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

Follow this and additional works at: https://digitalcommons.odu.edu/engtech_fac_pubs

Part of the Engineering Education Commons, and the Scholarship of Teaching and Learning Commons

Repository Citation
https://digitalcommons.odu.edu/engtech_fac_pubs/116

Original Publication Citation
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. 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. 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. 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). 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. However, VB6 does not support the straightforward I/O commands that C++ or QBASIC do. Nevertheless, when it comes to Windows NT/2000/XP, most of those I/O commands don’t function anyway.

**Table 1. Visual Basic Functions from IO.DLL**

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

**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.

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.

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**

<table>
<thead>
<tr>
<th>Pin #: DB25</th>
<th>Bit</th>
<th>Signal Name</th>
<th>Inverted?</th>
<th>Pin: Centronics</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>Data Bit 0</td>
<td>No</td>
<td>2</td>
<td>In/Out</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Data Bit 1</td>
<td>No</td>
<td>3</td>
<td>In/Out</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Data Bit 2</td>
<td>No</td>
<td>4</td>
<td>In/Out</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Data Bit 3</td>
<td>No</td>
<td>5</td>
<td>In/Out</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>Data Bit 4</td>
<td>No</td>
<td>6</td>
<td>In/Out</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>Data Bit 5</td>
<td>No</td>
<td>7</td>
<td>In/Out</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>Data Bit 6</td>
<td>No</td>
<td>8</td>
<td>In/Out</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>Data Bit 7</td>
<td>No</td>
<td>9</td>
<td>In/Out</td>
</tr>
</tbody>
</table>
Table 3. PC Parallel Port Status Register: at Address (Base Address + 1) = $379, or 0X379, or 889

<table>
<thead>
<tr>
<th>Pin #: DB25</th>
<th>Bit</th>
<th>Signal Name</th>
<th>Inverted?</th>
<th>Pin: Centronics</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>nError</td>
<td>No</td>
<td>32</td>
<td>In</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>Select</td>
<td>No</td>
<td>13</td>
<td>In</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>PaperEnd</td>
<td>No</td>
<td>12</td>
<td>In</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>nAck</td>
<td>No</td>
<td>10</td>
<td>In</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>Busy</td>
<td>Yes</td>
<td>11</td>
<td>In</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Pin #: DB25</th>
<th>Bit</th>
<th>Signal Name</th>
<th>Inverted?</th>
<th>Pin: Centronics</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>nStrobe</td>
<td>Yes</td>
<td>1</td>
<td>Out</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>nAutoLF</td>
<td>Yes</td>
<td>14</td>
<td>Out</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>nInit</td>
<td>No</td>
<td>31</td>
<td>Out</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>nSelectIn</td>
<td>Yes</td>
<td>36</td>
<td>Out</td>
</tr>
</tbody>
</table>

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 inverted 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.

![Diagram of the circuit design between the PC serial ports](image)

**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. 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.

![Diagram of the PC Serial Port Interface](image)

**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.\textsuperscript{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](image)

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\textsuperscript{16}.

### Table 5. EEPROM Command Byte

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

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.\textsuperscript{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

