A SERIAL DATA COMMUNICATIONS SYSTEM USING PERSONAL COMPUTER PARALLEL PORTS

Thesis
Submitted to
The School of Engineering of the
UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for
The Degree
Master of Science in Electrical Engineering

by
Colleen Barbara Donohue

UNIVERSITY OF DAYTON
Dayton, Ohio
May 1996
A SERIAL DATA COMMUNICATIONS SYSTEM USING PERSONAL COMPUTER PARALLEL PORTS

APPROVED BY:
ABSTRACT

A SERIAL DATA COMMUNICATIONS SYSTEM USING PERSONAL
COMPUTER PARALLEL PORTS

Name: Donohue, Colleen Barbara
University of Dayton, 1996

Advisor: Dr. F. A. Scarpino

The hardware and software necessary for serial data transfer between personal
computers using parallel ports has been designed, produced, and demonstrated. A
Windows application developed using Microsoft Visual C++ provides an easy to use
graphical interface for both transmitting and receiving data. A parallel port adapter
developed using the Altera Hardware Descriptive Language and implemented with Altera
EEPLDs transmits and receives serial data through the parallel port at a rate of 14,400
bits per second. An interrupt-driven device driver provides the interface between the
application and parallel port adapter. The device driver, written in C and assembly
language, is a Dynamically Linked Library imported by the application. The design and
implementation of the application, interrupt-driven device driver, and parallel port
adapter are presented in this paper.
ACKNOWLEDGMENTS

I sincerely thank my advisor, Dr. Frank Scarpino, for his guidance and support throughout this effort. His tremendous knowledge of and expertise with computer hardware and software enabled me to focus on the specific tasks to be completed and to pursue the best methods of implementation. I also appreciate his patience as I borrowed his oscilloscope and power supply for extended periods of time throughout the software development phase.

I also thank my husband, David Turner, for his patience as I dominated our computer both in the amount of time I used it and in the hardware I installed in it. I also appreciate the committee members, Dr. Rogers and Dr. Westerkamp, who took the time to review this paper.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. SERIAL COMMUNICATION BASICS AND THE PARALLEL PORT</td>
<td>4</td>
</tr>
<tr>
<td>Asynchronous Serial Communication</td>
<td></td>
</tr>
<tr>
<td>PC Serial Port Interface</td>
<td></td>
</tr>
<tr>
<td>PC Parallel Port Interface</td>
<td></td>
</tr>
<tr>
<td>3. PARALLEL PORT ADAPTER DESIGN AND IMPLEMENTATION</td>
<td>11</td>
</tr>
<tr>
<td>Transmitter Design</td>
<td></td>
</tr>
<tr>
<td>Receiver Design</td>
<td></td>
</tr>
<tr>
<td>Transmitter and Receiver Implementation</td>
<td></td>
</tr>
<tr>
<td>4. SOFTWARE DEMONSTRATION</td>
<td>22</td>
</tr>
<tr>
<td>Main Menu</td>
<td></td>
</tr>
<tr>
<td>Transmitting Files or User Input Text</td>
<td></td>
</tr>
<tr>
<td>Receiving Data</td>
<td></td>
</tr>
<tr>
<td>5. DEVICE DRIVER DESIGN</td>
<td>27</td>
</tr>
<tr>
<td>Software Overview</td>
<td></td>
</tr>
<tr>
<td>Port Data Structure</td>
<td></td>
</tr>
<tr>
<td>Writing to and Reading from the Parallel Port</td>
<td></td>
</tr>
<tr>
<td>Installing the Interrupt Service Routines</td>
<td></td>
</tr>
<tr>
<td>Start, Stop, and Continue Bytes</td>
<td></td>
</tr>
<tr>
<td>The SafePageLock() Function</td>
<td></td>
</tr>
</tbody>
</table>


6. CONCLUDING REMARKS ................................................................. 40

APPENDICES

Appendix A ................................................................. 43
Appendix B ................................................................. 47
Appendix C ................................................................. 58

BIBLIOGRAPHY ................................................................. 66
LIST OF ILLUSTRATIONS

1. 1-1: Transmitting and Receiving Processes .................................................. 2
2. 2-1: Serial Bit Stream for “A” without Parity ............................................. 5
3. 3-1: Transmitter Block Diagram .................................................................. 12
4. 3-2: Receiver Block Diagram ....................................................................... 16
5. 3-3: Sampling Input Data Bits .................................................................... 17
6. 3-4: Parallel Port Adapter Configuration .................................................... 19
7. 4-1: Application’s Main Menu ....................................................................... 23
8. 4-2: Transmit Menu .................................................................................... 23
9. 4-3: Receive Menu ..................................................................................... 23
10. 4-4: Transmit Menu after Selecting Initialize Parallel Port ....................... 24
11. 4-5: Transmit Menu after Opening a File .................................................. 25
12. 4-6: Dialog Window Displayed after Selecting Transmit User Input .......... 25
13. 4-7: Receive Menu after Data Has Been Received .................................... 26
14. 5-1: User Menu Selections and Corresponding Function Calls .................. 29
CHAPTER 1
INTRODUCTION

The hardware and software necessary for serial communication between personal computers using parallel ports has been designed, implemented, and demonstrated. Serial communication using parallel ports is a useful capability because a personal computer's two serial ports may be dedicated to other functions such as a mouse or an external modem. However, a personal computer has a parallel port which may be available for serial data communications. A parallel port adapter, an interrupt-driven device driver, and a Windows application enable simple and fast serial data transfer between two personal computers using their parallel ports.

The parallel port adapter consists of a transmitter and a receiver. The transmitter converts eight bit data words written to the parallel port into a serial stream of bits, and the receiver converts a serial stream of bits into eight bit data words which can be read from the parallel port. The device driver handles the necessary parallel port initialization and the actual writing of data to and reading of data from the parallel port. The application provides an easy to use graphical interface which enables the user to transmit either an existing file or an user input string and to save or clear received data. The parallel port adapter and device driver interact through hardware interrupts. The device
driver and application communicate through shared memory buffers. The following figure illustrates the transmitting and receiving processes.

During transmission, the application writes a block of data bytes to a memory buffer. The data bytes are either from an existing file opened by the user or from a text string entered by the user. The device driver reads the first data byte from that buffer and writes it to the parallel port. The parallel port adapter’s transmitter transmits the data byte serially and produces a hardware interrupt when it is done transmitting. The hardware interrupt causes the device driver to write the next data byte. When the buffer is empty or that last data byte has been transmitted, the device driver sends a message to the application. If additional data are to be transmitted, the application will write that data to the buffer, and the device driver will write it to the port. This process continues until all of the data has been transmitted.

The parallel port adapter’s receiver loads serial data transmitted to it, one bit at a time, into an eight bit register. When a byte has been received, a hardware interrupt is
generated. The device driver responds to this interrupt by reading the byte from the parallel port and writing it to a memory buffer. All received data are written into the memory buffer following this process. When the last data word has been received, the device driver sends a message to the application notifying it that the there is new data available in the buffer. The application displays the received data which can then be either saved in a file or cleared.

The parallel port adapter was developed with the Altera Hardware Descriptive Language and Altera EEPLDs. The device driver was written in C as a Dynamically Linked Library. The Windows application was developed using Microsoft Visual C++ and the Microsoft Foundation Classes. The design and implementation of the Windows application, the interrupt-driven device driver, and the parallel port adapter are presented in this paper.
CHAPTER 2

SERIAL COMMUNICATION BASICS AND THE PARALLEL PORT

The parallel port adapter essentially converts a parallel port to a serial port. Therefore, data can be transmitted and received serially through parallel ports. This chapter presents background information about serial communication, the serial port, parallel communication, and the parallel port to facilitate understanding the hardware and software developed for this thesis.

Asynchronous Serial Communication

Serial communication is the transmission of data, one bit at a time, over a single transmission line to a receiver. Each bit is transmitted for a specified period of time, one immediately after another, thereby producing a contiguous stream of bits. For example, one data bit may be transmitted for 69.44 $\mu$ seconds. Then another bit is transmitted for 69.44 $\mu$ seconds. With each bit transmitted for 69.44 $\mu$ seconds, 14,400 bits can be transmitted in one second. Serial communication is commonly used to transfer data between personal computers.

Typically, the data transferred between computers consists of ASCII (American Standard Code for Information Interchange) characters. ASCII characters are the binary coded representation of text characters and various nonprinting control characters. These
characters are seven bits long. An eighth bit can be used for parity\(^1\) checking to ensure no errors have occurred during transmission. These eight bits are transmitted in a “frame” with both a “start” bit which is a logic level 0 (0 volts) and a “stop” bit which is a logic level 1 (+5 volts). The ASCII code for “A” is 41\(_{16}\) or 1000001\(_2\). The transmitted “A” with no parity would be 110000010\(_2\). The signal for transmitting an “A” with no parity is presented in Figure 2-1. Note that the least significant bit is transmitted first and that when data is not being transmitted the signal stays high (+5 volts).

![Figure 2-1: Serial Bit Stream for “A” without Parity](image)

The high to low transition of the start bit signals the receiver that a character needs to be received. The receiver reads in the bits of the character, one at a time, until the stop bit is detected. The receiver stops reading until another start bit occurs. This is referred to as asynchronous communication since the receiver resynchronizes itself to the transmitter with each start bit.

---

\(^1\) For even parity, the eighth bit is 0 if the ASCII character has an even number of 1s or 1 if the ASCII character has an odd number of 1s. Likewise, for odd parity, the eighth bit is 0 if the ASCII character has an odd number of 1’s or 1 if the ASCII character has an even number of 1’s.

\(^2\) The UART’s transmitter actually inverts outgoing signals so that when an “A” is written to the serial port, 001111101\(_2\) is transmitted from the computer. The UART’s receiver also inverts incoming signals so 110000010\(_2\) would be received by another computer. This inversion of signals is not addressed since it does not affect the principles of serial communication.
PC Serial Port Interface

Two computers, with the necessary software and a cable connecting their serial ports, can exchange data. Most personal computers have two serial ports commonly referred to as COM1 and COM2. A serial port is comprised of a Universal Asynchronous Receiver Transmitter (UART) and special driver and receiver circuitry.

When the computer writes data bytes to the serial port for transmission, the UART converts the parallel data to a serial data stream and adds the parity, start, and stop bits. When data is received by the serial port, the UART converts the serial data stream into parallel data which can be read by the computer. The additional driver and receiver circuitry change the standard TTL voltage signals of 0 and +5 volts to -12 and +12 respectively. These voltages ensure that the transmitted and received signals are not corrupted by noise and distortion induced when long cables are used.

The UART has twelve registers which retain the transmitted and received data, the status of the port, and the port’s control settings. Eight of these registers can be read from and written to as they have their own unique addresses. These registers are used to set the bit rate, set the type of parity checking, and check for errors.

