16-Bit Serial Communications

by John Chaytor

This article describes asynchronous serial communications in the 16-bit Windows environment. We will briefly cover the nuts and bolts at the hardware level, to understand the terminology used in the Windows API, then I'll discuss in detail how an application opens, configures, initialises, communicates with and closes a serial device. Although the serial port can be connected to various devices (mouse, joystick, printer etc) we'll assume that we are communicating with a modem: ie two way, full duplex data transfer. By the end of the article we will have created a class to encapsulate the `Comm` device and a simple demo application.

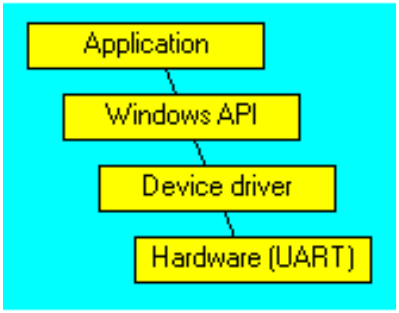
32-bit comms under Windows is quite a bit different, but don't worry: I have promised the Editor I'll write a follow-up article covering 32-bit issues, and hopefully the ZMODEM protocol too! The basic hardware discussion applies to both 16-bit and 32-bit, of course.

System View

Figure 1 illustrates the relationship between an application and the underlying hardware. As you can see, application code interfaces to serial devices via the various Windows APIs. In turn, Windows relies on a device driver (by default COMM.DRV) which performs all the low-level work needed to talk to the hardware. The serial hardware is a special chip called a UART (Universal Asynchronous Receiver/Transmitter) which is responsible for communicating with the outside world via the 9 pin cable plugged into your PC's COM port.

Hardware View

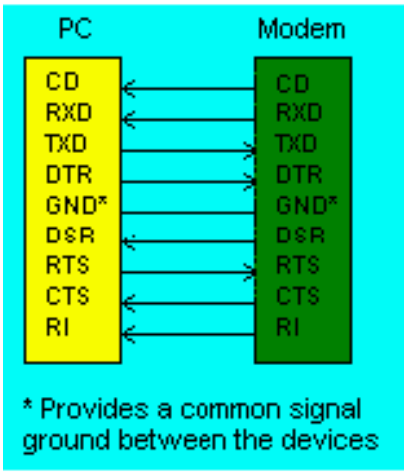
Table 1 details the names used for the nine lines (pins) in serial communications. When a cable is connected between a serial port and a modem the data flow for each line is in a single direction. Figure 2 shows the direction of data flow for each line. At any time each line can



➤ *Figure 1*

be high (on) or low (off). As we are describing communications with a modem, these lines can be broken down into the following categories:

- ***Device control.*** DTR and DSR are used in controlling and monitoring the connection between the PC and modem. The modem sets DSR high when it is powered on and has initialised. DTR is set high when the serial port on the PC has been initialised. It must remain high as long as the modem connection is required. Most modems will hang up the connection if DTR is dropped low. We will use this to implement a hang-up method.
- ***Flow control.*** CTS and RTS are used for hardware flow control.
- ***Status information.*** RI and CD (also called DCD or RLSD) are status lines. RI is high whenever the modem detects an incoming ringing tone. CD is high when the modem is connected to another modem.
- ***Data transfer.*** RXD and TXD are the lines which carry the data between the UART chip and the modem. The TXD (transmit line) on the local PC is logically connected to the RXD (receive line) of the remote PC and vice versa. As there is only one line for each direction, the data needs to be sent 1 bit at a time – hence the name serial. The UART chip converts the data between the internal PC representation (eg a byte) and an external format (bit stream) required for data transmission. How it does this depends on the configuration options specified for the `Comm` device.



➤ *Figure 2*

Pin	Acronym	Name
1	CD	Carrier detect
2	RXD	Received data
3	TXD	Transmitted data
4	DTR	Data terminal ready
5	GND	Ground
6	DSR	Data set ready
7	RTS	Request to send
8	CTS	Clear to send
9	RI	Ring indicator

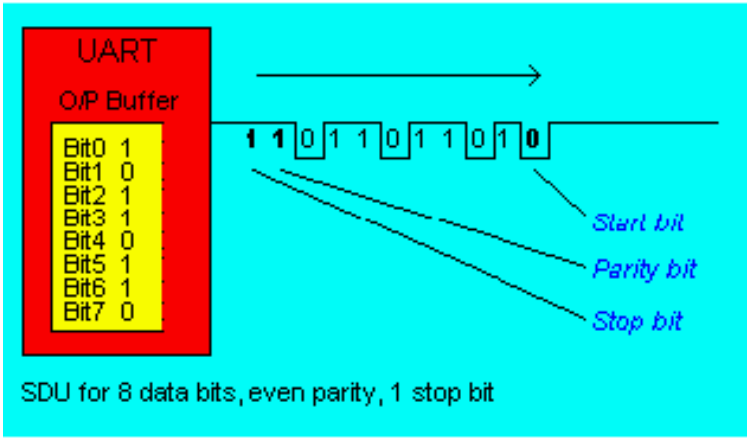
➤ *Table 1*

GND is of no interest to us: it's all to do with voltage levels!

Data Conversion

The data passed over a serial link needs to be converted from the internal PC format to a stream of bits. The UART chip does this conversion. Figure 3 illustrates the bit stream generated if `$6D` were written to the UART output buffer. If you look closely at the figure you will see that there are more bits leaving the UART chip than are placed in its output buffer.

This is because extra bits (shown in bold) need to be transmitted along with the raw data for



➤ *Figure 3*

Parity	Meaning	
None	No parity bit is passed in the SDU	
Even	Parity bit is set so the number of 1s in the data bits and parity bit is even	
Odd	Parity bit is set so the number of 1s in the data bits and parity bit is odd	
Mark	Parity bit is always set to 1	*These options can't detect if there have been errors in the data bits during transmission and are rarely used*
Space	Parity bit is always set to 0	

➤ *Table 2: Parity settings*

both ends of the communication link to interpret and validate the data correctly. These additional bits include synchronisation and parity bits. For each unit of data which is passed to the UART, up to 4 bits may be added.

