2004

An Integration of PC Hardware & Software in Teaching Engineering Technology Courses

Steve Hsiung
Old Dominion University, shsiung@odu.edu

Richard Jones
Old Dominion University, rljones@odu.edu

An Integration of PC Hardware & Software in Teaching Engineering Technology Courses

Steve Hsiung,
Richard Jones
Engineering Technology Department
Old Dominion University
Norfolk, VA 23529

Abstract

As technology advances, the price of a PC drops dramatically. This trend has resulted in PCs that are complex, powerful, and very affordable. Today's PC is a popular and essential tool in teaching software programming course(s) in C, C++, Visual Basic, or Java, running commercial software supporting courses in circuit simulation/design or circuit board layout, and acting as a workstation to gain access to the Internet or LAN networks. In most Engineering Technology curricula there is a limited amount of linkage between those PC applications. The actual effort to merge the hard-gained knowledge of hardware & software concepts together through a useful project implementation is also rare. This article is aimed at using the PC in ET upper-level courses as a focal point to help to reinforce knowledge between different fields of interest, such as communication, automation control, microprocessor, software programming, and system integration.

I. Introduction

If the standard Engineering Technology (ET) curricula, especially in the Electrical and Electronic areas were examined, the first complication usually noted is how wide a span of fields the typical curriculum covers. There are: analog hardware circuit theories & designs, digital hardware circuit theories & designs, microprocessor/microcontroller designs, applications & programming, high-level software programming, communication related issues in designs & networking, and senior project designs. Along with the breadth of the programs they usually have little overlap between these various fields of interest.

When the graduate ET students get in the real work place, they are usually confronted with tasks which are usually a combination of some, and often many, of the curriculum fields they have learned in school. This article proposes to implement a course(s) aimed at integrating different fields of interest into a useful project oriented course(s). This addition to the curricula will assist students in their future project implementations and/or employment skills.

This integration has three major elements: (a) software programming, (b) hardware circuit, (c) and communication. It requires the students to have comprehensive experiences in all of the related fields before they can take this course(s). This unique feature would generally place this course(s) in ET curricula at the senior level.

Software programming skills are essential to the success of ET job competency. Data manipulation and control, and chip level communication, are usually covered in the microprocessor/microcontroller related course(s) in assembly language format. This form is not portable and the design and development are limited to specific manufactured products.

Non-ET departments usually teach higher-level programming classes such as in C, C++, VB, or Java. Normally, they focus on programming language styles, data base structure, or Internet orientation. There is typically very little linkage between those two software-programming approaches.

The programming section of this course(s) will focus on portability. This means that a high-level language format in a PC environment will be used to handle the issues of controlling bits and bytes in or out of the hardware circuit.

In order to integrate the experiments into a standard PC environment, it is necessary to place a hardware interface between the PC and the desired hardware circuit(s). A major part of this interface will be high-level language drivers, which are needed to gain access to the ports of the PC. There are various hardware circuit designs available to fulfill the needs of many different applications.

It is impossible to have a single hardware system cover all the control needs of the real world. To better assist as many needs as possible, communication is necessary for integration between specific hardware designs. Communication schemes can be either in serial or parallel format depending on the specifications. If it is a serial format then it can be either wired or wireless. In any form of communication, integrity and security are issues that need to be addressed. A well-defined protocol is especially important for the safety and reliability of wired or wireless communications.

II. Software Programming

Accessing a PC's I/O (Input/Output) ports in computer interfacing once was straightforward and simple using QBASIC or a C compiler stdio INP and OUT commands in the old DOS environment[3]. As PC technology hardware and software advanced, the simplicity of the PC's I/O port control was carried over into C++ compilers in the new format of _INP and _OUT routines[8,9]. Unfortunately, ever since Windows NT/2000/XP came on the scene and implemented the CPU protected mode, it has become very difficult for computer interfacing enthusiasts to gain direct control over the I/O ports of a PC.

One method of getting around the CPU protection mode within a PC when attempting to perform I/O through the parallel printer port, is to write an I/O routine (a kernel mode driver) that works in the Windows NT/2000/XP environment. This is a formidable task best left to those in the PC software industry driver business. The other choice is to find a kernel mode driver that is readily available for the public to use, and adapt it to the specific needs of the project. Fred Bulback has written a kernel mode driver called IO.DLL that is available for free from his web site: www.geekhideout.com[4]. After downloading the IO.DLL file, simply copy it into the C:\WINNT\system32 folder. The IO.DLL program has thirteen functions that can be called from VB6 (see Table 1[4]). The VB and C++ code needed to access IO.DLL and examples of how to use the IO.DLL kernel mode driver are given in the Appendix. VB6 is a popular programming language that is widely used in industry and academic applications. Its GUI (Graphic User Interface) format performs many of the same functions as other languages such as C++, but VB6 makes the building and packaging of icons much easier[10,17]. However, VB6 does not support the straightforward I/O commands that C++ or QBASIC do[3,8,18]. Nevertheless, when it comes to Windows NT/2000/XP, most of those I/O commands don't function anyway.