The physical serial port is comprised of nine pins on the back of the computer. The pins are assigned according to the following table.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data Carrier Detect</td>
</tr>
<tr>
<td>2</td>
<td>Receive Data</td>
</tr>
<tr>
<td>3</td>
<td>Transmit Data</td>
</tr>
<tr>
<td>4</td>
<td>Data Terminal Ready</td>
</tr>
</tbody>
</table>

Some serial ports have 25 pins. The same signals are provided; they are simply assigned to different pin numbers.
A wire which connects the Transmit Data pin of the transmitting computer to the Receive Data pin of a receiving computer and a wire which connects the two ground pins together are sufficient for basic serial communication. Initially, the rate at which the data is to be transmitted and received must be written to the ports. Common rates are 9600, 14,400, and 28,800 bits per second (bps). Then the data byte to be transmitted is simply written to the port of the transmitting computer. The UART will output the data serially through the Transmit Data pin (pin 3). The UART in the receiving computer will automatically read the data in from the Receive Data pin (pin 2). If another wire were connected to the Receive Data pin of the transmitting computer and the Transmit Data pin of the receiving computer, data could be transferred back and forth between the two computers (full duplex). Serial communication is popular because it is simple to implement, cost effective, and reliable over long distances.

PC Parallel Port Interface

Eight bits of data can be transmitted simultaneously or received simultaneously through a computer’s parallel port. Centronics\(^4\) developed the parallel port for communicating with line printers. Although no longer used with line printing, the interface is still used with high quality printers and other devices. Most computers have two parallel ports referred to as LPT1 and LPT2.

\(^4\)Centronics was a company which manufactured line printers. It is no longer in business.
A parallel port consists of three registers: data register, status register, and control register. These registers are accessed by writing data to or reading data from the registers' unique addresses. The data register is used to transmit data to or read data from a connected device. The device communicates its current status to the computer via the status register. The computer controls the device through the control register. The bits in the status and control registers are defined in the following table.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Status Register</th>
<th>Control Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Strobe Line</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Auto LF line</td>
</tr>
<tr>
<td>2</td>
<td>IRQ Status</td>
<td>Initialize Printer (active low)</td>
</tr>
<tr>
<td>3</td>
<td>Error (active low)</td>
<td>Select in line</td>
</tr>
<tr>
<td>4</td>
<td>Select</td>
<td>enable IRQ line</td>
</tr>
<tr>
<td>5</td>
<td>Paper End</td>
<td>Port Direction</td>
</tr>
<tr>
<td>6</td>
<td>Acknowledge (active low)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Busy (active low)</td>
<td></td>
</tr>
</tbody>
</table>

The Port Direction bit of the control register is very important for communication because it determines whether the port direction is in or out. If the Port Direction bit is 0, data is written to the port (out). If it is 1, data is read from the port (in).

The physical parallel port consists of 25 pins accessible on the back of a personal computer. The pins are assigned according to the following table.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Strobe (active low)</td>
</tr>
<tr>
<td>2-9</td>
<td>Data Lines D0-D7</td>
</tr>
<tr>
<td>10</td>
<td>Acknowledge (active low)</td>
</tr>
<tr>
<td>11</td>
<td>Busy (active low)</td>
</tr>
<tr>
<td>12</td>
<td>Paper End</td>
</tr>
<tr>
<td>13</td>
<td>Select</td>
</tr>
<tr>
<td>14</td>
<td>Auto LF</td>
</tr>
<tr>
<td>15</td>
<td>Error (active low)</td>
</tr>
<tr>
<td>16</td>
<td>Initialize</td>
</tr>
<tr>
<td>17</td>
<td>Select In</td>
</tr>
<tr>
<td>18-25</td>
<td>ground</td>
</tr>
</tbody>
</table>

An important feature of the parallel port is the ability to generate a hardware interrupt. The Acknowledge pin of parallel port LPT1 is connected to the seventh
interrupt request line (IRQ7) of the Programmable Interrupt Controller (PIC) through an inverter. All hardware interrupts are processed by the PIC. It handles an interrupt by calling the appropriate interrupt service routine when no other interrupts of higher priority need to be serviced. IRQ7 is enabled by setting bit seven of the interrupt controller's interrupt mask register to 0. A low to high transition on IRQ7 generates an interrupt. The Acknowledge pin of the parallel port is enabled for interrupts by writing a 1 to bit four of the port's control register. Since IRQ7 is connected to the Acknowledge line through an inverter, a high to low transition on the Acknowledge pin actually generates the interrupt. Interrupt capability is important because any device attached to the parallel port can simply interrupt the computer when it needs service.

Originally, handshaking protocols were developed for printing because printers print slower than computers can write data to the port. The handshaking protocols regulate the rate at which new data is written to the port so that data is not lost. When the computer is ready to print a data byte, it checks to see if the printer is busy (Busy equal to 0). If the printer is available, the computer writes a data byte to the data register and sets the strobe line high by writing a 1 to bit 0 in the control register. The change in the strobe line signals the printer that a valid data byte is ready to be printed. The printer then latches the data byte and sets the Busy line high. The computer continuously checks the Busy line to see when it is low indicating that the printer is ready for another byte. Hence, the computer is essentially dedicated to the printer and cannot service other tasks.

Using interrupts, however, the computer is available to do other things while a data byte is being printed. With interrupts enabled, the printer simply sets the
Acknowledge line low when it is finished printing. This transition on the Acknowledge line interrupts the computer. The computer responds to the interrupt by writing the next data byte to the printer. The transmitter of the parallel port adapter generates an interrupt signal when it is done transmitting a data word to optimize the transmission rate and to permit the computer to perform other tasks during transmission. Likewise, the receiver of the parallel port adapter generates an interrupt signal when a byte has been received and is ready to be read by the computer.

Although the data transfer rate using the parallel port is eight times the serial communication rate, the parallel port is most frequently used just for printing. The parallel port outputs data using TTL voltage levels which are susceptible to noise over distances more than fifteen feet. In addition, the cables are bulky and expensive since there are eight data lines, control, and ground lines, and data can be transferred in only one direction at a time (half duplex). An adapter which essentially converts a parallel port into a serial port takes advantage of an available parallel port and eliminates some of the common problems of using the parallel port for data communication.
CHAPTER 3
PARALLEL PORT ADAPTER DESIGN AND IMPLEMENTATION

The parallel port adapter consists of a transmitter and a receiver. The transmitter converts eight bit data words written to the parallel port into a serial stream of bits transmitted at a rate of 14,400 bits per second (bps). The receiver converts transmitted serial data into eight bit data words which can be read from the parallel port. This chapter describes the receiver and transmitter designs and their implementation.

Transmitter Design

The strobe line and the eight data lines from the computer’s parallel port are the input to the transmitter. The output of the transmitter is the line which carries the serial data. This line can be used as input to an inverter connected to a serial port\(^5\) or as input to the receiver of a parallel port adapter. A block diagram of the transmitter is presented on the following page. The program written with the Altera Hardware Description Language which implements the transmitter design is listed in Appendix A.

A high to low transition of the strobe signal indicates that there is valid data which needs to be transmitted on the computer’s data lines. The transmitter reads the data into a register and adds a start bit (0 volts) and a stop bit (5 volts) to the data word.

\(^5\) Recall that the serial port inverts signals as they are transmitted and received.
Figure 3-1: Transmitter Block Diagram
(There is no parity checking with this design so the transmitter's eighth data bit is set at the computer's eighth data pin setting.) The transmitter then shifts the ten bits out of the register at the rate of 14.4 kbps. When no data are being transmitted, the transmitter sets its output constantly high (5 volts).

The transmitter is inactive until a high to low transition on the strobe line occurs. The transition is detected by two D flip flops. The strobe line is the input to one flip flop (Edge[1])\(^6\). The flip flop's output is connected to the input of the other flip flop (Edge[0]). Therefore, on each clock cycle, the strobe line is sampled by Edge[1], and the previous sample is stored in Edge[0]. A signal called LOAD is created by ANDing the inverted Edge[1] output and the Edge[0] output. When a transition occurs, Edge[1] will be low and Edge[0] will be high so LOAD will be set high (5 volts). The LOAD signal is used to load the data word currently on the computer's output pins into the transmitter's buffer and to generate the control signals necessary for correct operation of the transmitter.

The primary components of the transmitter are an eight bit buffer (InBuff) and a ten bit shift register (XmitReg). When LOAD goes high, the data word currently on the computer's output pins is loaded into the buffer (InBuff). As long as LOAD is low, the data word currently in the buffer is retained. If the transmit register (XmitReg) is empty\(^7\), the newly loaded data word is immediately transferred from the buffer to the register.

\(^6\) These names refer to the names used in the program written with the Altera Hardware Description Language for implementing the transmitter and receiver.

\(^7\) If the transmit register is empty and new data is in the buffer, the control signal LoadXmitReg will transition high so that the buffer's data will be transferred into the register.
The transmit register is ten bits to accommodate the start and stop bits. The input to the first bit of the register is low (ground) to generate the start bit. The input to the tenth bit is high (VCC) to generate the stop bit. Therefore, the eight bits of the buffer are loaded into the second through ninth bits of the transmit register. Once the transmit register is loaded, the ten bits are shifted out of the first bit of the register serially at a rate of 14.4 kbps.

The transmitter’s system clock rate is 3.6864 Mhz. Therefore, a shift signal (ShiftClock) with a frequency of 14.4 kHz must be generated so that the transmitter transmits data bits at 14.4 kbps. Conveniently, 256 multiplied by 14.4 kHz is 3.6864 Mhz so that a counter (ShiftCounter) which counts from 0 to 255 connected to a decoder generates the necessary ShiftClock signal. The decoder output is 0 for all counter values except 0. ShiftClock is the clock signal for the transmit register. All other clock signals in the transmitter are provided by the system clock.

If the transmit register is transmitting data when LOAD goes high, a new data word will be loaded into the buffer. However, the new data word will not be loaded into the transmit register until the transmit register is empty. Control signals are generated which monitor the current states of the buffer and transmit register to ensure that transmitter operates correctly.

The five control signals are InBuffFull, LoadXmitReg, XmitRegFull, WordFrame, and DataSent. InBuffFull is set when the buffer is full. The buffer is full if it has just been loaded or if it is currently full and the transmit register has not been loaded yet (not LoadXmitReg). The transmit register can be loaded (LoadXmitReg) if the buffer is full.
(InBuffFull) and the transmit register is currently empty (not XmitRegFull).

XmitRegFull is set if the transmit register has just been loaded or if the transmit register is currently full and the entire data word (including start and stop bytes) has not been transmitted (not DataSent). WordFrame is set when the transmit register is loaded and remains set as long as the transmit register is full. While WordFrame is set, the counter is active and the ShiftClock signal is generated. When ten ShiftClock pulses have occurred (i.e., all ten bits have been transmitted out), DataSent is set. WordFrame will reset if there is not another data word to transmit.

Receiver Design

The input to the receiver is simply a line carrying serial data. This line can be provided by the output of an inverter connected to the output pin of a serial port or by the transmitter output of a parallel port adapter. Figure 3.2 is a block diagram of the receiver. The program written with the Altera Hardware Description Language which implements the receiver design is listed in Appendix A.

The receiver is inactive until the start bit (0 volts) occurs. Recall that the transmitter output of the parallel port adapter is constantly high (5 volts) when no data is being transmitted. The receiver detects the start bit with two D flip flops. The data line is the input to one flip flop (Edge[1]). The flip flop's output is connected to the input of the other flip flop (Edge[0]). Therefore, on each clock cycle, the input value is sampled by Edge[1], and the previous sample is stored in Edge[0]. When a start bit occurs, Edge[1] will be low (0 volts) and Edge[0] will be high (5 volts). A signal called EdgeDet
Figure 3.2: Receiver Block Diagram
(Edge Detected) is created by ANDing the inverted Edge[1] output and the Edge[0] output. EdgeDet goes high when the start bit is detected.

