

Part I: Hardware Interfaces

Lesson1: Parallel Port

Lesson2: Serial Port

Lesson3: USB Devices

Part I I: Software Interfaces

Lesson4: Java GUI

Lesson5: Windows Forms

Lesson6: Java Web Services

Lesson7: C# Web Services

Hardware Interfaces

Lesson 1

Parallel Port

Lesson1: Parallel Port

1. Introduction

PC parallel port can be very useful I/O channel for connecting your own circuits to PC. The PC's parallel port can be used to perform some hardware interfacing experiments. The port is very easy to use after you first understand some basic tricks. This paper tries to show those tricks in an easy to understand way. You can see the parallel port connector in the real panel of your PC. It is a 25 pin female (DB25) connector (to which printer is connected). In some PCs only one parallel port is present, but you can add more by buying and inserting ISA/PCI parallel port cards. In modern personal computers is present only the USB for peripheral devices connection. The pin outs of DB25 connector are shown in Figure 1.

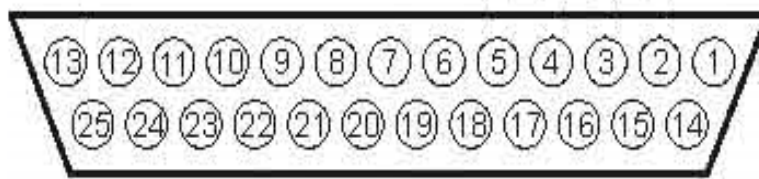


Figure 1: DB25 connector

The lines in DB25 connector are divided in to three groups, they are:

- *Data lines (data bus)*
- *Control lines*
- *Status lines*

Lesson1: Parallel Port

As the name refers, data is transferred over data lines, Control lines are used to control the peripheral and of course, the peripheral returns status signals back computer through Status lines. These lines are connected to Data, Control and Status registers internally. The details of parallel port signal lines are given in Table 1.

Table 1:Pin Assignments of the D-Type 25 pin Parallel Port Connector

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

- <http://www.beyondlogic.org/spp/parallel.htm#1>

2. Parallel port registers

The Data, Control and status lines are connected to their corresponding registers inside the computer. So by manipulating these registers in program, one can easily read or write to parallel port with programming languages like C.

The registers found in standard parallel port are:

- *Data register*
- *Status register*
- *Control register*

As their names specify, Data register is connected to Data lines, Control register is connected to control lines and Status register is connected to Status lines. (Here the word connection does not mean that there is some physical connection between data/control/status lines. The registers are virtually connected to the corresponding lines.). So whatever you write to these registers, will appear in corresponding lines as voltages. And whatever you give to Parallel port as voltages can be read from these registers (with some restrictions). For example, if we write '1' to Data register, the line Data0 will be driven to +5v. Just like this, we can programmatically turn on and off any of the data lines and Control lines.

In an IBM PC, these registers are IO mapped and will have unique address. We have to find these addresses to work with parallel port. For a typical PC, the base address of LPT1 is 0x378 and of LPT2 is 0x278. The

Lesson1: Parallel Port

data register resides at this base address, status register at baseaddress + 1 and the control register is at baseaddress + 2. So once we have the base address, we can calculate the address of each registers in this manner. The Table 2 shows the register addresses of LPT1 and LPT2, Figure 2, shows the signals for each port.

Table 2: Port Addresses

Register	LPT1	LPT2
data registrar(baseaddress + 0)	0x378	0x278
status register (baseaddress + 1)	0x379	0x279
control register (baseaddress + 2)	0x37a	0x27a

Port	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Date	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0
Stare	/Busy	Ack	PE	Select	Error	x	x	X
Control	x	x	X	IRQEn	Selln	/Init	Auto	Strobe

Figure 2: Ports Signals

3. Programming parallel port in C under Linux

Routines for accessing I/O ports are in `/usr/include/asm/io.h` (or `linux/include/asm-i386/io.h` in the kernel source distribution). The routines there are inline macros, so it is enough to `#include <asm/io.h>`.

- ***Permissions:***

Before you access any ports, you must give your program permission to do so. This is done by calling the `ioperm()` function (declared in `unistd.h`, and defined in the kernel) somewhere near the start of your program (before any I/O port accesses). The syntax is `ioperm(from, num, turn_on)`, where `from` is the first port number to give access to, and `num` the number of consecutive ports to give access to. For example, `ioperm(0x300, 5, 1)` would give access to ports `0x300` through `0x304` (a total of 5 ports). The last argument is a Boolean value specifying whether to give access to the program to the ports (true (1)) or to remove access (false (0)). You can call `ioperm()` multiple times to enable multiple non-consecutive ports.

The `ioperm()` call requires your program to have root privileges; thus you need to either run it as the root user. You can drop the root privileges after you have called `ioperm()` to enable the ports you want to use. You are not required to explicitly drop your port access privileges with `ioperm(..., 0)` at the end of your program; this is done automatically as the process exits.

Lesson1: Parallel Port

- *Accessing the ports:*

To input a byte (8 bits) from a port, call `inb(port)`, it returns the byte it got. To output a byte, call `outb(value, port)`. To input a word (16 bits) from ports `x` and `x+1` (one byte from each to form the word, using the assembler instruction `inw`), call `inw(x)`. To output a word to the two ports, use `outw(value, x)`. If you're unsure of which port instructions (byte or word) to use, you probably want `inb()` and `outb()`; most devices are designed for byte wise port access. Note that all port access instructions take at least about a microsecond to execute.

