Accessing the Serial Port

Rod Bird

Table of Contents

PC's use a serial communications standard usually referred to as RS232. This standard defines the signals and lines of the port and was first established in the early 1960s. Once regarded as complex, its present format is simplified and easy to implement in Liberty BASIC.

An RS232 serial port is always hardware based. It uses voltages higher than normally used in a PC, typically -15 to +15 volts and so needs dedicated hardware. The port will be implemented on the motherboard of older PC's, as a separate PCI card in later PC's and in more modern machines it will take the guise of a dongle on the end of a USB lead.

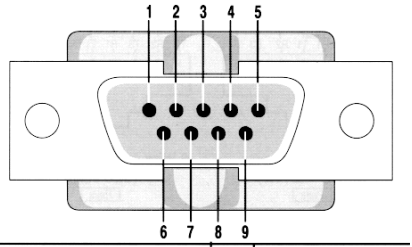
Let me differentiate the Universal Serial Bus (USB) and the serial port. USB is the new wired way to connect PC peripherals. Almost any kind of peripheral comes in a USB flavor. Joysticks, keyboards, printers etc.including serial port and parallel port interfaces. Now it matters not a jot that the serial port we are going to discuss is wired in via USB. The device itself will behave exactly like a serial port soldered on the motherboard.

When I say exactly, I should differentiate between the very original serial port and all other types, defined as "virtual" serial ports. The difference is that the originals had fixed memory addresses and could be programmed by "bit banging". That is directly accessing the memory addresses of the controlling hardware. Thus pins could be set and reset simply by writing to a single, fixed memory address.

Liberty BASIC allows this with INP and OUT but as so few of us will possess such a serial port I'm not going to cover it. There are examples of the technique on the Forum. Virtual ports don't allow "bit banging" but we can access the pins in other ways.

Pin out

Check out the diagram below. I'm showing the DB9 format, this is the simplest format, as we might have been discussing DB25 and its 25 pin names. Most new devices will use the DB9 format. If yours is DB25 it is pretty easy to research the pin outs. We will actually be using very few of the pins even in DB9 format.



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

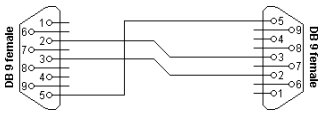
RTS & DTR are outgoing handshaking lines that we can set and hold. DCD, DSR, CTS, and RI are incoming handshaking lines that we can read. RX & TX are controlled by the serial port hardware and pass

the serial data.

A positive voltage at the TX pin indicates ON or SPACE, a negative voltage indicates OFF or MARK. The port itself is pretty robust and will withstand the odd short or grounding but do take care if connecting your own kit, make sure it will withstand the voltages. Some modern implementations sometimes work with 0 volts as OFF and +5 volts as ON but will only operate over a limited cable length.

Making the connection

Now how do we connect all of these pins between two devices to create a serial link? Well we only need three of the pins, TX, RX, and GND. Connect the TX of one to the RX of the other and vice versa, connect the GNDs and you have a working serial communications link with just three wires!



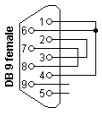
Because we cross connect the TX and RX pins, independent channels are established for two-way (full-duplex) communications. Each device has an input buffer and an output buffer. Messages, in the form of ASCII characters are accepted from the PC and queued in the output buffer to be passed over the communications link as time permits. At the other end the received characters are stored in an input buffer awaiting the receiving PC reading and emptying that buffer. Thus there are four buffers involved in any serial link.

Data is passed across the wire as a sequence of ON OFF bits, the rate bits are passed at is determined by the Baud rate, 9600 bits per second is common. Typically 8 data bits define one character and to that parity checking and stop bits are added. This additional bit information helps the serial hardware pass the messages efficiently.

With 8 data bits ASCII characters between 0 and 255 can be sent. If we drop down to seven bits we are limited to sending ASC values between 0 and 127. Always, the smallest component that can be sent is one character, a byte, a number between 0 and 127 or 255.