Once a start bit has been detected, the receiver reads in all the data bits until the stop bit is received. WordFrame is a control signal which is set high as long as there are data bits which need to be received. It is generated by a D flip-flop. If a start bit has been detected (EdgeDet=5 volts) or if WordFrame is currently high and the stop bit has not been received (LastBit=0 volts), WordFrame will be set (5 volts). Otherwise, it is reset.

As soon as WordFrame is set high, a counter is enabled to generate a sampling signal. The sampling signal (SampleClock) is used as the clock input to an eight bit shift register (InReg). The serial data is the data input to the shift register so that each time SampleClock goes high the input is "sampled" and the previously sampled value is shifted to the next register location. SampleClock is a delayed pulse signal with a cycle of 14.4 kHz. It is delayed so that the sampling of the input occurs "in the middle" of a bit, see figure 3.3.

![Input Data:](image)

Receiver's clock:

SampleClock:

Figure 3.3: Sampling Input Data Bits

The receiver's system clock cycle is 3.6864 MHz so the sampling signal must be generated using a counter and a decoder as is done with the transmitter. Since 256
multiplied by 14.4 kHz is 3 6864 MHz, a counter (SampleCounter) which counts from 0 to 255 connected to a decoder generates the necessary signal. The decoder output is low (0 volts) for all counter values except 128. When the counter equals 128, the decoder output is high (5 volts). The counter value of 128 assures that the incoming data bits are sampled “in the middle”.

SampleCock is also input to a counter (Bit Counter). BitCounter keeps track of how many bits have been received. The output of BitCounter is input to a decoder so that when BitCounter equals nine, a control signal, LastBit, is set high (5 volts).

Although ten bits are transmitted (eight data bits, start bit, stop bit), only the eight data bits are saved in the shift register. The start bit is “lost” as it is shifted out of the register as the last data bit is shifted in, and the stop bit is simply ignored since the bits are counted to determine when all have been received.

LastBit is set high just after the last data bit has been received. LastBit is used to enable an eight bit buffer (Buff). When LastBit goes high, the eight bits in the shift register are loaded into the buffer. The received data is output through the buffer. The data can be read from the buffer while another data word is being received into the register. Also, when LastBit goes high, WordFrame is reset signifying that data does not need to be received. The receiver simply listens for another start bit.

Transmitter and Receiver Implementation

The transmitter and receiver were built with Altera EEPLDs, CMOS crystal oscillators, and CMOS inverters. The EEPLDs were programmed from code written
Figure 3.4: Parallel Port Adapter Configuration
with the Altera Hardware Description Language. The code for the transmitter and receiver is listed in Appendix A and was discussed in the previous two sections of this chapter. The block diagram in figure 3.4 shows the implementation of the parallel port adapter’s transmitter and receiver and the configuration of the adapter as used in this thesis.

The transmitter requires two EEPLDS. The inputs to the transmitter are eight data lines (D0-D7), clock, strobe, power, and ground. The data and strobe inputs are provided by a personal computer’s parallel port. The clock input is provided by a CMOS crystal oscillator. Power and ground are supplied by a power supply. The outputs are the serial data line (XmitReg0) and the DataSent signal. DataSent is connected to the parallel port’s Acknowledge pin through an inverter. When the last bit has been transmitted, DataSent goes high and a hardware interrupt generated8 (IRQ7 and Acknowledge line must be enabled). The interrupt signals the computer to write another data byte to the port. The serial data line XmitReg0 is connected to the data input of the receiver.

The receiver fits completely on one EEPLD. The inputs to the receiver are a serial data line, clock, power, and ground. The outputs are eight data lines (D0-D7) and the LastBit signal. The eight data lines are connected to the parallel port’s eight data lines. LastBit is connected to the parallel port’s Acknowledge pin through an inverter. When the eight bits have been received, LastBit goes high and a hardware interrupt is generated. The interrupt prompts the computer to read the eight bits from the port.

8 A high to low transition on IRQ7 generates an interrupt. Since the parallel port inverts the Acknowledge signal, a low to high transition on the Acknowledge pin is required. Since DataSent goes high when the last bit has been transmitted, DataSent must be inverted.
This configuration simply tests the design of the parallel port adapter’s transmitter and receiver. Only one parallel adapter was built so the transmitter has to be connected to the parallel port of one computer and the receiver has to be connected to the parallel port of another computer. Using the developed software, data can be transmitted serially from one computer with to the other, one computer is always the transmitter and the other the receiver. Two parallel port adapters would be necessary for each computer to be able to both transmit and receive. The additional receiver and transmitter needed to build two complete adapters would be superfluous for this thesis as they are not necessary to demonstrate that the developed hardware and software enable serial communication using parallel ports.
CHAPTER 4
SOFTWARE DEMONSTRATION

If the parallel port adapter is connected to two computers as shown in figure 3.4 and the developed application is executed, data can be transmitted serially from the computer with the transmitter to the computer with the receiver through their parallel ports. The application provides an easy to use Windows based graphical user interface which guides the user through a series of menu selections to transmit files, transmit user input text, or to receive data. The device driver actually performs the requested tasks. The application’s features are demonstrated in this chapter. Chapter V presents the important design features of the device driver.

Main Menu

Figure 4.1 shows the application’s window displayed when the software is executed. The status bar at the bottom of the window displays “Inactive” since data is neither being transmitted or received. The window’s main menu has three headings: Exit, Transmit, and Receive. When the Exit heading is selected, a drop down menu with Exit as the only option is displayed. Selection of Exit again, simply exits the program. The Transmit and Receive drop down menus are shown in figures 4.2 and 4.2. They are extensive and require detailed explanation.
The application is designed so that the user is forced to transmit and receive in the correct sequence. Only the menu items which make sense to execute at a given time are active or black. The other menu items are “grayed” until the appropriate time. When the application is initially executed, Exit, Initialize Parallel Port on the Transmit Menu, and Initialize Port and Receive Data on the Receive Menu are the only active menu items. Therefore, the only actions permitted are to exit the application, configure the parallel port for transmitting, or configure the parallel port for receiving.
Transmitting Files or User Input Text

The parallel port must be initialized before data can be transmitted or received. If Initialize Parallel Port from the Transmit Menu is selected, the parallel port is configured for transmitting. Figure 4.4 shows the Transmit Menu after Initialize Parallel Port has been selected. Note that Close Parallel Port, Open File, and Transmit User Input are active and that the status bar displays 'In transmitting mode.'

![Transmit Menu](image)

Figure 4.4: Transmit Menu after Selecting Initialize Parallel Port

Close Parallel Port simply resets the parallel port to its default configuration. Open File pops up the 'Open' common dialog window so that the user can easily browse for the desired file and then open it. Transmit User Input pops up a dialog window which allows the user to enter text and transmit it.

An opened file is displayed in the application's main window. While the file is open, only the Transmit File and Close File options are active as shown in Figure 4.5. If

---

9 Once Initialize Parallel Port has been selected, no Receive Menu items are active until the port is closed from the Transmit Menu.
Transmit File is selected, the file is written to the parallel port. If Close File is selected, the file is cleared from the application’s main window and closed.

![Figure 4.5: Transmit Menu after Opening a File](image)

If the Transmit User Input menu item is selected, a dialog window, Figure 4.6, is displayed which allows the user to enter and transmit a string of up to twenty characters. The dialog window is closed and the entered string is transmitted by selecting Transmit. The dialog window is closed without doing anything by selecting Cancel.

![Figure 4.6: Dialog Window Displayed after Selecting Transmit User Input](image)

Receiving Data

Selecting Initialize Port and Receive Data from the Receive Menu, configures the parallel port to receive data and changes the status bar message to ‘In receiving mode.’
Any data that is transmitted to the port is read automatically into a temporary file and displayed in the application's main window. The Receive Menu, after data has been received, looks like the following figure.

![Receive Menu](image)

**Figure 4.7: Receive Menu after Data Has Been Received**

Selecting **Save Received Data in a File** displays the ‘Save As’ common dialog window so that the temporary file can be saved under a new name in any existing directory. Selecting **Clear Received Data** clears the received data from the screen and closes the temporary file. The application remains in receiving mode after the data has been saved in a file or cleared. Closing the port returns the application to its initial state from which it can be exited or put in transmitting or receiving modes.
CHAPTER 5
DEVICE DRIVER DESIGN

The device driver is the most important and most complicated component of the entire thesis project. The device driver is the interface between the application and the parallel port. It configures the parallel port for transmitting and receiving by writing to the port’s control register, and it installs/uninstalls the transmitting and receiving interrupt service routines. The interrupt service routines handle the actual writing of data to and reading of data from the parallel port. Critical design features of the device driver and its interaction with the application and the parallel port adapter are presented in detail in this chapter.

Software Overview

The application is a Visual C++ program. Its primary function is to display the graphical interface presented in Chapter 4. It displays the main window with its menus and status bar. It also responds to user selections by changing the status bar and drop down menus, displaying dialog windows for saving files, opening files, entering user text, displaying text in the main window, and making function calls into the device driver. The application was coded and built as a project using Microsoft Visual C++. The project contains numerous files necessary for coding and building the application. The
file DRIVAPP.CPP and its header file display the menus and handle user selections. They are listed in Appendix B. The design and implementation of the application is only discussed in the detail necessary to understand the device driver.

The device driver is a dynamic-link library. It configures the parallel port and installs/uninstalls the interrupt service routines. The device driver was coded in a combination of C and assembly language and built as a Dynamic-Link Library Project using Microsoft Visual C++. The library is linked to the application by listing the library's name in the application's Project Linker Options. The library is built from two files, INTPAR.C and ISR.C, and their header files. All of these files are listed in Appendix C.

The application communicates with the device driver through function calls; the device driver communicates with the application by posting messages. When the user selects a menu item, a function in the application tied to that menu item is executed. Some of these application functions then call a function in the device driver. The device driver posts messages to the application to notify the application that it has finished an activity or to prompt the application to perform an activity. The sequences of user menu selections and the corresponding functions executed are given in figure 5.1 on the following page.