- *Example:*

```
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378 /* lp1 */

int main()
{
    /* Get access to the ports */
    ioperm(BASEPORT, 3, 1);

    /* Set the data signals (D0-7) of the port to all low (0) */
    outb(0, BASEPORT);

    /* Sleep for a while (1 s) */
    sleep(1);

    /* Read from the status port (BASE+1) and display the result */
    printf("status: %d\n", inb(BASEPORT + 1));
}
```

Lesson1: Parallel Port

```
/* We don't need the ports anymore */  
ioperm(BASEPORT, 3, 0);  
}
```

- <http://www.faqs.org/docs/Linux-mini/IO-Port-Programming.html>

4. Class work

Ex1: Using the device presented in Figure 3, create an application for monitoring the parallel ports:

- Green Led – data port
- Red Led – status port
- Yellow Led – control port

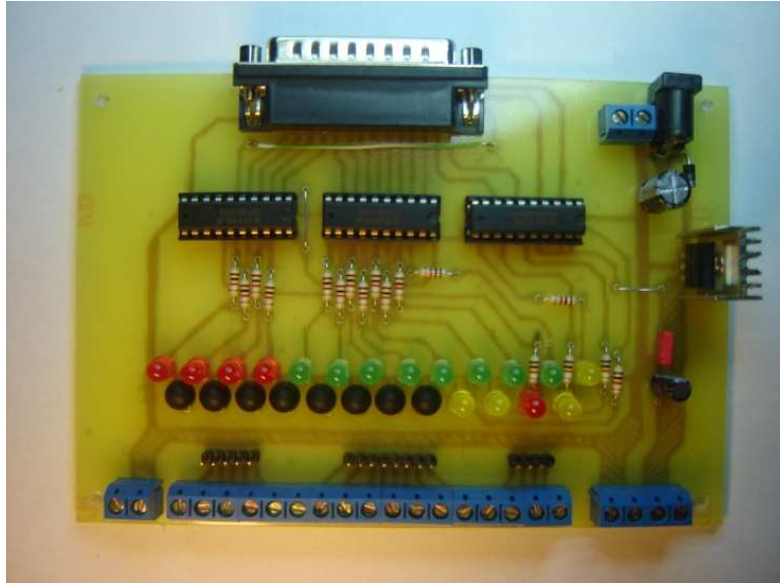


Figure 3: Parallel port interfacing device

The electrical circuit is presented in Figure 4.

Lesson1: Parallel Port

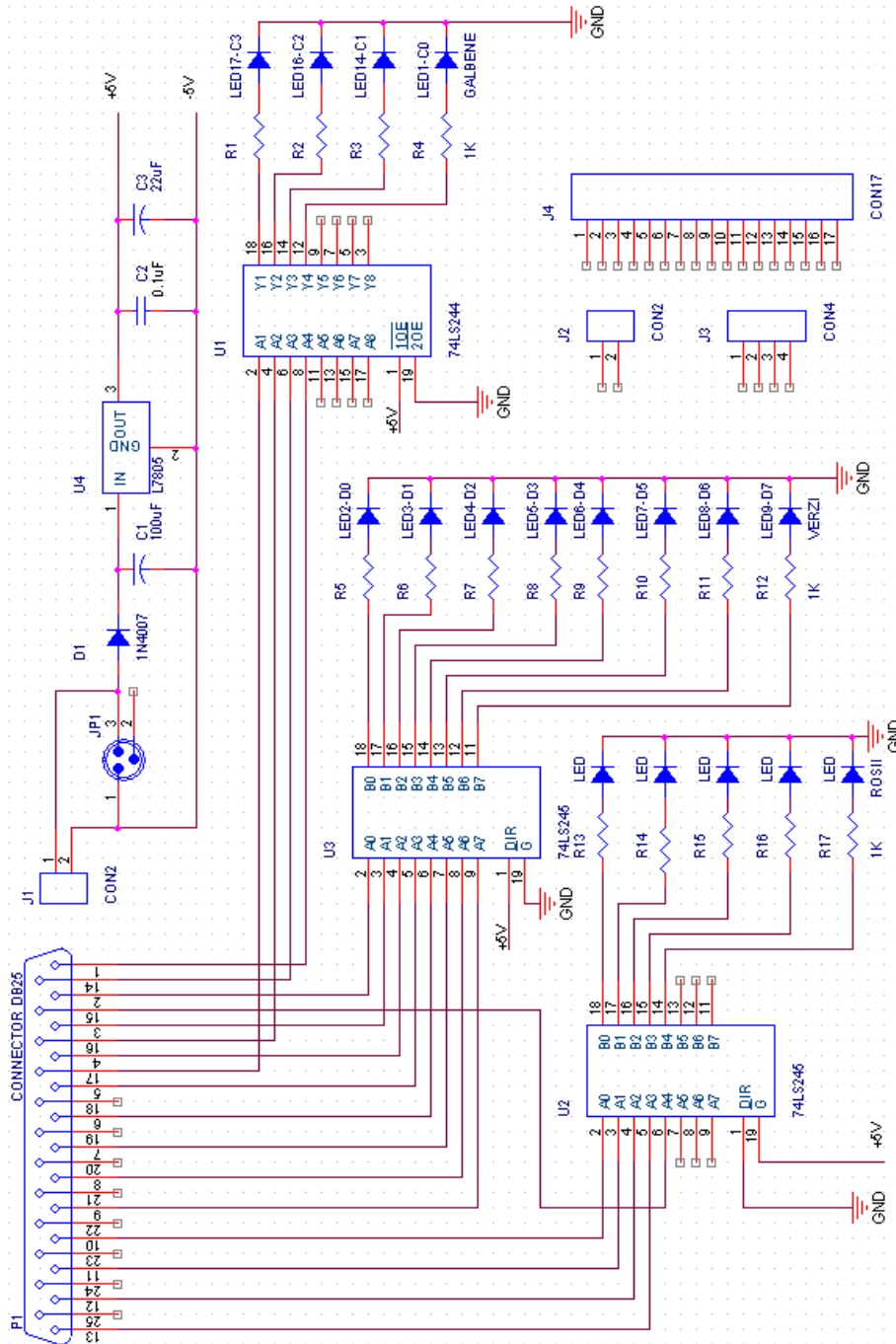


Figure 4: Electrical schema for parallel ports interfacing device

Lesson1: Parallel Port

Ex2: Using the device presented in Figure 5, create an application which will command the 8 segments display.



