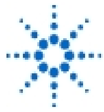


Agilent Technologies

EducatorsCorner.com Experiments

Serial Communication with Agilent Instruments under DOS, Windows 95/98/NT/2000, and UNIX

By: Michael A. Covington
Anil Bahuman
Vic Bancroft
Artificial Intelligence Center
The University of Georgia
Athens, GA 30602-7415
mc@ai.uga.edu

Abstract

Many Agilent instruments have serial ports through which they can communicate with personal computers. Although less versatile than GPIB, serial communication takes advantage of a port already installed in the computer. This paper describes how to control Agilent instruments through the serial port in various programming languages under DOS, Windows 95/98/NT/2000, and UNIX. Files associated with this report are available from ftp://ftp.ai.uga.edu/pub/serial.port.

Equipment

- Agilent 54645D Mixed-Signal Oscilloscope or Agilent 34401A Digital Multimeter
- A PC with DOS and/or Win 95/98/NT/2000
- Programming languages: Qbasic and/or C/C++
- An RS-232 cable (Agilent F1047-80002)

1. Introduction

Many Agilent instruments have serial ports through which they can communicate with personal computers. Through the serial port, you can set any of the front-panel controls of the instrument and read back whatever data it is displaying.

Although less versatile than GPIB, serial communication does not require special hardware in the computer; thus, it is convenient for quick setups and one-time experiments.

This paper describes how to communicate with Agilent instruments through the serial port in various programming languages under DOS, Windows 95/98/NT, and UNIX. The programming examples here are as simple as possible and are provided with the expectation that the user will add more functionality as needed. The techniques documented here were tried out on an Agilent 54645D oscilloscope, an Agilent 34401A multimeter, and an Agilent E3631A power supply.

2. The RS-232 hardware interface

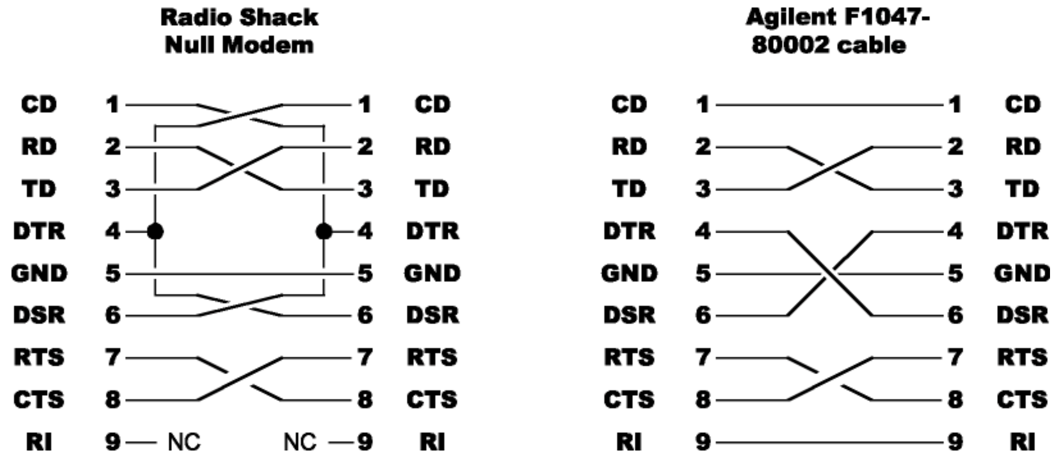
2.1. RS-232 versus GPIB

There are two ways to connect Agilent instruments to personal computers. The GPIB interface is a bus similar to SCSI; many Agilent instruments can be connected to a single GPIB card. The RS-232 serial interface supports only one Agilent instrument per serial port on the computer. The advantage of RS-232 is that it requires no special hardware and is supported by a larger variety of programming languages.

2.2. Cabling

The 9-pin serial port on an Agilent instrument has the same connector and pinout as that of an IBM PC. One way to cable the instrument to the PC is to use a Agilent F1047-80002 cable, which has DB-9 sockets (not plugs) at both ends and, internally, swaps the input and output lines as required.

Another way is to use a straight-through 9-conductor cable plus a *null modem adapter* such as Radio Shack's 9-pin null modem to swap the inputs and outputs; or you can make your own cable. Two sets of connections, both of which work, are the following:



Beware of 9-to-25-pin adapters that do not connect all the signal pins. Many of these adapters are designed to work only with a serial mouse.

2.3. Baud rate and parity

The Agilent instrument will, in general, give you a choice of baud rates and parity settings; in this report we always use 9600 baud, 8 data bits, no parity bit. (In addition, the signal has one start bit and two stop bits; that's standard for Agilent instruments and almost everything else.) See the instrument's manual for more details.

2.4. Handshaking

The Agilent instrument and the computer have to be able to tell each other not to transmit at certain times. This is called *handshaking.*

The serial port takes care of handshaking if it is set up properly. In the programming examples, we will show you how to do this. For reference, here are the details of the handshaking methods used.

Most newer Agilent instruments, including the E3631A and 34401A, use simple *DTR handshaking*, as follows:

- The DTR output of each device (Agilent instrument or PC) is connected to the DSR input of the other device. A "high" signal (+3V to +12V) on DTR indicates that transmission is allowed.
- The instrument's CTS (clear to send) line must also be held high in order for it to transmit.

The Agilent 54645D oscilloscope is different. Its “DTR handshaking” mode is actually RTS, not DTR, handshaking. That is:

- The DTR output of each device is connected to the DSR input of the other device. The oscilloscope holds DTR high as long as it is turned on.
- The RTS (Ready To Send) output of each device is connected to the CTS (Clear To Send) input of the other device. The oscilloscope holds RTS high when it is ready to accept input.

Some Agilent instruments also support another handshaking mode, XON/XOFF, that relies on transmitted characters rather than control lines. With this protocol, your software must scan the input for Ctrl-S (“XOFF”), which means “do not transmit,” and Ctrl-Q (“XON”), which means “resume transmitting.” Because this requires extra software action and because it is not available on all Agilent instruments, we do not use it here.

2.5. What to transmit

The full command language of each Agilent instrument is described in its manual, but for a quick start, don’t read about the language — go straight to the sample computer programs that Agilent gives you. Even if they use GPIB or some other interface, they’re full of working examples that tell you exactly what strings to send to it and what responses to expect.

In general, every command sent to an Agilent instrument must end with Line Feed (Ctrl-J, ASCII decimal 10). Whether to precede this with Return (Ctrl-M, ASCII decimal 13) is up to you.