Table1. Visual Basic Functions from IO.DLL

Name of Function	Description of Function
PortOut	Outputs a byte to a port
PortWordOut	Outputs 16 bits to a port
PortDWordOut	Outputs 32 bits to a port
PortIn	Reads a byte from a port
PortWordIn	Reads in 16 bits from a port
PortDWordIn	Reads in 32 bits from a port
SetPortBit	sets a particular bit at a port
ClrPortBit	clears a particular bit at a port
NotPortBit	inverts a particular bit at a port
GetPortBit	Returns the state of a particular bit from a port
RightPortShift	shifts a port to the right, the LSB is returned and the value passed becomes the MSB
LeftPortShift	shifts a port to the left, the MSB is returned and the value passed becomes the LSB
IsDriverInstalled	Returns a non-zero value if IO.DLL is installed and functioning

III. Hardware Circuit

There are several ways to have a PC control and communicate with the outside world. An interface board that plugs in directly to PC ISA or PCI slot is one possible solution. This interface requires complex circuit design and it takes up the limited available space inside of a PC. Another disadvantage is that inserting/removing the board is quite often an issue to general users. This interface board would not be a possibility for the notebook user. The other alternative approach is to use a PC's available ports (parallel and serial) that are contently accessible outside of the PC's and notebook computer box[1,2].

Either one of the above mentioned methods requires buffering circuit protection designs. The only drawback of using external ports on a PC is the fixed addresses that are assigned by the PC system. The definition of the PC parallel port bits are summarized in Tables 2, 3, and 4[1].

A simple direct connection between the PC parallel pins to the outside world with transistors and an FET to drive high-power outputs as well as a buffering FET to aid the read external signal inputs is presented in Figure 1.

Table 2. PC Parallel Port Data Register: at Address (Base Address) = \$378, or 0X378, or 888

Pin #: DB25	Bit	Signal Name	Inverted?	Pin: Centronics	Direction
2	0	Data Bit 0	No	2	In/Out
3	1	Data Bit 1	No	3	In/Out
4	2	Data Bit 2	No	4	In/Out
5	3	Data Bit 3	No	5	In/Out
6	4	Data Bit 4	No	6	In/Out
7	5	Data Bit 5	No	7	In/Out
8	6	Data Bit 6	No	8	In/Out
9	7	Data Bit 7	No	9	In/Out

Table 3. PC Parallel Port Status Register: at Address (Base Address + 1) = $379, or 0X379, or 889

Pin #: DB25	Bit	Signal Name	Inverted?	Pin: Centronics	Direction
15	3	nError	No	32	In
13	4	Select	No	13	In
12	5	PaperEnd	No	12	In
10	6	nAck	No	10	In
11	7	Busy	Yes	11	In

Table 4. PC Parallel Port Control Registers: at Address (Base Address + 2) = $37A, or 0X37A, or 890

Pin #: DB25	Bit	Signal Name	Inverted?	Pin: Centronics	Direction
1	0	nStrobe	Yes	1	Out
14	1	nAutoLF	Yes	14	Out
16	2	nInit	No	31	Out
17	3	nSelectIn	Yes	36	Out

DOUT_0 to DOUT_7 are the output control signals from a PC parallel port and OUT0 to OUT7 are the buffered output control signals to the external circuit that are each capable of handling a 14A DC current. Only DOUT_0 to OUT0 are shown in the schematic; the rest are duplications. DIN_0 to DIN_4 are the input control signals and IN0 to IN4 are the buffered signals from the external circuit. IN0 to DIN_0 are shown in the schematic; IN1 to DIN_1, IN2 to DIN_2, and IN3 to DIN_3 are duplications. IN4 to DIN_4 are presented differently due to the inversed logic on the PC parallel port.