This bit stream (containing 11 bits in the figure) is called a *Serial Data Unit* (SDU) and is the smallest data quantity processed by the serial interface. When the driver passes data to the UART chip it will generate an SDU of this format and send it out on the serial cable. If, for example, only a single character were sent to the UART it would send the SDU, then there would be a gap until the next character is sent. This is why serial communication is more accurately called asynchronous serial communication. Data can be sent or received at any time.

Due to the asynchronous nature of the data transmission, there needs to be a mechanism for the receiving end of the communication link to know when the data starts and ends. This is where the start and stop bits are used: they encapsulate every data unit transmitted. The start bit is always a single zero and is the first bit sent. Immediately after the start bit the UART sends the data bits, which were placed in its input buffer. The UART can be configured to process data units between 5 and 8 bits in length. The data bits may be followed by an optional parity bit. The value of the parity bit is generated by the UART chip and depends on the configuration option specified. Table 2 lists the possible options. Parity bits are of limited use. They can only accurately detect errors if there are errors in an odd number of data bits. If 2,4,6 or 8 data bits are in error, they cancel each other out and the parity bit suggests that the data was valid. As the majority of modems now use block CRC error checking and recovery you will often see that parity is not used for dial up services. After the parity bits have been sent the UART sends the stop bit(s) to indicate the end of the SDU. The end of the SDU is made up of 1, 1.5 or 2 stop bits (which are always '1' – high).

Whenever the UART has no data to send it keeps the TXD line high. If you refer to Figure 3 you will see that the line is high on both sides of the SDU. This fact is used by the receiving UART to detect when the next SDU has arrived. When it detects that its RXD line has dropped it assumes that an SDU start bit is being received and starts collecting the bits as defined by the current configuration. When the UART has collected the bits it processes the SDU and will place the data in its received buffer (assuming no errors were detected). At this point the UART will wait until another start bit is detected and start the process again.

You should be able to see from this brief hardware overview why it is essential for both ends of the communication link to specify the same configuration options for the data bits, parity and stop bits. If they are not the same the receiving UART will mis-interpret the bit pattern and hardware errors will be generated.

Hardware Detected Errors

The following errors are automatically detected by the hardware. See the later section *Detecting Errors/Events* to discover how an application can be informed when these errors occur.

- ***Framing error.*** The UART has detected an invalid stop bit. After reading the SDU start, data and optional parity bits it was expecting to find the '1' stop bit(s), but instead it found a '0'.
- ***Overrun error.*** The UART has received an SDU but its output buffer is full (the old style chips only have a 1 byte register, the newer chips have a 16 byte FIFO buffer). This happens if the driver has not processed the current received buffer, probably due to the application not reading data from the driver's buffers fast enough when flow control is not being used.
- ***Break error.*** The UART has detected that the RXD input line has been low for a longer time than it normally takes to receive an SDU. Even if it were receiving a byte of X'00' with even parity it should receive a stop bit in this time (and remember, the remote UART holds the line high if it is not sending data). This could indicate that the connection is broken or the transmitter has sent a break signal, which drops the TXD line.
- ***Parity error.*** After the SDU has been received successfully (ie none of the previous errors

were encountered), the UART performs any required parity checking. If the parity bit is incorrect this error will be generated. A configuration option allows you to replace these characters with the supplied character.

Baud Rate

Unlike data bits, parity and stop bits, the Baud rates for the communication devices at each end of a link can be different when using modems. The Baud rate you use to configure the communication devices determines the speed of data transfer between your PC and the attached modem (the modem automatically detects the values). When the two modems communicate with each other they negotiate a baud rate which can be supported by both ends (along with other parameters such as error recovery, compression etc).

If you connect two PCs using a NULL modem cable (which ensures that the pins at each end are connected correctly) you will need to set the baud rate to the same value at both ends.

Flow Control

Flow control is a mechanism used to ensure that the receiver's buffers do not overflow during data transfer leading to data loss. The basic mechanism behind flow control is that each receiver indicates to the transmitter (via an agreed on protocol) that it should stop transmitting data when its input buffer is in danger of overflowing. Once the receiver has processed some of the data in the buffer it then lets the transmitter know that it is now OK to send data. There are three options for flow control:

- ***None.*** Obviously, with no flow control you may lose data.
- ***Software.*** Special characters, XON and XOFF, are reserved to have special meaning. When a receiver detects that its buffers are nearly full it sends an XOFF character to the transmitter to indicate that it should stop sending data. Once the receiver has cleared the backlog it sends an XON character to indicate that the transmitter can now resume. XON/XOFF cannot be used when transferring binary data as the data may contain an XON/XOFF character, which would trigger a false flow control event. Also, it is only reliable at low transfer rates. Finally, there is a danger that one of the XON/XOFF characters could be corrupted during transmission and the transmitter will miss an important flow control instruction.
- ***Hardware.*** By far the best form of flow control is hardware flow control. In this, either the DTR/DSR or CTS/RTS pair of lines is used. For modems, CTS/RTS is used (the DSR/DTR lines are used for device control). When the transmitter needs to send data, it sets the RTS (request to send) line high. It then checks the CTS (clear to send) line. If this line is high it is OK to send data, if this line is low the receiver cannot accept the data: the transmitter must wait for the CTS line to go high.

Although we need to be aware of flow control, the driver software implements it on our behalf. We only need to configure the device for the chosen mechanism.

API	Description
BuildCommDCB	Builds a device control block from a device-definition string
ClearCommBreak	Restores character transmission after SetCommBreak
CloseComm	Closes a communications device
EnableCommNotification	Enables or disables posting of notification messages
EscapeCommFunction	Causes the communications device to carry out the extended function
FlushComm	Flushes the specified transmission or receiving queue
GetCommError	Retrieves the most recent error value and current status for the device
GetCommEventMask	Retrieves and then clears the event word for the communication device
GetCommState	Retrieves the device control block
OpenComm	Opens a communications device
ReadComm	Reads characters from a communications device
SetCommBreak	Suspends character transmission (TXD will go low)
SetCommEventMask	Enables events for the communication device
SetCommState	Sets the communications-device state using the device control block
TransmitCommChar	Places a character at the head of the transmission queue
UngetCommChar	Puts a character back in the receiving queue
WriteComm	Writes characters to a communications device

➤ *Table 3: Windows serial comms APIs*

Driver Configuration

Although I'm not going to cover this in detail you should be aware of the configuration options in the Windows control panel for the communication ports. The driver uses two parameters, Base I/O address and IRQ number, to know how to talk to the hardware. The Base I/O address refers to the address of the UART Port 0 in memory. This allows the driver to read and write to the ports on the UART chip. The IRQ number will be used to inform the driver when communication events occur. These two values need to be unique for the port to avoid conflicts with other hardware devices.

