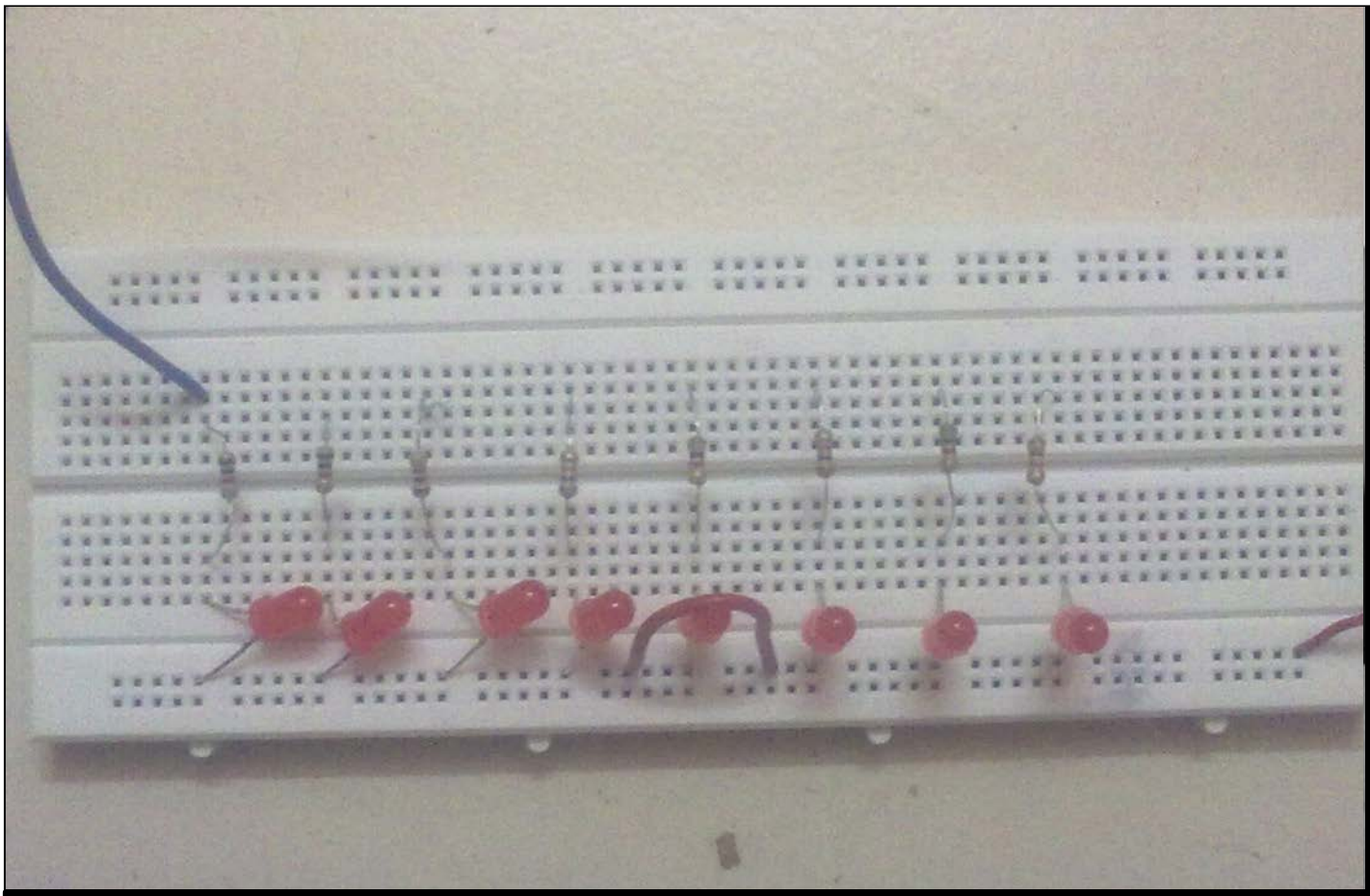


Krishna Y (CS09773)

B Ramesh (CS09775)

A Sachin (CS09776)

Under the guidance of Prof. J. Ravi Kumar

DEVICE DRIVERS

FOR CUSTOM HARDWARE, ON THE LINUX OS

This document discusses the device driver we have written and embedded inside the Linux kernel. It enables the user to control a series of LED's through the D-Sub 25 pin Serial I/O port.

DEVICE DRIVERS

FOR CUSTOM HARDWARE, ON THE LINUX OS (Implemented in C)

INTRODUCTION

The goal of the project was to construct a hardware device of our own and interface it with a personal computer, using our own driver.

The plan was to construct a series of LED's (8 in total), and connect them using the D-Sub 25 pin I/O port (also known as Parallel port).

The hardware was constructed using a breadboard, LEDs and resistors.

The code was written on a Linux computer conforming to POSIX standards. The kernel version was 2.6.

DISCUSSION

Construction

The construction of the hardware device is pretty straightforward.

Take one breadboard, and attach one end of each of the desired number (in our case, 8) of LEDs to the GROUND of the breadboard (0V) and the other end to the HIGH. Attach serially a resistor, in the HIGH columns, to each of the LEDs, as shown in Figure 1.

Why are we attaching a resistor? This is because without attaching the resistor the current would be too high and would burn out the LEDs.

Now add one connecting wire each to the GND and HIGH columns and attach them to 10V of direct current. In our case we will be using two 5V AA batteries.

Now this might seem like a contradiction: Why use two batteries and then reduce the current again with resistors?

Well, the addition of a battery was for the increase in voltage, not current increase.

CHOICE OF LANGUAGE

There are so many languages one can program in today. However, the two languages that came first to mind were C and Assembly, because of their advanced low-level capabilities. We settled on C because in addition to advanced low-level manipulation, it has an amazing high-level style of coding.

Why did we choose Linux, though? We thought that open source is the way to go, and inserting a module is way more straightforward than others, like Windows.