Loop back

If you want to have some fun you can link the TX and RX pins of your device. This is called "Loop back" and lets you send and receive messages from yourself. This is very useful for experimenting. You may also link pins 1,6 and 4 and pins 7 and 8 to mimic handshaking but all you really need are pins 2 and 3 linked.



Opening the port

This line of code might start to make a little more sense.

```
open "Com2:9600,n,8,1" for random as #commHandle
```

1. Com2: names the port to be used
2. 9600 defines the baud rate
3. n states no parity bit is added
4. 8 states that 8 bits are used to define the character
5. 1 states that 1 stop bit has been appended
6. random allows us to read and write to the port
7. #commhandle is the handle or Windows identifier of the port (actually a number)

Several serial ports may exist on your PC; by convention they are named Com1, Com2 and so on. Typically Com1 and Com2 are used to name the hardware on the motherboard, virtual ports can have wider ranging names; Com5,6,7,8 etc.

You will choose the rest of the settings to match the device you are attaching. Most devices define exactly how they should be set up on the serial link. If you don't know, experiment, you can't break anything.

Handshaking

Most often, simple devices will dispense with handshaking. Handshaking is all about signaling from one device to the other to start or stop sending data or to flag that the device is plugged in and ready to go. Carl recommends these additional settings in the open com statement.

```
open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle
```

1. ds0 - Set DSR timeout in milliseconds to 0, effectively off
2. cs0 - Set CTS timeout in milliseconds to 0, effectively off
3. rs - Disable detection of RTS (request to send)

Using this recommended style ensures that the simplest serial link is established, all that is required is for the message to be passed from TX to RX pin. All other pins are ignored.

That said the two pins that give Liberty BASIC coders most trouble are RTS and DTR, Liberty can set and reset these lines. RTS is on by default, DTR is not. If you have a device that refuses to work with Liberty BASIC, DTR may need set on.

Why do we need to set DTR on?

1. Handshaking, the purpose for which the RTS and DTR lines were designed, though the need for handshaking has diminished and few new devices need it.
2. Power, the port never did cater for powering external devices, it was primarily a communications port, but lots of folks wanted their devices to be lean and mean and not burdened with power supplies or batteries. So they stole power from the DTR pin, just a few milliamps, enough to power new cmos low power electronics.
3. Opto Isolation, this is power again but for a different reason. To be completely electrically isolated some hardware uses opto electronic devices that pass the signal by infrared light. To work, the PC side of these devices needs power for the infrared diodes.

Sending and receiving our first message

Let's assume you have linked together TX and RX, pins 2 and 3. This loop back test will allow us to experiment. Here is the code we will use, when you run it use the debugger, the ladybug icon, and single step through the program. You will of course have to know the Com name, use Windows Device Manager to check if you are unsure. Remember you can't break anything.

```
open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle
print #commHandle, "Hello"
dataRead$ = input$(#commHandle, 7)
print dataRead$
```

Now I asked you to step through the program for two reasons, firstly to let you see how the code and variables change but more importantly to give things time to actually happen. It takes time for Windows to open and establish the port. It takes time for the message to be sent bit by bit. In all of the code you write for serial ports you must think through this timing issue. Liberty is extremely fast, the serial port is very slow in comparison.

Buffers

The serial buffers allow us to manage this difference in speed. Messages will be queued for transmission and queued on receipt. It is up to us as programmers to time the writing, reading and clearing of the

buffers to keep loosely in pace with the serial hardware. The default size of the buffer is 8kb. This can be changed with the Com statement but you should rarely need to alter this. You would perhaps make it larger if you were dealing with high speed serial transfer but equally if your code keeps pace the buffer increase is not needed. For example;

```
Com = 16384
```

Message content

Let's talk through what the previous messaging code actually did. The Open Com statement opened the port and established the input and output buffers. Moments later the "Hello" text was sent to the output buffer. Immediately the serial hardware would start sending the info bit by bit from the TX pin to the RX pin and over time the "Hello" message would build up in the input buffer. We then sucked the message out of the input buffer with the input$ command.