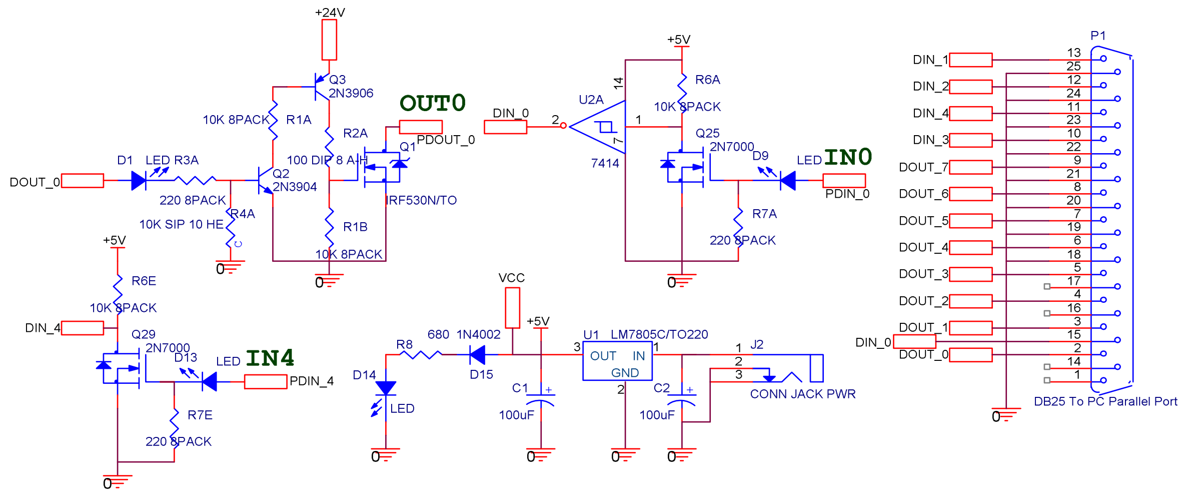

Figure 1. Straight Interface to the PC Parallel Port

There is a way to extend the limited available I/O mentioned above and still provide the user with a reasonable amount of controls that are suitable to most of the applications. The circuit design in Figure 2 uses a tri-state buffer, decoder, and a multiplexer as an interface between the PC parallel pins and the hardware to maximize the choice of I/O control up to 32 outputs and/or 37 inputs. The 74153 has dual 2-to-4 multiplexers that are used to select one of the four latches (CLK_0, CLK_1, CLK_2, or CLK_3) and/or tri-state buffers (Enable_0, Enable_1, Enable_2,or Enable_3) which are the U1 (74374) and U2 (74244) chips that need to be duplicated as shown in Figure 2.

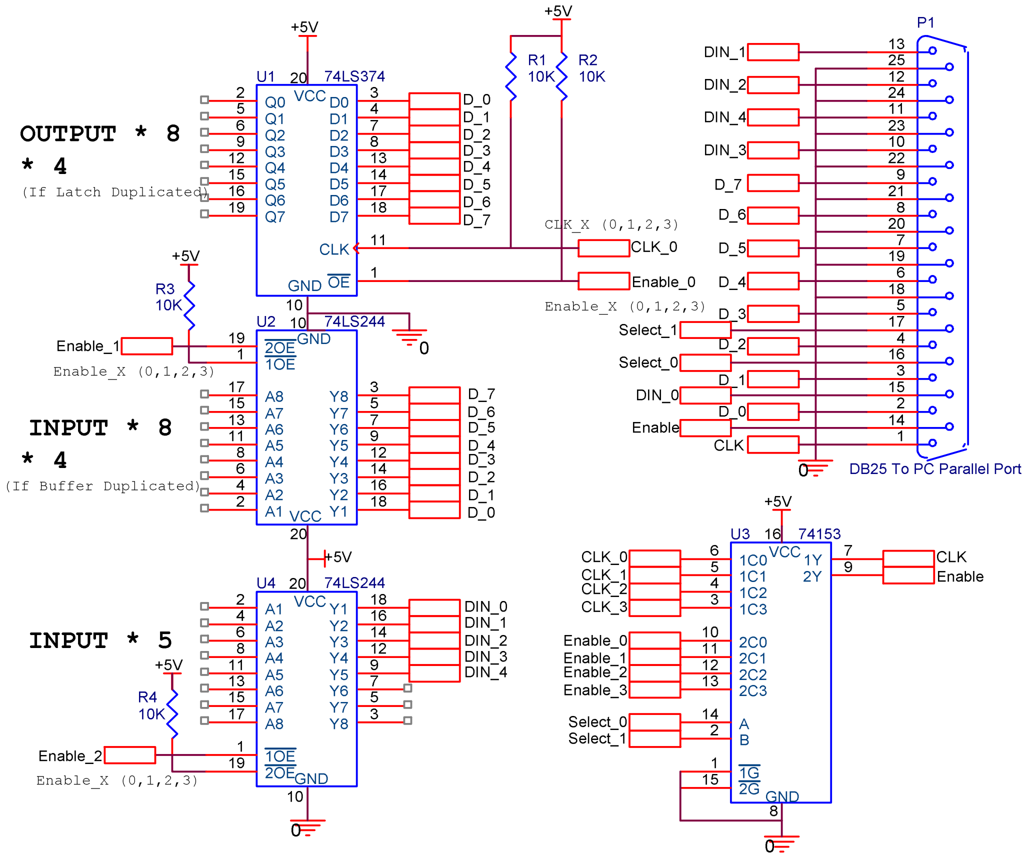

Figure 2. Extended Interface to the PC Parallel Port

The circuit design between the PC serial ports is aimed at converting a standard RS-232 signal into a general TTL compatible bit stream for chip level signal communications^2. This circuit in Figure 3 uses a versatile Max232 and its circuit is presented as follows. The transmit (TxD) and receive (RxD) lines are TTL compatible signals suitable for communications in the target hardware circuits.

