Interfacing homebrew hardware with the parallel port on a Windows PC

If your personal computer or laptop does not have a parallel port, then this document will not help you. If your PC has a parallel port, then you will be surprised at how simple it is to interface it with your own hardware circuit. The technique described here assumes that you have the following:

- a PC with a parallel port,
- a Windows operating system with Visual Basic 2010 Express (free download),
- experience with, or a desire to learn about, Microchip's microcontrollers and
- a hardware circuit you want to interface with your PC.

Associated documents

The associated documents are:

- PC-PCB_Interface-PICProgram.pdf – listing of assembly source code for the microcontroller,
- PC-PCB_Interface-HostProgram.pdf – listing of Visual Basic source code for the personal computer and
- inpout32_source_and_bins.zip – library containing Out() and Inp() subroutines (download only).

Overview of the three control registers

Any personal computer made within the last 20 years will have an ECP parallel port, if it has a parallel port at all. A huge fraction of the information available on the internet deals with versions of parallel ports which are obsolete. What is manufactured nowadays is an "extended capabilities" port which can be configured to emulate most of the earlier versions. Many of the details which used to distinguish one manufacturer's implementation of the parallel port from another's have long since been standardized.

You can see what your computer says about its parallel port by clicking down through My Computer > Control Panel > System > Hardware > Device Manager > Ports (COM and LPT) > Printer Port (LPT1). Your parallel port will invariably by labeled LPT1, reflecting the historical fact that the only output device in the early days was a line printer. Click on the Resources tab and you will see that the parallel port has the I/O Range of 0378-037A. Do not worry about the Drivers – these are what the Windows operating system uses to access the port. We will not access the parallel port using these drivers.

The numbers 0378 through 037A are absolute addresses in your computer's very low RAM memory. When your computers boots up, it reserves memory for certain precious bits of information that the operating system may need later on. These addresses are expressed in hexadecimal format, where each digit is a four-bit nibble. 0378 in hexadecimal format is equivalent to 0000 0011 0111 1000 in binary format and 888 in decimal format. 037A in hexadecimal format is equivalent to 0000 0011 0111 1010 in binary format and 890 in decimal format.

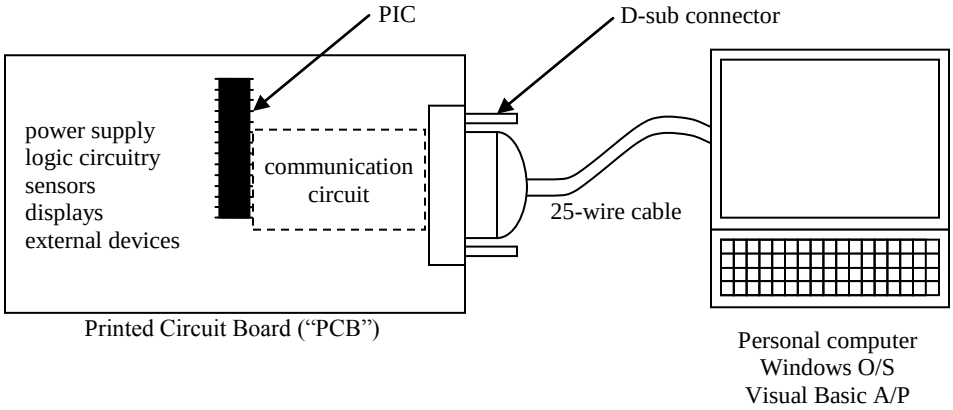
The number system, or base, in which a number is expressed is usually shown explicitly. A hexadecimal number would be written 0x0378. A binary number would be written b'0000001101111000' and a decimal number would be written d'888'.

So, your computer has reserved three consecutive addresses in its RAM memory for use in managing the parallel port. Back in the days of the Commodore64 and Sinclair computers, each address in memory consisted of eight bits, called one "byte". The earliest IBM personal computers had 16 bits at each memory address. The latest personal computers have either 32 bits or 64 bits at each memory address. The number of bits your PC expects in a word does not matter when using these three special addresses. Only eight bits in these special addresses are ever used.

The three addresses have names. 0x0378 is called the Data Register. 0x0379 is called the Status Register and 0x037A is called the Control Register. For historical reasons, an address in RAM which is accessed frequently is called a "register". Since there are eight bits per register, there are a total of 24 bits available for communication with a printer (or with your hardware). The individual bits all have names, too, most of which reflect their historical use in sending data to a line printer. A few of the 24 bits correspond directly to one of the physical wires in the cable which connects the external device to the parallel port. Most of the bits are used when sending data out of the PC, some are used only for incoming data and a few are able to work both ways. Some of the bits are "active high", in the sense that a positive voltage in the wire (called a logic "1") corresponds to the function assigned to the bit; others are "active low". Some of the bits are associated with specific electrical values relating to the port, such as the impedance or default voltage of a specific wire. The internet has much to say about all of these bits. The hard part is knowing what to ignore.

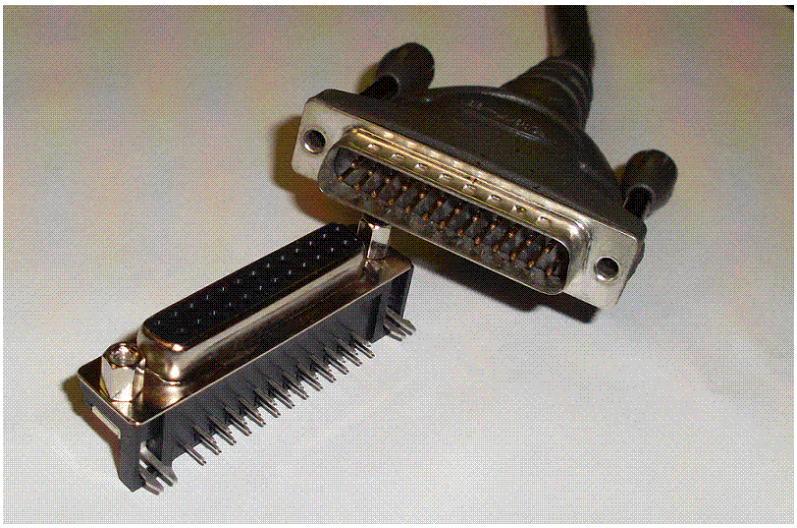
Overview of the application

The following figure gives an overview of the pieces of equipment which comprise the application.



You have or intend to build a circuit which does something. Your circuit will have a power supply and whatever other components you need for your application. You will likely construct your circuit on a printed circuit board (called a "PCB") or on a vectorboard of some kind. Along the way, you will need to add three components: (i) a microcontroller (shown above as the "PIC"), (ii) a D-sub connector / jack somewhere around the perimeter of the PCB and (iii) a few discrete components to wire the microcontroller to the D-sub connector (shown above as the "communication circuit"). If your circuit already includes a microcontroller, so much the better. It can be used to control communication with the PC, too.