Notice that we read in 7 bytes not 5. Why? Well you need to be aware that Liberty appends characters to print statements, CR, a mnemonic for chr$(13) and LF, a mnemonic for chr$(10). On screen this would cause the cursor to return to the left and move down one line, CR stands for Carriage Return, LF for Line Feed. This is standard windows protocol and these additional characters are called control characters.

These control characters are not visible; you only ever get to see displayable, ASCII characters. So keep in mind that serial messages can contain hidden characters and you may need to allow for them.

In the above coding example we could have suppressed the, CR LF pair by ending the print statement with a ";"

```
print #commHandle, "Hello";
```

The above code would only send 5 characters, no control characters would be added. You will of course tailor the message to suit the device you are messaging. Some will expect just the message. If it is a fixed length then fine but if it is not how will the device know it has all been received?

Adding a CR LF pair is how Windows does it, some microcomputers will expect a CR, chr$(13). Some will use a delimiter, perhaps "*" or "|" to signal the message is complete.

Don't be confused by how we name characters. There are only 256 characters that can be sent (0-255). I might display them to you (write them down) as "A" or chr$(65) both mean the same thing. I might use mnemonic CR or chr$(13) or even &H0D, Hexadecimal 0D = 13, all three mean the same thing.

Get yourself a good ASCII table, one that shows the ASCII value, the symbol and the hex value of each ASCII character. You will need to be aware of how ASC(), CHR$() and HEXDEC(), DECHEX$() work as functions when formatting or interpreting messages.

Message timing

This is often the first real problem you will encounter, how to keep pace with messages. There is one command that can help and that is the command to check the length of the text currently in the input buffer. But it is easily misused.

```
numBytes = lof(#commHandle)
```

The numBytes variable will hold the number of characters "currently" in the buffer. Now characters may be coming in continuously, in short random bursts or in evenly timed packets. Therefore we may have measured the buffer in the middle of a transmission and it may already be larger than we measured! What to do? Well there is no one answer, you must choose a strategy that suits the transmission timing.

Picture in your mind a buzzing bee, its buzzing away doing stuff, occasionally it stops and sucks nectar from a stamen. It takes an instant, the bee is happily buzzing and flying and it got what it wanted. Now imagine the bee sitting by a stamen waiting for the little drop of nectar to ooze out. It's not flying, it’s grounded and it isn't busy. It could have collected a thousand drops of nectar elsewhere while it waits for the nectar to ooze from the stamen.

Liberty buzzes and the buffer oozes, the important lesson is not to slow Liberty to the speed of the buffer. The fastest and most effective way to slow Liberty is to code something like this.

```
open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle

while numBytes=0
numBytes = lof(#commHandle)
wend
dataRead$ = input$(#commHandle, lof(#commHandle))
print dataRead$
```

Or this second coding trap, the character by character drip.

```
open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle

[read]
while lof(#commHandle)=0
wend
```

```
dataRead$ = input$(#commHandle,1)
print dataRead$
goto [read]
```

I see this coding often, can you see what is happening? Liberty is buzzing away at a million cycles per second looping round and round awaiting the slow serial transfer to complete.

Now you might say "so what", but it does matter. Windows is multi threaded but still serial. That means that if Liberty hogs processor cycles things slow down. Liberty won't be able to do anything else itself, it won't react to keyboard or mouse input, your computer and all other applications will become less responsive as Windows shares fewer cycles with other applications. Not good and completely avoidable.

Only use the above coding styles if you are dealing with very fast paced messages, hundreds if not thousands of messages per second. In most BASIC and microprocessor situations things will be happening very much slower than this.

