

DEVICE DRIVER USING C PROGRAM UNDER LINUX ©

VASUTHEVAN MAHESWARAN

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Engineering, Electrical Engineering

Department of Electrical Engineering
Ryerson Polytechnic University
Toronto, Ontario

May, 2000

Faculty Supervisor: Professor P.D.Hiscocks

Certification of Authorship

I Vasuthevan Maheswaran, declare that the sole authors of this work, and that no plagiarism of any other's work has taken place. Where other's work has been used, it is credited and documented.

Vasuthevan Maheswaran

Acknowledgment

My sincere thanks for the few people, who helped me technically, and directed me to accomplish the task of completing the project successfully and within the stipulated time.

First, I would like to thank Professor P. D. Hiscocks for providing with an opportunity to work with a Linux kernel level device driver, which utilizes the knowledge I learned throughout my four year University career. I would never have been able to finish the project, without the technical insight details and design coordination, which helped me to secure a job as a device driver developer.

Finally, I am delighted to thank my dad, brother and the rest of my family and friends; G Ruban, Ryan and Raj Mohan who helped me to finish the project.

Abstract

This thesis project involves the design and development of a parallel port device driver under Linux operating system platform. The most secure approach in implementing device drivers under Linux, includes user space application and kernel module. This device driver works only in the kernel version 2.2.0 and above, with the special hardware design for the project. This device driver program requires a personal computer running Linux operating system and DB-25 parallel port. The theory portion of the project of this report provides an overview of various technical considerations relevant to this project.

Contents

1	Introduction	8
2	Theory	10
2.1	What is a device driver?	10
2.2	What is major and minor number?	11
2.3	What is kernel?	11
2.4	What is loadable kernel module?	12
2.5	Parallel port communication	12
3	Implementation	14
4	Software Design	16
4.1	User Space Programming	16
4.1.1	Definition of System Calls	16
4.1.2	The <i>Open</i> System Calls	17
4.1.3	The <i>Write</i> System Calls	17
4.1.4	The <i>Read</i> System Calls	18
4.1.5	The <i>Close</i> System Calls	18
4.2	Kernel Module	18
4.2.1	The <i>init_module</i> System Calls	19
4.2.2	The <i>cleanup_module</i> System Calls	21

4.2.3	The <i>ParallelPort_open</i> System Calls	22
4.2.4	The <i>ParallelPort_close</i> System Calls	22
4.2.5	The <i>ParallelPort_write</i> System Calls	22
4.2.6	The <i>ParallelPort_read</i> System Calls	23
4.3	Setting up the System	25
4.3.1	Compiling and Linking the kernel module	25
4.3.2	Installing the kernel module	26
4.3.3	Register the character device	26
4.3.4	Removing the kernel module	27
4.3.5	Unregister the character device	27
5	Hardware Design	28
5.1	Purpose	28
5.2	Design Specification	28
5.2.1	Power Operational amplifier	28
5.2.2	General Purpose operational amplifier	31
5.2.3	Digital to analog converter DAC0808	31
5.2.4	Analog to digital converter ADC0804	31
5.2.5	Flip-Flop 74LS374	32
5.2.6	Octal Tri-STATE Buffer	32
5.2.7	The layout of the Hardware Design	32
5.2.8	The Tcl/Tk widget	32
A	User space C code	38
B	The C code for the ppt Hardware Device	41
C	The Tcl/Tk code for the GUI	48

D The cost for the Design project	51
E Dats Sheets	52

List of Figures

1.1	Block Diagram for the Design Project	9
2.1	Relationship between device driver and the Linux operating system	11
2.2	How the Linux kernel implements a device	12
2.3	25-way Female D-Type Connector	13
4.1	Parallel Port I/O Pin Configuration	19
4.2	The flowchart of the <code>init_module</code> System Calls	20
4.3	The flowchart of the <code>close_module</code> System Calls	21
5.1	The schematic of the Hardware design	29
5.2	The designed power supply	30
5.3	The schematic of the power operational amplifier [L165]	30
5.4	The schematic of the operational amplifier [LM741]	31
5.5	Timeing diagram of the conversion of the ADC0804	32
5.6	The Hardware layout	33
5.7	The GUI for control the DC motor and Generator	33

Chapter 1

Introduction

IBM introduced the parallel port in 1981 to replace the slower serial port [7]. In parallel port, one byte of data is transmitted in one clock cycle, which is faster than single bit data transmission through the serial port. The parallel port is normally used for printers, tape drives, external CD writers, etc. The purpose of this Design Project is to enable the parallel port, under the Linux platform, to control general purpose electronic equipment attached to it. To achieve this, a hardware device driver in C language was developed and installed. An application program which was also written in C would communicate in bi-direction to the parallel port through the hardware device driver. The PC which is running on Linux Operating system is selected for this project, because this Operating System is currently in demand in industry. To demonstrate the bi-directional communication to the parallel port a hardware circuit with a small dc motor mechanically coupled with a dc generator was used. A Graphical User Interface (GUI) developed using Tcl/Tk is used to control entire operation. Through the GUI the voltage to the dc motor can be changed and the corresponding voltage generated by the dc generator is fed back to the parallel port and it is displayed in the GUI. By this way both input and output to the parallel port would be demonstrated. The block diagram of the design is illustrated in Figure1.1.

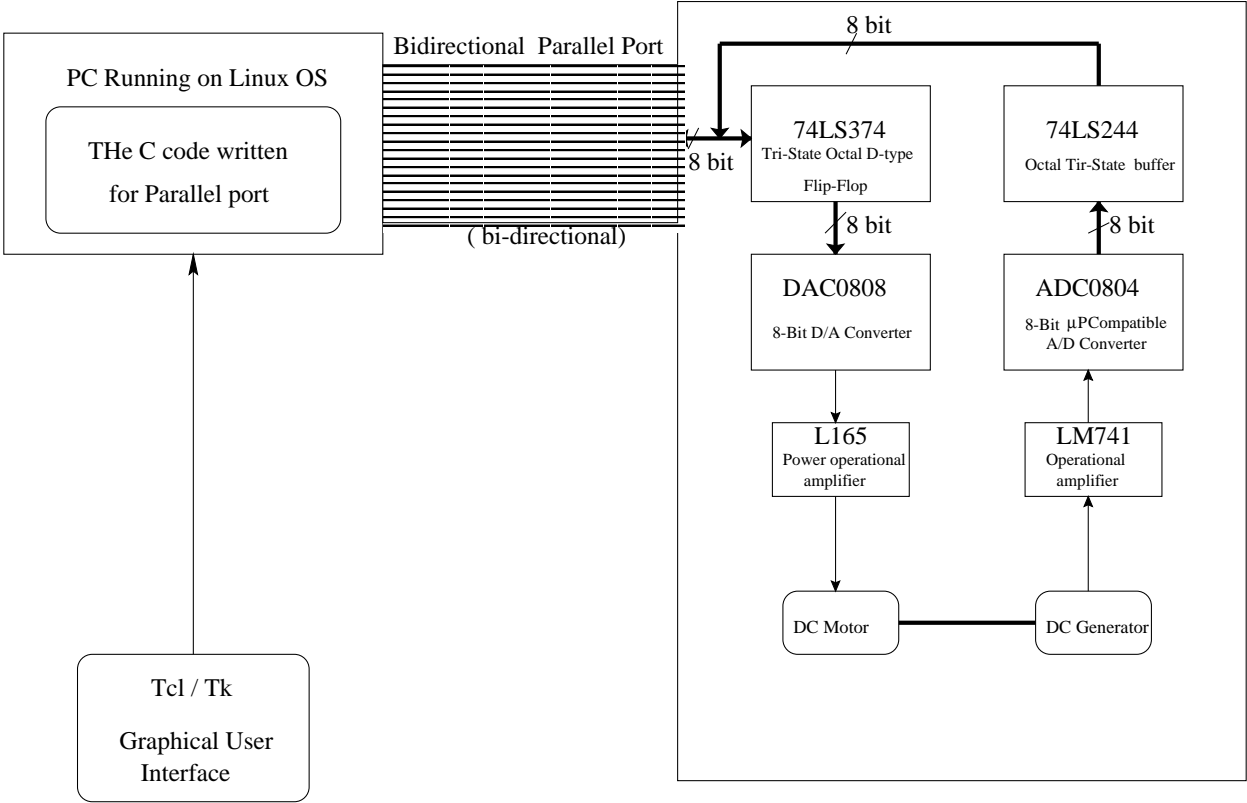


Figure 1.1: Block Diagram for the Design Project

Chapter 2

Theory

2.1 What is a device driver?

The software that handles or manages a hardware attached to a PC is known as the device driver. In other words a driver acts as a translator between the device and the programs that use the device. The device driver has to be installed with a kernel in super user mode. An application program in the user space would access this device driver in the kernel space to communicate to the designed hardware. The major difference between the user and super user modes is that, user mode application software needs a mediator to communicate with the hardware, which is kernel module. Figure 2.1 shows the relationship between device driver and the Linux system. Whenever the kernel recognizes that a particular action is required from the device in user mode, it calls the appropriate driver routine, which passes control from the user space to the kernel space. The hardware sends any action to user space device driver, which is returned to the user process when the driver routine is completed. There are two major different device drivers available in Linux, character devices and block devices. Character devices have open, read, write and close functions that are called directly whenever I/O is required, as there is no buffering performed e.g. line printer. These devices usually can only be accessed sequentially. Block device drivers have a buffer for requests. This can only be written to and read from in multiples of the block size. Block devices are accessed via the buffer cache and may be randomly accessed, i.e. any block can be read or written to no matter where it is on the device e.g. disks. These devices can be distinguished in the device listing; character device starts with "c", and block device, "b". The existing device list can be viewed using the following user command `cat /dev/ls -l`.

