4

Programming Tools

Many programs that access the parallel port do many of the same things, including reading and writing to the port registers and finding and testing ports on a system. Another common task is reading, setting, clearing, and toggling individual bits in a byte. This chapter introduces tools to perform these functions in any Visual-Basic program.

Routines for Port Access

Listing 4-1 is a set of subroutines and functions that simplify the tasks of reading and writing to the port registers and performing bit operations. You can add the file as a *.bas* module in your parallel-port programs (use *Add Module*) and call the routines as needed in your code.

The individual routines are very short. The reason to use them is convenience. For the port-write subroutines, you pass the base address of a port and a value to write to the port. The routines automatically calculate the register address from the base address and invert the appropriate bits, so the value passed matches the value that appears at the connector. You don't have to worry about calculating an address and inverting the bits every time you write to a port. For the port-read functions, you pass a base address and the function returns the value at the port connector. For the bit operations, you pass a variable and bit number, and the routine auto-

```
Function BitRead% (Variable%, BitNumber%)
'Returns the value (0 or 1) of the requested bit in a Variable.
Dim BitValue%
'the value of the requested bit
BitValue = 2 ^ BitNumber
BitRead = (Variable And BitValue) \ BitValue
End Function
```

```
Sub BitReset (Variable%, BitNumber%)
'Resets (clears) the requested bit in a Variable.
Dim BitValue, CurrentValue%
'the value of the requested bit
BitValue = 2 ^ BitNumber
Variable = Variable And (&HFFFF - BitValue)
End Sub
```

```
Sub BitSet (Variable%, BitNumber%)
'Sets the requested bit in a Variable.
Dim BitValue, CurrentValue%
'the value of the requested bit
BitValue = 2 ^ BitNumber
Variable = Variable Or BitValue
End Sub
```

```
Sub BitToggle (Variable%, BitNumber%)
'Toggles the requested bit in a Variable.
Dim BitValue, CurrentValue%
'the value of the requested bit
BitValue = 2 ^ BitNumber
'Is the current value 0 or 1?
CurrentValue = Variable And BitValue
Select Case CurrentValue
    Case 0
        'If current value = 0, set it
        Variable = Variable Or BitValue
    Case Else
        'If current value = 1, reset it
        Variable = Variable And (&HFFFF - BitValue)
End Select
End Sub
```

Listing 4-1: Routines for reading and writing to the parallel port registers and for reading, setting, clearing, and toggling individual bits in a byte. (Sheet 1 of 2)

```
Function ControlPortRead% (BaseAddress%)
'Reads a parallel port's Control port.
'Calculates the Control-port address from the port's
'base address, and inverts bits 0, 1, & 3 of the byte read.
'The Control-port hardware reinverts these bits,
'so the value read matches the value at the connector.
ControlPortRead = (Inp(BaseAddress + 2) Xor &HB)
End Function
```

```
Sub ControlPortWrite (BaseAddress%, ByteToWrite%)
'Writes a byte to a parallel port's Control port.
'Calculates the Control-port address from the port's
'base address, and inverts bits 0, 1, & 3.
'The Control-port hardware reinverts these bits,
'so Byte is written to the port connector.
Out BaseAddress + 2, ByteToWrite Xor &HB
End Sub
```

```
Function DataPortRead% (BaseAddress%)
'Reads a parallel port's Data port.
DataPortRead = Inp(BaseAddress)
End Function
```

```
Sub DataPortWrite (BaseAddress%, ByteToWrite%)
'Writes a byte to a parallel port's Data port.
Out BaseAddress, ByteToWrite
End Sub
```

```
Function StatusPortRead% (BaseAddress%)
'Reads a parallel port's Status port.
'Calculates the Status-port address from the port's
'base address, and inverts bit 7 of the byte read.
'The Status-port hardware reinverts these bits,
'so the value read matches the value at the connector.
StatusPortRead = (Inp(BaseAddress + 1) Xor &H80)
End Function
```

Listing 4-1: Routines for reading and writing to the parallel port registers and for reading, setting, clearing, and toggling individual bits in a byte. (Sheet 2 of 2)

matically sets, resets, toggles, or returns the value of the requested bit in the variable.

Most of the example programs in this book use these routines. The routines require the *Inpout* DLL described in Chapter 2. Because the routines are fundamental to accessing the parallel port, I'll explain them in detail.

Data Port Access

`DataPortWrite` and `DataPortRead` access a port's Data register (*D0-D7*), which controls the eight Data outputs (pins 2-9). In a printer interface, these lines hold the data to be printed. For other applications, you can use the Data lines for anything you want. If you have a bidirectional port, you can use the Data lines as inputs.

To control the states of pins 2-9 on the parallel connector, you write the desired byte to the Data register. The address of the Data register is the base address of the port. `DataPortWrite` has just one line of code, which calls `Out` to write the requested byte to the selected address. `DataPortRead` calls `Inp`. On an SPP or a bidirectional Data port configured as output, it returns the last value written to the port. On a bidirectional port configured as input, it returns the byte read on the Data lines at the connector.

Status Port Access

`StatusPortRead` reads a port's Status register (*S0-S7*). Bits 3-7 show the states of the five Status inputs at pins 15, 13, 12, 10, and 11. Bit 0 may be used as a time-out flag, but isn't routed to the connector, and bits 1 and 2 are usually unused.

The Status register is at *base address +1*, or 379h for a port at 378h. However, as Chapter 2 explained, the value that you read doesn't exactly match the logic states at the connector. Bits 3-6 read normally—the bits in the Status register match the logic states of their corresponding pins. But bit 7 is inverted between the pin and its register bit, so the logic state of bit 7 in the register is the complement of the logic state at its connector pin. To match the connector, you have to complement, or re-invert, bit 7.

Using Xor to Invert Bits

The Boolean Exclusive-Or (Xor) operator is an easy way to invert one or more bits in a byte, while leaving the other bits unchanged. This is the truth table for an Exclusive-OR operation:

A	B	A Xor B
0	0	0
0	1	1
1	0	1
1	1	0

The result is 1 only when the inputs consist of one 1 and one 0. Xoring a bit with 1 has the result of inverting, or complementing, the bit.

If the bit is 0:

```
0 Xor 1 = 1
```

and if the bit is 1:

```
1 Xor 1 = 0.
```

To invert selected bits in a byte, you first create a mask byte, where the bits to invert are 1s, and the bits to ignore are 0s. For example, to invert bit 7, the mask byte is 10000000 (binary) or 80h. If you Xor this byte with the byte read from the Status register, the result is the value at the connector. The zeros mask, or hide, the bits that you don't want to change. The StatusPortRead subroutine uses this technique to return the value at the connector.

Here's an example:

`10101XXX`	Status port, bits 3-7, at the connector. (X=don't care)
`00101XXX`	Result when you read the Status register. (Bit 7 is inverted.)
`10000000`	Mask byte to make bit 7 match the connector
`10101XXX`	The result of Xoring the previous two bytes (matches the byte at the connector)

StatusPortRead also automatically adds 1 to the base address passed to it. This way, the calling program doesn't have to remember the Status-port address. Because the Status port is read-only (except for the timeout bit in EPPs), there is no StatusPortWrite subroutine.

Control Port Access

`ControlPortRead` and `ControlPortWrite` access a port's Control register (*C0-C7*). Bits 0-3 show the states of the four Control lines at pins 1, 14, 16, and 17. On an SPP, the Control port is bidirectional and you can use the four lines as inputs or outputs, in any combination. The Control register's address is *base address + 2,* or 37Ah for a port with a base address of 378h.

Bits 4-7 aren't routed to the connector. When bit 4 = 1, interrupt requests pass from the parallel-port circuits to the interrupt controller. When bit 4 = 0, the interrupt controller doesn't see the interrupt requests.

If you don't want to use interrupts, bit 4 should remain low. However, in most cases just bringing bit 4 high has no effect because the interrupt isn't enabled at the interrupt controller or at the interrupt-enable jumper or configuration routine, if used. Chapter 10 has more on interrupt programming.

In ports with bidirectional Data lines, bit 5 (or rarely, bit 7) may configure the Data port as input (1) or output (0). Usually, you must enable bidirectional ability on the port before setting pin 5 will have an effect. But to be safe, you should take care not to change bit 5 in your programs unless you intend to change the direction of the Data port.

As on the Status port, the Control port has inverted bits. In fact, only bit 2 at the connector matches the logic state of its bit in the Control register. The circuits between the connector and the register invert bits 0, 1, and 3. In other words, if you write *1111* (Fh) to the lower four bits in the Control register, the bits at the connector will read *0100* (4h).

As with the Status port, you can make the bits match what you read or write by re-inverting the inverted bits. To make the value you write match the bits at the connector, Xor the value you want to write with 0Bh (00001011 binary). The Control-port routines use this technique so that the values passed to or read from the Control port match the logic states at the connector.

Keeping Bits Unchanged

In writing to the Control port, you can use logic operators to keep the upper bits from changing. (You can use the same technique anytime you want to change some bits in a byte, but keep others unchanged.)

These are the steps to changing selected bits:

1.	`XXXX1010`	Determine the bits to write. (X=don't change)
2.	`11001100`	Read the port's current value.
3.	`11111010`	Create a byte containing all 1s except the bits desired to be 0.
4.	`11001000`	AND the bytes in steps 2 and 3.
5.	`00001010`	Create a byte containing all 0s except the bits desired to be 1.
6.	`11001010`	OR the bytes in steps 4 and 5. Bits 0-3 now match the desired logic states from step 1 and bits 4-7 are unchanged from the original byte read in step 2.

Reading External Signals

To read an external input at a Control bit, you must first bring the corresponding output high. You can use the Control-port bits as inputs or outputs in any combination. Because of this, the ControlPortRead routine doesn't bring the bits high automatically; the application program is responsible for doing it. (To bring all four outputs high, call `ControlPortWrite` with `ByteToWrite=&h0F`.)

As with the outputs, the value read at the Control port has bits 0, 1, and 3 inverted from their logic states at the connector. To re-invert bits 0, 1, and 3 and return the value at the connector, `ControlPortRead` Xors the byte read with 0Bh.

Optimizing for Speed

These routines are designed for ease of use, rather than fast execution. These techniques will increase the speed of the routines:

Eliminate subroutine and function calls by placing the code directly in the routine that would otherwise make the calls. The routines are short, and easily copied.

Assign the Status and Control-port addresses to variables instead of calculating them from the base address each time. You then need to specify the appropriate address instead of using the base address. To use this technique, do the following:

Eliminate this line from StatusPortRead:

```
StatusPortAddress=BaseAddress+1
```

Eliminate this line from ControlPortWrite and ControlPortRead:

```
ControlPortAddress=BaseAddress+2
```

In your application:

Assign the Status and Control port's addresses to variables:

```
StatusPortAddress=BaseAddress+1
ControlPortAddress=BaseAddress+2
```

And use these calls:

```
StatusPortData = Inp(StatusPortAddress)
ControlPortWrite Value, ControlPortAddress
ControlPortData = Inp(ControlPortAddress)
```

Instead of re-inverting the inverted Status and Control bits each time you read or write to them, you can just take the inverted bits into account in the program. For example, if a 1 at Control bit 0 switches on a relay, have the software write 0 to the bit when it wants the relay to switch on. Keeping track of which bits are inverted can be difficult however! One way to keep the program readable is to assign the values to constants:

```
Const Relay3On% = 0
Const Relay3Off% = 1
```

Often, while you're developing an application, you don't have to be concerned about speed. When the code is working properly, you can do some or all of the above to speed it up.

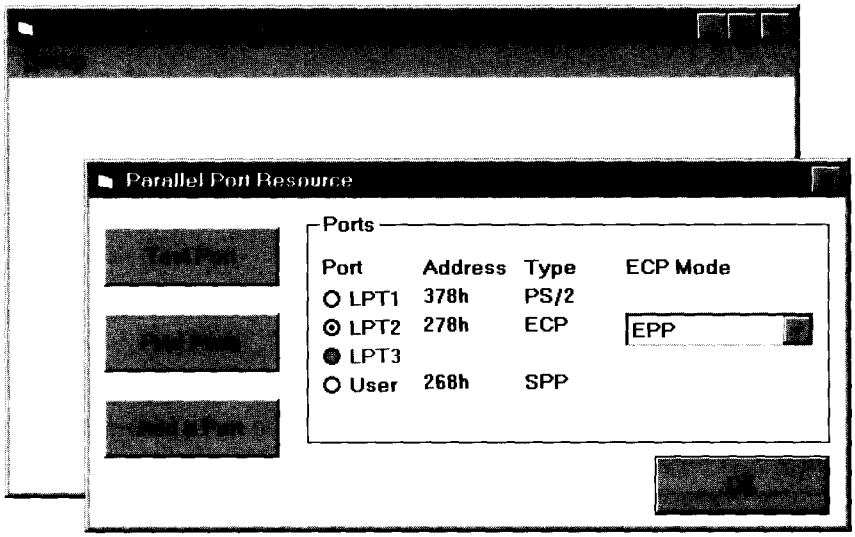


Figure 4-1: A form with a setup menu that enables uses to select and test ports.

Bit Operations

Sometimes you just want to set, reset, or toggle one bit in a byte, toggle a control signal, or set or read a switch. The `BitSet`, `BitReset`, `BitToggle`, and `BitRead` routines perform these operations, which you can use any time you want to read or write to a bit in an integer variable. Each routine is passed a variable and a bit number. The routine calculates the value of the selected bit and uses logic operators to perform the requested action on the individual bit.

For example, to set bit 4 in the variable PortData:

```
BitSet PortData, 4
```

and to read back this bit's value:

```
Bit4 = BitRead(PortData, 4)
```

A Form Template

Figure 4-1 shows a second tool for parallel-port programs: a set of Visual-Basic forms that you can use as a template, or starting point, for programs. The startup form is blank except for a Setup menu with a Port submenu, which displays a form that enables users to select a port, find the ports on a system, and test the ports. (You can add other items to the Setup menu.)

Most of the programs in this book use these elements as a base, with command buttons, text boxes, other controls and application-specific code added to the main form or in other modules.

Listing 4-2 contains the code for the form that displays the Ports. Listing 4-3 has the startup form's small amount of code. Most of the code is in a separate *.bas* module, Listing 4-4. In Visual Basic 3, procedures in a form module are local to the form, but all forms can access procedures in a *.bas* module. Version 4 is more flexible, with the ability to declare procedures `Public` or `Private`. Still, grouping the general routines in one module is useful for keeping the code organized.

The listings show the Visual Basic 4 version of the program. The Version-3 code differs in just a few areas, such as the calls for getting and saving initialization data. The companion disk includes both Version 3 and Version 4 code.

Saving Initialization Data

Each time the program runs, Listing 4-4's `GetIniData` subroutine retrieves information about the system's ports. When the program ends, `WriteInidata` stores the information to be retrieved the next time the program runs. This way, the program can remember what ports a system has, which port is selected, and any other information the program wants to store. Remembering these isn't essential, but it's a convenience that users will appreciate.

Ini Files

One way to access initialization data is to use Visual Basic's file I/O statements to read and write to a file. Under Windows, however, there are other options. Windows defines a standard method for storing data in *ini* files, which are text files normally found in the Windows directory. The best-known *ini* file is *win.ini,* which holds information used by Windows and may also contain data sections for individual applications. An application may also have its own *ini* file. This is the method used by Listing 4-4, which accesses a file called *Lptprogs.ini.* Listing 4-5 shows an example *ini* file. *Ini* files must follow a standard format consisting of one or more section names in square brackets `[lptdata]`, with each section name followed by data assignments.

Although you can use ordinary file I/O statements to read and write to an *ini* file, Windows provides API functions for this purpose. Calling an API function in a Visual-Basic program is much like calling other functions. As when calling a DLL, the program must declare the API function before it can call it. The listing includes the Declare statements for the API functions `GetPrivatePro-`

```
Private Sub cboEcpMode_Click(Index As Integer)
SetEcpMode (cboEcpMode(Index).ListIndex)
End Sub
```

```
Private Sub cmdAddPort_Click()
'Display a text box to enable user to add a port
'at a nonstandard address.
frmNewPortAddress.Show
End Sub
```

Listing 4-2: Code for Figure 4-1's form that enables users to find, test, and select ports. (Sheet 1 of 4)

```
Private Sub cmdFindPorts_Click()
'Test the port at each of the standard addresses,
'and at the non-standard address, if the user has entered one.
Dim Index%
Dim PortExists%
Dim Count%
Index = 0
'First, test address 3BCh
Port(Index).Address = &H3BC
PortExists = TestPort(Index)
'If the port exists, increment the index.
If Not (Port(Index).Address) = 0 Then
    Index = Index + 1
End If
'Test address 378h
Port(Index).Address = &H378
PortExists = TestPort(Index)
'If the port exists, increment the index.
If Not (Port(Index).Address) = 0 Then
    Index = Index + 1
End If
'Test address 278h
Port(Index).Address = &H278
PortExists = TestPort(Index)
'Disable option buttons of unused LPT ports
For Count = Index + 1 To 2
    optPortName(Count).Enabled = False
    Port(Count).Enabled = False
Next Count
If Not (Port(3).Address = 0) Then
    PortExists = TestPort(Index)
Else
    optPortName(3).Enabled = False
End If
End Sub
```

```
Private Sub cmdOK_Click()
frmSelectPort.Hide
End Sub
```

Listing 4-2: Code for Figure 4-1's form that enables users to find, test, and select ports. (Sheet 2 of 4)

```
Private Sub cmdTestPort_Click()
Dim PortExists%
Dim Index%
'Get the address of the selected port
Index = -1
Do
    Index = Index + 1
Loop Until optPortName(Index).Value = True
PortExists = TestPort(Index)
Select Case PortExists
    Case True
        MsgBox "Passed: Port " + Hex$(BaseAddress) + _
         "h is " + Port(Index).PortType + ".", 0
    Case False
        MsgBox "Failed port test. ", 0
End Select

End Sub
```

Listing 4-2: Code for Figure 4-1's form that enables users to find, test, and select ports. (Sheet 3 of 4)

```
Private Sub Form_Load()
Dim Index%
Left = (Screen.Width - Width) / 2
Top = (Screen.Height - Height) / 2

'Load the combo boxes with the ECP modes.
For Index = 0 To 3
    cboEcpMode(Index).AddItem "SPP (original)"
Next Index
For Index = 0 To 3
    cboEcpMode(Index).AddItem "bidirectional"
Next Index
For Index = 0 To 3
    cboEcpMode(Index).AddItem "Fast Centronics"
Next Index
For Index = 0 To 3
    cboEcpMode(Index).AddItem "ECP"
Next Index
For Index = 0 To 3
    cboEcpMode(Index).AddItem "EPP"
Next Index

'Enable the option buttons for existing ports.
For Index = 0 To 3
    optPortName(Index).Enabled = Port(Index).Enabled
Next Index
UpdateLabels
End Sub
```

```
Private Sub optPortName_Click(Index As Integer)
'Store the address and index of the selected port.
Dim Count%
BaseAddress = Port(Index).Address
IndexOfSelectedPort = Index
EcpDataPortAddress = BaseAddress + &H400
EcrAddress = BaseAddress + &H402
For Count = 0 To 3
    cboEcpMode(Count).Enabled = False
Next Count
cboEcpMode(Index).Enabled = True
End Sub
```

Listing 4-2: Code for Figure 4-1's form that enables users to find, test, and select ports. (Sheet 4 of 4)

```
Private Sub Form_Load()
StartUp
End Sub
```

```
Private Sub Form_Unload(Cancel%)
ShutDown
End
End Sub
```

```
Private Sub mnuPort_Click(Index%)
frmSelectPort.Show
End Sub
```

Listing 4-3: The startup form for the sample project is blank except for a menu. You can add whatever controls you need for a specific application.

`fileString` and `WritePrivateProfileString`. The API calls differ slightly under Windows 3.1 and Windows 95. The Version-4 code uses Visual Basic's conditional compile ability to decide which calls to declare. You can add these statements to any *.bas* module in a program. In Version 3, you use only the declares following `#Else`.

`GetIniData` uses `GetPrivateProfileString` to retrieve several values, including the address and type of each existing port, and a value that indicates the port that was selected the last time the program ran. `WriteIniData` uses `WritePrivateProfileString` to save these values when the program ends.

System Registry

Windows' System Registry offers another way to store program information. Visual Basic 4's `SaveSetting` and `GetSetting` are a simple way to store and retrieve information related to Visual Basic programs, and you can use these in a similar way to save port information.

Under Windows 95, two API functions enable programs to find and add system ports. `EnumPorts` returns the LPT number and a brief description of each parallel port that Windows is aware of, and `AddPort` displays a dialog box that enables users to add a port to the list.

Finding, Selecting, and Testing Ports

Because the parallel-port's address can vary, programs must have a way of selecting a port to use. There are several ways to accomplish this.

```
Private Sub Form_Load()
StartUp
End Sub
```

```
Private Sub Form_Unload(Cancel%)
ShutDown
End
End Sub
```

```
Private Sub mnuPort_Click(Index%)
frmSelectPort.Show
End Sub
```

Listing 4-3: The startup form for the sample project is blank except for a menu. You can add whatever controls you need for a specific application.

`fileString` and `WritePrivateProfileString`. The API calls differ slightly under Windows 3.1 and Windows 95. The Version-4 code uses Visual Basic's conditional compile ability to decide which calls to declare. You can add these statements to any *.bas* module in a program. In Version 3, you use only the declares following `#Else`.

`GetIniData` uses `GetPrivateProfileString` to retrieve several values, including the address and type of each existing port, and a value that indicates the port that was selected the last time the program ran. `WriteIniData` uses `WritePrivateProfileString` to save these values when the program ends.

System Registry

Windows' System Registry offers another way to store program information. Visual Basic 4's `SaveSetting` and `GetSetting` are a simple way to store and retrieve information related to Visual Basic programs, and you can use these in a similar way to save port information.

Under Windows 95, two API functions enable programs to find and add system ports. `EnumPorts` returns the LPT number and a brief description of each parallel port that Windows is aware of, and `AddPort` displays a dialog box that enables users to add a port to the list.

Finding, Selecting, and Testing Ports

Because the parallel-port's address can vary, programs must have a way of selecting a port to use. There are several ways to accomplish this.

```
#If Win32 Then
Declare Function GetPrivateProfileStringByKeyName& Lib _
"Kernel32" Alias "GetPrivateProfileStringA" _
(ByVal lpApplicationName$, ByVal lpszKey$, ByVal lpszDefault$, _
ByVal lpszReturnBuffer$, ByVal cchReturnBuffer&, ByVal lpszFile$)

Declare Function WritePrivateProfileString& Lib _
"Kernel32" Alias "WritePrivateProfileStringA" _
(ByVal lpApplicationName$, ByVal lpKeyName$, ByVal lpString$, _
ByVal lpFileName$)

Declare Function GetWindowsDirectory& Lib "Kernel32" _
Alias "GetWindowsDirectoryA" (ByVal lpBuffer$, ByVal nSize%)

#Else

Declare Function GetPrivateProfileStringByKeyName% Lib "Kernel" _
Alias "GetPrivateProfileString" _
(ByVal lpApplicationName$, ByVal lpKeyName$, ByVal lpDefault$, _
ByVal lpReturnedString$, ByVal nSize%, ByVal lpFileName$)

Declare Function WritePrivateProfileString% Lib "Kernel" _
(ByVal lpApplicationName$, ByVal lpKeyName$, _
ByVal lpString$, ByVal lpFileName$)

Declare Function GetWindowsDirectory% Lib "Kernel" _
(ByVal lpBuffer$, ByVal nSize%)

#End If
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 1 of 14)

```
Type PortData
    Name As String
    Address As Integer
    PortType As String
    EcpModeDescription As String
    EcpModeValue As Integer
    Enabled As Integer
End Type
Global Port(0 To 3) As PortData
Global BaseAddress%
Global PortType$
Global IniFile$

Global EcrAddress%
Global EcrData%
Global EcpDataPortAddress%
Global EppDataPort0Address%
Global IndexOfSelectedPort%
Global PortDescription$

Global EcpExists%
Global SppExists%
Global PS2Exists%
Global EppExists%
```

```
Function GetEcpModeDescription$(EcpModeValue%)
Select Case EcpModeValue
    Case 0
        GetEcpModeDescription = "SPP"
    Case 1
        GetEcpModeDescription = "PS/2"
    Case 10
        GetEcpModeDescription = "Fast Centronics"
    Case 11
        GetEcpModeDescription = "ECP"
    Case 100
        GetEcpModeDescription = "EPP"
    Case 110
        GetEcpModeDescription = "Test"
    Case 111
        GetEcpModeDescription = "Configuration"
End Select
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 2 of 14)

```
Sub GetIniData()
'Use the Windows API call GetPrivateProfileString to read
'user information from an ini file.
Dim NumberOfCharacters
Dim ReturnBuffer As String * 128
Dim Index%
Dim WindowsDirectory$
'Get the Windows directory, where the ini file is stored.
NumberOfCharacters = GetWindowsDirectory(ReturnBuffer, 127)
WindowsDirectory = Left$(ReturnBuffer, NumberOfCharacters)
IniFile = WindowsDirectory + "\lptprogs.ini"

'If the ini file doesn't exist, don't try to read it.
If Not Dir$(IniFile) = "" Then
    'The port addresses:
    Port(0).Address = _
CInt(VbGetPrivateProfileString("lptdata","Port0Address",
 IniFile))
    Port(1).Address = _
CInt(VbGetPrivateProfileString("lptdata","Port1Address",
 IniFile))
    Port(2).Address = _
CInt(VbGetPrivateProfileString("lptdata","Port2Address",
 IniFile))
    Port(3).Address = _
CInt(VbGetPrivateProfileString("lptdata","Port3Address",
 IniFile))

    'The port types:
    Port(0).PortType = _
VbGetPrivateProfileString("lptdata", "Port0Type", IniFile)
    Port(1).PortType = _
VbGetPrivateProfileString("lptdata", "Port1Type", IniFile)
    Port(2).PortType = _
VbGetPrivateProfileString("lptdata", "Port2Type", IniFile)
    Port(3).PortType = _
VbGetPrivateProfileString("lptdata", "Port3Type", IniFile)
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 3 of 14)

```
'Port enabled?
    Port(0).Enabled = _
CInt(VbGetPrivateProfileString("lptdata",
 "Port0Enabled",IniFile))
    Port(1).Enabled = _
CInt(VbGetPrivateProfileString("lptdata",
 "Port1Enabled",IniFile))
    Port(2).Enabled = _
CInt(VbGetPrivateProfileString("lptdata",
 "Port2Enabled",IniFile))
    Port(3).Enabled = _
CInt(VbGetPrivateProfileString("lptdata",
 "Port3Enabled",IniFile))

'The selected port
    IndexOfSelectedPort = _
Int(VbGetPrivateProfileString("lptdata", _
"IndexOfSelectedPort", IniFile))
End If
End Sub
```

```
Function ReadEcpMode%(TestAddress%)
'The Ecr mode is in bits 5, 6, and 7 of the ECR.
EcrAddress = TestAddress + &H402
EcrData = Inp(EcrAddress)
ReadEcpMode = (EcrData And &HE0) \ &H20
End Function
```

```
Function ReadEppTimeoutBit%(BaseAddress%)
'Reads and clears the EPP timeout bit (Status port bit 0).
'Should be done after each EPP operation.
'The method for clearing the bit varies, so try 3 ways:
'1. Write 1 to Status port bit 0.
'2. Write 0 to Status port, bit 0.
'3. Read the Status port again.
Dim StatusPortAddress%
StatusPortAddress = BaseAddress + 1
ReadEppTimeoutBit = BitRead(StatusPortRead(BaseAddress), 0)
Out StatusPortAddress, 1
Out StatusPortAddress, 0
ReadEppTimeoutBit = BitRead(StatusPortRead(BaseAddress), 0)
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 4 of 14)

```
Sub SetEcpMode(EcpModeValue%)
'Store the Ecp mode's value and description in the Port array.
Port(IndexOfSelectedPort).EcpModeValue = EcpModeValue
Port(IndexOfSelectedPort).EcpModeDescription = _
GetEcpModeDescription(EcpModeValue)
EcrAddress = BaseAddress + &H402
'Read the ECR & clear bits 5, 6, 7.
EcrData = Inp(EcrAddress) And &H1F
'Write the selected value to bits 5, 6, 7.
EcrData = EcrData + EcpModeValue * &H20
Out EcrAddress, EcrData
End Sub
```

```
Sub ShutDown()
WriteIniData
End
End Sub
```

```
Sub StartUp()
Dim PortExists%
Dim Index%
'Get information from the ini file.
GetIniData

'Load the forms.
frmMain.Left = (Screen.Width - frmMain.Width) / 2
frmMain.Top = (Screen.Height - frmMain.Height) / 2
Load frmSelectPort
frmSelectPort.optPortName(IndexOfSelectedPort).Value = True
frmMain.Show
End Sub
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 5 of 14)