For example, when the Initialize Port and Receive Data is selected from the Receive Menu, the function OnReceiveOpenPort() (Appendix B, lines 455-468) in the application file DRIVAPP.CPP is executed. OnReceiveOpenPort() calls the function

---

10 The device driver must be a DLL because its code segment, data segment, and dynamically allocated memory must be fixed. This can only be done in a DLL. See section F.
<table>
<thead>
<tr>
<th>User Menu Selection</th>
<th>Application Function</th>
<th>Device Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Parallel Port</td>
<td>OnTransmitInitPort()</td>
<td>PortOpenXmit()</td>
</tr>
<tr>
<td>Transmit File</td>
<td>OnTransmitFile()</td>
<td>PortWrite()</td>
</tr>
<tr>
<td>Transmit User Input Transmit Button</td>
<td>OnDriverXmitUserInput()</td>
<td>PortWrite()</td>
</tr>
<tr>
<td>[Transmit] Close Parallel Port</td>
<td>OnTransmitClosePort()</td>
<td>PortCloseXmit()</td>
</tr>
<tr>
<td>Initialize Port and Receive Data</td>
<td>OnReceiveOpenPort()</td>
<td>PortOpenReceive()</td>
</tr>
<tr>
<td>[Receive] Close Parallel Port</td>
<td>OnReceiveClosePort()</td>
<td>PortCloseReceive()</td>
</tr>
</tbody>
</table>

**Figure 5.1: User Menu Selections and Corresponding Function Calls**
PortOpenReceive() (Appendix C, lines 198-249) in the device driver file INTPAR.C. PortOpenReceive() configures the parallel port for receiving data and installs the Receive Interrupt Service Routine. If the adapter’s receiver is connected to the parallel port and data is input to the receiver, the receiver will generate an interrupt when it has eight data bits ready for the computer to read. The Receive Interrupt Service Routine (Appendix C, lines 379-401) reads the eight bits from the port and stores them in a buffer. The Receive Interrupt Service Routine reads all of the data input to the receiver following this process. When it recognizes that no additional data will need to be received, it posts a message to the application. The function OnReceiveData() (Appendix B, lines 524-547) in the application answers the message. OnReceiveData() copies the data from the buffer into a temporary file and displays the file in the application’s main menu.

The transmitting process is similar but more complicated than the receiving process. It will not be described as sequence of steps as was done in the previous paragraph for the receiving process. Instead, the significant design features of the device driver will be presented individually throughout the remainder of this chapter. These features in combination with figure 5.1 will explain the details of the transmitting and receiving processes.

Port Data Structure

The application and the device driver share data through a structure, PORTCONTEXT, as defined in the header file INTPAR.H (lines 42-56). *HPORT is a pointer to the PORTCONTEXT structure. The structure contains all of the data types
necessary for both transmitting and receiving data through a parallel port. The structure definition is

```c
typedef struct
{
    WORD PortBase;       // port base address (0x378 for LPT1)
    BYTE blrq;           // IRQ line number (in this case, 7 for IRQ7)
    HWND hwnd;           // window handle of calling application
    VOIDINTPROC pOldHandler; // address of default ISR
    char far *Buffer;    // pointer to transmit buffer
    WORD BufferLength;  // transmit buffer length
    WORD Location;      // pointer to current location in t buffer
    WORD NumBytes;      // number of bytes to transmit
    char far *RBuffer;  // pointer to receive buffer
    WORD RBufferLength; // receive buffer length
    WORD RLocation;     // pointer to current location in r buffer
    WORD RNumBytes;     // number of bytes received
} PORTCONTEXT, FAR *HPORT;
```

An external PORTCONTEXT structure named Port1 is created when the application is executed (Appendix C, line 91). Port1 is then accessed using *HPORT pointers throughout the device driver and the application.

**Writing to and Reading from the Parallel Port**

The C _outp() and _inp() functions are used to write to and read from the parallel port. From a programming perspective, the parallel port is simply three registers at consecutive addresses: Data Register at 0x378\(^{11}\), Status Register at 0x379, and Control Register at 0x37A. _outp() and _inp() write to and read from these registers:

```c
_outp( register address, byte to be written);     // port direction must be out
_inp( register address);                         // port direction must be in
```

In the device driver, the register address is determined by adding an a predefined offset to LPT1’s base address. The base address is the Data Register’s address, 0x378,

---

\(^{11}\) LPT1 is used throughout this effort. LPT2 could be used if desired.
and is stored in PORTCONTEXT. PortBase. The offsets are defined in the header file PAR.H:

```c
#define PAR_CON_REG 0x02
#define PAR_STAT_REG 0x01
#define PAR_DATA_REG 0x00
```

The forms of _inp() and _outp() used in the device driver are:

```c
_outp( hPort->PortBase + PAR_DAT_REG, buf[hPort->Location] );  // output byte
_inp( hPort->PortBase + PAR_DAT_REG );  // read byte
_outp( hPort->PortBase + PAR_CON_REG, ENABLE_IRQ | STROBE_FALSE | PORT_OUT);
```

The first two lines write a byte to and read a byte from the Data Register (hPort is an *HPORT pointer pointing to Port1, and buf[hPort->Location] is actually the byte at hPort->Location in hPort->Buffer). The last line writes three bytes to the Control Register. These three bytes configure the Control Register to transmit data through the parallel port adapter.

The Enable IRQ, Strobe, and Port Direction bits are bits 4, 5, and 1 of the Control Register. If the Enable IRQ bit is set, a transition on the port’s Acknowledge pin will cause an interrupt. Since the parallel port adapter’s transmitter and receiver both generate interrupts\(^\text{12}\), the Enable IRQ bit must be set while data is be transmitted or received. The port’s Strobe pin is high (+5 volts) if the Strobe bit is set and is low (0 volts) if the Strobe bit is reset. The Strobe pin must transition from low to high to trigger the transmitter to load the data on the port’s data pins. The Strobe bit is reset while data is being received since the Strobe pin is not needed. The port’s data direction is out if the Port Direction bit is reset and is in if the Port Direction is set. Therefore, the transmitter

\(^{12}\) When a data byte has been transmitted, the transmitter generates an interrupt to notify the computer that it is ready for the next byte. When a data byte has been received, the receiver generates an interrupt to notify the computer that a data byte is available to be read.
requires that the Port Direction bit be reset, and the receiver requires that it be set. These bits are set and reset by writing the following predefined bytes\(^{13}\) to the Control Register.

![Table of predefined bytes](image)

While transmitting, all three bits are set or reset each time the Control Register is written to. While receiving, only the Enable IRQ and Port Direction bits are set or reset since the Strobe pin is not needed.

The Transmit Interrupt Service Routine (Appendix C, lines 343-377) writes the data to be transmitted, one byte at a time, to the parallel port. The following lines of code transmit the byte at hPort->Location in hPort->Buffer (hPort->Buffer stores a block of data to be transmitted, see ‘Start, Stop, and Continue Bytes’):

```
_outp(hPort->PortBase + PAR_CON_REG, ENABLE_IRQ | STROBE_FALSE | PORT_OUT);
_outp(hPort->PortBase + PAR_DAT_REG, buf[hPort->Location]);
_outp(hPort->PortBase + PAR_CON_REG, ENABLE_IRQ | STROBE_TRUE | PORT_OUT);
_outp(hPort->PortBase + PAR_CON_REG, ENABLE_IRQ | STROBE_FALSE | PORT_OUT);
```

The first line enables interrupts, forces the port’s Strobe pin low, and makes the port’s direction out. The second line writes the byte to be transmitted to the Data Register. The third line forces the Strobe pin high triggering the transmitter to load the byte in the Data Register into its own buffer. The last line resets the Strobe pin low.

The Receive Interrupt Service Routine (Appendix C, lines 379-401) reads received bytes from the port’s Data Register. Before any byte can be read, the Control Register must be initialized:

```
_outp(hPort->PortBase+PAR_CON_REG, ENABLE_IRQ | PORT_IN);
```

\(^{13}\) These bytes are defined in INTPAR.H.
The code to read a byte is:

```c
inChar = _inp( hPort->PortBase+PAR_DATA_REG ); // read LPT1 into inChar
```

This code is simpler than the code to transmit a byte since the Strobe pin is not required. Once the Control Register is properly initialized, bytes can be read, one at a time, by the Receive Interrupt Service Routine without changing the Control Register.

### Installing the Interrupt Service Routines

The Device Driver’s PortOpenXmit and PortOpenRecv functions install the Transmit Interrupt Service Routine (TransmitISR, Appendix C, lines 343-377) and the Receive Interrupt Service Routine (ReceiveISR, Appendix C, lines 379-401) respectively. These interrupt service routines are installed by masking interrupts at the Programmable Interrupt Controller (PIC), loading the new interrupt service routine, and unmasking interrupts at the PIC.

The parallel port’s interrupt request line must be disabled while the new interrupt service routine is installed. The parallel port’s interrupt request line is IRQ7. IRQ7 is disabled by writing a 1 to bit 7 of the Programmable Interrupt Controller’s (PIC) interrupt mask register. The following code\(^{14}\) masks IRQ7 (Appendix C, lines 135-137, 233-235):

```c
mask = _inp( MASTER_PIC_MASK );
SET( mask, 1 << hPort->bIrq );
_outp( MASTER_PIC_MASK, mask );
```

The first line loads the current mask register contents into a variable called mask. The next line performs a logical OR of mask and 10000000\(_2\) which puts a 1 in bit 7 of

---

\( ^{14}\) This code is from page 77 of "Writing Windows™ VxDs and Device Drivers", see reference 1.
variable mask. Variable mask is then written to the PIC’s interrupt mask register by the last line.

The address of the current interrupt service routine is stored before the address of the new interrupt service routine is installed so that it can be reinstalled when the port is closed. The following code\(^1\) gets the old address and installs the new one.

\[
bVector = hPort->bIrq + 0x08, \\
hPort->pfOldHandler = DosGetIntVector(bVector); \\
DosSetIntVector(bVector, TransmitISR); \\
\]

bVector is the location in the interrupt vector table which contains the address of the interrupt service routine for IRQ7. It is passed to the function DosGetIntVector which returns the old interrupt service routine’s address. The function DosSetIntVector loads the bVector location with the address of the new interrupt service routine.

The DosGetIntVector and DosSetIntVector functions\(^2\) (Appendix C, lines 271-306) both use the ROM BIOS “int 21h” function call. When “int 21h” is executed with 35h the ah register, the DOS routine ‘get interrupt vector’ is called. The address of the current interrupt service routine for the location in the vector table specified by register al is returned in registers bx and es. Similarly, when “int 21 h” is executed with 25h in the ah register, the DOS routine ‘set interrupt vector’ is called. The address in the vector table at the location specified in register al will be set to the addresses in registers bx and ex. Once the new interrupt service routine has been installed, IRQ7 is enabled by clearing bit 7 of the PIC’s interrupt mask register.

\(^1\) This code is from page 77 of “Writing Windows™ VxDs and Device Drivers”, see reference 1.
\(^2\) The code for these functions is from pages 81 and 82 of “Writing Windows™ VxDs and Device Drivers”, see reference 1.
Start, Stop, and Continue Bytes

The text entered through the User Input Text Dialog Window is limited to an arbitrarily chosen twenty characters. The application appends a Start and a Stop Byte to the user input text before it is passed to the device driver for transmission. The Start and Stop Bytes frame the text just as start and stop bits frame an individual byte. The device driver uses the StartByte and StopByte to control the transmitting and receiving processes.

The StartByte is the first byte written to the parallel port during transmission. When the transmitter has finished transmitting the StartByte, it generates an interrupt. The Transmit Interrupt Service Routine responds to the interrupt by writing the next byte to the port. When the transmitter is done transmitting that byte, it generates another interrupt. In this manner, the Transmit Interrupt Service Routine writes each of the remaining bytes including the StopByte to the port. The StopByte indicates that there are no additional bytes to transmit so the Transmit Interrupt Service Routine resets the Enable IRQ bit in the Control Register (Appendix C, line 364). Consequently, the interrupt generated by the transmitter after it finishes transmitting the StopByte does not cause an interrupt.

