This article was first published in the July 1997 issue of *Nuts & Volts* magazine. For more about Jan's books and articles, visit Lakeview Research on the web at *www.lvr.com*.

Quick & Easy Real-world Control for PCs

copyright 1997, 1999 by Jan Axelson

BASIC has long been a favorite programming language for real-world projects that access the world beyond a computer's keyboard, display, and the usual assortment of peripherals. If you have a project that involves flipping switches, reading sensors, spinning motors, or controlling or watching the outside world in a unique way, a BASIC program and an available port can often do the job.

If your program will run under Windows, you can create it with Microsoft's Visual Basic, whose ease of use and many abilities have made it one of the PC's most popular programming languages. But there's one hitch - unlike other BASICs, Visual Basic includes no way to read and write directly to ports. The solution is to use a DLL (dynamic linked library) that enables any Windows 3.x or Windows 95 program to access ports.

In this article, I'll show how to use Visual Basic to access the PC's standard parallel port as well as ports on custom I/O cards. I'll also present an example program to get you started.

What's a Port?

One of the main jobs of every PC is to move information about. The CPU (the brains inside the PC) understands two types of locations for data: memory and ports. The CPU uses different instructions and addressing to access each.

Memory includes the system RAM, which provides temporary storage for currently running programs, application files, and other information that the system may need quick access to.

The CPU uses ports to communicate with just about everything else, including standard components like drives, displays, modems, and printers, as well as custom and homebuilt devices. Each device uses one or more port registers, which are storage locations that both the CPU and the device can read and write to. Every PC has at least 1024 possible register addresses, from 0 to 3FFh, though many of these are reserved for standard components.

How to Access Ports

For many common devices, Windows has built-in routines that simplify communications. For example, an application may call Windows’ StartDoc function to send a file to a printer. The application doesn’t have to concern itself with knowing how to communicate with specific printers, because the operating system handles the details.

If you’ve built your own device that connects to a port, there are no built-in routines; you have to write your own. You can do so if your programming language includes the ability to read and write to ports. Other BASICs, including QuickBasic and QBasic for DOS, include Inp and Out keywords for this purpose. For example, this QBasic statement writes the value 55h to a port at 378h:

```
Out &h378, &hA5
```

(In this article, I use the conventional trailing h to indicate a value expressed as a hexadecimal number, while the BASIC code requires a leading &h.)

If you prefer decimal numbers, the statement looks like this:

```
Out 888, 165
```

This statement reads the value of a port at 379h into the variable ByteRead:

```
ByteRead = Inp(&h379)
```

Or in decimal:

```
ByteRead = Inp(889)
Introducing the DLL
```

Under Windows 95 and Windows 3.x, you can use a DLL to enable your Visual Basic programs to access ports just as in other BASICs. The DLL is a file that contains program routines for reading and writing to ports.

The DLL itself must be stored on the user’s system, and an application that uses the DLL must include a declaration for each routine it calls. The declarations tell the operating system where to find the routines.

When the application runs, the DLL loads into system memory and the application may call any of the declared routines. The Visual-Basic statements that call the DLL’s routines are identical to QBasic’s Inp and Out statements.

Where can you find a DLL for port I/O? I’ve made two DLLs available for free downloading on my web site at http://www.lvr.com. Inpout16.dll is for use with 16-bit programs, and Inpout32.dll is for use with 32-bit programs. Although the program code to call the Inp and Out routines is identical for both types, each requires a different DLL and declarations.

Which DLL to use depends on which version of Visual Basic you're using. A 32-bit program requires Windows 95, while a 16-bit program may run on Windows 3.x or Windows 95. Programs created with Visual Basic Version 3 (VB3) are 16-bit. The Professional edition of Version 4 (VB4) includes both 16-bit and 32-bit versions, while the Standard edition of Version 4 and all editions of Version 5 are 32-bit only.

