HappyTerminal

A TTL-Level Terminal Emulator using ToothPIC

Richard Hoptroff

Draft 2

Feature article submission to Circuit Cellar

2300 words, listing 60 lines, 7 figures

All text and figures available in electronic format

HappyTerminal:

A TTL-Level Terminal Emulator using ToothPIC

Just 20 years ago, the most common method of communicating with a computer was a serial terminal. As you typed, the ASCII data was transmitted to the computer via an RS232 line to a mainframe. When a reply was received it was displayed on a screen (or if you're even older than me, on a printout).

For most people, those days are long gone. But since integrated circuits communicate serially, electronic engineers still need serial terminals in product development for evaluation of new devices, debugging prototypes and sometimes even to snoop under the hood of other peoples' products. Firing up HyperTerminal, the terminal emulator bundled free with Windows PCs, is common practice.

Using HyperTerminal has become more cumbersome as time has passed. Serial data inside circuits is usually at 5V (TTL) or 3.3V logic levels, rather than RS232 levels. And even RS232 ports are rapidly disappearing from PCs.

So now to use HyperTerminal, you need a USB to RS232 adapter and then a RS232 to TTL driver such as a MAX232 before you can even start talking.

ToothPIC: The PIC with Bluetooth

I was so used to using HyperTerminal that I was taken by surprise when a customer pointed out that ToothPIC, a microcontroller module designed by the company I work for, could do the job much more easily. It took less than a day to write the code, and I was kicking myself for not having thought of it a long time before. I decided to call the application HappyTerminal, which may give you an idea of why I'm never very welcome in marketing departments.

The ToothPIC module combines a PIC18LF6820 microcontroller with an integral Bluetooth radio and a few discrete components such as a voltage regulator (see figure 1). The 18LF6820 contains all the I/O and peripherals you would expect from a high-end PIC such as 12 analog inputs and 5 PWM outputs and, crucially for HappyTerminal, two serial UARTs. Once of these is dedicated to communication with the Bluetooth radio; the other will be used by the terminal emulator. In addition, it has two analog comparators which will also be used for signal buffering and inversion where required. The Bluetooth radio has a built-in antenna that allows serial connections to be made with other Bluetooth-equipped devices within a 100m range.

The most important feature of the ToothPIC is the *ToothPIC Services* firmware layer. This is a linkable software library that provides functions for Bluetooth communications. Two particularly advanced features are wireless field programming and the user interface server.

The Wireless Field Programming feature allows the microcontroller firmware to be programmed from any Windows PC or Pocket PC via Bluetooth; a conventional programmer is not required. As a tech support guy, this is fantastic – I can email upgrades to my customers.

The user interface server allows ToothPIC to create user interfaces on remote devices such as Windows computers, Pocket PCs and high-end cellphones. The server on the ToothPIC stores the user interface information – dialogs, controls, etc. Any remote device running user interface client software can connect to the server, read the information and display the user interface. The client software is freeware. It is also generic: it does not need to be customized for each new application. (See reference [1].)

For the end-user, the main advantage of the user interface server is reduced cost, since display, keyboards, etc are not required on the product containing the server. In addition, the platform-independent nature of the communications protocol allows many different devices to connect to the product – perhaps even devices that don't exist yet. For the developer, the

clear advantage is that no product development is required on the remote devices. Only the server needs to be programmed.

In HappyTerminal, the server will be used to 'emulate' the serial terminal on either a Windows PC or a Pocket PC. Figure 2 shows the HappyTerminal application in practice. The module is physically connected to the serial device under test and, via Bluetooth, to a Pocket PC or a Windows PC. To develop the product I used the following 3-step application development process:

Designing the user interface: the development of an easy-to-use product should always start with a clear idea of how a user would want to operate it. All technical considerations should work around this, not vice versa.

Hardware design: considering how to use the available technology available, in this case on-board peripherals, to achieve the aims of the application.

Firmware design: gluing the available technology to the user's needs.

User Interface Design

The user interface is stored on the user interface server on ToothPIC. When a client device connects to the ToothPIC, the user interface is transmitted to it and it is displayed (figure 3). Freely distributable user interface client software is available for Windows PCs, Pocket PCs, Microsoft Smartphones and Java cellphones. The advantage of the client/server approach is that no development is needed on the client device – the software is already written and is free – and the server works 'straight out of the box' with a variety of client devices.

A terminal emulator requires a large graphical display area so for this application we will focus on the Windows PC and Pocket PC clients. The user interface is designed using the Designer software in the ToothPIC development kit. Controls (buttons, images, list boxes, etc) can be created for a number of 'dialogs'; the ToothPIC chooses which dialog is displayed at any particular time.

