Parallel port programming under Linux environment
Mini guide

Parallel port is another type of circuit built on most desktop PC today. This interface circuit allow developer to easily interface with the outside world through standard 25 PIN connector. In order to start working with this parallel port circuit under linux environment, it is a good idea to have general understanding and some basic io port programing building block for this inteface circuit. Therefore, this mini guide have been developed in hope that most beginner can start working wtih this type of inteface.

This guide has 2 sections. First section provides basic introduction and programming for parallel port and second section will provide custom device driver development example with target Linux kernel 2.6.11.
It is interesting to note that linux kernel 2.4.x and linux kernel 2.6.x share some similality feature for low level hardware programing. However, there are some different feature such as pragma and make file which have to be handled correctly and most official hand book for linux hardware low level programing does not provide source code targeted for kernel 2.6.x. Therefore it is a good idea to provide basic programing example and **make file** for kernel 2.6.x.

Section1: Introduction to parallel port programming under linux OS

The parallel port, sometime people call printer port, was originally designed to support connection interface between PC and printer device. Over last decade, there are may external device that have been developed to interface with parallel port. These include scanner, removable device, data acquisition card, and medical equipment etc.
Nowaday, we can still find this parallel port connector on most of desktop PC. Physically, parallel port is 25 pin connector which provides 8 pins for bidirectional data lines, 4 pins for control lines and 5 pin for status input lines, and the less pin are all ground. This all together form a parallel port which can be accessed through a DB-25 femal connector. Basically, there are four different type of parallel port. These are

1. standard parallel port (SPP)
2. simple bidirectional port
3. enhanced parallel port (EPP)
4. extended capailities port (ECP)

Parallel port is originally designed to support only basic input output line seperately which data and control lines are used for output and status lines are used for input. Later, there are 3 more different standard that have been developed to provide higher communication speed and allow bidirectional use of data lines. For complete detail of parallel port standard, please refer to www.fapo.com/1284int.htm.
Most of modern computer now can be configured to operated in all of these 4 mode. By using bios setup, we can alter mode operation of parallel port.
To get start working with parallel port, SPP mode is selected for this mini guide. It is not the fastest operation mode but simple enough for beginer. For other different mode please refer to website: www.lvr.com.

Parallel port Characterisitic
Figure 1 illustrates picture of DB-25 connector

Figure 1. illustrates DB-25 connector, DB-25 cable and DB-25 breakout board

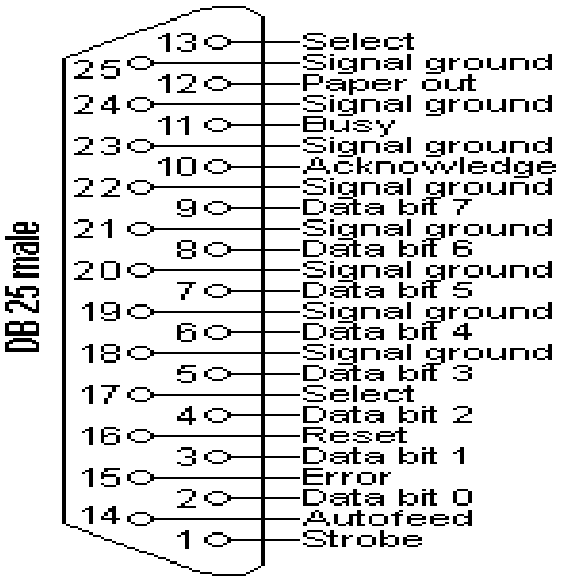


figure 2 ilustrates pinout of parallel port.