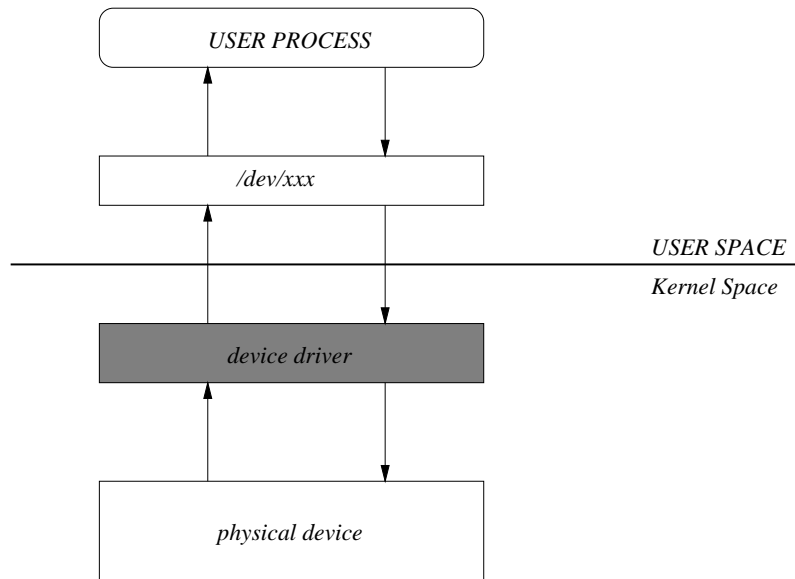


Figure 2.1: Relationship between device driver and the Linux operating system

2.2 What is major and minor number?

The fundamental for using the device driver in Linux is that both device drivers must register themselves in a kernel array. Linux keeps an array of device drivers, and each driver is identified by a number, called the major number, which is listed under `/proc/devices` in Linux operating system. This major number is nothing more than the index in the array of available drivers. All devices controlled by the same device driver are given the same major number, and of those with the same major number, different devices are distinguished by different minor numbers.

2.3 What is kernel?

The Linux kernel acts as a mediator for the programs and the hardware. First, the Linux kernel provides a virtual machine interface for user processes. The kernel is capable of running multiple processes concurrently (where each process has a copy of the kernel). In addition, it is responsible for allocating resources, where each process has a fair and mediating access to hardware resources, to implement write scheduling algorithms to allocate time slot for each process. There are some overheads, if the device is read by two processor, then each processor can access the kernel sequentially i.e. one at a time. This leads to context-switching between

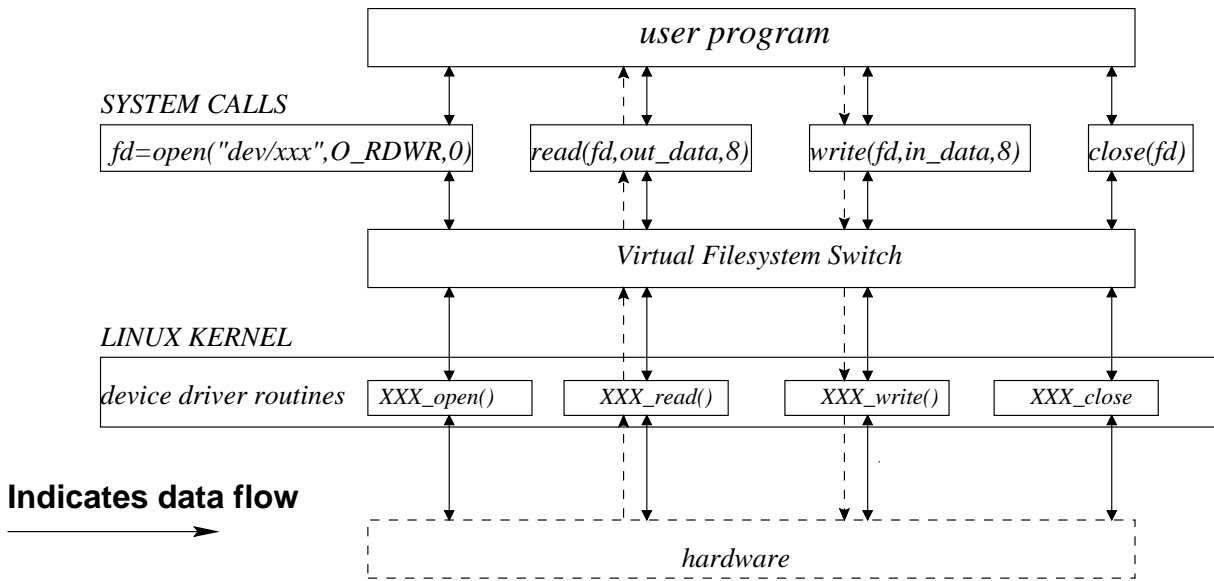


Figure 2.2: How the Linux kernel implements a device

the two processors. The figure 2.2 illustrates how the kernel module is being called, whenever the user mode processor tries to communicate with hardware.

2.4 What is loadable kernel module?

The Linux device drivers can be loaded, whenever the kernel module is on demand, and unloaded when they are no longer needed. In this case the module compiles separately, and can insert and remove them into the running kernel at almost any time. This can be achieved by using the `insmod filename` command in root privilege, which is listed under `/proc/module`. This makes the kernel very adaptable and efficient with the systems resources.

2.5 Parallel port communication

Parallel ports are mounted in motherboard or in ISA/PCI slot. Each computer has at least one parallel port connected; this port is normally used for printers, tape drives, CD writers, etc. Standard IBM parallel ports have 25 pins configuration, which is shown in figure 2.3 . Each parallel port has three consecutive address locations; the starting address of the parallel port can be verified according to the user, which can be configured by the user as 0x278, 0x378 or 0x3bc. The base address is used to transmit and receive 8-bit data

(BASE_ADDRESS). The next address belongs to the status port, it can input five data bits using pins 11, 10, 12 and 13 (BASE_ADDRESS+1). Final address belongs to control port, which can either input or output data, using pins 17, 16, 14, and 1 (BASE_ADDRESS+2). In the control port address the bits 17,14 and 1 has to be inverted, in order to get the actual values of these ports. This can be implemented by EXCLUSIVE-OR with 0x0B to obtain all highs in the parallel port output. The rest of the of the eight bits are not used hence they are grounded. The logic 1 corresponds to +3.8 to +5 Volts while a logic 0 corresponds to 0 to +0.8 Volts. These address port can be accessed through the system calls outb(), inb() under Linux.

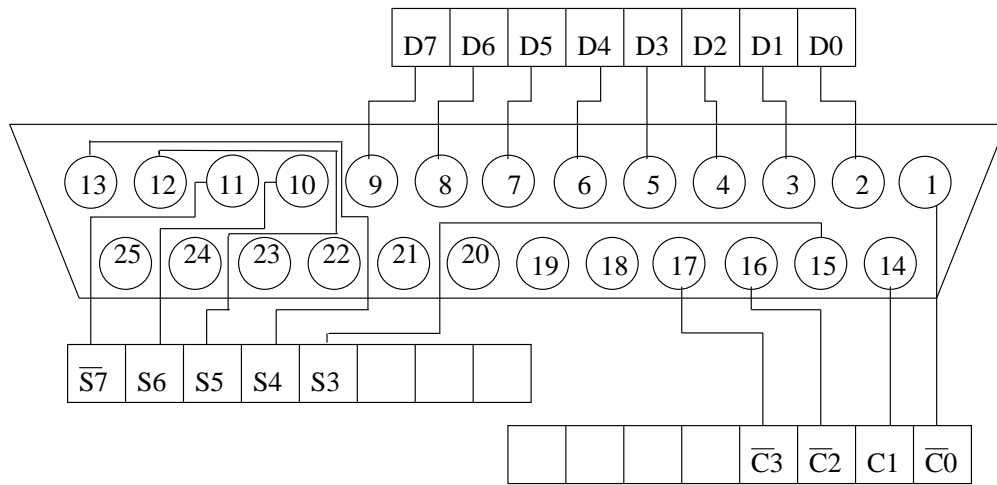


Figure 2.3: 25-way Female D-Type Connector

Chapter 3

Implementation

To write a C program to interface the parallel port in a Linux PC the following steps have to be considered. The parallel port interface is capable of controlling the input/output device attached to the PC. The 25 pin parallel port as implemented on the PC, consists of 8 Data bits, 4 control register, 5 status register and the remaining 8 pins are grounded. The Data registers are used to send, receive data [D7 .. D0] through the parallel port. The control registers are write only and control the status of the signals. The Status registers are used for handshake signals and peripheral errors. The hardware driver, which is written in C language will control the digital output of the parallel port. The output data from the parallel port is first latched to the Flip-Flop, and then the data is sent to the D/A converter, which drives the DC motor through the power amplifier. The corresponding voltage will be applied to the DC Motor.