```
Function TestForEcp%(TestAddress%)
'Test for the presence of an ECP.
'If the ECP is idle and the FIFO empty,
'in the ECP's Ecr (at Base Address+402h),
'bit 1(Fifo full)=0, and bit 0(Fifo empty)=1.
'The first test is to see if these bits differ from the
'corresponding bits in the Control port (at Base Address+2).
'If so, a further test is to write 34h to the Ecr,
'then read it back. Bit 1 is read/write, and bit 0 is read-only.
'If the value read is 35h, the port is an ECP.
Dim EcrBit0%, EcrBit1%
Dim ControlBit0%, ControlBit1%
Dim ControlPortData%
Dim TestEcrAddress%
Dim OriginalEcrData%
TestForEcp = False
EcrAddress = TestAddress + &H402
'Read ECR bits 0 & 1 and Control Port bit 1.
EcrData = Inp(EcrAddress)
EcrBit0 = BitRead(EcrData, 0)
EcrBit1 = BitRead(EcrData, 1)
ControlPortData = ControlPortRead(TestAddress)
ControlBit1 = BitRead(ControlPortData, 1)
If EcrBit0 = 1 And EcrBit1 = 0 Then
    'Compare Control bit 1 to ECR bit 1.
    'Toggle the Control bit if necessary,
    'to be sure the two registers are different.
    If ControlBit1 = 0 Then
        ControlPortWrite TestAddress, &HF
        ControlPortData = ControlPortRead(TestAddress)
        ControlBit1 = BitRead(ControlPortData, 1)
    End If
    If EcrBit1 <> ControlBit1 Then
        OriginalEcrData = EcrData
        Out EcrAddress, &H34
        EcrData = Inp(EcrAddress)
        If EcrData = &H35 Then
            TestForEcp = True
        End If
        'Restore the ECR to its original value.
        Out EcrAddress, OriginalEcrData
    End If
End If
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 6 of 14)

```
Function TestForEpp%(TestAddress%)
'Write to an Epp register, then read it back.
'If the reads match the writes, it's probably an Epp.
'Skip this test if TestAddress = 3BCh.
Dim ByteRead%
Dim StatusPortData%
Dim EppAddressPort%
Dim TimeoutBit%
Dim StatusPortAddress%
StatusPortAddress = TestAddress + 1
TestForEpp = False
'Use EppAddressPort for testing.
'SPPs, ECPs, and PS/2 ports don't have this register.
EppAddressPort = TestAddress + 3
Out EppAddressPort, &H55
'Clear the timeout bit after each EPP operation.
TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
ByteRead = Inp(EppAddressPort)
TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
If ByteRead = &H55 Then
    Out EppAddressPort, &HAA
    TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
    ByteRead = Inp(EppAddressPort)
    TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
    If ByteRead = &HAA Then
        TestForEpp = True
    End If
End If
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 7 of 14)

```
Function TestForEpp%(TestAddress%)
'Write to an Epp register, then read it back.
'If the reads match the writes, it's probably an Epp.
'Skip this test if TestAddress = 3BCh.
Dim ByteRead%
Dim StatusPortData%
Dim EppAddressPort%
Dim TimeoutBit%
Dim StatusPortAddress%
StatusPortAddress = TestAddress + 1
TestForEpp = False
'Use EppAddressPort for testing.
'SPPs, ECPs, and PS/2 ports don't have this register.
EppAddressPort = TestAddress + 3
Out EppAddressPort, &H55
'Clear the timeout bit after each EPP operation.
TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
ByteRead = Inp(EppAddressPort)
TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
If ByteRead = &H55 Then
    Out EppAddressPort, &HAA
    TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
    ByteRead = Inp(EppAddressPort)
    TimeoutBit = ReadEppTimeoutBit%(TestAddress%)
    If ByteRead = &HAA Then
        TestForEpp = True
    End If
End If
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 7 of 14)

```
Function TestForPS2%(TestAddress%)
'Tests a parallel port's Data port for bidirectional ability.
'First, try to tri-state (disable) the Data outputs by
'setting bit 5 of the Control port.
'Then write 2 values to the Data port and read each back
'If the values match, the Data outputs are not disabled,
'and the port is not bidirectional.
'If the values don't match,
'the Data outputs are disabled and the port is bidirectional.
Dim DataInput%
Dim ControlPortData%
Dim OriginalControlPortData%
Dim OriginalDataPortData%

'Set Control port bit 5.
ControlPortWrite TestAddress, &H2F
TestForPS2 = False
'Write the first byte and read it back:
DataPortWrite TestAddress, &H55
DataInput = DataPortRead(TestAddress)
'If it doesn't match, the port is bidirectional.
If Not DataInput = &H55 Then TestForPS2 = True
'If it matches, write another and read it back.
If DataInput = &H55 Then
    DataPortWrite TestAddress, &HAA
    DataInput = DataPortRead(TestAddress)
    'If it doesn't match, the port is bidirectional
    If Not DataInput = &HAA Then
        TestForPS2 = True
    End If
End If
'Reset Control port bit 5
ControlPortWrite TestAddress, &HF
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 8 of 14)

```
Function TestForSpp%(TestAddress%)
'Write two bytes and read them back.
'If the reads match the writes, the port exists.
Dim ByteRead%
'Be sure that Control port bit 5 = 0 (Data outputs enabled).
ControlPortWrite TestAddress, &HF
TestForSpp = False
DataPortWrite TestAddress, &H55
ByteRead = DataPortRead(TestAddress)
If ByteRead = &H55 Then
    DataPortWrite TestAddress, &HAA
    ByteRead = DataPortRead(TestAddress)
    If ByteRead = &HAA Then
        TestForSpp = True
    End If
End If
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 9 of 14)

```
Function TestPort%(PortIndex%)
'Test for a port's presence, and if it exists, the type of port.
'In order, check for presence of ECP, EPP, SPP, and PS/2 port.
'Update the information in the Port array and the display.
Dim EcpModeDescription$
Dim EcpModeValue%
Dim TestAddress%
TestPort = False
EcpExists = False
EppExists = False
SppExists = False
PS2Exists = False
PortType = ""
TestAddress = Port(PortIndex).Address
'Begin by hiding all port details.
frmSelectPort.lblAddress(PortIndex).Visible = False
frmSelectPort.lblType(PortIndex).Visible = False
frmSelectPort.cboEcpMode(PortIndex).Visible = False
EcpExists = TestForEcp(TestAddress)
If EcpExists Then
    PortType = "ECP"
    'Read the current Ecp mode.
    EcpModeValue = ReadEcpMode(TestAddress)
Else
    'If it's not an ECP, look for an EPP.
    'If TestAddress = 3BCh, skip the EPP test.
    'EPPs aren't allowed at 3BCh due to possible conflict
    'with video memory.
    frmSelectPort.cboEcpMode(PortIndex).Visible = False
    If TestAddress = &H3BC Then
        EppExists = False
    Else
        EppExists = TestForEpp(TestAddress)
    End If
    frmSelectPort.cboEcpMode(PortIndex).Visible = False
    EppExists = TestForEpp(TestAddress)
    If EppExists Then
        PortType = "EPP"
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 10 of 14)

```
    Else
        'If it's not an EPP, look for an SPP.
        SppExists = TestForSpp(TestAddress)
        If SppExists Then
            'Test for a PS/2 port only if the SPP exists
            '(because if the port doesn't exist,
            'it will pass the PS/2 test!)
            PS2Exists = TestForPS2(TestAddress)
            If PS2Exists Then
                PortType = "PS/2"
            Else
                PortType = "SPP"
            End If
        Else
            PortType = ""
        End If
    End If
End If

If PortType = "" Then
   frmSelectPort.optPortName(PortIndex).Enabled = False
   Port(PortIndex).PortType = ""
   Port(PortIndex).Address = 0
   Port(PortIndex).Enabled = False
Else
    TestPort = True
    Port(PortIndex).Enabled = True
    Port(PortIndex).PortType = PortType
    Port(PortIndex).Enabled = True
    If EcpExists Then
        Port(PortIndex).EcpModeValue = EcpModeValue
        Port(PortIndex).EcpModeDescription = _
          GetEcpModeDescription(EcpModeValue)
    End If
End If
UpdateLabels
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 11 of 14)

```
Sub UpdateLabels()
'Use the information in the Port array to update the display.
Dim Index%
Dim EcpModeValue%
For Index = 0 To 3
    frmSelectPort.lblAddress(Index).Caption = _
      Hex$(Port(Index).Address) + "h"
    If Port(Index).Enabled = True Then
        frmSelectPort.optPortName(Index).Enabled = True
        frmSelectPort.lblAddress(Index).Visible = True
        frmSelectPort.lblType(Index).Caption = _
          Port(Index).PortType
        frmSelectPort.lblType(Index).Visible = True
        If Port(Index).PortType = "ECP" Then
            EcpModeValue = ReadEcpMode(Port(Index).Address)
            frmSelectPort.cboEcpMode(Index).ListIndex = _
            EcpModeValue
            Port(Index).EcpModeValue = EcpModeValue
            Port(Index).EcpModeDescription = _
            GetEcpModeDescription(EcpModeValue)
            frmSelectPort.cboEcpMode(Index).Visible = True
        Else
            frmSelectPort.cboEcpMode(Index).Visible = False
        End If
    Else
        frmSelectPort.optPortName(Index).Enabled = False
        frmSelectPort.lblAddress(Index).Visible = False
        frmSelectPort.lblType(Index).Visible = False
        frmSelectPort.cboEcpMode(Index).Visible = False

    End If
Next Index
End Sub
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 12 of 14)

```
Sub UpdateLabels()
'Use the information in the Port array to update the display.
Dim Index%
Dim EcpModeValue%
For Index = 0 To 3
    frmSelectPort.lblAddress(Index).Caption = _
      Hex$(Port(Index).Address) + "h"
    If Port(Index).Enabled = True Then
        frmSelectPort.optPortName(Index).Enabled = True
        frmSelectPort.lblAddress(Index).Visible = True
        frmSelectPort.lblType(Index).Caption = _
          Port(Index).PortType
        frmSelectPort.lblType(Index).Visible = True
        If Port(Index).PortType = "ECP" Then
            EcpModeValue = ReadEcpMode(Port(Index).Address)
            frmSelectPort.cboEcpMode(Index).ListIndex = _
            EcpModeValue
            Port(Index).EcpModeValue = EcpModeValue
            Port(Index).EcpModeDescription = _
            GetEcpModeDescription(EcpModeValue)
            frmSelectPort.cboEcpMode(Index).Visible = True
        Else
            frmSelectPort.cboEcpMode(Index).Visible = False
        End If
    Else
        frmSelectPort.optPortName(Index).Enabled = False
        frmSelectPort.lblAddress(Index).Visible = False
        frmSelectPort.lblType(Index).Visible = False
        frmSelectPort.cboEcpMode(Index).Visible = False

    End If
Next Index
End Sub
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 12 of 14)

```
Sub WriteIniData()
Dim BaseAddressWrite%
Dim PortTypeWrite%
Dim Index%
Dim IniWrite

'Use Windows API call WritePrivateProfileString to save
'initialization information.
'If the ini file doesn't exist, it will be created and stored in
'the Windows directory.

'The port addresses:
IniWrite = WritePrivateProfileString _
("lptdata", "Port0Address", CStr(Port(0).Address), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port1Address", CStr(Port(1).Address), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port2Address", CStr(Port(2).Address), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port3Address", CStr(Port(3).Address), IniFile)

'The port types:
IniWrite = WritePrivateProfileString _
("lptdata", "Port0Type", Port(0).PortType, IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port1Type", Port(1).PortType, IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port2Type", Port(2).PortType, IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port3Type", Port(3).PortType, IniFile)

'Port enabled?
IniWrite = WritePrivateProfileString _
("lptdata", "Port0Enabled", CStr(Port(0).Enabled), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port1Enabled", CStr(Port(1).Enabled), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port2Enabled", CStr(Port(2).Enabled), IniFile)
IniWrite = WritePrivateProfileString _
("lptdata", "Port3Enabled", CStr(Port(3).Enabled), IniFile)
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 13 of 14)

```
'Find the selected port and save it:
Index = 4
Do
Index = Index - 1
Loop Until (frmSelectPort.optPortName(Index).Value = True) _
Or Index = 0
IniWrite = WritePrivateProfileString("lptdata", _
 "IndexOfSelectedPort", CStr(Index), IniFile)
End Sub
```

```
Function VbGetPrivateProfileString$(section$, key$, file$)
    Dim KeyValue$
    'Characters returned as integer in 16-bit, long in 32-bit.
    Dim Characters
    KeyValue = String$(128, 0)
    Characters = GetPrivateProfileStringByKeyName _
    (section, key, "", KeyValue, 127, file)
    KeyValue = Left$(KeyValue, Characters)
    VbGetPrivateProfileString = KeyValue
End Function
```

Listing 4-4: Code for finding and testing ports, and getting and saving initialization data from an *ini* file. (Sheet 14 of 14)

For a short test routine, you can just place the port address in the code:

```
Out &H378, &HAA
```

Or, you can set a variable equal to the port's address, and use the variable name in the program code:

```
BaseAddress = &H378
Out BaseAddress, &HAA
```

Using a variable has advantages. If the port address changes, you need to change the code in just one place. And for anyone reading the code, a descriptive variable name is usually more meaningful than a number.

Most programs will run on a variety of computers, and even on a single computer, the port that a program accesses may change. In this case, it's best to allow the software or user to select a port address while the program is running.

The Port Menu

In Figure 4-1, the startup form contains a *Port* item in the *Setup* menu. Clicking on *Port* brings up a form that enables users to find, test, and select ports. Clicking on *Find Ports* causes the program to look for a port at each of the three standard port addresses. If a port exists, the program tests it to find out whether it's an SPP,

```
[lptdata]
Port0Address=888
Port1Address=632
Port2Address=0
Port3Address=256
Port0Type=ECP
Port1Type=SPP
Port2Type=
Port3Type=SPP
Port0Enabled=-1
Port1Enabled=-1
Port2Enabled=0
Port3Enabled=-1
IndexOfSelectedPort=1
```

Listing 4-5: The contents of an ini file that stores information about the system ports.

PS/2-type, EPP, or ECP. If it's ECP, the program displays a combo box that shows the currently selected ECP mode, which the user can change.

To select a port, you click its option button. The *Test Port* command button tests an individual port and displays the result.

You can also use the routines to test a port under program control. For example, if you're writing a program that will run on many different computers, you may want the software to detect the port type so it can choose the best communications mode available.

Adding a Non-standard Port

The *Add A Port* command button brings up a form that allows you to enter an address of a user port with a non-standard address. You can then use Test Port to determine its type.

Detecting an ECP

In testing a port, you might think that the first step would be to test for an SPP, and work your way up from there. But if the port is an ECP, and it happens to be in its internal SPP mode, the port will fail the PS/2 (bidirectional) test. For this reason, the `TestPort` routine in Listing 4-4 begins by testing for an ECP.

An ECP has several additional registers. One of these, the extended control register (ECR) at *base address + 402h*, is useful in detecting an ECP.

Microsoft's ECP document (see Appendix A) recommends a test for detecting an ECP. First, read the port's ECR at and verify that bit 0 (FIFO empty) =1 and bit 1 (FIFO full) =0. These bits should be distinct from bits 0 and 1 in the port's Control register (at *base address + 2*). You can verify this by toggling one of the bits in the Control register, and verifying that the corresponding bit in the ECR doesn't change. A further test is to write 34h to the ECR and read it back. Bits 0 and 1 in the ECR are read-only, so if you read 35h, you almost certainly have an ECP.

If an ECP exists, you can read and set the port's internal ECP mode in bits 5, 6, and 7 of the ECR. In Listing 4-4, a combo box enables users to select an ECP mode when a port is ECP. Chapter 15 has more on reading, setting, and using the ECP's modes.

Detecting an EPP

If the port fails the ECP test, the program looks for an EPP. Like the ECP, an EPP has additional registers. In the EPP, they're at *base address + 3* through *base address + 6*. These additional registers, and the EPP's timeout bit, provide a couple of ways to test for the presence of an EPP.

One test is to write two values to one of the EPP registers and read them back, much as you would test for an SPP. If there is no EPP-compatible peripheral attached, the port won't be able to complete the EPP handshake. When the transfer times out, the state of the Data port and the EPP register are undefined. However, in my experiments, I was able to read back values written to an EPP register, while other port types failed the test. This is the method used in Listing 4-4. If the reads aren't successful, either the port isn't an EPP or it is an EPP but doesn't pass this test.

If the port's base address is 3BCh, the routine skips the EPP test. This address isn't used for EPPs because the added EPP registers (3BFh–3C3) may conflict with video memory. One such conflict is register 3C3h, which may contain a bit that enables the system's video adapter. Writes to this register can blank the screen and require rebooting!

Another possible test is to detect the EPP's timeout bit, at bit 0 of the Status port (*base address + 1*). On ports that aren't EPPs, this bit is unused. On an EPP, if a peripheral doesn't respond to an EPP handshake, the timeout bit is set to 1. If you can detect the setting of the timeout bit, then clear the bit and can read back the result, you almost certainly have an EPP.

The problem with using the timeout bit to detect an EPP is that ports vary in how they implement the bit. On some EPPs (type 1.9), the timeout bit is set if you attempt an EPP transfer with nothing attached to the port. On others (type 1.7), to force a timeout you must tie *nWait* (*Busy*, or Status port bit 7) low. Ports also vary

in the method required to clear the timeout bit. On some ports, you clear the bit to 0 by writing 1 to it. On others, reading the Status port twice clears the bit. And it's possible that on still other ports, you clear the bit in the conventional way, by writing 0 to it.

So, to use the timeout bit to detect an EPP, you need to bring Status bit 7 low (in case it's type 1.7), then attempt an EPP read or write cycle, by writing a byte to *base address + 3,* for example. Then read the timeout bit. If it's set to 1, write both 1 and 0 to the bit to attempt to clear it, then read the bit. If it's zero, you have an EPP. (You can also use this difference to detect whether an EPP is type 1.7 or 1.9.) Some controller chips, such as Intel's 82091, don't seem to implement the timeout bit at all, or at least don't document it. (The chip's data sheet doesn't mention the timeout bit.)

Detecting an SPP

If a port fails both the ECP and EPP tests, it's time to test for an SPP. To do this the program writes two values to the Data port and reads them back. If the values match, the port exists. Otherwise, the port doesn't exist, or it's not working properly. Also note that the port-test routine only verifies the existence of the Data port. It doesn't test the Status and Control lines. The other port types should also pass this test.

Detecting a PS/2-type Port

If the port passes the SPP test, the final test is for simple bidirectional ability (PS/2-type). The program first tries to put the port in input mode by writing 1 to bit 5 in the port's Control register (*base address + 2).* If the port is bidirectional, this tri-states the Data port's outputs. Then the test writes two values to the Data port and reads each back. If the outputs have been tri-stated, the reads won't match what was written, and the port is almost certainly bidirectional. If the reads do match the values written, the program is reading back what it wrote, which tells you that the Data-port outputs weren't disabled and the port isn't bidirectional.

An ECP set to its internal PS/2 mode should also pass this bidirectional test. Some EPPs support PS/2 mode, while other don't. You should test for a PS/2-type port only after you've verified that a port exists at the address. Because the PS/2 test uses the failure of a port read to determine that a port is bidirectional, a non-existent port will pass the test!

Using the Port Information

The program stores information about the ports in a user-defined array. For each port, the array stores the base address, port type, and whether or not it's the

selected port. For ECPs, the array also stores two values: an integer equal to the ECP's currently selected internal mode (as stored in the ECR) and a string that describes the mode ("SPP", "ECP", etc.). The port's array index ranges from 0 to 2, or *Lpt number - 1*, with the user port, if available, having an index of 3.

Applications can use the information in the port array to determine which port is selected, and what its abilities are.

When the program ends, the *ini* file stores the port information. When the program runs again, it reads the stored information into the port array. This way, the program remembers what ports are available and which port the program used last. If you add, remove, or change the configuration of any ports in the system, you'll need to click *Find Ports* to update the information.

Automatic Port Selection

Rather than testing each of the standard addresses to find existing ports, another approach is to read the port addresses stored in the BIOS data area beginning at 40:00. In 16-bit programs, you can use VbAsm's `VbPeekW` (See Chapter 2) to read these addresses:

```
Dim PortAddress(1 to 3)%
Dim Segment%
Dim LptNumber%
'memory segment of BIOS table
Segment = &H40
For LptNumber = 1 to 3
    Offset = LptNumber * 2 + 6
    PortAddress(LptNumber) = vbPeekw(Segment, Offset)
Next LptNumber
```

Autodetecting a Peripheral

An intelligent peripheral can enable an application to detect its presence automatically. For example, on power-up, the peripheral might write a value to its Status lines. The PC's software can read each of the standard port addresses, looking for this value, and on detecting it, the PC's software can write a response to the Data lines. When the peripheral detects the response, it can send a confirming value that the PC's software recognizes as "Here I am!" The program can then select this port automatically, without the user's having to know which port the peripheral connects to.

5

Experiments

You can learn a lot about the parallel port by doing some simple experiments with it. This chapter presents a program that enables you to read and control each of the port's 17 bits, and an example circuit that uses switches and LEDs for port experiments and tests.

Viewing and Controlling the Bits

Figure 5-1 shows the form for a program that enables you to view and control the bits in a port's Data, Status, and Control registers. The program is based on the form template described in Chapter 4. Listing 5-1 shows the code added to the template for this project.

The screen shows the Data, Status, and Control registers for the port selected in the *Setup* menu. Clicking the *Read All* button causes the program to read the three registers and display the results. Clicking a Data or Control bit's command button toggles the corresponding bit and rereads all three registers. The Status port is read-only, so it has no command buttons. On the Control port, bits 6 and 7 have no function and can't be written to. These bits do have command buttons, and you can verify that the values don't change when you attempt to toggle them. On an SPP, Control port bit 5 is read-only, and its state is undefined. In other modes, set-

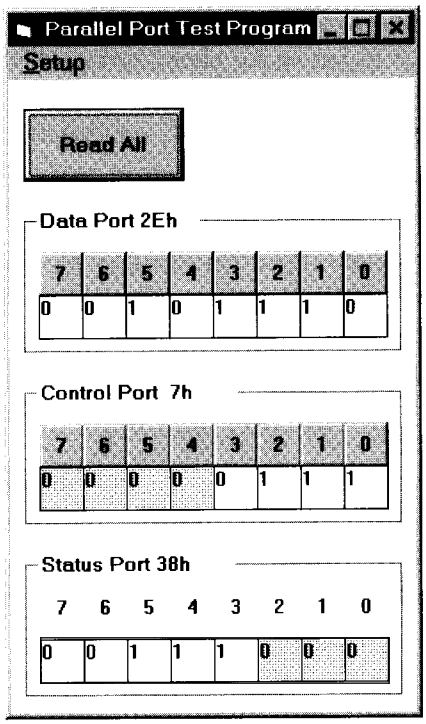


Figure 5-1: The form for the port-test program.

ting bit 5 to 1 disables the Data outputs, so if this bit is 1, you won't be able to toggle the Data-port bits.

Circuits for Testing

Figure 5-2 Figure 5-3, and Figure 5-4 show circuits you can use to test the operation of a parallel port, using Figure 5-1's program or your own programs.

In Figure 5-2, the port's Data outputs each control a pair of LEDs. As you click on a Data button, the LEDs should match the display: red for 1 and green for 0. Instead of using LEDs, you can monitor the bits with a voltmeter, logic probe, or oscilloscope.

In Figure 5-3, switches determine the logic states at the Status inputs. Opening a switch brings an input high, and closing it brings the input low. After clicking *Read Ports*, the display should match the switch states.

Figure 5-4 shows the Control port. As with the Data port, a pair of LEDs shows the states of the Control outputs. On an SPP, writing 1 to a Control bit enables you to read the state of the switch connected to that bit. If you have an ECP, EPP, or PS/2-type port, the Control bits may be open-collector type only when in SPP

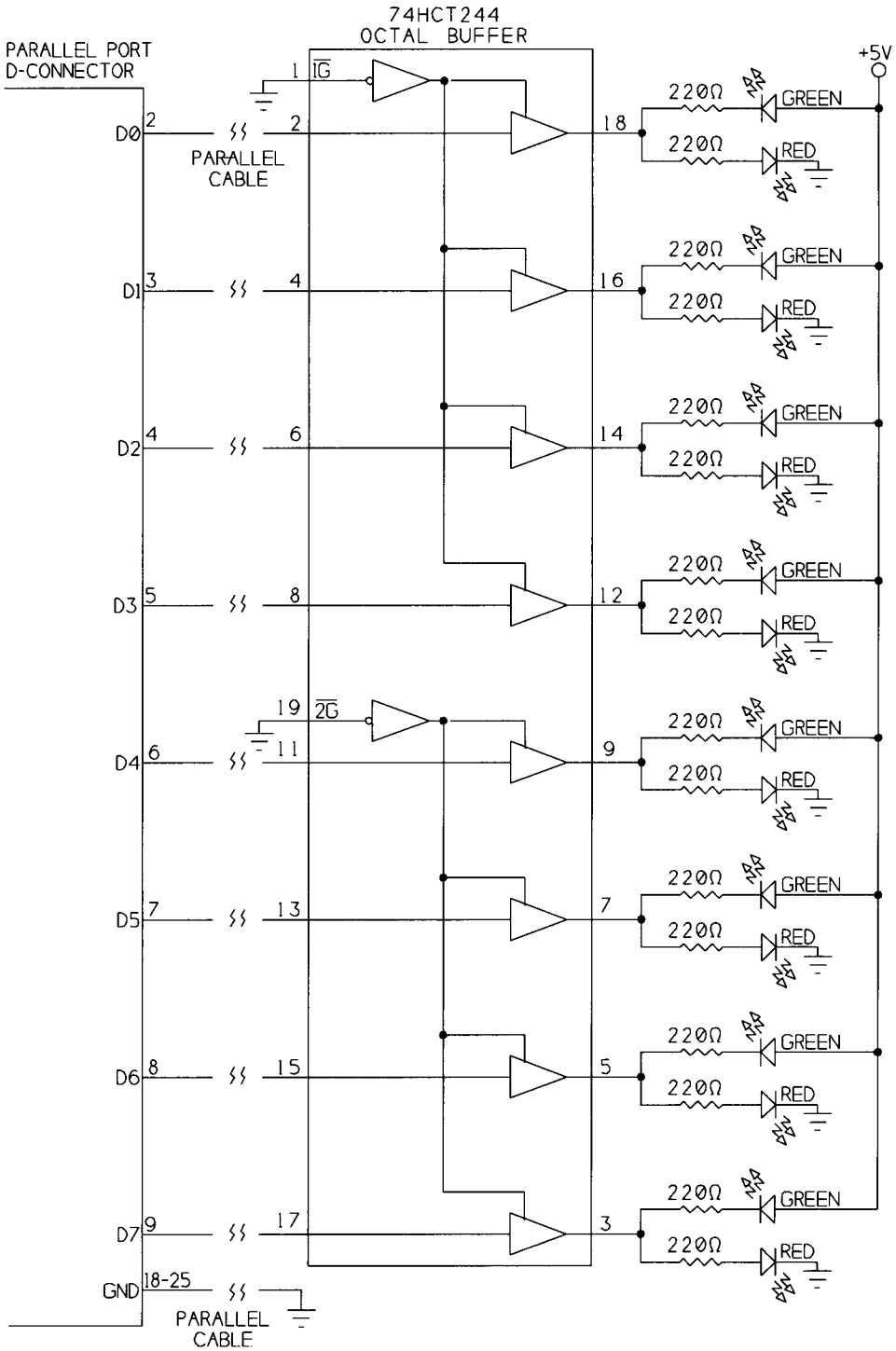


Figure 5-2: Buffer and LEDs for monitoring Data outputs.

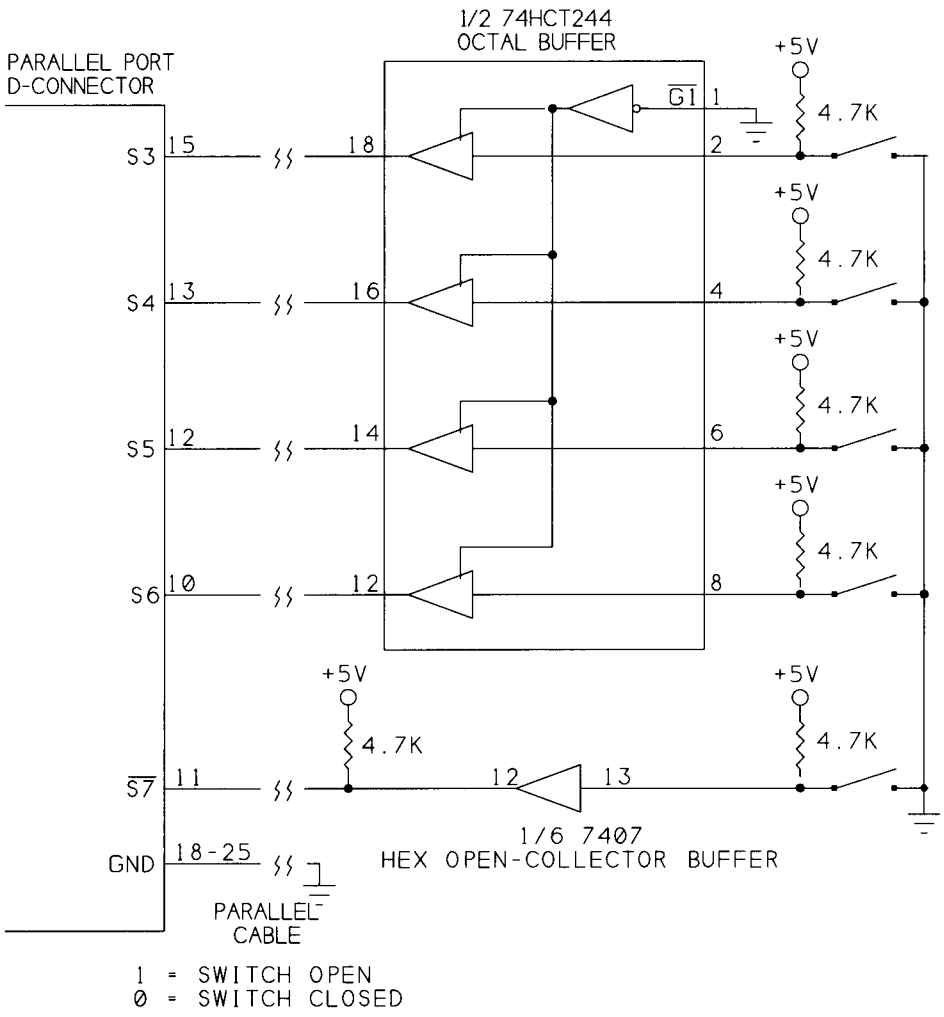


Figure 5-3: Driver and switches for testing Status port.

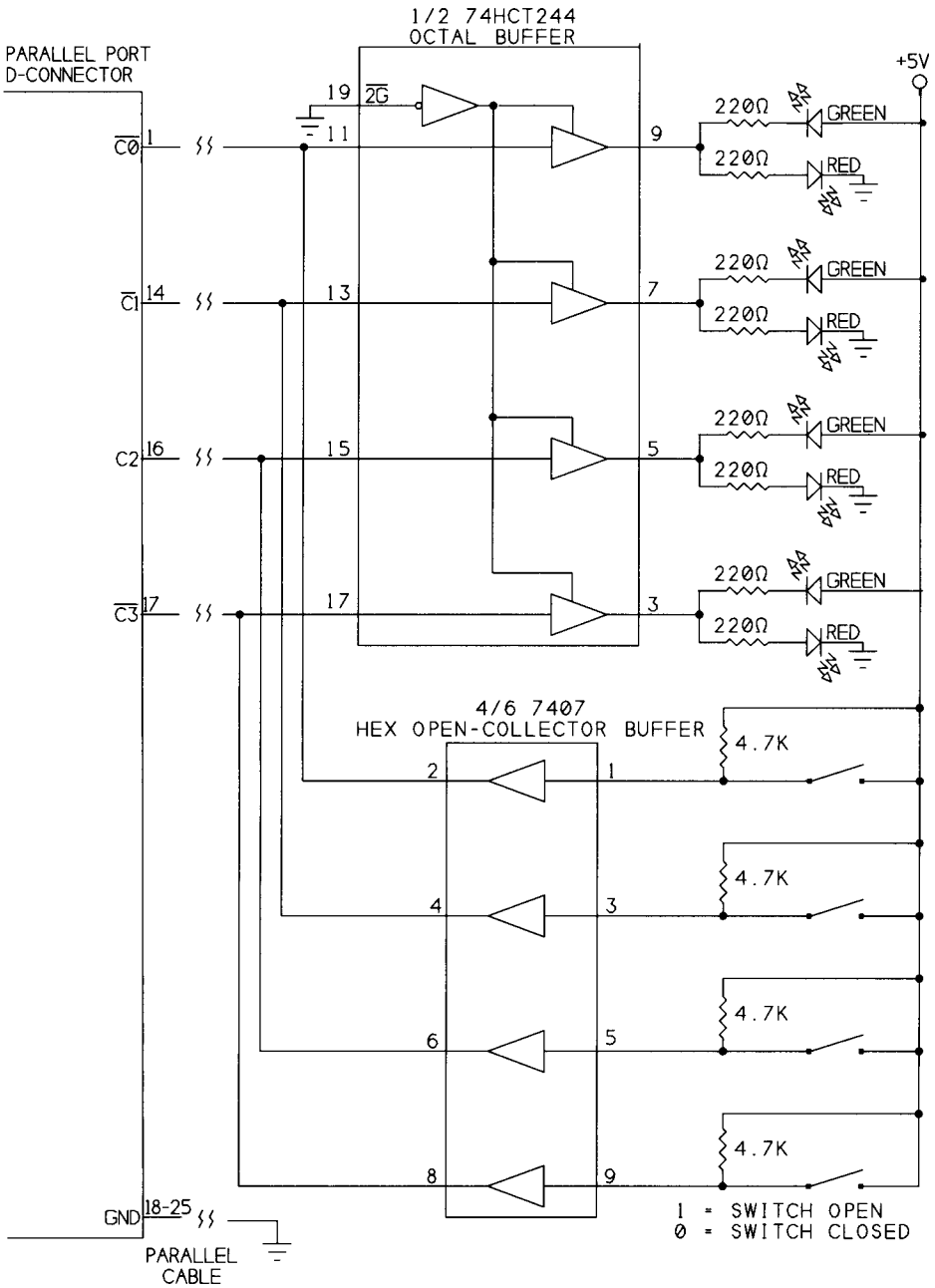


Figure 5-4: Buffer/driver, LEDs, and switches for Control-port testing.