Software View: The TCommDevice Class

Now that we have a basic understanding of the terminology used in serial communications we will step through opening, configuring, initialising, communicating with and closing a communication device. The code examples shown in the listings are simplified examples of

the routines provided on the disk, to allow us to concentrate on the important areas.

Table 3 details the relevant APIs used in serial communications. Of those listed, BuildCommDCB, TransmitCommChar and UngetCommchar are not used in the TCommDevice class.

I've developed the class TCommDevice to show how to access the serial ports on your PC. The class is not 'industrial strength', but demonstrates the principles.

```
type
  TCommDevice = class
    private
      { ....see the disk for details }
    public
      Constructor Create;
      Destructor Destroy; override;
      { Methods }
      procedure BreakTransmission;  { ** rarely used ** }
      procedure Close;
      function  Dial(const Number: string): Boolean;
      function  FlushInput: Boolean;
      function  FlushOutput: Boolean;
      procedure HangUp;
      procedure Open(Port: Integer);
      procedure ResumeTransmission; { ** rarely used ** }
      function  WriteBlock(Buff: PChar; BuffLen: Integer): Boolean;
      function  WriteLn(const S: string): Boolean;
      { Properties which affect device operating parameters (ie affect the DCB) }
      property BaudRate: Word read FDCB.BaudRate write SetBaudRate;
      property CtsTimeOut: Word read FCtsTimeout write FCtsTimeout;
      property DataBits: Byte read FDCB.ByteSize write SetDataBits;
      property DsrTimeOut: Word read FDsrTimeout write FDsrTimeout;
      property FlowControl: TFlowControl read FFlowControl write FFlowControl;
      property Parity: Byte read FDCB.Parity write SetParity;
      property ParityCheck: Boolean read FParityCheck write FParityCheck;
      property ParityDoReplaceChar: Boolean
        read FParityDoReplaceChar write FParityDoReplaceChar;
      property ParityReplacementChar: Char
        read FParityReplacementChar write FParityReplacementChar;
      property ReceiveQueueSize: Integer
        read FReceiveQueueSize write SetReceiveQueueSize;
      property StopBits: Byte read FDCB.StopBits write SetStopBits;
      property TransmitQueueSize: Integer
        read FTransmitQueueSize write SetTransmitQueueSize;
      property XFlowOffLimit: Word read FXFlowOffLimit write FXFlowOffLimit;
      property XFlowOnLimit: Word read FXFlowOnLimit write FXFlowOnLimit;
      property XOffChar: Char read FXOffChar write FXOffChar;
      property XOnChar: Char read FXOnChar write FXOnChar;
      { Read only properties which show status information }
      property CDHigh: Boolean read GetCDHigh;
      property CTSHigh: Boolean read GetCTSHigh;
      property DeviceOpen: Boolean read GetDeviceOpen;
      property DSRHigh: Boolean read GetDSRHigh;
      property InputByteCount: Integer read GetInputByteCount;
      property OutputByteCount: Integer read GetOutpuByteCount;
      property RIHigh: Boolean read GetRIHigh;
      { Other properties }
      property InitString: string read FInitString write FinitString;
      { Events }
      property OnBreak: TOnErrEvent read FOnBreak write FOnBreak;
      property OnCDChange: TOnLineStatusChange read FOnCDChange write FOnCDChange;
      property OnCTSChange: TOnLineStatusChange
        read FOnCTSChange write FOnCTSChange;
      property OnData: TOnDataEvent read FOnData write FOnData;
      property OnDSRChange: TOnLineStatusChange
        read FOnDSRChange write FOnDSRChange;
      property OnOverrunErr: TOnErrEvent read FOnOverrunErr write FOnOverrunErr;
      property OnParityErr: TOnErrEvent read FOnParityErr write FOnParityErr;
      property OnFrameErr: TOnErrEvent read FOnFrameErr write FOnFrameErr;
  end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
Constructor TCommDevice.Create;
begin
  FDeviceId := DeviceNotOpen;
  FReceiveQueueSize := DefaultInBuffer;
  FTransmitQueueSize := DefaultOutBuffer;
  FParity := ByteNotSet;
  FStopBits := ByteNotSet;
  FCtsTimeOut := DefaultXflowTimeout;
  FDsrTimeOut := DefaultXflowTimeout;
  FXonChar := #17;
  FXoffChar := #19;
  FXFlowOnLimit := 32;
  FXFlowOffLimit := 512;
  FParityCheck := True;
  FParityDoReplaceChar := True;
  FParityReplacementChar := '*';
  FInitString := 'ATZ';
  FEvents := EV_RXCHAR or EV_TXEMPTY or EV_ERR or EV_BREAK
    or EV_CTS or EV_DSR or EV_RLSD;
  FReadBuffer := MemAlloc(ReadBufferSize);
  if FReadBuffer = nil then
    Raise Exception.Create('MemAlloc failed creating internal buffer');
  FTempOutBuffer := MemAlloc(OutBufferSize);
  if FTempOutBuffer = nil then
    Raise Exception.Create('MemAlloc failed creating temporary output buffer');
end;
```

Table 3 details the relevant APIs used in serial communications. Of those listed, BuildCommDCB, TransmitCommChar and UngetCommchar are not used in the TCommDevice class.

I've developed the class TCommDevice to show how to access the serial ports on your PC. The class is not 'industrial strength', but demonstrates the principles.

Listing 1 shows the public section of the declarations for TCommDevice. As you can see, most of this consists of property declarations, which I have grouped into three types. Device operating parameters have a direct correlation with values understood by the Windows API and affect the device configuration. Status information properties are read only and all call property access routines to retrieve this information. Of these, all but DeviceOpen read data specific to the device. The DeviceOpen property access routine references a field stored within the object itself. Finally, we have the InitString property for the modem.