The Tcl/Tk is the Graphical User Interface which is used to control the device drivers. Tcl/Tk is more user friendly and it is used to control electronic equipments attached to the PC. Whenever the input data is available for the device, the processor calls the driver and displays the current status of the data on the Tcl/Tk widget. The Tcl widget was used to send data to the attached hardware through the PC parallel ports. The scale widget was used to control the voltage through the Tcl/Tk graphical user interface. The corresponding hardware device driver routines will invoke and send the signal to the parallel port. The DC motor was chosen to demonstrate and ensure that the parallel port data transfer was working properly. The speed of the DC motor will change with the applied voltage. Since different voltages are applied to the DC motor, the speed changes accordingly. A mechanically coupled DC motor is used as a power voltage generator. The feedback analog voltage is converted to digital input signal. This was done through the following steps; first, the output voltage of the DC generator is amplified and fed it to the A/D converter. Finally, the output data

is read from the parallel port. Every time the input signal changes, the the device driver program invokes Tcl/Tk and updates the data to the Tcl widget.

Chapter 4

Software Design

The Software Design in this project involves writing a kernel module for the parallel port, an application program to communicate to the parallel port and a GUI. In Linux Operating System, the kernel and user spaces are distinct. The kernel space is accessible only to the super user (in supervisory mode) and all other users do not have access to the kernel space. The kernel module, is compiled with the kernel which constitutes the major part of the Linux Operating System and then runs within the kernel space. Application programs which run in user space will communicate to the kernel module through system calls.

4.1 User Space Programming

This user space program is capable of interfacing with hardware through the kernel module developed for the parallel port device driver.

4.1.1 Definition of System Calls

The following system calls are used to access hardware through */dev/ppt/*, which is the kernel module developed and installed in the Linux operating system. Introduction to the system calls used in the software development are:

- the open call
- the write call
- the read call
- the open call

These system calls are the ones used to interface to any character device on the Linux platform. The following section explains how each system calls are used to interface to the parallel port hardware.

4.1.2 The *Open* System Calls

The open system calls are used to open the hardware device for input/output with the PPT device, which is the name of the device file listed under /dev/ppp. To access the device read and write permission, O_RDWR must be provided. The following open system has been used to access the PPT device.

```
fd = open("/dev/ppp",O_RDWR);
if(fd == -1){
    /* Could not open port.*/
    fprintf(stderr,"ERROR: Unable to open /dev/ppp -%s\n",strerror(errno));
    exit(-1);
}
```

If the operating system is successful in opening the PPT device, it returns the positive file descriptor number, else a negative value (-1) is returned. The file descriptors integer number is utilized by all system calls that interact with the PPT.

4.1.3 The *Write* System Calls

The write system calls read the data form the user space application and write them to a specific open file descriptor (fd) through the kernel module, which talks to the hardware directly. When the value is passed through the write system calls the receiver system calls receives it and sends it to the parallel port through designed hardware, which will be talked about in the next chapter.

For example:

```
int value,status;
status = write(fd,&value,(sizeof(unsigned char))*1);
if(status < 0){
    fprintf(stderr,"ERROR: Write byte failed\n");
    exit(-1);
}
```

On success of write, returned is the number of bytes written, else failed (-1) is returned. Different operating systems have different byte system, because of this reason the sizeof system calls is used to determine the size.

4.1.4 The *Read* System Calls

The read system calls, receives data from the kernel module, the PPT, and sends it to the user space application program through a specific open file descriptor, which talks to the hardware directly. When the data is available in parallel port, the read system call reads data, which is produced by the attached designed hardware connected to the parallel port.

For example:

```
int value,status;
status = read(fd,&Data,0);
if(status == 0){
    fprintf(stderr,"Error: End of file\n");
    exit(-1);
}
```

On success of read, the number of bytes read is returned. Zero indicates end of file, different operating system has different byte system, therefore the sizeof system call is used. For more information about read system calls go to man page.

4.1.5 The *Close* System Calls

The device should be closed to indicate the user space program is finished with the kernel module, this will close the file descriptor which was created by the open system calls. This will allow other user application programs to interface with the kernel module.

```
status = close(fd);
if (status = -1){
    /* Could not close port.*/
    fprintf(stderr,"ERROR: Unable to close /dev/ppt\n");
    exit(-1);
}
```

close returns zero on success, or -1 if an error occurred

The complete list of application program (final.c) that communicates to the parallel port is listed in the appendix A. The command to compile this program is as follows:

```
# gcc final.c -o final.exe
```

At the end of the execution of the above command, the executable file final.exe will be created. The GUI created by Tcl/Tk, uses the *exec* command to execute this program.

4.2 Kernel Module

The advantage of using kernel module programming is that it utilizes the resources of the computer system. The kernel module cannot work on its own initiative, like a processor,

		7	6	5	4	3	2	1	0	
DATA	(D)	×	×	×	×	×	×	×	×	BASE = 278/378/3bc (Hex)
STATUS	(S)	×	×	×	×	×	—	—	—	BASE+1
CONTROL	(C)	—	—	—	—	$\overline{\times}$	×	$\overline{\times}$	$\overline{\times}$	BASE+2

Figure 4.1: Parallel Port I/O Pin Configuration

they run only when a processor calls via system calls such as read and write. In this documentation, CONTROL register, STATUS register, and DATA register are displayed in the I/O pin configuration table shown in Figure 4.1.

Writing a kernel module involves developing a set of functions which will help to communicate to a hardware device. All the necessary data structures needed to write a kernel module for any hardware device are already defined in the kernel. The *printk* command is used in kernel program instead of *printf* which is used in all user programs.

The following functions are developed for the parallel port kernel device are as follows:

- `init_module()`
- `ParallelPort_open()`
- `ParallelPort_close()`
- `ParallelPort_read()`
- `ParallelPort_write()`
- `void cleanup_module()`

Brief descriptions of each module in the kernel code as presented below

4.2.1 The *init_module* System Calls

When the user command *insmod* is used, this automatically invokes the *init_module* system call which loads the kernel module image into kernel space and runs the module's init function. The user command *insmod* is described in Section 4.3. The *init_module* module image begins with a module structure and is followed by code and data as appropriate

```

int init_module()
{
    int reg_value;
    /*
    *Registers the character "MAJOR_NUM" for the device.
    */
    reg_value = register_chrdev(MAJOR_NUM,DEVICE_NAME,&ParallelPort_fops);
    if (reg_value < 0)
    {
    /*
    * if the MAJOR_NUM returned a negative value, error message will be printed on
    * screen.
    */
        printk ("The character device is not registered %d \n",reg_value);
        return reg_value;
    }
    /* if the MAJOR_NUM is successfully registered, then the following messages
    * will be displayed
    */
    printk("The Registration was successful\n");
    printk("The Major device number is %d\n",MAJOR_NUM);
    printk("In order to communicate from the USER MODE \n");
    printk("to the device driver,\n");
    printk("The following command has to be run on root access \n");
    printk ("mknod /dev/ppt c %d 0\n", MAJOR_NUM);
    return 0;
}

```

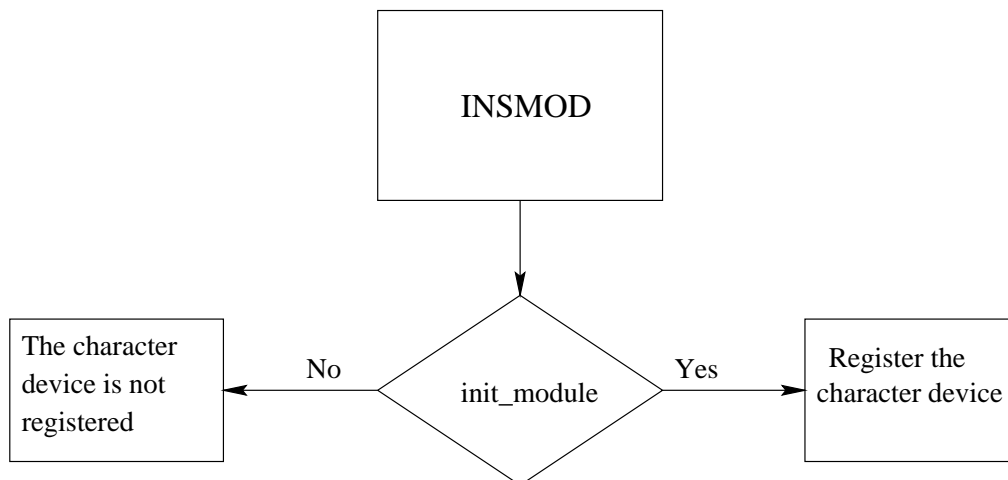


Figure 4.2: The flowchart of the `init_module` System Calls

The function `register_chrdev()` is called in the first argument which the major number. The second argument is the device name, and the final argument contains the `file_operations` data structure. The most important function is `register_chrdev()` which registers our driver

with the major number 25, and adds the device driver to kernels character device driver table. The flowchart Figure 4.2 describes the functionality of the `init_module`.