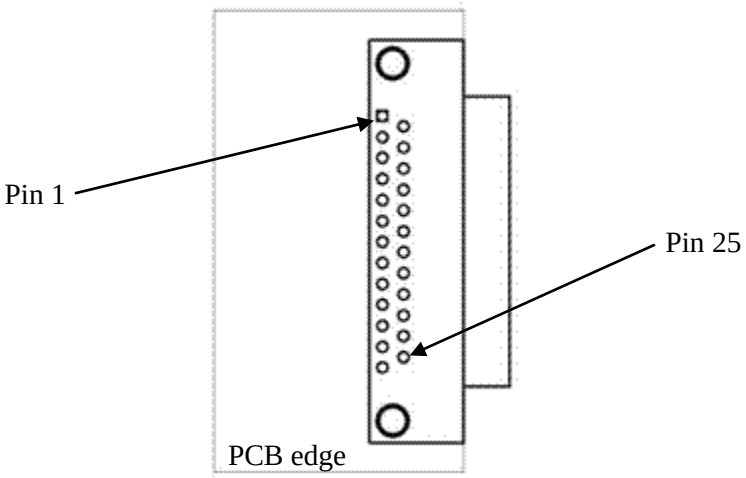
The PCB and the personal computer are connected by a 25-wire cable. The cable plugs into the D-sub connector mounted on the perimeter of the printed circuit board and into a similar connector on the back of the PC. My preference is to use a cable which has a male D-sub plug on both ends. Since virtually all parallel ports on personal computers have a female D-sub jack, this cable will be compatible with virtually all personal computers. It is useful if the fittings on the other end are the same. The following photograph shows a typical D-sub connector jack and plug.



The cable with plug is shown on the right. The 25 pins are visible within the metal shell. The jack is shown on the left. This particular pair of connectors has knobbed bolts on the plug. After the plug is inserted into jack, the bolts are turned tight.

The jack shown in the photograph is a "right-angle" PCB-pin connector. It is intended to be soldered directly onto the printed circuit board, oriented in a manner which keeps the plug in the same plane as the PCB.

In order to ensure that there is no confusion when laying out the PCB, the following diagram shows the holes which need to be drilled in the printed circuit board. The holes for pins 1 and 25 are identified.

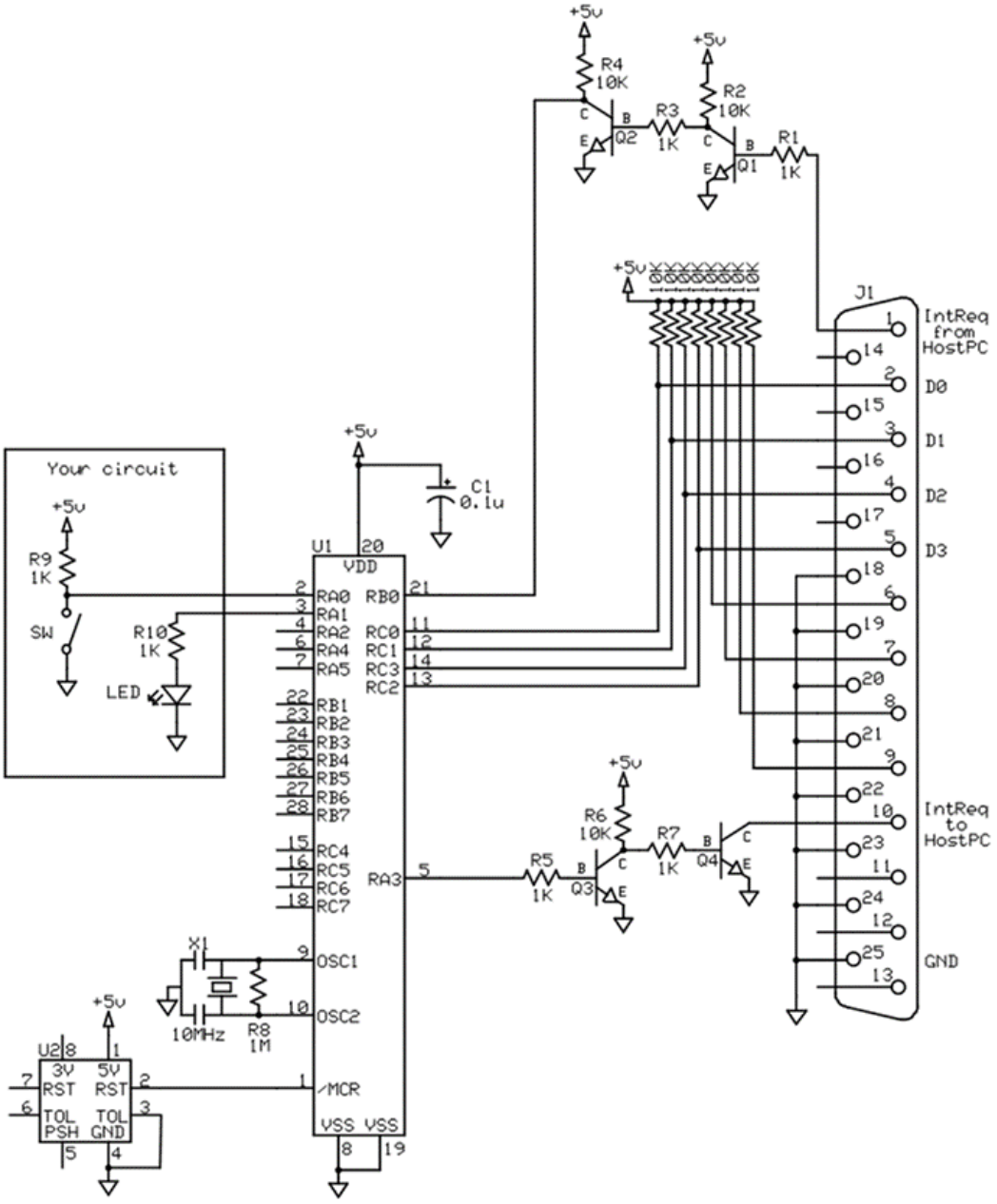


In the figure, the D-sub connector is shown mounted on the right-hand edge of the PCB. The view is from above the PCB, looking down through the holes, with the copper side down. As an aside, note that the hole-spacing on a D-sub jack is not the standard 0.1-inch.

From the point-of-view of software, two programs are required. The microprocessor will be running a program which, among other possible tasks, will govern the PIC's side of communications with the personal computer. The personal computer, called the "Host PC", will be running a separate program which, among other possible tasks, will govern the PC's side of communications with the PIC.

The communication circuit

Let us look next at the hardware required for the communication circuit. This circuit is based on a data bus which is four bits wide, meaning that four bits of data are exchanged at a time, and uses two control lines. The schematic diagram is shown below.