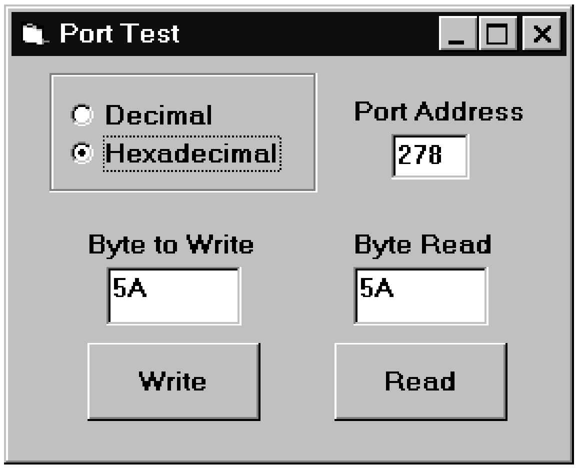
Using the DLL

The parallel printer port is one of the standard ports found on every PC, so it's a natural choice for testing and experimenting with the Inpout DLLs.

Although originally intended for printer communications, the parallel port has also become popular as an interface to many other devices, including homebuilt projects of all types. The basics of the parallel port are covered in many places, so this time around I'll include only the essential information for accessing a port with my example program. (See www.lvr.com for more on parallel-port access and interfacing.)

Most parallel ports are located at a base address of 378h, 278h, or 3BCh. In Windows 95, to find the base address of a parallel port, open the Control Panel, then click on System, Device Manager, Ports, select an LPT port, then click the Resources tab. The addresses of installed parallel ports are also displayed in the CMOS setup screens that you can access on bootup.

Figure 1 is the user screen for a program that tests the DLL's operation. Because some prefer hexadecimal numbers while others prefer decimal, the program allows a choice. To use the program, you enter the address of your port in the Port Address text box. To write a value to the port, enter the value in the Write text box and click the Write command button. To read the port, click the Read command button and the Read text box will show the value read.


Port Test

Decimal [✓ Hexadecimal]

Port Address: 278

Byte to Write: 5A

Byte Read: 5A

Figure 1. Use this Visual-Basic program to experiment with reading and writing to the standard parallel port and custom I/O ports.

A caution: this program allows you to attempt to write to any port address. Under Windows, many system-critical ports are protected from unauthorized access, but the chance remains that writing to a port can crash your system, or in rare cases even cause permanent damage. Writing to a standard parallel-port address is safe if nothing is connected to the port. Disconnect the cable to any device connected to a port before you experiment with the port. Reading of any port will do no harm.

At the end of this article, Listing 1 has the code for the example program, and Listing 2 has declarations for the DLLs.

The program was created with VB4, and will run under either the 16-bit or 32-bit edition of VB4, and in later versions. To run in VB3, you need to edit the declarations so they include only the lines between #Else and #Endif, and in the remaining two declarations delete the word Public and the underscore (line continuation) characters. In VB3, each declaration must be entered as a single line, with no carriage returns or line feeds.

The DLL (either inpout16.dll or inpout32.dll) must reside on any system that runs the program. Windows will search for the DLL in the following locations: the default Windows directory (usually \Windows), Windows' System directory (usually \Windows\System), and the project's working directory. (When you run the program in the Visual Basic environment, the working directory is your Visual-Basic directory.) Copy the appropriate DLL into any of these locations before you run the program. If you want to store the DLL in a different location, include its path in the Declarations.

Parallel Port Experiments

For a simple test of the example program, you can enter the base address of your parallel port, write a value, then read it back. The value shown should match logic levels of the Data bits (D0-D7) on the connector. Figure 2 shows the pin locations for each of the parallel port's signals.

All parallel ports use at least three port addresses. In addition to reading and writing to the port's base address, or Data port, you can read the Status port at base address + 1. (For example, with a base address of 378h, the Status port is at 379h.) Bits 3 through 7 of the Status register (S3-S7) are inputs. Bit 7 reads the inverse of the logic state at the connector. So if you read 78h, all five inputs are high at the connector. Bits 0-2 are usually unused and read as 0's. Writes to the Status port are ignored.

You can also read and write to the Control port, at base address + 2. (For a base address of 378h, the Control port is at 37Ah.) Bits 0 through 3 of the Control register (C0-C3) are outputs. Bits 0, 1, and 3 in the Control register are the inverse of the logic states at the connector. If you read 04h, all four bits are high at the connector. If you read 0Bh, all four bits are low. On some ports, the Control bits are open-collector-type outputs with pull-up resistors. These can be used to read external signals if you first write 4 to the Control port to pull the outputs high.