4.2.2 The *cleanup_module* System Calls

This function is called when a request is made to remove a module from the kernel. The module can be unregistered through the command `#rmmod pport`.

```
void cleanup_module()
{
int num;
/*
 *if the device is not registered, the proper message will be printed
 */
num = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
if(num < 0)
    printk("ERROR: The character DEVICE is not registered \n");
else
    printk("Kernel module Successfully removed \n");
}
```

The function `unregister_chrdev()` is called to remove the loadable module from the kernel with the major number as its first argument and the device name as its second argument. The function returns a positive integer for success and negative value for failure. Only when the module use count is zero, can the module be unloaded. The flowchart Figure 4.3 describes the steps taken, once the `rmmod` user command is called:

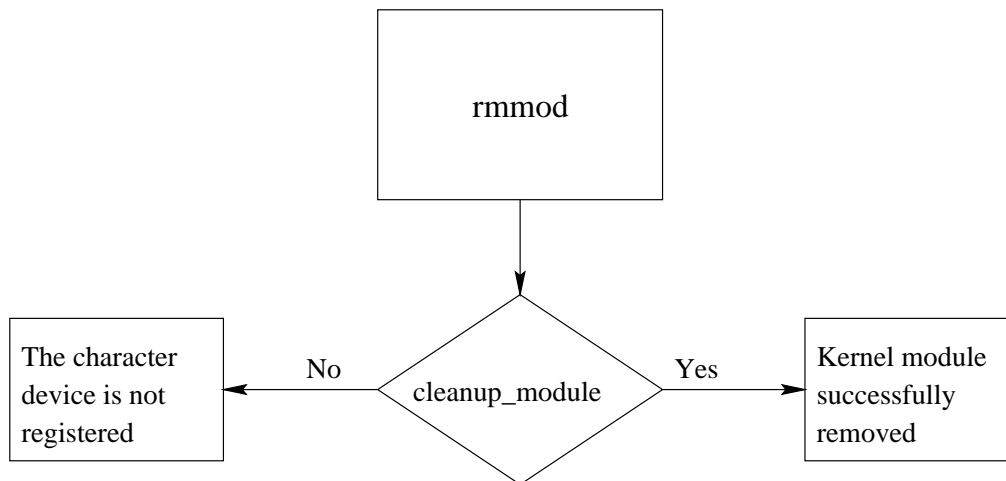


Figure 4.3: The flowchart of the `close_module` System Calls

4.2.3 The *ParallelPort_open* System Calls

This function is called in the processor open system calls to open the hardware device ppt. The file_operations structure will be initialized while the open system calls opens the device. The ParallelPort_fops pointer, points to a table of operations such as open, read, write, etc. The operation not in use contains NULL pointer unsupported function . The macro *MOD_INC_USE_COUNT* has the same effect with respect to module loading and unloading. Each time *MOD_INC_USE_COUNT* is invoked, the kernel increments the counter referencing to the module, and if a *MOD_DEC_USE_COUNT* is invoked then the kernel decrements the counter to release the driver. The example code is follows:

```
static int ParallelPort_open(struct inode *inode,
                           struct file *file)
{
    MOD_INC_USE_COUNT;
    return 0;
}
```

4.2.4 The *ParallelPort_close* System Calls

When the open systems calls is invoked, the kernel opens a file descriptor and to close this file descriptor the ParallelPort_close command is used. Each time the processor closes a kernel module that was being used, it decrements the *MOD_DEC_USE_COUNT* to release the driver. The example of the code is follows:

```
static int ParallelPort_close(struct inode *inode,
                             struct file *file)
{
    MOD_DEC_USE_COUNT;
    return 0;
}
```

4.2.5 The *ParallelPort_write* System Calls

This function is called when the processor tries to write some operations to the kernel device driver. The write function contains four parameters. First, is the file descriptor which was created by the open system calls. The file descriptor is used as mediator to interface the desired device driver. The second, is the buffer to fill with the data, the third is the length of the buffer and finally the offset to the file. This function is communicating directly with the parallel port, using outb() system calls to send data i.e. it writes the data to the parallel port BASE_ADDRESS (default parallel port base address 0x378). To send the data to the hardware attached to the PC's parallel port to drive the DC motor, the 74LS374 FLIP-FLOP is used. The Flip-Flop needs a clock pulse to latch the data to the DAC0808 (D/A), which

drives the DC motor through power operational amplifier. The control register Pin1 (C0) is used to generate the clock pulse, which is write logic 0 to the pin and to transfer the data the clock is raised by, write logic 1 to the control Pin1. On the positive edge trigger the data is being transfered to ADC0808. Since the Pin C1 is logic 1the Octal Tri Sate Buffer will be disabled, this will make sure that only the output data is in the bus. The following code demonstrate the action:

```
static ssize_t ParallelPort_write (struct file *file,
    /* The buffer to be filled with the data */
    const char *buffer,
    /* The length of the buffer */
    size_t length,
    /* offset value to the file */
    loff_t *offset)
{
    char value = *buffer;
    /*
    * To set the clock for the 74LS374 positive-edge triggered flip-flops, set the
    * control register C0 = logic "0".
    */
    outb(CTRL(0x0E),BASE_ADDRESS+2);
    /*
    * Send the Data to the parallel port BASE_ADDRESS.
    */
    outb((unsigned char)value, BASE_ADDRESS);
    /*
    * To latch the data in the positive-edge triggered flip-flops, write logic
    * high to control register C0, which outputs the data. Since, C1 was set to
    * logic "1" the Octal tri state buffer will be disabled; this will make sure
    * that only the output data is in the bus.
    */
    outb(CTRL(0x0F),BASE_ADDRESS+2);
    return 0;
}
```

4.2.6 The *ParallelPort_read* System Calls

This function is called when the processor tries to read some operations from the kernel device driver. The read function contains four parameters. First, is the file descriptor, which was created by the open system calls. The file descriptor is used as mediator to interface to the desired device driver. The second is the buffer to fill with the data, the third is the length of the buffer and finally the offset to the file. This function communicates with the parallel port, using inb(), outb() system calls to receive, send data respectively. To receive data from the hardware attached to the PC's parallel port, first, the ADC0804 (A/D) uses the feedback from DC motor through the operational amplifier, which is used as a power generator. For the reset condition for the A/D, logic 0 is written to C3 which starts the conversion process. To check the output signal, INTRis to set to logic 1. INTR with logic

indicates the conversion has been initiated. The digital data will be read if RD is set by the CONTROL register C2 (C2 = 1). The data will be made available in the A/D output bus. The logic 0 written to the CONTROL register C1, enables the tri state buffer and puts data into the output bus. Since input Flip-Flop transfers data in a positive edge trigger, the data in the bus will not be latched. That will make sure the output data does not interact with input data. To make the parallel port bi-directional the control register bit 7 has to be set and clear. The input data can be received by setting bit 7 of the CONTROL register to logic 1, which will allow the input data through the parallel port. The code is used to achieve this is displayed below:

```
static ssize_t ParallelPort_read(
    struct file *file,
    /* The buffer to filled with the data */
    char *buffer,
    /* The length of the buffer */
    size_t length,
    /* offset value to the file */
    loff_t *offset)
{
    unsigned char adc_status;
    char temp;
    /*
     * Reset the ADC0804 converter to reset condition, write logic low to the
     * C3 to start the conversion process.C3 = logic "0"
     */
    outb(CTRL(0x07),BASE_ADDRESS+2);
    /*
     * Checks the output signal "INTR" to set to logic "1". INTR with logic "1"
     * indicates the conversion is initiated.
     * Performing the detection of the signal "INTR" from the status register S3.
     */
    adc_status = (inb(BASE_ADDRESS+1) & 0x08);
    /*
     * The logic "high-to-low" ("1" to "0") transition on the WR signal pin, puts
     * the ADC to the RESET condition. C3 = logic "1".
     */
    outb(CTRL(0x0F),BASE_ADDRESS+2);
    /*
     * Wait until the internal conversion is completed; upon completion,
     * the INTR signal will goes "0". This will be detected form status register
     * S3.
     */
    while(adc_status)
    {
        adc_status = inb(BASE_ADDRESS+1) & 0x08;
    }
    /*
     * The digital data will be read if "RD" is set ("1") by the control registerC2.
     * The data will be available in the ADC0804 output buses when "RD" is high.
     */
    outb(CTRL(0x0B), BASE_ADDRESS+2);
    /*
```

```

* When logic "0" written to the control register C1, it enables the tri-state
* buffer and put the data into the output buses. Hence, the data can be read
* through the parallel port.
*/
    outb(CTRL(0x09), BASE_ADDRESS+2);
/* To make the parallel port bi-directional the control register bit 7 has
* to be set or clear.
* clear - output from parallel port
* set   - input to parallel port
*/
    outb(CTRL(0x60), BASE_ADDRESS+2);
/*
* Read the input Data and put it into the address space, where the buffer is
* pointing to.
*/
    *buffer = inb(BASE_ADDRESS);
/*
*Copy the Data to the variable temp to return the value
*/
    temp = *buffer;
    return temp;
}

```

4.3 Setting up the System

This section deals with the Compiling, Linking and Loading the kernel module to enable the user program to access the hardware device (parallel port).