PC-to-PCB Interface

Digikey part numbers

U1=PIC16F872-I/SP-ND 2Kx14 20MHz 22I/O EE64x8 RAM128x8
U2=DS1834+1ND 8-DIP econo-reset
X1=490-1213-ND 10MHz ceramic resonator with built-in caps
Q1-Q4=P2N2222AGOS-ND general purpose NPN
J1=A35103-ND D-sub 25-pin female right-angle connector
C1=478-1831-ND 0.1uF 35V tantalum
Resistors all 1/4W 1% (Digikey part number P****CACT-ND)

The important features to note first are:

1. The four data lines are labeled D0, D1, D2 and D3, with D0 being the low-order bit and D4 being the high-order bit. For example, the hexadecimal digit 0xC, which corresponds to decimal d'12', is written b'1100' in binary and would be sent along the cable with D3=1, D2=1, D1=0 and D0=0.
2. The four data lines are wires #2 through #5 of the cable.
3. The control line which the Host PC uses to interrupt the PIC is wire #1 of the cable.
4. The control line which the PIC uses to interrupt the Host PC is wire #10 of the cable.
5. Wires #18 through #25 are grounded at both ends, at the PCB and at the Host PC.

.

The microcontroller requires a +5V power supply, with capacitor C1 providing a bypass for high-frequency noise. The personal computer will power the parallel port (at its end) with about +5V as well, but the voltage could be anywhere in the range from 3V to 6V. Transistors Q1 through Q4 have been introduced to ensure that the interrupt signals are delivered to each computer (the PC and the PIC) at the appropriate operating voltage.

Any voltage on wire #1 greater than approximately one volt will saturate transistor Q1 and, therefore, cut off transistor Q2. This will permit the voltage at Q2's collector to float up to the +5V supply voltage. On the other hand, any voltage on wire #1 below about 700mV will cut off transistor Q1 and, therefore, permit the base of transistor Q2 to float up to the +5V supply voltage. This will forward-bias Q2 into saturation and force its collector to ground.

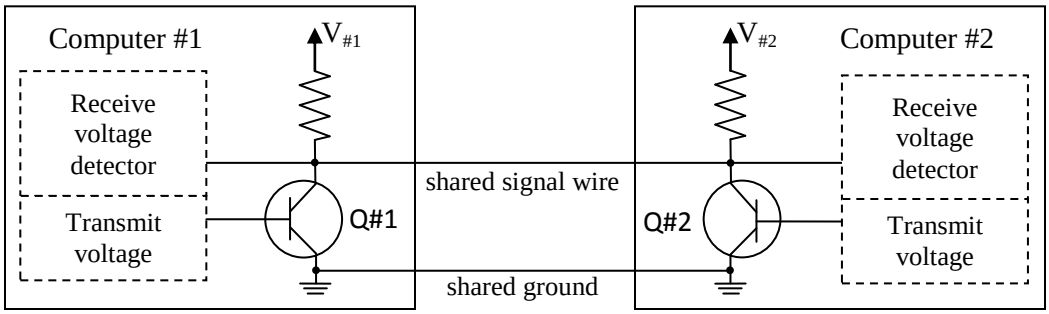
A similar process applies to transistors Q3 and Q4, which deliver the PIC's interrupt signal to wire #10. Note that the collector of Q4 does not have a pull-up resistor. It is said, and I have found it to be true, that pin 10 in all modern implementations of the parallel port are constructed with a pull-up resistor. The PC's pull-up resistor will allow the collector of Q4 to float up to the PC's operating voltage, which the PC will then recognize as an interrupt request.

The four data lines have not been provided with this type of protection against voltage differences. I have never experienced any problems allowing a PIC and PC to exchange data at TTL levels. However, the four data lines have been given pull-up resistors to ensure that a CMOS input is never left unconnected. It is said, but I have never tested it, that pins 2 though 9 (eight wires) in all modern implementations of the parallel port have pull-up resistors available to them. I have tried to choose my words carefully. As I say, I believe that the pull-up resistors exist. However, I believe that there are advanced settings for the parallel port which can be set in software and which will disconnect the pull-up resistors from the wires. Rather than risk that this could be the default setting of your PC on boot-up, or that the disconnection might be executed inadvertently, I have explicitly added pull-up resistors.

You may ask: why are there eight pull-up resistors, including four on wires #6 through #9, which this circuit does not use? The answer is this. The parallel port was designed as a data bus eight bits wide. For reasons that I will explain below, I have chosen to use only four of the lines. The PC reads and writes all eight lines at the same time and they should be wired to reflect that.

You may also ask: why are pull-up resistors needed at all? The answer is this. Data is transmitted through the data bus in both directions. When data is being transmitted, the transmitting computer has control of the wires and imposes on them the voltages it wants. The receiving computer needs to read these voltages without affecting them. Problems would arise if the two computers tried to impose different voltages on the same wire at the same time. The difficulties using the same line to transmit at

some times and receive at others are resolved by using so-called "open collector" or "open drain" connections. The principle behind this is shown in the following figure.



The final stage of each computer's transmission circuit is a transistor. I have labeled them Q#1 and Q#2, respectively. Although I have referred to the transistors as having "open collectors", I have shown pull-up resistors. If the transmit voltages of both computers are low, then both transistors will be cut off and the voltages on their collectors will not be determined by the transistors, but by the wiring outside of them. In the circuit shown, the voltage on the shared signal wire will float up to the midpoint of the two pull-up resistors. It will be somewhere between $V_{\#1}$ and $V_{\#2}$. (It is assumed that the input impedances of the receive voltage detectors is very high and does not affect the voltage on the shared signal wire.)

When the shared signal wire is in this floating condition, either computer (for example, Computer #1) can force it down, to zero, by raising the transmit voltage on the base of its output transistor Q#1. Current will flow downwards through both pull-up resistors and Q#1 must be able to carry, or "sink", that combined current. If Computer #2's receive voltage detector reads the shared signal wire, it will observe a low voltage, or logic '0'.

In order for this relationship to work, the computer at the receiving end must disable, or cut off, its output transistor. When it does this, the computer at the transmitting end can leave the voltage on the shared signal wire high (which sends logic '1') or force it low (which sends logic '0').

It is not necessary that there be two pull-up resistors on the shared signal wire. One would suffice. But, there must be at least one. Otherwise, the voltage on the shared signal wire will not be fixed at all when the transistors at both ends are cut-off. If either receive voltage detector reads the shared signal wire when the voltage is not fixed, there will likely be problems. In many electrical circuits, particularly when CMOS components are used, a floating or unconnected input can cause all kinds of voltage transients.

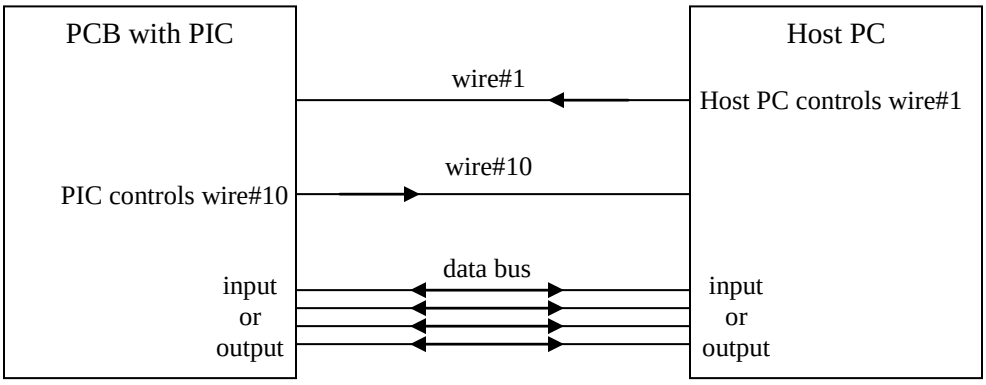
In the schematic, pull-up resistors are added for certainty's sake, even though they may be redundant if the PC already has them.