Two other bits on the Control port don't appear on the connector, but can affect port operation. On bidirectional ports, bit 5 determines the direction of the Data port. If you have a bidirectional port, be careful with this bit! The default is 0, which configures the Data bits as outputs. Setting the bit to 1 disables the Data outputs and allows you to use the port to read external signals. (In rare cases, bit 7 performs this function.)

Bit 4 determines whether or not hardware interrupt signals (from Status port bit 6) are passed on to the interrupt controller. The default is 0, disabled. Just setting the bit to 1 usually has no effect, however, because the interrupt must also be enabled at the system's interrupt controller. But it's best to keep this bit at 0, just to be safe, unless you intend to use the hardware interrupt.

In short, you can write values from 0 to Fh (15 in decimal) to the Control port, and the corresponding outputs will change, while the upper bits remain zeros.

To find out if you have a bidirectional Data port, write 20h to the Control port to bring bit 5 high. Then write a couple of values to the Data port and read each back. If the values don't match what you wrote, the Data outputs are disabled and you should be able to read external logic signals on the Data lines. If the values do match what you wrote, the Data outputs are still enabled and you can't use the port to read external signals.

No matter what changes you make to a port's registers, rebooting restores the original configuration.

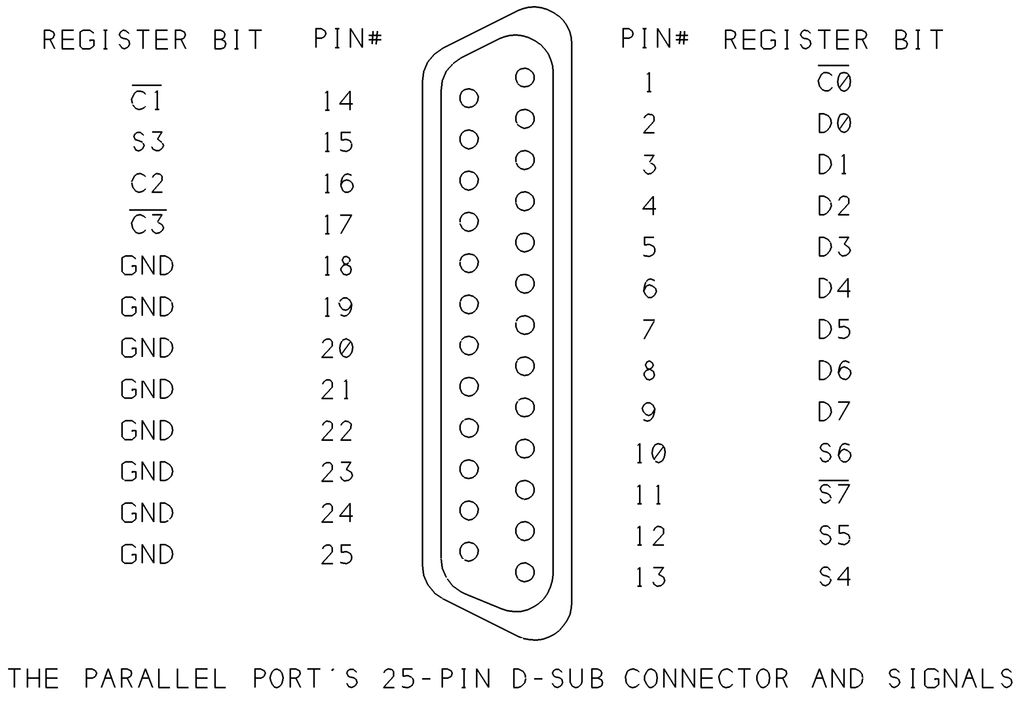


Figure 2. Locations of the 17 signals on the parallel port's D-sub connector, including Data bits D0-D7, Status bits S3-S7, and Control bits C0-C3. An overbar indicates a signal that is the complement, or inverse, of the corresponding bit in the PC's port register.