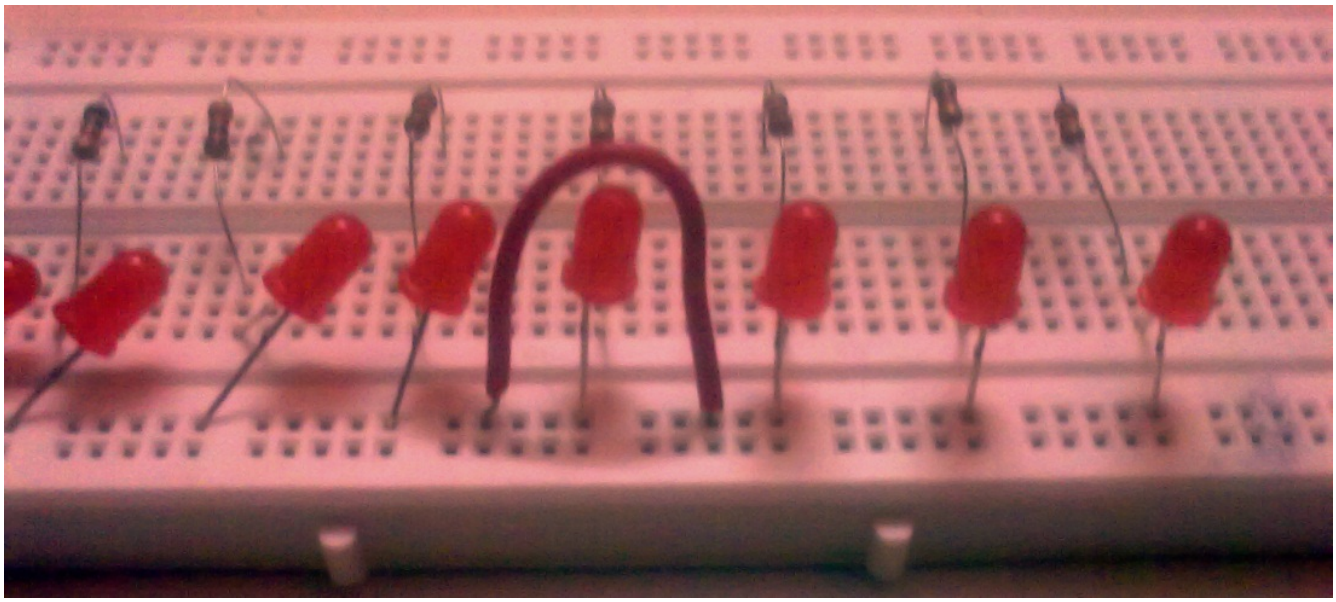


Figure 1. Close-up view of custom hardware

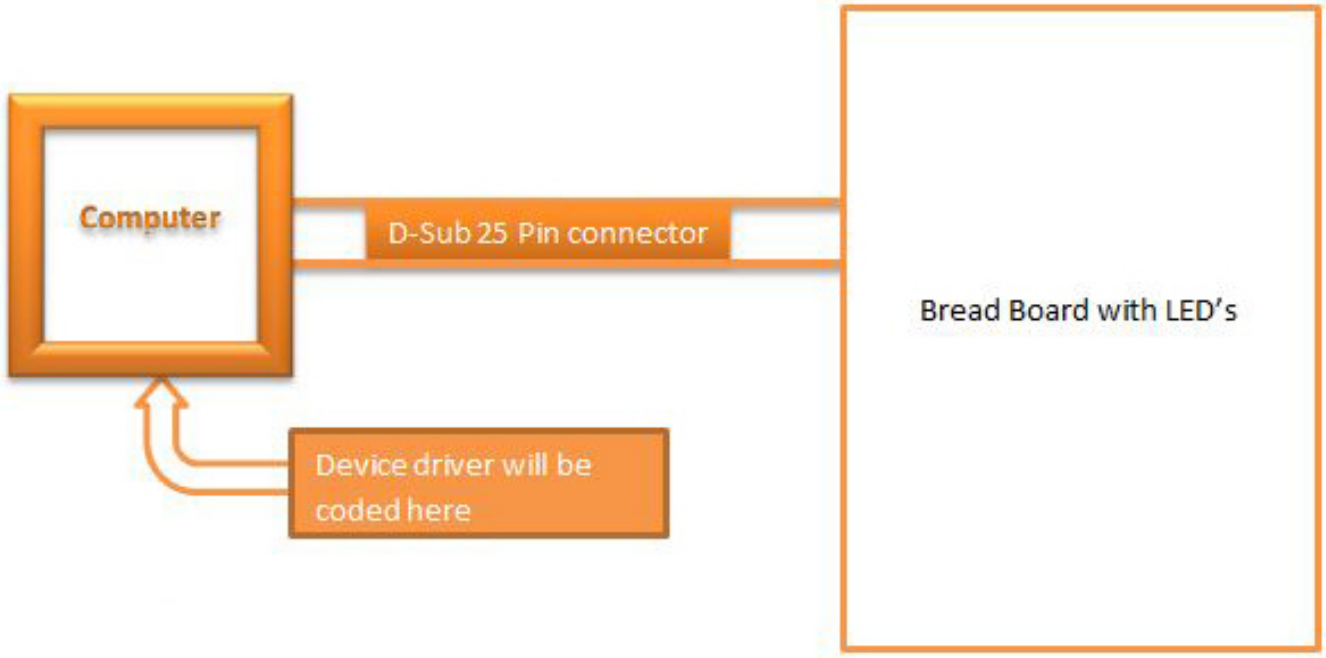


Figure 2. Flowchart depicting communication process

Now comes the difficult part. We have to take a D-Sub 25-pin connector, as shown in figure 3.

We are using ports 2 through 9 to send data, with pin 25 as the ground. Thus we are able to send 8 bits of data (2 to 9), to the corresponding 8 LED lights. Each of these bits carries a 0 or 1, thus having a corresponding OFF or ON state on the lights.