The LED and switch SW in the schematic are not part of the communication circuit. They are included to give the test programs something to do (described below). Your application will provide a circuit with a more meaningful function.

The "handshake" protocol

A procedure must be specified for both computers to use when they exchange data. The objective is to ensure that the receiving computer gets the data and, equally importantly, that the transmitting computer knows that the receiving computer got it. (Note that this is not the same as ensuring that the data received is identical to the data sent, which is a different problem.)

In our design, two control lines are used. Wire#1 in the cable, which leads to pin 21 of the PIC (labeled RB0) is the line the Host PC uses to send interrupts to the PIC. Wire#10 in the cable, which leads from pin 5 of the PIC (labeled RA3) is the line the PIC uses to send interrupt requests to the Host PC. Acknowledgements of interrupt requests are sent along the same wires, in the same directions. This arrangement can be depicted as follows:



In the steady state, between communication cycles: (i) the Host PC holds wire#1 low, (ii) the PIC holds wire#10 low and (iii) both computers place their connections to the wires in the data bus in a high-impedance state (the voltage is left floating), called "configured for input". In this state, either computer is able to initiate a communication, which it does by "setting high" (raising the voltage) on its outgoing control wire.

Obviously, the other computer, at the receiving end, must be wired or programmed to detect this low-to-high transition. Otherwise, the communication will not even get started. Generally speaking, there are two ways that the receiving computer can handle detection.

1. The more desirable way is for the receiving computer to be set up in such a manner that its on-going work is automatically interrupted when the low-to-high transition occurs, and processing diverted to whatever procedure handles incoming communications. Not surprisingly, this method is called "interrupt" processing. This is the faster method for handling communications, since the receiving computer responds almost immediately to the interrupt.

2. The less desirable way, from the point-of-view of timing, is for the receiving computer to "poll" the interrupt line. By polling, we mean that the computer's main program is written in such a way that it periodically checks the voltage on the interrupt line. If it is found to be high, then the main program starts to run the procedure which handles incoming communications.

For now, let us not worry about which way to use. Let us just assume that we have got the attention of the receiving computer.

The receiving computer acknowledges the interrupt by "setting high" its outgoing control wire. This is called acknowledging the interrupt request. The transmitting computer has been awaiting this acknowledgement. Now, both computers know what they are doing and both have readied themselves for the transfer of data.

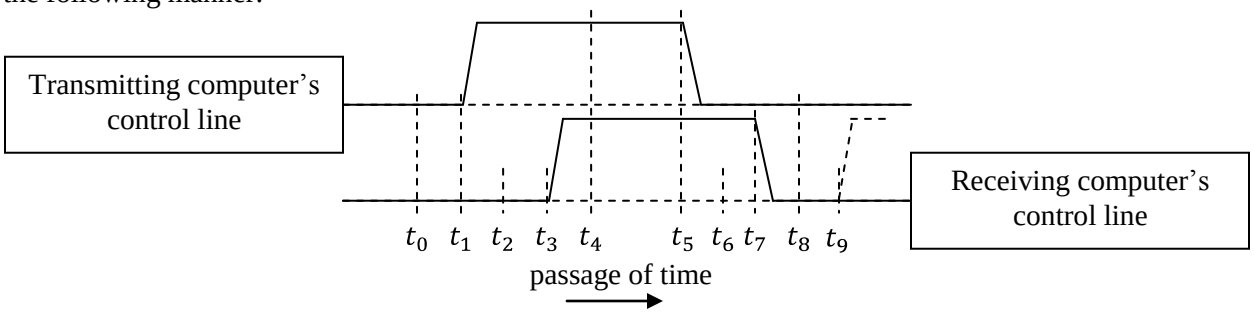
The transmitting computer then "places the data on the data bus" by setting the appropriate voltages on wires #2 through #5. It then tells the receiving computer to go ahead and read the data. It tells this to the receiving computer by "asserting low" (lowering the voltage) on its outgoing control wire. The receiving computer has been awaiting this high-to-low transition. When it sees it, it "reads the data on the data bus"

by making a note of the voltages. After it has done that, it tells the transmitting computer that it has finished reading. It tells this to the transmitting computer by asserting low its outgoing control wire. Since the receiving computer is done with the communication, it returns to the business which was interrupted when the cycle began.

The transmitting computer has been awaiting this high-to-low transition. When it sees it, it knows the data has been sent, and then goes about its other business.

At the end of the cycle, both control wires are back in their low voltage states, in which state either computer can begin a new communication cycle.

The voltage waveforms which characterize this kind of synchronized behavior are commonly shown in the following manner.



I have shown nine particular times during the cycle.

time t_0 – This is the moment when the transmitting computer decides to send a nibble to the receiving computer. The transmitting computer should not proceed unless the receiving computer's control line is low. If the receiving computer's control line is not low, then the receiving computer is probably not ready to receive a communication. There may be some other housekeeping items, too, such as starting a timer to detect a failure of the transmission.

time t_1 – This is the moment when the transmitting computer transmits the interrupt request, raising its control line high.

time t_2 – This is the moment the receiving computer recognizes that it is getting an interrupt request. Before going any further, the receiving computer may need a moment to wind up the affairs of whatever else it was doing and transfer execution to its interrupt service routine. There may be some other housekeeping items, too, such as starting a timer, ensuring that the data bus is configured for input (that is, to receive voltages), and so forth.

time t_3 – When it is ready, the receiving computer acknowledges the interrupt request by raising its own control line high.

time t_4 – This is the moment when the transmitting computer recognizes that it has received an acknowledgement from the receiving computer. Now, and not before, the transmitting computer can configure the data bus for output (that is, prepare to set the voltages on the wires). It can then actually set the voltages on the wires of the data bus. After that is done, it can proceed to the next step.

time t_5 – The transmitting computer asserts its control line low, to tell the receiving computer that the "data is on the data bus" and that the receiving computer can go ahead and read the voltages.

time t_6 – It will take a moment for the receiving computer to see the high-to-low transition, which it does at this time, t_6. The receiving computer then reads the voltages and stores the corresponding logic bits into memory.

time t_7 – The receiving computer tells the transmitting computer it has "read the data". It asserts its control line low and goes about its other business.

time t_8 – It will take a moment for the transmitting computer to see the high-to-low transition. When it does, it knows that the receiving computer got the data. Only now can the transmitting computer release control of the data bus, re-configure it for input and go about its other business.

time t_9 – This is some later time at which one computer or the other starts a new communication cycle.

This protocol looks pretty compelling. Every time either computer does something, it waits for the other computer to respond in a certain way. The two computers move ahead in step. Nothing should go wrong.

Even though they should not, things can go wrong. We need to look at some of the reasons why. The likelihood that things go wrong increases as the operating speeds (the "clock speed", say) of the two computers increase and as the difference between their operating speeds increases. It all boils down to the implicit assumptions you often make, as a designer, about how long it takes to complete specific tasks.