<table>
<thead>
<tr>
<th>Part #</th>
<th>Freq.</th>
<th>Vcc</th>
<th>Icc</th>
<th>Modulation</th>
<th>Baud Rate</th>
<th>Audio</th>
<th>Range</th>
<th>Price</th>
<th>Maker</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCT-433-AS</td>
<td>433.92 MHz</td>
<td>2-12 Vdc</td>
<td>5ma@3V</td>
<td>ASK/OOK</td>
<td>4800</td>
<td>No</td>
<td>100-300 ft</td>
<td>$4.90</td>
<td>Radiotronix</td>
</tr>
<tr>
<td>TXE-315-KH</td>
<td>315 MHz</td>
<td>2.7-5.2 Vdc</td>
<td>3 ma</td>
<td></td>
<td>4800</td>
<td>No</td>
<td></td>
<td>$9.98</td>
<td>LINX</td>
</tr>
<tr>
<td>TXM-315-LC</td>
<td>315 MHz</td>
<td>2.7-5.2 Vdc</td>
<td>3 ma</td>
<td></td>
<td>4800</td>
<td>No</td>
<td></td>
<td>$6.90</td>
<td>LINX</td>
</tr>
<tr>
<td>TXLC-434</td>
<td>433.92 MHz</td>
<td></td>
<td></td>
<td>CPCA</td>
<td>5000</td>
<td>No</td>
<td>300 ft</td>
<td>$14.95</td>
<td>Reynolds Electronics</td>
</tr>
<tr>
<td>TM1V</td>
<td>418 MHz</td>
<td>5 Vdc</td>
<td></td>
<td></td>
<td>4800</td>
<td>No</td>
<td>300 ft</td>
<td>$16.40</td>
<td>GLOLAB</td>
</tr>
<tr>
<td>TLP-434</td>
<td>433.92 MHz</td>
<td>2-12 Vdc</td>
<td>5 ma</td>
<td>ASK</td>
<td>3000</td>
<td>No</td>
<td></td>
<td></td>
<td>Reynolds, LAIPAC Technology</td>
</tr>
<tr>
<td>TRF4900PW</td>
<td>850-950 MHz</td>
<td>2.2-3.6 Vdc</td>
<td>58 ma</td>
<td>FM/FSK</td>
<td>20 MHz Clock</td>
<td>No</td>
<td></td>
<td>$4.75</td>
<td>Texas Instruments</td>
</tr>
<tr>
<td>TH7107E</td>
<td>315/433 MHz</td>
<td></td>
<td></td>
<td>FSK/FM/ASK</td>
<td></td>
<td></td>
<td></td>
<td>$11.29</td>
<td>Melexis</td>
</tr>
</tbody>
</table>

### Table 7. Available Receiver Products

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

### Table 8. Available Transceiver Products

<table>
<thead>
<tr>
<th>Part #</th>
<th>Freq.</th>
<th>Vcc</th>
<th>Icc</th>
<th>Modulation</th>
<th>Baud Rate</th>
<th>Audio</th>
<th>Applications</th>
<th>Price</th>
<th>Maker</th>
</tr>
</thead>
<tbody>
<tr>
<td>EWM-900-FDTC-BS</td>
<td>902-928 MHz</td>
<td>3 Vdc</td>
<td>35ma Rx</td>
<td>FM/FSK</td>
<td>19.2K</td>
<td>Yes</td>
<td>Full-Duplex Data &amp; Audio 500-1000 ft</td>
<td>$69.00</td>
<td>Radiotronix</td>
</tr>
<tr>
<td>EWM-900-FDTC-BS</td>
<td>902-928 MHz</td>
<td>3 Vdc</td>
<td>35ma Rx</td>
<td>FM/FSK</td>
<td>19.2K</td>
<td>Yes</td>
<td>Full-Duplex Data &amp; Audio 500-1000 ft</td>
<td>$69.00</td>
<td>Radiotronix</td>
</tr>
</tbody>
</table>

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. 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**:
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

14. System Management Bus (SMBus) Specification, Revision 2.0, Smart Battery System Specifications,
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 Kaohsiung 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 IO_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()

www.sbs-forum.org, Email: battery@sbsforum.org, 2001.
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
        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 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);

Proceedings of the 2004 American Society for Engineering Education Annual Conference & Exposition
Copyright © 2004, American Society for Engineering Education
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 short int (WINAPI *SETPORTBIT)(short int Port, char Bit);
typedef void (WINAPI *CLRPORTBIT)(short int Port, char Bit);
typedef short int (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 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);
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

```c
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

```c
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();
        DELAY();
        SCLK_LO();
        DELAY();
    }
}

/* I_3 Pin Input Bit 3 Read Control */
unsigned char GET_BYTE()
{
    int count;
    return;
}
```
unsigned char byte, temp;
byte = 0;
for (count = 0; count <= 7; count++)
{
    byte = byte << 1;
    temp = INBIT3();
    if (temp == 1)
        byte = byte | 0x01;
    else
        byte = byte | 0x00;
}
return(byte);