Generally, parallel port circuit are directly designed to talk with PC mother board through x86 ISA bus at I/0 address 0x378. In addtion, When writing C program, we can read and write this parallel port via 3 software registers. These are Data register, Status register, and control register located at io addres 0x378, 0x378+1, and 0x378+2 respectively. By writing or reading from these 3 registers we can input and output digital logic to the outside world. For SPP mode, we can write DATA and Control register to output logic signal. In addtion, we can read from Status register to obtain logic input from the real world. In particular, all of 3 hardware register have been designed to connected with I/O bus of mother board and output-input buffer of parallel port circuit. When writing data byte to address 0x378 (data register), PC will generate proper control signal to to transfer data byte to output buffer on parallel port device. In turn, data byte on output buffer will be later converted into electrical logic signal in which 5V is corresponding to bit =1 and 0V for bit =0. All of this steps is taken place within microsecond. It is important to know exact information about pin mapping of parallel port that associate with each bit of those data registers.

Table 1 provides information of relationship for PIN number, signal name and associated data register.

Pin No (D-Type 25)	Pin No (Centronics)	SPP Signal	Direction In/out	Register	Hardware Inverted
1	1	nStrobe	In/Out	Control	Yes
2	2	Data 0	Out	Data	
3	3	Data 1	Out	Data	
4	4	Data 2	Out	Data	
5	5	Data 3	Out	Data	
6	6	Data 4	Out	Data	
7	7	Data 5	Out	Data	
8	8	Data 6	Out	Data	
9	9	Data 7	Out	Data	
10	10	nAck	In	Status	
11	11	Busy	In	Status	Yes
12	12	Paper-Out / Paper-End	In	Status	
13	13	Select	In	Status	
14	14	nAuto-Linefeed	In/Out	Control	Yes
15	32	nError / nFault	In	Status	
16	31	nInitialize	In/Out	Control	
17	36	nSelect-Printer / nSelect-In	In/Out	Control	Yes
18 - 25	19-30	Ground	Gnd		

Table 1 lists particular DB-25 pin associated 3 data register.

Programing parallel port under linux

I/O port operations with linux is simple. There are 2 basic comand to work with. First command is **inb()** and second is **outb()**. The following example illustrates how to perform input and output operation for parallel port in SPP mode.

Exam 1.

```
//file Name : Ex1.c
#include <asm/io.h>
#include <stdio.h>
void main()
{ char a;                                   //store 8 bit data
        if(ioperm(0x378,3,1)) exit(1);      //obtain hardware i/o access permission
                                            //at io address 0x378
                                            //kernel. If fail, exit
        outb(0x0F,0x378)                    //write 1111 1111 to data line
                                            //where 1 represent logic high
        a = inp(0x378+1)                    //read data from input status register
                                            //at address 0x379
        print("Hello parallel port\n");

}
```

From above example, header file "asm/io.hJ" is needed for basic I/O access for linux. For output operation, we use **outb** command with first argument for data logic to be written and second argument for data register.

For input operation, we can call **inp** command with specified i/o address of status register. Then logic value from outside world will be stored in to 'a' variable. Notice that this example is simple way to interface with parallel port. However, there is some disadvantage with this scheme. First, programmer need to keep track of individual bit of these 3 registers. Secondly, there are possiblity that multiple bash script file will take control over these registers at the same time. One script may set first bit of data register while the other script clear first bit of data register. This somehow, script need to be able to store and recall the last written byte value of registers. To overcome this problem, custom device driver programing will be addressed in the section 2.