TCommDevice is created by using TCommDevice.Create (Listing 2), which initialises most of the property fields to default values and allocates memory for its read and write buffers. You can see that some of the property fields (eg FParity) are set to a constant called ByteNotSet ($FF). This is used within the class to determine if you have set the relevant property. If not, the device default is used.

One important line to note is the assignment of the field FEvents, which will contain the flags of all the events we are interested in. See the later section *Detecting Errors/Events* to see how this is used.

The device is not opened during the creation of the object, so you can change the operating parameters before opening. To allow this capability, the relevant property access routines store the information in fields within the object. These are then referenced when the device needs to be configured. See the later section *Configuring The Device*.

Opening A Device

Listing 3 shows how a communication device is opened by the TCommDevice.Open method. The port number is passed as a parameter: Windows supports values between 1 and 9 although 1 to 4 is more common. This method calls the OpenComm API function, which has

three parameters. The port name (eg 'COM2') and the input and output queue sizes to be allocated by the device driver. It returns a device ID that needs to be passed to the other APIs to identify the device. If the value returned is a positive value (including 0) then the device has been opened successfully and you have exclusive access to the device. This ID is stored in FDeviceId. If the value returned is negative there was an error opening the device. You can see from the listing that Windows provides constants for the common causes of error. If the device could not be opened an exception is raised to indicate the cause of the error. If the device was opened OK the Open method configures and initialises the device.

Configuring The Device

After the Open method has successfully opened the device it calls the private ConfigureDevice method to change the operating parameters for the device. When the device is first opened the driver will assign default values for all the parameters. To change these we call the GetCommState and SetCommState APIs. GetCommState is passed the DeviceId (returned from the OpenComm function in the Open method) along with a reference to a DCB (device control block) structure. This function fills the DCB structure with the current values for the device. We can then amend the

➤ *Listing 3*

```
procedure TCommDevice.Open(Port: Integer);
var CommDeviceName: array[0..4] of char;
    TmpID; Integer;
begin
   TmpID :=
     OpenComm(StrPCopy(CommDeviceName, Format('COM%D',[Port])), 2048, 1024);
   case FDeviceId of
     0..32767:
       begin
         FDeviceID := TmpID;
         ConfigureDevice;
         InitialiseDevice;
       end
     IE_BADID:    Raise Exception.Create(
       'The device identifier is invalid or unsupported.');
     IE_BAUDRATE: Raise Exception.Create(
       'The device''s baud rate is unsupported.');
     IE_BYTESIZE: Raise Exception.Create(
       'The specified byte size is invalid.');
     IE_DEFAULT:  Raise Exception.Create(
       'The default parameters are in error.');
     IE_HARDWARE: Raise Exception.Create(
       'The hardware is not available (is locked by another device).');
     IE_MEMORY:   Raise Exception.Create(
       'The function cannot allocate the queues.');
     IE_NOPEN:    Raise Exception.Create('The device is not open.');
     IE_OPEN:     Raise Exception.Create('The device is already open.');
    else
       Raise Exception.CreateFmt('OpenComm failed with error %d.',[FDeviceId]);
    end;
end;
```

➤ *Listing 4*

```
type
  PDCB = ^TDCB;
  TDCB = record
    Id: Byte; { Internal Device ID as returned by OpenComm  }
    BaudRate: Word;          { Baud rate at which running }
    ByteSize: Byte;          { Number of bits/SDU, 4-8 }
    Parity: Byte;            { 0-4=None,Odd,Even,Mark,Space }
    StopBits: Byte;          {  0,1,2 = 1, 1.5, 2 }
    RlsTimeout: Word;        { Timeout for RLSD to be set }
    CtsTimeout: Word;        { Timeout for CTS to be set  }
    DsrTimeout: Word;        { Timeout for DSR to be set  }
    Flags: Word;             { See below }
    XonChar: char;           { Tx and Rx X-ON character   }
    XoffChar: char;          { Tx and Rx X-OFF character  }
    XonLim: Word;            { Transmit X-ON threshold    }
    XoffLim: Word;           { Transmit X-OFF threshold   }
    PeChar: char;         { Parity error replacement char }
    EofChar: char;           { End of Input character     }
    EvtChar: char;           { Received Event character   }
    TxDelay: Word; { Time between chars,not used in Windows }
  end;
const
  {  Flag definitions for the TDCB.Flags field : }
  dcb_Binary      = $0001; { Binary Mode (skip EOF check    }
  dcb_RtsDisable  = $0002; { Don't assert RTS at init time  }
  dcb_Parity      = $0004; { Enable parity checking         }
  dcb_OutxCtsFlow = $0008; { CTS handshaking on output      }
  dcb_OutxDsrFlow = $0010; { DSR handshaking on output      }
  dcb_DtrDisable  = $0080; { Don't assert DTR at init time  }
  dcb_OutX        = $0100; { Enable output X-ON/X-OFF       }
  dcb_InX         = $0200; { Enable input X-ON/X-OFF        }
  dcb_PeChar      = $0400; { Enable Parity Err Replacement  }
  dcb_Null        = $0800; { Enable Null stripping          }
  dcb_ChEvt       = $1000; { Enable Rx character event.     }
  dcb_Dtrflow     = $2000; { DTR handshake on input         }
  dcb_Rtsflow     = $4000; { RTS handshake on input         }
procedure TCommDevice.ConfigureDevice;
var RC: Integer;
begin
  RC := GetCommState(FDeviceId,FDCB);
  If RC = 0 then begin
    if FBaudRate > 0 then FDCB.BaudRate := FBaudRate;
    if FDataBits > 0 then FDCB.ByteSize := FDataBits;
    if FParity <> ByteNotSet then FDCB.Parity := FParity;
    if FStopBits <> ByteNotSet then
      FDCB.StopBits := FStopBits;
    With FDCB do begin
      { Set flow control }
      Flags := dcb_Binary;
      XOnLim := 0;
      XOffLim := 0;
      CtsTimeOut := 0;
      DsrTimeOut := 0;
      XOnChar := #0;
      XOffChar := #0;
      case FFlowControl of
        fcHardwareDSRDTR:
          begin
            Flags := Flags or dcb_OutxDsrFlow or dcb_Dtrflow;
            XOnLim := XFlowOnLimit;
            XOffLim := XFlowOffLimit;
            DsrTimeOut := FdsrTimeout;
           end;
        fcHardwareCTSRTS:
          begin
            Flags := Flags or dcb_OutxCtsFlow or dcb_Rtsflow;
            XOnLim := XFlowOnLimit;
            XOffLim := XFlowOffLimit;
            CtsTimeOut := FCtsTimeout;
          end;
        fcSoftware:
          begin
            Flags := Flags or dcb_InX or dcb_OutX;
            XOnLim := XFlowOnLimit;
            XOffLim := XFlowOffLimit;
            XOnChar := FXOnChar;
            XOffChar := FXOnChar;
          end;
      end;
      { Set parity checking options }
      if ParityCheck then Flags := flags or dcb_Parity;
      if ParityDoReplaceChar then begin
        Flags := flags or dcb_PeChar;
        FDCB.PeChar := ParityReplacementChar;
      end;
    end; { With FDCB do }
    RC := SetCommstate(FDCB);
    if RC = 0 then begin
      { Set up notification events }
      FNotifyWindow := AllocateHWnd(NotifyProcedure);
      SetCommEventMask(DeviceId,FEvents);
      EnableCommNotification(DeviceId, FNotifyWindow,
        ReceiveTrigger, TransmitTrigger);
    end else
      Raise Exception.CreateFmt('Failed to configure '+
        'Device. SetCommstate ended with error %d.',[RC]);
  end else
    Raise Exception.CreateFmt('GetCommState ended with %d '+
      'when trying to configure Device.',[RC]);
end;
```