The Transmit Buffer (Port1->Buffer, Appendix C, line 92) has a length of 22 bytes to store the user input characters, the StartByte, and the StopByte. Since a file most likely has more than 20 bytes, files are transmitted in multiple blocks using a

---

17 The StartByte is written by the PortWrite function in INTPAR.C. This is necessary to generate the first interrupt so that the Transmit Interrupt Service Routine can then write the remaining bytes.
ContinueByte\textsuperscript{18}. The application checks how many bytes are in the file. If there are more than 20, the application reads only the first 20 bytes from the file. It appends a StartByte and ContinueByte to these bytes. The Transmit Interrupt Service Routine handles the writing of these bytes to the port as described previously. However, when the Transmit Interrupt Service Routine detects the ContinueByte, it posts a message to the application (the ContinueByte is not transmitted). The application function OnNeedFileData (Appendix B, lines 549-553) responds to this message by reading the next 20 bytes from the file and adding the StartByte and ContinueByte. These bytes are then transmitted. In this manner, the entire file is transmitted in blocks of 20 bytes.

When the application reaches the end of the file, the StopByte is appended to the 20 or fewer remaining bytes. The Transmit Interrupt Service Routine will transmit the StopByte and reset the Enable IRQ bit in the Control Register.

The receiving process is simpler than the transmitting process because the receive buffer is large and only the StopByte is necessary to control the process. The Receive Buffer, Port1->RBuffer, holds 256 bytes (Appendix C, lines 93, 218, and 219). The Receive Interrupt Service Routine reads the bytes received by the parallel port adapter’s receiver from the Data Register (Appendix C, line 388). All the bytes except the StartByte and StopByte are copied into the Receive Buffer (Appendix C, line 396). When the StopByte is received, the Receive Interrupt Service Routine posts a message to the application (Appendix C, line 392). The application then copies the bytes from the

\textsuperscript{18}The transmit buffer could have been made very large enough that transmitting in blocks would not have been necessary for this work. It was made small to force transmitting files in blocks simulating how commercial software works with files of all sizes.
Receive Buffer to a temporary file (Appendix B, line 538). This design is adequate for demonstration purposes even though only 256 bytes or less can be received.

The StartByte, StopByte, and ContinueByte are bytes not used in the ASCII table. The StartByte is 1 or 00000001, the StopByte is 3 or 00000011, and the ContinueByte is 5 or 00000101. They are defined in INTPAR.H (Appendix C, lines 30-32). As an example, if the text is "This is a test", the string transmitted is "\x01This is a test\x03".

**The SafePageLock() Function**

The device driver’s code segment, data segment, and dynamically allocated memory are fixed, non-discardable, and page-locked. Fixed means that the Windows memory manager cannot move them to another location in memory. Non-discardable means that the Windows memory manager cannot simply delete them from memory and reload them from disk. Page-locked means that they cannot be swapped out of memory to disk and loaded back into memory at a later time. The fixed, non-discardable, and page-locked attributes prevent the memory manager from inadvertently losing data and prevent the system from inadvertently re-entering DOS (DOS is not reentrant). For a full discussion of why these attributes are necessary, refer to Hazzah [1].

The SafePageLock() function is used to make the device driver’s code segment (Appendix C, line 129), data segment (line 128), and dynamically allocated memory (line 125) fixed, non-discardable, and page-locked. The SafePageLock() function calls the Windows GlobalPageLock() function. GlobalPageLock() fixes and locks a memory segment. However, it moves the segment down to low memory which has undesired consequences. Windows and DOS use the low 1 Mb of memory for storing Program
Segment Prefix (PSP) data structures. These PSPs are created when an application is run. If the low memory is occupied by the device driver, its PSP cannot be created and it will not run. To prevent this problem, SafePageLock() initially allocates all of the low memory (Appendix C, lines 313-319). Then it page locks the memory of the selector passed to it using GlobalPageLock() (line 321) and frees the low memory (lines 323-328). Again, for more details refer to Hazzah [1].
CHAPTER 6
CONCLUDING REMARKS

The parallel port adapter, application, and device driver enable simple serial data communication using parallel ports. However, the implementations of the parallel port adapter and the device driver limit the speed, efficiency, and integrity of the communication process. Design features which would improve upon the current implementations are addressed herein.

A beneficial feature to add to the parallel port adapter would be the ability to transmit and receive data at different speeds. The parallel port adapter currently transmits and receives data at a fixed rate of 14.4 kbs. Recall that a 14.4 kHz shift clock signal (transmitter) and a 14.4 kHz sample clock (receiver) are generated by a counter and decoder connected to a 3.6868 MHz system clock. Changing the counter and decoder logic would change the generated clock rates. For example, the user could select one of a few speeds listed on the application’s Transmit Menu. The application would append a “speed” byte in addition to the start byte to the data. The parallel port adapter would have the logic circuitry to recognize the various speed bytes and change the counter/decoder logic as necessary. Since the receiver automatically detects incoming bytes, it would automatically detect the speed byte and make the corresponding adjustments to its counter/decoder logic.
Another feature that could be added to the parallel port adapter is parity checking to detect transmitting and receiving errors. The simplest implementation would be to fix the parity to either odd or even in the transmitter and receiver hardware (i.e., the user would not be able to select the type of parity). The transmitter hardware would add either a 1 or 0 to the bit 7 of the current data byte to achieve the desired parity and transmit the data byte. Since the parity is fixed, the receiver would check the parity of the incoming data. If the parity is incorrect, the buffer would be loaded with a “parity error” byte and an interrupt signal would be generated. The device driver would recognize that a parity error occurred and post a message to the application. The application would notify the user that an error occurred. With the necessary hardware, the receiver could also signal the transmitter that a parity error occurred. The transmitter would force a high to low transition on the Error pin (pin 15) of the parallel port and generate an interrupt. The interrupt service routine would recognize that an error occurred and retransmit the current block of data.

Two aspects of the device driver design decrease the speed at which the driver services the parallel port adapter. The first is the buffer implementation. The transmit buffer is filled by the application and emptied by the device driver. It may then be filled again by the application if there additional data to transmit. Likewise, the receive buffer is filled by the device driver and emptied by the application. It would be much more efficient to implement the transmit and receive buffers as circular buffers which could be read from and written to with optimal synchronization. For example, the Transmit Interrupt Service routine would continuously write characters to the port. While the
transmitter is converting the data to a serial bit stream, the application could refill the buffer without affecting the rate at which characters are transmitted.

The second reason that the device driver is “slow” is a limitation of Windows device drivers in general. The interrupt service routines are not actually writing to and reading from the parallel port’s registers directly. There is a layer of virtualizing software Windows inserts between the interrupt service routine code and the hardware. This virtualizing software introduces unavoidable delays. The only way to eliminate these delays is to not use a Windows device driver. DOS device drivers are quite efficient but are extremely difficult to interface to Windows applications. Virtual device drivers (VxDs) are also efficient and are readily interfaced to Windows applications. However, their implementation is quite complex and requires thorough knowledge of the Virtual Machine Manager. Since the Windows device driver development is comparatively straightforward, it was implemented.
Altera HDL Code for Transmitter:

SUBDESIGN 'Trans'
(Clock,Strobe,DataIn[7..0]:INPUT;
ShiftClock,DataSent,XmitReg[9..0]:OUTPUT;
)

VARIABLE
Edge[1..0]:DFF;
InBuf[7..0]:DFF;
XmitReg[9..0]:DFF;
InBuffFull:DFF;
XmitRegFull:DFF;
ShiftCounter[7..0]:DFF;
BitCounter[3..0]:DFF;
Load:NODE;
LoadXmitReg:NODE;
WordFrame:NODE;

BEGIN
Edge[1..0].clk=Clock;
InBuf[7..0].clk=Clock;
ShiftCounter[7..0].clk=Clock;

Edge[1] d = Strobe;
Edge[0].d = Edge[1].q;
Load = !Edge[1] & Edge[0];

InBuf[7..0].d = ( DataIn[7..0] & Load) OR ( InBuf[7..0].q & !Load);

WordFrame = LoadXmitReg OR XmitRegFull;
IF WordFrame THEN
    ShiftCounter[] d = ShiftCounter[] q + 1;
ELSE
    ShiftCounter[7..0].d = GND;
END IF;
Transmitter cont...

IF ((ShiftCounter[] == H"00") AND WordFrame) THEN
    ShiftClock=VCC; % decoder to generate ShiftClock from ShiftCounter
ELSE
    ShiftClock=GND;
END IF;

XmitReg[9..0].clk=ShiftClock; % XmitReg uses ShiftClock to xmit at 14.4kbps
XmitReg[9].d = VCC;
XmitReg[8..1].d = (LoadXmitReg & InBuf[7..0]) OR (XmitRegFull & XmitReg[9..2]);
XmitReg[0].d = (LoadXmitReg & GND) OR (XmitRegFull & XmitReg[1]);

BitCounter[3..0].clk=ShiftClock; % BitCounter counts # of bits transmitted so % know when done.
BitCounter[3..0].clm=WordFrame;
BitCounter[3..0].d = BitCounter[3..0].q+1;

IF BitCounter[]=11 THEN % 11 = 10 bits plus ShiftClock to load XmitReg
    DataSent=VCC;
ELSE
    DataSent=GND;
END IF;

InBufFull.clk=Clock;
XmitRegFull.clk=Clock;

InBufFull.d = Load OR (InBufFull & !LoadXmitReg);
LoadXmitReg = InBufFull & !XmitRegFull;
XmitRegFull.d = LoadXmitReg OR (XmitRegFull & !DataSent);

END:
Altera HDL Code for Receiver:

SUBDESIGN 'Receiver'

Clock, DataIn: INPUT;
Buff[7..0]: OUTPUT;
)

VARIABLE
Edge[1..0]: DFF;
EdgeDet: NODE,
WordFrame: DFF;
InReg[7..0]: DFF;
Buff[7..0]: LATCH;
SampleCounter[7..0]: DFF;
BitCounter[3..0]: DFF;
LastBit: NODE;
SampleClock: NODE;

BEGIN
Edge[1..0].clk=Clock;
WordFrame clk=Clock;
SampleCounter[7..0].clk=Clock;

Edge[1].d = DataIn;
Edge[0].d = Edge[1].q;
EdgeDet = Edge[1] & !Edge[0];

WordFrame d = EdgeDet # (WordFrame & !LastBit);

IF WordFrame THEN
    SampleCounter[] d = SampleCounter[] q+1;
ELSE
    SampleCounter[7..0].d = GND;
END IF;

IF SampleCounter[]="H'80" THEN
    SampleClock = VCC;
ELSE
    SampleClock = GND;
END IF;

InReg[7].d = DataIn;
InReg[6].d = InReg[7];
InReg[5].d = InReg[6];
InReg[4].d = InReg[5];
InReg[3].d = InReg[4];
InReg[2].d = InReg[3];
InReg[1].d = InReg[2];
InReg[0].d = InReg[1];
InReg[7..0].clk = SampleClock;

% detect StartBit
% WordFrame high while receiving a byte
% while receiving, increment SampleCounter
% to generate SampleClock
% "80" so sample in middle of bit
% when SampleClock high, shift in next bit
% shift register construction
Receiver cont:

Buff[7..0].d = InReg[7..0];
Buff[7..0].ena = LastBit;