The first couple of commands in each dialogue are generally something like `:SYST:REM` to put the instrument in remote mode, `*RST` to reset it, and `*CLR` to clear errors. Clearing errors is important because if the instrument receives an unrecognized command, it will remain hung until powered off or reset.

Agilent instruments transmit numbers as character strings in E format, such as +2.034E+2 to represent 203.4 (that is, $+2.034$ _ 10^{+2}). Out-of-range values are represented by extremely large or small numbers, not by non-numeric strings.

2.6. Troubleshooting

Here are solutions to some common problems that arise when setting up an RS-232 interface.

No communication in either direction: Check that the instrument and the computer are set to the same baud rate. Check that appropriate cable lines are connected, including RTS and CTS. Most Agilent instruments must have CTS high (positive) in order to communicate. Be sure to reset and clear (“*RST”, “*CLR”) the instrument at the beginning of the dialogue. Be sure to end all commands with Line Feed.

Instrument responds to commands from PC, but PC never receives its responses: Usually, this indicates a problem with the CTS data line into the instrument.

3. A basic set of software functions

As far as possible, all the computer programs in this report implement the same basic set of functions, namely:

SerMessage to put a message on the screen;
SerCrash to put a message on the screen and register an error (or, in BASIC, just terminate the program);

SerStatus to inquire whether the most recent operation was successful (not needed in BASIC);
SerOpen to open the serial port (with a parameter to indicate whether to use DTR or RTS handshaking);

SerClose to close the serial port;
SerPut to transmit a character string to the instrument;
SerGet to receive a line of data from the instrument as a character string;
SerGetNum to receive a line of data from the instrument as a number; and
SerGetChar to receive a single byte of data from the instrument.

The arguments required by these functions depend somewhat on the programming language.

4. Interfacing with QBASIC under DOS

By far the quickest and simplest way to try out a serial interface is to use QBASIC, a free Microsoft BASIC interpreter provided with recent versions of DOS and Windows. QBASIC includes full on-line documentation. On the Windows 95 distribution disk, QBASIC.EXE and QBASIC.HLP are located in `\other\oldmsdos`; they can also be downloaded from *www.microsoft.com*.

QBASIC buffers the data coming into the serial port so that your program can read it whenever it's ready. Many operating systems do this, including UNIX and Windows, but DOS does not. Thus, if you are developing a DOS-based serial communication program in any other language, you will, in general, have to provide an interrupt service routine for the serial port. QBASIC eliminates that chore.

QBASIC runs under DOS or in a DOS box under Windows 95, 98, 2000, or NT. The Microsoft QuickBasic compiler, a commercial product, compiles QBASIC programs into DOS executables.

4.1 Two very simple prototypes

Here is the a very simple QBASIC program that demonstrates serial communication with an Agilent 54645D oscilloscope:

```
' File AGILENTSCOPE1.BAS, QBASIC, M. Covington 1999
CLS
PRINT "Very simple communication with Agilent 54645D oscilloscope"
PRINT
PRINT "Set oscilloscope to 9600 baud, DTR handshaking."
PRINT "Press Enter..."
LINE INPUT junk$

OPEN "COM1:9600,n,8,1,LF" FOR RANDOM AS #1
PRINT "COM1 opened"

' Reset the scope and clear errors
PRINT #1, "*RST"      ' Reset it and clear errors
PRINT #1, "*CLS"

' Display a message on the oscilloscope screen
PRINT #1, ":SYSTEM:DSP 'Communicating with computer'"

PRINT "Taking a measurement..."
PRINT #1, ":AUTOSCALE"
PRINT #1, ":MEASURE:VRMS? ANALOG1"

' Pause a few seconds for response to become ready
SLEEP 3

' Grab the response
data$ = INPUT$(LOC(1), #1)
PRINT "Analog channel 1 V(rms) = ", VAL(data$)

' Say goodbye
PRINT
PRINT "Press Enter..."
LINE INPUT junk$

CLOSE #1
PRINT "COM1 closed."

END
```

The program opens the serial port as file #1, sends commands using the PRINT statement (which causes each command to end with Carriage Return and Line Feed), and receives input by simply grabbing the entire contents of the receive buffer. This last step is, as you might

guess, somewhat risky because the oscilloscope may not have sent all the data yet. That's why it's preceded by a 3-second delay.

Here's a very similar program that communicates with an Agilent 34401A multimeter. Note the addition of "CS" to the OPEN statement to tell the computer to handshake only on the DTR line, not the RTS line.

```
' File AGILENTMETER1.BAS, QBASIC, M. Covington 1999
CLS
PRINT "Very simple communication with Agilent 34401A multimeter"
PRINT
PRINT "Set multimeter to 9600 baud, no parity, SCPI language."
PRINT "Press Enter..."
LINE INPUT junk$

OPEN "COM1:9600,n,8,1,LF,CS" FOR RANDOM AS #1  ' note CS added
PRINT "COM1 opened"

' Select remote mode
PRINT #1, "SYST:REM"

' Reset the meter and clear errors
PRINT #1, "*RST"      ' Reset it and clear errors
PRINT #1, "*CLS"

' Display a message on the oscilloscope screen
PRINT #1, "DISP:TEXT 'HELLO'"

PRINT "Taking a measurement..."
PRINT #1, ":MEASURE:VOLTAGE:DC?"

' Pause for the response to become ready
SLEEP 3

' Grab the response
data$ = INPUT$(LOC(1), #1)
PRINT "'"; data$; "'"
PRINT "DC voltage = ", VAL(data$)

' Say goodbye
PRINT
PRINT "Press Enter..."
LINE INPUT junk$

CLOSE #1
PRINT "COM1 closed."

END
```

4.2. A more sophisticated QBASIC program

Here is a more sophisticated QBASIC program that communicates with the same oscilloscope, but this time, the various operations are encapsulated into subroutines. The input routine is also smarter. This program receives input by grabbing bytes one by one until a line terminator (Carriage Return or Line Feed) is received or ten seconds have elapsed. That is, it will wait for a complete line of input to arrive, but if the communication link goes dead, it will bail out after ten seconds.