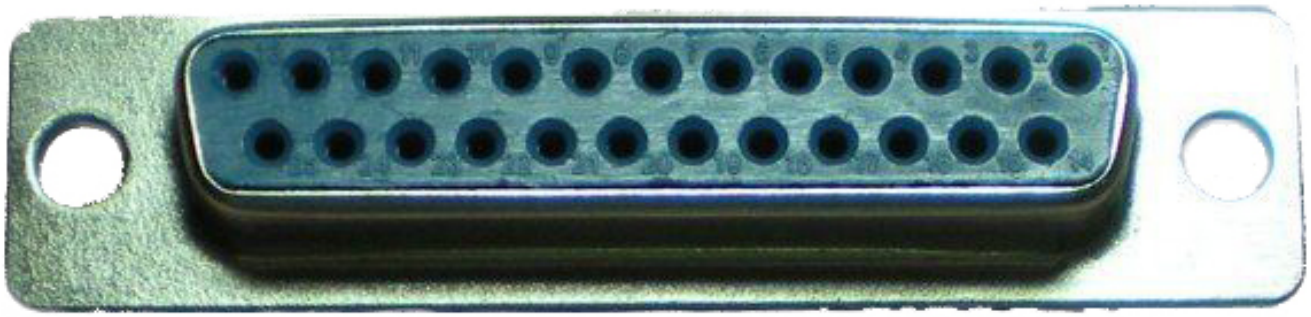


Figure 3. The D-Sub 25 pin connector (Parallel I/O port)

Pin No (DB25)	Pin No (36 pin)	Signal name	Direction	Register - bit	Inverted
1	1	Strobe	In/Out	Control-0	Yes
2	2	Data0	Out	Data-0	No
3	3	Data1	Out	Data-1	No
4	4	Data2	Out	Data-2	No
5	5	Data3	Out	Data-3	No
6	6	Data4	Out	Data-4	No
7	7	Data5	Out	Data-5	No
8	8	Data6	Out	Data-6	No
9	9	Data7	Out	Data-7	No
10	10	Ack	In	Status-6	No
11	11	Busy	In	Status-7	Yes
12	12	Paper-Out	In	Status-5	No

13	13	Select	In	Status-4	No
14	14	Linefeed	In/Out	Control-1	Yes
15	32	Error	In	Status-3	No
16	31	Reset	In/Out	Control-2	No
17	36	Select-Printer	In/Out	Control-3	Yes
18-25	19-30,33,17,16	Ground	-	-	-

Table 1. The D-Sub 25 pin connector pinouts [1]

As we can see, we are only bothered with bits 2-9 (the Data bits). As they are not inverted, 1 means ON and 0 means OFF.

CODING

Coding was a large undertaking in this project. The output is seen in the form of two files: One will be the module that will be inserted into the kernel; the other is the program that allows the user interaction with the LED displays.

(Both programs are attached at the end of this document for your viewing.)

SOURCE FILE 1: PARLELPORT.C

This is the file that inserts the driver into the kernel; it is composed of mainly these functions:

- Load module
- Open device
- Read device
- Write device
- Close device
- Remove module

Before we go through some important parts of the source code, it is necessary to note the following points:

- The major number of the parallel port is 61. What is a major number? Major and minor numbers are associated with the device special files in the */dev* directory and are used by the operating system to determine the actual driver and device to be accessed by the user-level request for the special device file. [2]
- The address locations are from `0x378` to `0x37f`, `0x278` to `0x27f` and `0x3bc` to `0x3bf`. We will be using the location `0x378` to `0x37f`.
- The port cannot be read from or written to when another application is using it. Thus we have to check the port before using it.

FUNCTION LIST (In Order of Appearance)

(A) USER-DEFINED FUNCTIONS

1. **`int parlelport_open(struct inode*, struct file*)`**

   This function opens the parallel port for our application to use. While initially it may seem odd that this function and `parlelport_release()` are empty and not doing anything, that is just because of the scale of this project. Generally, these two functions are responsible for initializing a device before an operation, and freeing it after. Consider a printer, which would probably send its ink levels before any job is started. Since we don't have any such parts here, we are skipping it.

ARGUMENTS

Argument	Description
`struct inode*`	`inode*` stores the major and minor number that is sent to the kernel
`struct file*`	`file*` contains information relative to the operations that can be performed on a file.

Return type: `int`

Returns: `0` always

2. `int parlelport_release(struct inode*, struct file*)`

This function opens the parallel port for our application to use. While initially it may seem odd that this function and `parlelport_open()` are empty and not doing anything, that is just because of the scale of this project. Generally, these two functions are responsible for initializing a device before an operation, and freeing it after. Consider a printer, which would probably send its ink levels before any job is started. Since we don't have any such parts here, we are skipping it.

ARGUMENTS

Argument	Description
`struct inode*`	`inode*` stores the major and minor number that is sent to the kernel
`struct file*`	`file*` contains information relative to the operations that can be performed on a file.

Return type: `int`

Returns: `0` always

3. `ssize_t parlelport_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)`

This function reads data from the parallel port.

ARGUMENTS

Argument	Description
`struct file *filp`	`*filp` specifies the file to be read from, in this case PARLELPORT, our parallel port device.
`char *buf`	`buf` specifies where this information should be stored.
`size_t count`	`count` specifies the number of bytes to be read.
`loff_t *f_pos`	`*f_pos` specifies the offset from which the reading should occur.

Return type: `ssize_t`

Returns: `1` if location contains 0

`0` on all other cases.

4. `ssize_t parlelport_write(struct file *filp, char *buf, size_t count, loff_t *f_pos)`

   This function writes data into the port.

ARGUMENTS

Argument	Description
`struct file *filp`	`*filp` specifies the file to be written to, in this case `PARLELPORT`, the parallel port.
`char *buf`	`buf` is where the information to be written should be retrieved from.
`size_t count`	`count` specifies the number of bytes to be written.
`loff_t *f_pos`	`*f_pos` specifies the offset from which the writing should occur.

Return type: `ssize_t`

Returns: `1` always

5. `void parlelport_exit(void)`

   This function first frees (unregisters) the major number and then frees the port. This function is called when the module exits.