```
Sub cmdControlBitToggle_Click (Index As Integer)
'toggle a bit at the Control port
Dim ControlPortData As Integer
ControlPortData = ControlPortRead(BaseAddress)
BitToggle ControlPortData, Index
ControlPortWrite BaseAddress, ControlPortData
ReadPorts (BaseAddress)
End Sub
```

```
Sub cmdDataBitToggle_Click (Index As Integer)
'toggle a bit at the Data port
Dim DataPortData As Integer
DataPortData = DataPortRead(BaseAddress)
BitToggle DataPortData, Index
DataPortWrite BaseAddress, DataPortData
ReadPorts (BaseAddress)
End Sub
```

```
Sub cmdReadAll_Click ()
ReadPorts (BaseAddress)
End Sub
```

Listing 5-1: Code for Figure 5-1's program. (Sheet 1 of 2)

```
Sub ReadPorts (BaseAddress As Integer)
'Read the Data, Status, and Control ports of selected port.
'Display the byte read and each bit in the byte.
Dim ByteRead%
Dim BitNumber%
Dim BitValue%

ByteRead = DataPortRead(BaseAddress)
frmMain.lblDataPortByte(0).Caption = Hex$(ByteRead) + "h"
For BitNumber = 0 To 7
    BitValue = BitRead(ByteRead, BitNumber)
    frmMain.lblDataBit(BitNumber).Caption = BitValue
Next BitNumber

ByteRead = StatusPortRead(BaseAddress)
frmMain.lblStatusPortByte(0).Caption = Hex$(ByteRead) + "h"
For BitNumber = 0 To 7
    BitValue = BitRead(ByteRead, BitNumber)
    frmMain.lblStatusBit(BitNumber).Caption = BitValue
Next BitNumber