The algorithms in this program were used as a basis for the Visual Basic, C, and C++ programs that follow.

```
' File AGILENTSCOPE2.BAS, QBASIC, M. Covington 1999

DECLARE SUB SerOpen (Port$, Handshake$)
DECLARE SUB SerClose ()
DECLARE SUB SerPut (data$)
DECLARE SUB SerGetChar (byte$)
DECLARE SUB SerGet (data$)
DECLARE SUB SerGetNum (x AS SINGLE)

CLS
PRINT "Modular QBASIC program for communicating"
PRINT "with Agilent 54645D oscilloscope"
PRINT
PRINT "Set oscilloscope to 9600 baud, DTR handshaking."
PRINT "Press Enter..."
LINE INPUT junk$

SerOpen "COM1", "RTS"

' Reset the scope and clear errors
SerPut "*RST"       ' Reset it and clear errors
SerPut "*CLS"

' Display a message on the oscilloscope screen
SerPut ":SYSTEM:DSP 'Communicating with computer'"

' Take some measurements
SerPut ":AUTOSCALE"
SerPut ":MEASURE:VRMS? ANALOG1"
SerGetNum vrms
PRINT "Analog channel 1 V(rms) = ", vrms
SerPut ":MEASURE:VPP? ANALOG2"
SerGetNum vpp2
PRINT "Analog channel 2 V(p-p) = ", vpp2

' Say goodbye
PRINT
PRINT "Press Enter..."
LINE INPUT junk$
SerClose
END

SUB SerClose
  CLOSE #99
  PRINT "[ Serial port closed.] "
END SUB

SUB SerGet (data$)
' Inputs a line as a string of characters
  data$ = ""
  ' Eat any Return or Line Feed characters
  ' left over from end of previous line
  DO
    SerGetChar byte$
  LOOP UNTIL (LEN(byte$) = 0) OR (ASC(byte$) > 31)
  ' Read characters until Line Feed, Return,
  ' or timeout occurs
  DO
    IF byte$ = CHR$(10) THEN EXIT SUB
```

```
    IF byte$ = CHR$(13) THEN EXIT SUB
    IF byte$ = "" THEN EXIT SUB
    data$ = data$ + byte$
    SerGetChar byte$  ' the next one
  LOOP
END SUB

SUB SerGetChar (byte$)
' Retrieves a character from the serial port,
' or times out, returning an empty string,
' if nothing is received for 10 s.
  seconds = 0     ' time elapsed
  t$ = TIME$      ' changes once per second
  WHILE seconds < 10
    IF LOC(99) > 0 THEN
      byte$ = INPUT$(1, 99)
      EXIT SUB
    END IF
    IF t$ <> TIME$ THEN
      seconds = seconds + 1
      t$ = TIME$
    END IF
  WEND
  byte$ = "" ' timed out, return empty string
END SUB

SUB SerGetNum (x AS SINGLE)
' Inputs a number from the serial port.
' No error checking.
  SerGet data$
  x = VAL(data$)
END SUB

SUB SerOpen (Port$, Handshake$)
' Opens "COM1" or "COM2", 9600 baud.
' Handshake$ is "DTR" or "RTS".
' Times out if unsuccessful.
  IF UCASE$(LEFT$(Handshake$, 1)) = "R" THEN
    ' RTS/CTS handshaking
    OPEN Port$ + ":9600,n,8,1,LF" FOR RANDOM AS #99

ELSE
    ' DTR/DSR handshaking
    OPEN Port$ + ":9600,n,8,1,LF,CS" FOR RANDOM AS #99
  END IF
  PRINT "[ Serial port " + Port$ + " opened.]"
END SUB

SUB SerPut (data$)
' Transmits a string, adding CR and LF at end
  PRINT #99, data$
  ' If you wanted to end with LF only, do this:
  ' PRINT #99, data$ + CHR$(10);
END SUB
```

To communicate with an Agilent 34401A multimeter, only the main program needs to be changed, as follows:


```
SerOpen "COM1", "DTR"
```

```
SerPut ":SYST:REM"    ' Remote mode
SerPut "*RST"         ' Reset and clear errors
SerPut "*CLS"
SerPut ":DISP:TEXT 'READY'"

SerPut ":MEASURE:VOLTAGE:DC?"
SerGetNum V
PRINT "DC voltage = ", V
```

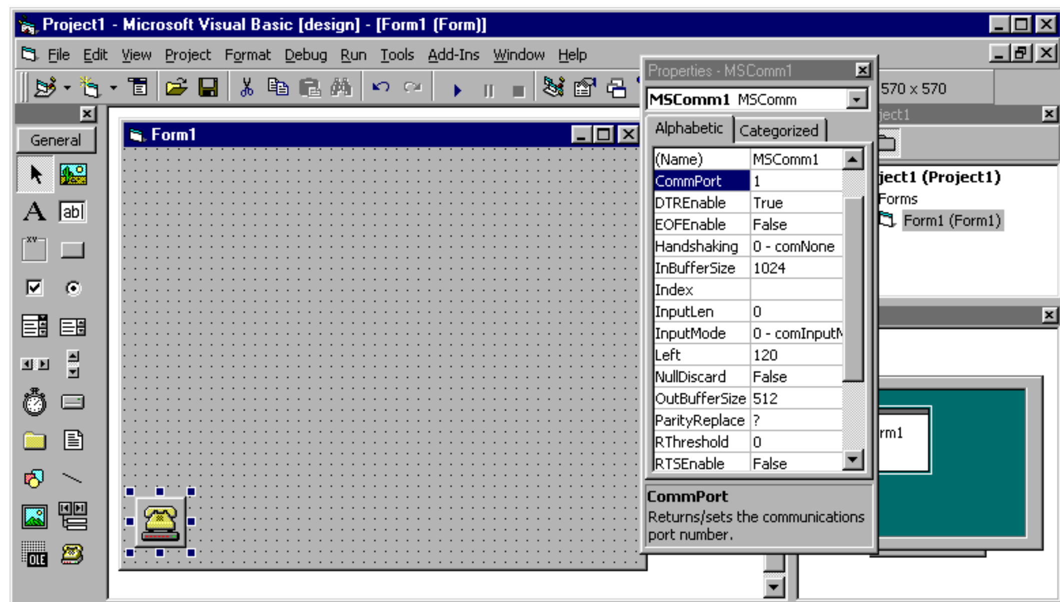
Everything else is the same.

5. Interfacing under Windows 95/98/2000/NT

Unlike DOS, Windows *does* provide buffering of incoming serial data, together with an elaborate application program interface (API) for serial communication. This API is so elaborate that many Windows programming environments encapsulate it somehow to make accessing it easier. In what follows, we describe three ways to do serial communication under Windows.

5.1. Using Visual Basic's MSCOMM object

In Visual Basic, the serial port is encapsulated in the MSCOMM component, which you can place on your application's main form. It looks like a telephone:



If MSCOMM is not shown on your palette, right-click on a blank area of the palette, select "Components," and check "Microsoft Comm Control" or, if that is not on the menu, search for MSCOMM32.OCX.