BitCounter[3..0] clk = SampleClock;
BitCounter[3..0].clrn = WordFrame;
BitCounter[3..0].d = BitCounter[3..0].q + 1;

IF BitCounter[] = 9 THEN
    LastBit = VCC;
ELSE
    LastBit = GND;
END IF;

END,
APPENDIX B

DRIVAPP.H

// main header file for the DRIVER application
#include "resource.h"    // main symbols
#include "status.h"      // define "C"
extern "C"
{
    #include "\intpar\intpar.h"
}
#define DISPLAY 0x01       // flags for painting screen
#define CLEAR 0x00

class CDriverApp : public CWinApp
{
    public:
    virtual BOOL InitInstance();
};
class CDriverWindow : public CFrameWnd
{
    public:
    RECT StatRect;       // rect for status bar
    CString StatusMessage;  // message to be displayed in status bar
    CDriverWindow();
    ~CDriverWindow();

    protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    int OnCreate(LPCREATESTRUCT lpCreateStruct);
    void OnSize(UINT nType, int cx, int cy);

    // functions to "grey" menu items
    void OnUpdateMenuOpenPort(CCmdUI *cmd);
    void OnUpdateMenuTransmitting(CCmdUI *cmd);
    void OnUpdateMenuTransmittingFile(CCmdUI *cmd);
    void OnUpdateMenuCloseReceiving(CCmdUI *cmd);
    void OnUpdateMenuReceiving(CCmdUI *cmd);

    // functions to handle menu selections
    void OnExit();
    void OnDriverXmitUserInput();
    void OnTransmitClosePort();

void OnTransmitInitPort();
void OnReceiveClosePort();
void OnReceiveOpenPort();
void OnReceiveData();
void OnOpenFile();
void OnTransmit();
void OnCloseFile();
void OnPaint();
void OnTransmitFile();
void OnSaveFile();
void OnClear();

// functions to handle messages posted by ISR
LRESULT OnReceiveData(WPARAM wParam, LPARAM lParam);
LRESULT OnNeedFileData(WPARAM wParam, LPARAM lParam);

DECLARE_MESSAGE_MAP()

private:
BOOL mode;           // flags for "greying" of menu items,
int ValidOpen;       // determining status,
BOOL transmitting;   // and what to display
BOOL receiving;
BOOL portopen;
BOOL fileopen;
BOOL datareceived;

HPORT hPort_Transmit;
CString FileName;     // name of file to be xmitted
CFile Transmit_File;  // CFile object for xmit file
CFile *Transmit_File_Display_ptr;    // display pointer for xmit file
CFile Transmit_File_Display;         // pointer for xmit file
long FilePosition;                  // position in file to xmit

HPORT hPort_Receive;
CFile Receive_File;                // CFile object for temp rcv file
CFile *Receive_File_Display_ptr;   // display pointer for rcv file
char *pTempRecvFileName;           // filename of temp rcv file
HWND hWndVw;                       // handle to this app
CStatus *StatusBar;                // pointer to status bar
}

DRIVAPP.CPP

// the main application

#include "stdafx.h"
#include "driver.h"
#include "dlguser.h"
extern "C"
{
    #include <conio.h>
    #include <stdio.h>
    #include <string.h>
}

// Initialization Stuff
CDriverApp theApp;

BOOL CDriverApp::InitInstance()
{
    CDriverWindow* pFrame = new CDriverWindow;
    if (!pFrame->LoadFrame(IDR_FRAME))
        return FALSE;
    pFrame->ShowWindow(m_nCmdShow);
    pFrame->UpdateWindow();
    m_pMainWnd = pFrame;
    return TRUE;
}

CDriverWindow::CDriverWindow() // constructor: used for initialization
{
    transmitting = FALSE; //initialize "menu greying" states
    receiving = FALSE;
    portopen = FALSE;
    fileopen = FALSE;
    datareceived = FALSE;
    Transmit_File_Display_ptr = NULL;
    StatusBar = NULL;
    StatusMessage = "Inactive";
    FilePosition = 0L;
}

CDriverWindow::~CDriverWindow() // destructor
{
}

BOOL CDriverWindow::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.y = 5;
    cs.x = 5; // set size and location of window
    cs.cx = 600;
    cs.cy = 400;
    return CFrameWnd::PreCreateWindow(cs);
}

int CDriverWindow::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // display status bar
}
if (CFrameWnd::OnCreate(lpCreateStruct)!=-1)
    return -1;
RECT rc;
rc.top=rc.bottom-25; // position of status bar
StatusBar = new CStatus(this); // create status bar object
BOOL ret=StatusBar->Create(NULL,"Status",WS_CHILD|WS_BORDER|
    WS_VISIBLE,rc,this,-1);
return 0;

void CDriverWindow::OnSize(UINT nType, int cx, int cy)
{
    // resizes status bar if app window is resized
    RECT rc;
    CFrameWnd::OnSize(nType,cx,cy);
    GetClientRect(&rc);
    MoveWindow(StatusBar->m_hWnd,0,rc.bottom-25,rc.right,25,TRUE);
}

void CDriverWindow::OnPaint()
{
    // handles the painting of the application window
    int line,x,y;
    char Buf[256],Buf2[256];
    int nBytes,BufPosition;
    BOOL AllCharSent;
    BOOL AtLeastOneCR;
    CFrameWnd::OnPaint(); // do CFrameWnd's default OnPaint
    CClientDC dc(this);
    if (mode==DISPLAY && transmitting==TRUE)
    {
        // display contents of file to be transmitted
        if (Transmit_File_Display_ptr != NULL)
            Transmit_File_Display_ptr->SeekToBegin();
        line=5;
        UINT NBytes = Transmit_File_Display_ptr->Read(Buf,256);
        nBytes = NBytes;
        do
        {
            for (x=0,y=0; x<nBytes;x++)
            {
                AllCharSent=FALSE;
            }
if (Buf[x]>=20) && (Buf[x]<=126) 
    Buf2[y++] = Buf[x];
if (Buf[x]==0x0A) 
    
    Buf2[y++] = 0x00;
    dc.TextOut(5, line, Buf2, strlen(Buf2));
    y = 0;
    line += 20;
    AllCharSent = TRUE;
    BufPosition = x;

if (AllCharSent == FALSE) 
{ 
    long ChangeFilePosition = BufPosition - nBytes + 1;
    Transmit_File_Display_ptr->Seek(ChangeFilePosition, CFile::current);

    NBytes = Transmit_File_Display_ptr->Read(Buf, 256);
    nBytes = NBytes;
} while (nBytes > 0);

if (mode == DISPLAY && datareceived == TRUE) 
{ // display received data
    AtLeastOneCR = FALSE;
    if (Receive_File_Display_ptr != NULL) 
        Receive_File_Display_ptr->SeekToBegin();

    line = 5;
    UINT NBytes = Receive_File_Display_ptr->Read(Buf, 256);
    nBytes = NBytes;
    do 
    { 
        for (x = 0, y = 0, x < nBytes; x++) 
            
            AllCharSent = FALSE;
            if (Buf[x]>=20) && (Buf[x]<=126) 
                
                Buf2[y++] = Buf[x];
            } 
            if (Buf[x] == 0x0A) 
                
                AtLeastOneCR = TRUE;
                Buf2[y++] = 0x00;
                dc.TextOut(5, line, Buf2, strlen(Buf2));
                y = 0;
                line += 20;
                AllCharSent = TRUE;
                BufPosition = x;
        
    } 

}
if (nBytes<256) && (AllCharSent==FALSE) && (AtLeastOneCR == FALSE) )
    {
        Buf2[y++]=0x00;
        dc.TextOut(5, line, Buf2, strlen(Buf2));
    }

if((nBytes<256) && (AllCharSent==FALSE) && (AtLeastOneCR = TRUE))
    {
        long ChangeFilePosition=BufPosition-nBytes+1;
        Receive_File_Display_ptr->Seek(ChangeFilePosition, CFile::current);
    }

NBytes = Receive_File_Display_ptr->Read(Buf,256);
nBytes=NBytes;

} while (nBytes > 0);

ValidateRect(NULL);

void CDriverWindow::OnUpdateMenuOpenPort(CCmdUI *cmd)
{
    if (transmitting && !receiving && !portopen && !fileopen)
        cmd->Enable(TRUE);
    else
        cmd->Enable(FALSE);
}

void CDriverWindow::OnUpdateMenuTransmitting(CCmdUI *cmd)
{
    if (transmitting && !receiving && portopen && !fileopen)
        cmd->Enable(TRUE);
    else
        cmd->Enable(FALSE);
}

void CDriverWindow::OnUpdateMenuTransmittingFile(CCmdUI *cmd)
{
    if (transmitting && !receiving && portopen && fileopen)
        cmd->Enable(TRUE);
    else
        cmd->Enable(FALSE);
}

void CDriverWindow::OnUpdateMenuCloseReceiving(CCmdUI *cmd)
{
    if (transmitting && receiving && portopen && !fileopen)
        cmd->Enable(TRUE);
    else
        cmd->Enable(FALSE);
}
void CDriverWindow::OnUpdateMenuReceiving(CCmdUI *cmd)
{
    // handles greying of menu items - only Receiving menu items are available
    if (!transmitting && receiving && portopen && fileopen)
        cmd->Enable(TRUE);
    else
        cmd->Enable(FALSE);
}

void CDriverWindow::OnExit()
{
    // called when Exit menu item selected
    SendMessage(WM_CLOSE);
}

void CDriverWindow::OnTransmitInitPort()
{
    // called when Initialize Parallel Port from Transmit Menu selected
    // call function in DLL, DeviceOpenXmit, which initializes parallel port for transmitting
    hDevice_Transmit = DeviceOpenXmit(this->m_hWnd);
    transmitting = TRUE;  // set status flags
    portopen = TRUE;
    StatusMessage = "In transmitting mode ";  // change status bar's message
    StatusBar->InvalidateRect(&StatRect,TRUE);
}

void CDriverWindow::OnTransmitClosePort()
{
    // called when Close Parallel Port from Transmit Menu selected
    // call function in DLL, DeviceCloseXmit, which resets parallel port to its default settings
    DeviceCloseXmit(hDevice_Transmit);
    transmitting = FALSE;  // set status flags
    portopen = FALSE;
    StatusMessage = "Inactive ";  // change status bar's message
    StatusBar->InvalidateRect(&StatRect,TRUE);
}

void CDriverWindow::OnDriverXmitUserInput()
{
    // called when Transmit User Input from Transmit Menu selected
    char XmitBuffer[22];
    int NumBytes, i;
    CDrvUser DlgUser;  // create dialog box object
    if (DlgUser.DoModal()==TRUE)  // display dialog box
    {
        // user input string is returned in m_edit_strg_to_xmit
        NumBytes = DlgUser.m_edit_strg_to_xmit.GetLength();
    }
if( NumBytes > 0 )
   { // if something to xmit, copy into a xmit buffer
      XmitBuffer[0] = START_BYTE; // add start byte to xmit buffer
      NumBytes+=1;
      for( i=1; i<NumBytes; i++ ) // copy data into xmit buffer
         XmitBuffer[i] = DlgUser.m_edit_strg_to_xmit[i-1];
      XmitBuffer[NumBytes++] = STOP_BYTE; // add stop byte
      DeviceWrite( hDevice_Transmit, (LPBYTE) (const char *) XmitBuffer, (LPWORD) &NumBytes);
   } // call function in DLL which actually xmits the user input data

void CDriverWindow::OnOpenFile()
   { // called when Open File from Transmit Menu selected
      CFileDialog dlg(TRUE,NULL,"*. *"); // create Open File common dialog
      if dlg.DoModal() == IDOK ) // display Open File common dialog
      {
         FileName = dlg.GetPathName(); // get file to open
         ValidOpen = Transmit_File.Open(FileName,CFile::modeRead); // open file
         Transmit_File_Display_ptr = Transmit_File.Duplicate();
         fileopen=TRUE; // make duplicate pointer for displaying file
         mode=DISPLAY; // set flags so file will be displayed
         InvalidateRect(NULL,TRUE); // force OnPaint to be called
      }
   }

void CDriverWindow::OnTransmitFile()
   { // called when Transmit File from Transmit Menu selected
      char FileBuf[20]; //temp buffer for chars read from file
      char XmitBuf[22]; //buffer for chars and start and stop bytes
      int nXmitBytes; //# bytes to xmit

      if( FilePosition == 0 ) // set "pointer" at beginning of file
         Transmit_File.SeekToBegin();
      else //set "pointer" at current FilePosition
         Transmit_File.Seek(FilePosition, CFile::begin);

