README for VBTERM

The enclosed VB project comprises a simple VT100-like terminal emulator. I should warn you that it is neither complete (in that it doesn't support all VT100 escape sequences, handle function or cursor keys, etc.), well designed, or fully debugged.

*** *see note at end of file* ***

The VT100 emulation seems sufficient to use with CompuServe, which was a prime requirement of mine. On the other hand, more demanding systems (such as UNIX's VI editor) will not work.

The design of the program reflects its random creation. I tried a lot of different things (some of which I'll discuss below) to make this work, and if I was to do this all over again it would be better structured.

The program seems to be reliable at this point, but there are undoubtedly more bugs to be found. If you find any and fix them, let me know and I'll incorporate them in a future upload. Send those fixes to [76701,11].

DESIGN OVERVIEW

As I mentioned, the design of the program isn't elegant, and so it's hard to tell what's just randomly present and what was a painfully generated bit of twisted code. So let me explain ...

There's two non-obvious aspects of this program. The first is how to interact with the serial port, while the second is how to draw, scroll, and maintain a "terminal" screen using VB and windows functions.

The serial stuff is quite easy, and is all held in the file **SERIAL.BAS**. There are 6 API calls I make to windows to handle the the serial port. The first is the **OpenComm()** routine, which returns a handle to the open port. The next two routines allow me to configure the port as I need. First, I call **BuildCommDCB()**, which takes an MS/DOS style MODE command (e.g., COM1:9600,n,8,1) and builds a data structure known as a **DCB**. Then, I pass the **DCB** (after adjusting a few parameters to my liking) to windows using the **SetCommState()** routine. At this point, the serial port is opened and configured for use. To write to the serial port, I use the **WriteComm()** routine, and to read from it I use **ReadComm()**. The only difficulty I encountered is that whenever windows detects an error on the serial port, be it buffer overflow, line noise, etc., it will stop retuning data with **ReadComm()**. To clear the error, you have to call the routine **GetCommError()**, which will clear the error condition and allow you to resume reading data.

Driving the screen was a bit more difficult. Initially, I just used

the **Print** command in VB to output data, but it was obvious that it alone was insufficient. First of all, it doesn't handle any VT100 style escape sequences, and second it doesn't scroll the screen. (It also turns out that it doesn't handle CR LF sequences very well, giving the effect of double spaced type for everything you receive).

Clearly, I was going to have to handle each character received individually. If you look into the file **vt100.bas**, you'll see there's a routine called **Term_Put** that takes a string and displays it in our terminal window. You'll see that (along with a lot of other stuff) it loops through each character, deciding if it is a character that is "magic" (such as an escape, return, line feed, etc.), or just a printing character. When it encounters a magic character, it performs whatever action is necessary (e.g., for the return character, it sets the current x position to 0). When it encounters a displayable character, it just buffers the character up for later display.

[A brief digression: The overhead for making calls to the windows routines that display text is very high. In the first iteration of the program, I would output each character as I received it. This worked, but the program was unable to keep up with even 2400 baud. For reasons of speed alone, I decided to save up as much text as I could before passing it along to windows; this has resulted in a tremendous performance gain. Hence, you see all of the code involved with checking to see if there's any buffered text to decide if something needs to be written.]

The logic for handling escape sequences is crude. When an ESC character is seen, a flag is set, and all incoming data is diverted to the **AddEscape()** subroutine. The end of the sequence is detected when a letter is received (this works on almost all ANSI style escape sequences with only a few esoteric exceptions). I then do a couple of SELECT CASE statements to get to some code that knows how to handle the escape sequence properly. (See below for comments on attributes).

The actual output is done with the windows **TextOut()** API, rather than using the VB **Print** command. I thought the performance was marginally better, although I didn't test this enough to make a solid comparison. The **TextOut()** routine takes, as its argument, the **hDC** of the window I'm updating (I called the form TTY), the x and y coordinates of where I want to write, the string, and its length. All very straightforward.

Scrolling presents an interesting problem for the terminal emulator. When the emulator receives a line feed and the cursor is on the last line of the window, it needs to move the text up rather than moving the "cursor" down. To do this I could just repaint all the text, but that's too slow. Instead, I used the API routine **BitBlt()**, which will copy one portion of the display (in this case lines 2 through 24) to another portion of the display (lines 1 through 23). **BitBlt()** also serves to clear the

last line out after the scroll (in fact, you see I use it elsewhere to clear to end of line, etc.).

The "cursor" also uses **BitBlt()**; to draw the cursor, I just copy a region onto itself inverting the data in the process.

Attributes (underline, etc.) are handled by just setting the various VB "font" properties. Since there's no "fontreverse" property, I just simulate it by setting reversed text to grey instead of black. It's a cop-out, and I may fix it later (or you can fix it and send me the update! ;-)).

The last thing about the display that needs to be mentioned is handling "paint" messages. If I want to be able to pop windows above the terminal screen, or minimize it, etc., the code has to be able to recreate the screen when a "paint" message is received. To do this, I have to remember all the text that is on the display. This is done using two buffers, **ScrImage** and **ScrAttr**. The first buffer holds the actual text that's on the dislay, while the second hold the attributes of each character (so I know to redraw it as underline, etc.). A fair amount of work is done making sure that these buffers match what is on the screen.

Last, it's worth mentioning how these things get tied together. Unlike the keyboard, the com ports don't generate any "messages", so any com program has to constantly check for new data rather than being told about it. My first go at the program had a timer that constantly checked for new data. This worked OK, but it really didn't have the crispness I wanted. So, I settled for an endless polling loop in the program. All the routine does is constantly read from the serial port and send any received text off to the display. In order to prevent the program from bringing windows to a halt, it calls the VB routine **DoEvents()** on every loop, which lets other applications on the system run.

Other than these features, there's a **capture-to-file** feature which I've thrown in; all that does is open a file and print to it with the received text. It should be simple to add XMODEM to the package, but that is left to the reader...

I hope you find this useful in your use of VB. As I said, the code is far from perfect, but it does work and illustrates some non-obvious points. Best of Luck!

Charles McGuinness
[76701,11]

****** MODIFICATIONS ******

Several corrections have been made to the source

code originally distributed as VBTERM.ZIP. From a functional standpoint, correct updating of the screen has been added, as well as calculating the cursor position. Also, full screen movement commands have been mostly added as well as some color support. The result of this is that you can probably call just about any interactive service which supports ANSI or VT100 full screen text interfaces and have the program work correctly. The main problem the previous version experienced was an incorrect declaration for a function in the Windows GDI DLL, where a ByVal keyword was missing.

I began to add xmodem and ymodem transfer to this program, but it will be simpler and more appropriate to add this as callable DLL function. Please look for this version in the next month or so.

If anyone else has plans on working on this, please see if you can organize the menu to support a dialing directory, etc. so that modem commands are not required. This will make the program much more useable for most persons.

I would like to cite the creator of the original version of this software, Charles McGuinness for his efforts and his support of the public domain concept, which I am hopefully also endorsing here.

Robert C. Evans, Jr.
6121,17 [also Jerry Gentry]