ByteRead = ControlPortRead(BaseAddress)
frmMain.lblControlPortByte(0).Caption = Hex$(ByteRead) + "h"
For BitNumber = 0 To 7
    BitValue = BitRead(ByteRead, BitNumber)
    frmMain.lblControlBit(BitNumber).Caption = BitValue
Next BitNumber
End Sub
```

Listing 5-1: Code for Figure 5-1's program. (Sheet 2 of 2)

mode, or not at all. If in doubt, don't connect the 7407's buffer outputs in Figure 5-4.

If you have a bidirectional port, you can use Figure 5-5's circuit. It has a buffer and switches that you can connect to bidirectional Data lines when you're using the lines for input. To prevent the buffer outputs from being enabled when the parallel port's Data outputs are enabled, it's best to have a way to enable and disable the buffers' outputs under program control. In the schematic, a Control line from the parallel port controls the output-enable input of the buffers. Bit $\overline{C3}$ is normally low on bootup. An inverter brings the bit high and disables the buffers. You could also use a manual switch to enable and disable the outputs.

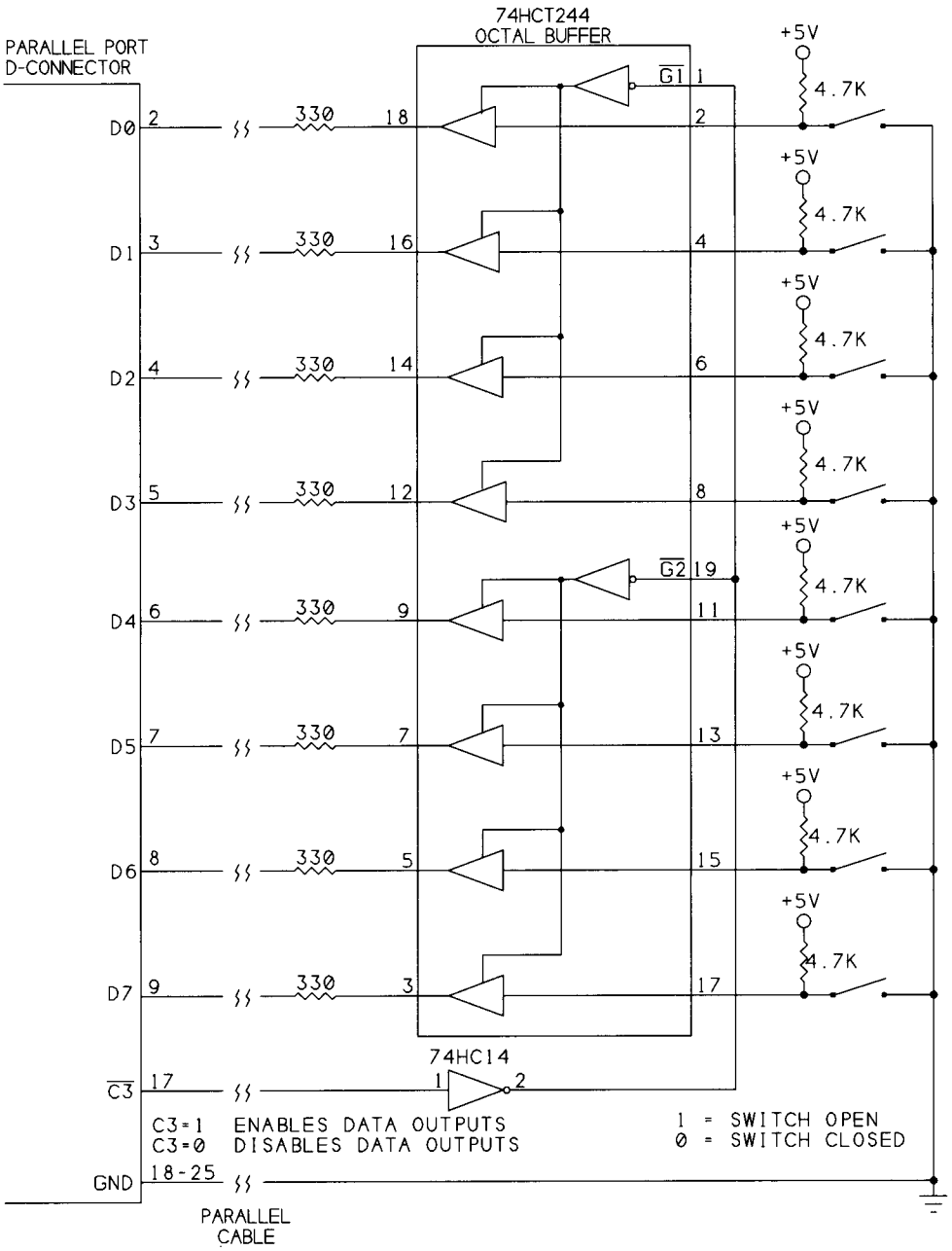


Figure 5-5: Circuit for reading external inputs on a bidirectional Data port.

The 330-ohm resistors protect the circuits on both ends of the link in case the parallel port's outputs and the buffer outputs happen to be enabled at the same time. The resistors limit the current in each line to under 15 milliamperes.

You can connect both Figure 5-2's and Figure 5-5's circuits to the Data port at the same time. Connect the buffer inputs of the '244 (pins, 2, 4, etc.) in Figure 5-2 to the PC (parallel-port D-sub connector) side of the 330-ohm resistors in Figure 5-5.

Buffers and Drivers

The circuit uses HCTMOS-family driver/buffers at inputs *D0-D7* and $\overline{C0}$-$\overline{C3}$ and outputs *S3-S6*. Using HCT-family logic has two benefits. HCT devices have TTL-compatible input voltages, which are compatible with the parallel-port's outputs. Plus, unlike TTL logic, HCT-family outputs can both source and sink enough current to power an LED from either a high or low output.

The outputs that drive inputs $\overline{C0}$-$\overline{C3}$ are 7407 open-collector buffers. One of the remaining 7407 buffers drives $\overline{S7}$, only because any other choice would require adding another chip to the circuit. (You could use a 7407 in place of the 'HC14 in Figure 5-5 as well. Just remember to add a pull-up resistor, and be aware that the 7407 doesn't invert like the 'HC14.)

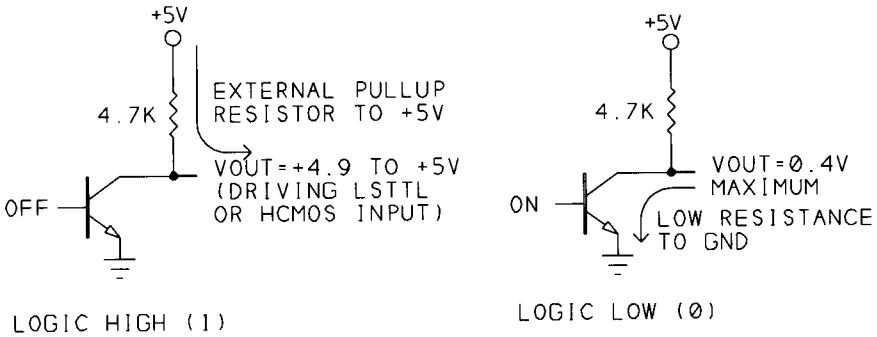
The 7407's open-collector outputs help to protect the Control port's outputs. Each Control output also connects to an input buffer. In early parallel ports, the Control-port outputs were 7405 open-collector inverters with 4.7K pull-up resistors. When an open-collector Control output is high, you can drive its input buffer with another digital output, which you can then read at the Control register. In newer designs, the Control outputs may be push-pull type, so if you want a design to be usable with any port, don't use the Control bits as inputs.

Output Types

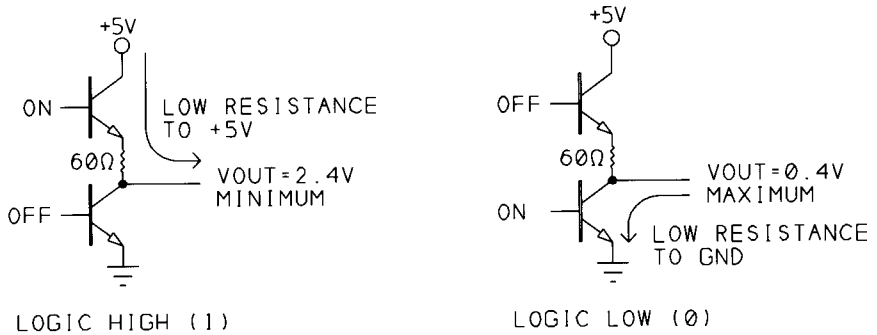
To understand how to use the Control lines (and bidirectional Data lines) for input, it helps to understand the circuits that connect to the port pins. Output configurations common to digital logic are open-collector/open-drain, totem-pole, push-pull, and 3-state.

Open Collector and Open Drain

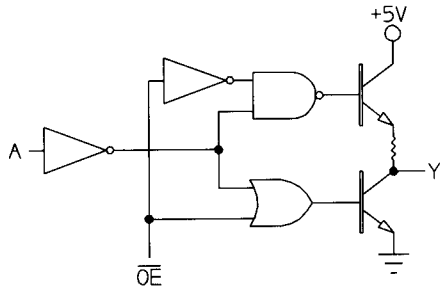
Figure 5-6A shows an open-collector output. The collector of its output transistor is open, or not connected to any circuits on-chip. To use the output, you have to add a pull-up resistor to +5V. When the output transistor switches on, the low resistance from the output pin to ground results in a logic-low output. When the



(A) Open collector outputs:
When two open-collector outputs connect together, any low output brings the combined output low.



(B) Totem-pole outputs:
Can't be tied together. If one output is high and the other is low, the logic level is unpredicatable and the resulting high currents may damage the components.

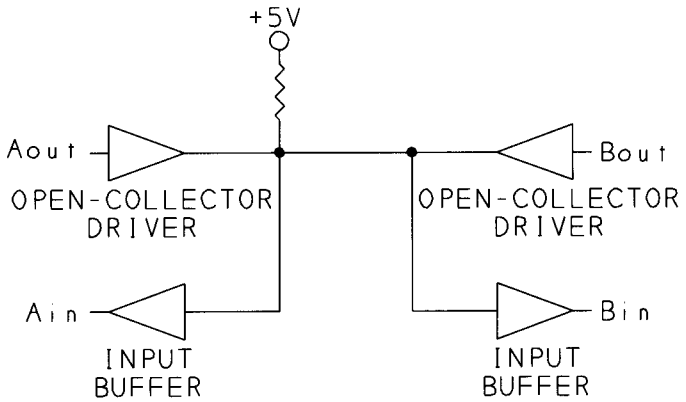


$\overline{OE}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Z = HIGH IMPEDANCE

(C) 3-state outputs:
When $\overline{OE}$ is low, the Y output follows the A input.
When $\overline{OE}$ is high, the output is high impedance.

Figure 5-6: Output types used in digital logic.



Aout	Bout	Ain, Bin
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5-7: A simple way to make a bidirectional link is to use open-collector drivers. When Aout is high, Ain follows Bout. When Bout is high, Bin follows Aout.

output transistor is off, the pull-up resistor brings the output pin to +5V. Another name for the pullup resistor is *passive pullup.*

An advantage to open-collector logic is the ability to tie two or more outputs together. When any of the outputs goes low, the low resistance from the output to ground brings the combined output low.

This arrangement is sometimes called a wired-OR output, though it actually behaves like an OR gate only if you assume negative logic, where a low voltage is a logic 1 and a high voltage is logic 0. Using the more common positive logic, if the individual gates are non-inverting buffers, the circuit behaves like an AND gate: any low input brings the combined output low. If the gates are inverters, the circuit is a NOR gate: any high input brings the combined output low.

You can use the ability to tie outputs together to create a bidirectional data line. Figure 5-7 shows an example of a link with two nodes. Each node has an open-collector output and an input buffer. When 1 is written to Aout, the input buffers follow Bout. When 1 is written to Bout, the input buffers follow Aout. With this arrangement, you can send data in either direction, one way at a time. If both nodes' outputs are low at the same time, the inputs will be low, and the pull-up resistor will limit the current.

In a link with multiple lines like this, you can configure the individual bits at each node to act as inputs or outputs according to the needs of your circuit.

A disadvantage to open-collector logic is its slow switching speed. When an output switches from low to high, the cable's capacitance has to charge through the resistance of the pull-up. The larger the resistance, the more slowly the output voltage changes.

In CMOS components, the equivalent to open-collector is the open-drain output. An example is the 74HCT03, a CMOS quad NAND gate with open-drain outputs. The technology is different, but the operation is much the same.

Some NMOS and CMOS devices have outputs that behave in a way similar to open-collector or open-drain outputs. Instead of an external, passive pull-up, this type of device has an internal transistor with a high resistance that acts as weak, active pull-up. As with open-collector logic, writing 1 to this type of output enables you to read an external logic signal at the bit. The ports on the 8051 and 80C51 microcontrollers are examples of this type of output. Another name for these outputs is *quasi-bidirectional.*

Totem Pole

In contrast to open-collector logic, many LSTTL devices use a type of output called *totem pole,* with two transistors stacked one above the other. Figure 5-6B illustrates. When the output is low, the bottom transistor conducts, creating a low-resistance path from the output to ground, as in an open-collector output. When the output is high, the top transistor conducts, creating a low-resistance path to +5V. The original parallel port used the totem-pole outputs of a 74LS374 to drive the Data lines (*D0-D7*).

In TTL logic, the resistance from a logic-high output to +5V is greater than the resistance of a logic-low output to ground, so a totem-pole output can sink more current to ground than it can source from +5V.

Their lower output resistance means that as a rule, totem-pole outputs can switch faster than open-collector outputs. But it also means that the outputs aren't suitable for bidirectional links. If you tie two totem-pole outputs together, if one is high and the other is low, you have one output with a low resistance to +5V and another with a low resistance to ground. The result is an unpredictable logic level and large currents that may destroy the components involved.

Tying a totem-pole output to an open-collector output is OK as long as the open-collector output stays high. If the open-collector output goes low and the totem-pole output is high, you can end up with the same high current and unpredictable result.

On the parallel port, you can avoid the problem by using only open-collector outputs to drive the Control-port inputs on the parallel port. If you do connect a

totem-pole output to an open-collector output, a 330-ohm series resistor in the line will protect the circuits (though it will slow the switching speed).

Push-pull

Outputs on most digital CMOS logic chips have complementary outputs that are similar to totem-pole, except that the current-sourcing and sinking abilities of the outputs are equal. This type of output is called *push-pull.*

3-state

A third type of output is *3-state*, or *tri-state,* which has a control signal that disables the outputs entirely. For all practical purposes, disabling, or tri-stating, an output electrically disconnects it from any circuits it physically connects to. Figure 5-6C illustrates. When the Output Enable line ($\overline{OE}$) is low, the output follows the input. When $\overline{OE}$ is high, both output transistors are off and the output has no effect on external circuits.

Outputs that connect to computer buses are often 3-state, with address-decoding circuits controlling the output-enable pins. This enables memory chips and other components to share a data bus, with each enabled only when the computer selects the component's addresses.

As with totem-pole logic, if two connected 3-state outputs are on at the same time, the result will be unpredictable. If you can't guarantee the behavior of the outputs in your circuit, open-collector is the safest choice.

Three-state logic also requires an extra input to control each set of outputs. One output-enable bit typically controls all of the bits in a data bus. With open-collector logic, you can easily configure individual bits as either inputs or outputs, with no extra control lines required.

Component Substitutions

If you don't have the exact chips on hand for the circuits in this chapter, you can substitute. With some cautions, you can use almost any HC, HCT, or TTL/LSTTL inverters in many simple circuits. The buffer/driver chips are recommended because they have stronger drivers and their inputs have hysteresis, which gives a clean output transition even when an input is noisy or changes slowly. If you use the Control port for input, open-collector drivers will protect the circuits, as described above.

Logic Families

If you use a 74HC-family buffer instead of the 74HCT244 at *D0-D7*, add a 10K pullup resistor from each buffer's input to +5V. The pullup ensures that the port's outputs will go high enough to meet the 74HC-family's minimum for a logic high. If you don't use a pullup, the circuit will probably work. However, a logic-high TTL output is usually guaranteed to be just 2.4V, while 5V HC-family logic requires at least 3.5V for a logic-high input. HCT-family logic is designed to work with TTL logic voltages, so pull-ups aren't needed.

The Control outputs should already be pulled up by the port circuits, so you shouldn't have to add pullups to them.

You can use a 74LS244 buffer instead of the 74HCT244, but because TTL logic can sink, but not source, enough current to drive an LED, remove the red LEDs and their current-limiting resistors. The green LEDs will light when the corresponding outputs are low, and they will be off when the corresponding outputs are high.

If you use 74HCT240 inverting buffers, swap the red and green LEDs. (Be sure to keep the polarity of the LEDs correct. The cathode always connects to the more negative voltage.) With inverters, the switches will read 1 when closed and 0 when open.

Switches and Power Supplies

You can use any SPST (single-pole, single-throw) toggle or slide switches to control the Data, Status, and Control inputs. Power the circuit with any +5V supply that can provide at least 300 milliamperes. (The LEDs use most of the current.)

Inverting Bits in Hardware

One reason you might use inverters for some of the bits is to reinvert the bits that the port's circuits invert between the connector and the register where you read the port. If you use inverting buffers and drivers for just these bits, you don't have to reinvert bits in software when you read or write to the ports.

For example, in Figure 5-3 you could replace bit 7's buffer with an inverting buffer such as a 7405. If the inverter is an ordinary LSTTL or HCMOS logic gate (not a driver), wire the inverter's output to the 7407's input, and let the 7407 drive the line.

You could also invert the signal by replacing the normally open switch with a normally closed one. Or rewire the normally open switch with a pull-down resistor instead of a pull-up, so that an open switch is logic-low rather than logic-high. With TTL and HCTMOS inputs, however, a pull-up resistor gives better noise immunity. (Noise is usually a greater problem when the switch is open. With a

pull-up, there's a 3V difference between +5V and the minimum TTL logic-high input of 2V. With a pull-down, there's just 0.8V between 0V and the maximum logic-low input.)

Using any of these approaches to reinvert the inverted signals, the values that you write to a port will match the bits at your outputs, with no software complementing required. But if you use any code that assumes that the bits will be inverted as usual, you'll either have to change the routines or reinvert the bits elsewhere in your program. The examples in this book assume no special inversions in the hardware.

Cables & Connectors for Experimenting

Connecting a printer or another commercial product to a parallel port is usually just a matter of plugging the device's cable into the computer and the printer. But for experimenting, you need a cable that allows access to all of the lines. There are several options, depending on whether you're soldering or wire-wrapping components onto perfboard, or using a solderless breadboard.

One approach is to use a standard printer cable and wire a mating Centronics connector to your circuits. This is probably the best solution because you can use a readily available shielded printer cable for the link from the computer to your device. You can buy PC-board-mountable connectors that solder onto perfboard. Or you can use a solder-cup connector and solder individual wires to the connector, with the other ends of the wires soldered to perfboard or plugged into a solderless breadboard.

Another option is to use a cable with D-sub connectors on both ends. Although there are PC-board-mountable D-subs, the pin spacings on the connector don't match the 0.1" grid used by most perfboards. If you want to use perfboard, you'll need to look for one with a hole pattern that will accept a D-sub. Of course, if you're designing your own printed-circuit board, you can add holes and solder pads for the D-sub. Or use a solder-cup D-sub and solder the individual wires to perfboard or plug them into a breadboard.

Yet another possibility is to use ribbon cable with a dual-row socket connector crimped onto one end, and plug the connector into a dual header soldered onto perfboard.

For solderless breadboards, which typically have two parallel rows of contacts spaced 0.3" apart, a convenient solution is to use a ribbon cable with a D-sub on one end and a ribbon-cable DIP connector on the other. The DIP connector has two rows of pins with the same spacing as a DIP IC: the pins within a row are 0.1"

apart, and the rows are 0.3" or 0.5" apart. Use an IDC (insulation-displacement connector) tool or a vise to press the cable onto the contacts. Then plug the connector into a breadboard or perfboard.

It's best to limit cable length to 10 feet if possible, 15 at most. You can try longer cables - even much longer - and you may be able to use them without problems. But if you stretch the limits like this, there are no guarantees. Chapter 6 has more on cables and cable length.

Making an Older Port Bidirectional

If you have one of the older expansion cards that uses a 74LS374 for the Data outputs, a fairly simple modification will enable you to use the Data port for input. Although buying a board with a true bidirectional port is a quick and inexpensive solution, this section describes an alternative for the determined.

Cautions

First of all, be warned that this method works only with parallel-port cards that use the TTL chips described below. Not all cards will follow the exact design of the original port, so unless you happen to have a schematic of your card, you'll need to do some signal tracing with an ohmmeter to find out exactly how the signals on your card are routed. The modification requires cutting one lead on the 74LS374 and adding at least one jumper wire. You've been warned; proceed at your own risk!

Second, there is one difference about cards modified with this method. The modification allows you to use Control bit 5 to enable and disable the Data outputs, as you do on other bidirectional ports. On these other ports, you can read this bit as well as write to it. On a port that's modified to be bidirectional, the bit is write-only, because the early cards have no input buffer for Control bit 5 (unless you can find a spare buffer and wire the connections). Because of this difference, you have to be careful not to inadvertently turn off the Data outputs by writing 1 to Control bit 5.

Reading the bit on a modified port returns a 1. This means that if you read the Control port, then write the same value back to the port, bit 5 will be set to 1, which disables the Data outputs. A program that writes to the Control port of a modified port should always write 0 to Control bit 5 if the Data port is being used for output. If the Data port is being used for input, the program should always write 1 to Control bit 5.

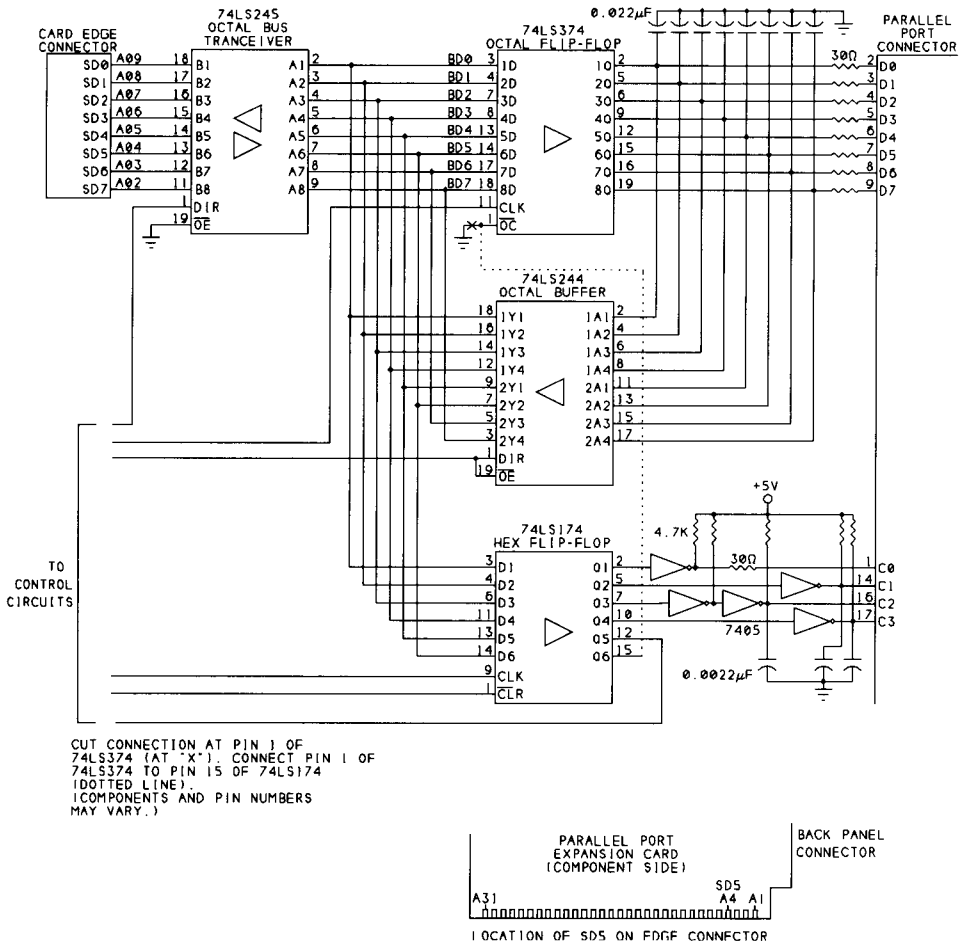


Figure 5-8: On many older parallel ports, you can make the Data port bidirectional by cutting one connection and adding a jumper wire.

On most true bidirectional ports, you don't have to worry about whether the Data port is input or output. You can just read the port and write back the same value for bit 5, and the bit won't change.

The Circuits

Figure 5-8 shows the relevant parts in the design of a typical early parallel port. Not shown are the Control and Status port's input buffers or the address-decoding and other control signals.

Lines *SD0-SD7* on the expansion bus carry Data bits *D0-D7*. On the parallel-port card, a 74LS245 octal transceiver buffers *AD0-AD7*. The lines that connect to

A1-A8 on the transceiver form a bidirectional, buffered Data bus (*BD0-BD7*). When the 74LS245's direction Control input (*DIR*) is low, *B1-B8* are inputs and *A1-A8* are outputs. When *DIR* is high, *A1-A8* are inputs and *B1-B8* are outputs.

(Most of the chips in this circuit use the numbering 1 through 8 for sets of eight bits, but the parallel port's Data and Control bits and the buffered data bus are numbered beginning with 0.)

When the CPU writes to the Data port, *BD0-BD7* drive the inputs of a 74LS374 octal flip-flop. The outputs of the flip-flops connect through 30-ohm resistors to *DC0-DC7* on the parallel-port connector. These lines also connect to the inputs of a 74LS244 octal buffer, and the buffer's outputs connect back to *BD0-BD7*. This buffer is what enables you to read the last byte written to the Data port.

The '374's Output-Control input ($\overline{OC}$) connects to *GND*, so its outputs are always enabled. If you could disable the outputs, external signals at the connector's *D0-D7* could drive the '244's inputs, and reading the Data port would tell you the logic states of *D0-D7* at the connector.

At the Control port, six bits (*C0-C5*) drive the inputs of a 74LS174 hex flip-flop. Outputs *Q1-Q4* connect to 7405 open-collector inverters, whose outputs are wired to *C0-C3* at the connector. Output *Q5* (*C4* in the Control register) controls the interrupt-enable circuits. and output *Q6* (*C5*) connects to nothing at all. This is the bit you can use to enable and disable the Data outputs.

The Changes

To make the modification, you cut the connection from the 74LS374's $\overline{OC}$ (pin 1) to ground and instead wire this pin to *Q6* (pin 15) on the 74LS174.

To break pin 1's connection, use a wire snips to clip pin 1's lead, then bend the stub on the chip so it doesn't touch the bottom of the leg it's cut away from. Then take a short length of insulated wire (#30 wire-wrap wire works well) and trim 1/8" or so of insulation from each end. Solder one end of the wire to the stub of pin 1 on the '374, and solder the other end to pin 15 on the 74LS174.

Bit *C5* will then determine the port's direction. Writing 0 to *C5* enables the Data outputs, for an output port, and writing 1 to *C5* disables the outputs and allows you to use the Data port for input. Because *C5* has no input buffer, you can't read it; all reads of the bit will return 1.

Not all cards will follow the exact wiring of Figure 5-7. To determine the wiring on your card, first use an ohmmeter to find the connection between *SD5* and the 74LS245. The schematic shows the location of *SD5* (at *A4*) on the card connector. The 74LS245 may be wired with either the *A* or *B* lines connected to the expansion bus, so check all 16 signal pins to find the connection.

If you don't find a connection, your card is too different from the original design to speculate on here, so you're out of luck unless you can figure out the connections yourself.

If you do find a connection, you can determine which pin on the 74LS245 is the corresponding I/O pin. For example, in Figure 5-8, pin 13 (*B6*) corresponds to pin 7 (*A6*). (Again, the signal names are numbered from 1 to 8 rather than from 0 to 7.) This pin should connect to one of the *D* inputs on the 74LS174. Use an ohmmeter to find the connection.

On one board that I modified, there was no connection from *BD5* to the '174, but the '174 did have an unused input. If you don't find the connection on your board, you can use the process of elimination to see if you have a spare input. Use an ohmmeter to trace the existing connections from *BD0-BD5* to the 74LS174. Then determine which input remains. If you don't see any pc-board traces connected to this pin (check both sides of the board), chances are that it's unused and you can solder a wire from it to *BD5* (in Figure 5-8, pin 7 of the '245).

When you've found the pin, determine its corresponding *Q* output. For example, in Figure 5-8, pin 14 (*D6*) of the '174 corresponds to pin 15 (*Q6*). Wire this *Q* output to the stub of pin 1 on the 74LS374 and you're done. Reinstall the port card and you're ready to test it. (Chapter 4 has a bidirectional-test program.)

Note that the Data outputs of this port are the totem-pole outputs of a 74LS374. If you intend to use the Data port for input, you must disable the Data outputs before you connect external outputs to the Data lines. Otherwise, you risk damaging the port circuits. To protect the outputs, you can add a 330-ohm series resistor on each Data line, to limit the current in case this situation occurs. This will affect the impedance match on the lines and limit the link's performance at high speeds, however.

6

Interfacing

Because parallel-port signals may travel over cables of ten feet or more, the cable's design and the circuits that interface to the cable can mean the difference between a circuit that works reliably and one that fails, if not completely and immediately, then intermittently and unpredictably. The cable and interface can also affect the maximum speed of data transfers. This chapter includes tips on designing circuits that connect to the parallel port, and on choosing cables to connect the circuits. There's also a section on how and when you can use the parallel port as a power source for low-power devices.

Port Variations

Many parallel ports use ordinary TTL logic, or at best bus drivers and buffers, as the cable interface. On the original parallel port, a 74LS374 flip-flop drove the eight Data lines, 7405 open-collector inverters drove the Control lines, and the Status lines connected to inputs of LSTTL logic gates. These days there's no way to know exactly what components a PC or peripheral may use for its parallel-port circuits.

Although all parallel ports have the same 17 bits, the bits can differ in characteristics such as output impedance and noise immunity. Although every parallel port's outputs *should* have at least the same current-sourcing and sinking ability as the

original port, some ports do have weaker drivers. A symptom of weak drivers is when a port works only with short cables, or at low speeds. Some very low-power devices that connect to the parallel port don't use an external power supply, and draw their current from the port's outputs, and these devices may not work with weak ports.

The outputs of many of the newer port controllers meet the improved Level 2 interface described in IEEE 1284. These ports can use cables of over 30 feet, if they connect to another Level 2 device.

Drivers and Receivers

The IEEE 1284 standard specifies characteristics for parallel-port drivers and receivers. It describes two types of devices: Level 1 devices are similar to the design of the original parallel port, while Level 2 devices give better performance while remaining compatible with the original interface. A port with Level-2 drivers and receivers can connect to a port with Level-1 drivers and receivers without problems, though you won't get the full benefit of using Level 2 devices unless they're present on both ends of the link. Both assume a power supply of +5V.

Level 1 Devices

The specification for Level-1 drivers and receivers are met by off-the-shelf LSTTL, TTL, and HCTMOS components, including those in the original parallel port.

Drivers

These are the characteristics of Level 1 drivers:

Logic-high outputs: +2.4V minimum at 0.32ma source current.

Logic-low outputs: +0.4V maximum at 12ma sink current.

Pullup resistors (if used): 1.8K minimum on Control and Status lines, 1.0K minimum on Data lines.

Not surprisingly, since they were the chips used in the original parallel port, LSTTL drivers are a good choice for the Data outputs, with 7405s or similar TTL gates for the open-collector Control outputs.

LSTTL chips characterized as buffer/drivers easily meet the requirements. These include the 74LS24X series and the 74LS374 octal flip-flop. On the 74LS240, low outputs are guaranteed to sink 12 milliamperes at 0.5V, and high outputs are guaranteed to source 3 milliamperes at 2.4V, compared to 4 and 0.4 milliamperes for ordinary LSTTL. Table 6-1 shows chips you might use:

Table 6-1: Level-1 driver and buffer chips for parallel-port circuits.

Drivers for the Data, Status, and Control inputs:	
74LS244, 74HC(T)244	octal buffer
74LS240, 74HC(T)240	octal inverting buffer
7405, 7406	open-collector hex inverting buffer
7407, 7417	open-collector hex buffer
(Use open-collector drivers for the Control lines.)	
Schmitt-trigger buffers for the Data or Control outputs:	
74LS14, 74HCT14	hex inverter
74LS374	octal buffered flip-flop
74LS244	octal buffer
74LS240	octal inverting buffer

In normal operation, the outputs don't provide their maximum rated currents continuously, but the ability to source and sink high currents means that the output has low impedance, and this in turn implies that the output can switch quickly. As an output switches, the voltage must charge or discharge through the cable's capacitance, and the lower the output impedance, the faster the voltage can change.

Ordinary LSTTL logic gates, like the 74LS14 hex inverter, are guaranteed to sink just 8 milliamperes at 0.4V, so these aren't recommended for driving a parallel cable. Standard TTL, such as the 7405, does meet the requirements. The drawback to using standard TTL is that each chip draws 20–40 milliamperes, compared to 8–12 milliamperes for an equivalent LSTTL chip, or 15–35 milliamperes for an LSTTL octal driver.

The HCMOS family has equivalents to most LSTTL chips. However, the data sheets for the 74HC24X buffer/drivers don't include enough information to guarantee that these chips meet the Level 1 requirements. With a power supply of 4.5V, the outputs are guaranteed to sink 6 milliamperes at 0.33V. The sink current will be greater with a 5V supply and 0.4V output, but the data sheets don't include figures for these conditions. Overall, the outputs of HCMOS driver chips aren't are strong as LSTTL, although in most situations, they'll work without problems.

Receivers

These are the characteristics of Level 1 receivers:

Logic-high inputs: 2.0V maximum at 0.32ma sink current.

Logic-low inputs: 0.8V minimum at 12ma source current.

Pullup resistors (if used): recommended minimum values are 470 ohms on Control and Status lines, 1000 ohms on Data lines.

Rise and fall time (between 0.8V and 2.0V): 120ns maximum.

Input limits: inputs should withstand transient voltages from -2.0V to +7.0V.

Just about any LSTTL or HCTMOS input will meet the above requirements. HCMOS chips aren't a good choice, however, because their minimum voltage guaranteed for a logic-high input is 3.5V, which is 1.5V greater than the 2V (TTL-compatible) requirement. If you do use an HCMOS chip, add a pull-up resistor from the input to +5V. HCTMOS devices have TTL-compatible inputs, so you don't need the pullups.

Although the specification doesn't mention it, Schmitt-trigger inputs will give greater noise immunity. A Schmitt-trigger input has two switching thresholds: one that determines when the gate switches on a low to high transition, and a second, lower, threshold that determines when the input switches on a high to low transition.

For example, the output of a 74LS14 inverter won't go low until the input rises to at least 1.6V. After the output switches low, it won't go high again until the input drops to at least 0.8V. The 0.8V hysteresis, or difference between the two thresholds, means that the input will ignore noise or ringing of up to 0.8V. The hysteresis also prevents the output from oscillating when a slowly changing input reaches the switching threshold.

The inputs of the 74LS24X buffer/driver series have Schmitt-trigger inputs with 0.4V of hysteresis. However, inputs of the 74HC(T)24X equivalents are ordinary, non-Schmitt-trigger type. (But you may decide to use HCT inputs anyway, for lower power consumption or CMOS's greater noise immunity.

Level 2 devices

Level 2 devices have stronger drivers and inputs with hysteresis.

Drivers

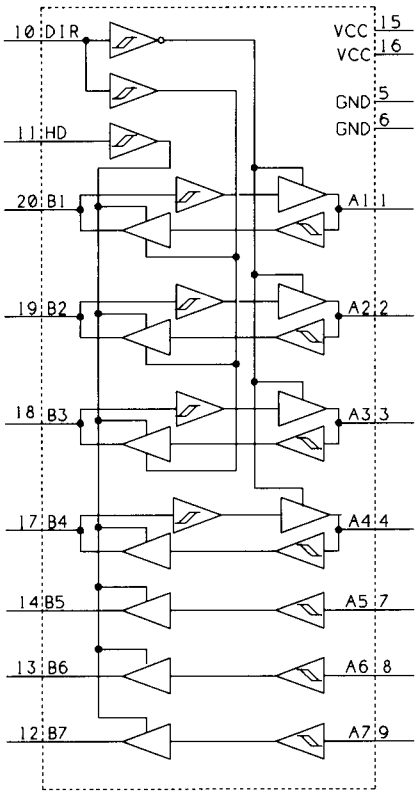
These are the characteristics of Level 2 drivers:

Logic-high outputs: +2.4V minimum at 12ma source current. This is much greater than Level 1's requirement of 0.32ma.

Logic-low outputs: +0.4V maximum at 12ma sink current. This is the same as the Level-1 specification.

Driver output impedance: 45-55 ohms at the measured (V_{OH} - V_{OL}).

Driver slew rate: 0.05 to 0.40 V/nsec.



DIR	HD	DIRECTION	OUTPUT TYPES
0	0	B1-B4 TO A1-A4	A1-A4, TOTEM POLE
		A5-A7 TO B5-B7	B5-B7, OPEN DRAIN
0	1	B1-B4 TO A1-A4	A1-A4, TOTEM POLE
		A5-A7 TO B5-B7	B5-B7, TOTEM POLE
1	0	A1-A7 TO B1-B7	B1-B7, OPEN DRAIN
1	1	A1-A7 TO B1-B7	B1-B7, TOTEM POLE

Figure 6-1: National's 74ACT1284 is a transceiver with seven lines that meet IEEE 1284's Level 2 interface standard.

Ordinary LSTTL drivers can't sink enough current to meet the specification. HC(T)MOS devices have equal source and sink currents, but aren't strong enough to meet the standard's minimum. The outputs of many of the new controller chips, including those from SMC and National, do meet the Level-2 requirements.

For simple parallel-port I/O with a Level-2 interface, you can use National's 74ACT1284 IEEE 1284 transceiver, which, as the name suggests, is designed specifically as a parallel-port interface. Figure 6-1 shows the chip and pinout. It includes four bidirectional lines and three one-way buffer/drivers. A Direction input (*DIR*) sets the direction of the bidirectional lines. A high-drive-enable input (*HD*) determines whether the B-side outputs are open-drain or push-pull type.

You can wire the 74ACT1284's in any of a number of ways, depending on your application. For example, using three chips, you could use eight bidirectional bits for the Data lines, four more for the Control lines, and use five of the remaining bits for Status inputs, with four bits left over. For bidirectional use, the Control outputs can emulate the original port's open-collector design. If you don't need

bidirectional Control lines, you can use two chips for the Data and Status bits and one Control bit, and use cheaper buffers for the remaining Control bits.

The 74ACT1284 is available in two surface-mount packages: an SOIC with 0.05" lead spacing, and a *very* tiny SSOP with 0.025" lead spacing.

Receivers

These are the characteristics of Level 2 receivers:

Logic-high input: 2.0V maximum at 20µa sink current. (Same voltage as Level 1 devices, but much lower current.)

Logic-low input: 0.8V minimum at 20µa source current. (Same voltage as Level 1 devices, but much lower current.)

Receiver hysteresis: 0.2V minimum. Greater hysteresis, up to 1.2V, will give greater noise immunity.

Again, many new parallel-port controller chips meet the Level-2 requirements for receivers.

For simple I/O applications, you can use 74HCT14 Schmitt-trigger inverters or 74HCT24X series buffer/drivers as receivers. LSTTL inputs draw too much current to meet the requirement. The inputs of the 74ACT1284 are also suitable as Level 2 inputs, with a minimum input hysteresis of 0.35V.

Interfacing Guidelines

When you're designing circuits that connect to the parallel port, following some guidelines will help to ensure that the link between the port and your device works reliably.

General Design

These are general guidelines for interfacing digital logic to a cable:

Use plenty of decoupling capacitors. Connect a capacitor from +5V to ground near each IC that connects to the cable. Use a type with good high-frequency response, such as ceramic, mica, or polystyrene. Keep the wires or traces between the capacitor's leads and the chip's +5V and ground pins as short as possible. A good, general-purpose value is 0.01µF, but the precise value isn't critical. Also connect a 10µF electrolytic capacitor from +5V to ground, near where the 5-volt supply enters the board.

The decoupling capacitors store energy needed by the logic gates as they switch. All logic gates draw current as they respond to changes at their inputs. When the current can be drawn from a nearby capacitor, the gate can switch quickly, without causing voltage spikes in the power-supply or ground lines. The capacitor should be near the chip it supplies, to minimize the inductance of the loop formed by the electrical path connecting the capacitor and the chip. Lower inductance means faster response.

The large electrolytic capacitor stores energy that the smaller capacitors can draw on to recharge.

Buffer all clock and control signals. Add buffers like those in Table 6-1 to help isolate clock and control signals from noise on the cable. Critical signals include inputs and outputs of flip-flops, counters, and shift registers. Some chips, like the 74LS374 octal flip-flop, have buffered outputs on-chip.

Use the slowest logic family possible. LSTTL and HCTMOS chips are fine for many links. Higher-speed logic can cause unwanted transmission-line effects (described below).

Don't leave CMOS inputs open. If you have unused inputs, tie them to +5V or ground. A floating CMOS input can cause the chip to draw large amounts of current. You can leave unused TTL inputs open, or pull them high with a 4.7K pullup resistor. Without the pullup, a TTL input will float at around 1.1 to 1.4V, which is usually treated as a logic high, though it's less than the 2V minimum specification for a logic high input. An open TTL input won't draw large currents like CMOS can, however.

Port Design

These guidelines apply specifically to PC parallel-port interfaces:

Status line cautions. If you're using DOS interrupts or other LPT functions to access the port, tie *S3* high and *S5* and $\overline{S7}$ low (unless you're using these bits for their intended purposes). The BIOS interrupt requires only $\overline{S7}$ to be low.

Control line cautions. Use the Control bits as inputs on the PC only on SPPs or ports that emulate the SPP. If you do use the Control lines as inputs, drive them with open-collector outputs. This will protect the port's circuits if a low Control-port output should connect to a high output. If you don't use open collector devices, place a 330-ohm resistor in series with each Control line.

Bidirectional data cautions. Use series resistors to protect the outputs when you use a bidirectional Data port for input. (Some controllers have current-limiting circuits that protect against damaging currents, but this isn't guaranteed.)

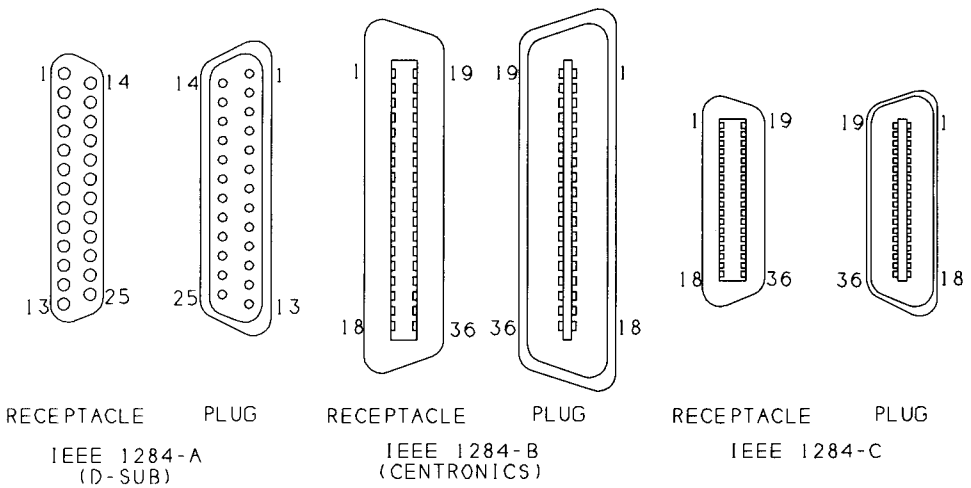


Figure 6-2: Parallel-port devices and cables may use any of these connector types.

Cable Choices

Parallel-port cables may vary in connector type, shielding, the arrangement of the wires in the cable, and the number of ground wires.

Connectors

The IEEE-1284 standard describes both the PC's D-sub connector and the Centronics connector found on many peripherals. It describes the conventional uses for the connectors—a female D-sub on the PC and female Centronics connector on the peripheral—but it doesn't recommend a particular connector for either device. The standard does recommend using connectors with metal shells for shielding continuity.