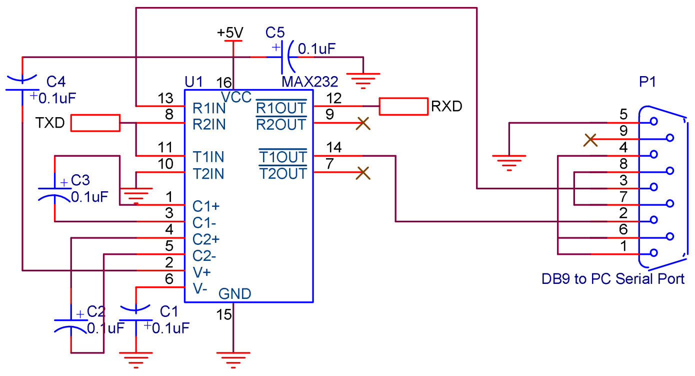

Figure 3. The PC Serial Port Interface

IV. Communication

Two PCs can communicate with each other through their serial or parallel port even when using different programming languages. The serial port communication can be easily achieved with the built-in communication module in VB6 with a small amount of additional coding[9,17]. The parallel port implementation can be achieved with bit banging in the C or C++ programming languages. The sample code is used to read/write a byte to/from the X25640 serial EEPROM shown in Appendix:

One possible exercise is the communication between a PC and a peripheral integrated circuit such as a serial EEPROM for a data logging application. The hardware and software implementation of this exercise is presented in Figure 4.

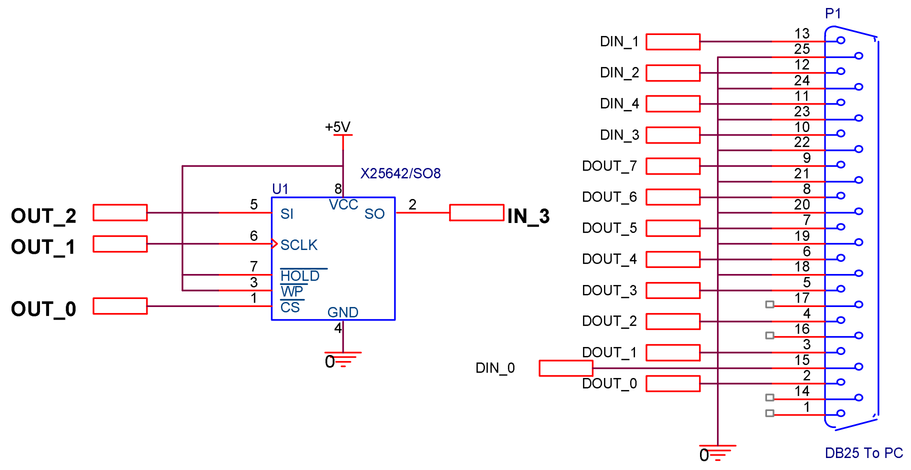

Figure 4. PC Parallel Port to Serial EEPROM Interface

There are defined instructions (Hex code) for this serial EEPROM to operate properly. Each read and write of up to 32 bytes should follow these instruction sequences to gain access to the information stored in the EEPROM. These instructions are presented in Table 5[16].

Table 5. EEPROM Command Byte

Instruction Name	Instruction Code	Operations
WREN	\$06 or 0X06	Enable Write Operation
WRDI	\$04 or 0X04	Disable Write Operation
RDSR	\$05 or 0X05	Read Status Register
WRSR	\$01 or 0X01	Write Status Register
READ	\$03 or 0X03	Read Data from Memory Array at Selected Address
WRITE	\$02 or 0X02	Write Data to Memory Array at Selected Address (1 to 32 Bytes)

Adding an RF communication module into the exercises gives the students wireless communication experiences. There are many RF modules available in the market. Tables 6, 7, & 8 present the commonly used modules[7,11,12,13,15].

The designer must make security a prime concern when implementing wireless control in a system. A well thought-out communication protocol is essential to the security solution.

Table 6. Available Transmitter Products

Part #	Freq.	Vcc	Icc		Baud Rate	Audio	Range	Price	Maker
RCT-433-AS	433.92 MHz	2-12 Vdc	5ma@3V	ASK/OOK	4800	No	100-300 ft.	$4.90	Radiotronix
TXE-315-KH	315 MHz	2.7-5.2 Vdc	3 ma		4800	No		$9.98	LINX
TXM-315-LC	315 MHz	2.7-5.2 Vdc	3 ma		4800	No		$6.90	LINX
TXLC-434	433.92 MHz			CPCA	5000	No	300 ft.	$14.95	Reynolds Electronics
TM1V	418 MHz	5 Vdc		On-Off and Pulse	4800	No	300 ft.	$16.40	GLOLAB
TLP-434	433.92 MHz	2-12 Vdc	5 ma	ASK	3000	No			Reynolds, LAIPAC Technology
TRF4900PW	850-950 MHz	2.2-3.6 Vdc	58 ma	FM/FSK	20 MHz Clock			$4.75	Texas Instruments
TH7107EFF	315/433 MHz			FSK/FM/ASK				$11.29	Melexis