Accessing Other Ports

You can also use the Inpout DLLs to access ports on I/O cards that have custom ports. These are available from many vendors, in many configurations. Advantages are that you usually get more than the parallel port's 17 bits, and you don't have the hassle of dealing with the standard port's inverted bits. Some cards have analog inputs or outputs, with the converter circuits on-board, or features such as isolated outputs or relay-driver circuits. A card with built-in features like these can simplify your design work.

Custom I/O cards may use any unused port addresses in your system. Address ranges that are free for use in many systems include 250-277h, 280-2AFh, 300-377h, and 390-39Fh.

Developing Your Application

Once you have the DLL tested and working on your port, whether it's the standard port or a custom one, you're ready to design circuits to connect to the port, and the software to control them. The Sources box includes resources to get you started.

Jan Axelson is the author of *Parallel Port Complete: Programming, Interfacing, and Using the PC's Parallel Printer Port* (ISBN 0-9650819-1-5). You can contact Jan by email at jaxelson@lvr.com. To download the Inpout DLLs and source code for the example program in this article, visit *http://www.lvr.com* on the Web.

***

Another Way to Access Ports

Both Windows 3.x and Windows 95 allow software to read and write directly to ports. However, unlike DOS, Windows is a multi-tasking operating system, which means that a user may run multiple applications at once. If two programs try to access the same port at the same time, for different purposes, the result can be a mess!

For this reason, Windows makes it possible to manage accesses to a port from any application. A virtual device driver, or VxD, contains code that can read and write to a port. The VxD also can register the port with the operating system and specify whether or not it will share the port with other applications. If another program or driver attempts to access the port, the operating system will know whether to allow access or to inform the requesting software that access is blocked.

Sounds great, right? However, writing VxDs isn't for everyone. It requires an extensive knowledge of system hardware and Windows programming, plus expertise in assembly-language or C programming. Most device-driver writers use a variety of special tools, including Microsoft's Device Driver's Kit. There's no way to write a VxD in Visual Basic.

If your program communicates with a port that other applications have no reason to access, and if you're running Windows 3.x or Windows 95, direct port I/O is a simple and safe enough way to access the port. If you want to use a VxD to access a port, a quick solution is to buy one of the OCX's or other controls designed for this purpose. (See *www.lvr.com* for vendors.) There are also controls designed for use with Windows NT, which prohibits direct port accesses, and for using the parallel port's hardware interrupt.

Using an OCX in a program is straightforward. You install the OCX on your system, place the OCX on a form in your Visual-Basic program, and configure it with a range of port addresses and other optional information. You can then use Visual Basic statements to read and write to the ports. When the program runs, the port accesses are handled by the OCX, which communicates with a VxD (Windows 95) or a kernel-mode driver (Windows NT).

***

For Delphi Fans

If Borland/Inprise's Delphi is your programming language of choice, you can access ports without using a DLL. In Delphi 1.0, which creates 16-bit programs, use `Port` to read and write to ports.

Delphi 2.0 and higher, for creating 32-bit programs, have no port functions built-in, but you can access ports using in-line assembly code in your programs.

This code writes the value 55h to a port at 378h:

```
var
    ByteToWrite:byte;
    PortAddress:word;
begin
    PortAddress:=$378;
    ByteToWrite:=$55;
    asm
        push al
        push dx
        mov dx,PortAddress
        mov al,ByteToWrite
        out dx,al
        pop dx
        pop al
    end;
end;
```

This code reads the value of a port at 379h into the variable ByteRead:

```
var
    ByteRead:byte;
    PortAddress:word;
begin
    PortAddress:=$379;
    asm
        push al
        push dx
        mov dx, PortAddress
        in al,dx
        mov ByteRead, al
        pop dx
        pop al
    end;
end;
```

You can use this same technique with any programming language that supports in-line assembly code. Delphi programmers can also use the OCX's described in "Other Ways to Access Ports" in this article.