Field	Flow control type			
	None	DTR/DSR	CTS/RTS	Software
dcb_DTrFlow	0	1	0	0
dcb_RtsFlow	0	0	1	0
dcb_InX	0	0	0	1
dcb_OutxCtsFlow	0	0	1	0
dcb_OutDsrFlow	0	1	0	0
dcb_OutX	0	0	0	1
XOnLim	0	50	50	50
XOffLim	0	500	500	500
CtsTimeout	0	0	300	0
DsrTimeout	0	300	0	0
XOnChar	0	0	0	17
XOffChar	0	0	0	19

➤ *Table 4: Configuring flow control*

RlsTimeout	We use event handlers when the CD line changes state
dcb_RtsDisable	This would stop the driver using the RTS signal
dcb_DtrDisable	This would stop the driver using the DTR signal
dcb_NULL	This would cause the driver to skip NULL characters
dcb_ChEvt	Generates an EV_RXFLAG event if character EvtChar is read
EofChar	Used to indicate the end of data if dcb_Binary is clear
EvtChar	Character which will cause the EV_RXFLAG event if dcb_ChEvt is set
TxDelay	Not used by Windows

➤ *Table 5: Unused flags and fields*

fields of this structure as required and call the SetCommState function to update the values. Listing 4 shows the ConfigureDevice method along with the declaration for the TDCB structure.

To set the baud rate we change the BaudRate field of the TDCB structure. As the field is only a WORD value it cannot handle baud rates over 65535. To get round this, a convention is used by the driver to interpret this value. If the high byte of this field is $FF the low byte refers to an entry in an internal table which is used to set the actual baud rate. If the high byte is not $FF then the value itself will be used. Windows has pre-defined constants which should be used to set this field (beginning with CBR_). For example, CBR_128000 equates to $FF23.

To set the number of data bits per SDU we change the ByteSize field of the TDCB structure. This should be between 5 and 8.

To set the number of parity bits we change the Parity field in TDCB. Windows has pre-defined constants for this (NoParity, OddParity, EvenParity, MarkParity, SpaceParity). To enable parity checking the dcb_Parity flag of the TDCB.flags field needs to be set (in TCommDevice this is done by default). If you wish to replace the characters which generated parity errors with a different character you will need to set the dcb_PeChar bit of the TDCB.flags field and set the peChar of the TDCB structure to the value of the replacement character.

To set the number of stop bits we change the StopBits field in TDCB. Windows has pre-defined constants (OneStopBit, One5StopBits and TwoStopBits) for these options.

Setting flow control is more complicated as it needs a combination of fields to be set correctly for it to work. Table 4 shows how the TDCB values should be set for the flow control options. The examples provided assume that a flow control will be activated when the buffer is within 50 bytes of being full and deactivated when the buffer is within 500 bytes of becoming empty and the driver should wait a maximum of 300 ms for the flow control protocol to indicate that it is OK to transmit data.

DTR/DSR is implemented in TCommDevice but is not used in the demo application as these lines are used for device control with modems. The case FFlowControl statement in the ConfigureDevice method sets these fields.

The dcb_Binary flag in the TDCB structure, when set, indicates that we are processing binary data. This is always set in TCommDevice.

Table 5 lists the flags/fields in the TDCB structure which are not used by TCommDevice as their function is either incompatible with the function of the class or irrelevant.

So then, how do we configure the device after it is open? The ConfigureDevice method described above applies all the parameters in one go. After it has called SetCommState the FDCB structure contains a copy of all the parameters currently used by the device driver. However, some properties can be changed after the device has been opened (BaudRate, DataBits, Parity and StopBits). Listing 5 contains the property access routine for the DataBits property as an example. If you look at this code you will see that it first updates FDataBits with the new value, which will be used when the device is next opened. It then checks to see if the device is already open. If so, it updates the FDCB structure and calls SetCommState to update the operational parameters. If successful the device driver will start using this updated value. If this call fails the status of the FDCB structure is restored and an exception is raised.

Initialising The Device

In the TCommDevice.Open method, after calling ConfigureDevice the private method InitialiseDevice is called (see Listing 6), which uses the EscapeCommFunction API. This causes the device driver to carry out the extended function (passed as the second parameter) to the device specified in the first parameter. We pass SETDTR to ensure

that the DTR line is asserted. This should have happened anyway when the device was opened but sometimes this does not occur. After ensuring that the DTR line is asserted the method then writes out any initialisation string (it defaults to ATZ) using the WriteLn method: this is covered later.