Table 7. Available Receiver Products

Part #	Freq.	Vcc	Icc	Modulation	Baud Rate	Audio	Distance	Price	Maker
RCR-433-RP	433.92 MHz	5 Vdc	4.5 ma	ASK/00K	4800	Yes	100-300 ft.	$5.50	Radiotronix
RCR-433-HP	433.92 MHz	5 Vdc	4.5 ma	ASK/OOK	4800	No	300-800 ft.	$13.80	Radiotronix
RXM-315-LC	315 MHz	2.7-4.2 Vdc	6 ma		5000	No		$13.79	LINX
RXD-315-KH	315 MHz	2.7-4.2 Vdc	7 ma		4800	No		$15.93	LINX
RXLC-434	433.92 MHz	2.7-5.2 Vdc		CPCA	5000	No	300 ft.	$22.95	Reynolds Electronics
RM1V	418 MHz	5 Vdc		On-Off and Pulse	4800	No	300 ft.	$23.75	GLOLAB
RLP-4334	433.92 MHz	4.5-5.5 Vdc		ASK	3000	No			Reynolds, Laipac Technologies
TH71101ENE	315/433 MHz	5 Vdc		FSK/FM/ASK				$15.58	Melexis

Table 8. Available Transceiver Products

Part #	Freq.	Vcc	Icc	Modulation	Baud Rate	Audio	Applications	Price	Maker
EWM-900-FDTC-BS	902-928 MHz	3 Vdc	35ma Rx 25ma Tx	FM/FSK	19.2K	Yes	Full-Duplex Data & Audio 500-1000 ft.	$69.00	Radiotronix
EWM-900-FDTC-HS	902-928 MHz	3 Vdc	35ma Rx 25ma Tx	FM/FSK	19.2K	Yes	Full-Duplex Data & Audio 500-1000 ft.	$69.00	Radiotronix

Communication protocols are just a matter of the implementation of different rules in sending and receiving series of bits/bytes. Normally, there is a Start byte, Acknowledge (ACK) byte, Address byte, Command byte, Control Data byte, and Stop byte[14]. Which bytes are available depends on the definition of all the different types of bytes. One byte of data can have 254 different variations excluding all 0's (0X00 or $00) and all 1's (0XFF or $FF). So, the programmer can define one byte each for Start, Stop, and ACK. This leaves 251 bytes for Address and Command bytes. If only two bytes are used for the commands Read and Write, there will be 249 bytes available for unique addresses. This means that a total of 249 devices may be addressed.

For example:

Start byte = 0X01 or $01
Stop Byte = 0X02 or $02
ACK Byte = 0X03 or $03
Command Byte Read = 0X04 or $04
Command Byte Write = 0X05 or $05
Address Byte = Range from 0X06 or $06 to 0XFE or $FE
Control Data Byte = Range from 0X00 or $00 to 0XFF or $FF

It is possible to integrate the Read/Write option into a single bit in the address byte. Normally, this Read/Write bit is the last bit in the address byte where bit 7 = "0" means Write and bit 7 = "1" means Read. Using this strategy will limit available address bits to 7 bits and the total available addresses become 2^7, or 128 different devices. But this definition will provide 256 different command bytes for any specific application in mind.

For example:

Start byte = 0X01 or $01
Stop Byte = 0X02 or $02
ACK Byte = 0X03 or $03
Address Byte = Range from 0X80/0X80 (Read) to 0XFF/$FF (Read) or Range from $00/$00 (Write) to 0X7F/$7F (Write) = A Total of 128 Different Addresses for Read and 128 Different Addresses for Write
Command Byte = 0X00 or $00 – 0XFF or $FF = A Total of 256 Different Commands
Control Data Byte = 0X00 or $00 – 0XFF or $FF

Protocol Rules[14]**:**

1. Only a transmitter can send the Start and Stop bytes.
2. The receiver has to send/respond an ACK byte when a transmitter calls its address.
3. The addressed receiver has to send an ACK byte after every byte following the address byte.
4. There should be a defined time-out period (about 25 ms): Any byte sent by a transmitter shall expect an ACK byte from a receiver. There is a timeout period for the receiver to respond to an ACK byte. If the ACK byte from a receiver is not received, the transmitter shall terminate the communication by sending a Stop byte.
5. Any period after time-out period shall be considered to be a transition error. A new or repeated communication can start or initiate from a transmitter again.

The cost and security of the system will become very attractive when the above-mentioned protocols are implemented. As long as the designer makes sure that the rules are followed for software driven embedded control systems, there shouldn't be any problems.

We can elaborate on the protocol further by using CRC-8 to improve the integrity of the wireless communication. CRC-8 is called 8-bit Cyclic Redundancy Check that uses a generator polynomial ($G(X) = X^8 + X^2 + X^1 + 1$) to calculate each byte stream as a FCS (Frame Check Sequence)[5]. The way it works is that each transmitter will send an additional byte as a CRC-8 byte and the receiver will have to calculate its own CRC-8 byte after receiving the entire packet of information. The receiver needs to verify the received CRC-8 byte and calculated CRC-8 byte match; both bytes have to be identical to be considered as a valid transmission. If they are not, then the communication is treated as a failure[5,14]. The way the receiver signals this CRC-8 byte mismatch is by not sending an ACK command which causes a time-out condition to occur. Another communication has to be reestablished and everything has to start over again. This protocol implementation has the advantage of providing cleaner communications and eliminates most errors, but it also brings a heavy load on software coding and CPU execution time.