4.3.1 Compiling and Linking the kernel module

Only user having super user root access can compile the kernel module. To compile and link the kernel module, the following command is executed

```
# gcc -O2 -Wall DMODULE D__KERNEL__ -DLINUX -c I/usr/include/linux/version.h pport.c
```

The compiler options used in the above command are described below

- "-O2"
 - is included in the compiling command since it is the inline macro needed to compile with the optimization enabled.
- "-Wall"
 - enables all the warnings, which is a best tool for DEBUG
- "D__KERNEL__"
 - informs the header file that the following code is a kernel module

- "DMODULE"
informs the header to give the appropriate definition for the kernel module
- "DLINUX"
allows to compile the kernel module in different operating system
- "-c"
this option creates object code file *pport.o*
- "I/usr/include/linux/version.h"
enables the "CONFIG_MODVERSIONS", this will have to be defined when compiling.
- "pport.c"
the written C code file for the kernel module

4.3.2 Installing the kernel module

At the end of compiling and linking, the object file *pport.o* is created. To install the kernel module into the loadable kernel, the object file of the kernel module is used by the following super user command:

```
# insmod pport.o
```

4.3.3 Register the character device

Once the kernel module is installed, the user program cannot access the parallel port until the device is registered with the kernel. Also, permissions must be granted for other users to read and write to the device. To achieve this, the super user command *mknod* creates a path for file system node (file, device special file) named path name. It also, specifies both the permissions to use and the type of node to be created. A path is created by the following command:

```
# mknod -m 0666 /dev/ppt c 25 0
```

Where *ppt* is the device name chosen for the parallel port which is defined at the beginning of the program in appendix B and the unused number, 25, in the device driver table, is used as the major device number for the parallel port.

Once this command is executed, the device is registered to the device driver table in the kernel which gives all users the permission to read and write.

4.3.4 Removing the kernel module

If it is needed to remove the loadable kernel module from the kernel, the super user command *lsmod* is used to gather information about all loaded modules. The format is:

[name] [size] [list of referring modules]

The loadable kernel is removed by the following command:

```
#rmmod pport
```

4.3.5 Unregister the character device

To delete the path of the character device from the device driver table the following command is executed.

```
#rm ppt
```

Chapter 5

Hardware Design

5.1 Purpose

The purpose of this project was to design, implement and make operational hardware to drive a DC motor, which was mechanically coupled with another similar motor. The mechanically coupled DC motor is acts as a power generator. The PC parallel port digital output data is used to drive the DC motor; which is connected through a tri state buffer and a DAC0808 (D/A). The output current is converted to voltage, using L165 power operational amplifier. The output voltage of the power amplifier is set to 5Volts (maximum) to drive the motor. In the feedback circuit, the maximum voltage obtained from the DC motor is 2.42 Volts, which is amplified, using the general purpose power amplifier. This voltage is fed into the ADC0804 (A/D), then to 74LS244 Tri State Buffer, and parallel port input. The schematic of the Hardware design is described in Figure 5.1, Power supply design schematic is in Figure 5.2 , the data sheets are placed in Appendix E.

5.2 Design Specification

The D/A and A/D was designed to meet the specifications described in the following section:

5.2.1 Power Operational amplifier

From Figure 5.3 the output voltage depends on the current drawn by the DAC0808, which draws upto a maximum current of 3.25mA, therefore the feed back resistor used, has to be 1.53Kohms, to make the output voltage to + 5 Volt. A 0.1 μ F capacitance is used to damp out the oscillations in the power supply. The power supplied was +/- 13 Volts.

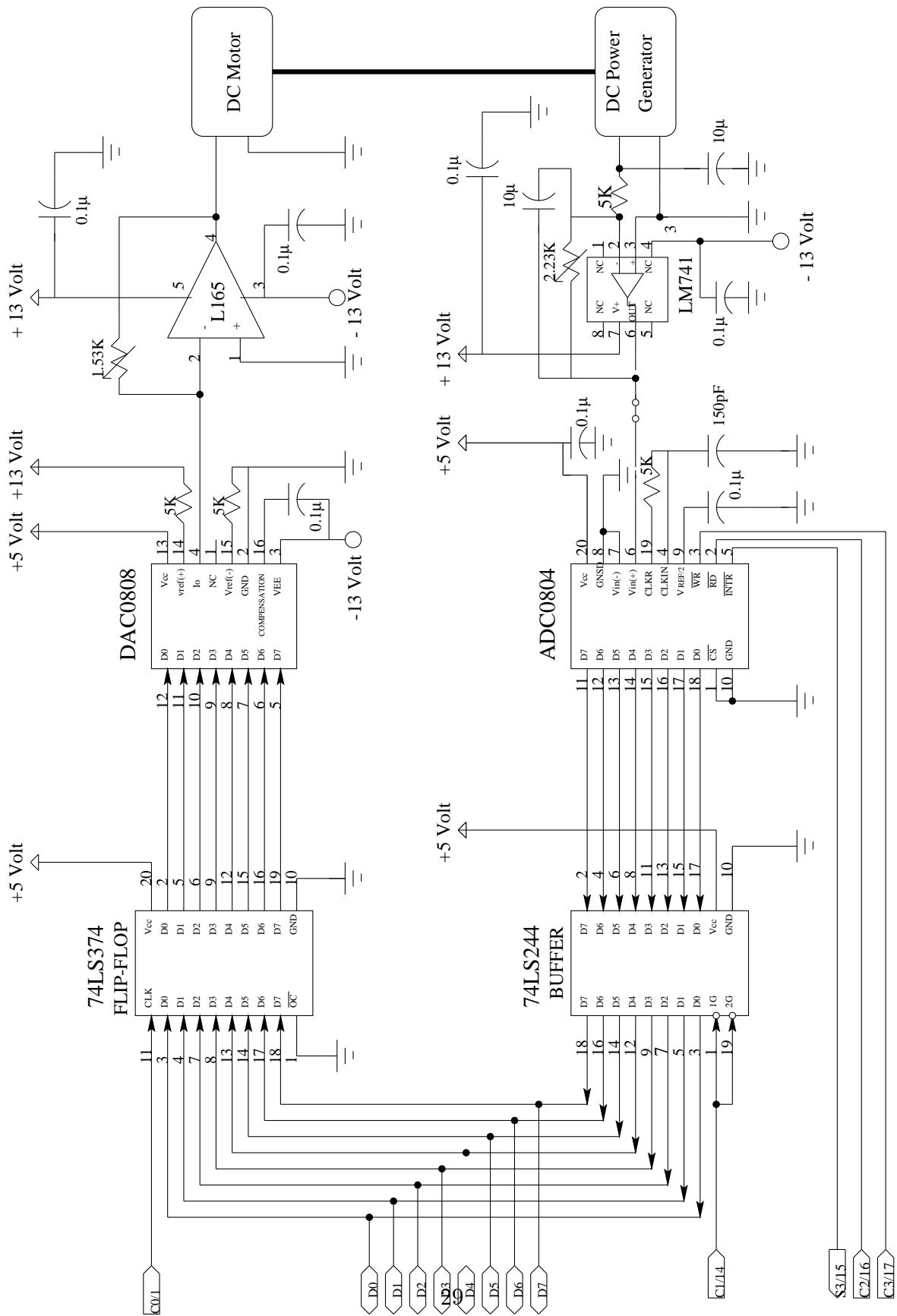


Figure 5.1: The schematic of the Hardware design

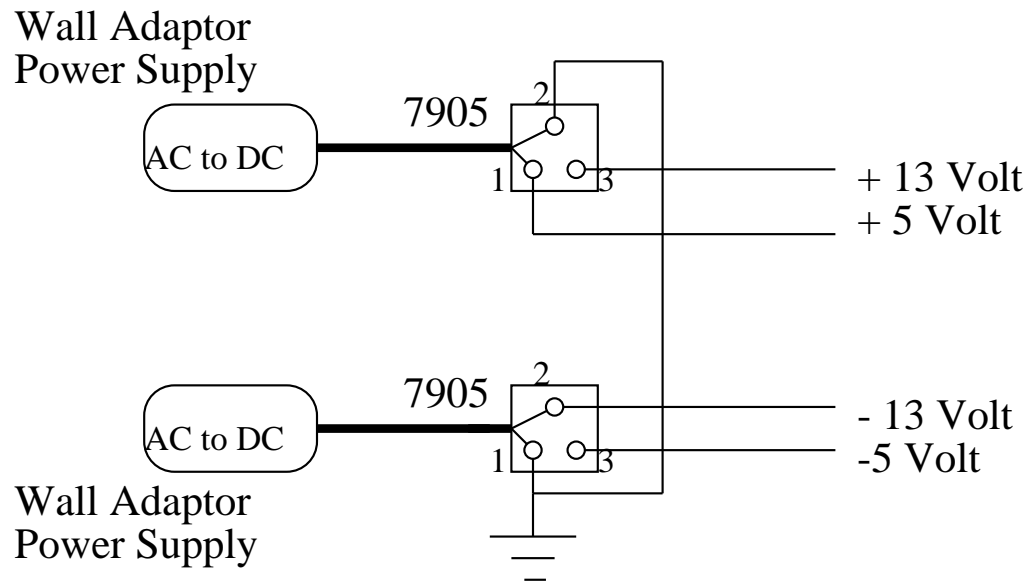


Figure 5.2: The designed power supply

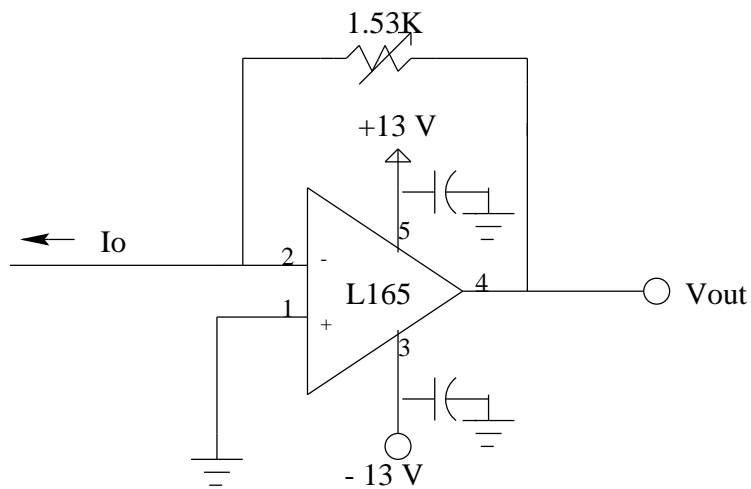


Figure 5.3: The schematic of the power operational amplifier [L165]