      // read 20 or until EndOfFile chars into FileBuf
      UINT nBytes = Transmit_File.Read(FileBuf,20);
      nXmitBytes = (int)nBytes; // store # bytes to xmit

      if( (nBytes >0) && (nBytes <20) ) //less than 20 chars to xmit
         {
            XmitBuf[0] = START_BYTE; //copy chars FileBuf to XmitBuf
            nXmitBytes+=1; // and add start and stop bytes
            for( int i=1; i< nXmitBytes; i++ )
               XmitBuf[i] = FileBuf[i-1];
            XmitBuf[nXmitBytes++] = STOP_BYTE; // reset FilePosition to beginning
            FilePosition = 0L;
            DeviceWrite( hDevice_Transmit, (LPBYTE) (const char *) XmitBuf,
                        (LPWORD) &nXmitBytes);
         } // call function in DLL to transmit file data
else //exactly 20 bytes were read from file
    [ // transmit these 20 bytes and record position in file so
        XmitBuf[0]=START_BYTE, // remaining chars transmitted when these done
        nXmitBytes+=1;
        for( int i=1; i< nXmitBytes; i++ )
            XmitBuf[i] = FileBuff[i-1];
        XmitBuf[nXmitBytes++] = CONTINUE_BYTE,
        FilePosition += nBytes; // record current file position
        DeviceWrite( hDevice_Transmit, (LPBYTE) (const char *) XmitBuf,
                    (LPWORD) &nXmitBytes);
    ]
    }

void CDriverWindow::OnCloseFile()
    { // called when Close File from Transmit Menu selected
        fileopen=FALSE; // set flags so correct menu items "grayed"
        mode = CLEAR; // and OnPaint does not try to display file
    }

void CDriverWindow::OnReceiveOpenPort()
    { // called when Initialize Port and Receive Data on Receive Menu selected
        hWndVw=THIS->h_Wnd;
        receiving = TRUE; // set flags for correct "greying"
        portopen = TRUE; // of status message and received
        fileopen = FALSE; // data is displayed in OnPaint
        StatusMessage = "In receiving mode.", // change status message
        statusBar->InvalidateRect(&StatRect,TRUE); // force call to OnPaint
        hDevice_Receive = DeviceOpenRecv(THIS->h_Wnd);
        // call function in DLL which initializes parallel port for receiving data
    }

void CDriverWindow::OnReceiveClosePort()
    { // called when Close Parallel Port on Receive Menu selected
        DeviceCloseRecv( hDevice_Receive), // call function in DLL
               // which resets parallel port to default setting
        receiving = FALSE; // set flags
        portopen = FALSE;
        StatusMessage = "Inactive.", // change status message
        statusBar->InvalidateRect(&StatRect,TRUE); // force call to OnPaint
    }

void CDriverWindow::OnSaveFile()
    { // called when Save Received Data in a File on Receive Menu selected
        CFileDialog dlg(FALSE,NULL,"*.txt"); //create common dlg object
if (dlg.DoModal() == IDOK)  //display Save As common dlg window
{
    CString SaveAsFileName = dlg.GetPathName();  // get filename
    fileopen=FALSE;  //set flags
datareceived=FALSE;
    Receive_File_Display_ptr->Close();  // close temp file holding
    Receive_File.Close();  // received data
    CFile: Rename(pTempRecvFileName, (const char *) SaveAsFileName);
    hDevice_Receive->RLocation = 0;
    hDevice_Receive->RNumBytes = 0;
    mode=CLEAR;  // set mode so that OnPaint will clear App's window
    InvalidateRect(NULL,TRUE);  //force call to OnPaint
}
else
{
    OnClear();  // if don't save file, clear it
}

void CDriverWindow::OnClear()
{
    Receive_File_Display_ptr->Close();  // clear duplicate pointer
    Receive_File.Close();  //close temp file holding received data
    fileopen=FALSE;  //set flags
datareceived=FALSE;
    mode=CLEAR;  // set mode so OnPaint will clear App's window
    InvalidateRect(NULL,TRUE);  //force call to OnPaint
    hDevice_Receive->RLocation = 0;
    hDevice_Receive->RNumBytes = 0;
}

LRESULT CDriverWindow::OnReceiveData(WPARAM wParam, LPARAM lParam)
{  // called in response to message posted by the ISR: data received
    const char far* Receive_Buf;
    CFileException e;
    Receive_Buf = hDevice_Receive->RBuffer;  //copy received chars into buffer
    pTempRecvFileName = "c:\temp.txt";  //create temp file name
datareceived=TRUE;  // set flags
    fileopen=TRUE;
    // Open a temp file and write into it the chars in the buffer
    Receive_File.Open(pTempRecvFileName, CFile::modeCreate | CFile::modeReadWrite | CFile::shareCompat, &e);
    Receive_File.Write(Receive_Buf, (UINT)hDevice_Receive->RNumBytes);
Receive_File_Display_ptr = Receive_File.Duplicate(); // create duplicate file

// pointer for OnPaint to use to display received chars

mode=DISPLAY; // set mode so OnPaint displays received chars in App's window
InvalidateRect(NULL,TRUE); // force call to OnPaint

return 0L;

} // called CDriverWindow::OnNeedFileData(WPARAM wParam, LPARAM lParam)

BEGIN_MESSAGE_MAP(CDriverWindow, CFrameWnd)

ON_COMMAND(ID_EXIT, OnExit)

ON_COMMAND(ID_DRIVER_XMIT_USER_INPUT, OnDriverXmitUserInput)

ON_COMMAND(ID_TRANSMIT_CLOSE_PORT, TransmitClosePort)

ON_COMMAND(ID_TRANSMIT_INIT_PORT, TransmitInitPort)

ON_COMMAND(ID_RECEIVE_CLOSEPARALLELPORT, ReceiveClosePort)

ON_COMMAND(ID_RECEIVE_OPENDEVICE, ReceiveOpenPort)

ON_COMMAND(ID_TRANSMIT_FILE, TransmitFile)

ON_COMMAND(ID_SAVE_FILE, SaveFile)

ON_COMMAND(IDM_CLEAR, Clear)

ON_WM_PAINT()

ON_WM_CREATE()

ON_WM_SIZE()

ON_WM_DESTROY()

ON_MESSAGE(FILE_RECEIVED_MSG, OnFileReceived)

ON_MESSAGE(NEED_FILE_DATA_MSG, OnNeedFileData)

ON_UPDATE_COMMAND_UI(ID_TRANSMIT_INIT_PORT, TransmitInitPort)

ON_UPDATE_COMMAND_UI(ID_RECEIVE_OPENDEVICE, ReceiveOpenPort)

ON_UPDATE_COMMAND_UI(ID_TRANSMIT_CLOSE_PORT,
TransmitClosePort)

ON_UPDATE_COMMAND_UI(ID_DRIVER_XMIT_USER_INPUT,
DriverXmitUserInput)

ON_UPDATE_COMMAND_UI(ID_OPEN_FILE, OpenFile)

ON_UPDATE_COMMAND_UI(ID_TRANSMIT_FILE, TransmitFile)

ON_UPDATE_COMMAND_UI(ID_SAVE_FILE, SaveFile)

ON_UPDATE_COMMAND_UI(IDM_CLEAR,
Clear)

ON_WM_DESTROY()
APPENDIX C

PAR.H

#define PAR_CON_REG 0x02 // control register
#define PAR_STAT_REG 0x01 // status register
#define PAR_DATA_REG 0x00 // data register
#define ENABLE_IRQ 0x10 // set parallel port control register bits
#define DISABLE_IRQ 0x00
#define PORT_IN 0x20
#define PORT_OUT 0x00
#define STROBE_TRUE 0x01
#define STROBE_FALSE 0x00

INTPAR.H

#include <stdio.h>

#define MASTER_PIC_CTRL 0x20 // PIC address
#define MASTER_PIC_MASK 0x21
#define EOI 0x20
#define START_BYTE 0x01
#define STOP_BYTE1 0x03
#define CONTINUE_BYTE 0x05
#define SET(value, mask) value |= mask
#define CLR(value, mask) value &= (~mask)
#define FILE_RECEIVED_MSG WM_USER
#define NEED_FILE_DATA_MSG WM_USER+1

typedef void (FAR interrupt *VOIDINTPROC)();
typedef struct
{
    WORD PortBase; // port base address
    BYTE bIrq; // IRQ line number
    HWND hwnd; // window handle of calling application
    VOIDINTPROC pOldHandler; // default ISR address
    char far *Buffer; // pointer to xmit buffer
    WORD BufferLength; // xmit buffer length
    WORD Location; // pointer to current location in buffer
    WORD NumBytes; // number of bytes to xmit
    char far *RBuffer; // pointer to rcv buffer
}
WORD RBufferLength;  // rcv buffer length
WORD RLocation;  // pointer to current location in buffer
WORD RNumBytes;  // number of bytes to recv

} PORTCONTEXT, FAR *HPORT;

HPORT FAR PASCAL PortOpenXmit( HWND hwnd );
int FAR PASCAL PortCloseXmit(HPORT);
int FAR PASCAL PortWrite( HPORT, LPBYTE lpData, LPWORD pcBytes );
HPORT FAR PASCAL PortOpenRecv( HWND hwnd );
int FAR PASCAL PortCloseRecv( HPORT hPort );

extern PORTCONTEXT Port1;

ISR.H

void interrupt FAR TransmitISR( void );  // Transmit Interrupt Service Routine
void interrupt FAR ReceiveISR( void );  // Receive Interrupt Service Routine

INPAR.C

#include <dos.h>
#include <conio.h>
#include <windows.h>
#include <windowsx.h>
#include <memory.h>
#include <string.h>
#include <intpar.h>
#include "isr.h"
#include "par.h"

#define DOS_GET_INT_VECTOR 0x35
#define DOS_SET_INT_VECTOR 0x25

PORTCONTEXT Port1 = { 0x378, 7, NULL };  // parallel port LPT1, IRQ line 7
WORD BufferLengthT = { 22 };  // buffer size = 22
WORD BufferLengthR = { 256 },