V. Conclusion

Teaching ET students should not be limited to providing them with the fundamental building blocks for their future career construction. In this ever changing technology era, we as educators, not only have to trigger students' interest in learning but also have to bring real-life applications into classes. Educating ET students with integrated concepts toward real world needs is the best way for students to gain employment skills.

The purpose to this article is to use a PC to perform hardware and software exercises. There could be various integrations between a standard PC environment and other fields of interest with the assistance of high level and low level assembly programming languages. These types of integrations could be incorporated into course(s) to lead students to their project designs and real world applications as well as to build interest in hardware programming methods. This integration provides the student with interesting concepts and a better understanding of the links between hardware and software along with their potential applications in the workplaces.

VI. Bibliographic

1. Axelson, J., “Parallel Port Complete”, Lakeview Research, 2000.
2. Axelson, J., “Serial Port Complete”, Lakeview Research, 2000.
3. Baumann, S. K. and Mandell, S. L., “QBasic”, West Publishing Company, 1992.
4. Bulback, F., “ IO.DLL” www.geekhideout.com, 2003.
5. CRC-8 Implementation White Paper, USAR System Inc., www.semtech.com, 1999.
6. Ekedahl, M. and Newman, W., “Visual Basic.NET: An Object-Oriented Approach”, Course Technology Thompson Learning, 2003.
7. EWM-900-FDTC Radiotronix Data Sheet, 1141 SE Grand Suite 118, Oklahoma City, OK 73129, www.radiotronix.com, 2000.
8. Horton, I., “Beginning Visual C++ 6”, Wrox Press, 1998.
9. Microsoft Visual Studio 6.0, www.microsoft.com/vstdio, 2003.
10. Perry, G. and Hettihewa, S., “Teach Yourself Visual Basic 6.0 in 24 Hours”, Sams Publishing, 1998.
11. RCR-433-RP Radiotronix Data Sheet, 1141 SE Grand Suite 118, Oklahoma City, OK 73129, www.radiotronix.com, 2001.
12. RCT-433-AS Radiotronix Data Sheet, 1141 SE Grand Suite 118, Oklahoma City, OK 73129, www.radiotronix.com, 2001.
13. RXM-315-LC-S, Linx Technologies Data Sheet, 575 SE Ashley Place, Grants Pass, OR 97526, www.linxtechnologies.com, 2001.
14. System Management Bus (SMBus) Specification, Revision 2.0, Smart Battery System Specifications,

www.sbs-forum.org , Email: battery@sbsforum.org, 2001.
15. TXM-315-LC Linx Technologies Data Sheet, 575 SE Ashley Pl., Grants Pass, OR 97526, www.linxtechnologies.com, 2001.
16. Xicor Product Specification, www.xicor.com/pdf_files/x25640.pdf, 2004.
17. Zak, D., "Visual Basic 6.0 Enhanced Edition", Course Technology Thomson Learning, 2001.
18. Zak, D., "Visual Basic.NET", Course Technology Thomson Learning, 2002.

VII. Biography

STEVE C. HSIUNG
Steve Hsiung is an associate professor of electrical engineering technology at Old Dominion University. Prior to his current position, Dr. Hsiung had worked for Maxim Integrated Products, Inc., Seagate Technology, Inc., and Lam Research Corp., all in Silicon Valley, CA. Dr. Hsiung also taught at Utah State University and California University of Pennsylvania. He earned his BS degree from National Kauhsiung Normal University in 1980, MS degrees from University of North Dakota in 1986 and Kansas State University in 1988, and PhD degree from Iowa State University in 1992.

RICHARD L. JONES
Richard Jones has been teaching at ODU since 1994. He is a retired United States Navy Submarine Service Lt. Commander with sub-specialties in Ballistic Missile, Torpedo, Sonar, and Radio systems. Richard has previously taught Mechanical Engineering Design at the United States Military Academy, West Point, N.Y., and Electrical Engineering at the United States Naval Academy, Annapolis, Md. He holds an ASEET from Cameron University, a BSEET from Oklahoma State University, and a Master of Engineering in Electronics Engineering from the Naval Postgraduate School at Monterey, California. Richard is currently focusing his research on methods of teaching accredited upper-level electronics labs via the internet.