Figure 5: 8-segment display

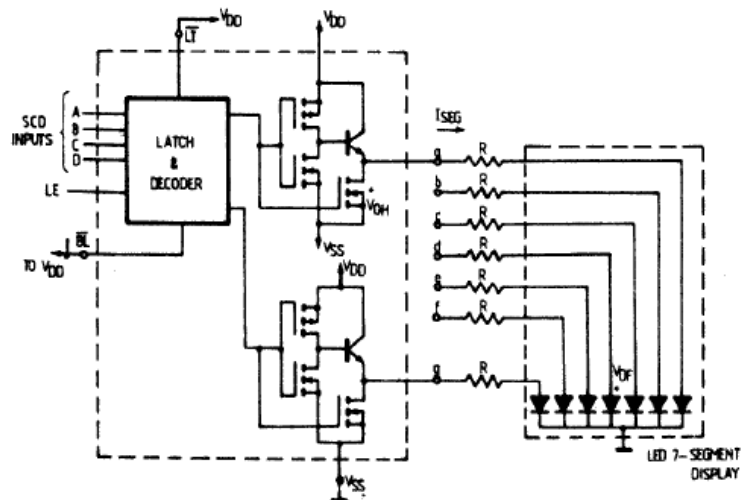


Figure 6: Electrical schema

Lesson1: Parallel Port

D	C	B	A	a	b	c	d	e	f	g	Display
X	X	X	X	1	1	1	1	1	1	1	8
X	X	X	X	0	0	0	0	0	0	0	Blank
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	0	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	0	0	1	1	9
1	0	1	0	0	0	0	0	0	0	0	Blank
1	0	1	1	0	0	0	0	0	0	0	Blank
1	1	0	0	0	0	0	0	0	0	0	Blank
1	1	0	1	0	0	0	0	0	0	0	Blank
1	1	1	0	0	0	0	0	0	0	0	Blank
1	1	1	1	0	0	0	0	0	0	0	Blank
X	X	X	X	0	0	0	*	0	0	0	*

Figure 7: Control table

Ex3: Using the didactic device creates an application, which will turn on/off the led's like in Figure 8.











Sequence	Led_1	Led_2	Led_3	Led_4
1		x	x	
2	x		x	
3	x	x		
4				

Figure 8: Led's sequence

Lesson1: Parallel Port

The signals used for the led's connection are presented in Figure 9.

Wire Color	Red	Blue	Purple	Yellow	Black	Green
Pin	2	3	4	5	6	20
Signal	D0	D1	D2	D3	D4	GND

Figure 9: Led's signals

Ex4: Using the didactic device presented in Figure 10, create an application to control barrier movements. The electrical circuit is presented in Figure 11.



Figure 10: Barrier didactic device

Lesson1: Parallel Port

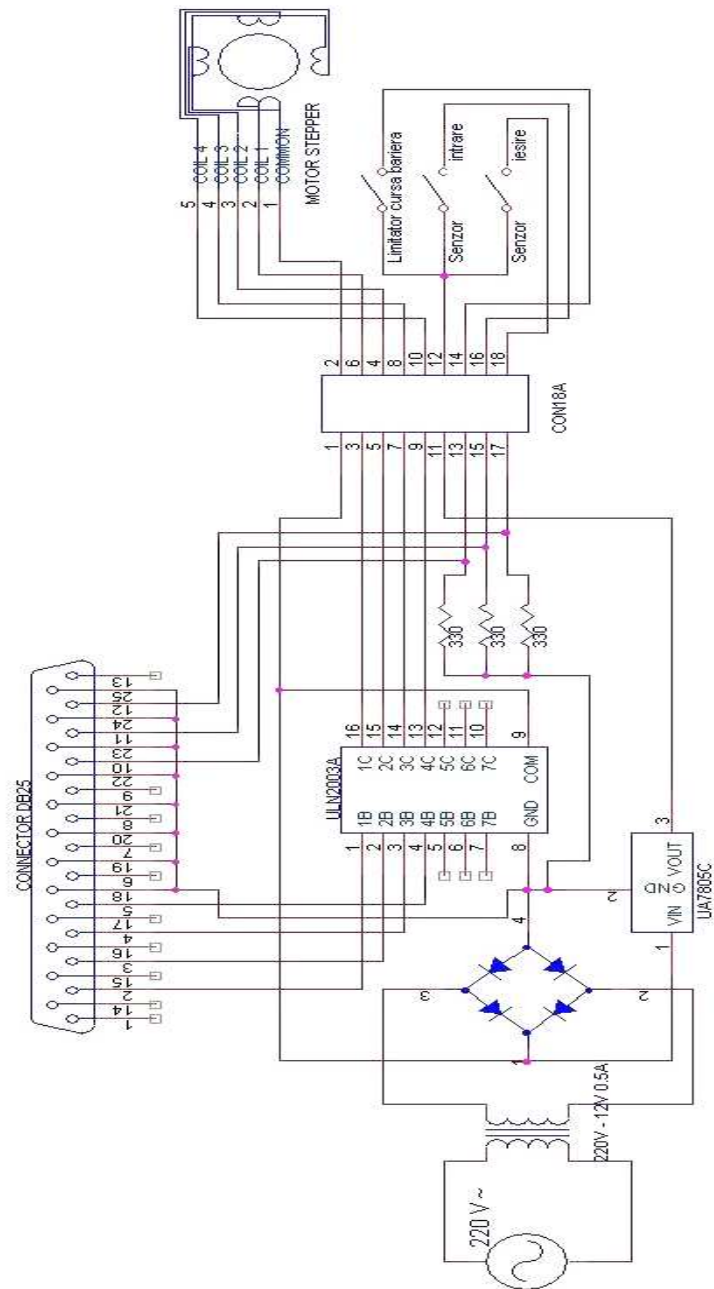


Figure 11: Barrier electrical schema

5. Home work

Ex1: Create a GUI monitoring application for parallel ports.