The standard calls the D-sub the 1284-A connector, and the Centronics connector, the 1284-B. The standard also introduces a new connector, the 1284-C. It's a 36-contact connector similar to the Centronics type, but more compact, with the contacts on 0.05" centers rather than 0.85". With this connector, the standard recommends using female (receptacle) connectors on both the host and peripheral, with male (plug) connectors on the cable. Table 6-2 shows the pin assignments for all of the connectors.

Figure 6-2 shows the pin numbering for the connectors. The pin numbers are labeled on most connectors, but the labeling typically consists of tiny,

Table 6-2: Pin assignments for D-sub, Centronics, and IEEE 1284C connectors.

Signal Name	Register bit	Signal Pin			Ground Return Pin		
		D-sub (IEEE 1284-A)	Centron-ics (IEEE 1284-B)	IEEE 1284-C	D-sub (IEEE 1284-A)	Centron-ics (IEEE 1284-B)	IEEE 1284-C
Data bit 0	*D0*	2	2	6	19	20	24
Data bit 1	*D1*	3	3	7	19	21	25
Data bit 2	*D2*	4	4	8	20	22	26
Data bit 3	*D3*	5	5	9	20	23	27
Data bit 4	*D4*	6	6	10	21	24	28
Data bit 5	*D5*	7	7	11	21	25	29
Data bit 6	*D6*	8	8	12	22	26	30
Data bit 7	*D7*	9	9	13	22	27	31
nError (nFault)	*S3*	15	32	4	23	29	22
Select	*S4*	13	13	2	24	28	20
PaperEnd	*S5*	12	12	5	24	28	23
nAck	*S6*	10	10	3	24	28	21
Busy	$\overline{S7}$	11	11	1	23	29	19
nStrobe	$\overline{C0}$	1	1	15	18	19	33
nAutoLF	$\overline{C1}$	14	14	17	25	30	35
nInit	*C2*	16	31	14	25	30	32
nSelectIn	$\overline{C3}$	17	36	16	25	30	34
HostLogicHigh				18			18
PeriphLogicHigh				36			36

hard-to-read numbers molded into the cable shell. Use bright light and a magnifier!

Cable Types

For a non-critical, low-speed link with a short cable, you can use just about any assortment of wires and connectors without problems. For example, if you're using the parallel port's inputs to read manual switches and using the outputs to

light LEDs, it doesn't really matter if the signals change slowly or have a few glitches as they switch.

At other times, especially at higher speeds and over longer cables, cable design may mean the difference between a link that works reliably and one that doesn't.

Some interfaces are designed to be able to carry signals over long cables. In an RS-232 serial link, the drivers use large voltage swings and limited slew rates (the rate at which the output switches) to help provide a good-quality signal at the receiver. The RS-485 serial interface use differential signals, where the transmitting end sends both the signal and its inverse and the receiving end detects the voltage difference between the two. An advantage to this type of transmission is that any noise common to both lines cancels out.

When you're using the PC's parallel port, you have to make do with many of the limits built into the design. IEEE 1284's Level 2 drivers and receivers are improved over the original design, but the improvement isn't dramatic because the Level-2 components are designed to be compatible with the original interface. You still can't use the parallel port for a 100-foot link. There are some things you can do to ensure reliable communications, however.

Ground Returns

Most importantly, even though you might get by with just 18 wires in a parallel-port cable, a full 25-wire cable is better, and a 36-wire twisted-pair cable is better still.

In all circuits, current must flow back to its source. In a cabled link, the ground wires provide the return path for the current. Although you may think of a ground wire as having no voltage, every wire has some impedance, and current in the wire induces a voltage. When multiple signals share a ground return, each of the inputs sees the ground voltages caused by all of the others.

In the original Centronics interface, most signals had their own ground returns, with the signal wire and its return forming a twisted pair in the cable. In a twisted pair, two wires spiral around each other, with a twist every inch or so.

The PC's D-sub connector has room for just eight ground contacts. The reduced number of grounds is a compromise caused by the decision to use a 25-contact connector on the PC, rather than Centronics' 36-contact connector. A few of the contacts are designated as ground returns for a particular signal, while others are the ground return for two signals. Some signals have no designated ground return at all.

If a peripheral uses a 36-contact connector, each of the shared ground wires in a 25-wire cable connects to two or three contacts. For example, the returns for

nStrobe and *D0* share a wire. Using 1284-C connectors allows the return 36 contacts on both ends.

In reality, ground currents will take the path of least resistance, and there's no way to guarantee that a current will flow in a particular wire. Multiple ground wires do lower the overall impedance of the ground returns, however, and this reduces ground currents.

If you eliminate seven of the ground wires and wire all of the ground contacts to a single wire, the interface will probably work, most of the time, especially at low speeds and over short distances. But a cable with at least 25 wires is preferable.

In a ribbon cable that connects to a dual header, the ground lines (18-25) alternate with signal lines, and this helps to reduce noise in the cable. Although ribbon cables usually aren't shielded, they're acceptable for low-speed, shorter links.

36-wire Cables

IEEE 1284 introduces a new cable for the parallel port. The cable contains 18 twisted pairs, with each signal line paired with its own ground return. Compared to the original parallel cable's 10-foot limit, the new cable may be as long as 10 meters, or 33 feet. A cable that meets the standard's requirements may be labeled *IEEE Std. 1284-1994 compliant.*

The 18th pair (at pins 18 and 36) has the only wires with new functions. The host and peripheral each use this pair to detect the presence of the other device. At the host, pin 18, *HostLogicHigh*, is a logic-high output, and pin 36 is an input with 7.5K impedance to ground. At the peripheral, pin 36, *PeripheralLogicHigh*, is a logic-high output and pin 18 is the 7.5K input. When there is no device connected, or when a device isn't powered, the inputs read logic low. With this arrangement, the host can read pin 36 and the peripheral can read pin 18 to detect whether or not the opposite device is present and powered.

If you use the new cable with 1284-C connectors, each contact connects to one wire, as Table 6-3 shows. You can also use this cable with 1284-A and -B connectors. In these cases, the ground returns for two or more signals connect to a single contact on the connector. (Even though the Centronics connector has 36 contacts, its conventional use doesn't include a ground return for every signal.) Table 6-4 shows the recommended wiring for a link with one D-sub and one Centronics connector. Other combinations of connectors can use similar wiring schemes, with each signal wire twisted with its ground wire.

Table 6-3: Wiring for a 36-wire, twisted-pair cable with two IEEE 1284-C connectors.

Cable Pair	Host		Peripheral	
	Signal	Pin	Pin	Signal
1	$\overline{S7}$ (Busy)	1	1	$\overline{S7}$ (Busy)
	Signal Ground ($\overline{S7}$)	19	19	Signal Ground ($\overline{S7}$)
2	*S4* (Select)	2	2	*S4* (Select)
	Signal Ground (*S4*)	20	20	Signal Ground (*S4*)
3	*S6* (nAck)	3	3	*S6* (nAck)
	Signal Ground (*S6*)	21	21	Signal Ground (*S6*)
4	*S3* (nError)	4	4	*S3* (nError)
	Signal Ground (*S3*)	22	22	Signal Ground (*S3*)
5	*S5* (PaperEnd)	5	5	*S5* (PaperEnd)
	Signal Ground (*S5*)	23	23	Signal Ground (*S5*)
6	Data Bit 0 (*D0*)	6	6	Data Bit 0 (*D0*)
	Signal Ground (*D0*)	24	24	Signal Ground (*D0*)
7	Data Bit 1 (*D1*)	7	7	Data Bit 1 (*D1*)
	Signal Ground (*D1*)	25	25	Signal Ground (*D1*)
8	Data Bit 2 (*D2*)	8	8	Data Bit 2 (*D2*)
	Signal Ground (*D2*)	26	26	Signal Ground (*D2*)
9	Data Bit 3 (*D3*)	9	9	Data Bit 3 (*D3*)
	Signal Ground (*D3*)	27	27	Signal Ground (*D3*)
10	Data Bit 4 (*D4*)	10	10	Data Bit 4 (*D4*)
	Signal Ground (*D4*)	28	28	Signal Ground (*D4*)
11	Data Bit 5 (*D5*)	11	11	Data Bit 5 (*D5*)
	Signal Ground (*D5*)	29	29	Signal Ground (*D5*)
12	Data Bit 6 (*D6*)	12	12	Data Bit 6 (*D6*)
	Signal Ground (*D6*)	30	30	Signal Ground (*D6*)
13	Data Bit 7 (*D7*)	13	13	Data Bit 7 (*D7*)
	Signal Ground (*D7*)	31	31	Signal Ground (*D7*)
14	*C2* (nInit)	14	14	*C2* (nInit)
	Signal Ground (*C2*)	32	32	Signal Ground (*C2*)
15	($\overline{C0}$) nStrobe	15	15	($\overline{C0}$) nStrobe
	Signal Ground ($\overline{C0}$)	33	33	Signal Ground ($\overline{C0}$)
16	$\overline{C3}$ (nSelectIn)	16	16	$\overline{C3}$ (nSelectIn)
	Signal Ground ($\overline{C3}$)	34	34	Signal Ground ($\overline{C3}$)
17	$\overline{C1}$ (nAutoFd)	17	17	$\overline{C1}$ (nAutoFd)
	Signal Ground ($\overline{C1}$)	35	35	Signal Ground ($\overline{C1}$)
18	Host Logic High	18	18	Host Logic High
	Peripheral Logic High	36	36	Peripheral Logic High
-	Shield			Shield

Table 6-4: Wiring for a 36-wire, twisted-pair cable with one 25-pin D-sub (IEEE 1284-A) and one Centronics (IEEE 1284-B) connector.

Cable Pair	Host (D-sub)		Peripheral (Centronics)	
	Signal	**Pin**	**Pin**	**Signal**
1	$\overline{S7}$ (Busy)	11	11	$\overline{S7}$ (Busy)
	Signal Ground ($\overline{S7}$, *S3*)	23	29	Signal Ground ($\overline{S7}$)
2	*S4* (Select)	13	13	*S4* (Select)
	Signal Ground (*S4*, *S5*, *S6*)	24	28	Signal Ground (*S4*)
3	*S6* (nAck)	10	10	*S6* (nAck)
	Signal Ground (*S4*, *S5*, *S6*)	24	28	Signal Ground (*S6*)
4	*S3* (nError)	15	32	*S3* (nError)
	Signal Ground (*S4*, *S5*, *S6*)	23	29	Signal Ground (*S3*)
5	*S5* (PaperEnd)	12	12	*S5* (PaperEnd)
	Signal Ground (S4, *S5*, S6)	24	28	Signal Ground (*S5*)
6	Data Bit 0 (*D0*)	2	2	Data Bit 0 (*D0*)
	Signal Ground (*D0*, *D1*)	19	20	Signal Ground (*D0*)
7	Data Bit 1 (*D1*)	3	3	Data Bit 1 (*D1*)
	Signal Ground (D0, *D1*)	19	21	Signal Ground (*D1*)
8	Data Bit 2 (*D2*)	4	4	Data Bit 2 (*D2*)
	Signal Ground (*D2*, *D3*)	20	22	Signal Ground (*D2*)
9	Data Bit 3 (*D3*)	5	5	Data Bit 3 (*D3*)
	Signal Ground (D2, *D3*)	20	23	Signal Ground (*D3*)
10	Data Bit 4 (*D4*)	6	6	Data Bit 4 (*D4*)
	Signal Ground (*D4*, *D5*)	21	24	Signal Ground (*D4*)
11	Data Bit 5 (*D5*)	7	7	Data Bit 5 (*D5*)
	Signal Ground (D4, *D5*)	21	25	Signal Ground (*D5*)
12	Data Bit 6 (*D6*)	8	8	Data Bit 6 (*D6*)
	Signal Ground (*D6*, *D7*)	22	26	Signal Ground (*D6*)
13	Data Bit 7 (*D7*)	9	9	Data Bit 7 (*D7*)
	Signal Ground (*D6*, *D7*)	22	27	Signal Ground (*D7*)
14	*C2* (nInit)	16	31	*C2* (nInit)
	Signal Ground (*C1*, $\overline{C2}$, *C3*)	25	30	Signal Ground (*C2*)
15	($\overline{C0}$) nStrobe	1	1	($\overline{C0}$) nStrobe
	Signal Ground ($\overline{C0}$)	18	19	Signal Ground ($\overline{C0}$)
16	$\overline{C3}$ (nSelectIn)	17	36	$\overline{C3}$ (nSelectIn)
	Signal Ground ($\overline{C1}$, *C2*, $\overline{C3}$)	25	30	Signal Ground ($\overline{C3}$)
17	$\overline{C1}$ (nAutoFd)	14	14	$\overline{C1}$ (nAutoFd)
	Signal Ground (*C1*, $\overline{C2}$, *C3*)	25	30	Signal Ground ($\overline{C1}$)
18	tied together, no connection at host		18	Host Logic High
			36	Peripheral Logic High
-	Shield			Shield

Reducing Interference

Interference occurs in a cabled link when signals couple from one wire into another, either within a cable or between a cable and a signal outside the cable. The coupling may be capacitive, inductive, or electromagnetic. Capacitive coupling occurs when an electric field, such as that generated by a voltage on a wire, interacts with an adjacent electric field. Inductive, or magnetic, coupling occurs when a magnetic field generated by a voltage on a wire interacts with an adjacent magnetic field. Electromagnetic coupling occurs when a wire acts as a transmitting or receiving antenna for signals that radiate through the air.

You can reduce interference by shielding, or blocking, signals from entering or leaving a wire, or by reducing the amplitude of the interfering signals.

Shielding

Metal shielding is an effective way to block noise due to capacitive, electromagnetic, and high-frequency magnetic coupling. A good parallel-port cable will have a metal shield surrounding the conductors and extending to the metal connectors.

The cable should have no large gaps where the conductors are unshielded. In particular, instead of a single wire, or "pigtail" connecting the shield to the connector, the full 360 degrees of the shield should contact the connector shell. The connectors in turn plug into the metal chassis of the PC or peripheral.

Solid shields provide the best protection, but they tend to be rigid and likely to break. Many cables instead use a more flexible braided shield made by interleaving bundles of thin metal strands into a shield that surrounds the wires. Although a braided shield doesn't cover the wires completely, it's durable, flexible, and effective enough, especially at higher frequencies.

IEEE-1284-compliant cables have two shielding layers. A solid aluminum or polyester foil surrounds the wires, and this is in turn surrounded by braided shield with 85% optical covering. The shield has a 360-degree connection to the connector's shell, which connects to the grounded chassis of both devices. The standard also recommends wire size of AWG 28 or lower. (Lower AWG numbers indicate larger wire diameters.)

Twisted Pairs

Using twisted pairs is another way to reduce interference in a cabled link. A twisted pair has two insulated wires that spiral around each other with a twist every inch or so. IEEE 1284 specifies a minimum of 36 twists per meter. The simple act of twisting results in benefits.

Twisting reduces magnetically coupled interference, especially from low-frequency signals such as 60-Hz power-line noise. Changing voltages on a wire cause the wire to emanate a magnetic field. The magnetic field in turn induces voltages on wires within the field.

The fields that emanate from a signal wire and its ground return have opposite polarities. Each twist causes the wires to physically swap positions, causing the pair's magnetic field to reverse polarity. The result is that the fields emanating from the wires tend to cancel each other out. In a similar way, the twisting reduces electro-magnetic radiation emitted by the pair.

Cable-buying Tips

Buying a cable labeled *IEEE-1284 compliant* is a simple way to guarantee good cable design. Other than this, there often is no easy way to tell how many wires are in a cable, or what type of shielding it has, if any, or whether the wires are in twisted pairs. The connectors are normally molded to the cable, so there's no way to peek inside without cutting the cable apart. Some catalogs do include specifications for the cables they offer. Whatever you do, don't mistakenly buy a 3-wire or 9-wire serial cable for parallel-port use. These cables may have 25-pin D-subs, but because serial links rarely use all 25 lines, they often have just three or nine wires.

Line Terminations

Another factor that affects signal quality in a link is the circuits that terminate the wires at the connector. To understand cable termination, you have to think of the cable as more than a simple series of connections between logic inputs and outputs.

Transmission Lines

When a long wire carries high-frequency signals, it has characteristics of a *transmission line,* defined as a circuit that transfers energy from a source to a load. Because the fast transitions of digital signals contain high-frequency components, most digital circuits are considered high frequency, even if the transmission rate (bits per second) is slow. To ensure reliable performance, transmission lines use line terminations, which are circuits at one or both ends that help ensure that the signals arrive in good shape at the receiver.

In many cases, especially when the cable is short and transmission speed is slow, an interface will work without any special attention to terminations. However, there are basic facts about transmission lines that are helpful when you're dealing with a cabled interface, especially if you need to stretch the limits.

At low speeds and over short distances, you can consider a short wire or PC-board trace to be a perfect connection: a logic high or low at one end of the wire or trace instantly results in a matching high or low at the opposite end. Most of the time, you don't have to concern yourself with delays, signal loss, noise, or other problems in getting a signal from an output to an input.

But when the connection is over a 10-foot or longer cable, and the signals are short pulses with fast rise and fall times, these factors can become important. Specifically, when a cable is physically long in relation to the highest wavelength it carries, it's considered to be a transmission line, which behaves differently than a cable that carries only low frequencies relative to its physical length. Transmission-line effects are significant when the wire length is greater than 1/10 to 1/20 of the wavelength of the highest frequency signal transmitted on the wire.

A 5-Megahertz sine wave has a wavelength of 60 meters, and 1/20 of that is 3 meters, or about 10 feet, which is the length of a typical parallel cable. From this, you might think that a parallel cable isn't a transmission line because the parallel port's maximum rate of transmitting is much less than 5 Mhz. But what's important isn't how often the voltages switch, but rather how quickly they switch.

This is because the frequencies that make up a digital waveform are much higher than the bits-per-second rate of the signal. Mathematically, a square wave (a waveform with equal, alternating high and low times) is the sum of a series of sine waves, including a fundamental frequency plus odd harmonics of that frequency. A 1000-Hz square wave actually consists of sine waves of 1000 Hz, 3000 Hz, 5000 Hz, and so on up.

A perfect square wave has an infinite number of harmonics and instant rise and fall times. Real-life components can pass limited frequencies, and their outputs require time to switch. A signal with fast rise and fall times will contain higher harmonics than a similar signal with slower rise and fall times. Parallel-port signals usually aren't square waves, but the principles apply generally to digital waveforms.

LSTTL and HCMOS logic are fast enough that transmission-line effects can be a factor on a parallel cable. Whether or not the effects will cause errors in an application depends in part on the bits-per-second rate of the transmitted signal and also on the hardware and software that detects and reads the signals. In a slow, short link that allows time between when an output switches and when the corresponding input is read, the software probably won't see any transmission-line effects, which occur mainly as the outputs switch. If you're pushing a link to its limit with either a long cable or high transmitting frequencies, you may have to consider the effects of the cable.

Characteristic Impedance

One way that a transmission line differs from other connections is that the transmission line has a *characteristic impedance.* Measuring the characteristic impedance of a wire involves more than a simple measurement with an ohmmeter. The characteristic impedance is a function of the wire's diameter, insulation type, and the distance between the wire and other wires in the cable.

It doesn't, however, change with the length of the wire. This seems to violate a fundamental rule of electronics, which says that a longer wire has greater resistance from end-to-end than a shorter one. But in most transmission lines, wire length isn't a major factor.

For the most efficient energy transfer from the source (output) to the load (input), the load's input impedance should match the characteristic impedance of the wire. When the impedances match, all of the energy is transferred from the source to the load and the logic level at the receiver matches the logic level at the driver.

If the impedances don't match, some of the energy reflects back to the source, which sees the reflection as a voltage spike. The reflections may bounce back and forth between the source and load several times before dying out. If the receiver reads the input before the reflections die out, it may not read the correct logic level, and in extreme cases, high-voltage reflections can damage the components.

If you're designing an interface from the ground up, you can specify terminations to match your design. But with the parallel port, things aren't as straightforward, because the driver and receiver components can vary. The wrong termination can cause reflected signals and errors in reading the inputs, or it may just slow the signal transitions and reduce the port's maximum speed.

Cable manufacturers often specify the characteristic impedance of their products. Typical values for twisted-pair and ribbon cable are around 100 to 120 ohms.

Example Terminations

A line termination may be located at the output, or source, or at the input, or receiver. In a bidirectional link, each end may have both a source and receiver termination.

Figure 6-3A shows a termination used on some ports. A series resistor at the driver and a high-impedance receiver cause an impedance mismatch that, amazingly, results in a received voltage that equals the transmitted voltage. The series resistor should equal the cable's characteristic impedance, minus the output impedance of the driver. Many parallel ports use series resistors of 22 to 33 ohms. You can add similar resistors in series with outputs that you use to drive the Status or Control inputs on a PC's port.

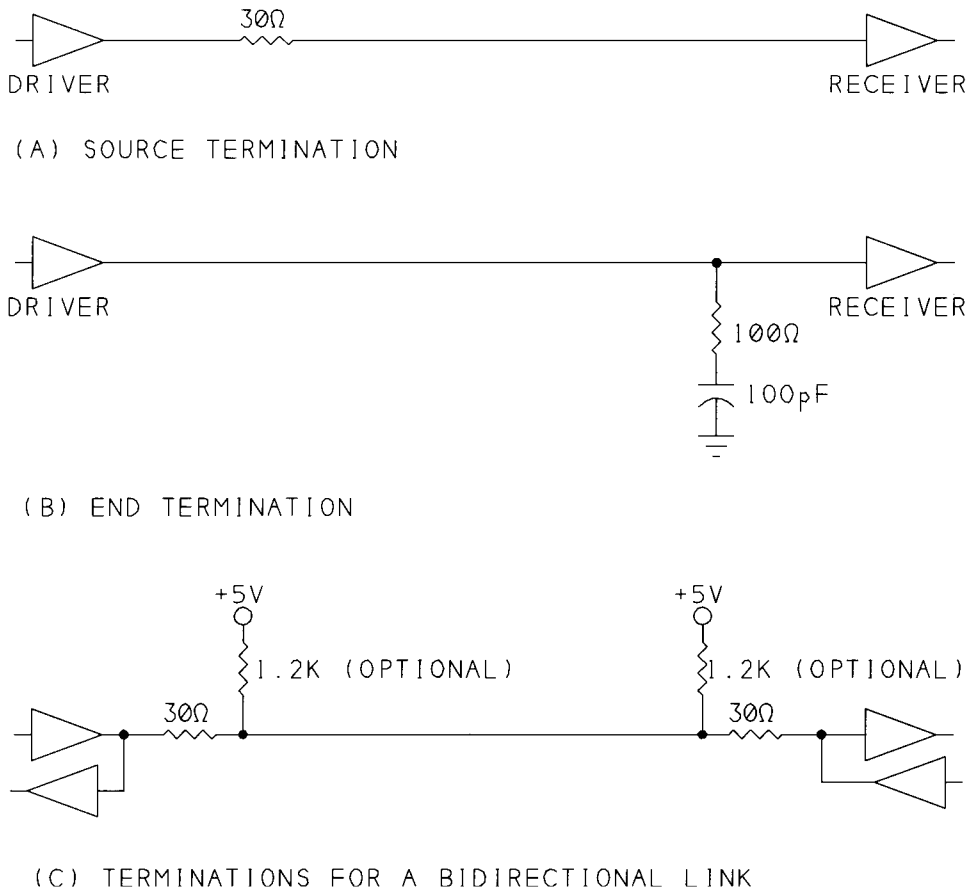


Figure 6-3: Line terminations for parallel-port cables.

When the driver switches, half of the output voltage drops across the combination of the series resistor and the driver's output impedance, and the other half reaches the receiver's input. Losing half of the output voltage doesn't sound like a good situation, but in fact, the mismatch has a desirable effect.

On a transmission line, when a signal arrives at a high-impedance input, a voltage equal to the received signal reflects back onto the cable. The reflection plus the original received voltage result in a signal equal to the original voltage, and this combined voltage is what the receiver sees. The reflected voltage travels back to the source and drops across the source impedance, which absorbs the entire reflected signal and prevents further reflections.

The impedance match doesn't have to be perfect, which is a good thing because it's unlikely that it will be. The driver's output impedance varies depending on the output voltage and temperature, so an exact match is impossible. If the impedance

at the source doesn't exactly match the cable's impedance, the signal at the receiver won't exactly match the original, and small reflections may continue before dying out. In general, an output impedance slightly smaller than the cable impedance is better than one that is slightly larger.

Figure 6-3B shows another option, an end termination at the receiver, consisting of a resistor and capacitor in series between the signal wire and ground. The resistor equals the characteristic impedance of the wire, and the capacitor presents a low impedance as the output switches. Unlike some other input terminations, this one is usable in both TTL and CMOS circuits. However, this type of termination doesn't work well with a series termination at the driver, because the series termination is designed to work with a high-impedance input. Because many parallel-port outputs have series terminations built-in, it's best not to use this end termination unless you're designing for a specific port that you know can use it effectively.

Figure Figure 6-3C shows IEEE 1284's recommended terminations for a Level-2 bidirectional interface. The standard specifies a characteristic cable impedance of 62 ohms, and assumes that each signal line will be in a twisted pair with its ground return. The outputs have series resistor terminations. If the inputs have pull-ups, they should be on the cable side of the source termination.

Transmitting over Long Distances

If the parallel port's 10 to 15-foot limit isn't long enough for what you want to do, there are options for extending the cable length.

If the interface isn't a critical one, and especially at slower speeds, you can just try a longer cable and see if it works. You may be able to stretch the interface without problems. But this approach is only recommended for casual, personal use, where you can take responsibility for dealing with any problems that occur.

A shielded, 36-wire, twisted-pair cable allows longer links than other cables. If you know that both the port and the device that connects to it have Level 2 interfaces, this type of cable should go 30 feet without problems.

Parallel-port extenders are also available from many sources. One type adds a line booster, or repeater, that regenerates the signals in the middle of the cable, allowing double the cable length. Other extenders work over much longer distances by converting the parallel signals into a serial format, usually RS-232, RS-422, or RS-485.

The serial links use large voltage swings, controlled slew rates, differential signals, and other techniques for reliable transmission over longer distances. You

could do the same for each of the lines in a parallel link, but as the distance increases, it makes sense to convert to serial and save money on cabling.

One drawback to the parallel-to-serial converters is that most are one way only, and don't include the parallel port's Status and Control signals. You can use the converters for simple PC-to-peripheral transfers, but not for bidirectional links. Also, serial links can be slow. After adding a stop and start bit for each byte, a 9600-bits-per-second link transmits just 960 data bytes per second.

If you need a long cable, instead of using a serial converter, you might consider designing your circuit to use a serial interface directly.

Port-powered Circuits

Most devices that connect to the parallel port will require their own power supply, either battery cells or a supply that converts line voltage to logic voltages. But some very low-power circuits can draw all the power they need from the port itself.

When to Use Port Power

The parallel-port connector doesn't have a pin that connects to the PC's +5V supply, so you can't tap directly into the supply from the connector. But if your device requires no more than a few milliamperes, and if one or more of the Data outputs is otherwise unused, you may be able to use the port as a power source.

As a rule, CMOS is a good choice for low-power circuits. CMOS components require virtually no power when the outputs aren't switching, and they usually use less power overall than TTL or NMOS.

Powering external circuits is especially easy if the circuits can run on +3V or less. Some components aren't particular about supply voltage. HCMOS logic can use any supply from +2V to +6V, with the logic high and low levels defined in proportion to the supply voltage. (*Minimum logic high input = 0.7(supply voltage); maximum logic low input = 0.3(supply voltage).*) National's LP324 quad op amp draws under 250µa of supply current and can use a single power supply as low as +3V. If you need +5V, there are new, efficient step-up regulator chips that can convert a lower voltage to a regulated +5V.

The parallel port's inputs require TTL logic levels, so any logic-high outputs that connect to the parallel-port inputs should be at least 2.4V. (Status-port inputs may have pullups to +5V, but this isn't guaranteed.)

The source for port power is usually one or more of the Data pins. If you bring a Data output high by writing 1 to it, you can use it as a power source for other circuits. The available current is small, and as the current increases, the voltage drops, but it's enough for some designs.

Of course, if you're using a Data pin as a power supply, you can't use it as a data output, so any design that requires all eight Data lines is out. One type of component that's especially suited to using parallel-port power is anything that uses a synchronous serial interface, such as the DS1620 digital thermometer described in Chapter 9. These require as few as one signal line and a clock line, leaving plenty of bits for other uses.

Abilities and Limits

One problem with using parallel-port power is that the outputs have no specification that every port adheres to. If you're designing something to work on a particular computer, you can experiment to find out if the outputs are strong enough to power your device. If you want the device to work on any (or almost all) computers, you need to make some assumptions. One approach is to assume that the current-sourcing abilities of a port's outputs are equal to those of the original port. Most ports do in fact meet this test, and many newer ports have the more powerful Level 2 outputs. It's a good idea to also include the option to run on an external supply, which may be as simple as a couple of AA cells, in case there is a port that isn't capable of powering your device.

On the original port, the eight Data outputs were driven by the outputs of a 74LS374 octal flip-flop. If you design for the '374's typical or guaranteed source current, your device should work on just about all ports. Typical output current for a 74LS374 is 2.6 milliamperes at 3.1V (2.4V guaranteed). A logic-low output of a '374 can sink much more than this, but a low output doesn't provide the voltage that the external circuits need.

Level 2 outputs can source 12 milliamperes at 2.5V. If you know that your port has Level 2 outputs, you have more options for using parallel-port power.

What about using the Control outputs as a power source? On the original port, these were driven by 7405 inverters with 4.7K pullups. The pull-ups on the outputs make it easy to calculate how much current they can source, because the output is just a 4.7K resistor connected to +5V. These outputs can source a maximum of 0.5 milliampere at 2.5V, so the Data outputs are a much better choice as current sources. On some of the newer ports, in the advanced modes, the Control outputs switch to push-pull type and can source as much current as the Data outputs.

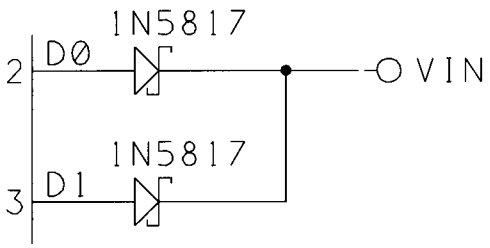


Figure 6-4: You can use spare Data outputs as a power source for very low-power devices. If you use more than one output, add a Schottky diode in series with each line.

Using Control bits as supplies is an option for these ports, but it isn't practical for a general-purpose circuit intended for any port.

I ran some informal tests on a variety of parallel ports, and found widely varying results, as Table 6-5 shows. The port with 74LS374 outputs actually sourced much more current than the specification guarantees, about the same as the Level 2 outputs on an SMC Super I/O controller. A port on an older monochrome video card had the strongest outputs by far, while a port on a multifunction board was the weakest, though its performance still exceeded the '374's specification.

Examples

If the exact supply voltage isn't critical, you can use one or more Data outputs directly as power supplies. If you use two or more outputs, add a Schottky diode in each line to protect the outputs, as Figure 6-4 shows. The diodes prevent current from flowing back into an output if one output is at a higher voltage. Schottky diodes drop just 0.3V, compared to 0.7V for ordinary silicon signal diodes.

How much output current is a safe amount? Again, because the components used in ports vary, there is no single specification. Also, because a power supply isn't the conventional use for a logic output, data sheets often don't include specifications like maximum power dissipation.

The safest approach is to draw no more than 2.6 milliamperes from each output, unless you know the chip is capable of safely sourcing higher amounts. At higher currents, the amount of power that the driver chips have to dissipate increases, and you run the risk of damaging the drivers.

If you need a regulated supply or a higher voltage than the port can provide directly, a switching regulator is a very efficient way to convert a low voltage to a steady, regulated higher (or lower) value. For loads of a few milliamperes,

Table 6-5: Results of informal tests of current-sourcing ability of the Data outputs on assorted parallel ports.

Card	No Load Voltage	Source Current at Data output (milliamperes)		
		4V	3V	2V
Original-type, LS374 outputs	3.5	-	11	25
Monochrome video card, single-chip design	4.9	18	35	35
Older multifunction card, with IDE and floppy controller	4.9	2.7	5	7
SMC Super I/O controller	4.9	0.6	7.5	27

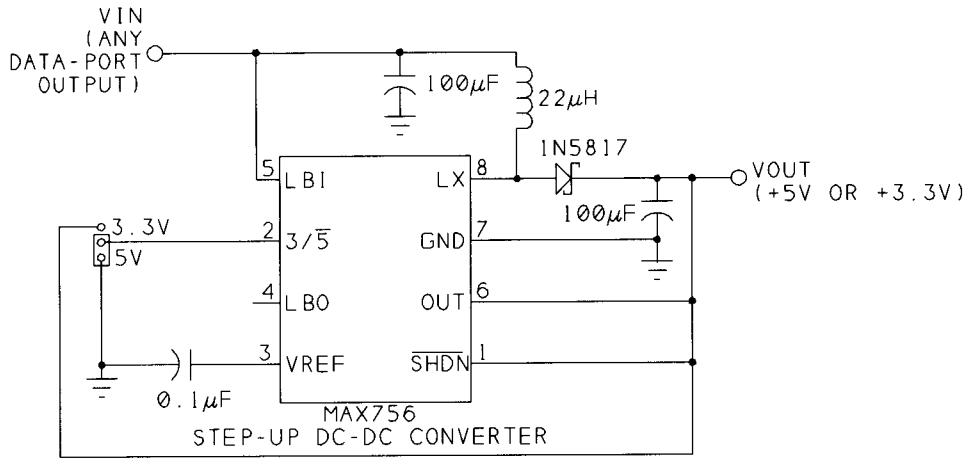


Figure 6-5: Maxim's Max756 can convert a Data output to a regulated +5V or +3.3V supply.

Maxim's MAX756 step-up converter can convert +2.5V to +5V with over 80% efficiency. Figure 6-5 shows a supply based on this chip.

As an example, assume that you want to power a circuit that requires 2 milliamperes at +5V, and assume that the parallel port's Data outputs can provide 2.6 milliamperes at 2.1V (2.4V minus a 0.3V drop for the diodes). This formula calculates how much current each Data pin can provide:

*(load supply (V)) * (output current (A)) =*

*converter efficiency * (source voltage (V)) * (source current (A))*

which translates to:

*5 * (output current) = 0.8 * 2.1 * (0.0026)*

and this shows that each Data pin can provide just under 0.9 milliampere at +5V. Three Data outputs could provide the required total of 2 milliamperes, with some

to spare. In fact there is a good margin of error in the calculations, and you could probably get by with two or even one output. If the port has Level 2 outputs, each pin can source 4 milliamperes, so all you need is one pin. You can do similar calculations for other loads.

The '756 has two output options: 5V and 3.3V. The '757 has an adjustable output, from 2.7V to 5.5V.

The selection of the switching capacitor and inductor is critical for the MAX756 and similar devices. The inductor should have low DC resistance, and the capacitor should be a type with low ESR (effective series resistance). Maxim's data sheet lists sources for suitable components, and Digi-Key offers similar components. Because of the '756's high switching speed, Maxim recommends using a PC board with a ground plane and traces as short as possible.

If you just need one supply, Maxim sells an evaluation kit that's a simple, no-hassle way of getting one up and running. The kit consists of data sheets and a printed-circuit board with all of the components installed.

7

Output Applications

One category of use for the parallel port is control applications, where the computer acts as a smart controller that decides when to switch power to external circuits, or decides when and how to switch the paths of low-level analog or digital signals. This chapter shows examples of these, plus a port-expansion circuit that increases the number of outputs that the port controls.

Output Expansion

The parallel port has twelve outputs, including the eight Data bits and four Control bits. If these aren't enough, you can add more by dividing the outputs into groups and using one or more bits to select a group to write to.

Figure 7-1 shows how to control up to 64 TTL- or CMOS-compatible outputs, a byte at a time.

U1 and U4 buffer *D0-D7* and $\overline{C0}$-$\overline{C3}$ from the parallel port. Four bits on U4 are unused.

U5 is a 74HCT138 3-to-8-line decoder that selects the byte to control. When U5 is enabled by bringing *G1* high and $\overline{G2A}$ and $\overline{G2B}$ low, one of its *Y* outputs is low. Inputs *A, B,* and *C* determine which output this is. When *CBA = 000, Y0* is low; when *CBA = 001, Y1* is low; and so on, with each value at *CBA* corresponding to

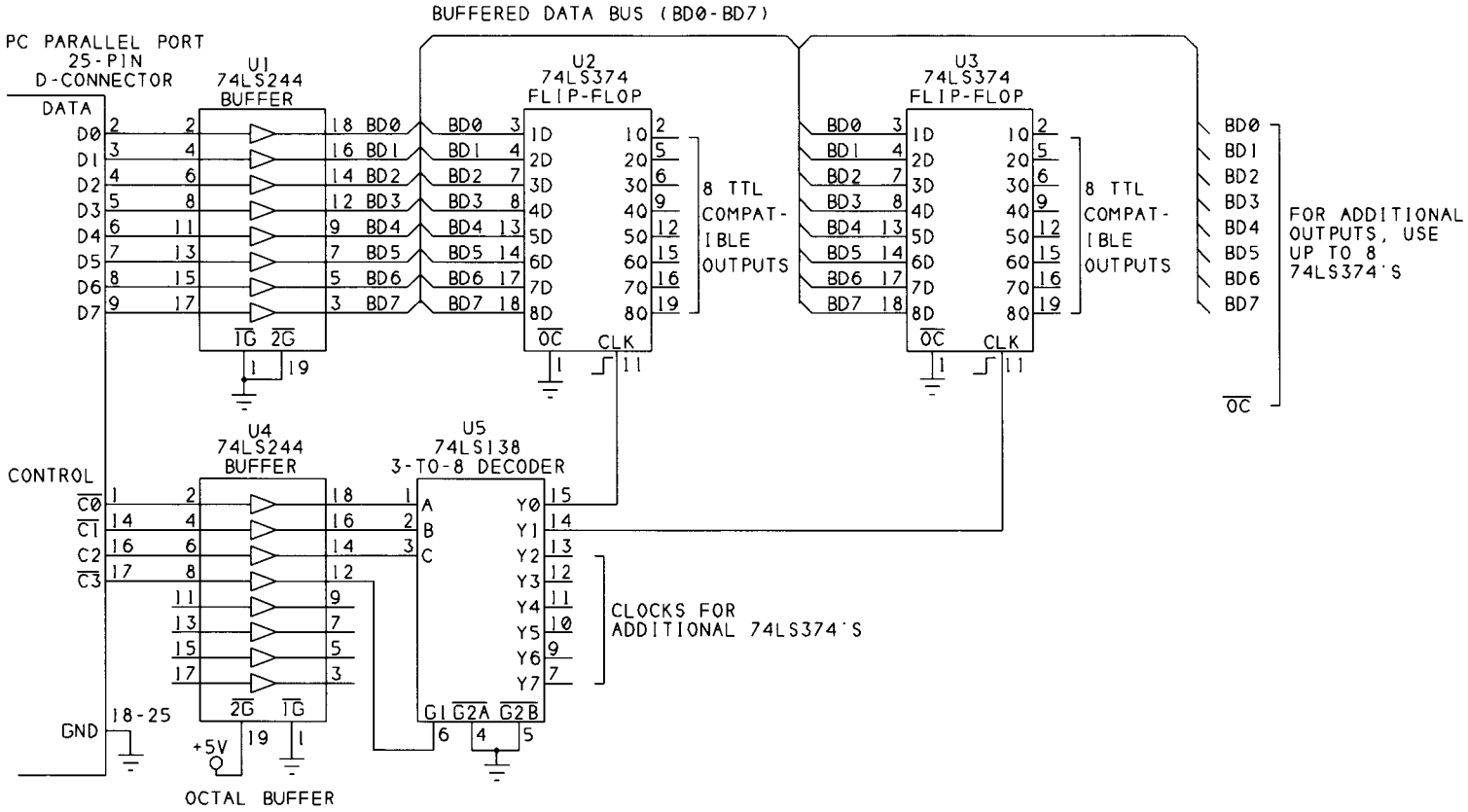


Figure 7-1: The eight data lines on the parallel port can control 64 latched outputs. The four control lines select a byte t
write to.

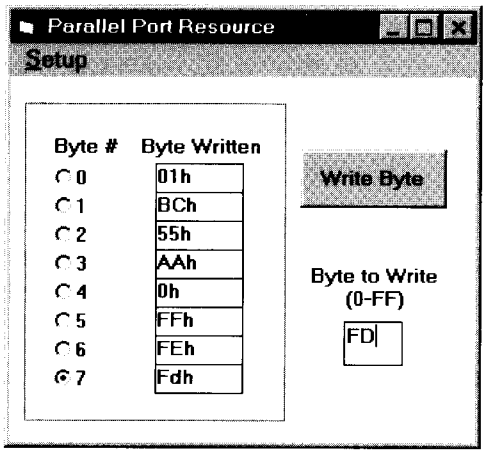


Figure 7-2: User screen for Listing 7-1's program code.

a low *Y* output. At the parallel port, bits $\overline{C0}$-*C2* determine the values at *A*, *B*, and *C*. If *G1* is low or either $\overline{G2A}$ or $\overline{G2B}$ is high, all of the *Y* outputs are high.

U2 is a 74HCT374 octal flip-flop that latches *D0-D7* to its outputs. The Output Control input ($\overline{OC}$, pin 1) is tied low, so the outputs are always enabled. A rising edge at *Clk* (pin 11) writes the eight *D* inputs to the *Q* outputs.

U3 is a second octal flip-flop, wired like U2, but with a different clock input. You may have up to eight 74HCT374s, each controlled by a different *Y* output of U5.

To write a byte, do the following:

1. Write the data to *D0-D7*.
2. Bring $\overline{C3}$ high and write the address of the desired '374 to $\overline{C0}$, $\overline{C1}$, and *C2* to bring a *Clk* input low.
3. Bring $\overline{C3}$ low, which brings all *Clk* inputs high and latches the data to the selected outputs. You can write just one byte at a time, but the values previously written to other '374's will remain until you reselect the chip and clock new data to it.

Listing 7-1 contains program routines for writing to the outputs. Figure 7-2 shows the form for a test program for the circuit. These demonstrate the circuit's operation by enabling you to select a latch, specify the data to write, and write the data.

You can use HCT-family or LSTTL chips in the circuit. If you can get by with 56 or fewer outputs, you can free up $\overline{C3}$ for another use, and bring *Y0-Y6* high by selecting *Y7*. One possible use for $\overline{C3}$ would be to enable and disable the '374's outputs by tying it to pin 1 of each chip.