ARGUMENTS

None.

Return type: `void`

6. `int parlelport_init(void)`

   This function registers the major number and the port number. This function is called when the module initializes.

ARGUMENTS

None.

Return type: `int`

Returns: `errno` if unable to allocate the major number or port

`0` on success.

(B) BUILT-IN FUNCTIONS

1. **`MODULE_LICENSE(char*)`**
   Accepts the license for the module that we are entering into the kernel. We chose a Dual BSD/GPL license.

2. **`MODULE_AUTHOR(char*)`**
   Accepts the name(s) of the author(s) of the module.

3. **`MODULE_DESC(char*)`**
   Accepts the user-define description of what the module does.

4. **`MODULE_SUPPORTED_DEVICE(char*)`**
   This allows for auto-configuration of the device in some cases.

5. **`module_init(`*`init_function`*`)`**
   This calls *init_function* as the initialization routine for the module.

6. **`module_exit(`*`exit_function`*`)`**
   This calls *exit_function* as the end routine for the module.

7. **`int register_chrdev(unsigned int major, const char* name, const struct file_operations* fops)`**
   This function registers a major number for character devices.

ARGUMENTS

Argument	Description
`unsigned int major`	**`major`** specifies the major number. In our case it is **`61`**.
`const char* name`	**`name`** specifies the name of this range of devices. In our case, it is **`parlelport`**.
`struct file_operations* fops`	**`file_operations`** is a structure that keeps track of the operations that can be performed on the device.

Return type: **`int`**

Returns: **`errno`** if unable to allocate the major number or port

`0` on success.

8. `int check_region(unsigned long start, unsigned long len)`

This function is called to check whether a range of ports is available for allocation.

ARGUMENTS

Argument	Description
`unsigned long start`	**`start`** specifies the starting address of where you want to output your data (as mentioned, we will take **`0x378`**).
`unsigned long len`	**`len`** specifies what the size of the region you want is.

Return type: `int`

Returns: `errno` if unable to allocate the major number or port

Non-negative integer on success.

9. `struct resource *request_region(unsigned long start, unsigned long len, char *name)`

This function is called to actually allocate these port ranges.

ARGUMENTS

Argument	Description
`unsigned long start`	**`start`** specifies the starting address of where you want to output your data (as mentioned, we will take **`0x378`**).
`unsigned long len`	**`len`** specifies what the size of the region you want is.
`char *name`	**`name`** specifies the name of this range of devices. In our case, it is **`parlelport`**.

Return type: `resource*`

Returns: `NULL` if unable to allocate the port

Non-`NULL` pointer on success.

10. `void unregister_chrdev(unsigned int major, const char* name)`

This function unregisters a major number for character devices.

ARGUMENTS

Argument	Description
`unsigned int major`	**`major`** specifies the major number. In our case it is **`61`**.
`const char* name`	**`name`** specifies the name of this range of devices. In our case, it is **`parlelport`**.

Return type: **`void`**

11. `void release_region(unsigned long start, unsigned long len)`

ARGUMENTS

Argument	Description
`unsigned long start`	**`start`** specifies the starting address of where you want to output your data (as mentioned, we will take **`0x378`**).
`unsigned long len`	**`len`** specifies what the size of the region you want is.

Return type: **`void`**

12. `char inb(unsigned long addr)`

The **`inb`** function reads a byte from and 8-bit I/O port.

ARGUMENTS

Argument	Description
`unsigned long addr`	**`addr`** specifies the address of the parallel port.

Return type: **`char`**

Returns: The content of the port.

13. `unsigned long copy_to_user(void __user* to, const void* from, unsigned long n)`

The **`copy_to_user`** function transfers the device read data from the kernel into the user space.

ARGUMENTS

Argument	Description
`void __user* to`	Destination address, in user space
`const void* from`	Source address, in kernel space.
`unsigned long n`	Number of bytes to copy.

Return type: `unsigned long`

Returns: Number of bytes that could not be copied, if copy was not successful

`0`, if copy was successful.

14. `int outb(char c, unsigned long addr)`

The `outb` function writes a byte to an 8-bit I/O port.

ARGUMENTS

Argument	Description
`char c`	`c` contains the data to be written into the port.
`unsigned long addr`	`addr` specifies the address of the parallel port.

Return type: `void`

15. `unsigned long copy_from_user(void* to, const void __user* from, unsigned long n)`

The `copy_from_user` function transfers the data to be written from the user space to the kernel.