Once MSCOMM is there, the program code to access it is very simple. Here is an example. In this program, the MSCOMM object is named *MSComm1.* Note that much of the program code is almost identical to the preceding QBASIC example. The main program – actually the event procedure for clicking a button – is at the end.

Unfortunately, *MSCOMM does not support plain DTR handshaking.* Thus, although it works fine with the Agilent54645D oscilloscope, it cannot handshake with newer Agilent instruments. Most of the time, you can simply turn handshaking off (`MSComm1.Handshaking = comNone`) and proceed. Sometimes you may have to introduce delays in your program in order not to transmit to an instrument when it isn't ready. As an alternative to MSCOMM, Visual Basic can also call the DLL described later in this report.

```
' Visual Basic example code - M. Covington - 1999

Sub SerMessage(msg$)
' Displays a message to the user.
  MsgBox msg$
End Sub

Sub SerCrash(msg$)
' Displays a message and ends the program.
  SerMessage msg$
  End   ' Forces program to end immediately.
End Sub

Sub SerOpen(portnum%)
' Opens the serial port.
' portnum% = 1 for COM1, 2 for COM2, etc.
' Requires an MSComm control on the main form.
  MSComm1.CommPort = portnum%
  MSComm1.Settings = "9600,N,8,1"
  MSComm1.Handshaking = comRTS
  MSComm1.InputLen = 1  ' read one byte at a time
  MSComm1.PortOpen = True
  If Not MSComm1.PortOpen Then
    SerCrash "Cannot open port COM" + Str$(portnum)
  End If
  ' SerMessage "The serial port is open"
End Sub

Sub SerClose()
' Closes the serial port
  MSComm1.PortOpen = False
  ' SerMessage "The serial port is closed"
End Sub

Sub SerPut(data$)
' Transmits a string, followed by CR and LF
  MSComm1.Output = data$ + Chr$(13) + Chr$(10)
End Sub

Sub SerGetChar(c$)
' Retrieves a character from the serial port,
' or times out, returning an empty string,
' if nothing is received for 10 s.
  start = Timer ' Seconds elapsed since midnight.
  While Timer - start < 10
    If (Timer - start) < 0 Then
      ' midnight rollover occurred
      start = 0
    End If
    If MSComm1.InBufferCount > 0 Then
      c$ = MSComm1.Input
      Exit Sub
    End If
  Wend
  c$ = "" ' timed out, return empty string
End Sub

Sub SerGet(data$)
' Inputs a line as a string of characters
  data$ = ""
  ' Eat any Return or Line Feed characters
  ' left over from end of previous line
  Do
    SerGetChar c$
  Loop Until (Len(c$) = 0) Or (Asc(c$) > 31)
```

```
    ' Read characters until Line Feed, Return,
    ' or timeout occurs

  Do
      If c$ = Chr$(10) Then Exit Sub
      If c$ = Chr$(13) Then Exit Sub
      If c$ = "" Then Exit Sub
      data$ = data$ + c$
      SerGetChar c$  ' the next one
    Loop
End Sub

Sub SerGetNum(x As Variant)
' Inputs a number from the serial port.
' No error checking.
  SerGet data$
  x = Val(data$)
End Sub

Private Sub Command1_Click()
' Demonstrates communication with Agilent 54645D oscilloscope.
  SerOpen 1
  SerPut "*RST"       ' Reset it and clear errors
  SerPut "*CLS"
  SerPut ":SYSTEM:DSP 'Communicating with computer'"
  SerPut ":AUTOSCALE"
'
  SerPut ":MEASURE:VRMS? ANALOG1"
  SerGetNum vrms
  Print "Analog channel 1 V(rms) = ", vrms
'
  SerPut ":MEASURE:VPP? ANALOG2"
  SerGetNum vpp2
  Print "Analog channel 2 V(p-p) = ", vpp2
'
  SerClose
End Sub
```