Lesson 2

Serial Port

Lesson2: Serial Port

1. Introduction

In computing, a serial port is a serial communication physical interface through which information transfers in or out one bit at a time (in contrast to the parallel port). Throughout most of the history of personal computers, data transfer through serial ports connected the computer to devices such as terminals and various peripherals.

In modern personal computers the serial port has largely been replaced by USB and Firewire for connections to peripheral devices. Many modern personal computers do not have a serial port since this legacy port has been superseded for most uses. Serial ports are commonly still used in applications such as industrial automation systems, scientific analysis and some industrial and consumer products. Server computers may use a serial port as a control console for diagnostics. Network equipment (such as routers and switches) often use serial console for configuration. Serial ports are still used in these areas as they are simple, cheap and their console functions are highly standardized and widespread. A serial port requires very little supporting software from the host system.

The electrical specifications of the serial port is contained in the EIA (Electronics Industry Association) RS232C standard. It states many parameters such as:


- Logic 0 - will be between +3 and +25 Volts.
- Logic 1 - will be between -3 and -25 Volts.
- The region between +3 and -3 volts is undefined.

Lesson2: Serial Port

- An open circuit voltage should never exceed 25 volts. (In Reference to GND).
- A short circuit current should not exceed 500mA. The driver should be able to handle this without damage.


The RS-232D has existed in two types: D-TYPE 25(Table 1) pin connector and D-TYPE 9(Table 2) pin connector, which are male connectors on the back of the PC. You need a female connector on your communication from Host to Guest computer.

Table 1: 25 Pin Connector on a DTE device (PC connection)

Male RS232 DB25	
Pin Number	Direction of signal
1	Protective Ground
2	Transmitted Data (TD) Outgoing Data (from a DTE to a DCE)
3	Received Data (RD) Incoming Data (from a DCE to a DTE)
4	Request To Send (RTS) Outgoing flow control signal controlled by DTE
5	Clear To Send (CTS) Incoming flow control signal controlled by DCE
6	Data Set Ready (DSR) Incoming handshaking signal controlled by DCE
7	Signal Ground Common reference voltage
8	Carrier Detect (CD) Incoming signal from a modem
20	Data Terminal Ready (DTR) Outgoing handshaking signal controlled by DTE
22	Ring Indicator (RI) Incoming signal from a modem

Lesson2: Serial Port

Table 2: Pin Connector on a DTE device (PC connection)

Male RS232 DB9	
Pin Number	Direction of signal
1	Carrier Detect (CD) (from DCE) Incoming signal from a modem
2	Received Data (RD) Incoming Data from a DCE
3	Transmitted Data (TD) Outgoing Data to a DCE
4	Data Terminal Ready (DTR) Outgoing handshaking signal
5	Signal Ground Common reference voltage
6	Data Set Ready (DSR) Incoming handshaking signal
7	Request To Send (RTS) Outgoing flow control signal
8	Clear To Send (CTS) Incoming flow control signal
9	Ring Indicator (RI) (from DCE) Incoming signal from a modem

- <http://www.beyondlogic.org/serial/serial.htm>

2. Serial Port's Registers (PC's)

- *Port Addresses(Table 3)*

Table 3: Standard Port Addresses

Name	Address	IRQ
COM 1	3F8	4
COM 2	2F8	3
COM 3	3E8	4
COM 4	2E8	3

Lesson2: Serial Port

- *Table of Registers(Table 4)*

Table 4: Table of Registers

Base Address	DLAB	Read/Write	Abr.	Register Name
+ 0	=0	Write	-	Transmitter Holding Buffer
	=0	Read	-	Receiver Buffer
	=1	Read/Write	-	Divisor Latch Low Byte
+ 1	=0	Read/Write	IER	Interrupt Enable Register
	=1	Read/Write	-	Divisor Latch High Byte
+ 2	-	Read	IIR	Interrupt Identification Register
	-	Write	FCR	FIFO Control Register
+ 3	-	Read/Write	LCR	Line Control Register
+ 4	-	Read/Write	MCR	Modem Control Register
+ 5	-	Read	LSR	Line Status Register
+ 6	-	Read	MSR	Modem Status Register
+ 7	-	Read/Write	-	Scratch Register

- *Used Baudrate Divisors(Table 5)*

Table 5: Table of Commonly Used Baudrate Divisors

Speed (BPS)	Divisor (Dec)	Divisor Latch High Byte	Divisor Latch Low Byte
50	2304	09h	00h
300	384	01h	80h
600	192	00h	C0h
2400	48	00h	30h
4800	24	00h	18h
9600	12	00h	0Ch
19200	6	00h	06h
38400	3	00h	03h
57600	2	00h	02h

Lesson2: Serial Port

115200	1	00h	01h
--------	---	-----	-----

- *Interrupt Enable Register (Table 6)*

Table 6: Interrupt Enable Register

Bit	Notes
Bit 7	Reserved
Bit 6	Reserved
Bit 5	Enables Low Power Mode (16750)
Bit 4	Enables Sleep Mode (16750)
Bit 3	Enable Modem Status Interrupt
Bit 2	Enable Receiver Line Status Interrupt
Bit 1	Enable Transmitter Holding Register Empty Interrupt
Bit 0	Enable Received Data Available Interrupt