VIII. Appendix

1. Visual Basic Code Needed to Access IO.DLL

```
Public Declare Sub IO_Out Lib "IO.DLL" Alias "PortOut" (ByVal Port As Integer, ByVal Data As Byte)
Public Declare Sub IO_WD_Out Lib "IO.DLL" Alias "PortWordOut" (ByVal Port As Integer, ByVal Data As Integer)
Public Declare Sub IO_DWD_Out Lib "IO.DLL" Alias "PortDWordOut" (ByVal Port As Integer, ByVal Data As Long)
Public Declare Function IO_In Lib "IO.DLL" Alias "PortIn" (ByVal Port As Integer) As Byte
Public Declare Function IO_WD_In Lib "IO.DLL" Alias "PortWordIn" (ByVal Port As Integer) As Integer
Public Declare Function IO_DWD_In Lib "IO.DLL" Alias "PortDWordIn" (ByVal Port As Integer) As Long
Public Declare Sub IO_Bit_Set Lib "IO.DLL" Alias "SetPortBit" (ByVal Port As Integer,  ByVal Bit As Byte)
Public Declare Sub IO_Bit_Clr Lib "IO.DLL" Alias "ClrPortBit" (ByVal Port As Integer,  ByVal Bit As Byte)
Public Declare Sub IO_Bit_Not Lib "IO.DLL" Alias "NotPortBit" (ByVal Port As Integer,  ByVal Bit As Byte)
Public Declare Function IO_Bit_Read Lib "IO.DLL" Alias "GetPortBit" (ByVal Port As Integer, ByVal Bit As Byte) As Boolean
Public Declare Function IO_Bit_RShift Lib "IO.DLL" Alias "RightPortShift" (ByVal Port As Integer, ByVal Val As Boolean)
As Boolean
Public Declare Function IO_Bit_LShift Lib "IO.DLL" Alias "LeftPortShift" (ByVal Port As Integer, ByVal Val As Boolean) As
Boolean
Public Declare Function I_Driver Lib "IO.DLL" Alias "IsDriverInstalled" () As Boolean
```

2. Examples in VB6 of how to use the IO.DLL to access the PC parallel port [17,18]

```
Dim PortAddress As Integer
Private Sub Combo1_Change()
PortAddress = Combo1.Text
End Sub
Private Sub Form_Load()
```

```
'PortAddress = 888
Text1.Text = PortAddress
End Sub
Private Sub Command1_Click()
   Do While PortAddress = 888
   IO_Out PortAddress, Text2.Text
   DoEvents
   For x = 0 To 10
   For y = 0 To 500000
   Next y
   Next x
   IO_Out PortAddress, 0
   For y = 0 To 1000000
   Next y
   Loop
End Sub
Private Sub Command2_Click()
Text3.Text = IO_In(PortAddress)
End Sub
Private Sub Command3_Click()
Text3.Text = 0
Text2.Text = 0
IO_Out PortAddress, 0
End
End Sub
Private Sub Command4_Click()
IO_Out PortAddress, 0
IO_Bit_Set PortAddress, Text4.Text
End Sub
Private Sub Command5_Click()
Text5.Text = IO_Bit_Read(PortAddress, Text6.Text)
End Sub
Private Sub Command6_Click()
IO_Out PortAddress, 0
Text2.Text = 0
Text3.Text = 0
Text4.Text = 0
Text5.Text = 0
Text6.Text = 0
End Sub
Private Sub Command7_Click()
IO_Bit_RShift PortAddress, True
End Sub
Private Sub Command8_Click()
IO_Bit_LShift PortAddress, True
End Sub
Private Sub Command9_Click()
IO_Bit_RShift PortAddress, False
End Sub
Private Sub Command10_Click()
IO_Bit_LShift PortAddress, False
End Sub
Private Sub Text1_Change()
Text1.Text = PortAddress
End Sub
Private Sub Text3_Change()
Text3.Text = Text3.Text
End Sub
```