```
Sub cmdWriteByte_Click ()
'Write the value in the "Byte to Write" text box
'to the selected output (1-8).
DataPortWrite BaseAddress, CInt("&h" & txtByteToWrite.Text)
'Select an output by writing its number to
'Control Port, bits 0-2, with bit 3 = 1.
'This brings the output's CLK input low.
'Then set Control bit 3 = 0 to bring all CLK inputs high.
'This latches the value at the data port to the selected output.
ControlPortWrite BaseAddress, ByteNumber + 8
ControlPortWrite BaseAddress, 0
'Display the result.
lblByte(ByteNumber).Caption = ""
lblByte(ByteNumber).Caption = txtByteToWrite.Text & "h"
End Sub
PortType = Left$(ReturnBuffer, NumberOfCharacters)
```

```
Sub optByte_Click (Index As Integer)
ByteNumber = Index
End Sub
```

Listing 7-1: To write to Figure 7-1's bytes, you write a value to the data port, then latch the value to the selected output byte.

Switching Power to a Load

The parallel port's Data and Control outputs can control switches that in turn control power to many types of circuits. The circuits may be powered by a +5V or +12V supply, another DC voltage or voltages, or AC line voltage (115V). In a simple power-control switch, bringing an output high or low switches the power on or off. To decide when to switch a circuit on or off, a program might use sensor readings, time or calendar information, user input, or other information.

Power-switching circuits require an interface between the parallel port's outputs and the switch that you want to control. In an electromagnetic, or mechanical, relay, applying a voltage to a coil causes a pair of contacts to physically separate or touch. Other switches have no moving parts, and operate by opening and closing a current path in a semiconductor.

Choosing a Switch

All switches contain one or more pairs of switch terminals, which may be mechanical contacts or leads on a semiconductor or integrated circuit. In addition,

electronically controlled switches have a pair of control terminals that enable opening and closing of the switch, usually by applying and removing a voltage across the terminals.

An ideal switch has three characteristics. When the switch is open, the switch terminals are completely disconnected from each other, with infinite impedance between them. When the switch is closed, the terminals connect perfectly, with zero impedance between them. And in response to a control signal, the switch opens or closes instantly and perfectly, with no delay or contact bounce.

As you might suspect, although there are many types of switches, none meets the ideal, so you need to find a match between the requirements of your circuit and what's available. Switch specifications include these:

Control voltage and current. The switch's control terminals have defined voltages and currents at which the switch opens and closes. Your circuit's control signal must meet the switch's specification.

Load current. The switch should be able to safely carry currents greater than the maximum current your load will require.

Switching voltage. The voltage to be switched must be less than the maximum safe voltage across the switch terminals.

Switching speed. For simple power switches, speed is often not critical, but there are applications where speed matters. For example, a switching power supply may switch current to an inductor at rates of 20 kilohertz or more. You can calculate the maximum switching speed from the switch's turn-on and turn-off times. (*Maximum switching speed = 1/(max. turn-on time + max. turn-off time.*)

Other factors to consider are cost, physical size, and availability.

Figure 7-3 shows some common configurations available in mechanical switches. Electronic switches can emulate these same configurations. You can also build the more complex configurations from combinations of simpler switches.

As the name suggests, a normally open switch is open when there is no control voltage, and closes on applying a control voltage. A normally closed switch is the reverse—it's closed with no control voltage, and opens on applying a voltage.

A single-throw (*ST*) switch connects a switch terminal either to a second terminal or to nothing, while a double-throw (*DT*) switch connects a switch terminal to either of two terminals. In a single-pole (*SP*) switch, the control voltage controls

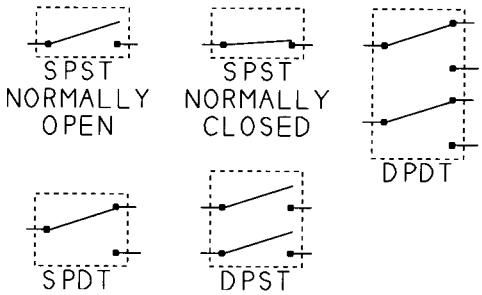


Figure 7-3: Five types of switches.

one set of terminals, while in a double-pole (*DP*) switch, one voltage controls two sets of terminals. A double-pole, double-throw (*DPDT*) switch has two terminals, with each switching between another pair of terminals (so there are six terminals in all).

Logic Outputs

For a low-current, low-voltage load, you may be able to use a logic-gate output or an output port bit as a switch. For higher currents or voltages, you can use a logic output to drive a transistor that will in turn control current to the load. In either case, you need to know the characteristics of the logic output, so you can judge whether it's capable of the job at hand.

Table 7-1 shows maximum output voltages and currents for popular logic gates, drivers, and microcontrollers, any of which might be controlled, directly or indirectly, by a PC's parallel port. The table shows minimum guaranteed output currents at specific voltages, usually the minimum logic-high and the maximum logic-low outputs for the logic family.

To use a logic output to drive a load other than a logic input, you need to know the output's maximum source and sink current and the power-dissipation limits of the chip. Many logic outputs can drive low-voltage loads of 10 to 20 milliamperes. For example, an LED requires just 1.4V. Because you're not driving a logic input, you don't have to worry about valid logic levels. All that matters is being able to provide the voltage and current required by the LED.

Figure 7-4 illustrates source and sink current. You might naturally think of a logic output as something that "outputs," or sends out, current, but in fact, the direction of current flow depends on whether the output is a logic-high or logic-low.

You can think of source current as flowing from a logic-high output, through a load to ground, while sink current flows from the power supply, through a load, into a logic-low output. Data sheets often use negative numbers to indicate source

current. In most logic circuits, an output's load is a logic input, but the load can be any circuit that connects to the output.

CMOS logic outputs are symmetrical, with equal current-sourcing and sinking abilities. In contrast, TTL and NMOS outputs can sink much more than they can source. If you want to use a TTL or NMOS output to power a load, design your circuit so that a logic-low output turns on the load.

All circuits should be sure to stay well below the chip's absolute maximum ratings. For example, an ordinary 74HC gate has an absolute maximum output of 25 milliamperes per pin, so you could use an output to drive an LED at 15 milliamperes. (Use a current-limiting resistor of 220 ohms.) If you want 20 milliamperes, a better choice would be a buffer like the 74HC244, with an absolute maximum output of 35 milliamperes per pin. In Figure 7-5, A and B show examples.

Don't try to drive a high-current load directly from a parallel-port output. Use buffers between the cable and your circuits. Because the original parallel port had no published specification, it's hard to make assumptions about the characteristics of a parallel-port output, except that it should be equivalent to the components in the original PC's port. Using a buffer at the far end of the cable gives you known output characteristics. The buffer also provides some isolation from the load-control circuits, so if something goes wrong, you'll destroy a low-cost buffer rather than your parallel port components. A buffer with a Schmitt-trigger input will help to ensure a clean control signal at the switch.

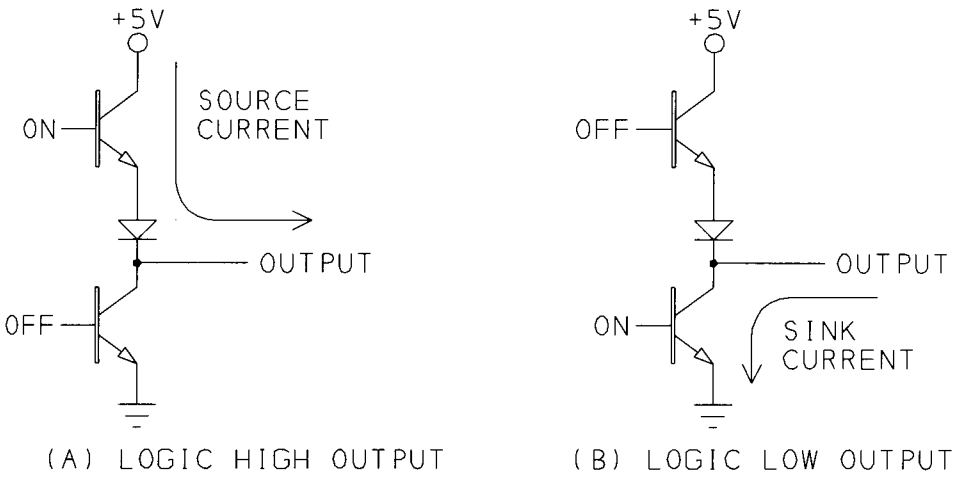


Figure 7-4: A logic-high output sources current; a logic-low output sinks current.

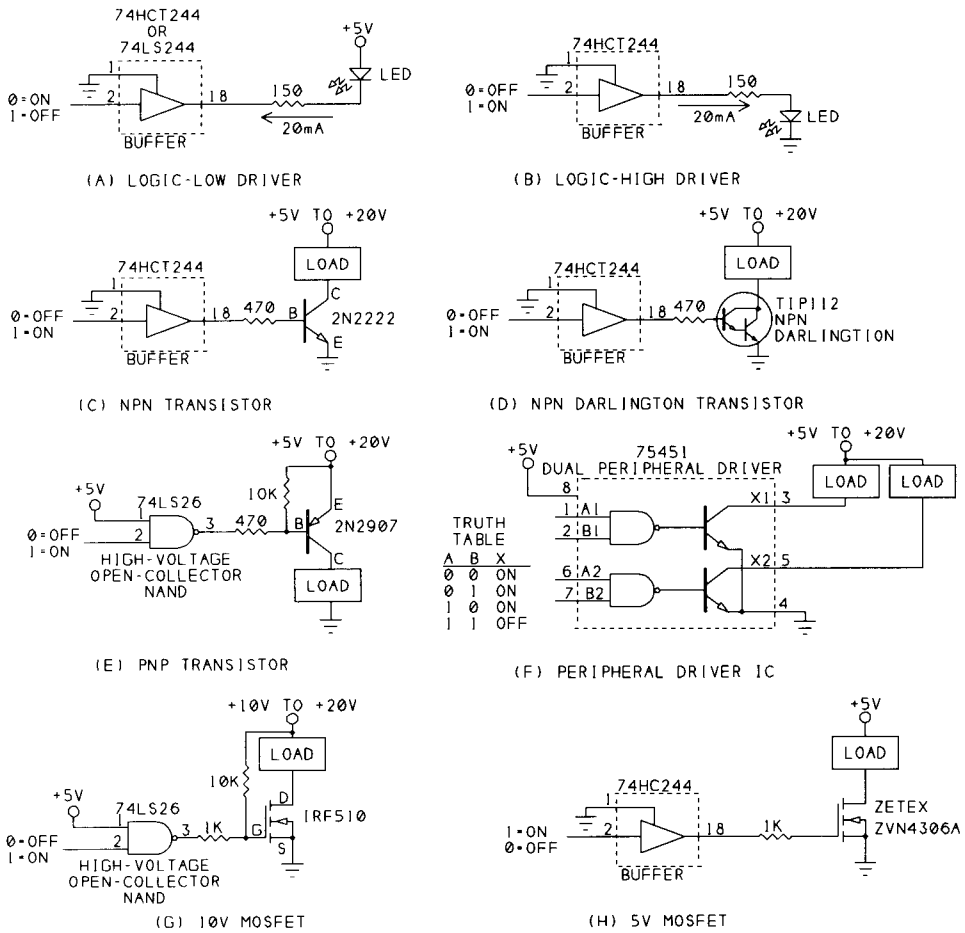


Figure 7-5: Interfaces to high-current and high-voltage circuits.

Bipolar Transistors

If your load needs more current or voltage than a logic output can provide, you can use an output to drive a simple transistor switch.

A bipolar transistor is an inexpensive, easy-to-use current amplifier. Although the variety of transistors can be bewildering, for many applications you can use any

Table 7-1: Maximum output current for selected chips.

Chip	Output high voltage (VOH min)	Output low voltage (VOL max)	Supply Voltage	Absolute maximums
74LS374 flip-flop, 74LS244 buffer	2.4V@-2.6mA	0.5V@24mA	4.5 to 5.5	-
74HC(T)374 flip-flop, 74HC(T)244 buffer	Vcc-0.1@-20µA 3.84V@-6mA	0.1V@20 µA 0.33V@6mA	4.5	35mA/pin, 500mW/package
74LS14 inverter	2.7V@-0.4mA	0.5V@8mA	4.5 to 5.5	-
74HC(T)14 inverter	4.4V@-20µA 4.2V@-4mA	0.1V@20µA 0.33V@4mA	4.5	25mA/pin, 500mW/package
8255 NMOS PPI (programmable peripheral interface)	2.4V@-200µA	0.45V@1.7mA (on any 8 Port B or C pins)	4.5 to 5.5	4mA/pin
82C55 CMOS PPI (programmable peripheral interface)	3V@-2.5mA	0.4V@2.5mA	4.5 to 5.5	4.0mA/pin
8051 NMOS microcontroller	2.4V@-80µA	0.45V@1.6µA	4.5 to 5.5	-
80C51 CMOS microcontroller	Vcc-0.3@-10µA Vcc-0.7@-30µA Vcc-1.5@-60µA	0.3@100µA 0.45@1.5mA 1.0@3.5mA	4 to 5	10mA/pin, 15mA/port, 71mA/all ports
68HC11 CMOS microcontroller	Vdd-0.8@-0.8mA	0.4@1.6mA	4.5 to 5.5	25mA/pin; also observe power dissipation limit for the chip
PIC16C5x CMOS microcontroller	Vdd-0.7@-5.4mA	0.6@8.7mA	4.5	+25/-20mA/pin, +50/-40mA/port, 800mW/package

general-purpose or saturated-switch transistor that meets your voltage and current requirements.

Figure 7-5C uses a 2N2222, a widely available NPN transistor. A logic-high at the control output biases the transistor on and causes a small current to flow from base to emitter. This results in a low collector-to-emitter resistance that allows current to flow from the power supply, through the load and switch, to ground. When the transistor is switched on, there is a small voltage drop, about 0.3V, from collector to emitter, so the entire power-supply voltage isn't applied across the load.

The exact value of the transistor's base resistor isn't critical. Values from a few hundred to 1000 ohms are typical. The resistor needs to be small enough so that the transistor can provide the current to power the load, yet large enough to limit the current to safe levels.

The load current must be less than the transistor's maximum collector current (I_C). Look for a current gain (h_{FE}) of at least 50. Many parts catalogs include these specifications.

The load's power supply can be greater than +5V, but if it's more than +12V, check the transistor's collector-emitter breakdown voltage (V_{CEO}), to be sure it's greater than the voltage that will be across these terminals when the switch is off.

For large load currents, you can use a Darlington pair, as Figure 7-5D shows. One transistor provides the base current to drive a second transistor. Because the total current gain equals the gain of the first times the gain of the second, gains of 1000 are typical. The TIP112 is an example of a Darlington pair in a single TO-220 package. It's rated for collector current of 2 amperes and collector-to-emitter voltage of 100V. A drawback is that the collector-to-emitter voltage of a Darlington is about a volt, much higher than for a single transistor.

The above circuits all use NPN transistors and require current from a logic-high output to switch on. If you want to turn on a load with a logic-low output, you can use a PNP transistor, as Figure 7-5E shows. In this circuit, a logic-low output biases the transistor on, and a voltage equal to the power supply switches it off. If the load's power supply is greater than +5V, use a high-voltage open-collector or open-drain output for the control signal, so that the pullup resistor can safely pull logic-high outputs to the supply voltage.

Another handy way to control a load with logic is to use a peripheral-driver chip like those in the 7545X series (Figure 7-5F). Each chip in the series contains two independent logic gates, with the output of each gate controlling a transistor switch.

There are four members of the series:

75451 dual AND drivers
75452 dual NAND drivers
75453 dual OR drivers
75454 dual NOR drivers

Each output can sink a minimum of 300 milliamperes at 0.7V (collector-to-emitter voltage).

MOSFETs

An alternative to the bipolar transistor is the MOSFET. The most popular type is an enhancement-mode, N-channel type, where applying a positive voltage to the gate switches the MOSFET on, creating a low-resistance channel from drain to source.

P-channel MOSFETs are the complement of N-channel MOSFETs, much as PNP transistors complement NPNs. An enhancement-mode, P-channel MOSFET switches on when the gate is more negative than the source. In depletion-mode MOSFETs (which may be N-channel or P-channel), applying a gate voltage opens the switch, rather than closing it.

Unlike a bipolar-transistor switch, which can draw several milliamperes of base current, a MOSFET gate has very high input resistance and draws virtually no current. But unlike a bipolar transistor, which needs just 0.7V from base to emitter, a MOSFET may require as much as 10V from gate to source to switch on fully.

One way to provide the gate voltage from 5V logic is to use a device with an open-collector or open-drain output and a pull-up resistor to at least 10V, as Figure 7-5G shows. Some newer MOSFETs have lower minimum *on voltages.* Zetex's ZVN4603A can switch 1.5 amperes with just +5V applied to the gate (Figure 7-5H).

MOSFETs do have a small on resistance, so there is a voltage drop from drain to source when the device is switched on. The on resistance of the ZVN4603A is 0.45 ohms at 1.5 amperes, which would result in a voltage drop of about 0.7V. At lower currents, the resistance and voltage drop are less.

Include a gate resistor of around 1K (as shown) to protect the driver's output if you're switching a relay, motor, or other inductive load.

High-side Switches

Another way of controlling a load with a logic voltage is to use a high-side switch like the LTC1156, a quad high-side MOSFET driver chip from Linear Technology, shown in Figure 7-6. The chip allows you to use the cheaper, more widely available N-channel MOSFETs in your designs and adds other useful features. Single and dual versions are also available, and other manufacturers have similar chips.

Most of the previous circuits have used a low-side switch, where one switch terminal connects to ground and the other connects to the load's ground terminal. In a high-side switch, the load's ground terminal connects directly to ground and the switch is between the power supply and load's power-supply terminal.

A high-side switch has a couple of advantages. For safety reasons, some circuits are designed to be off if the switch terminals happen to short to ground. With a low-side switch, shorting the switch to ground would apply power to the load. With a high-side switch, although shorting the switch to ground may destroy the

P-channel MOSFETs are the complement of N-channel MOSFETs, much as PNP transistors complement NPNs. An enhancement-mode, P-channel MOSFET switches on when the gate is more negative than the source. In depletion-mode MOSFETs (which may be N-channel or P-channel), applying a gate voltage opens the switch, rather than closing it.

Unlike a bipolar-transistor switch, which can draw several milliamperes of base current, a MOSFET gate has very high input resistance and draws virtually no current. But unlike a bipolar transistor, which needs just 0.7V from base to emitter, a MOSFET may require as much as 10V from gate to source to switch on fully.

One way to provide the gate voltage from 5V logic is to use a device with an open-collector or open-drain output and a pull-up resistor to at least 10V, as Figure 7-5G shows. Some newer MOSFETs have lower minimum *on voltages.* Zetex's ZVN4603A can switch 1.5 amperes with just +5V applied to the gate (Figure 7-5H).

MOSFETs do have a small on resistance, so there is a voltage drop from drain to source when the device is switched on. The on resistance of the ZVN4603A is 0.45 ohms at 1.5 amperes, which would result in a voltage drop of about 0.7V. At lower currents, the resistance and voltage drop are less.

Include a gate resistor of around 1K (as shown) to protect the driver's output if you're switching a relay, motor, or other inductive load.

High-side Switches

Another way of controlling a load with a logic voltage is to use a high-side switch like the LTC1156, a quad high-side MOSFET driver chip from Linear Technology, shown in Figure 7-6. The chip allows you to use the cheaper, more widely available N-channel MOSFETs in your designs and adds other useful features. Single and dual versions are also available, and other manufacturers have similar chips.

Most of the previous circuits have used a low-side switch, where one switch terminal connects to ground and the other connects to the load's ground terminal. In a high-side switch, the load's ground terminal connects directly to ground and the switch is between the power supply and load's power-supply terminal.

A high-side switch has a couple of advantages. For safety reasons, some circuits are designed to be off if the switch terminals happen to short to ground. With a low-side switch, shorting the switch to ground would apply power to the load. With a high-side switch, although shorting the switch to ground may destroy the

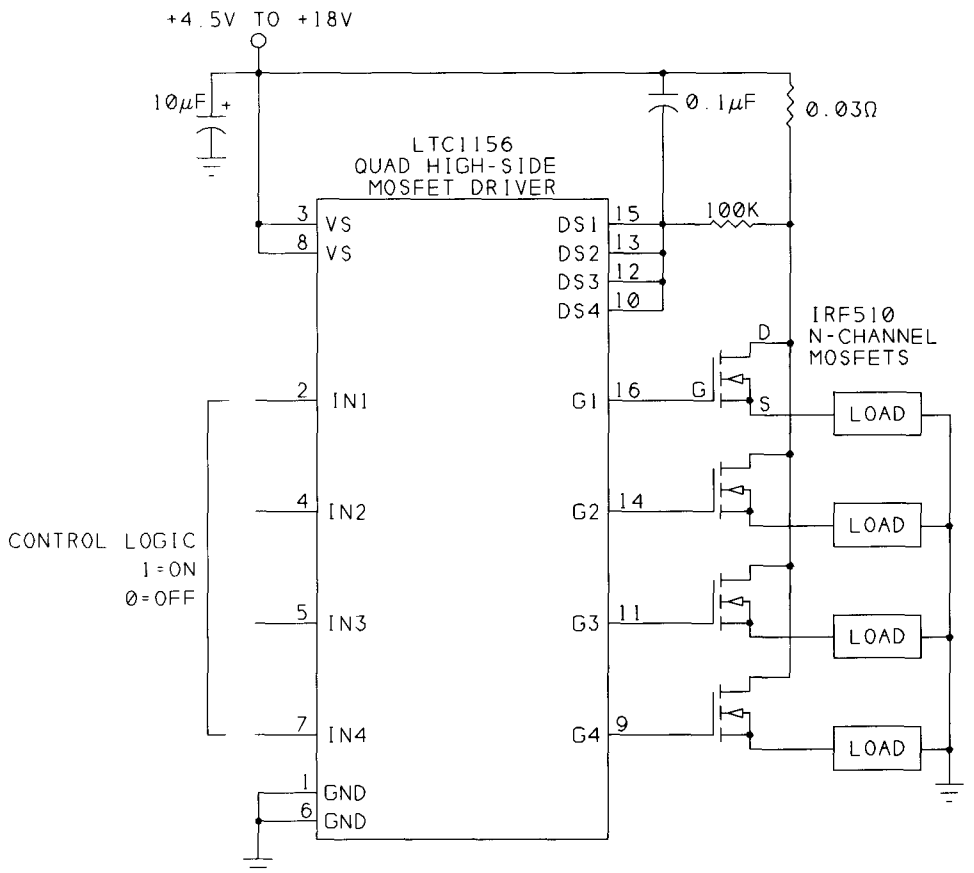


Figure 7-6: A high-side switch connects between the load and the power supply. Linear Technology's LTC1156 control high-side MOSFET switches with logic signals.

switch, it removes power from the load. (Most switches fail by opening permanently.)

Connecting the load directly to ground can also help to reduce electrical noise in the circuit. With a low-side switch, the load always floats a few tenths of a volt above ground.

The LTC1156 can control up to four MOSFETs. You can use any 5V TTL or CMOS outputs as control signals, because the switches turn on at just 2V.

Providing a high-enough gate voltage can be a problem when using an N-channel MOSFET in a high-side switch. When the MOSFET switches on, its low drain-to-source resistance causes the source to rise nearly to the supply voltage. For the MOSFET to remain on, the gate must be more positive than the source.

The LTC1156 takes care of this with charge-pump circuits that bring the gate voltages as much as 20V above the supply voltage.

By adding a small current-sensing resistor, you can cause the outputs to switch off if the MOSFETs' drain current rises above a selected value (3.3A with 30 milliohms in the circuit shown). The outputs switch off when the voltage drop across the current-sensing resistor is 100 millivolts.

Solid-state Relays

Another way to switch power to a load is to use a solid-state relay, which offers an easy-to-use, optoisolated switch in a single package. Figure 7-7A shows an example.

In a typical solid-state DC relay, applying a voltage across the control inputs causes current to flow in an LED enclosed in the package. The LED switches on a photodiode, which applies a control voltage to a MOSFET's gate, switching the MOSFET on. The result is a low resistance across the switch terminals, which effectively closes the switch and allows current to flow. Removing the control voltage turns off the LED and opens the switch.

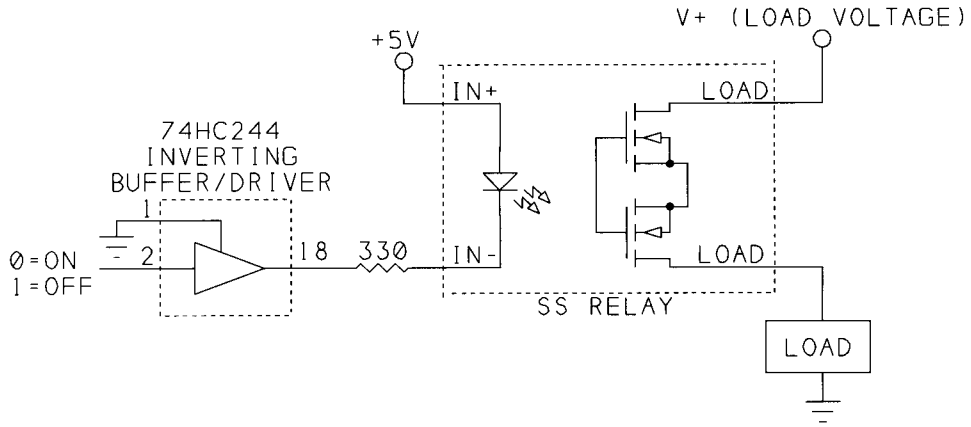
Solid-state relays are rated for use with a variety of load voltages and currents. Because the switch is optoisolated, there need be no electrical connection at all between the control signal and the circuits being switched.

Solid-state relays have an on resistance of anywhere from a few ohms to several hundred ohms. Types rated for higher voltages tend to have higher on resistances. Solid-state relays also have small leakage currents, typically a microampere or so, that flow through the switch even when off. This leakage current isn't a problem in most applications.

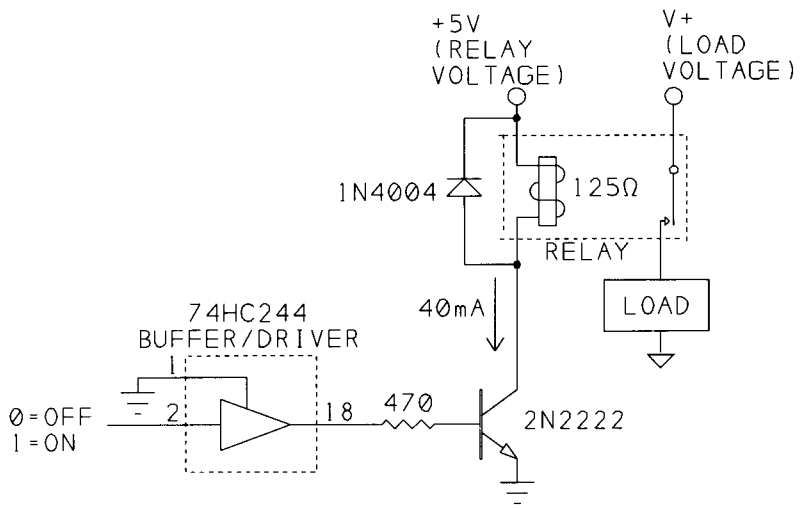
There are solid-state relays for switching AC loads as well. These provide a simple and safe way to use a logic signal to switch line voltage to a load. Inside the relay, the switch itself is usually an SCR or TRIAC. Zero-voltage switches minimize noise by switching only when the AC voltage is near zero.

Electromagnetic Relays

Electromagnetic relays have been around longer than transistors and still have their uses. An electromagnetic relay contains a coil and one or more sets of contacts attached to an armature (Figure 7-7B). Applying a voltage to the coil causes current to flow in it. The current generates magnetic fields that move the armature, opening or closing the relay contacts. Removing the coil voltage collapses the magnetic fields and returns the armature and contacts to their original positions.



(A) SOLID-STATE RELAY



(B) MECHANICAL RELAY

Figure 7-7: Solid-state and electromagnetic, or mechanical, relays are another option for switching power to a circuit. An advantage to relays is that the load is electrically isolated from the switch's control signal.

A diode across the relay coil protects the components from damaging voltages that might otherwise occur when the contacts open and the current in the coil has nowhere to go. In fact, you should place a diode in this way across any switched inductive DC load, including DC motor windings. For AC loads, use a varistor in place of the diode. The varistor behaves much like two Zener diodes connected anode-to-cathode on both ends.

Two attractions of electromagnetic relays are very low on resistance and complete physical isolation from the control signal. Because the contacts physically touch, the on resistance is typically just a few tenths of an ohm. And because the contacts open or close in response to magnetic fields, there need be no electrical connection between the coil and the contacts.

Drawbacks include large size, large current requirements (50-200 milliamperes is typical for coil current), slow switching speed, and the need for maintenance or replacement as the contacts wear. One solution to the need for high current is to use a latching relay, which requires a current pulse to switch, but then remains switched with greatly reduced power consumption.

Controlling the Bits

For simple switches, a single output bit can control power to a load. The bit routines introduced in Chapter 4 make it easy to read and change individual bits in a byte. If you store the last value written to the port in a variable, there's no need to read the port before each write.

X-10 Switches

A different way to control power to devices powered at 115V AC is to use the X-10 protocol, which can send *on, off,* and *dim* commands to a device, using a low-voltage signal carried on 115V, 60-Hz power lines. An X-10 interface is a simple way to control lights and plug-in appliances using only the existing wiring in the building.

Besides the popular manually programmed X-10 controllers and appliance modules, there are devices that enable you to program an X-10 controller from a PC, usually using a serial or parallel link to communicate with the controller.

Signal Switches

One more type of switch worth mentioning is the CMOS switch for low-power analog or digital signals. A logic signal controls the switch's operation.

Simple CMOS Switch

The 4066B quad bilateral switch is a simple and inexpensive way to switch low-power, low-frequency signals. As Figure 7-8 shows, the chip has four control

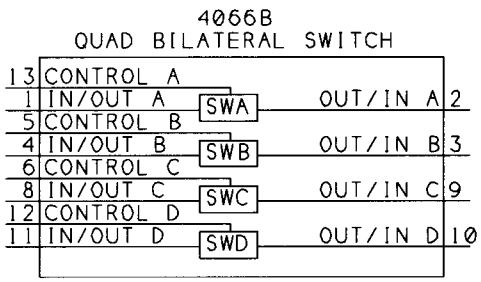


Figure 7-8: The 4066B contains four CMOS switches, each controlled by a logic signal.

inputs, each of which controls two I/O pins. A logic-high at a control input closes a switch and results in a low resistance between the corresponding I/O pins. A logic-low opens the switch, and opens the connection between the I/O pins.

The 4066B's power supply can range from 3 to 15V. With a 5V power supply, the on resistance of each switch is about 270 ohms, with the resistance dropping at higher supply voltages. The on resistance has no significant effect on standard LSTTL or CMOS logic or other signals that terminate at high-impedance inputs. An HCMOS version, the 74HC4066, has lower on resistance and, unlike other HCMOS chips, can use a supply voltage of up to 12V.

Controlling a Switch Matrix

A more elaborate switching device is the crosspoint switch, which allows complete control over the routing of two sets of lines. Examples are Harris' 74HCT22106 *Crosspoint Switch with Memory Control* and Maxim's MAX456 *8 x 8 Video Crosspoint Switch.*

Figure 7-9 shows how you can use the parallel port to control an 8 x 8 array of signals with the '22106. You can connect any of eight *X* pins to any of eight *Y* pins, in any combination. Possible applications include switching audio signals to different monitors or recording instruments, selecting inputs for test equipment, or any situation that requires flexible, changeable routing of analog or digital signals.

The '22106 simplifies circuit design and programming. It contains an array of switches, a decoder that translates a 6-bit address into a switch selection, and latches that control the opening and closing of the switches.

To connect an *X* pin to a *Y* pin, set $\overline{MR}$=1 and $\overline{CE}$=0. Then do the following:

1. Write the address of the desired *X* pin to *A0-A2* and write the address of the desired *Y* pin to *A3-A5*. Set $\overline{Strobe}$=1. Set *Data*=1.

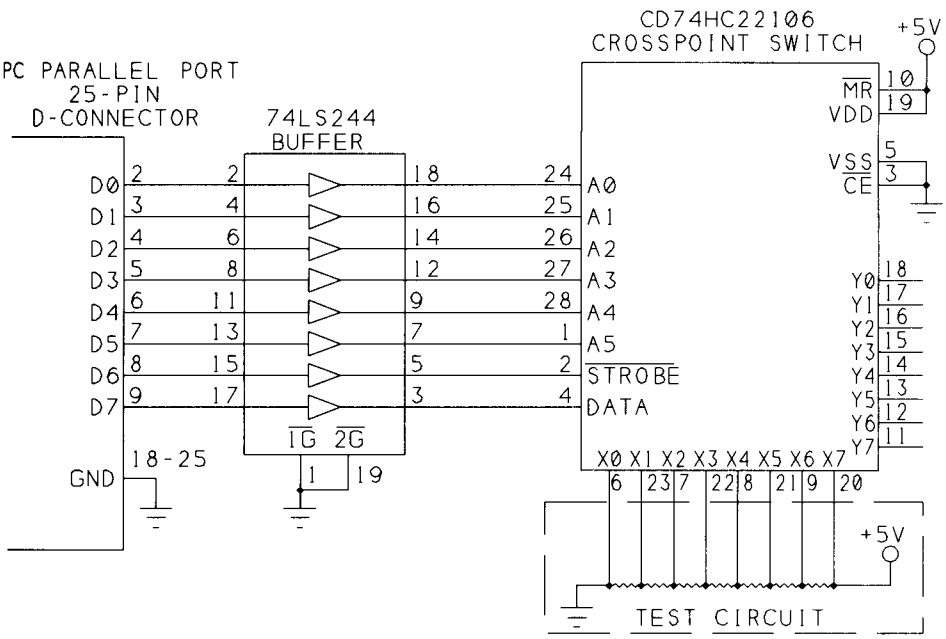


Figure 7-9: The parallel port's data lines can control an 8 x 8 crosspoint switch.

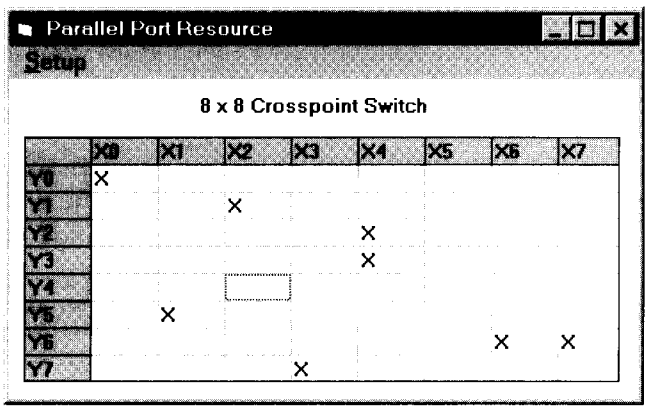


Figure 7-10: Clicking on a grid cell opens or closes the matching switch.

3. Set $\overline{Strobe}$=0 to close the requested switch, connecting the selected *X* and *Y* pins.
3. Set $\overline{Strobe}$=1.

To break a connection, do the same thing, except bring the *Data* input low to open the switch.

Figure 7-10 shows the screen for Listing 7-2's program, which demonstrates the operation of the switch matrix. The program uses Visual Basic's Grid control to