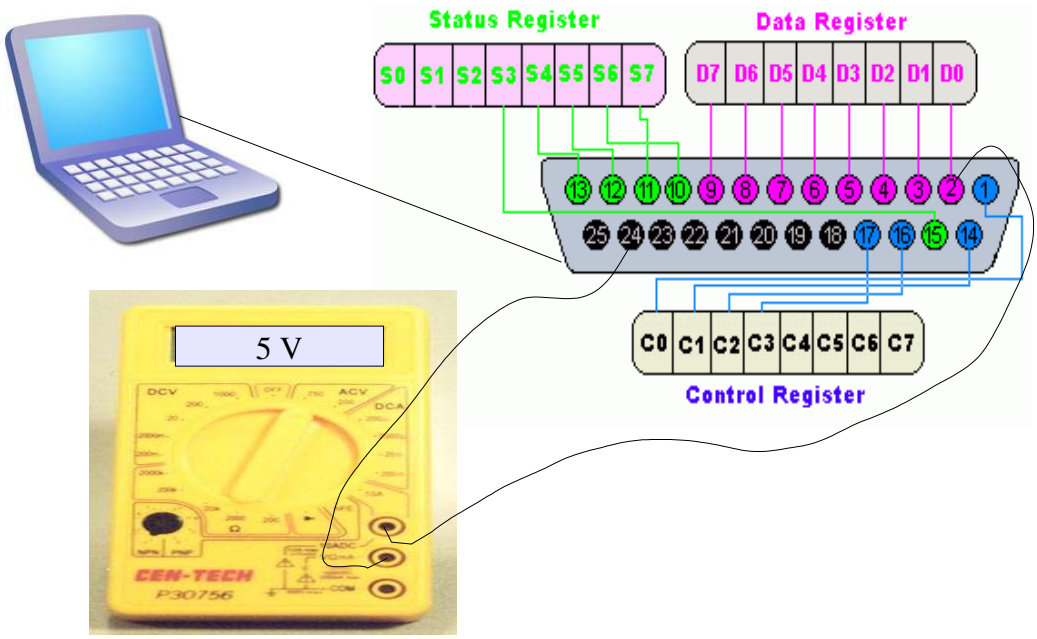
Code compilation

Before compiling any C code under linux, please make sure that there is GNU suit avialable in your system. Most of linux distribution have already provide this basic feature for basic installation. However, developer may install GNU suit by themself using tarball or RPM file.

Code compilation of example 1

>> gcc -o ex1 ex1.c

This will produce ex1 executable file. When running this program, console will show message “Hello parallel port”. This example also send out 0x0F to data register. This mean that all data lines signal will have logic 1 signal at output. To observe this signal, we can use osciloscope or simple multi meter to measure voltage signal of this data lines as depicted in figure 3.



Section 2: Custom device driver programing

Linux operating system allow us to develop custom hardware device driver to run in kernel space to provide following benefit.

1. driver can be load and unload using loadable modules mechanisim.
2. driver can be accessed through file operation.
3. device driver operation is performed throught file operation in /dev directory.
4. device driver can exist in /proc directory.
5. device drive shield develop from interface logic specifics.
6. /proc directory device fiels are created on-the-fly and do not collide with other device file.

In order to understand basic understanding of custom device driver, please refer to helloworld example's source code provided in appendix A. For more specific detail of custom device driver programing, please refer to
http://www.xml.com/ldd/chapter/book/

Compilation and Test process

1. For kernel 2.6.x , please make sure that content of the make file is defined as follow.

Make file

```
#
# Makefile for hello.c file
#
KDIR:=/lib/modules/$(shell uname -r)/build

obj-m:=hello.o

default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
	$(RM) .*.cmd *.mod.c *.o *.ko -r .tmp*
```

2. compile this device driver using follow command

>> make

This will create hello.ko

3. insert module

login as root and then type following command

>> insmod ./hello.ko

Check for greeting message in log message

>> tail /var/log/messages

You will result similar to following message

Aug 3 09:13:42 SmileBox kernel: Hello World Man !

4. remove module

>> rmmod hello.ko

This will generate folllowing message

Aug 3 09:18:48 SmileBox kernel: Good bye Dear friend!

However, This example does not incoporate process file concept. In order to read and write to normal process file. We need to incorporate process handler feature inside helloworld example. For this, h_proc example source code is provided in appendix A. For complete detail of process file programing example please refer to http://www.xml.com/ldd/chapter/book/

After insert h_proc using insmod command. We can observed this module using following command

>> ls /proc

The list will include module **myParallel2** in the process file's list

We can read and write to process file using cat and echo command respectively. Example

>> cat /proc/myParallel2

This generates message

“Data buffer is 1234”

>> echo “567” > /proc/parallel port

This will write 567 to this process's file buffer

>> cat /proc/myParallel2

this generates message

“Data buffer is 567”

From this example we can then apply this method to keep track of what last byte word were written to parallel port. This also make sure that we also obtained current status of each bit of 3 registers.