ARGUMENTS

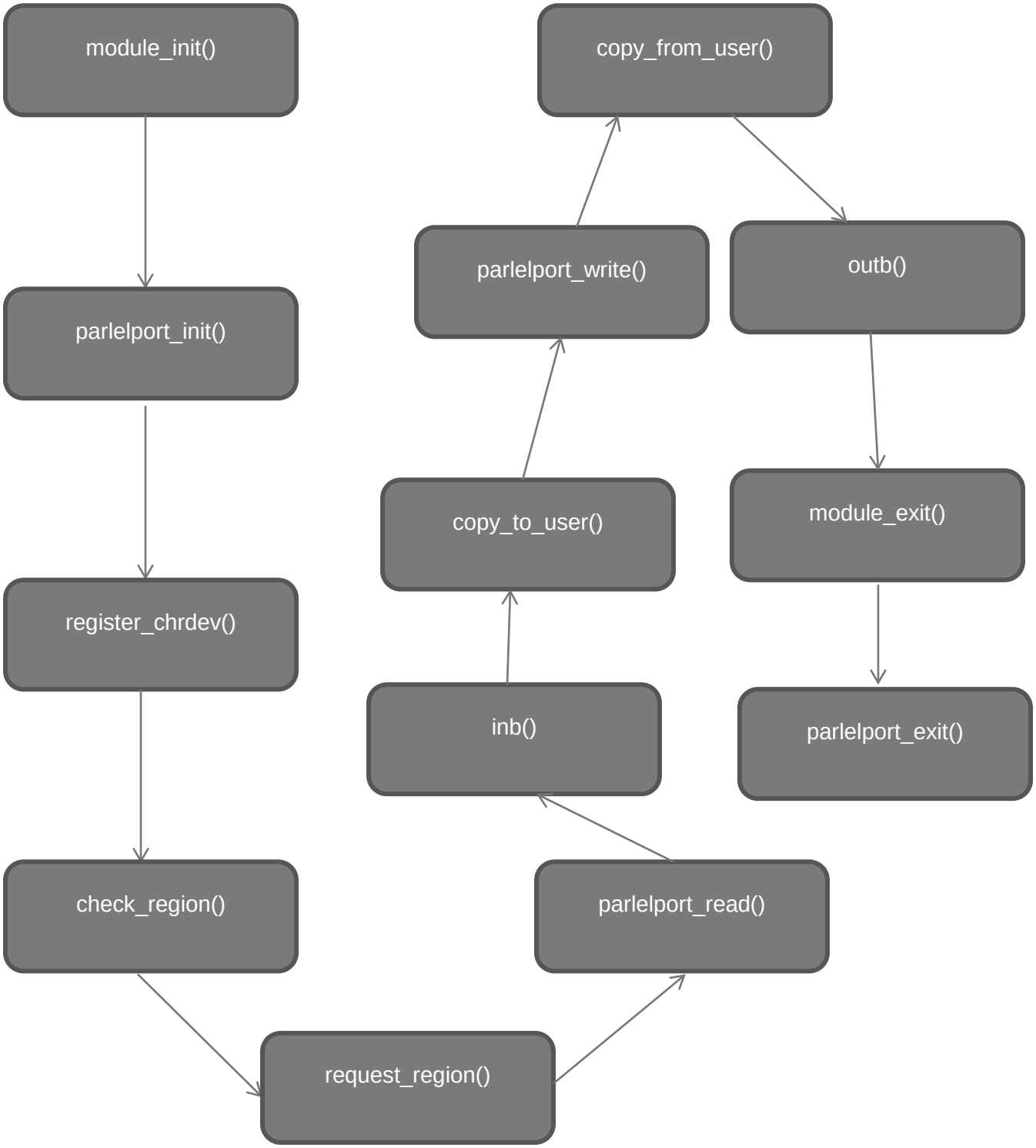
Argument	Description
`void* to`	Destination address, in kernel space.
`const void __user* from`	Source address, in user space.
`unsigned long n`	Number of bytes to copy.

Return type: `unsigned long`

Returns: Number of bytes that could not be copied, if copy was not successful

`0`, if copy was successful.

FLOWCHART



THAT COMPLETES OUR STUDY OF THE FIRST SOURCE FILE, WHICH WILL BE THE MODULE LOADED INTO THE KERNEL.

SOURCE FILE 2: LIGHTS_UPGRADED.C

If you look at the project from the view of the developer, our job is actually done. We needed to provide a basic framework for the port to be controlled, and we did. Now it is time for the user/programmer to write his own code and do whatever he wants with the port.

However, we have to do three things

(1) Test to see if our module works as intended
(2) Set out some sample code for others so that they understand how to implement this and
(3) The final aim of the project was to control the LED lights.

So we wrote some more code in C, to control the LED lights.

It is important to understand that these two programs are completely separate and independent of each other.

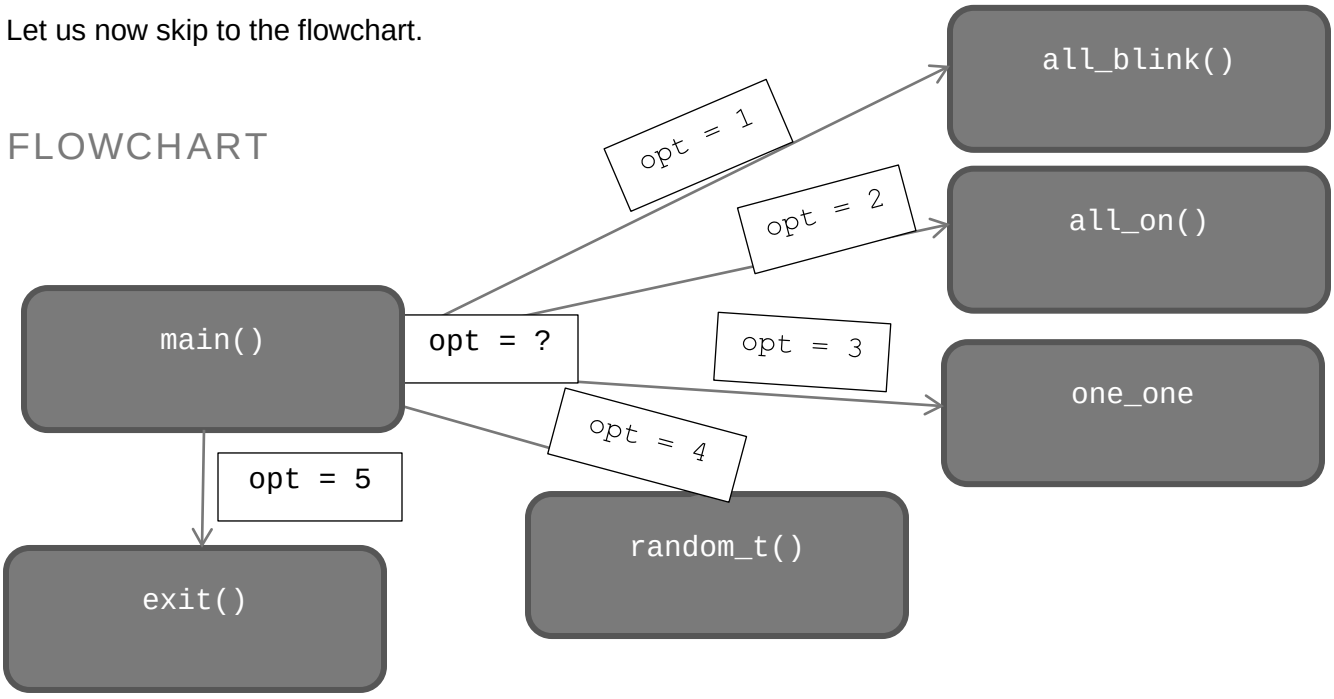
The first program, **PARLELPORT.C** was only the driver that was loaded into the kernel. By itself, it doesn't do anything. Imagine an audio driver. It is there, but you are not playing any music. This second program is like the Media Player that plays the music.