5.2. Using the Win32 API in C++

You can also use the Windows API directly. This is most easily done from C or C++, and the following example was developed in Borland C++ Builder. This happens to be a *console application* – that is, it does not use windowing – but all the C functions can be called equally well from a windowed program. Although compiled in C++, this is, for all practical purposes, a C program; the only C++ extensions used are `cin` and `cout` in the main program.

```
// AGILENTSCOPE.CPP - Borland C++ Builder - M. Covington 1999 Dec 1

#include <condefs.h>   // Main pgm is a console application

#include <time.h>      // Used when detecting timeouts
#include <windows.h>   // We use some Windows system calls
#include <iostream.h>  // We use cout, etc.

HANDLE   hSerPort;     // Global variables
int      iSerOK;

void SerMessage(char* msg1, char* msg2)
// Displays a message to the user.
// msg1 = main message
// msg2 = message caption
// Change this to your preferred way of displaying msgs.
{
  MessageBox(NULL,msg1,msg2,MB_OK);
```

```
}

void SerCrash(char* msg1, char* msg2)
// Like SerMessage, but ends the program.
{
  SerMessage(msg1,msg2);
  iSerOK = 0;
}

int SerOK()
// Returns nonzero if most recent operation succeeded,
// or 0 if there was an error
{
  return iSerOK;
}

void SerOpen(char* portname, char* handshake)
// Opens the serial port.
// portname = "COM1", "COM2", etc.
// handshake = "RTS" or "DTR"
{
  //
  // Open the port using a handle
  //
  hSerPort = CreateFile(
    portname,GENERIC_READ|GENERIC_WRITE,0,NULL,
    OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
  if (hSerPort == INVALID_HANDLE_VALUE)
    SerCrash("Can't open comm port",portname);
  //
  // Set baud rate and other attributes
  //
  DCB dcbSerPort;
  GetCommState(hSerPort,&dcbSerPort);
  dcbSerPort.BaudRate = CBR_9600;
  dcbSerPort.fBinary = TRUE;
  dcbSerPort.fParity = FALSE;
  // Both handshaking modes are sensitive to DSR and DTR...
  dcbSerPort.fDsrSensitivity = TRUE;
  dcbSerPort.fDtrControl = DTR_CONTROL_HANDSHAKE;
  dcbSerPort.fOutxDsrFlow = TRUE;
  // .. but not XON/XOFF.
  dcbSerPort.fOutX = FALSE;
  dcbSerPort.fInX = FALSE;
  // Now choose the handshaking mode...
  if ((handshake[0] == 'r') || (handshake[0] == 'R'))
  {
    dcbSerPort.fOutxCtsFlow = TRUE;
    dcbSerPort.fRtsControl = RTS_CONTROL_HANDSHAKE;
  }
  else
  {
    dcbSerPort.fOutxCtsFlow = FALSE;
    dcbSerPort.fRtsControl = RTS_CONTROL_ENABLE;
  }
  dcbSerPort.fNull = FALSE;
  dcbSerPort.fAbortOnError = FALSE;
  dcbSerPort.ByteSize = 8;
  dcbSerPort.Parity = NOPARITY;
  dcbSerPort.StopBits = ONESTOPBIT;
  iSerOK = SetCommState(hSerPort,&dcbSerPort);
  if (!iSerOK) SerCrash("Bad parameters for port",portname);
  //
  // Disable Windows' timeouts; we keep track of our own
  //
```

```
COMMTIMEOUTS t = {MAXDWORD,0,0,0,0};
  SetCommTimeouts(hSerPort,&t);
}

void SerClose()
// Closes the serial port.
{
  iSerOK = CloseHandle(hSerPort);
  if (!iSerOK) SerCrash("Problem closing serial port","");
}

void SerPut(char* data)
// Transmits a line followed by CR and LF.
// Caution!  Will wait forever, without registering an error,
// if handshaking signals tell it not to transmit.
{
  DWORD nBytes;
  iSerOK  = WriteFile(hSerPort,data,strlen(data),&nBytes,NULL);
  iSerOK &= WriteFile(hSerPort,"\r\n",2,&nBytes,NULL);
  if (!iSerOK) SerCrash("Problem writing to serial port","");
}

void SerGetChar(char* c, int* success)
// Receives a character from the serial port,
// or times out after 10 seconds.
// success = 0 if timeout, 1 if successful
{
  time_t finish;
  finish = time(NULL) + 10;
  *success = 0;
  DWORD nBytes = 0;
  while ((*success == 0) && (time(NULL) < finish)) {
    ReadFile(hSerPort,c,1,&nBytes,NULL);
    *success = nBytes;
  }
  if (*success == 0)
    SerCrash("Timed out waiting for serial input","");
}

void SerGet(char* buf, int bufsize)
// Inputs a line from the serial port, stopping at
// end of line, carriage return, timeout, or buffer full.
{
  int bufpos = 0;
  buf[bufpos] = 0;     // initialize empty string
  int bufposmax = bufsize-1;  // maximum subscript
  char c = 0;
  int success = 0;
  //
  // Eat any Return or Line Feed characters
  // left over from end of previous line
  //
  do { SerGetChar(&c,&success); }
  while ((success == 1) && (c <= 31));
  //
  // We have now read the first character or timed out.
  // Read characters until any termination condition is met.
  //

while (1) {
    if (success == 0) return;          // timeout
    if (c == 13) return;               // carriage return
    if (c == 10) return;               // line feed
    if (bufpos >= bufposmax) return;   // buffer full
```

```
     buf[bufpos] = c;
     bufpos++;
     buf[bufpos] = 0;  // guarantee validly terminated string
     SerGetChar(&c,&success);
   }
}

void SerGetNum(double* x)
// Reads a floating-point number from the serial port.
{
    char buf[20];
    SerGet(buf,sizeof(buf));
    iSerOK = sscanf(buf,"%lf",x);   // "%lf" = "long float" = double
    if (iSerOK < 1) {
      SerCrash("Problem converting string to number",buf);
      *x = 0.0;
    }
}

int main()
{
 char b[20];
 double x;

 // Sample dialogue for Agilent 54645D oscilloscope
 SerOpen("COM1","RTS");   // note RTS handshaking
 SerPut("*RST");
 SerPut("*CLS");
 SerPut(":SYSTEM:DSP 'Communicating with computer'");
 SerPut(":AUTOSCALE");
 SerPut(":MEASURE:VRMS? ANALOG1");
 SerGetNum(&x);
 cout << "Channel 1 RMS voltage (as number) = " << x << "\n";
 SerPut(":MEASURE:VPP? ANALOG2");
 SerGet(b,sizeof(b));
 cout << "Channel 2 P-P voltage (as string) = " << b << "\n";

 // Say goodbye
 cout << "Press Enter...";
 cin.getline(b,1);
 SerClose();
 return 0;
}
```

To communicate with an Agilent 34401A multimeter using DTR handshaking, the relevant lines of the main program can be changed to:

```
 SerOpen("COM1","DTR");   // note DTR handshaking
 SerPut(":SYST:REM");
 SerPut("*RST");
 SerPut("*CLS");
 SerPut(":DISP:TEXT 'OK'");
 SerPut(":MEASURE:VOLT:DC?");
 SerGetNum(&x);
 cout << "Voltage (as number) = " << x << "\n";
 SerPut(":MEASURE:VOLT:DC?");
 SerGet(b,sizeof(b));
 cout << "Voltage (as string) = " << b << "\n";
```

5.3. Encapsulating the interface in a DLL

A DLL (Dynamic Link Library) is the simplest type of Windows software component. It is a file containing pre-compiled functions that can be loaded and called by a running program. Much of the Windows operating system resides in DLLs.

The University of Georgia distributes, from *ftp://ftp.ai.uga.edu/pub/serial.port,* a DLL containing the C functions described in the previous section. In this section we'll describe how the DLL is constructed, and in the next section, how to use it. To build a DLL, you first tell your C compiler that you want to generate a DLL rather than an EXE file. Most compilers automatically generate the `DllEntryPoint` function when you do this. Then you put in the functions that you want to place into the DLL.

The first line of each function requires some additional declarations that are not used in ordinary C. Each function is declared `_export` so it will be externally callable, and `extern "C"` so its name in the DLL will be the same as its name in the source code (otherwise we'd'have trouble finding it). In addition, we declare all the routines `_stdcall` so that they will use the Windows standard calling sequence rather than the C calling sequence, making it easier to call them from other languages. The order of these declarations is significant; `_export` must come right before the function name.

Note! When compiling your DLL, you must make sure it does not require any other DLLs other than those that come with Windows. In C++ Builder 4, the way to do this is to go to Project, Options, Linker, and uncheck "Use dynamic RTL"; and go to Project, Options, Packages and uncheck "Build with runtime packages."