3. C++ Code Needed to Access IO.DLL

```
#include <windows.h>
typedef void (WINAPI *PORTOUT) (short int Port, char Data);
typedef void (WINAPI *PORTWORDOUT)(short int Port, short int Data);
typedef void (WINAPI *PORTDWORDOUT)(short int Port, int Data);
typedef char (WINAPI *PORTIN) (short int Port);
typedef short int (WINAPI *PORTWORDIN)(short int Port);
typedef int (WINAPI *PORTDWORDIN)(short int Port);
typedef void (WINAPI *SETPORTBIT)(short int Port, char Bit);
typedef void (WINAPI *CLRPORTBIT)(short int Port, char Bit);
typedef void (WINAPI *NOTPORTBIT)(short int Port, char Bit);
typedef short int (WINAPI *GETPORTBIT)(short int Port, char Bit);
typedef short int (WINAPI *RIGHTPORTSHIFT)(short int Port, short int Val);
typedef short int (WINAPI *LEFTPORTSHIFT)(short int Port, short int Val);
typedef short int (WINAPI *ISDRIVERINSTALLED)();
extern PORTOUT PortOut;
extern PORTWORDOUT PortWordOut;
extern PORTDWORDOUT PortDWordOut;
extern PORTIN PortIn;
extern PORTWORDIN PortWordIn;
extern PORTDWORDIN PortDWordIn;
extern SETPORTBIT SetPortBit;
extern CLRPORTBIT ClrPortBit;
extern NOTPORTBIT NotPortBit;
extern GETPORTBIT GetPortBit;
extern RIGHTPORTSHIFT RightPortShift;
extern LEFTPORTSHIFT LeftPortShift;
extern ISDRIVERINSTALLED IsDriverInstalled;
extern int LoadIODLL();

#include "io.h"
PORTOUT PortOut;
PORTWORDOUT PortWordOut;
PORTDWORDOUT PortDWordOut;
PORTIN PortIn;
PORTWORDIN PortWordIn;
PORTDWORDIN PortDWordIn;
SETPORTBIT SetPortBit;
CLRPORTBIT ClrPortBit;
NOTPORTBIT NotPortBit;
GETPORTBIT GetPortBit;
RIGHTPORTSHIFT RightPortShift;
LEFTPORTSHIFT LeftPortShift;
ISDRIVERINSTALLED IsDriverInstalled;
HMODULE hio;
void UnloadIODLL() {
        FreeLibrary(hio);}
int LoadIODLL() {
        hio = LoadLibrary("io");
        if (hio == NULL) return 1;
        PortOut = (PORTOUT)GetProcAddress(hio, "PortOut");
        PortWordOut = (PORTWORDOUT)GetProcAddress(hio, "PortWordOut");
        PortDWordOut = (PORTDWORDOUT)GetProcAddress(hio, "PortDWordOut");
        PortIn = (PORTIN)GetProcAddress(hio, "PortIn");
        PortWordIn = (PORTWORDIN)GetProcAddress(hio, "PortWordIn");
        PortDWordIn = (PORTDWORDIN)GetProcAddress(hio, "PortDWordIn");
        SetPortBit = (SETPORTBIT)GetProcAddress(hio, "SetPortBit");
        ClrPortBit = (CLRPORTBIT)GetProcAddress(hio, "ClrPortBit");
        NotPortBit = (NOTPORTBIT)GetProcAddress(hio, "NotPortBit");
        GetPortBit = (GETPORTBIT)GetProcAddress(hio, "GetPortBit");
        RightPortShift = (RIGHTPORTSHIFT)GetProcAddress(hio, "RightPortShift");
        LeftPortShift = (LEFTPORTSHIFT)GetProcAddress(hio, "LeftPortShift");
        IsDriverInstalled = (ISDRIVERINSTALLED)GetProcAddress(hio, "IsDriverInstalled");
        atexit(UnloadIODLL);
        return 0;}
```

4. Examples in C or C++ of how to read/write a bit[6]

```
/*********************************************************************/
/* P_0 Pin Output Bit 0 ON (Hi) and OFF (Lo) Control                 */
/*********************************************************************/
void OUTBIT0_HI()                                   /* Control of P_0 pin Hi */
        {
        mask = mask | 0x01;                         /* forces P_0 pin Hi */
        PortOut(data_port,mask);
        }
void OUTBIT0_LO()                                   /* Control of P_0 pin Lo */
        {
        mask = mask & 0xFE;                         /* forces P_0 pin Lo */
        PortOut(data_port,mask);
        }

/*********************************************************************/
/* I_3 Pin Input Bit 3 Read Control                                  */
/*********************************************************************/
unsigned char INBIT3()
        {
        unsigned char BIT3_value;
        CLK_HI();                                   /* provide clock */
        BIT3_value = PortIn(status_port) & 0x08;    /* get value on pin I-3 and isolate */
        BIT3_value = BIT3_value >> 3;               /* shift to LSB */
        DELAY();                                    /* time delay */
        CLK_LO();
        DELAY();
        return(BIT3_value);
        }
```

5. Sample Codes in C or C++ of how to read/write a byte[16]

```
/*********************************************************************/
/* Routine transmits a data byte to the SPI memory.                  */
/* The data byte is passed to this routine directly when called.     */
/*********************************************************************/
void SEND_BYTE(unsigned char byte)
        {
        char count;
        for (count = 0; count <= 7; count++)
                {                                   /* loop to pass each bit */
                SCLK_LO();
                if ((byte & 0x80) == 0)             /* is the bit LOW? */
                        SI_LO();
                else
                        SI_HI();
                byte = byte << 1;                   /* rotate to get next bit */
                DELAY();                            /*unsigned char mask;
                SCLK_HI();                          /* provide clock */
                DELAY();
                SCLK_LO();
                DELAY();
                }
        }

/*********************************************************************/
/* Routine receives a data byte from the SPI memory and              */
/* passes it back to the calling routine as an unsigned char.        */
/*********************************************************************/
unsigned char GET_BYTE()
        {
        int count;
        unsigned char byte, temp;
        byte = 0;                                          /* reset byte holder */
        for (count = 0; count <= 7; count++)
                {                                          /* loop to get each bit */
                byte = byte << 1;                          /* rotate for next bit */
                temp = INBIT3();                           /* read SO pin */
                if (temp == 1)
                        byte = byte | 0x01;                /* reconstruct current bit */
                else
                        byte = byte | 0x00;
                }
        return(byte);
        }
```