Sources

If you want to learn more about how to use the parallel port and custom I/O ports in your own projects, visit Parallel Port Central (http://www.lvr.com) for tutorials, program code, and links to resources of all kinds.

```
Option Explicit
Dim PortAddress%
Dim Hexadecimal%
Dim ByteToWrite%
Dim ByteRead%
Dim ValueToWrite$
Dim ValueRead$
Dim PortAddressAsText$

Private Function fncConvertValueToText$(ValueToCon-
vert%)
‘Converts an integer to a string
‘that displays the integer’s hex or decimal value.
If Hexadecimal Then
    fncConvertValueToText = Hex(ValueToConvert)
Else
    fncConvertValueToText = Str(ValueToConvert)
End If
End Function

Private Function fncDecimalToHex$(ValueToConvert$)
‘Converts a string’s decimal value to hexadecimal.
fncDecimalToHex = Hex(Val(ValueToConvert))
End Function

Private Function fncGetValueOfString%(StringToConvert$)
‘Returns the hex or decimal value of a string.
If Hexadecimal Then
    fncGetValueOfString = (Val(“&h” & StringToConvert))
Else
    fncGetValueOfString = Val(StringToConvert)
End If
End Function
```

Listing 1. Source code for Figure 1’s program. Page 1 of 3.

```
Private Function fncHexToDecimal$(ValueToConvert$)
‘Converts a string’s hexadecimal value to decimal.
fncHexToDecimal = Str(Val(“&h” & ValueToConvert))
End Function

Private Sub cmdReadPort_Click()
‘Read the port and display the value read
‘in the Read text box.
ByteRead = Inp(PortAddress)
txtReadPort.Text = fncConvertValueToText(ByteRead)
End Sub

Private Sub cmdWriteToPort_Click()
‘Get the value in the Write text box and write it
‘to the port.
ByteToWrite = fncGetValueOfString(ValueToWrite)
Out PortAddress, ByteToWrite
End Sub

Private Sub Form_Load()
‘Initial settings.
optNumberBase(1).Value = True
txtPortAddress.Text = 378
End Sub
```

Listing 1. Source code for Figure 1’s program. Page 2 of 3.

```
Private Sub optNumberBase_Click(Index As Integer)
‘The user may read and display values as decimal
‘or hexadecimal numbers.
‘When the number-base selection changes,
‘change the display to match.
If optNumberBase(1) = True Then
    Hexadecimal = True
    txtPortAddress.Text = _
       fncDecimalToHex(PortAddressAsText)
    txtWriteToPort.Text = fncDecimalToHex(ValueToWrite)
    txtReadPort.Text = fncDecimalToHex(ValueRead)
Else
    Hexadecimal = False
    txtPortAddress.Text = _
      fncHexToDecimal(PortAddressAsText)
    txtWriteToPort.Text = fncHexToDecimal(ValueToWrite)
    txtReadPort.Text = fncHexToDecimal(ValueRead)
End If
End Sub

Private Sub txtPortAddress_Change()
‘Get the value of the port address in the text box.
PortAddressAsText = txtPortAddress.Text
PortAddress = fncGetValueOfString(PortAddressAsText)
End Sub

Private Sub txtReadPort_Change()
‘Store the contents of the text box in ValueRead.
ValueRead = txtReadPort.Text
End Sub

Private Sub txtWriteToPort_Change()
‘Store the contents of the text box in ValueToWrite.
ValueToWrite = txtWriteToPort.Text
End Sub
```

Listing 1. Source code for Figure 1’s program. Page 3 of 3.

```
‘Declarations for Inp and Out routines for port I/O
‘There are two sets of declarations,
‘one for 32-bit programs and the other
‘for 16-bit programs.

‘The appropriate DLL (inpout32.dll or inpout16.dll)
‘must be stored in one of the following directories
‘on the user’s system: \Windows, \Windows\System,
‘or the current working directory.

#If Win32 Then
  Public Declare Function Inp Lib “inpout32.dll” _
    Alias “Inp32” _
    (ByVal PortAddress As Integer) As Integer
  Public Declare Sub Out Lib “inpout32.dll” _
    Alias “Out32” _
    (ByVal PortAddress As Integer, _
    ByVal Value As Integer)

#Else
  Public Declare Function Inp Lib “inpout16.Dll” _
  Alias “Inp16” _
    (ByVal PortAddress As Integer) As Integer
  Public Declare Sub Out Lib “inpout16.Dll” _
    Alias “Out16” _
    (ByVal PortAddress As Integer, _
    ByVal Value As Integer)
#End If
```

Listing 2. Include these declarations in any Visual Basic project that uses an Inpout DLL.