The source code for UGASER01.DLL is as follows. A more sophisticated version named UGASER02.DLL will be developed later.

```
// UGASER01.CPP - Borland C++ Builder - M. Covington 1999 Dec 1
// Source code for UGASER01.DLL

#include <vcl.h>
#include <time.h>       // Used when detecting timeouts
#include <windows.h>    // We use some Windows system calls
#include <iostream.h>   // We use cout, etc.

HANDLE   hSerPort;      // Global variables, not exported
int      iSerOK;

extern "C" void _stdcall _export SerMessage(char* msg1, char* msg2)
// Displays a message to the user.
// msg1 = main message
// msg2 = message caption
// Change this to your preferred way of displaying msgs.
{
  MessageBox(NULL,msg1,msg2,MB_OK);
}

extern "C" void _stdcall _export SerCrash(char* msg1, char* msg2)
// Like SerMessage, but ends the program.
{
  SerMessage(msg1,msg2);
  iSerOK = 0;
}

extern "C" int _stdcall _export SerOK()
// Returns nonzero if most recent operation succeeded,
// or 0 if there was an error
{
  return iSerOK;
}
```

...all the other function definitions, with `extern "C"` *and* `_stdcall _export` *added to their declarations, and finally...*

```
// The following makes the DLL identify itself:
//
int WINAPI DllEntryPoint(HINSTANCE hinst,
                    unsigned long reason, void*)
{
```

```
        return 1;
}
```

Compile it appropriately, and you have a DLL.

5.4. Calling the functions in a DLL

In order to be called from your program, a DLL must reside either in the same directory as the program, or in `\windows\system`,[1] or in a directory listed in the PATH environment variable.

It must also have a different name than any other DLL anywhere on the computer. That is a potential pitfall, because if it doesn't, you won't get an error message; you may just get the wrong DLL loaded into memory. Name collisions between unrelated DLLs are rare; one memorable example is that there were two completely different TCP packages both called WINSOCK.DLL. More common are collisions between different "versions" of the same DLL. In our opinion, Microsoft should have provided a way to guarantee the uniqueness of DLL names. Since they did not, the best we can do is start all our DLL names with UGA (for *University of Georgia*) in the hope that no one else will choose the same three letters.

There are two ways to link the DLL to the calling program, *statically* (at compile time) or *dynamically* (during execution). Only the latter will be described here.

5.4.1. Calling from C or C++

Normally, of course, you won't use a DLL to call our C routines from C or C++. However, it's useful to illustrate how to do so, simply because it shows you exactly what system calls are involved.

Basically, the calling program must:

(1) load the DLL;

(2) find the desired function in the DLL, and obtain a pointer to it;

(3) call it using this pointer;

(4) free the DLL when it's finished with it.

Of these, step (3) is the hard one in C because C is a typed language; you must therefore declare a pointer type that matches the function you are calling.

The following is an example of calling a DLL dynamically from C++. *Caution: You will not get recognizable error messages if you call a function with the wrong type or number of arguments.*[2] You will probably get messages about stack overflow or underflow or incorrect calling sequence – or you may not get any error messages at all.

```
// File DemoDLL.CPP - M. Covington 1999
// Demonstrates calling a dynamically loaded DLL

#include <condefs.h>     // This is a console application
#include <iostream.h>    // We're using cout
#include <windows.h>     // We're using Windows system calls

// Because C is a typed language, we need a different
// pointer type for each combination of function/argument types.
// For convenience, we make each of them a type unto itself.

typedef void _stdcall TSerMessage(char*,char*);
typedef void _stdcall TSerCrash(char*,char*);
typedef int  _stdcall TSerOK();
typedef void _stdcall TSerOpen(char*,char*);
typedef void _stdcall TSerClose();
typedef void _stdcall TSerPut(char*);
typedef void _stdcall TSerGetChar(char*,int*);
typedef void _stdcall TSerGet(char*,int);
typedef void _stdcall TSerGetNum(double*);

int main()
```

```
{
  cout << "Finding and loading the DLL..." << endl;
  HINSTANCE h = LoadLibrary("UGASER01.DLL");
  if (h == NULL) {
    cout << "Could not load the DLL." << endl;
    return -1;
  }

  cout << "Let's try out SerMessage..." << endl;
  TSerMessage* pSerMessage =
        (TSerMessage*)GetProcAddress(h,"SerMessage");
  if (pSerMessage == NULL) {
    cout << "Could not find SerMessage in the DLL" << endl;
    return -1;
  }
  (*pSerMessage)("Greetings from a DLL!","Hi!");

  cout << "Freeing the DLL..." << endl;
  FreeLibrary(h);

  return 0;
}
```

If you're not adept at using the C pointer system, this may seem a little obscure. Here's what's going on in the middle part: `TSerMessage` is the type we define for `SerMessage` (i.e., a function that takes two char* arguments and returns nothing). We cast the result of `GetProcAddress` to `TSerMessage*` (i.e., pointer to an object of type `TSerMessage`) and assign it to `pSerMessage` (the variable that holds the actual pointer). We then call `(*pSerMessage)` (i.e., the function to which `pSerMessage` points). There are plenty of other ways to do this, of course!

5.4.2. Calling from LPA Prolog

Calling functions in a DLL from other languages is often easier. Here we call some of the DLL functions from LPA Prolog (a product of Logic Programming Associates, London, England, *www.lpa.co.uk*). LPA Prolog provides a built-in routine for calling Windows system functions, but it can actually call any function in any DLL as long as the arguments are 32-bit integers or 32-bit pointers.

Crucially, `LoadLibrary` is actually `LoadLibraryA` because we are passing it a pointer to an ASCII (8-bit-per-character) character string. There is also a `LoadLibraryW` for wide (16-bit, UNICODE) strings. The same is true of every Windows system call that takes string arguments. Some programming languages, including C, take care of this distinction for you, but in Prolog you're on your own.

The bare minimum Prolog code to call our DLL is the following:

```
test :-
   winapi((kernel32,'LoadLibraryA'),[`ugaser01.dll`],Handle),
   winapi((ugaser01,'SerMessage'),[`Greetings`,`Hello`],_),
   winapi((kernel32,'FreeLibrary'),[Handle],_).
```

LPA's documentation explains in more detail how `winapi/3` works. Basically, the schema is

`winapi(`*DLLname*`,` *Function*`)`,*ArgumentList*, *ReturnedValue*`)`

where *DLLname* and *Function* are Prolog atoms, *ArgumentList* is a list (possibly empty) of arguments, and *ReturnedValue* is a Prolog variable to hold the returned value, or an anonymous variable (underscore mark, "_") to discard it. In addition, `wintxt`, used in the following program, provides a way to retrieve text from buffers passed to the called function.