```
Const OPENSWITCH% = 0
Const CLOSESWITCH% = 1
```

```
Sub ActivateSwitch (OpenOrClose%)
Dim Strobe%
Dim XY%
'Data port bit 7 = OpenOrClose (0=open, 1=close)
OpenOrClose = OpenOrClose * &H80
'Data port bit 6 = Strobe.
Strobe = &H40
'Data port bits 0-2 hold the X value, bits 3-5 hold the Y value.
XY = grdXY.Col - 1 + (grdXY.Row - 1) * 8
'Write the address, select open or close, Strobe = 1
DataPortWrite BaseAddress, XY + Strobe + OpenOrClose
'Pulse the Strobe input.
DataPortWrite BaseAddress, XY + OpenOrClose
DataPortWrite BaseAddress, XY + Strobe + OpenOrClose
End Sub
```

```
Sub DisplayResults ()
Select Case SwitchState
    Case "Closed"
        grdXY.Text = "X"
    Case "Open"
        grdXY.Text = ""
End Select
End Sub
```

```
Sub Form_Load ()
StartUp
LabelTheGrid
End Sub
```

```
Sub grdXY_Click ()
Select Case grdXY.Text
    Case "X"
        ActivateSwitch OPENSWITCH
        SwitchState = "Open"
        DisplayResults
    Case Else
        ActivateSwitch CLOSESWITCH
        SwitchState = "Closed"
        DisplayResults
End Select
End Sub
```

Listing 7-2: Controlling an 8 x 8 crosspoint switch (Sheet 1 of 2)

```
Sub LabelTheGrid ()
Dim Row%
Dim Column%
grdXY.Col = 0
For Row = 1 To 8
    grdXY.Row = Row
    grdXY.Text = "Y" & Row - 1
Next Row
grdXY.Row = 0
For Column = 1 To 8
    grdXY.Col = Column
    grdXY.Text = "X" & Column - 1
Next Column
lblXY.Caption = "8 x 8 Crosspoint Switch"
End Sub
```

Listing 7-2: Controlling an 8 x 8 crosspoint switch (Sheet 2 of 2)

display the switch matrix. When you click on a cell, the associated switch opens or closes. An *X* indicates a closed switch, an empty cell indicates an open switch.

You can make and break as many connections as you want by writing appropriate values to the chip. All previous switch settings remain until you change them by writing to the specific switch. The switches can connect in any combination. For example, you can connect each *X* pin to a different *Y* pin to create eight distinct signal paths. Or, you can connect all eight *Y* pins to a single *X* pin, to route one signal to eight different paths. The *X* and *Y* pins may connect to external inputs or outputs in any combination.

Figure 7-9 shows the '22106 powered at +5V, but the supply voltage may range from 2 to 10V, and *Vss* (and *Vdd*) may be negative. (The HCT version (74HCT22106) requires a +5V supply.) The chip can switch any voltages within the supply range. However, the maximum and minimum values for the address and control signals vary with the supply voltage. For example, if *Vdd* is +5V and *Vss* is -5V, the address and control signals can no longer use 5V CMOS logic levels, because the logic levels are in proportion to the supply voltage. The maximum logic low for these signals drops from +1.5V to -2V (*Vss* + 0.3(|*Vdd-Vss*|)), and the minimum logic high drops from +3.5V to +2V (*Vss* +0.7(|*Vdd-Vss*|)).

At 5V, the switches' typical on resistance is 64 ohms, dropping to 45 ohms at 9V. The chip can pass frequencies up to 6 Megahertz with ±4.5V supplies.

In Figure 7-9, the parallel port's *D0-D7* control the switch array. The 74HCT244 buffer has TTL-compatible inputs and CMOS-compatible outputs. If you use a 74LS244, add a 10K pull-up resistor from each output to +5V, to ensure that logic

highs meet the '22106's 3.3V minimum. If you use a 74HC244, add pullups at the inputs to bring the parallel port's high outputs to valid CMOS logic levels.

For a simple test of the switches, you can connect a series of equal resistors as shown to the *X* inputs. Each *X* input will then be at a different voltage. To verify a switch closure, measure the voltages at the selected *X* and *Y* inputs; they should match.

Pin 3 ($\overline{CE}$) is tied low. To control multiple switches from a single parallel port, connect each switch's $\overline{CE}$ to one of the Control outputs, and wire *D0-D7* to all of the switches. You then can use the Control lines to select a switch to write to. The Reset input ($\overline{MR}$) is tied high. If you want the ability to reset all of the switches, tie this pin to one of the Control outputs.

Maxim's '456 is similar, but can pass frequencies up to 25 Megahertz, separate analog and digital ground pins, and *V+* and *V-* inputs. The address and control signals use 5V logic levels even if the chip uses another supply voltage.

Displays

Because the parallel port resides on a personal computer that has its own full-screen display, there's usually little need to use the port's outputs to control LEDs, LCDs (liquid crystal displays), or other display types. You might want to use LEDs as simple indicators to show troubleshooting or status information. And of course, you can use the port's Data and Control outputs to control other types of displays if the need arises.

8

Input Applications

Because the parallel port's most common use is to send data to a printer, you might think that the port is useful only for sending information from a PC to a peripheral. But you can also use the parallel port as an input port that reads information from external devices. SPPs have five Status inputs and four bidirectional Control lines, and on many newer ports, you can use the eight Data lines as inputs as well.

This chapter shows a variety of ways to use the parallel port for input. The examples include latched digital inputs, an expanded input port of 40 bits, and an interface to an analog-to-digital converter.

Reading a Byte

On the original parallel port, there is no way to read eight bits from a single port register. But there are several ways to use the available input bits to put together a byte of information.

Chapter 2 showed how to perform simple reads of the Status, Control, and bidirectional Data bits, and later chapters show how to use IEEE 1284's Nibble, Byte, EPP, and ECP modes to read bytes and handshake with the peripheral sending the information. The following examples show other options, including a simple way

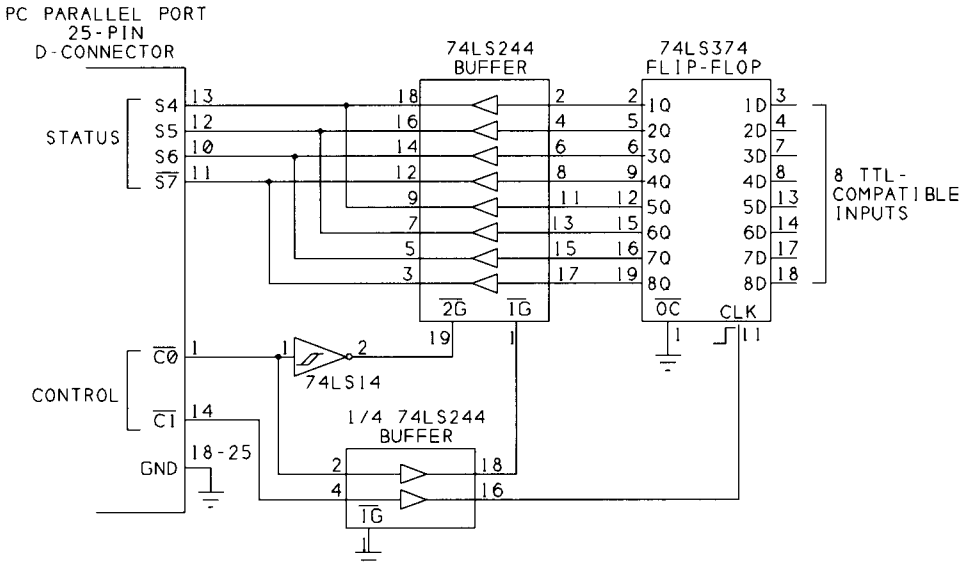


Figure 8-1: A '374 flip-flop latches a byte of data, and a Control bit selects each of two nibbles to be read at the Status port.

to read a byte in two nibbles at the Status port and how to add a latch to store the data to be read.

Latching the Status Inputs

Figure 8-1 and Listing 8-1 show a way to read bytes at the Status port. The circuit stores two nibbles (1 nibble = 4 bits), which the program reads in sequence at the Status port. One Control bit latches the data, and another selects the nibble to read.

The latch is a 74LS374 octal flip-flop. The rising edge of the *Clk* input latches the eight *D* inputs to the corresponding *Q* outputs. Even if the inputs change, the outputs will remain at their latched values until $\overline{C1}$ goes low, then high again. This ensures that the PC's software will read the state of all of the bits at one moment in time. Otherwise, the PC may read invalid data. For example, if the byte is an output from an analog-to-digital converter, the output's value may change by one bit, from *1Fh* when the PC reads the lower four bits, to *20h* when the PC reads the upper four bits. If the data isn't latched, the PC will read *2Fh*, which is very different from the actual values of *1Fh* and *20h*.

A 74LS244 buffer presents the bits to the Status port, four at a time. When $\overline{1G}$ is low, outputs *1Q-4Q* are enabled, and the PC can read inputs *1D-4D*. When $\overline{2G}$ is low, outputs *5Q-8Q* are enabled and the PC can read inputs *5D-8D*. A second '244 buffers the two Control signals. You can substitute HCT versions of the chips.