5.2.2 General Purpose operational amplifier

From Figure 5.4, the feedback DC motor power generator produced a maximum of 2.41 Volts, this voltage needed to be amplified in order to drive the ADC0804 in full digital out, 0xFF. To obtain maximum voltage of +5 Volts, the general-purpose operational amplifier was used. The feedback resistor was calculated to be 2.23Kohms, and a $10\mu\text{F}$ capacitor was placed in parallel to filter the oscillation of the input voltage. The $0.1\mu\text{F}$ capacitance used to damp out the power supply oscillation, the $10\mu\text{F}$ capacitance connecting the input voltage to ground also reduce the oscillation. The power supply applied was ± 13 Volts.

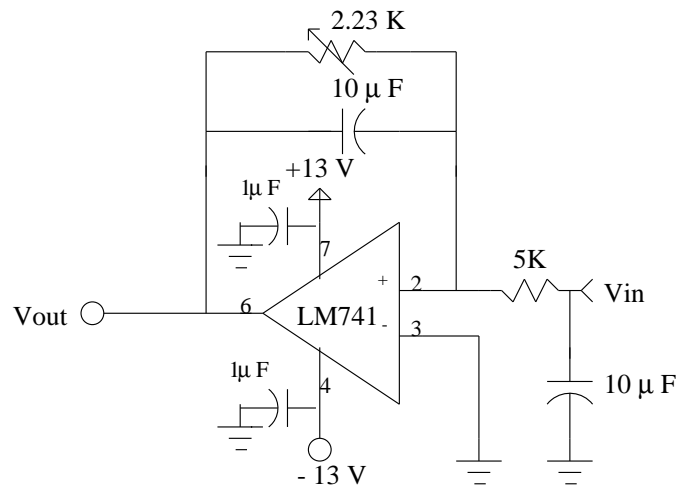


Figure 5.4: The schematic of the operational amplifier [LM741]

5.2.3 Digital to analog converter DAC0808

The typical application in the Page 4 on the data sheet located Appendix D was referred to for the design of this converter. Power supply voltage range was shown to be ± 4.5 V to ± 18 V. According to the change for the digital inputs the current drawn by the DAC will verify the maximum current of 3.25mA drawn when all the signals are logic high, and 0 mA when all the signals are logic low

5.2.4 Analog to digital converter ADC0804

Changing the input voltage at pin6 ($V_{IN}(+)$), from 0 +5 Volts, changes the digital output accordingly. The 0 Volt represent all logic low in the output, and +5 represent the all logic

high respectively. The figure 5.5 will be referred for the ADC0804 reset, data conversion, and complete conversion, which is described in ADC software design section 4.2.6

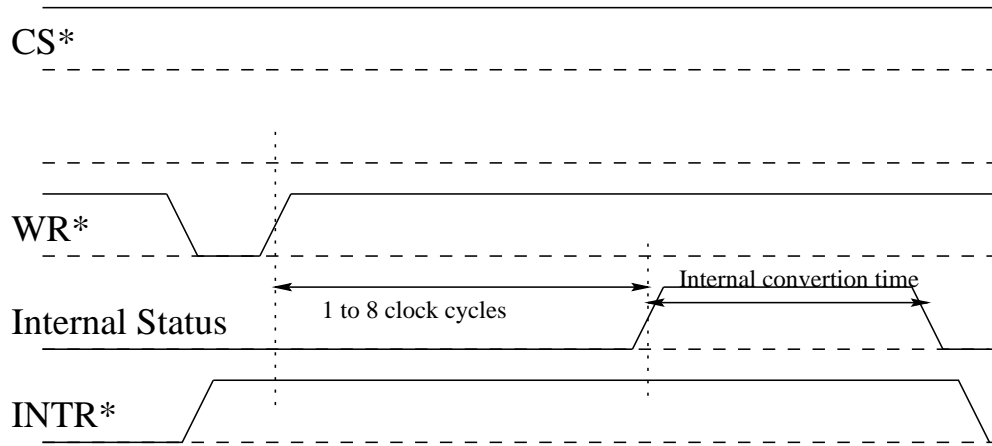


Figure 5.5: Timing diagram of the conversion of the ADC0804

5.2.5 Flip-Flop 74LS374

The Tri-State Octal D-Type Transparent Latches and Edge-Triggered Flip-Flop was used to latch the output data from the parallel port output. The advantage of using this Flip-Flop is when the input data and output data are sharing the same bus, the only output data is latched, through the positive edge triggered of the CLOCK generated by the software.

5.2.6 Octal Tri-STATE Buffer

The Octal Tri-STATE Buffer was used to put the digital output in the parallel port bus. The output is made available, when the tri state buffers are enabled, and the advantage is that no output data transfer through the buffer.

5.2.7 The layout of the Hardware Design

The layout of the Hardware design is shows in the Figure 5.6

5.2.8 The Tcl/Tk widget

Tcl/Tk is freely available software and is compatible with C language program. The Tcl/Tk Graphical User Interface was used to enable the user to provide the input and to monitor the output for the entire operation. The Tcl program uses the *catch* command to pass the user

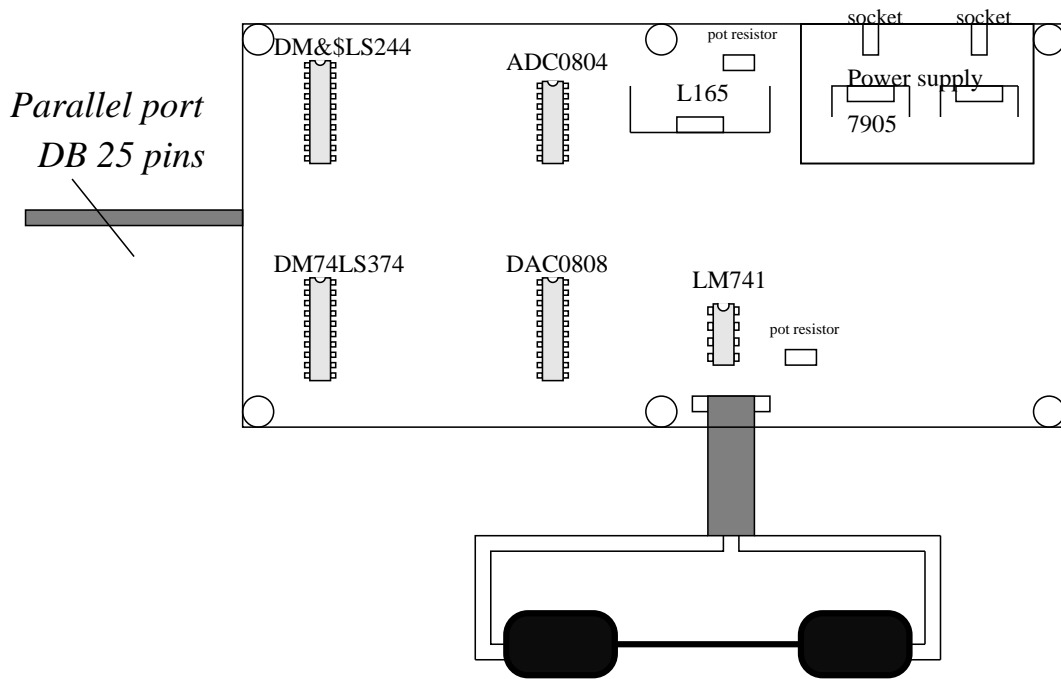


Figure 5.6: The Hardware layout

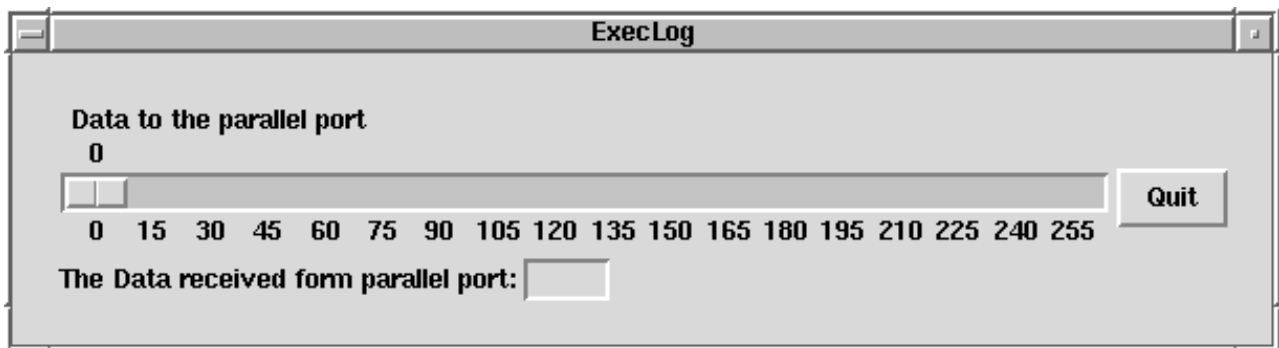


Figure 5.7: The GUI for control the DC motor and Generator

input to the C program and also executes it which in-turn will communicate to the parallel port. The output produced by the C program, read from the parallel port, was captured using the *gets* command and displayed in the GUI. The input value can be changed using the sliding bar in the GUI which varies from 0-255 or maximum value of 11111111 in binary. This input value will be fed into the D/A converter of the hardware circuit built for this specific application. The analog value received from the generator of the hardware is fed through the A/D converter and then is fed to the input channel of the parallel port. The GUI used for the design project is in Figure 5.7.