1. Loss of causality in one computer due to multi-threading

   For example, in the activities following time t_4, the transmitting computer configures its end of the data bus for output, then sets the voltages on the wires and then asserts its control line low. That is the intended sequence of events. In a low-level programming environment, such as programming the PIC, instructions are carried out strictly in the order they are listed. Furthermore, the assigned function of each instruction is completed before the end of the instruction cycle. In a higher-level programming environment, there is no such certainty. Principal activities may be carried out in subroutines and the subroutines may, in turn, require service by the operating system.

   Things can get out of order if the operating system is slow to carry out its duties or, just as dangerously, slower to carry out one duty than another. The application program can compound these problems if it works in various threads.

2. Loss of causality due to capacitance in the cable or circuits

   It may be necessary to wait a moment or two after changing any voltage. Although there are no capacitors in the communication circuit, at least according to the schematic diagram, there is capacitance in the cable and along the traces on the printed circuit board. The capacitance might not be much, but it will slow down the response to a step-change in voltage. For example, if a pin was connected to the power supply through a 10KΩ resistor and if its wire in the cable had a capacitance of 10pF, then the R-C constant of its voltage response would be $\tau = RC = 0.1\mu s$. It will take a few hundred nanoseconds for the voltage on the cable to stabilize after a step-change.

This is about the same as the instruction cycle time of the PIC. But it is a lot longer than the instruction cycle time of a modern PC.

The physical response time is obviously an issue when either computer sets the voltages onto the cable for transmission. Less obviously, it is also a potential problem when either computer configures the data bus for input or output. Re-configuration may cause the voltages on the wires to rise or fall and must be allowed to finish before voltages are actually set on the wires.

There are two ways to deal with this problem. The way used in the test programs is to build in a software delay after any change with affects the cable. The more sophisticated way, but beyond the scope of an amateur project, is to build in a feedback detector which can determine when the voltages on the cable have reached the desired values. For example, the PIC waits 100μs before starting any transmission.

.

3. Successive communication cycles

It is possible to start a communication cycle too quickly after a preceding cycle finishes. Look at times t_7, t_8 and t_9 in the timing diagram. At time t_7, the receiving computer is done with the first communication cycle. After asserting its output control line low, it goes back to its unfinished business. The transmitting computer labours on. At time t_8, it detects the high-to-low transition sent by the receiver. It then cleans up. It has to re-configure the data bus for input, stop its timer and who knows what else.

What if all that is not done by time t_9, when the receiving computer decides to start a new communication cycle? I described above that a computer which wants to transmit should check the other computer's control line before sending an interrupt request. That is not sufficient here. The transmitting computer asserted its output control line low away back at time t_5. The receiving computer will see that control line low, even although the transmitting computer may still be doing housekeeping to wrap up the first communication cycle.

There are several ways to deal with this problem, none of them perfect.

A. Introduce a third control line between the two computers specifically for them to use to tell each other that they are ready for a receive cycle to begin. We have seen that just making sure the other computer's control line is low is not sufficient. (Disadvantage - this solution requires more hardware.)

B. Since this problem arises when the receiving computer starts a transmission too soon after receiving a message, one could ensure that a transmission is never started too soon after a reception. However, this phrase "too soon" depends on the speed of the other computer, which may not be known. (Disadvantage - this solution requires adding delays in software which might not have been necessary.)

C. Both computers should be running timers throughout the communication cycle. If it takes too long for a transmission or reception to proceed to the next step, the computer which is waiting can assume that the computer at the other end has encountered a problem. I have called this event a "communication timeout". If such a timeout occurs, it is possible to look behind it, to discover a likely cause. For example, if one computer sends an interrupt request, but then waits and waits but never gets an answer, it might be best to assume that the other computer simply never got the interrupt request. Corrective

action could be to assert the control line low (a high-to-low transition which the other computer will not treat as an interrupt request) and then re-set it high. In effect, this repeats the interrupt request. (Disadvantage - this solution requires more software.)

The test programs described below use a combination of approaches B and C.

(i) The Host PC has a built-in delay – a human being. The Host PC sends data to the PIC only when the human operator tells it to. It is infrequent (but it does happen), that the user will click to send a nibble at just the instant when the Host PC is receiving data from the PIC.

(ii) The PIC's program, on the other hand, relies on the fact that the Host PC is much faster. Although the PIC responds to most commands by immediately sending a reply to the Host PC, it is likely (but not guaranteed) that the Host PC will be ready to receive before the PIC is ready to send.

(iii) To deal with the occasional communication failure, both programs detect communication timeouts and can recover from a bad cycle.

Enough said about this.

Overview of the Microchip 16F872 microprocessor

One of the handiest and convenient computers is the microcontroller. It is ideal for applications like this one. A microcontroller is a general-purpose computer with a limited set of instructions. A microcontroller will have internal RAM and, often, internal EEPROM (electronically-erasable programmable read-only memory) as well. A microcontroller will have physical I/O (input / output) pins which are directly accessible by external hardware. All of this will be enclosed in, say, a 28-pin DIP package. Many microchips have additional, very valuable, capabilities. These include on-chip analogue-to-digital conversion, on-chip pulse-width modulation, on-chip serial communication modules, on-chip timers, and so on. Overall, a microcontroller can be extremely useful to the homebrew enthusiast.

The granddaddy of companies in the microcontroller business is Microchip. Their website has extensive documentation and their development package, including simulators and debuggers as well as an assembly-language compiler, is free. A good, useful chip is their 16F872, which is widely-available.

The basic architecture of a microchip differs from the architecture of the processor in your PC. Your PC's processor, an Intel chip, for example, is based on the architecture first proposed by a mathematician named Von Neumann. A program, which consists of a set of instructions, and any data used by the program, are both stored in memory, but not necessarily sequentially. When the program is started, the first instruction is executed. Unless the instruction is some kind of a jump, the second instruction is then executed. If the instruction is a jump, then the next instruction to be executed is the one where the jump landed. Data which the program needs is also stored in memory. When the program needs to look at or change a byte of data, it references the data by the address at which it is stored in memory. The program's instructions and the data are mixed up together in memory. Any particular memory location may contain an instruction of the program (called "code") or it may be a piece of data which the program treats as a mathematical variable when executing the code. It is not unknown for executing code to find itself trying to execute data. Your personal computer probably "hangs up" because of this fairly often.

In a microcontroller, the instructions and data are saved in separate memories. Furthermore, the data is saved in a way and place where it is accessible more quickly than it otherwise would be. The data is saved in specific registers. The instruction set is optimized so that these registers can be accessed directly and quickly. The characteristics of this kind of architecture were first studied by computer scientists at Harvard University. It is called a Harvard architecture.

Neither architecture is "better" overall than the other. Each is better suited for some applications but not others.

The 16F872 chip has 22 pins (out of its 28 pins) which can be used for I/O. Each of these pins can be individually controlled by the program stored in the chip. Effective communication with the Host PC can be managed using only six of these pins – two for control and four for data. This leaves 18 pins for interacting with your circuit. In many cases, you might find it possible to have the 16F872 become the heart of your circuit and to have it manage communication with the Host PC as a side-task.

