README for VBTerm (Second Version)

The enclosed Visual Basic project comprises a simple VT100-like terminal emulator. It is meant **only** as an example of what is possible using Visual Basic. I should warn you that it is neither complete, well designed, fast, or fully debugged.

The program also supports a simple scripting language as well as XMODEM upload and download.

It is entirely written in VB, and uses only standard windows DLLs. It does use Windows APIs, so some parts are not easily understood without the windows API docs. But those portions are reasonably isolated, and you should be able to use the modules without worrying about the APIs.

A few notes:

This program is Copyright 1991 by Charles McGuinness.

Portions are derrived from Copyrighted works by Microsoft (specifically, the CONSTANT.TXT and WINAPI.TXT header files have been used in this program, and the forms FILECREATE and FILEINFO are based on examples given in the Visual Basic manuals.).

You are granted a royalty free license to use my sources in any way you see fit, commercial or otherwise. I assume that Microsoft feels similarly about their sources, but I cannot speak for them...

Any bug fixes, corrections, suggestions, etc., may be passed along to Charles McGuinness, CompuServe ID [76701,11].

Using VBTerm

When you first start VBTerm, a window will open and allow you to set your communications parameters. In the code, only COM1 and 2 are supported (other have added COM3 and COM4 support easily). After the window closes, you are in "terminal" mode.

The **File** menu contains two options, **Script ...** and **Exit**. The script choice allows you to start a script (more on that later).

The **Settings** menu contains three choices. The first, **Com Parameters** allows you to change the baud rate, com port, etc., that you are using. The second, **Monitor Mode**, allows you to see the Escape sequences being sent to the program rather than having the program execute them. I use this mostly for debugging the vt100 emulation. The third option, **Break** will send a short (~200ms) break.

The **Transfers** menu allows you to up and down load files. **Capture to File ...** allows you to write all received text to a file for later use. The **Upload Text...** option allows you to send a text file without the benefit of a protocol -- it's just a straight dump of the contents.

The **Rec Xmodem Binary ...** choice allows you to receive a binary file via XMODEM, while **Rec Xmodem Ascii...** allows you to receive a text file via XMODEM. The difference is that the ascii version will strip off ^Zs that appear in the file (which xmodem tends to add at the end of the file). **Tx Xmodem...** allows you to upload a file via XMODEM (there is no difference between binary and text uploads).

<u>Writing VB Scripts</u>

As mentioned above, VBTerm supports a very simple scripting language that allows you to automate some tasks (I use it to log into CompuServe, for example). When you select the **Scripts...** menu choice, it will ask you for the name of a script (*.scr), and then will execute the script.

The script files are just text files (use Notepad to create them, if you like), and have the following syntax: The first character on the line is the command, and the rest of the line are the arguments. The commands are:

Stext	The text is sent to the host (with a CR)
Dfile	VBTerm starts capturing to the file
Ufile	VBTerm uploads the file to the host
Wlen,pat1,pat2,...	VBTerm will wait to receive text that matches pat1, pat2, etc., for up to len seconds. The number of patterns can be from 0 to 10
Llabel	Represents a branching label in the script
Ipatnum,label	If the pattern "patnum" was matched in the last **W**ait command, goto "label". Note that patnum=0 means if the wait timed out.
Glabel	Goto label unconditionally
Mtext	Display "text" in a message box.

E	End the script
Tfile	Transmit file via XMODEM
RAfile	Receive ascii file "file" via XMODEM
RBfile	Receive binary file "file" via XMODEM

Following is a sample logon script for CompuServe:

```
SAT
W5,OK
SATM0DT6086021
W60,CONNECT,NO CARRIER
I0,NORESP
I2,NOCARRIER
S
W10,:
I0,NORESP
SCIS
W10,:
I0,NORESP
S77777,7777
W10,:
I0,NORESP
Spassword
E
LNOCARRIER
MUnable to connect
E
LNORESP
MNo response from host
```

<u>**DESIGN OVERVIEW**</u>

This second release of VBTerm is a bit cleaner than the first, but also a bit more complex. The code is highly modularized, broken into forms and code modules along functional lines. The key modules in the application are:

GLOBAL.BAS	Those things that must be in the global module, and global variables and constants
CONFIG.FRM	The form that solicits the com settings from the user.
FILECREAte.FRM	The form that solicits a new file name from the user (thus doesn't display existing file names).

FILEOPEN.FRM	The form that solicits an existing file name from the user.
RX.FRM	The form (and code) that performs an XMODEM download.
TTY.FRM	The main terminal display screen.
TX.FRM	The form (and code) that performs an XMODEM upload.
SCRIPT.BAS	The code module that implements the VBTerm scripting language
SERIAL.BAS	The code module that provides the interface to the Windows serial API calls.
VT100.BAS	The code module that implements the VT100 emulator.

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.

I've sprinkled comments thoughout the program, and the best place to figure out how things work is to look at the code. However, some notes on indivual forms/modules:

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.

VT100.BAS:
Driving the screen is a bit 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! ;-)). **Note, however, that all the code relating to attribute support is commented out; this is for performance reasons.**

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.

TTY.FRM:

This form ties the application 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.

The code to implement this is in the routine **ReceiveLoop()**. It is probably unnecessarily complex as I have played with it quite a bit to improve performance.

The main concept of this routine is to just loop, checking the com port for new data and, if finding any, passing it along to the vt100 emulator. The complexity starts coming in when we deal with a few performance issues. The first involves buffering text to write to the window. Since its expensive to make the TextOut call to windows, I want to buffer up as much as I can before doing this. But how much is "enough"? The vt100 emulator will automatically flush the buffer whenver it receives a control character, so CRLFs, backspaces, etc., will force a write to the screen. But what if you've received only part of a line -- as when you are sitting at a prompt at CompuServe? To handle this, the program notices how long its been since data was received. If it's over a maximum (currently, 110 ms), it flushes whatever has been received, even if only a partial line.

This works wonderfully, except in one case -- where the host is echoing data you type. In that situation, you want to see the text instantly, not a bit later. A slight delay will make the system feel sluggish. To get around this, whenever you type a key, the flush delay is dropped to zero -- text is written out immediately. A timer is set to go off a second later (long enough for the host to echo your data!) that will set the delay back to 110 ms. Of course, if you keep typing, the delay keeps getting reset to a second later.

Charles McGuinness
[76701,11]