```
Option Explicit
Const SelectHighNibble% = 1
Const Clock% = 2
```

```
Sub cmdReadByte_Click ()
Dim LowNibble%
Dim HighNibble%
Dim ByteIn%
'Latch the data
ControlPortWrite BaseAddress, Clock
ControlPortWrite BaseAddress, 0
'Read the nibbles at bits 4-7.
LowNibble = StatusPortRead(BaseAddress) \ &H10
ControlPortWrite BaseAddress, SelectHighNibble
HighNibble = StatusPortRead(BaseAddress) And &HF0
ByteIn = LowNibble + HighNibble
lblByteIn.Caption = Hex$(ByteIn) + "h"
End Sub
```

Listing 8-1: Reading a byte in two nibbles at the Status port.

Listing 8-1 latches a byte of data, then reads it in two nibbles, recombines the nibbles into a byte, and displays the result. The data bits are the upper four Status bits, which makes it easy to recombine the nibbles into a byte. In the upper nibble, the bits are in the same positions as in the original byte, so there's no need to divide or multiply to shift the bits. For the lower nibble, just divide the value read by *&h10*.

Latched Input Using Status and Control Bits

Figure 8-2 is similar to the previous example, but it uses both Status and Control bits for data. Control bits 0-2 are the lower three bits, and Status bits 3-7 are the upper five bits, so each bit has the same position as in the original byte. Control bit 3 latches the data.

For this circuit, multi-mode ports must be in SPP mode to ensure that the Control bits can be used for input. Some multi-mode ports can't use the Control bits as inputs at all.

The three Control lines are driven by 7407 open-collector buffers. The remaining Control input uses another buffer in the package.

You must write *1* to Control bits 0-2's corresponding outputs in order to use them as inputs. (Because bits *0, 1,* and *3* are inverted between the port register and the connector, you actually write *4* to bits 0–3 to bring all outputs high.)

Listing 8-2 latches 8 bits, reads the Status and Control ports, recreates the original byte, and displays the result.

5 Bytes of Input

If you have a lot of inputs to monitor, Figure 8-3 shows how to read up 5 bytes at the Status port. Five outputs of a 74LS244 octal buffer drive the Status inputs, and the other 3 bits buffer the bit-select signals from $\overline{C0}$-*C2*.

Outputs $\overline{C0}$, $\overline{C1}$, and *C2* select one of eight inputs at each of five 74LS151 data selectors. At each '151, the selected input appears at output *Y*, and also in inverted form at $\overline{W}$. An output of each '151 connects through a buffer to one of the Status inputs. To read a bit from each '151, you write to $\overline{C0}$-C2 to select the bit, then read S3-$\overline{S7}$.

Listing 8-3 reads all 40 bits, 5 bits at a time, combines the bits into bytes, and displays the results. Figure 8-4 is the program screen. Since the '151 has both normal and inverted outputs, you could use the $\overline{W}$ output at $\overline{S7}$ to eliminate having to reinvert the bit in software. Listing 8-3 uses the *StatusPortRead* routine that automatically reinverts bit 7, so Figure 8-3 uses the *Y* output.

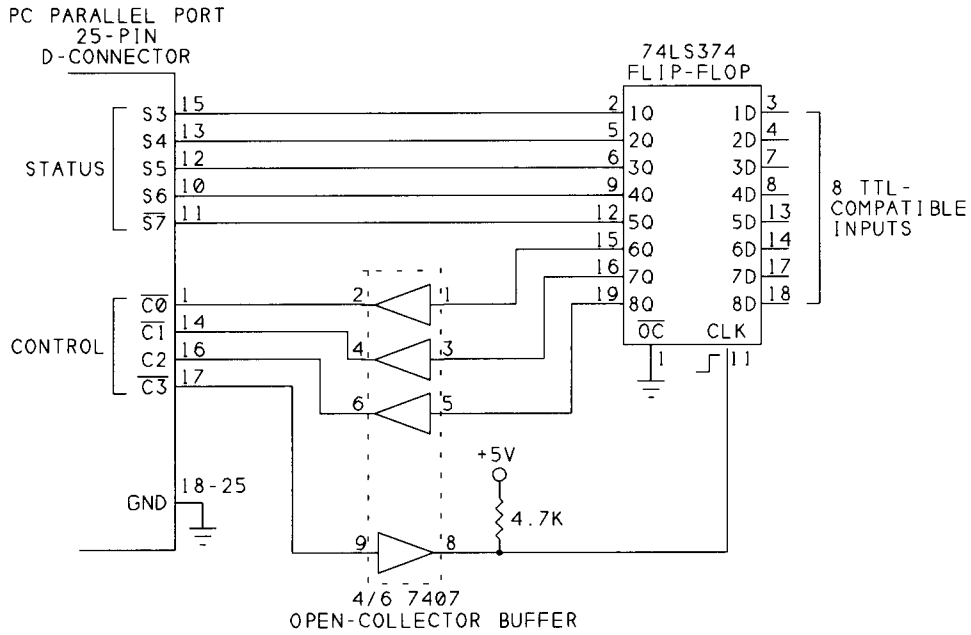


Figure 8-2: Eight latched input bits, using the Status and Control ports.

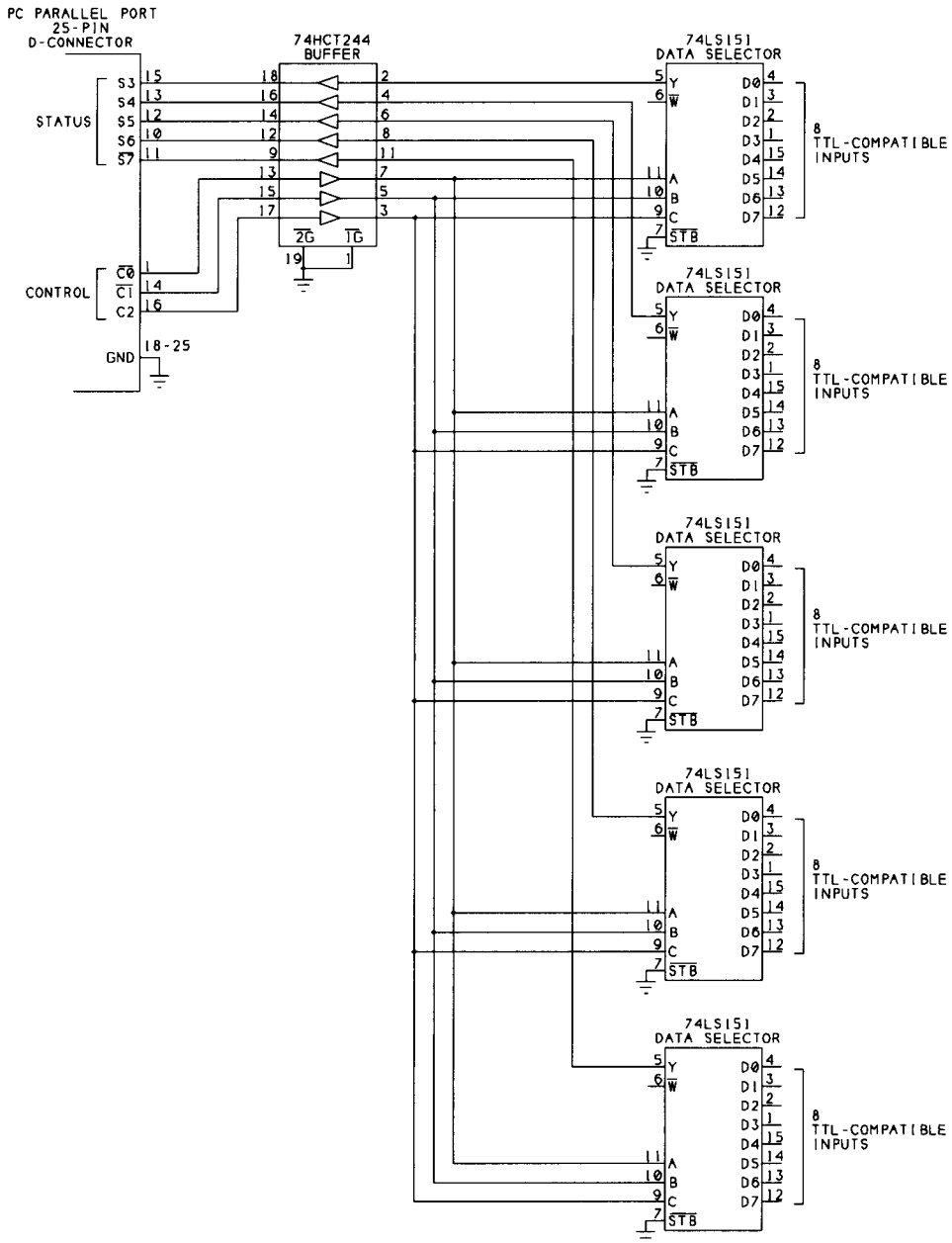


Figure 8-3: Forty input bits, read in groups of five.

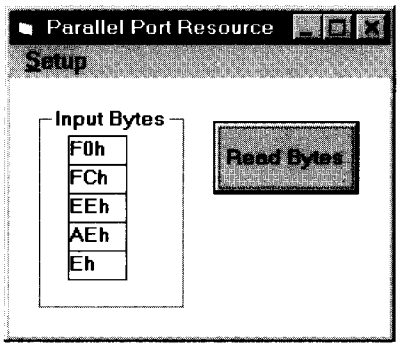


Figure 8-4: Screen for Listing 8-3's program.

Using the Data Port for Input

If you have a bidirectional data port, you can use the eight data lines as inputs. You can also use the port as an I/O port, both reading and writing to it, as long as you're careful to configure the port as input whenever outputs are connected and enabled at the data pins. In other words, when the data lines are configured as outputs, be sure to tristate, or disable, any external outputs they connect to. You can use a '374 to latch input at the Data port, as in the previous examples.

Reading Analog Signals

The parallel port is a digital interface, but you can use it to read analog signals, such as sensor outputs.

Sensor Basics

A sensor is a device that reacts to changes in a physical property or condition such as light, temperature, or pressure. Many sensors react by changing in resistance. If a voltage is applied across the sensor, the changing resistance will cause a change in the voltage across the sensor. An analog-to-digital converter (ADC) can convert the voltage to a digital value that a computer can store, display, and perform calculations on.

Simple On/Off Measurements

Sometimes all you need to detect is the presence or absence of the sensed property. Some simple sensors act like switches, with a low resistance in the presence

```
'Clock is Control bit 3.
Const Clock% = 8
'Write 1 to bits C0-C2 to allow their use as inputs.
Const SetControlBitsAsInputs% = 7
```

```
Sub cmdReadByte_Click ()
Dim LowBits%
Dim HighBits%
Dim ByteIn%
'Latch the data.
ControlPortWrite BaseAddress, SetControlBitsAsInputs + Clock
ControlPortWrite BaseAddress, SetControlBitsAsInputs
'Read the bits at C0-C2, S3-S7.
LowBits = ControlPortRead(BaseAddress) And 7
HighBits = StatusPortRead(BaseAddress) And &HF8
ByteIn = LowBits + HighBits
lblByteIn.Caption = Hex$(ByteIn) + "h"
End Sub
```

```
Sub Form_Load ()
'(partial listing)
'Initialize the Control port.
ControlPortWrite BaseAddress, SetControlBitsAsInputs
End Sub
```

Listing 8-2: Reading 8 bits using the Status and Control ports.

of the sensed property, and a high resistance in its absence. In this case, you can connect the sensor much like a manual switch, and read its state at an input bit. Sensors that you can use this way include magnetic proximity sensors, vibration sensors, and tilt switches.

Level Detecting

Another common use for sensors is to detect a specific level, or intensity, of a property. For this, you can use a comparator, a type of operational amplifier (op amp) that brings its output high or low depending on which of two inputs is greater.

Figure 8-5 shows how to use a comparator to detect a specific light level on a photocell. The circuit uses an LM339, a general-purpose quad comparator. The resistance of a Cadmium-sulfide (CdS) photocell varies with the intensity of light on it. Pin 4 is a reference voltage, and pin 5 is the input being sensed. When the sensed

input is lower than the reference, the comparator's output is low. When the sensed input is higher than the reference, the comparator's output is high.

As the light intensity on the photocell increases, the photocell's resistance decreases and pin 5's voltage rises. To detect a specific light level, adjust *R2* so that *Vout* switches from low to high when the light reaches the desired intensity. You can read the logic state of *Vout* at any input bit on the parallel port.

R4 is a pull-up resistor for the LM339's open-collector output. *R3* adds a small amount of hysteresis, which keeps the output from oscillating when the input is near the switching voltage.

You can use the same basic circuit with other sensors that vary in resistance. Replace the photocell with your sensor, and adjust *R2* for the switching level you want.

Reading an Analog-to-digital Converter

When you need to know the precise value of a sensor's output, an analog-to-digital converter (ADC) will do the job. Figure 8-6 is a circuit that enables you to read eight analog voltages. The ADC0809 converter is inexpensive, widely available, and easy to interface to the parallel port. The ADC0808 is the same chip with higher accuracy, and you may use it instead.

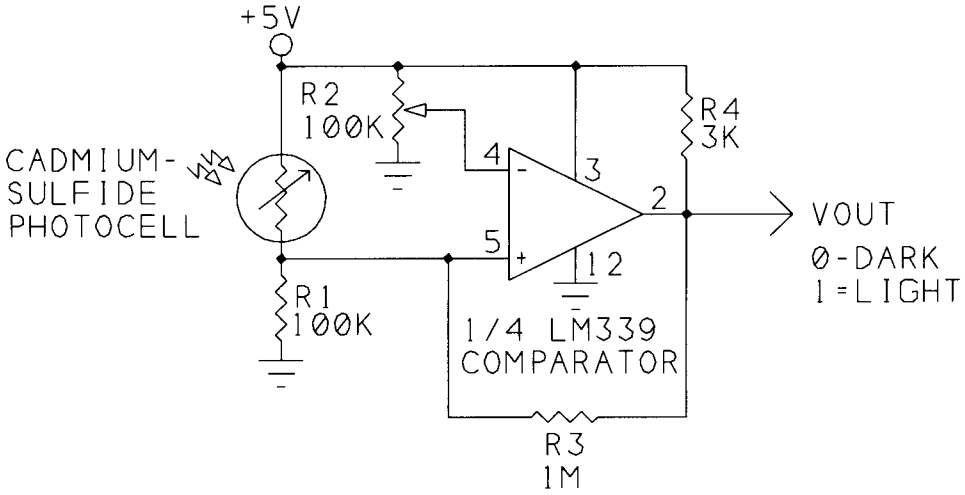


Figure 8-5: A comparator can detect a specific voltage.

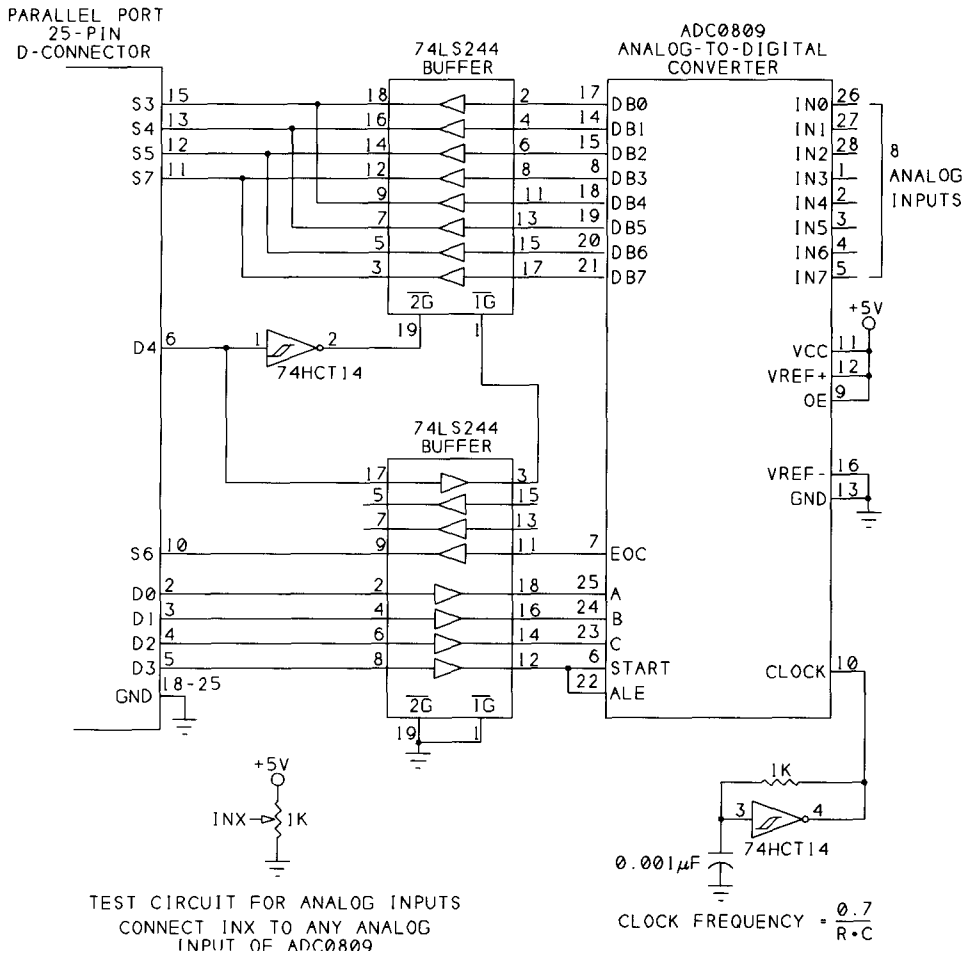


Figure 8-6: The ADC0809 analog-to-digital converter provides a simple way to read 8 analog channels at the parallel port.

The ADC0809 has eight analog inputs (*IN0-IN7*), which may range from 0 to +5V. To read the value of an analog input, you select a channel by writing a value from 0 to 7 to inputs *A-C*, then bringing *Start* and *Ale* high, then low, to begin the conversion. When the conversion is complete, *Eoc* goes high and the digital outputs hold a value that represents the analog voltage read.

The chip requires a clock signal to control the conversion. A 74HCT14 Schmitt-trigger inverter offers a simple way to create the clock. The frequency can range from 10 kilohertz to 1280 kilohertz. If you prefer, you can use a 555 timer for the clock, although the maximum frequency of the 555 is 500 kilohertz. Conversion time for the ADC is 100 microseconds with a 640-kilohertz clock.

```
Dim DataIn%(0 To 7)
Dim DataByte%(0 To 4)
```

```
Sub cmdReadBytes_Click ()
Dim BitNumber%
'The Control port selects a bit number to read.
'The Status port holds the data to be read.
For BitNumber = 0 To 7
    ControlPortWrite BaseAddress, BitNumber
    DataIn(BitNumber) = StatusPortRead(BaseAddress)
Next BitNumber
GetBytesFromDataIn
DisplayResults
End Sub
```

```
Sub DisplayResults ()
Dim ByteNumber%
For ByteNumber = 0 To 4
lblByteIn(ByteNumber).Caption = Hex$(DataByte(ByteNumber)) & "h"
Next ByteNumber
End Sub
```

Listing 8-3: Reading 40 inputs. (Sheet 1 of 2)

```
Sub GetBytesFromDataIn ()
'Bits 3-7 of the 8 bytes contain data.
'To make 5 data bytes from these bits,
'each data byte contains one bit from each byte read.
'For example, data byte 0 contains 8 "bit 3s,"
'one from each byte read.
Dim ByteNumber%
Dim BitNumber%
Dim BitToAdd%
For ByteNumber = 0 To 4
    DataByte(ByteNumber) = 0
    'BitRead gets the selected bit value (ByteNumber + 3)
    'from the selected byte read (DataIn(BitNumber)).
    'To get the bit value for the created data byte,
    'multiply times 2^BitNumber.
    'Add each bit value to the created byte.
    For BitNumber = 0 To 7
        BitToAdd = (BitRead(DataIn(BitNumber), ByteNumber + 3)) _
         * 2 ^ BitNumber
        DataByte(ByteNumber) = DataByte(ByteNumber) + BitToAdd
    Next BitNumber
Next ByteNumber
End Sub
```

Listing 8-3: Reading 40 inputs. (Sheet 2 of 2)

Inputs *Vref+* and *Vref-* are references for the analog inputs. When an analog input equals *Vref-*, the digital output is zero. When the input equals *Vref+*, the digital output is 255. You can connect the reference inputs to the +5V supply and ground, or if you need a more stable reference or a narrower range, you can connect other voltage sources to the references.

Listing 8-4 reads all eight channels and displays the results. It reads the data in two nibbles at *S3–S5* and $\overline{S7}$. Outputs *D0–D2* select the channel to convert, *D3* starts the conversion, and *D4* selects the nibble to read. Optional input *S6* allows you to monitor the state of the ADC's end-of-conversion (*Eoc*) output.

A 74LS244 drives the Status bits. When *D4* is low, you can read the ADC's *DB0–DB3* outputs at the Status port. When *D4* is high, you can read *DB4–DB7*.

A second 74LS244 interfaces the other signals to the ADC. Bringing *D3* high latches the channel address from *D0–D2*, and bringing *D3* low starts a conversion.

Bit *S6* goes high when the ADC has completed its conversion. You can monitor *S6* for a logic high that signals that the conversion is complete, or you can use the

```
Const Start% = 8
Const HighNibbleSelect% = &H10
Dim DataIn%(0 To 7)
Dim ChannelNumber%
Dim LowNibble%
Dim HighNibble%
```

```
Sub cmdReadPorts_Click ()
Dim EOC%
For ChannelNumber = 0 To 7
    'Select the channel.
    DataPortWrite BaseAddress, ChannelNumber
    'Pulse Start to begin a conversion.
    DataPortWrite BaseAddress, ChannelNumber + Start
    DataPortWrite BaseAddress, ChannelNumber
    'Wait for EOC
    Do
        DoEvents
        LowNibble = StatusPortRead(BaseAddress)
        EOC = BitRead(LowNibble, 6)
    Loop Until EOC = 1
    'Read the byte in 2 nibbles.
    DataPortWrite BaseAddress, ChannelNumber + HighNibbleSelect
    HighNibble = StatusPortRead(BaseAddress)
    DataIn(ChannelNumber) = MakeByteFromNibbles()
Next ChannelNumber
DisplayResult
End Sub
```

```
Sub DisplayResult ()
For ChannelNumber = 0 To 7
    lblADC(ChannelNumber).Caption = _
    Hex$(DataIn(ChannelNumber)) & "h"
Next ChannelNumber
End Sub
```

Listing 8-4: Reading 8 channels from an ADC. (Sheet 1 of 2)

```
Function MakeByteFromNibbles% ()
Dim S0%, S1%, S2%, S3%, S4%, S5%, S6%, S7%
S0 = (LowNibble And 8) \ 8
S1 = (LowNibble And &H10) \ 8
S2 = (LowNibble And &H20) \ 8
S3 = (LowNibble And &H80) \ &H10
S4 = (HighNibble And 8) * 2
S5 = (HighNibble And &H10) * 2
S6 = (HighNibble And &H20) * 2
S7 = HighNibble And &H80
MakeByteFromNibbles = S0 + S1 + S2 + S3 + S4 + S5 + S6 + S7
End Function
```

Listing 8-4: Reading 8 channels from an ADC. (Sheet 2 of 2)

rising edge at *S6* to trigger an interrupt, or you can ignore *S6* and just be sure to wait long enough for the conversion to complete before reading the result.

The circuit uses *S6* as end-of-convert because it's the parallel port's interrupt pin. If you don't use interrupts, you can wire the ADC's data outputs to S4–*S7* for an easier (and faster) conversion from nibbles to byte.

At each analog input, you can connect any component whose outputs ranges from 0 to +5V.

Sensor Interfaces

If the output range of your sensor voltages is much less than 5V, you can increase the resolution of the conversions by adjusting the reference voltages to a range that is slightly wider than the range you want to measure.

To illustrate, consider a sensor whose output ranges from 0 to 0.5V. The 8-bit output of the converter represents a number from 0 to 255. If *Vref+ is* 5V and *Vref-* is 0V, each count equals 5/255, or 19.6 millivolts. A 0.2V analog input results in a count of 10, while a 0.5V input results in a count of 26. If your input goes no higher than 0.5V, your count will never go higher than 26, and the measured values will be accurate only to within 20 millivolts, or 1/255 of full-scale.

If you lower *Vref+* to 0.5V, each count now equals 0.5/255, or 0.002V. A 0.2-volt input gives a count of 102, a 0.5-volt input gives a count of 255, and the measured values can be accurate to within 2 millivolts.

If you decrease the range, you also increase the converter's sensitivity to noise. With a 5V range, a 20-millivolt noise spike will cause at most a 1-bit error in the

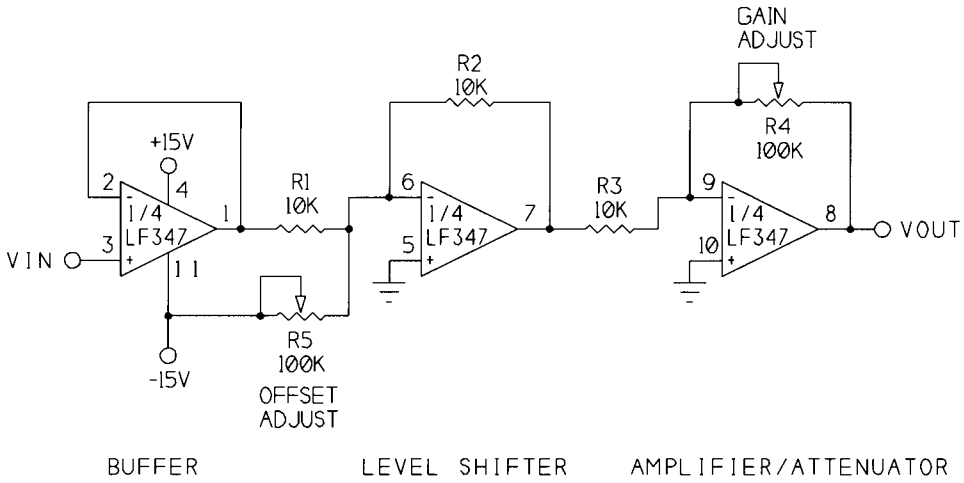


Figure 8-7: With this circuit you can adjust the offset and amplitude of an analog signal.

output. With a 0.5V range, the same spike can cause an error of 10 bits, since each bit now represents just 2 millivolts, rather than 20.

The lower reference doesn't have to be 0V. For example, the output of an LM34 temperature sensor is 10 millivolts per degree Fahrenheit. If you want to measure temperatures from 50 to 100 degrees, you can set *Vref-* to 0.5V and *Vref+* to 1V, for a 50-degree range, or 0.2 degree per bit.

Signal Conditioning

Not every sensor has an output that can connect directly to the ADC0809's inputs. A sensor's output may range from -2 to 0V, from -0.5 to +0.5V, or from -12 to +12V. In all of these cases, you need to shift the signal levels and/or range to be compatible with a converter that requires inputs between 0 and 5 volts.

Figure 8-7 shows a handy circuit that can amplify or reduce input levels, and can also raise or lower the output by adding or subtracting a voltage. Separate, independent adjustments control the gain and offset. The circuit is a series of three op amps: a buffer, a level shifter, and an amplifier. The circuit uses three of the devices in an LF347 quad JFET-input op amp, which has fast response and high input impedance. You can use another op amp if you prefer.

The first op amp is a noninverting amplifier whose output at pin 1 equals *Vin*. The op amp presents a high-impedance input to VIN. The second op amp is an inverting summing amplifier that raises and lowers pin 1's voltage as *R5* is adjusted. Varying *R5* changes the voltage at pin 7, but the signal's shape and peak-to-peak amplitude remain constant. The third op amp is an inverting amplifier whose gain

is adjusted by *R4*. This amplifier increases or decreases the peak-to-peak amplitude of its input.

As an example of how to use the circuit, if *Vin* will vary from +0.2V to -0.2V, set *Vin* to +0.2V and adjust *R4* until Vout is +2.5V. Then set *Vin* to -0.2V and adjust *R5* until *Vout* is 0V.

If the range of *Vin* is too large, use R4 to decrease the gain instead of increasing it. If you need to shift the signal level down (to a lower range) instead of up, connect *R5* to +15V instead of -15V. If you don't need level shifting, you can remove *R5* and connect pin 6 only to *R1* and *R2*.

Minimizing Noise

Rapid switching of digital circuits can cause voltage spikes in the ground lines. Even small voltage spikes can cause errors in analog measurements. Good routing of ground wires or printed-circuit-board traces can minimize noise in circuits that mix analog and digital circuits.

To reduce noise, provide separate ground paths for analog and digital signals. Wire or route all ground connections related to the analog inputs or reference voltages together, but keep them separate from the ground connections for the digital circuits, including the clock and buffer/driver chips. Tie the two grounds together at one place only, as near to the power supply as possible. Also be sure to include decoupling capacitors, as described in Chapter 6.

Using a Sample and Hold

An additional component that you may need for rapidly changing analog inputs is a sample-and-hold circuit. To ensure correct conversions, the analog input has to remain stable while the conversion is taking place.

A sample-and-hold circuit ensures that the analog signal is stable by sampling the signal at the desired measurement time and storing it, usually as a charge on a capacitor. The converter uses this stored signal as the input to be converted.

When do you need a sample-and-hold? Clocked at 640 kHz, the ADC0809 requires 100 microseconds to convert, and you'll get good results with inputs that vary less than 1 bit in this amount of time. For rapidly changing inputs, sample-and-hold chips like the LF398 are available, or you can use a converter with a sample-and-hold on-chip.