Here is a more sophisticated Prolog program that conducts a dialogue with an Agilent 54645D oscilloscope and an Agilent 34401A multimeter. The C functions are wrapped in Prolog predicates that fail (i.e., cause backtracking) whenever they cannot carry out the requested operation. Thus, Prolog backtracking will automatically bail out of unsuccessful situations.

```
% File DEMO2.PL - M. Covington 1999
% Dialogue with Agilent instruments in LPA Prolog


% Prolog routines to call the C functions.
%
%  These are arranged so that they fail if the operation
%  is unsuccessful.  Thus, Prolog backtracking will
%  automatically bail out of an unsuccessful operation.

:- dynamic serHandle/1.    % global data


% serOK -- succeeds if previous serial operation was successful.

serOK :-
  winapi((ugaser01,'SerOK'),[],Result),
  Result > 0.


% serOpen(+Port,+Handshake)
%                  -- loads the DLL and open the serial port.
%                     Port = `COM1` or the like.
%                     Handshake = `RTS` or `DTR`.

serOpen(Port,Handshake) :-
  winapi((kernel32,'LoadLibraryA'),[`ugaser01.dll`],Handle),
  asserta(serHandle(Handle)),
  winapi((ugaser01,'SerOpen'),[Port,Handshake],_),
  serOK.


% serClose -- close the serial port and free the DLL.
%             (Succeeds even if unable to do so, because
%             in general we want to clean up as best we can.)

serClose :-
  winapi((ugaser01,'SerClose'),[],_),
  retract(serHandle(Handle)),
  winapi((kernel32,'FreeLibrary'),[Handle],_).


% serPut(+String) -- send a string out the serial port.
%                    String is a backquoted string.

serPut(String) :-
  winapi((ugaser01,'SerPut'),[String],_),
  serOK.


% serGet(-String) -- read a string in from the serial port.
%                    String is instantiated to a backquoted string.

serGet(String) :-
  fcreate(buffer,[],-2,256),
  winapi((ugaser01,'SerGet'),[buffer,256],_),
  wintxt(buffer,0,String),
  fclose(buffer),
  serOK.


% serGetNum(-Num) -- read a string in from serial port as number.
%                    Num is instatiated to a floating-point number.
%                    (Because LPA Prolog can't retrieve floats from
```

```
%                    the DLL, we retrieve text and convert it here.)

serGetNum(Num) :-
  serGet(String),             % Error checking is done by SerGet.
  number_string(Num,String).



% Main program  -- Simple dialog with Agilent 54645D oscilloscope
%                  and Agilent 34401A multimeter

test1 :- serOpen(`COM1`,`RTS`),    % For Agilent 54645D oscilloscope
         serPut(`*RST`),
         serPut(`*CLS`),
         serPut(`:SYSTEM:DSP 'Hello, world'`),
         serPut(`:AUTOSCALE`),
         serPut(`:MEASURE:VRMS? ANALOG1`),
         serGet(S),
         write('Channel 1 RMS voltage (as text) = '), write(S), nl,
         serPut(`:MEASURE:VRMS? ANALOG1`),
         serGetNum(N),
         write('Channel 1 RMS voltage (as number) = '), write(N), nl,
         serClose,
        !.

test1 :- serClose.   % fall back to here if anything failed along the way



test2 :- serOpen(`COM1`,`DTR`),    % For Agilent 34401A multimeter
         serPut(`:SYST:REM`),
         serPut(`*RST`),
         serPut(`*CLS`),
         serPut(`:DISP:TEXT 'READY'`),
         serPut(`:MEASURE:VOLT:DC?`),
         serGetNum(V),
         write('Voltage (as number) = '), write(V), nl,
         serClose,
         !.

test2 :- serClose.   % fall back to here if anything failed along the way
```

6. Interfacing with C under UNIX

Like Windows and unlike DOS, UNIX handles the serial port so that you can read and write it like a file. The UNIX API is somewhat simpler than that of Windows, particularly if you don't want to analyze communication errors. We have been assuming throughout that the user of an instrument will know whether or not it's working and that elaborate error handling is unnecessary.

The method for setting handshaking varies a good bit from one UNIX system to another. In practice, Agilent instruments do not necessarily require handshaking; at 9600 baud, if transmitting only modest amounts of data, you may well be able to do without it.