- *Interrupt Identification Register (Table 7)*

Table 7: Interrupt Identification Register

Bit	Notes		
Bits 6 and 7	Bit 6	Bit 7	
	0	0	No FIFO
	0	1	FIFO Enabled but Unusable
	1	1	FIFO Enabled
Bit 5	64 Byte Fifo Enabled (16750 only)		
Bit 4	Reserved		
Bit 3	0	Reserved on 8250, 16450	
	1	16550 Time-out Interrupt Pending	
Bits 1	Bit	Bit	

Lesson2: Serial Port

and 2	2	1	
	0	0	Modem Status Interrupt
	0	1	Transmitter Holding Register Empty Interrupt
	1	0	Received Data Available Interrupt
	1	1	Receiver Line Status Interrupt
Bit 0	0	Interrupt Pending	
	1	No Interrupt Pending	

- *First In / First Out Control Register (Table 8)*

Table 8: FIFO Control Register

Bit	Notes		
Bits 6 and 7	Bit 7	Bit 6	Interrupt Trigger Level
	0	0	1 Byte
	0	1	4 Bytes
	1	0	8 Bytes
	1	1	14 Bytes
Bit 5	Enable 64 Byte FIFO (16750 only)		
Bit 4	Reserved		
Bit 3	DMA Mode Select. Change status of RXRDY & TXRDY pins from mode 1 to mode 2.		
Bit 2	Clear Transmit FIFO		
Bit 1	Clear Receive FIFO		
Bit 0	Enable FIFO's		

Lesson2: Serial Port

- *Line Control Register (Table 9)*

Table 9: Line Control Register

Bit 7	1	Divisor Latch Access Bit		
	0	Access to Receiver buffer, Transmitter buffer & Interrupt Enable Register		
Bit 6	Set Break Enable			
Bits 3, 4 And 5	Bit 5	Bit 4	Bit 3	Parity Select
	X	X	0	No Parity
	0	0	1	Odd Parity
	0	1	1	Even Parity
	1	0	1	High Parity (Sticky)
	1	1	1	Low Parity (Sticky)
Bit 2	Length of Stop Bit			
	0	One Stop Bit		
	1	2 Stop bits for words of length 6,7 or 8 bits or 1.5 Stop Bits for Word lengths of 5 bits.		
Bits 0 And 1	Bit 1	Bit 0	Word Length	
	0	0	5 Bits	
	0	1	6 Bits	
	1	0	7 Bits	
	1	1	8 Bits	

3. Programming serial port in C under Linux

Routines for accessing I/O ports are in `/usr/include/asm/io.h` (or `linux/include/asm-i386/io.h` in the kernel source distribution). The routines there are inline macros, so it is enough to `#include <asm/io.h>`.

- ***Permissions***

Before you access any ports, you must give your program permission to do so. This is done by calling the `ioperm()` function (declared in `unistd.h`, and defined in the kernel) somewhere near the start of your program (before any I/O port accesses). The syntax is `ioperm(from, num, turn_on)`, where `from` is the first port number to give access to, and `num` the number of consecutive ports to give access to. For example, `ioperm(0x300, 5, 1)` would give access to ports `0x300` through `0x304` (a total of 5 ports). The last argument is a Boolean value specifying whether to give access to the program to the ports (true (1)) or to remove access (false (0)). You can call `ioperm()` multiple times to enable multiple non-consecutive ports.

The `ioperm()` call requires your program to have root privileges; thus you need to either run it as the root user. You can drop the root privileges after you have called `ioperm()` to enable the ports you want to use. You are not required to explicitly drop your port access privileges with `ioperm(..., 0)` at the end of your program; this is done automatically as the process exits.

Lesson2: Serial Port

- *Accessing the ports:*

To input a byte (8 bits) from a port, call `inb(port)`, it returns the byte it got. To output a byte, call `outb(value, port)`. To input a word (16 bits) from ports `x` and `x+1` (one byte from each to form the word, using the assembler instruction `inw`), call `inw(x)`. To output a word to the two ports, use `outw(value, x)`. If you're unsure of which port instructions (byte or word) to use, you probably want `inb()` and `outb()` --- most devices are designed for byte-wise port access. Note that all port access instructions take at least about a microsecond to execute.

- *Example*

```
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>
#define port 0x3F8
/* Defines Serial Ports Base Address */

/* COM1 0x3F8 */
/* COM2 0x2F8 */
/* COM3 0x3E8 */
/* COM4 0x2E8 */
int main()
{
    char c='a';
    outb(0, port + 1); /* Turn off interrupts - Port1 */
    /* port - Communication Settings */
    outb(0x80, port + 3); /* SET DLAB ON */
    outb(0x03, port + 0); /* Set Baud rate - Divisor Latch Low Byte */
    /* Default 0x03 = 38,400 BPS */
    /* 0x01 = 115,200 BPS */
    /* 0x02 = 57,600 BPS */
    /* 0x06 = 19,200 BPS */
    /* 0x0C = 9,600 BPS */
    /* 0x18 = 4,800 BPS */
    /* 0x30 = 2,400 BPS */
}
```

Lesson2: Serial Port

```
outb(0x00, port + 1); /* Set Baud rate - Divisor Latch High Byte */
outb(0x03, port + 3); /* 8 Bits, No Parity, 1 Stop Bit */
outb(0xc7, port + 2); /* FIFO Control Register */
outb(0x0B, port + 4); /* Turn on DTR, RTS, and OUT2 */
printf("\nSample Comm's Program\n");
outb(c, port); /* Send Char to Serial Port */
}
```

- <http://www.faqs.org/docs/Linux-mini/IO-Port-Programming.html>

4. Class work

Ex1: Create a cable connection like in Figure 1.

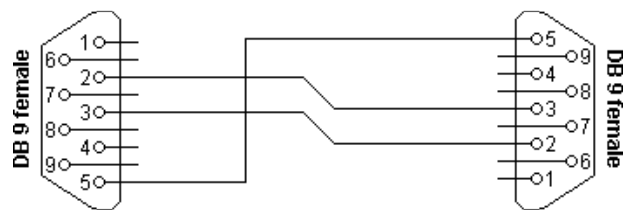


Figure 1: Simple cable connection

Ex2: Create an application that send the char ,a' form PC1, to PC2, and display the char on the PC2 console.