I prefer to use an external clock to run the 16F872. This is the ceramic resonator X1 shown in the schematic. A parallel 1MΩ resistor (R8) ensures that the resonator starts oscillating. Pin 1 of the 16F872, labeled MCCR, is the reset input. The 16F872 re-starts when the voltage on this pin is raised from zero to the supply voltage. This event should be delayed slightly from the application of power to the circuit as a whole. The EconoRest chip (U2) does this with precision and, as well, allows two separate voltages to be reset. It may be a bit of overkill for your circuit. In many circuits, wiring pin 1 to the positive voltage supply through a 10KΩ resistor and wiring it to ground through a 10μF capacitor will work as well. The time constant of this R-C pair is $\tau = RC = 100ms$. The 16F872 will start up about a quarter second after power is applied to the circuit.

That, plus the program, is all that is needed to use the PIC. (You may ask: what is a "PIC"? Actually, it is the trade name which Microchip uses to identify its microcontrollers. The 16F872 is officially a PIC16F872. But, like "fridge" or "xerox", what was at first a manufacturer's name or trade name came into common use as a noun or verb. So it is that "PIC" is sometimes used as a general substitute for "microcontroller".)

The LED at pin 3 is used by the test program as a visual output. Switch SW at pin 2 is used by the test program as a user-controlled input. Neither is needed to communicate with the Host PC. Both can be removed and replaced with your circuit.

The schematic shows that the PIC uses pin 5, which is labeled RA3, as the output line for its communication control wire. The label RA3 tells us that this pin is the fourth bit of the eight-bit data register which the PIC knows as portA. When the PIC's program runs the instruction "`bsf portA,3`", the fourth bit of portA is set. Setting this pin high sends an interrupt request to the Host PC. Note that the bits in the register are labeled starting with bit 0 being the least significant. There is no magic to using this particular pin. All of the I/O pins on the 16F872 can be configured for use as inputs or as outputs. Using pin 5 of portA was an arbitrary choice.

Using pin 21, labeled RB0, as the input line for interrupt requests from the Host PC is not so arbitrary. The PIC has built-in wiring which suits using this particular pin as a detector for interrupts. When the chip is set up this way, a low-to-high voltage transition on pin 1 will immediately transfer execution to the instruction at address 0x0004 in program memory. The procedure which handles incoming interrupts always begins at that address. By the way, the label "RB0" tells us that this pin is the least significant bit in the data register which the PIC knows as portB. Also by the way, a voltage transition on pin 1 is not the only way an external device or devices can interrupt the PIC, but it is eminently suited for a single-line interrupt.

The data bus in the test circuit is connected to the four pins 11, 12, 13 and 14, which are labeled RC0, RC1, RC2 and RC3, respectively. They form the low-order nibble of the data register which the PIC knows as portC. Although any four (or eight) pins could be used to connect to the data bus, there is some rationale to using these four. You will note that RC0 is connected to the low-order bit (D0) of the data

bus and that RC3 is connected to the high-order bit (D3) of the data bus. This means that, when the PIC reads a nibble from the data bus, it will be stored in portC, at the lower end. In fact, the nibble read will be the low-order nibble in the eight-bit register portC. Having the data read directly into the low-order nibble of a register means that it is not necessary for the program to make any adjustment, or to shift the bits, to put the bits into the correct right-hand position.

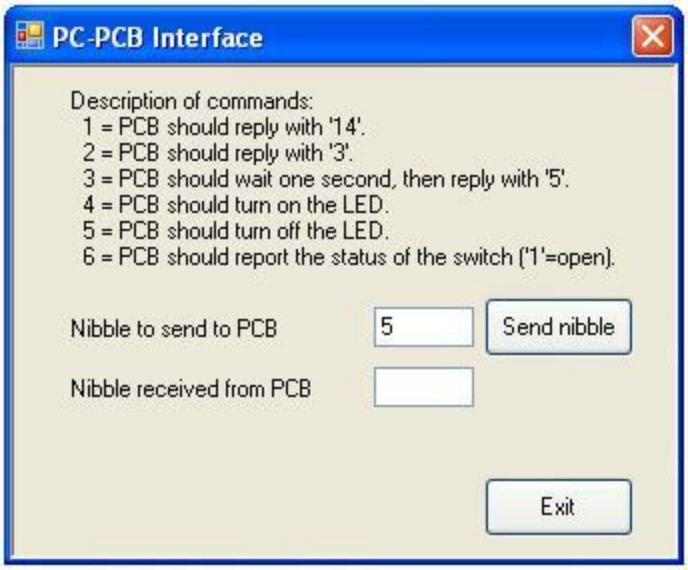
So, the communication circuit uses six of the PIC's 22 I/O pins. That leaves 16 I/O pins for use by your circuit. You will find out, though, that I/O pins are very much like space in a garage: no matter how many you have, you will want more.

It is for this reason that I prefer to use a four-bit data bus rather than the, arguably more traditional, eight-bit bus. Four lines allow for 16 unique combinations. If your communication consists mostly of sending commands from one computer to the other, then 16 unique commands may be sufficient. If not, it is simple to introduce sub-commands beneath the primary commands and to send a complete command as two successive nibbles.

On the other hand, if your application involves the exchange of a lot of data between the computers, then an eight-bit data bus would be better. As always, any arbitrary eight pins can be assigned to this task. On the other hand, the program will be simpler if the eight bits are sent and received in some kind of natural order, such as into the eight pins which comprise portC. In any event, the 25-wire D-sub cable and the parallel port were / are intended to handle eight bits of data and you should feel free to do so, too.

Overview of the test programs

The test programs are listed in accompanying documents. There are two of them – one for the PIC and another for your personal computer. The activity of the programs is illustrated in the following screen shot, which is the form that will appear on your screen.



The test program running on your PC permits you to send any nibble as a command to the PIC. The nibble is typed into the textbox just to the left of the "Send nibble" button. Nibbles consist of the integers 0 through 15. Type in the number, click the button and the nibble will be sent to the PIC.

What happens next is shown in the "Description of commands:". The PIC will respond to the six commands shown, which correspond to the nibbles 1 through 6. If you send a 0, or any nibble between 7

and 15, the PIC will respond by sending back a 12. A d'12' is the PIC's response to any command which it did not recognize. The nibble the PIC sends back, if any, will be displayed in the text box to the right of the label "Nibble received from PCB".

The test program for the PIC

The test program for the PIC is written in Microchip's assembly language. I will not regurgitate the many comments which are included in the code. But there are a couple of points worth noting.

1. The PIC has two sources of interrupt. Pin RB0 treats a low-to-high voltage transition as a request for an interrupt, and the processor responds to that waveform by causing execution to begin at the instruction at address 0x0004. The program uses the PIC's internal Timer0 module as a second source of interrupts.