We thought that this second file was pretty self-explanatory and did not feel the need to laboriously document it. Hence, here is a brief description of what the program does:

- As soon as the program loads `all_blink()` is called, and all the lights start blinking. This is a separate thread, so it is not blocking the main menu.
- The main menu loads and certain blinking options are presented to the user. The user selects one of them (or exit) and the program quits.
- The beauty of this program is that it is programmed using threads. Hence, the lights can keep blinking and the user can stop, pause, control whenever he wants, without anything getting blocked.

Let us now skip to the flowchart.

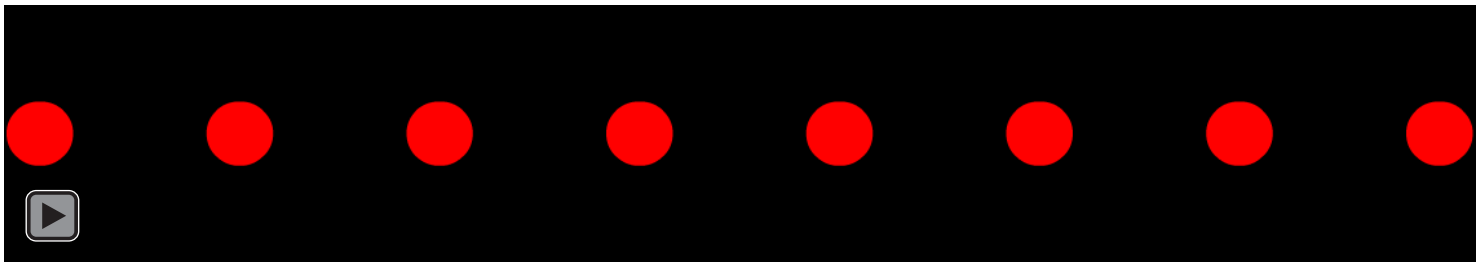
FLOWCHART



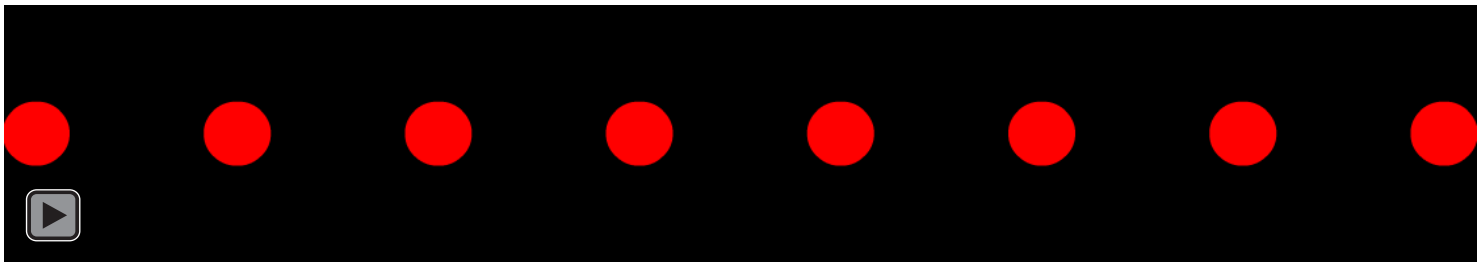
DEMONSTRATION

When the entire setup is working perfectly, it should look like this (click on the **Play** button to start playback, right click and select **Disable Content** to stop):

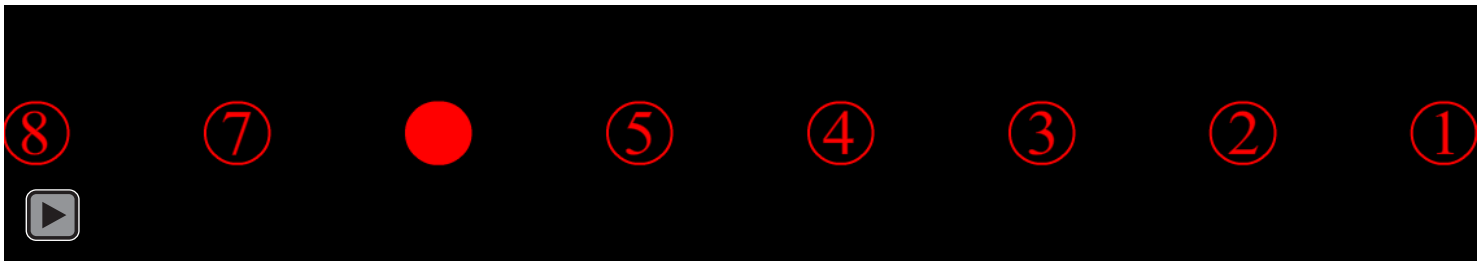
All lights blinking (`opt = 1`, function `all_blink()`)



All lights on (`opt = 2`, function `all_on()`)



One by one (`opt = 3`, function `one_one()`)



BUILDING THE PROJECT AND DEPLOYING IT

It is important to note a few compilation/build points and the shell journey after that:

THE MAKEFILE

You need to have something called a makefile. What is a makefile? It is simply a utility that directs the compiler on how to build the executable program.

To avoid repetitive tasks, we try to automate the build process by creating the makefile. Our makefile is called `Makefile` and has the following contents:

```
obj-m := parlelport.o

KDIR  := /lib/modules/2.6.28-11-generic/build

PWD   := $(shell pwd)

default:

	$(MAKE) -C $(KDIR) M=$(PWD) modules
```

Variable	Description
obj-m	The object file
KDIR	The kernel directory
PWD	The present working directory
$(shell pwd)	This sends the command **pwd** to the shell and stores the path that is returned.
default	The default action to execute when the **Makefile** is called.