Ex3: Create an application that send a string (the string will be read from the input) form PC1, to PC2, and display the string on the PC2 console.

Lesson2: Serial Port

Ex4: Create an application that send an int, bigger than 255, form PC1, to PC2, and display the int on the PC2 console.

5. Home Work

Ex1: Create an application for a bidirectional communication, where each PC1 and PC2 can transmit and receive data.

Lesson 3

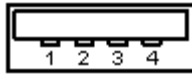
USB Devices

1. Introduction

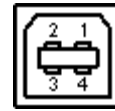
USB is the most successful personal-computer interface ever. Every recent PC has USB ports that can connect to keyboards, mice, game controllers, scanners, cameras, printers, drives, and more. USB is reliable, fast, versatile, power-conserving, inexpensive, and supported by major operating systems. USB is a likely solution any time you want to use a computer to communicate with an external device. Internal devices, such as fingerprint readers, can use USB as well. The interface is suitable for mass-produced, consumer devices as well as specialized, small-volume products and one-of-a-kind projects. To be successful, an interface has to please two audiences: the users who want to use the devices and the developers who design the hardware and write the code that communicates with the devices. USB has features to please both groups.

All USB devices have an upstream connection to the host and all hosts have a downstream connection to the device. Upstream and downstream connectors are not mechanically interchangeable, thus eliminating illegal loopback connections at hubs such as a downstream port connected to a downstream port. There are commonly two types of connectors, called type A and type B which are shown below.

Lesson3: USB Devices



Type A USB
Connector



Type B USB
Connector

Type A plugs always face upstream. Type A sockets will typically find themselves on hosts and hubs. For example type A sockets are common on computer main boards and hubs. Type B plugs are always connected downstream and consequently type B sockets are found on devices.

The maximum length of a standard USB cable (for USB 2.0 or earlier) is 5.0 metres.

Table 1: USB 1.x/2.0 cable wiring

Pin	Name	Cable color	Description
1	VCC	Red	+5 V
2	D-	White	Data –
3	D+	Green	Data +
4	GND	Black	Ground

USB communication takes the form of packets. Initially, all packets are sent from the host, via the root hub and possibly more hubs, to devices. Some of those packets direct a device to send some packets in reply. All packets are made of 8-bit bytes, transmitted least-significant bit first. The

Lesson3: USB Devices

first byte is a packet identifier (PID) byte. The PID is actually 4 bits; the byte consists of the 4-bit PID followed by its bitwise complement. This redundancy helps detect errors. (Note also that a PID byte contains at most four consecutive 1 bits, and thus will never need bit-stuffing, even when combined with the final 1 bit in the sync byte. However, trailing 1 bits in the PID may require bit-stuffing within the first few bits of the payload.)

Communication on the USB network can use any one of four different data transfer types:

- ***Control transfers:*** These are short data packets for device control and configuration, particularly at attach time;
- ***Bulk transfers:*** These are data packets in relatively large quantities. Devices like scanners or SCSI adapters use this transfer type;
- ***Interrupt transfers:*** These are data packets that are polled periodically. The host controller will automatically post an interrupt at a specified interval;
- ***Isochronous transfers:*** These are data streams in real time with higher requirements for bandwidth than for reliability. Audio and video devices generally use this transfer type;

Like a serial port, each USB port on a computer is assigned a unique identification number (port ID) by the USB controller. When a USB device is attached to a USB port, this unique port ID is assigned to the device and the *device descriptor* is read by the USB controller. The device descriptor includes information that applies globally to the device, as well as information on the *configuration* of the device. A configuration defines the functionality and I/O behavior of a USB device. A USB device may have

Lesson3: USB Devices

one or more configurations, which are described by their corresponding configuration descriptors. Each configuration has one or more *interfaces*, which can be considered as a physical communication channel. Each interface has zero or more endpoints, which can be either data providers or data consumers, or both. Interfaces are described by interface descriptors, and endpoints are described by end-point descriptors. Furthermore, a USB device might also have string descriptors to provide additional information such as vendor name, device name, or serial numbers.

- <http://www.usb.org/>
- <http://www.beyondlogic.org/usbnutshell/usb2.htm>

2. Accessing USB devices from Java applications

Most general-purpose operating systems provide support for USB devices, and it is relatively easy to develop applications in C or C++ that access such peripherals. However, the Java programming language by design provides very little support for hardware access, so writing Java applications that interact with USB devices has proved quite difficult.

The jUSB project was created by Mojo Jojo and David Brownell in June 2000. Its objective was to provide a set of free software Java APIs to access USB devices on Linux platforms. The API is distributed under the Lesser GPL (LGPL), which means that you can use it in proprietary as well as free

Lesson3: USB Devices

software projects. The API provides multithreaded access to multiple physical USB devices, and supports both native and remote devices. Devices with multiple interfaces can be accessed by multiple applications (or device drivers) simultaneously, with each application (or device driver) claiming a different interface. The API supports control transfers, bulk transfers, and interrupt transfers. Currently, the API works on GNU/Linux distributions with either the Linux 2.4 kernel or a back port into 2.2.18 kernel. For example, the API works on Red Hat 7.2 and 9.0 without any patches or other upgrades.