Table 6 lists the extended functions applicable to serial ports.

From our hardware discussion you will remember that the DTR line will cause the modem to hang up the connection if it goes low. Since the EscapeCommFunction function accepts CLRDTR and SETDTR as parameters we can use this to implement a hardware driven hang-up method. If you look at the code on the disk you will see that EscapeCommFunction is called twice with a 500ms delay to drop then re-assert the DTR line. This is not to be confused with the Hayes modem command to force a hang up (ATH). This is a brutal, hardware driven, method which should not fail. If the modem is not currently in command mode (not accepting AT commands) it will not process an ATH command, but all modems should react when the DTR line drops.

Detecting Errors/Events

Before I describe how to read and write data to the communication device we'll look at how to detect errors. There are two situations where errors may occur which would be of interest to us. The first is during an API call and for these we just check the function's return value. The second is when errors occur independently of our processing. Due to the asynchronous nature of communications, errors can occur at any time, regardless of what our program is doing.

Let's deal first with detecting errors after API calls. If you review the online help for the APIs listed in Table 3 you will see that they all return values to indicate if the function worked successfully. Some functions, like OpenComm, will indicate by the value returned the cause of the error. However, other functions, for example WriteComm, notify that an error was encountered (in this case by returning the

Function	Meaning
CLRDTR	Clears the DTR (data-terminal-ready) signal
CLRRTS	Clears the RTS (request-to-send) signal
GETMAXCOM	Returns the maximum COM port identifier supported by the system
SETDTR	Sends the DTR (data-terminal-ready) signal
SETRTS	Sends the RTS (request-to-send) signal
SETXOFF	Causes transmission to act as if an XOFF character has been received
SETXON	Causes transmission to act as if an XON character has been received

➤ *Table 6*

```
procedure TCommDevice.SetDataBits(Value: Byte);
var OldByteSize: Byte;
    RC: Integer;
begin
  if Value in [5..8] then begin
    FDataBits := Value;
    if DeviceOpen then begin
      OldByteSize := FDCB.ByteSize;
      FDCB.ByteSize := Value;
      RC := SetCommstate(FDCB);
      if RC <> 0 then begin
        FDCB.ByteSize := OldByteSize;
        Raise Exception.CreateFmt('Failed to change Device data size. '+
          'SetCommstate ended with error %d.',[RC]);
      end;
    end
 end;
end;
```

➤ *Listing 5*

```
procedure TCommDevice.InitialiseDevice;
begin
  EscapeCommFunction(FDeviceId,SETDTR); { Assert the DTR line }
  FTempOutputStoredBytes := 0;
  if InitString <> '' then
    WriteLn(InitString);
end;
```

➤ *Listing 6*

```
const
  { GetCommError status flags }
  ce_RXOver   = $0001;   { Receive Queue overflow }
  ce_Overrun  = $0002;   { Receive Overrun Error }
  ce_RXParity = $0004;   { Receive Parity Error }
  ce_Frame    = $0008;   { Receive Framing error }
  ce_Break    = $0010;   { Break Detected }
  ce_CTSTO    = $0020;   { CTS Timeout }
  ce_DSRTO    = $0040;   { DSR Timeout }
  ce_RLSDTO   = $0080;   { RLSD Timeout }
  ce_TXFull   = $0100;   { TX Queue is full }
  ce_Mode     = $8000;   { Requested mode unsupported }
type
{ TCommStat structure }
  PComStat = ^TComStat;
  TComStat = record
    Flags: Byte;
    cbInQue: Word;       { count of characters in Rx Queue}
    cbOutQue: Word;      { count of characters in Tx Queue}
  end;
const
  { Flag definitions for the TComStat.Flags byte }
  com_CtsHold = $0001;   { Transmit is on CTS hold }
  com_DsrHold = $0002;   { Transmit is on DSR hold }
  com_RlsdHold = $0004;  { Transmit is on RLSD hold }
  com_XoffHold = $0008;  { Received handshake }
  com_XoffSent = $0010;  { Issued handshake }
  com_Eof = $0020;       { End of file character found }
  com_Txim = $0040;      { Character being transmitted }
var ComStat: TComStat;
    ErrorFlags: Integer;
BytesRead := ReadComm(DeviceId,@Buffer,BufferLen);
if BytesRead <= 0 then begin
  ErrorFlags := GetCommError(DeviceId,Comstat);
  if ErrorFlags <> 0 then ProcessError(ErrorFlags);
end;
```

➤ *Listing 7*

Flag	Value	Meaning
EV_RXCHAR	$0001	Set when any character is received into the receiving queue
EV_RXFLAG	$0002	Set when the event character (TDCB.EvtChar) is received into the receiving queue (not used as we are using dcb_Binary).
EV_TXEMPTY	$0004	Set when the transmission queue becomes empty
EV_CTS	$0008	Set when the CTS line changes state
EV_DSR	$0010	Set when the DSR line changes state
EV_RLSD	$0020	Set when the RLSD (CD) line changes state
EV_BREAK	$0040	Set when a break is detected in transmission
EV_ERR	$0080	Set when a frame, overrun or parity error has occurred
EV_RING	$0100	Set when the modem detects the ring tone *
EV_CTSS	$0400	Set to indicate the current state of the CTS signal *
EV_DSRS	$0800	Set to indicate the current state of the DSR signal *
EV_RLSDS	$1000	Set to indicate the current state of the RLSD (CD) signal *
EV_RINGTE	$2000	Set to indicate ring trailing edge indicator *
* Not used, due to COMM.DRV not providing the correct status. Instead we determine the state of these lines using a technique provided by Microsoft to circumvent the bug.		

➤ *Table 7*

negative of the number of bytes written) but not the cause. In these situations the `GetCommError` function is used to determine the cause of the error. When an error occurs Windows locks the device until `GetCommError` is called.