INSTALLING THE MODULE

We have to first check to see if there is any driver installed. If there is, we have to remove that first before trying to install our driver.

Step 1: Check

We list all installed modules:

```
$lsmod
```

Step 2: Remove

If there is a conflict, we remove the conflicting module:

```
$rmmod conflicting_module(s)
```

Step 3: Install

Next, we install our module:

```
$insmod parlelport.ko
```

Step 4: Verify

We again list all modules to verify if our driver has been installed successfully.

```
$lsmod
```

Step 5: Link major number

Now, we need to link the major number to the module. You may be wondering, “Didn’t we already specify that in the program?” Yes, but we need to link it so that the operating system knows this.

Take, for example, when we run the `lsmod` command. We need the operating system to have our module listed there. If we rely solely on the source code, yes, we will be able to run our program, but it may cause a huge mess because other modules didn't know your module was using that port. So:

```
$mknod /dev/parlelport c 61 0
```

Recall that `61` was the major number. `0` is the minor number, something we don't need to bother about for a project of this scale.

Step 6: Access permissions

Now, the whole point of writing a device driver is so that users can read from, and write to, the device. Hence, we need to allow users to access the port (interpreted as the file `/dev/parlelport`). So, we change the permissions:

```
$chmod 666 /dev/parlelport
```

Step 7: Running the programs

Now, the installation/loading process is complete. All we have to do now is to run a program in the user space. Make sure that the device is connected to your computer before turning it on, and then run the user program.

Command	Description
`lsmod`	This command shows which kernel modules are currently loaded (reads from `/proc/modules`).
`rmmod`	This command removes a module from the kernel.
`insmod`	This command inserts a module into the kernel.
`mknod`	Make special files. The c stands for character.
`chmod`	Change file access permissions. `666` means read and write permissions for all (user, group and others).

EXTRA NOTES

- To compile source file 1, (**PARLELPORT.C**), you will need an image of the kernel that you want the driver installed into.
- To compile source file 2, (**LIGHTS_UPGRADED.C**), you need to specify the `–pthread` flag as the program uses POSIX threads.

PRECAUTIONS

It is important to earth your computer properly. In case it is not, it could cause damage to the port as well as to the casing.

WORKS CITED

[1] "Parallel Port," December 2011. [Online]. Available: http://en.wikipedia.org/wiki/Parallel_port#Pinouts.

[2] "Major and minor numbers," September 2004. [Online]. Available: http://uw714doc.sco.com/en/HDK_concepts/ddT_majmin.html.

BIBLIOGRAPHY

[1] Xavier Calbet, "Writing device drivers in Linux: A brief tutorial", Free Software Magazine.