Appendix A.

Hello world device driver source code

```
// file name: hello.c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>



#define MODULE_VERSION "0.1"
#define MODULE_NAME  "MyParallel"

//====================================/
// Standard Parallel Port Definition  /
//====================================/
#define SPPDATAPORT     0x378
#define SPPSTATUSPORT  (SPPDATAPORT +1)
#define SPPCONTROLPORT (SPPDATAPORT +2)



MODULE_LICENSE("Dual BSD/GPL");

static int __init hello_init(void)
{ int i;
  unsigned char v;
        printk( "Hello World Man !\n");
        for (i=0;i<10;i++){ }
        v = inb(SPPDATAPORT);

return 0;
}

static void __exit hello_exit(void)
{
printk( "Good bye Dear friend!\n");
}

module_init(hello_init);
module_exit(hello_exit);
//EXPORT_NO_SYMBOL
```

Source code file myParallel.c

```
//Author : Anan Osothsilp
//Date : 7/10/05
//Device driver for parallel port Module
//---------------------------------------



#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define MODULE_VERSION "1.0"
#define MODULE_NAME "helloworld proc module"


#define HW_LEN 8
struct h_data_t{
	char value[HW_LEN+1];
};


// Process file Pointer
static struct proc_dir_entry *h_file;


// Data Variable
struct h_data_t h_data;


//procedure: Read
static int proc_read_hworld(char *page,char **start, off_t off, int count, int *eof, void
*data)
{
	int len;

	//cast conversion from data --> h_data
	struct h_data_t *h_data = (struct h_data_t *)data;
```

```
        len = sprintf(page, "Data buffer is %s\n",h_data->value);
        return len;

}

//procedure: Write
static int proc_write_hworld(struct file *file, const char *buffer, unsigned long count,
void *data)
{
        int len;
        struct h_data_t *h_data = (struct h_data_t *)data;

        if(count > HW_LEN)
                len = HW_LEN;
        else
                len = count;

        if(copy_from_user(h_data->value,buffer,len))
        {
                return -EFAULT;
        }

        h_data->value[len] = '\0';

        return len;
}

//Procedure: Initialization
static int __init init_helloworld(void)
{
        int rv =0;

        //create proc entry and make it readable by all 0666
        h_file = create_proc_entry("myParallel2",0666,NULL);
        if(h_file == NULL)
        {
                return -ENOMEM;
        }

        //set default value of data to Flyer
        strcpy(h_data.value, "1234");
```

```
	//set process file field bit

	h_file->data		=	&h_data;
	h_file->read_proc	=	&proc_read_hworld;
	h_file->write_proc	=	&proc_write_hworld;
	//h_file->owner		=	THIS_MODULE;

	printk(KERN_INFO "%s %s initialized\n", MODULE_NAME,
MODULE_VERSION);
	return 0;
}

//Procedure: Exit
static void __exit cleanup_helloworld(void)
{
	remove_proc_entry("myParallel2",NULL);
	printk(KERN_INFO "%s %s
remove\n",MODULE_NAME,MODULE_VERSION);
}

module_init(init_helloworld);
module_exit(cleanup_helloworld);
//MODULE_AUTHOR("Andy 14");
//MODULE_DESCRIPTION("Helo w proc");

EXPORT_NO_SYMBOLS;
```

This short guide is developed by Anan Osothsilp
email: anan14@siu.edu

Please feel free to use for any project. If modifcation were made to this original document, please also notify me or send me a copy. I would appreciate for that. Also there are more list of topic that may be useful for this guide. This include hardware circuit diagram and list of real world application using parallel port under linux environment, and GUI development for parallel port interfacing. For the time of writing this hand guide, those topic are not included. However, you may expect the new version in the near future.