Conclusions

The objective of this project was to study how hardware device driver works in user mode application as well as in kernel mode. The device driver development involves a depth knowledge of operating system, hardware design, and programming language. Upon successful completion of the Design project, it was evident that the design project was constructed using familiar Hardware components. The overall project provided important information concerning the concept used. Although, implementation of device is almost dealt with the operating system, this may crash the computer. Therefore writing kernel level device driver is considered challenging and risky. The hardware, which was connected to the PC parallel port may damage the port if the signal's voltage exceeded +5 Volts. Some computer parallel ports are mounted to the motherboard, and these systems may damage the whole motherboard when the voltage exceeds the limit.

Recommendations

The parallel port was designed for the printer, and is being used by other devices presently. In this project out of the active pins, only 13 were used. This project could be improved by using IRQ (interrupt request) to utilize the computer resources. The IRQ method would be better than polling, because the polling wastes the CPU time . In order to use the IRQ techniques, the A/D converter strobes a logic high signal whenever the data is available for input. The high strobes signal will be handled by software routine.

Bibliography

- [1] Axelson, Jan, *Parallel port complete*, Lakeview Research Madison, WI53704, 1997
- [2] Dhananja V.Gadre, *Programming the Parallel port*, R & D Books, 1998
- [3] Rubini, Alessandro, *LINUX Device Drivers*, O'Reilly & Associates Inc., 1998
- [4] K.Johnson, Michael and W.Troan, Erik, *Linux Application Development* Addison Wesley Longman, inc., 1998
- [5] J.M. Naughton, *communicating with hardware on the Linux Platform*, <http://www.ee.ryerson.ca/~jnaughto>, Jan. 2000
- [6] Eric Foster & Johnson, *Graphical applications with Tcl&Tk* M&T Books, California.1997
- [7] Warp Nine Engineering 1998-2000 <http://www.fapo.com/porthist.htm>

Appendix A

User space C code

```

/*****
 *
 *           User space application program
 *
 *****/
/*
 * Copyright Notice
 * =====
 * Copyright (C) by Vasuthevan Maheswaran
 *
 * File ref: final.c
 *
 * This program outputs and inputs the data value to the PC parallel
 * port data pins (default lpt1 I/O address 0x378).
 * - writes data bytes from the Tcl widget to the parallel port via
 *   kernel device. The maximum value can be 255 or Hex. 0xFF
 * - reads the parallel port input and display it using Tcl widget
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

/*
 * Function: byte_write
 *
 * Description: This function writes a byte to the device driver.
 *
 * @param:
 *   fd      - Given open file descriptor
 *   value   - The byte pass through kernel module to the
 *             positive-edge triggered flip-flops hardware
 */

```

```

* @return:
*     Nothing
*
* EXIT STATUS
*     void function
*/
void byte_write(int fd, unsigned char value){
    unsigned char Data;
    int status;
    status = write(fd,&value,(sizeof(unsigned char))*1);
    if(status < 0){
        fprintf(stderr,"Error: Write byte failed\n");
        exit(-1);
    }
}

/*
* Function: byte_read
*
* Description: This function captures a byte of data from the parallel
*              port BASE address 0x378.
*
* @param:
*     fd      - Given open file descriptor
*
* @return:
*     Data    - The byte read from the parallel port when the data
*              being output from the Octal tri-state buffer
*
* EXIT STATUS
*     void function
*/
void byte_read(int fd){
    int status;
    unsigned char Data,Speed;
    status = read(fd,&Data,0);
    fprintf(stderr,"Error: Read byte failed\n");
    if(status == 0)
        exit(-1);
    /*
    * Calculation of the speed of the motor
    */
    Speed = (Data/4);
    printf("%d",Speed);
}

main(int argc, char **argv)
{
    int fd;

```

```

int value;
/*
 * open port() - Opens the parallel port device for communication.
 *              - returns the file descriptor on success or "-1" when
 *              - an error encountered
 */
fd = open("/dev/ppt",O_RDWR);

if(fd == -1){
/*
 * When the port can not be opened, it prints an error message
 */
fprintf(stderr,"Error: Unable to open /dev/ppt -%s\n",strerror(errno));
exit(-1);
}
/*
 * Converts ASCII to integer;
 */
value = atoi(argv[1]);
byte_write(fd,value);
/*
 * Generates a time delay.
 * The DC motor works in the range of Hz, ICs work in the range of KHz,
 * Hence, a time delay is needed between DC motor and DC power generator
 * in order to get data's.
 */
usleep(400000);
byte_read(fd);
/*
 * Close the file since reading and writing are done to it.
 */
close(fd);
}

```

Appendix B

The C code for the ppt Hardware Device

```
/* *****
*
*                               Run-time
*                               Loadable kernal module
*
* *****/

/*
* Copyright Notice
* =====
* Copyright (C) by Vasuthevan Maheswaran April 2000
*
* file ref: pport.c
*
* Character Device
* =====
* Device driver provides low-level interface to hardware device drivers
* To pick a MAJOR number, make sure the number is being not used by the other
* character device, the number chosen for this module is 25. The device
* driver can support multiple similar devices all share the same MAJOR
* number, but are distinguished by their MINOR numbers. All Character devices
* are listed in the /proc/devices directory.
*
* Hardware Environment
* =====
* This kernel module can be run on the Hardware described in Figure xx
* ADC0804 - 20 pins Analog to digital converter " see Appendix E for more
* details"
* DAC0808 - 16 pins Digital to analog converter "see Appendix E for more
* details"
* 74LS374 - 20 pins Flip-Flop the data will be latched, positive edge
* triggered "see Appendix E for more details"
*
* 74LS244 - 20 pins Tri state buffer "see Appendix E for more detailed"
```

```

*
* Overview
* =====
* The kernel module described here can be accessed by multiple processors,
* but only one processor at a time. In order to compile the kernel module,
* the super user privilege must be needed. But in the user mode, user can
* only access the hardware driver if the permission for read and write was
* given by the super user.
*
* In parallel port, the control pins are:
*   X -> pins are used
*   - -> pins not used
*
* CONTROL Register C7 C6 C5 C4 C3 C2 C1 C0
*                 - - - - X X X X
*
* STATUS Register  S7 S6 S5 S4 S3 S2 S1 S0
*                 X X X X X - - -
*
* DATA Register   D7 D6 D5 D4 D3 D2 D1 D0
*                 X X X X X X X X
*/

/*
 * All the modules are defined in this library
 */
#include <linux/module.h>

/*
 *Deal with CONFIG_MODVERSIONS
 */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/*
 * The inb()and outb() system calls are defined in this header file for
 * parallel port character devices access.
 */
#include <asm/io.h>

/*
 *The character device definitions are defined here
 */
#include <linux/fs.h>

/*
 * Invert proper bits to parallel port CONTROL register, so that the parallel
 * port control lines are active high, because if the data to be written is
 * 00000100, then the output will be 00001111, to avoid this bit inverting the
 * following MACROS being used

```

```

*/
#define CTRL(x) ((x) ^ 0x0B)

/*
 * The text file under the following linux directory:
 *     /usr/src/linux/Documentation/devices.txt
 *                                     must be checked, before assigning any numbers for
 * the Major device number.
 */
#define MAJOR_NUM 25

/*
 * The name of the device file created is added to the directory:
 *     "/dev/ppt"
 * For the user to access the this file, the permission must be given by the
 * super user. The following command can be used to give the permission:
 *     chmod 666/dev/ppt
 */
#define DEVICE_FILE_NAME "ppt"

/*
 * The name of the character device is "char_dev". When the device is
 * successfully installed, it will appear in /proc/devices directory.
 */
#define DEVICE_NAME "char_dev"

/*
 * The base address of the 1st parallel port
 */
#define BASE_ADDRESS 0x378          /* 1st Parallel Port */

/*
 * This function will be called whenever a process attempts to open
 * the device file. Make sure only one processor can talk to the
 * device at a time
 */
static int ParallelPort_open(struct inode *inode,
                             struct file *file)
{
    MOD_INC_USE_COUNT;
    return 0;
}

/*
 * This function will be called whenever a process closes the device file.
 */
static int ParallelPort_close(struct inode *inode,
                              struct file *file)
{
    MOD_DEC_USE_COUNT;
    return 0;
}