[2] W. Richard Stevens & Stephen A. Rago, "Advanced Programming in the UNIX® Environment", Second Edition, Pearson: 2005.

```
1  obj-m :=parlelport.o
2  KDIR  := /lib/modules/2.6.28-11-generic/build
3  PWD   := $(shell pwd)
4  default:
5      $(MAKE) -C $(KDIR) M=$(PWD) modules
6
```

```c
#include <linux/init.h>         // Initialisation of a module
#include <linux/module.h>       // Loading a module in kernel
#include <linux/kernel.h>       // printk()
#include <linux/slab.h>         // kmalloc()
#include <linux/fs.h>           // File table, structues, etc
#include <linux/errno.h>        // Error code
#include <linux/types.h>        // size_t
#include <linux/proc_fs.h>      // Virtual file system, as everything including ports are treated as files
#include <linux/fcntl.h>        // O_ACCMODE
#include <linux/ioport.h>       // Input and output operations on ports
#include <asm/system.h>         // cli(), *_flags
#include <asm/uaccess.h>        // copy_from/to_user
#include <asm/io.h>             // inb, outb

#define DRIVER_AUTHOR "Krishna Y, B Ramesh, A Sachin"
#define DRIVER_DESC   "Flashing LED Lights"
#define DRIVER_LICENSE "Dual BSD/GPL"

MODULE_LICENSE(DRIVER_LICENSE);
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_SUPPORTED_DEVICE("parlelport");

// Function declaration of parlelport.c
int parlelport_open(struct inode *inode, struct file *filp);
int parlelport_release(struct inode *inode, struct file *filp);

ssize_t parlelport_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t parlelport_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);

void parlelport_exit(void);
int parlelport_init(void);

// Structure that declares the common file access functions
struct file_operations parlelport_fops = {
                                            read: parlelport_read,
                                            write: parlelport_write,
                                            open: parlelport_open,
                                            release: pa rlelport_release
                                          };
```

```
41
42
43  int parlelport_major = 61;              // Driver global variables Major number
44
45
46  int port;                               // Control variable for memory reservation of the parallel port
47  module_init(parlelport_init);
48  module_exit(parlelport_exit);
49
50  int parlelport_init(void)
51  {
52         int result;
53
54         result = register_chrdev(parlelport_major, "parlelport", &parlelport_fops);  // Registering device
55
56         if (result < 0)
57         {
58               printk("<1>parlelport: cannot obtain major number %d\n", parlelport_major);
59               return result;
60         }
61
62        port = check_region(0x378, 1);                // Registering port
63
64        if (port)
65        {
66              printk("<1>parlelport: cannot reserve 0x378\n");
67              result = port;
68              goto fail;
69        }
70
71        request_region(0x378, 1, "parlelport");
72
73        printk("<1>Inserting parlelport module\n");
74        return 0;
75
76        fail:
77           parlelport_exit();
78           return result;
79
80  }
```

```
81
82
83  void parlelport_exit(void)
84  {
85         unregister_chrdev(parlelport_major, "parlelport");       // Free major number
86
87        if (!port)                                                // Free port
88        {
89              release_region(0x378,1);
90        }
91
92         printk("<1>Removing parlelport module\n");
93
94  }
95  int parlelport_open(struct inode *inode, struct file *filp)
96  {
97     // Open port: Success as there is nothing to initialize
98     return 0;
99
100 }
101
102 int parlelport_release(struct inode *inode, struct file *filp)
103 {
104
105    // Release port: Success as there is nothing to free
106    return 0;
107
108 }
109
110 ssize_t parlelport_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
111 {
112
113        char parlelport_buffer;                    // Buffer to read the device
114
115        parlelport_buffer = inb(0x378);
116
117        copy_to_user(buf, &parlelport_buffer, 1);    // We transfer data to user space
118
119        /* We change the reading position as best suits */
120
```

```
121         if (*f_pos == 0)
122          {
123             *f_pos += 1;
124
125             return 1;
126
127          }
128
129          else
130          {
131             return 0;
132          }
133
134   }
135
136   ssize_t parlelport_write(struct file *filp, char *buf, size_t count, loff_t *f_pos)
137   {
138           char *tmp;
139
140           char parlelport_buffer;                 // Buffer writing to the device
141
142           tmp = buf + count-1;
143
144           copy_from_user(&parlelport_buffer, tmp, 1);
145
146           outb(parlelport_buffer,0x378);          // Writing to the port
147
148           return 1;
149
150   }
151
152
```

```
1   #include<stdio.h>
2   #include<unistd.h>
3   #include<pthread.h>
4   #include<stdlib.h>
5   #include<time.h>
6
7   #define TRUE 1
8   #define FALSE 0
9
10  int QUIT = FALSE;
11
12  pthread_mutex_t INUSE;
13
14  unsigned char byte;
15
16  FILE* PARLELPORT;
17
18  void *all_blink(void* arg)
19  {
20
21      QUIT = FALSE;
22
23      pthread_mutex_lock(&INUSE);
24
25      while (1)
26
27      {
28
29          byte = 0xFF;
30
31          fwrite(&byte, 1, 1, PARLELPORT);
32
33          sleep(0.5);
34
35          byte = 0;
36
37          fwrite(&byte, 1, 1, PARLELPORT);
38
39          sleep(0.5);
40
```

```
41          if (QUIT == TRUE)
42
43          {
44
45              pthread_mutex_unlock(&INUSE);
46
47              pthread_exit(NULL);
48
49          }
50
51      }
52
53  }
54
55  void *all_on(void* arg)
56  {
57
58      QUIT = FALSE;
59
60      pthread_mutex_lock(&INUSE);
61
62      byte = 0xFF;
63
64      fwrite(&byte, 1, 1, PARLELPORT);
65
66      while (1)
67
68      {
69
70          if (QUIT == TRUE)
71
72              break;
73
74      }
75
76
77      byte = 0;
78
79      fwrite(&byte, 1, 1, PARLELPORT);
80
```

```
81        pthread_mutex_unlock(&INUSE);
82
83        pthread_exit(NULL);
84
85    }
86
87    void *one_one(void* arg)
88    {
89
90        QUIT = FALSE;
91
92        pthread_mutex_lock(&INUSE);
93
94        byte = 1;
95
96        while (1)
97
98        {
99            fwrite(&byte, 1, 1, PARLELPORT);
100
101           byte <<= 1;
102
103           if (QUIT == TRUE)
104
105               break;
106
107           if (byte == 0)
108
109               byte = 1;
110
111           sleep(0.5);
112
113       }
114
115
116       byte = 0;
117
118       fwrite(&byte, 1, 1, PARLELPORT);
119
120       pthread_mutex_unlock(&INUSE);
```

```
121
122         pthread_exit(NULL);
123
124     }
125
126     void *random_t(void* arg)
127     {
128
129         QUIT = FALSE;
130
131         pthread_mutex_lock(&INUSE);
132
133         time_t seconds;
134
135         seconds = time(NULL);
136
137         srand((unsigned)seconds);
138
139         while(1)
140
141         {
142
143             byte = (char)(rand() % 256);
144
145             fwrite(&byte, 1, 1, PARLELPORT);
146
147             sleep(0.5);
148
149             if (QUIT == TRUE)
150
151                 break;
152
153         }
154
155         byte = 0;
156
157         fwrite(&byte, 1, 1, PARLELPORT);
158
159         pthread_mutex_unlock(&INUSE);
160
```

```
161         pthread_exit(NULL);
162
163     }
164
165     int main()
166     {
167
168         char dummy;
169
170         int opt;
171
172         PARLELPORT = fopen("/dev/parlelport", "w");
173
174         setvbuf(PARLELPORT, &dummy, _IONBF, 1);
175
176         pthread_t start, all_lights_on, all_lights_blinking, one_by_one, random;
177
178         pthread_mutex_init(&INUSE, NULL);
179
180         pthread_create(&start, NULL, all_blink, NULL);
181
182         while (1)
183
184         {
185
186             printf("Welcome to our LED show! Please select an option:\n");
187
188             printf("1. All lights on\n2. All lights blinking\n3. One by one\n4. Random\n5. Exit\n");
189
190             scanf("%d", &opt);
191
192             if ((opt > 5) || (opt < 1))
193
194             {
195
196                 printf("Sorry! That is outside the range. Please try again:\n");
197
198                 sleep(0.5);
199
200                 break;
```

```
201
202             }
203
204             QUIT = TRUE;
205
206             switch(opt)
207
208             {
209
210                 case 1: pthread_create(&start, NULL, all_blink, NULL);
211
212                         break;
213
214                 case 2: pthread_create(&all_lights_on, NULL, all_on, NULL);
215
216                         break;
217
218                 case 3: pthread_create(&one_by_one, NULL, one_one, NULL);
219
220                         break;
221
222                 case 4: pthread_create(&random, NULL, random_t, NULL);
223
224                         break;
225
226                 case 5: fclose(PARLELPORT);
227
228                         exit(0);
229
230             }
231
232         }
233
234         return 0;
235
236     }
237
```