If you look at Listing 7 you will see an example of a call which reads some data from the `Comm` device to a buffer, then checks for an error: in which case `BytesRead` will be negative. If 0 is returned we still need to check for an error (as opposed to nothing to be read). It calls `GetCommError` to determine the cause of the error. `GetCommError` has two purposes. Its return value is an integer, which contains flags indicating the cause of the last error detected by the driver (detailed in Listing 7). It also returns the current status of the device into the `TComStat` structure passed as the second parameter. `TCommstat` contains a `Flags` field which indicates what the device driver is currently doing. It also has two fields which show how many bytes are in the driver's input and output queues. This last feature is used in the property access routines for the `InputByteCount` and `OutputByteCount` properties of `TCommDevice`.

As I stated before, the device driver can detect errors at any time. For example, hardware errors detected by the UART chip will be communicated to the device driver. We can configure the device to inform us when these events occur. As well as errors, we can also be informed when the status of the device changes (eg CD line changes state) or an event of interest to us has occurred (eg the output queue has become empty).

During the discussion of the `ConfigurePort` routine I conveniently ignored the final part of that routine which sets up notification events. We'll discuss this now (refer back to Listing 4). To do this we need to do three things.

First, we need to allocate a window handle to which `WM_COMMNOTIFY` messages will be posted by the device driver when events occur. This is done by calling `AllocateHWnd` and passing the `TCommDevice.NotifyProcedure` method to process the messages as they are generated.

Secondly, we indicate which events we are interested in by calling `SetCommEventMask` with the device ID returned when we opened the device, along with an event mask word containing the event flags we are interested in. From the code you can see it passes `FEvents` to the function which was initialised in the `Create` constructor. Table 7 shows the flags which are available. After this call the driver will record the events allowing you to detect if they have occurred.

Finally, we enable notification of events by calling `EnableCommNotification`, which expects four parameters. The first two are the device ID and the window handle to receive the messages. The third is a threshold value to inform you if there are at least this number of characters to be read from the driver buffer. The fourth parameter is a threshold value to indicate if the buffer capacity drops below this value, to prompt you to transmit more data. We pass -1 for these values as we use `EV_RXCHAR` to be informed when data has arrived at the port and `EV_TXEMPTY` to be informed when the output queue becomes empty (see the later sections on reading and writing data). Also, the MSDN describes situations where the system can get flooded with `WM_COMMNOTIFY` messages when using the threshold values.

An Event Handler

When the `Comm` driver has detected an event which we are interested in it posts a `WM_COMMNOTIFY` message to our window procedure. Each message may represent more than one event. The `wParam` parameter for this message is the device ID. The `LoWord(lParam)` parameter can contain a combination of these flags:

- `CN_EVENT`: At least one of the enabled events (as in `FEvents` in our case) has occurred.
- `CN_RECEIVE`: The input buffers threshold is greater than that supplied in the `EnableCommNotification` call.
- `CN_TRANSMIT`: The output buffer contains fewer characters than that supplied to the `EnableCommNotification` call.

Listing 8 shows a skeleton message handler procedure to process the notification message (see the disk for the actual implementation). As the `LoWord(lParam)` can contain a combination of all three `CN_` values we test for each flag in turn. The `CN_RECEIVE` could be used to trigger the reading of data from the serial port. The `CN_TRANSMIT` could be used to indicate that more data should be written to the port. The most interesting part of this routine is the processing when a `CN_EVENT` occurs.

When the CN_EVENT is processed we call GetLastError to find out which error flags have been generated (it may return 0 as events don't always equate to errors). A call is then made to GetCommEventMask. This serves two purposes: it returns the events which have been triggered, which are stored in the local variable EventFlags, and also causes the driver to clear these events, to allow them to be generated again.

Comm Lines Status

There is a bug in the Windows driver software which means that you can get misleading information for the status of the Comm lines if you attempt to use the status flags in Table 7. Microsoft has provided a solution to this problem, which has been used in TCommDevice to get the correct status of the CTS, DSR, RI and CD lines via their property access routines (eg see GetCTSHigh).

The UART chip has a register called the Modem Status Register (Port 6) which contains the status of the control lines. The driver software keeps a copy of this register called the MSR shadow in its internal buffers. You can get at this byte by using the fact that the SetCommEventMask function returns a pointer to the event word for the device. The MSR shadow register is at an offset of 35 bytes from this address. By testing the bits in this byte we determine the actual status of the lines. Do not change its value! The property access routines pass FEvents to SetCommEventMask so no change is made to the events we wish to process.

Writing Data

Writing data to the Comm device is achieved by simply calling the WriteComm function passing the DeviceId, a pointer to the buffer and the number of bytes to be written. The function returns the number of bytes written. If there was an error the result will be negative.

When deciding the functionality of the write function it became apparent that there were numerous options available. One was to not return control until all the bytes have been accepted by the device driver, another was to store all writes in a linked list and write them later. Note: the fact that the device driver has accepted the data does not mean that it has hit the wire. It will (hopefully) be sent in due course...

The method I chose was to write as much as I could to the device driver, storing any remaining bytes in a temporary buffer, to be sent later. As long as this action was performed the method returns True to indicate the data was accepted (I'll describe what happens to this temporary data shortly). If a subsequent call is made to the Write method it will wait for up to 5 seconds for the temporary buffer to be written before processing the buffer. If the temporary buffer is not written after 5 seconds the method returns False to indicate that no data was written. That is, the function will either accept the whole buffer or nothing.

Now, what happened to the temporary buffer stored away? As you can see from Listing 9, if the routine could not pass the whole buffer on to the device driver it calls StoreRemainderInTempBuffer which will copy the bytes left into a holding buffer. Now, in the previous section we discussed notification of events. One of the events we are interested in is the EV_TXEMPTY. This is sent when the driver's output queue becomes empty. Although not shown in this article, you can see from the code supplied on the disk that the NotifyProcedure event handler writes data from this temporary buffer to the device driver. This clears up any backlog to allow the next write to occur.