// function declarations
VOIDINTPROC DosGetIntVector( BYTE Irq );
void DosSetIntVector( BYTE Irq, VOIDINTPROC pHandler );
void interrupt FAR DeviceIsrXmit( void );
void interrupt FAR DeviceIsrRecv( void );
UINT SafePageLockf( HGLOBAL sel );

HPORT FAR PASCAL _export PortOpenXmit( HWND hwnd )

// This function initializes LPT1 for transmitting.
// Two important actions:
// 1) set LPT1 interrupt vector to the transmit ISR DeviceIsrXmit
// 2) set LPT1 port direction to out and enable interrupts

HPORT  hPort;
BYTE   bVector, mask;
WORD   mycs, myds;
hPort = &Port1;
hPort->BufferLength = BufferLengthT;  // set Port1's trans buffer length
hPort->Location = 0;    // point to start of buffer
hPort->hwnd = hwnd;    // apps window handle

// lines 121-149 from "Writing Windows VxD and Device Drivers", see
// reference 1
hPort->Buffer = GlobalAllocPtr( GMEM_SHARE | GMEM_MOVEABLE | GMEM_NODISCARD, hPort->BufferLength );
if(!hPort->Buffer) {
    OutputDebugString( "ERROR GlobalAlloc Rx
");
    return (hPort)-1;
}
SafePageLock( (HGLOBAL)SELECTOROF( hPort->Buffer ));
asm mov myds, ds
asm mov mycs, cs
SafePageLock( myds );
SafePageLock( mycs );

// Configure parallel port
_outp( hPort->PortBase+PAR_CON_REG, ENABLE_IRQ | STROBE_FALSE | PORT_OUT );

mask = _inp( MASTER_PIC_MASK );    // get contents of PIC's mask register
_SET( mask, 1 << hPort->bIrq );    // set bit 7 to 1 to disable IRQ7
_outp( MASTER_PIC_MASK, mask );

// enable new interrupt handler
bVector = hPort->bIrq + 0x08;    // interrupt vector table location of IRQ7
hPort->pfOldHandler = DosGetIntVector( bVector );    // get address of current ISR
DosSetIntVector( bVector, TransmitISR );    // install address of new ISR

// Unmask IRQ7 at PIC
mask = _inp( MASTER_PIC_MASK );    // get contents of PIC's mask register
_CLR( mask, (1 << hPort->bIrq));    // clear bit 7 to enable IRQ7
_outp( MASTER_PIC_MASK, mask );

return hPort;
}

int FAR PASCAL _export PortCloseXmit( HPORT hPort )

// This function resets the parallel port
// It performs these 2 important actions:
// 1) disables interrupts for LPT1
// 2) sets the interrupt vector to the default address
BYTE mask, bVector;

// Disable parallel port interrupt
_outp( hPort->PortBase + PAR_CON_REG, DISABLE_IRQ | STROBE_FALSE | PORT_OUT);

// lines 167-174 from "Writing Windows VxD and Device Drivers", see
// reference 1
// mask the interrupt
mask = __inp( MASTER_PICMASK);
SET( mask, 1 << hPort->bIrq );
_outp( MASTER_PICMASK, mask );
bVector = hPort->bIrq + 0x08;
DosSetlntVector( bVector, hPort->pfOldHandler ); // reset interrupt vector
GlobalFreePtr( hPort->Buffer );
return 0;
}

int FAR PASCAL _export PortWrite( HPORT hPort, LPBYTE lpData, LPWORD pcBytes )
{
    int i;
    for (i=0; i < *pcBytes; i++)
        hPort->Buffer[i] = lpData[ i ]; // copy data to be xmitted into buffer
    hPort->Location=0; // Location is pointer to next char to xmit
    hPort->NumBytes=(*pcBytes);
    // transmit the first char in Buffer (start byte)
    _outp( 0x037A, ENABLE_IRQ | STROBE_FALSE | PORT_OUT );
    _outp( 0x0378, hPort->Buffer[0]);
    _outp( 0x037A, ENABLE_IRQ | STROBE_TRUE | PORT_OUT );
    _outp( 0x037A, ENABLE_IRQ | STROBE_FALSE | PORT_OUT );
    hPort->Location++; // point to next char to xmit
    return 0;
}

HPORT FAR PASCAL _export PortOpenRecv( HWND hwnd )
ivec // This function initializes LPT1 for receiving.
// Two important actions:
    // 1) set LPT1 interrupt vector to the receive ISR DeviceIsrRecv
    // 2) set LPT1 port direction to in and enable interrupts

HPORT hPort;
BYTE bVector, mask;
WORD mycs, myds;
hPort = &Port1;

hPort->RBufferLength = BufferLengthR; // length of receive buffer
hPort->RLocation=0; // location pointer at beginning of buffer
hPort->RNumBytes=0; // number of bytes received = 0
hPort->hwnd = hwnd;

// Configure parallel port - enable interrupts and port direction in
_outp(hPort->PortBase+PAR_CON_REG, ENABLE_IRQ | PORT_IN);

// lines 221-250 from “Writing Windows VxD and Device Drivers”, see
// reference 1

hPort->RBuffer = GlobalAllocPtr( GMEM_SHARE | GMEM_MOVEABLE |
    GMEM_NODISCARD, hPort->RBufferLength );

if (!hPort->RBuffer)

    OutputDebugString( "ERROR GlobalAlloc Rx\n");
    return (hPort)-1;

SafePageLock( (HGLOBAL)SELECTOROF( hPort->RBuffer ));

_asm mov myds, ds
_asm mov mycs, cs
SafePageLock( myds );
SafePageLock( mycs );

// mask the IRQ7 at PIC
mask = _inp( MASTER_PIC_MASK );
SET( mask, 1 << hPort->bIrq );
_outp( MASTER_PIC_MASK, mask );

// enable new interrupt handler
bVector = hPort->bIrq + 0x08;
 inspected vector table location of IRQ7
hPort->pfOldHandler = DosGetIntVector( bVector );
DosSetIntVector( bVector, ReceiveISR );

// Unmask IRQ7 at PIC.
mask = _inp( MASTER_PIC_MASK );
CLR( mask, (1 << hPort->bIrq) );
_outp( MASTER_PIC_MASK, mask );

return hPort;

int FAR PASCAL _export PortCloseRecv( HPORT hPort )
{
    BYTE mask, bVector;

    // Disable parallel port interrupt - disable interrupts and port direction out
    _outp( hPort->PortBase + PAR_CON_REG, DISABLE_IRQ | PORT_OUT);

    // lines 266-275 from “Writing Windows VxD and Device Drivers”, see
    // reference 1

    // mask the interrupt
    mask = _inp( MASTER_PIC_MASK );
    SET( mask, 1 << hPort->bIrq );
outp( MASTER_PIC_MASK, mask );
bVector = hPort->bIrq + 0x08;
DosSetIntVector( bVector, hPort->pfOldHandler ); // reset interrupt vector
GlobalFreePtr( hPort->Buffer );
return 0;

// lines 283-341 from "Writing Windows VxD and Device Drivers", see
// reference 1

VOIDINTPROC DosGetIntVector( BYTE vector )
{
WORD selHandler, offHandler;
_asm
{
    mov al, vector  // vector table location of IRQ7
    mov ah, DOS_GET_INT_VECTOR  // load ah with 35h
    push es
    int 21h  // call ROM BIOS function 'Get Vector'
    mov offHandler,bx  // address of current interrupt service
    mov selHandler,es    // routine returned in bx & es
    pop es
}
return( MAKELP( selHandler, offHandler ) );  // return address of current ISR
}

void DosSetIntVector( BYTE vector, VOIDINTPROC pHandler )
{
WORD offHandler, selHandler;
_selHandler = SELECTOROF( pHandler );
_offHandler = OFFSETOF( pHandler ),
_asm
{
    mov al, vector  // vector table location of IRQ7
    mov ah, DOS_SET_INT_VECTOR  // load ah with 25h
    mov dx, offHandler  // load dx & bx with address of new ISR
    mov bx, selHandler
    push ds
    mov ds, bx
    int 21h  // call ROM BIOS function 'Set Vector'
    pop ds    // which installs new ISR
}
}

UINT SafePageLock( HGLOBAL sel )
{
WORD i, rc,
static WORD SelArray[ 1024 ];
memset(SelArray, 1024 * sizeof(WORD), 0);
for (i=0; i<1024; i++)
{
    SelArray[i] = LOWORD(GlobalDosAlloc(1024));
    if (!SelArray[i])
        break;
}
rc = GlobalPageLock(sel);
for (i=0; i<1024; i++)
{
    if (!SelArray[i])
        break;
    GlobalFree(SelArray[i]);
}
return rc;

ISR.C
#include <conio.h>
#include <windows.h>
#include <dos.h>
#include "intpar.h"
#include "par.h"
int i = 0;

void interrupt FAR TransmitISR (void)
{
    LPBYTE buf;
    PORTCONTEXT *hPort;
    hPort = &Port1;
    buf = hPort->Buffer;
    if ( (buf[hPort->Location] != STOP_BYTE1) && (buf[hPort->Location] != CONTINUE_BYTE) )
        // simply output char pointed to by Location to LPT1
        _outp(hPort->PortBase+PAR_DATA_REG, buf[hPort->Location++]);
        _outp(hPort->PortBase+PAR_CON_REG, ENABLE_IRQ | STROBE_TRUE | PORT_OUT);
        _outp(hPort->PortBase+PAR_CON_REG, ENABLE_IRQ | STROBE_FALSE | PORT_OUT);
    else if ( buf[hPort->Location] == STOP_BYTE1 )
        // output stop byte and disable interrupts
        _outp(hPort->PortBase+PAR_DATA_REG, buf[hPort->Location++]);
        _outp(hPort->PortBase+PAR_CON_REG, DISABLE_IRQ | STROBE_TRUE | PORT_OUT);
_outp( hPort->PortBase+PAR_CON_REG, DISABLE_IRQ | STROBE_FALSE | PORT_OUT);

} else if ( buf[hPort->Location] == CONTINUE_BYTE )
  {// additional data is needed from app so disable interrupts and post message to app
    _outp( hPort->PortBase+PAR_CON_REG, DISABLE_IRQ | STROBE_FALSE | PORT_OUT);
    PostMessage(hPort->hwnd, NEED_FILE_DATA_MSG, 0, 0L);
  };

_outp( MASTER_PIC_CTRL, EOI );

void interrupt FAR ReceiveISR( void )
{
  LPBYTE buf;
  PORTCONTEXT *hPort;
  unsigned char inChar = 'a';

  hPort = &Port1;
  buf = hPort->RBuffer;

  inChar = _inp( hPort->PortBase+PAR_DATA_REG ); // read LPT1 into inChar

  if ( inChar == STOP_BYTE )
    {// if stop byte read in, post message to app indicating done receiving
      PostMessage(hPort->hwnd, FILE_RECEIVED_MSG, 0, 0L);
    }
  else if (inChar != START_BYTE)
    {// store inChar in buffer shared with app
      buf[hPort->RLocation++] = inChar;
      hPort->RNumBytes++;
    }

_outp( MASTER_PIC_CTRL, EOI );
}
BIBLIOGRAPHY