2. The Timer0 module and the related data register called timer0 are used by the program to detect when a communication process has taken so long that it should be treated as having failed. The designer (me) decided that the appropriate length of time to wait was one second. If a transmission is not completed within one second, the program begins to execute a procedure named CommTimeoutTransmit. Similarly, if a reception is not completed within one second, the program begins to execute a procedure named CommTimeoutRecv.

3. The Timer0 module is configured so that register timer0 is automatically decremented (in the background) every 256 instruction cycles. With a 10MHz clock and a Timer0 pre-scalar set to 256:1, timer0 will be decremented every 256 * 400ns = 102.4μs. Register timer0 will count from 0 to 0 (256 counts), every 256 * 102.4μs = 26.2144ms. Every time register timer0 decrements to zero, the Tiimer0 module will request an interrupt and, like the RB0-type interrupt, processing will continue from address 0x0004. In the program, a Timer0-interrupt decrements another data register, named CommTO (short for "communication timeout"). So, as time passes, register CommTO will be decremented once every 26.2144ms. CommTO will count out 38 steps in 38 * 26.2144ms = 996.1ms, which is close enough to one second for our purposes. Register CommTO is used by the program in the following way. When a transmission or reception begins, register CommTO is set to the value 38. From time-to-time during the handshake process, the program checks the value of CommTO. If CommTO reaches zero, a timeout has occurred.

4. Since an interrupt can occur for two different reasons, the first thing the Interrupt Service Routine (which is what one calls the procedure which runs in response to an interrupt request) must do is figure out whether it is an RB0-type interrupt or a Timer0-type interrupt.

5. Like most digital computers, the processor uses certain important registers to keep track of what it is doing. These include: (i) a general-purpose register called the "accumulator", (ii) a "status" register whose bits indicate conditions like zero and carry, (iii) a pair of registers with the address of the instruction currently being executed, called the "program counter", and so on. When an interrupt occurs, and execution unexpected begins at a new address in the code, these registers will change as the instructions in the Interrupt Service Routine are executed. When the interrupt is done, execution will resume where it left off. If the important registers now have different values than they had when the interrupt began, the program will fail. To prevent this, the Interrupt Service Routine must, as soon as it starts, save copies of any registers which might be changed during execution of the ISR.

The test program deals with three related matters: (i) a Timer0-type interrupt can occur up to 38 times during one RB0-type interrupt, so it may be necessary to make more than one copy of the registers needed by the interrupted routine, (ii) the ISR in the test program only changes the w and status registers, so other registers do not need to be saved (the program counter is saved automatically) and (iii) the act of saving copies of the registers can itself change the status register, so the two registers must be saved using instructions which do not themselves modify them.

6. There is not much of a main program. It toggles the LED on and off every three seconds to give the user an indication that it is working. And, every time the LED turns on, an incremented counter is sent to the Host PC for display. All of the interesting activity is carried out inside the Interrupt Service Routine itself.

7. Having the Interrupt Service Routine carry out all of the activity related to communications is sometimes a good idea and sometimes not. In our test program, doing this is acceptable because the transmission and the response are of particular interest.

   Compare this way of handling an RB0 interrupt to the opposite way. We could have defined another data register, named RB0IntHasOccured, for example. The only thing the Interrupt Service Routine would do after an RB0 interrupt would be to set this flag high. The main program would have to be changed so that it checked this flag ever so often, at a frequency set by the time of a loop. If the flag was determined to be high, the main program itself would complete the reception of the nibble, then decode the command and respond as expected. And, of course, it would have to un-set the flag RB0IntHasOccured so it could be used again during the next reception cycle.

   Alternatively, one could have the Interrupt Service Routine read and save the nibble sent by the Host PC, but not execute the command. The ISR would set a flag like RB0IntDataReady to alert the main program that it must do some follow-up activity based on the value in the saved nibble.

   How to split the work between the ISR and the main program needs to be decided on a case-by-case basis.

8. The 22 I/O pins on the 16F872 are combined into three I/O ports: portA (six bits only), portB and portC. The program defines three data registers, named portAmirror, portBmirror and portCmirror. Why? When the PIC writes a value to a pin, it does so it with a twist. It uses a so-called read-modify-write cycle. It first reads the port, then modifies the bit or bits according to the instruction and, finally, it writes the whole port. Problems can arise if a bit is changed shortly after another bit of the same port is changed. When the first bit is changed, it will be a moment or two before the voltages in the circuit stabilize at the new value. If there is any R-C element in the circuit the pin is driving, as there always is, the circuit voltage will not change instantaneously. If the program executes another bit change quickly, before the circuit has stabilized at the new value, the old unchanged value will be read. This possibility can be avoided if the program keeps an up-to-date copy (the mirror registers) of what it previously wrote to the port and re-writes all of the bits every time it makes a change.

Essentials of working with the Host PC

Do you remember your old "386" PC? It was powered by an Intel 80386 processor. At the time, it was the worthy successor to a fine series whose forerunners were the 80286 (if you were around early in the

PC era, you might even have had a "286" PC), the 80186, the 8086 or 8085 (if you had a Radio Shack computer or Commodore64), the 8080 (if you manufactured space shuttles) or Z8 (if you had a Sinclair), and the forerunner of them all, a four-bit processor called the 4040 (used in hand calculators).

Up until the 80286, the machine-language programmer had direct access to the three control registers we described above: the Control register, the Data register and the Status register. Do you remember Peek and Poke? It was possible to run the 80386 in a mode which emulated the 80286 and so provided such access with the 80386 as well.

The "486" and later "586" (where numbering stopped and Pentium-ing began) deny such access. The control registers lie in so-called "protected memory", and cannot be accessed directly by application programs. This is not to say that they cannot be accessed at all. Accessing them requires a "driver".

Using interrupts with the Pentium is also difficult. Even the 8080, which is now 40 years old, had "vectored" interrupts which allowed interrupt-processing to be directed by hardware to eight different addresses (unlike the one address at 0x0004 where interrupts begin on a PIC). Interrupt hardware has come a long way in 40 years. Depending on one's point-of-view, understanding how interrupts work on a modern PC is either a nightmare or a life-time commitment.

It is for this reason that the test program for the Host PC does not use true interrupts. Instead, it polls wire #10. It is true, in general, that the response time to an interrupt will be faster than the detection time of polling. But, sometimes, that does not matter. The PC-PIC interface is one of the times when it does not matter. Your PC can execute instructions a hundred times more quickly than a PIC. Your PC can probably run through a simple loop in the time it takes a PIC to execute just one instruction. Your PC will take much, much longer to refresh the screen, for example, than to detect that the PIC is requesting an interrupt.

If we accept the limitations of polling (but gain in simplicity), our task is reduced to figuring out what to do with the three parallel port registers. Here we go:

Set ControlRegister<5> = 1 to configure DataRegister for input and
assert ControlRegister<5> = 0 to configure DataRegister for output.

ControlRegister<0> is connected to wire #1, but is inverted, so that:
ControlRegister<0> going from 1 to 0 sends an interrupt request to the PIC and
ControlRegister<0> going from 0 to 1 sends a handshake acknowledgement, when required.

StatusRegister<6> is connected to wire #10, so that:
StatusRegister<6> going from 0 to 1 should be treated by the Host PC as an interrupt request and
StatusRegister<6> going from 1 to 0 should be treated by the Host PC as an acknowledgement.