```

```

/*
 *This function will be called whenever a process, which has already
 *opened the device file, attempts to read it.
 */
static ssize_t ParallelPort_read(
    struct file *file,
    /* The buffer to filled with the data */
    char *buffer,
    /* The length of the buffer */
    size_t length,
    /* offset value to the file */
    loff_t *offset)
{
    unsigned char adc_status;
    char temp;

    /*
     * Reset the ADC0804 converter to reset condition, write logic low to the
     * C3 to start the conversion process.C3 = logic "0"
     */
    outb(CTRL(0x07),BASE_ADDRESS+2);

    /*
     * Checks the output signal "INTR" to set to logic "1". INTR with logic "1"
     * indicates the conversion is initiated.
     * Performing the detection of the signal "INTR" from the status register S3.
     */
    adc_status = (inb(BASE_ADDRESS+1) & 0x08);

#ifdef DEBUG
    printk("adc_status == %x\n",adc_status);
#endif

    /*
     * The logic "high-to-low" ("1" to "0") transition on the WR signal pin, puts
     * the ADC to the RESET condition. C3 = logic "1".
     */
    outb(CTRL(0x0F),BASE_ADDRESS+2);

    /*
     * Wait until the internal conversion is completed; upon completion,
     * the INTR signal will goes "0". This will be detected form status register
     * S3.
     */
    while(adc_status)
    {
        adc_status = inb(BASE_ADDRESS+1) & 0x08;
    }

#ifdef DEBUG
    printk("adc_status 2 == %x\n",adc_status);
#endif
}

```

```

#endif

/*
 * The digital data will be read if "RD" is set ("1") by the control registerC2.
 * The data will be available in the ADC0804 output buses when "RD" is high.
 */
    outb(CTRL(0x0B), BASE_ADDRESS+2);

/*
 * When logic "0" written to the control register C1, it enables the tri-state
 * buffer and put the data into the output buses. Hence, the data can be read
 * through the parallel port.
 */
    outb(CTRL(0x09), BASE_ADDRESS+2);

/* To make the parallel port bi-directional the control register bit 7 has
 * to be set or clear.
 * clear - output from parallel port
 * set    - input to parallel port
 */
    outb(CTRL(0x60), BASE_ADDRESS+2);

/*
 * Read the input Data and put it into the address space, where the buffer is
 * pointing to.
 */
    *buffer = inb(BASE_ADDRESS);
/*
 * Copy the Data to the variable temp to return the value
 */
    temp = *buffer;

#ifdef DEBUG
    printk("\n Read Data_byte  ===== %d\n",buffer);
#endif

    return temp;
}

/*
 * This function will called whenever a process has already opened the device
 * file and attempts to write to the parallel port.
 */

static ssize_t ParallelPort_write (struct file *file,
    /* The buffer to be filled with the data */
    const char *buffer,
    /* The length of the buffer */
    size_t length,
    /* offset value to the file */
    loff_t *offset)
{

```

```

    char value = *buffer;

#ifdef DEBUG
    printk ("device_write(%p,%s,%d)",
           file, buffer, length);
#endif

/*
 * To set the clock for the 74LS374 positive-edge triggered flip-flops, set the
 * control register C0 = logic "0".
 */
    outb(CTRL(0x0E),BASE_ADDRESS+2);

/*
 * Send the Data to the parallel port BASE_ADDRESS.
 */
    outb((unsigned char)value, BASE_ADDRESS);

/* To latch the data in the positive-edge triggered flip-flops, write logic
 * high to control register C0, which outputs the data. Since, C1 was set to
 * logic "1" the Octal tri state buffer will be disabled; this will make sure
 * that only the output data is in the bus.
 */
    outb(CTRL(0x0F),BASE_ADDRESS+2);
    return 0;
}

/*
 * This "file_operations" structure will be called whenever the processor
 * try to do some operation with the device. The ParallelPort_fops is a
 * pointer, which points to a table of operations such as open, read, write, etc.
 * This can contain NULL pointer for unsupported functions.
 */

struct file_operations ParallelPort_fops = {
    NULL,          /* lseek - default */
    ParallelPort_read, /* read */
    ParallelPort_write, /* write */
    NULL,          /* readdir - default */
    NULL,          /* select - default*/
    NULL,          /* ioctl - default */
    NULL,          /* mmap - default*/
    ParallelPort_open, /* open */
    NULL,          /* flush - default*/
    ParallelPort_close /* close */
};

/*
 *Initialize the module - Registers the character device.
 */
int init_module()
{

```

```

    int reg_value;
/*
 *Registers the character "MAJOR_NUM" for the device.
 */
    reg_value = register_chrdev(MAJOR_NUM,DEVICE_NAME,&ParallelPort_fops);
    if (reg_value < 0)
    {
/*
 * if the MAJOR_NUM returned a negative value, error message will be printed on
 * screen.
 */
        printk ("The character device is not registered %d \n",reg_value);
        return reg_value;
    }

/* if the MAJOR_NUM is successfully registered, then the following messages
 * will be displayed
 */
    printk("The Registration was successful\n");
    printk("The Major device number is %d\n",MAJOR_NUM);
    printk("In order to communicate from the USER MODE \n");
    printk("to the device driver,\n");
    printk("The following command has to be run on root access \n");
    printk ("mknod /dev/ppt c %d 0\n", MAJOR_NUM);
    return 0;
}

/* This function will be called when a request is made to remove
 * the module from the kernel. Once it is removed it will no longer
 * be listed under /proc/module directory.
 */
void cleanup_module()
{
int num;
/*
 *if the device is not registered, the proper message will be printed
 */
num = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
if(num < 0)
    printk("ERROR: The character DEVICE is not registered \n");
else
    printk("Kernel module Successfully removed \n");
}

```

Appendix C

The Tcl/Tk code for the GUI

```
#####
#
#           Tcl/Tk Graphical User Interface programming           #
#
#
#####
#
#
#   Copyright Notice
#   =====
#   Copyright (C) by Vasuthevan Maheswaran April 2000
#
#   file ref: tcl_tk.tcl
#
# This Tcl/Tk widget is used to send receive data to the running processor.
#
# INPUTS: Digital input from the printf("%d",Data);
#
# OUTPUT: Digital value from the scaled widget
# *****

# Specifying the path and name of the executable to be called.

#!/root/vasu/final
# main window
wm title . ExecLog

# Create a frame for buttons and entry.
frame .top -borderwidth 20
pack .top -side top -fill x

# Create the command buttons.

button .top.quit -text Quit -command exit

pack .top.quit -side right
# command to create the scale widget
scale .top.scale -from 0 -to 255 -length 500 -variable x \
```

```

        -orient horizontal -label "Data to the parallel port" \
        -tickinterval 15 -showvalue true
# create the scale with pack
pack .top.scale
# binding the scale widget with a mouse button release
bind .top.scale <ButtonRelease> Run

# set the executable in /home/vasu/mod/final to variable 'ppt'.
set ppt "/home/vasu/mod/final"

# label for the header
label .top.l -text "The Data received form parallel port:" -padx 0
entry .top.cmd -width 5 -relief sunken \
    -textvariable common
pack .top.l -side left
pack .top.cmd -side left

# Procedure to run the C program and arrange to read its input.
# This procedure starts the program specified in the 'ppt' variable.
# The 'ppt' is run in a pipeline so that it executes in the -
# - background. The leading | in the argument to open indicates that -
# - a pipeline is created.

proc Run {} {
    global ppt input log x
    if [catch {open "|$ppt $x |& cat"} input] {
        $log insert end $input\n
    } else {
        fileevent $input readable Log
    }
}

# Procedure to read and log output from the program. It also -
# - set the text variable 'common' to be the output value -
# - from the program so that it will be displayed in the GUI.

proc Log {} {
    global input log common
    if [eof $input] {
        Stop
    } else {
        gets $input line
        set common $line
    }
}

# Procedure to stop the program and to close the input/output -
# - variables. This procedure terminates the program by closing -
# - the pipeline

proc Stop {} {

```

```
    global input
    catch {close $input}
}
```

Appendix D

The cost for the Design project

Quantity	Description	Price
1	ADC0804 8bit μ P Compatible AD converter	\$4.20
1	DAC0804 8bit D/A Converter	\$2.20
1	74LS244 Octal 3STATE Buffer	\$1.25
1	74LS374 3STATE Octal DType Flip-Flops	\$1.25
1	7905 -5Volt Power regulator	\$0.75
1	L324 +5Volt Power regulator	\$0.75
1	L165 Power operational amplifier	\$2.35
1	LM741 General purpose operational amplifier	\$0.75
10	μ F Capacitance	\$1.50
1	150pF Capacitance	\$1.50
2	3V DC motor	\$2.00
1	PCB board	\$ 10.00
1	parallel port cable	\$14.00
1	wires	\$4.00
4	Resistor	\$0.60
2	Pot Resistor	\$3.50
	Total	\$50.60

Table D.1: Parts Price List

Appendix E

Dats Sheets