A second write method provided is WriteLn, for use when issuing

```
procedure TCommDevice.NotifyProcedure(var Message: TMessage);
var LastError: Word;
    EventFlags: Word;
begin
  With Message do
    if Msg = WM_COMMNOTIFY then begin
      if LoWord(LParam) and CN_EVENT = CN_RECEIVE then begin
        { ...Read data here and process }
      end;
      if LoWord(LParam) and CN_EVENT = CN_TRANSMIT then begin
        { ...Write any data which needs to be sent here }
      end;
      if LoWord(LParam) and CN_EVENT = CN_EVENT then begin
        LastError := GetCommError(DeviceId,nil);
        EventFlags := GetCommEventMask(DeviceId,FEvents);
        { We now have the events which occurred in EventFlags
          and the errors in LastError }
      end;
    end;
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
function TCommDevice.Write(Buff: PChar; BuffLen: Integer): Boolean;
var
  BytesWritten: Integer;
  Errors: Integer;
  Comstat: TComStat;
  StartTicks: LongInt;
begin
  if DeviceOpen then begin
    { If we are already waiting to send the last block,
      caller will have to call us again after receiving false }
    Result := False;
    StartTicks := GetTickCount;
    While FTempOutputStoredBytes <> 0 do begin
      Application.ProcessMessages;
      if GetTickCount - StartTicks > 5000 then
        Break;
    end;
    if FTempOutputStoredBytes = 0 then begin
      BytesWritten := WriteComm(DeviceId,Buff,BuffLen);
      if BytesWritten < 0 then begin
        Errors := GetCommError(DeviceId,ComStat);
        ProcessComError(Errors);
        BytesWritten := -BytesWritten;
        StoreRemainderInTempBuffer(Buff,BuffLen,BuffLen - BytesWritten);
      end else if BytesWritten = 0 then begin
        StoreRemainderInTempBuffer(Buff,BuffLen,BuffLen);
      end else if BytesWritten < BuffLen then begin
        StoreRemainderInTempBuffer(Buff,BuffLen,BuffLen - BytesWritten);
      end;
      Result := True;
    end;
  end else
    Raise Exception.Create('Comm Device is not open.');
end;
```

commands to a modem. It accepts a string, adds a carriage return (#13) then calls the Write method to pass the data to the device driver.

Receiving Data

If you refer back to Listing 1 you will see that there is no read method! Also, this section is called receiving data rather than reading data. This is intentional. I decided that rather than have a read method to allow you to attempt to read data which may, or may not, be in the input queue, all notification of data received would be event driven. So, all processing for the reading of data is driven by the event notification routine. We make use of the EV_RXCHAR event flag to ensure we get a message when any characters arrive in the input queue.

If you refer to the NotifyProcedure code on the disk, you will see that, when it receives this event, it repeatedly calls the ReadComm routine to read the current input buffer then calls the OnData event procedure (if it is assigned) to pass this data on to the user application. A disadvantage of this approach is that you tend to get these events quickly (ie every few characters).

Closing And Destroying

The device is closed by calling the TCommDevice.Close method. Listing 10 shows the Close method in detail. After ensuring that the device is really open it cleans up then closes the device. It initially calls the EnableCommNotification function with a windows handle value of 0 to stop the device driver from sending any more notification messages. It then frees the window handle. Once this initial cleanup has been performed it then calls the CloseComm API which will cause the device driver to close the device and make it available to other processes. At this point we can no longer access the device so the FDeviceId field is set to the constant DeviceNotOpen. If we attempt to call any of the methods an exception will be raised.

When the object is destroyed it calls Close to close the device (if open), then frees the memory allocated for its read and write buffers.

Demo Application

A simple terminal type application is included on the disk, SERDEMO, which makes use of TCommDevice to allow you to explore the details described in this article. When you first execute the program you are presented with a window similar to Figure 4 (without the Event descriptions window). The first area to explore is the Options menu. This allows you to configure all the operating parameters for the device (eg Baud rate, parity, flow control etc). Once you have configured these options to your liking you can use the File menu option to open the device. When you do this you should see that the pretty lights in the Status of communication lines area change from grey to a combination of green (on) and red (off). While the device is open these lights will show the status of these lines. The Terminal commands edit box allows you to type commands which, when you press Enter, will be sent to the modem. Below this edit box is the most recent commands issued, to allow you to retrieve them to re-execute them (the cursor keys, intercepted when the edit box has focus, will scroll through these commands).

If you look at the Actions menu you will see there is an option to Show events. Select this and a textual description of any events (excluding the data transmit/receive events) is given to show that they are being detected. The events shown in the screen shot were generated by opening the device, powering off the modem then powering it back on again. The final CD is now online was generated when I logged onto CompuServe. In fact, CompuServe is strange in that, officially, you need to log onto it with 7 data bits, even parity and 1 stop bit (7E1). After you have got past the initial logon script you need to switch over to 8 data bits, no parity and 1 stop bit (8N1). However, it just happens that 7E1 and 8N1 are the same length. So, if you log onto CompuServe with 8N1 you will be

➤ *Listing 10*

```
procedure TCommDevice.Close;
begin
  if DeviceOpen then begin
    if FNotifyWindow <> 0 then begin
      EnableCommNotification(DeviceId,0,-1,-1);
      DeallocateHWnd(FNotifyWindow);
      FNotifyWindow := 0;
    end;
    if DeviceOpen then
      CloseComm(DeviceId);
    FDeviceId := DeviceNotOpen;
    FillChar(FDCB,sizeof(TDCB),0);
  end;
end;
```

prompted to enter CIS, your User ID and Password (although the screen will look garbaged). Then when you get to the initial menu all will look fine. However, if you log on with 7E1, the initial logon script will look fine, then once you have logged on, the event window will list numerous parity error events. This is because CompuServe has switched to 8N1 but the Comm device is configured as 7E1. Therefore, for 50% of characters a parity error will be generated. If you configure the device to 8N1 you will see garbage during the logon script phase, but you will not see parity errors generated. This is because, from a hardware viewpoint, 7E1 is acceptable when the device is configured to 8N1 as the bit streams are the same length.

Conclusion

I hope this article has stirred your enthusiasm to begin delving into the world of serial communications and that the TCommDevice class proves a useful starting point. There are, of course, good commercial comms libraries about, but there are also occasions when something simple and home-grown will fit the bill. Should you then upgrade to something more sophisticated, you will also have the benefit of a good understanding of what's actually happening!

John Chaytor is a freelance programmer who lives and works in Brighton, UK, and can be contacted via CompuServe as 100265,3642

➤ *Figure 4*