Keeping pace

So, the first thing to understand is the pace of transmission. A serial multimeter reading voltages might send four bytes, four times per second. A serial GPS device may send more complex messages, four positional thirty byte messages and one twenty byte update message every second. The multimeter may use a "*" delimiter, the GPS may name messages with a four letter header but also delimit each with "GPS*". You need to be aware of the format and frequency of the messages you will be receiving.

You need to consider how often the message "suite" is transmitted. In the case of the multimeter four times per second and the GPS device, despite its complexity, once per second. No need to query the buffer a million times between messages, you just need to query and read it loosely in pace with the message suite.

Consider the multimeter and the two erroneous coding styles listed above. Can you see that four times per second the message will be received and the code will print it? Each transmission will take about 4 milliseconds. Can you also see that in between times the code will be locked in the while wend loops? So for 984 milliseconds every second the code will be stuck in a loop, for the other 16 milliseconds it will be processing the message, what else gets done?

Breathing

To let the processor breathe and manage other tasks you should time the reading of the buffer. If you are receiving four messages per second then you might plan to check the buffer and read it four times per second. However, it is generally better to check twice as fast as you expect the messages to be received.

That way you never get behind and un-read messages never build up in the buffer. You might imagine just

missing a message, you therefore did not read it in and you wait another quarter of a second before trying again. Another message arrives, you read the first but a second remains sitting in the buffer and you will forever be one message behind. If you check at twice the transmission rate you never fall in arrears.

Timer

The timer is our friend when planning to keep pace with messages. There may be occasions when you have a lot of computation to do and you may be able to balance the time that takes with the timing of the buffer read but generally speaking you are far better off setting up a repeating timer event to read the buffer.

```
open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle
timer 125, [bufferread]
wait
[bufferread]
If lof(#commHandle) >= 4 then
dataRead$ = input$(#commHandle, 4)
print dataRead$
end if
wait
```

That will happily read the multimeter, catch up if it misses a message and allow plenty breathing space for the PC to manage other tasks. Remember that the timer is a repeating event. Turn it off if you pause or branch off to do other things.

Double buffer

You can by now probably envision how you would grab fixed length messages, but what about variable length messages? How do we keep pace? Well keeping pace is easy; it’s just what we have described above. The complication is how much to pull out the buffer. You cannot see the delimiters in the buffer and you don't know how long the message actually is. The solution is to pull whatever is in the buffer into a secondary buffer and parse it.

Here I have commented out the comm statements, copy and paste the code and it will run without a serial connection. Use the debugger and step through the code.

```
'create some dummy buffer info
delimiter$=chr$(10)+chr$(13)
buffer$= "This is the first message"+delimiter$
```

```
buffer$=buffer$+"This is the second message"+delimiter$
buffer$=buffer$+"This is the third message"+delimiter$
buffer$=buffer$+"This is the fourth"+delimiter$

'open "com2:9600,n,8,1,ds0,cs0,rs" for random as #commHandle

'timer 100, [bufferread]
'wait

'[bufferread]
'if lof(#commHandle)>0 then
'dataRead$=input$(#commHandle, lof(#commHandle))

[doublebuffer]
buffer$=buffer$+dataRead$
delimit=instr(buffer$,delimiter$,1)

[loop]
if delimit>0 then
dataRead$=left$(buffer$,delimit)
buffer$=right$(buffer$,len(buffer$) - delimit -1)
print dataRead$
end if
delimit=instr(buffer$,delimiter$,1)
if delimit>0 then [loop]
'end if
'wait for next [bufferread]
wait
```

The code reads the serial buffer at the usual pace, it does not care how many characters are in the serial buffer it simply pulls them all out and appends them to the remnants of the previous read. Then we parse out the message by seeking the delimiter. If we find one we print the message, delete it and take another look at the double buffer. We [loop] If there are any more complete segments and print that out to, until all complete messages are parsed out.

Now some of you might be thinking, what if we missed the very start of the messages, how do we know that the first character in the buffer is the start of the message? Well nothing is sent until the port is ready. Once it is ready messages are queued in the input buffer, nothing is lost and it will sit patiently till you suck it out. Seeking the end of the message is all that is required.