The following example program, in C, was tested under Linux on an Intel Pentium-based computer. Since everything except the handshaking selection is POSIX-compliant, it should be very portable.

```
// agilentscope.c - GNU g++ egcs-2.91.66, V. Bancroft  1999

#include <stdio.h>     // ISO C Standard: 4.9 INPUT/OUTPUT
#include <time.h>      // ISO C Standard: 4.12 DATE and TIME
#include <termios.h>   // POSIX Standard: 7.1-2 General Terminal Intf.
#include <unistd.h>    // POSIX Standard: 2.10 Symbolic Constants
#include <fcntl.h>     // POSIX Standard: 6.5 File Control Operations
#include <sys/types.h> // POSIX Standard: 2.6 Primitive System Data Types
```

```
#include <sys/stat.h>  // POSIX Standard: 5.6 File Characteristics

// map DWORD to the GNU variant for read and write sizes
#define DWORD                size_t
// a file handle is an integer descriptor,
#define HANDLE               int
// open failure results in negative one,
#define INVALID_HANDLE_VALUE -1
// default device string
#define DEVICE               "/dev/ttyS0"
// define values for iSerOK
#define TRUE                 1
#define FALSE                0

HANDLE   hSerPort;     // Global variables
int      iSerOK;

void SerMessage(char* msg1, char* msg2)
// Displays a message to the user.
// msg1 = main message
// msg2 = message caption
// Change this to your preferred way of displaying msgs.
{
  printf( "%s ( %s )\n", msg1, msg2);
}

void SerCrash(char* msg1, char* msg2)
// Like SerMessage, but ends the program.
{
  SerMessage(msg1,msg2);
  iSerOK = 0;
}

int SerOK()
// Returns nonzero if most recent operation succeeded,
// or 0 if there was an error
{
  return iSerOK;
}

void SerOpen(char* portname, char* handshake)
// Opens the serial port.
// portname = "COM1", "COM2", etc.
// handshake = "RTS" or "DTR"
{
  //
  // Open the port using a handle
  //
  hSerPort = open( portname, O_RDWR );
  if (hSerPort == INVALID_HANDLE_VALUE) {
    SerCrash("Can't open comm port",portname);
    iSerOK = FALSE;
  }
  termios term;
  if ( tcgetattr( hSerPort, &term ) ) {
    SerMessage( "Can't get device settings.", "tcgetattr");
    iSerOK = FALSE;
  }

  // Now choose the handshaking mode...
  if ((handshake[ 0] == 'r') || (handshake[ 0] == 'R'))
  {
    term.c_cflag =  CRTSCTS;    // RTS/CTS flow control of output
  }
  else
```

```
  {
    term.c_cflag &=  ~( CRTSCTS ); // disable CTS flow control of output
  }

  //
  // Set baud rate and other attributes
  //
  term.c_cflag |=   CS8;       // eight bit character set
  term.c_cflag |=  CREAD;     // enable receiver
  term.c_cflag |=  HUPCL;     // lower modem lines on last close
  term.c_cflag |=  CLOCAL;    // ignore modem status lines
  term.c_iflag = 0;           // turn off input processing
  // term.c_iflag = IGNPAR;      // ignore parity errors
  term.c_lflag = 0;           // turn off local processing
  term.c_oflag = 0;           // turn off output processing


cfsetispeed( &term, B9600 );
  cfsetospeed( &term, B9600 );
  if (tcsetattr( hSerPort, TCSANOW, &term )) {
    SerCrash("Bad parameters for port",portname);
    iSerOK = FALSE;
  }

  if (!iSerOK) SerCrash("Bad parameters for port",portname);
  //
  // Disable Windows' timeouts; we keep track of our own
  //
  // COMMTIMEOUTS t = {MAXDWORD,0,0,0,0};
  // SetCommTimeouts(hSerPort,&t);
}

void SerClose()
// Closes the serial port.
{
  iSerOK != close(hSerPort);
  if (!iSerOK) SerCrash("Problem closing serial port","");
}

void SerPut(char* data)
// Transmits a line followed by CR and LF.
// Caution!  Will wait forever, without registering an error,
// if handshaking signals tell it not to transmit.
{
  DWORD nBytes;
  iSerOK =  nBytes=write(hSerPort,(const void *)data,strlen(data));
  iSerOK &= nBytes=write(hSerPort,"\r\n",2);

  if (!iSerOK) SerCrash("Problem writing to serial port","");
}

void SerGetChar(char* c, int* success)
// Receives a character from the serial port,
// or times out after 10 seconds.
// success = 0 if timeout, 1 if successful
{
  time_t finish;
  finish = time(NULL) + 10;
  *success = 0;
  DWORD nBytes = 0;
  while ((*success == 0) && (time(NULL) < finish)) {
    nBytes=read(hSerPort,c,(size_t)1);
    *success = nBytes;
  }
  if (*success == 0)
```

```
    SerCrash("Timed out waiting for serial input","");
}

void SerGet(char* buf, int bufsize)
// Inputs a line from the serial port, stopping at
// end of line, carriage return, timeout, or buffer full.
{
  int bufpos = 0;
  buf[bufpos] = 0;     // initialize empty string
  int bufposmax = bufsize-1;  // maximum subscript
  char c = 0;
  int success = 0;

//
  // Eat any Return or Line Feed characters
  // left over from end of previous line
  //
  do { SerGetChar(&c,&success); }
  while ((success == 1) && (c <= 31));
  //
  // We have now read the first character or timed out.
  // Read characters until any termination condition is met.
  //
  while (1) {
    if (success == 0) return;         // timeout
    if (c == 13) return;              // carriage return
    if (c == 10) return;              // line feed
    if (bufpos >= bufposmax) return;  // buffer full
    buf[bufpos] = c;
    bufpos++;
    buf[bufpos] = 0;  // guarantee validly terminated string
    SerGetChar(&c,&success);
  }
}

void SerGetNum(double* x)
// Reads a floating-point number from the serial port.
{
   char buf[20];
   SerGet(buf,sizeof(buf));
   iSerOK = sscanf(buf,"%lf",x);   // "%lf" = "long float" = double
   if (iSerOK < 1) {
     SerCrash("Problem converting string to number",buf);
     *x = 0.0;
   }
}

// Sample main program to conduct dialogue
// with Agilent 54645D oscilloscope,
// 9600 baud, DTR handshaking

int main()
{
 char b[20];
 double x;

 // Sample dialogue for Agilent 64645D oscilloscope

 SerOpen("COM1","RTS");   // note RTS handshaking
 SerPut("*RST");
 SerPut("*CLS");
 SerPut(":SYSTEM:DSP 'Communicating with computer'");
 SerPut(":AUTOSCALE");
 SerPut(":MEASURE:VRMS? ANALOG1");
 SerGetNum(&x);
```

```
 printf( "Channel 1 RMS voltage (as number) = %d\n", x );
 // cout << "Channel 1 RMS voltage (as number) = " << x << "\n";
 SerPut(":MEASURE:VPP? ANALOG2");
 SerGet(b,sizeof(b));
 printf( "Channel 2 P-P voltage (as string) = %s\n", b );
 // cout << "Channel 2 P-P voltage (as string) = " << b << "\n";

/*
// Sample dialogue for Agilent 34401A multimeter
 SerOpen("COM1","DTR");   // note DTR handshaking
 SerPut(":SYST:REM");
 SerPut("*RST");
 SerPut("*CLS");
 SerPut(":DISP:TEXT 'OK'");
 SerPut(":MEASURE:VOLT:DC?");
 SerGetNum(&x);
 printf( "Voltage (as number) = %d\n", x );
 // cout << "Voltage (as number) = " << x << "\n";
 SerPut(":MEASURE:VOLT:DC?");
 SerGet(b,sizeof(b));
 printf( "Voltage (as string) = %s\n", b );
 // cout << "Voltage (as string) = " << b << "\n";
*/

 // Say goodbye

 printf( "Press Enter...");
 scanf("\r");

 SerClose();
 return 0;
}
```

7.0. Some ideas for exploiting computer-to-instrument communication

Once you can combine the intelligence of a general-purpose computer with the input and output versatility of Agilent test equipment, a wide range of new applications becomes possible. Here, in no particular order, are a few ideas for possible projects, ranging from simple to complex.

- Build a “talking instrument” interfaced to a PC with a speech synthesizer.
- Test, cycle, and recharge NiCd batteries under computer control; report when a battery seems to be defective.
- Monitor the quality of the 120-volt AC line for a whole day or more.
- Compute the statistics of a set of components (such as a collection of resistors that are supposed to be the same value).
- Build an expert system in Prolog that diagnoses failures in a piece of equipment, telling the technician where to connect the probes at each step.
- Build an expert system that learns – that is, you use it to make measurements on one or more known-good pieces of equipment under varying conditions, and it learns to distinguish normal from abnormal measurements.
- Why stop at PCs? Using a microcontroller and a MAX-232 chip, you can build a small module that can deliver elaborate commands to a piece of test equipment. This allows even more customization than Agilent provided for.

Those are just a few of many possibilities. Bear in mind that if multiple Agilent instruments are involved, you should consider using GPIB (multiple instruments on one bus) rather than serial ports.

-end-

[1] More precisely, the directory returned by calling `GetSystemDirectory`; in Windows NT/2000, normally `\windows\system32`.
[2] As Cartic Ramakrishnan pointed out to us, having found it out the hard way.