The jUSB API includes the following packages:

- `usb.core`: This package is the core part of the jUSB API. It allows Java applications to access USB devices from USB hosts;
- `usb.linux`: This package contains a Linux implementation of a `usb.core.Host` object, bootstrapping support, and other classes leveraging Linux USB support. This implementation accesses the USB devices through the virtual USB device file system (`usbdevfs`);
- `usb.windows`: This package has a Windows implementation of a `usb.core.Host` object, bootstrapping support, and other classes leveraging Windows USB support. This implementation is still in its very early stage;
- `usb.remote`: This package is a remote version of the `usb.core` API. It includes an RMI proxy and a daemon application, which allow Java applications to access USB devices on a remote computer;

Lesson3: USB Devices

- `usb.util`: This package provides some useful utilities to download firmware to USB devices, dump the content of the USB system into XML, and convert a USB device with only bulk I/O into a socket;
- `usb.devices`: This optional package collects Java code to access a variety of USB devices with the jUSB API, including Kodak digital cameras and Rio 500 MP3 Players. These APIs were specially written to simplify the process of accessing the designated USB devices and cannot be used to access other devices. The APIs were built upon the `usb.core` APIs, and they will work on any operating system where jUSB is supported;
- `usb.view`: This optional package provides a simple USB tree browser based on Swing;

Although the implementation of the `usb.core`. Host object varies from operating system to operating system, a Java programmer needs to understand only the `usb.core` package to start developing applications with the jUSB APIs. Table 2 outlines the interfaces and classes from `usb.core` with which a Java programmer should be familiar.

Table 2: Interfaces and classes in jUSB

Interface	Description
Bus	Connects a set of USB devices to a Host
Host	Represents a USB controller with one or more Buses
Class	Description
Configuration	Provides access to a USB configuration supported by a device and to the interfaces associated with that configuration
Descriptor	Base class for entities with USB typed descriptors
Device	Provides access to a USB device

Lesson3: USB Devices

DeviceDescriptor	Provides access to a USB device descriptor
EndPoint	Provides access to a USB end-point descriptor, structuring device data input or output in a given device configuration
HostFactory	Contains bootstrapping methods
Hub	Provides access to a USB hub descriptor and some hub operations
Interface	Describes sets of endpoints, and is associated with a particular device configuration
PortIdentifier	Provides stable string identifiers for USB devices, appropriate for use in operations and troubleshooting

The normal procedure to access a USB device with the jUSB API is as follows:

- Bootstrap by getting the USB Host from the HostFactory;
- Access the USB Bus from the Host, then access the USB root hub (which is a USB Device) from the Bus;
- Obtain the number of USB ports available on the hub, and traverse through all the ports to find the appropriate Device;
- Access the USB Device that is attached to a particular port. A Device can be accessed directly from the Host with its PortIdentifier, or can be found by traversing the USB Bus starting from the root hub;
- Interact with the Device directly with ControlMessage, or claim an Interface from the current Configuration of the Device and perform I/O with the Endpoint available on the Interface;

The next example illustrates how to obtain the content of a USB system with the jUSB API. The program as written simply looks at the root hub for

Lesson3: USB Devices

available USB devices, but it would be easy to improve it to traverse the whole USB tree.

```
import usb.core.*;

public class ListUSB
{
    public static void main(String[] args)
    {
        try
        {
            // Bootstrap by getting the USB Host from the HostFactory.
            Host host = HostFactory.getHost();

            // Obtain a list of the USB buses available on the Host.
            Bus[] bus = host.getBusses();
            int total_bus = bus.length;

            // Traverse through all the USB buses.
            for (int i=0; i<total_bus; i++)
            {
                // Access the root hub on the USB bus and obtain the
                // number of USB ports available on the root hub.
                Device root = bus[i].getRootHub();
                int total_port = root.getNumPorts();

                // Traverse through all the USB ports available on the
                // root hub. It should be mentioned that the numbering
                // starts from 1, not 0.
                for (int j=1; j<=total_port; j++)
                {
                    // Obtain the Device connected to the port.
                    Device device = root.getChild(j);
                    if (device != null)
                    {
                        // USB device available, do something here.
                    }
                }
            }
        } catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Lesson3: USB Devices

The next example illustrates how to perform bulk I/O with Interface and EndPoint, assuming that the application has successfully located the Device.

```
if (device != null)
{
    // Obtain the current Configuration of the device and the number of
    // Interfaces available under the current Configuration.
    Configuration config = device.getConfiguration();
    int total_interface = config.getNumInterfaces();

    // Traverse through the Interfaces
    for (int k=0; k<total_interface; k++)
    {
        // Access the currently Interface and obtain the number of
        // endpoints available on the Interface.
        Interface itf = config.getInterface(k, 0);
        int total_ep = itf.getNumEndpoints();

        // Traverse through all the endpoints.
        for (int l=0; l<total_ep; l++)
        {
            // Access the endpoint, and obtain its I/O type.
            Endpoint ep = itf.getEndpoint(l);
            String io_type = ep.getType();
            boolean input = ep.isInput();

            // If the endpoint is an input endpoint, obtain its
            // InputStream and read in data.
            if (input)
            {
                InputStream in;
                in = ep.getInputStream();
                // Read in data here
                in.close();
            }
            // If the Endpoint is and output Endpoint, obtain its
            // OutputStream and write out data.
            else
            {
                OutputStream out;
                out = ep.getOutputStream();
                // Write out data here
                out.close();
            }
        }
    }
}
```

Lesson3: USB Devices