DataRegister<3-0> is the four-bit data bus.

In Visual Basic, one can use Int32-type variables for the addresses and contents of the three registers. For example, the StatusRegister variables can be defined using:

```
Private StatusRegAddress as Int32
Private StatusRegContents as Int32
```

Actually, since the address of the status register is never changed by the program, we could recognize that it is a constant and set its value:

```
Private Const StatusRegAddress as Int32 = &H379
```

```
Private StatusRegContents as Int32
```

Note that Visual Basic uses the prefix &H for a hexadecimal number.

In order to select a specific bit from a register, one can use logical operations (ANDs and ORs). For example, the sixth bit from the right in the ControlRegister can be set to 0 by the following statement:

```
ControlRegContents = ControlRegContents AND &HDF
```

Note that &HDF = b'11011111', in which all bits are high except for the sixth bit from the right. All of the bits of the ControlRegister will keep their current values (whether they are currently 1 or 0), except for ControlRegContents<5>, which will be set to 0. Note that the right-most bit is ControlRegister<0>, the second bit from the right is ControlRegister<1> and the most significant bit in the byte is ControlRegister<7>.

To set the same bit to 1, one would use:

```
ControlRegContents = ControlRegContents OR &H20
```

Note that &H20 = b'00100000'. The OR operation in Visual Basic is an "inclusive" OR. All of the bits of the ControlRegister will keep their values, except for ControlRegister<5>, which will be set to 1 (whether it is currently 1 or 0).

To write a variable, such as ControlRegContents, to the ControlRegister, for example, one would call the subroutine Out(), using:

```
Out(ControlRegAddress, ControlRegContents)
```

To read the contents of the StatusRegister into a variable such as StatusRegContents, for example, one would call the subroutine Inp(), using:

```
StatusRegContents = Inp(StatusRegAddress)
```

Subroutines Out() and Inp() are not subroutines in the Visual Basic installation or anywhere else in the Windows universe. They are modern substitutes for the old POKE and PEEK functions. They were coded by some gifted amateurs who made them available on the internet. They deserve many kudos. To make these two subroutines available to the Visual Basic program running on your Host PC, follow the steps:

1. Download the file inpout32_source_and_bins.zip. For your convenience, I have included a copy of this file as an associated document. The "Download" button beside its name will start the usual download dialogue.
2. This file is a WinRAR Zip Archive file, with a .zip extension. The files inside this zipped file need to be extracted. You will see that the files it contains include the C++ source files, so you can if you wish delve more deeply into the details.
3. However, the only file you really need is the one named inpout32.dll in the directory \inpout32_source_and_bins\binaries\Dll. This file is a "dll" or dynamically-loaded library. Like other such libraries, it contains one or more subroutines which can be called from your application program. The question is: where to put the file? This file, inpout32.dll, should be copied and pasted (or cut and pasted) into your Windows directory. In particular, you need to copy it into the directory on your PC's C:\ drive named C:\WINDOWS\system32\. After you have copied it, it will appear in the list of files in that directory as just another Application Extension, among many others.

4. Lastly, you application program needs to be told that the two subroutines Out() and Inp() exist. In Visual Basic, that availability is indicated by defining these two functions as follows:

```
Private Declare Sub Out Lib "inpout32.dll" Alias "Out32" _
        (ByVal PortAddress As Int32, ByVal Value As Int32)

Private Declare Function Inp Lib "inpout32.dll" Alias "Inp32" _
        (ByVal PortAddress As Int32) As Int32
```

Oops! A fourth parallel port register

In the early PC's, there were only the three parallel port registers described above. Ports have become more sophisticated and, in particular, configurable. The advantage of configurable ports is that they can be configured as one wants. The disadvantage is that they have to be configured at all.

Our objective here is to configure your PC's parallel port so it operates like a Standard Parallel Port (SPP) which can both send and receive (called "bidirectional").

A modern parallel port has lots of registers in addition to the basic three. Only one of these is needed to force configuration as a parallel port. It is the Extended Capabilities Register and it is to be found at address 0x77A. It will always be the case that the Extended Capabilities Register is located &H400 higher in memory than the Control Register.

Changing the high-order three bits of the Extended Capabilities Register is the first step to configuring the parallel port. When these three bits are set to b'001', the port will be configured as a bidirectional SPP. Fortunately, that is the last step we need to take. So, to ensure that the parallel port is configured as a bidirectional Standard Parallel Port, all we need for code is:

```
Private Const ECRegAddress as Int32 = &H77A
Private ECRegContents as Int32
ECRegContents = Inp(ECRegAddress)
ECRegContents = ECRegContents AND &H1F
ECRegContents = ECRegContents OR &H20
Out(ECRegAddress, ECRegContents)
```

The first of the four instructions after the definitions simply reads the contents of the address and stores the contents in an integer variable. Then, the top three bits are zeroed out. Next, the fifth bit from the right is set high. Then, the contents are written back into the address. This procedure ensures that the other five bits of the contents of the Extended Capabilities Register are not changed.

Aside: On some computers, the characteristics of the parallel port can be set or changed in BIOS. If you interrupt the power-up on your computer, you can determine whether this is the case. It never hurts, however, to have the program set the characteristics exactly as it wants them.

There is one thing left to do to configure the parallel port as we want it. As described above, the Host PC's test program does not use interrupts. The default setting of the Standard Parallel Port is not to use interrupts. Just to be safe, however, it is a good idea to disable interrupts explicitly. This is done by asserting the Control Register's bit<4> low. This is easily done:

```
ControlRegContents = Inp(ControlRegAddress)
ControlRegContents = ControlRegContents AND &HEF
Out(ControlRegAddress, ControlRegContents)
```

Aside: If you want to modify the Host PC's program to use interrupts, the place to start is to set this bit high.

The test program for the Host PC

The test program for the Host PC is written in Visual Basic 2010 Express. I will not regurgitate the many comments which are included in the code. But there are a couple of points worth noting.

1. The instruction Application.DoEvents() appears from time-to-time throughout the code. Closer inspection will show that this instruction appears in all of the "infinite" loops. An "infinite" loop is one which continues until some condition is met, at which time an Exit Do takes matters outside the loop. What the Application.DoEvents() statement does is to pause execution of the loop for a moment or two, to give other event-handlers in the application a chance. Other event-handlers include such things as your having clicked on a button. Other event-handling subroutines in the application will not run if execution is confined to an infinite loop.

2. The program uses a background timer to detect timeouts during communication. If the background timer runs until expiry (set to two seconds, or 2000 milliseconds, in the program), a flag PCBCommError is set. The transmission and reception procedures keep their eye on this flag and take action if it becomes high.

3. For reasons described above, the PIC's program waits 100µs before starting a transmission. The Host PC's program should wait the same amount of time before it starts a transmission. Instead, it waits one whole millisecond. The reason is that the Threading.Thread.Sleep(1) function cannot accept an argument less than one, representing one millisecond.

Jim Hawley
December 2011

An e-mail pointing out errors and omissions would be appreciated.