Reading and writing the handshaking lines

You can do a limited amount of input and output with the handshaking pins. Remember RTS and DTR are outputs that we can turn on and off. CTS DSR RI and DCD are inputs that we can read. If you are

interested in controlling external electronics I would advise you to purchase an add on card, something like the BitWhacker boards. These can be sent messages via the serial port and are hugely more flexible. If your needs are simple then you might make use of the handshaking pins as follows.

Obtaining the serial port #handle

The additional functionality we need to manage the hardware requires API calls. Not too complex but you might want to read up on some of the excellent tutorials that exist. We first of all need to know the #handle Windows assigns to the comm port. Stefan shows us how to do this. We can then use the EscapeCommFunction to manage the handshaking pins.

```
'Open the port briefly using an API call to determine the handle given
 by Windows
'We will use this handle later in Liberty, its the only way to get it.
'substitute your own port number
lpFileName$ = "Com2"
dwCreationDistribution = _OPEN_EXISTING
hTemplateFile = _NULL
calldll #kernel32, "CreateFileA", _
lpFileName$ as ptr, _
dwDesiredAccess as ulong, _
dwShareMode as ulong, _
lpSecurityAttributes as ulong, _
dwCreationDistribution as ulong, _
dwFlagsAndAttributes as ulong, _
hTemplateFile as ulong, _
hFileHandle as ulong

calldll #kernel32, "CloseHandle", _
hFileHandle as ulong, _
result as long

'hFileHandle now contains
'the #handle, a number, that identifies the port

'Create a struct to recieve the incoming handshaking data
'this data contains CTS DSR RI and RLSD info.
'For more detail go to
'http://msdn.microsoft.com/en-us/library/aa363258(VS.85).aspx
struct modem,DSRCTS as long

'Now open the com port in Liberty and use the hFileHandle value in API
```

```
 calls
open lpFileName$;":9600,n,8,1,ds0,cs0,rs" for random as #com

'now set reset and read the handshake lines like this
print "setting DTR"
CALLDLL #kernel32, "EscapeCommFunction", hFileHandle as ulong,
 _SETDTR as long,_
result as long


print "re-setting DTR"
CALLDLL #kernel32, "EscapeCommFunction", hFileHandle as ulong,
 _CLRDTR as long,_
result as long

print "re-setting RTS"
CALLDLL #kernel32, "EscapeCommFunction", hFileHandle as ulong,
 _CLRRTS as long,_
result as long

print "setting RTS"
CALLDLL #kernel32, "EscapeCommFunction", hFileHandle as ulong,
 _SETRTS as long,_
result as long

print "reading handshake lines"
CALLDLL #kernel32, "GetCommModemStatus", hFileHandle as ulong,
modem as struct,_
result as void
print "DSR/CTS Byte = ";modem.DSRCTS.struct
'Use AND to determine which signal lines are on eg:
if modem.DSRCTS.struct and _MS_CTS_ON then print "CTS pin is on"
if modem.DSRCTS.struct and _MS_DSR_ON then print "DSR pin is on"
if modem.DSRCTS.struct and _MS_RING_ON then print "RI pin is on"
if modem.DSRCTS.struct and _MS_RLSD_ON then print "DCD pin is on"

Print "Closing com port"
close #com
end
```

Pauses

Notice that if you simply ran this code it would all happen in an instant. Use the debugger and step through to see it all take place. In your own code you may need to insert short pauses or delays to allow things to happen. Opening the port for example, if you immediately try and read it you may get an error. Give it a little time to establish. I code delays like this.

```
code
gosub [delay]
code

[delay]
timer 500,[enddelay]
wait

[enddelay]
timer 0
return
```

Closing the port

It is best to open the port and leave it open till you are completely finished.

```
Print "Closing com port"
close #com
end
```

Now you have all of the code you need to manage the serial port. I hope you have lots of fun doing so.