```
    {
        OutputStream out;
        out = ep.getOutputStream();
        // Write out data here.
        out.close();
    }
}
```

The release of the API, version 0.4.4, was made available on February 14, 2001. Only some minor progress has been reported since that time, probably due to the success of the IBM group in becoming a candidate extended standard of the Java language. However, several third-party applications have been developed based on jUSB, including the JPhoto project (an application using jUSB to connect to digital cameras) and the jSyncManager project (an application using jUSB to synchronize with a Palm OS-based PDA). The objective of the project was to develop a USB interface for the Java platform that will allow full access to the USB system for any Java application or middleware component. The JSR-80 API provides full support for all four transfer types defined by the USB specification. Currently, the Linux implementation of the API works on GNU/Linux distributions with 2.4 kernel support, such as Red Hat 7.2 and 9.0.

The JSR-80 project includes three packages: javax-usb (the javax.usb API), javax-usb-ri (the common part of the OS-independent reference implementation), and javax-usb-ri-linux (the reference implementation for the Linux platform, which connects the common reference implementation

Lesson3: USB Devices

to the Linux USB stack). All three parts are required to form a complete functioning java.usb API on the Linux platform.

Although the OS-dependent implementation of the JSR-80 APIs varies from operating system to operating system, a Java programmer needs to understand only the javax.usb package to start developing applications. Table 3 lists the interfaces and classes in javax.usb with which a Java programmer should be familiar.

Table 3: Interfaces and classes in the JSR-80 APIs

Interface	Description
UsbConfiguration	Represents a configuration of a USB device
UsbConfigurationDescriptor	Interface for a USB configuration descriptor
UsbDevice	Interface for a USB device
UsbDeviceDescriptor	Interface for a USB device descriptor
UsbEndpoint	Interface for a USB endpoint
UsbEndpointDescriptor	Interface for a USB endpoint descriptor
UsbHub	Interface for a USB hub
UsbInterface	Interface for a USB interface
UsbInterfaceDescriptor	Interface for a USB interface descriptor
UsbPipe	Interface for a USB pipe
UsbPort	Interface for a USB port
UsbServices	Interface for a javax.usb implementation
Class	Description
UsbHostManager	Entry point for javax.usb

The normal procedure for accessing a USB device with the JSR-80 API is as follows:

Lesson3: USB Devices

- Bootstrap by getting the appropriate `UsbServices` from the `UsbHostManager`;
- Access the root hub through the `UsbServices`. The root hub is considered as a `UsbHub` in the application;
- Obtain a list of the `UsbDevices` that are connected to the root hub. Traverse through all the lower-level hubs to find the appropriate `UsbDevice`;
- Interact with the `UsbDevice` directly with a control message (`UsbControlIrp`), or claim a `UsbInterface` from the appropriate `UsbConfiguration` of the `UsbDevice` and perform I/O with the `UsbEndpoint` available on the `UsbInterface`;
- If a `UsbEndpoint` is used to perform I/O, open the `UsbPipe` associated with it. Both upstream data (from the USB device to the host computer) and downstream data (from the host computer to the USB device) can be submitted either synchronously or asynchronously through the `UsbPipe`;
- Close the `UsbPipe` and release the appropriate `UsbInterface` when the application no longer needs access to the `UsbDevice`;

The next example obtain the content of the USB system with the JSR-80 API. The program recursively traverses through all the USB hubs on the USB system and locates all the USB devices connected to the host computer.

```
import javax.usb.*;  
import java.util.List;
```


Lesson3: USB Devices

```
public class TraverseUSB
{
    public static void main(String argv[])
    {
        try
        {
            // Access the system USB services, and access to the root
            // hub. Then traverse through the root hub.
            UsbServices services = UsbHostManager.getUsbServices();
            UsbHub rootHub = services.getRootUsbHub();
            traverse(rootHub);
        } catch (Exception e) {}
    }

    public static void traverse(UsbDevice device)
    {
        if (device.isUsbHub())
        {
            // This is a USB Hub, traverse through the hub.
            List attachedDevices =
                ((UsbHub) device).getAttachedUsbDevices();
            for (int i=0; i<attachedDevices.size(); i++)
            {
                traverse((UsbDevice) attachedDevices.get(i));
            }
        }
        else
        {
            // This is a USB function, not a hub.
            // Do something.
        }
    }
}
```

The next example illustrates how to perform I/O with Interface and EndPoint, assuming that the application has successfully located a Device. This code snippet can also be modified to perform I/O of all four data transfer types.

Lesson3: USB Devices

```
public static void testIO(UsbDevice device)
{
    try
    {
        // Access to the active configuration of the USB device, obtain
        // all the interfaces available in that configuration.
        UsbConfiguration config = device.getActiveUsbConfiguration();
        List totalInterfaces = config.getUsbInterfaces();

        // Traverse through all the interfaces, and access the endpoints
        // available to that interface for I/O.
        for (int i=0; i<totalInterfaces.size(); i++)
        {
            UsbInterface interf = (UsbInterface) totalInterfaces.get(i);
            interf.claim();
            List totalEndpoints = interf.getUsbEndpoints();
            for (int j=0; j<totalEndpoints.size(); j++)
            {
                // Access the particular endpoint, determine the direction
                // of its data flow, and type of data transfer, and open the
                // data pipe for I/O.
                UsbEndpoint ep = (UsbEndpoint) totalEndpoints.get(j);
                int direction = ep.getDirection();
                int type = ep.getType();
                UsbPipe pipe = ep.getUsbPipe();
                pipe.open();
                // Perform I/O through the USB pipe here.
                pipe.close();
            }
            interf.release();
        }
    } catch (Exception e) {}
}
```

Both the jUSB API and the JSR-80 API provide Java applications with the capability to access USB devices from a machine running the Linux operating system. The JSR-80 API provides more functionality than the jUSB API, and has the potential of becoming an extended standard of the Java language.

Lesson3: USB Devices

- <http://www.ibm.com/developerworks/library/j-usb.html>

3. Class work

Ex1: Test the examples presented in chapter 2.

Ex2: Create a simple USB LED Light (Figure 1). Turn the led ON, wait 2 seconds and turn the led OFF.

- male USB plug;
- 22 ohm resistor;
- LED;



Figure 1: USB Led

4. Home Work

Ex1: Make a Universal Serial Bus presentation: History; Overview; Device classes; USB mass-storage; Human-interface devices (HIDs); Signaling; Data packets; Handshake packets; Token packets; Connector properties; Protocol analyzers; Connector types; Comparisons with other device connection technologies;