For HyperTerminal, two 'dialogs' are defined, one for the main display screen (figure 4) and one to modify the terminal emulator settings (figure 5). On the main screen, 20 single-line text controls make up the received data display. Optionally, the received data can be displayed as hexadecimal and ASCII text side-by-side (visible on the screen in figure 2), so a further 20 lines are

defined to display the second data format. At the bottom of the screen is an edit text control with an OK button. Text to be transmitted is entered there and it is transmitted when the OK button is pressed. On the settings screen, various list, edit and check boxes are used to specify settings.

Inside the Designer software, the controls for the two dialogs are first created and their logical properties defined (figure 6). Then they are laid out as they finally appear on the Pocket PC (figure 2) or Windows PC (figures 4, 5). The software has a simulation mode allowing the actual appearance to be tested.

When the user interface has been laid out, Designer creates a `.c` source code file to be linked into the application containing the user interface information required by the user interface server. A `.h` header file is also created containing useful macros for accessing that information. For example, the macro:

```
Get_Echo_29( pUChar )
```

retrieves the value of the check box that selects whether transmitted characters are echoed to the display screen.

Hardware Design

As already mentioned, the PIC has a spare on-board UART which can be used for serial transmission and reception. However, it is limited in that it cannot accept inverted serial data, and it cannot accept 3.3V logic.

Inverted serial data is generated by 'port-powered' RS232 devices such as BASIC Stamps (see reference [2]). Port powered serial lines obtain their TxD negative voltage supply from the RxD line. The result is that when connected to an RS232 device, they appear to operate at RS232 voltage levels; when connected to a TTL device such as the ToothPIC, they work at TTL levels, but the signal is inverted.

3.3V logic levels are common and it would be nice to connect these directly to the RxD pin of the ToothPIC. Unfortunately, the RxD UART input has a Schmidt trigger and will not accept 3.3V as logic 1 when powered at 5V.

Thankfully, both of these drawbacks can be overcome using the PIC18F6720's on-board comparators. These can be configured to compare an input pin relative to 2.5V and output high or low accordingly. Thus they can be configured as buffers or inverters as required. The various possible operating modes are shown in figure 7.

The ToothPIC has two on-board LEDs. The green LED will be used to indicate that a remote device has connected. The red LED will be used to indicate that an error occurred on the UART, most likely a baud rate mismatch.

Firmware Development

The firmware is developed in MPLAB, the PIC development environment provided by Microchip Technology, and their C18 compiler. Once the machine code has been generated, it can be programmed into the ToothPIC using a conventional programmer such as the ICD2 debugger, or directly via Bluetooth without the need for any programming hardware at all.

The ToothPIC Services firmware layer operates more or less transparently. Calls can be made to it at any time; if it has any news for the main application, it calls one of the six following callback functions which the developer must provide:

`main()`, where control is passed once the ToothPIC Services firmware layer has initialized.

`HighInterrupt()`, if a developer-initiated high priority interrupt occurs.

`LowInterrupt()`, if a developer-initiated low priority interrupt occurs.

`ErrorStatus()`, if a ToothPIC Services error occurs.

`BMTEvent()`, if an event occurs to the Bluetooth radio, such as connection or disconnection.

`FxPEvent()`, if an event occurs on the User Interface server, such as a user pressing a button on the client.

The important elements of the source code will now be discussed. The main source file is `HappyTerminal.c` as detaied in the *Project Files* section. The callback `FxPEvent()` is shown in listing 1.

Most of the code is event driven and processed from within callbacks. To allow the various callbacks to communicate with each other, the static variable `HTSemaphores` is used. One callback can change an `HTSemaphores` value and another can inspect it. Also, a static 256-byte buffer `HTRxBuff` is declared where received data is temporarily stored while it is waiting to be processed.

Since the code is largely event driven, the `main()` routine has very little in it (see project files for listings). After initializing the user interface server and the UART, it simply flashes the green LED. If a UART error is detected, it turns the red LED on for half a second before resetting the UART.

The high priority interrupt is intended for high-speed, short duration event processing. Low priority interrupts can to take as long as they like but they may be suspended to service high priority interrupts. If neither type of interrupt is being processed, the main program loop executes. In the HappyTerminal application, the high priority interrupt code stores received bytes in a receive buffer (see project files). A software interrupt flag is then raised to tell the low priority interrupt that there is data in the buffer to process. The low priority interrupt can then remove the buffered bytes at leisure and place them in the main display screen.

In the HappyTerminal, the `ErrorStatus` and `BMTEvent` callbacks are not expected to be called. If they are, the `BreakPoint` function is called. This flashes an error number on the LEDs.

The `FxPEvent` callback demonstrates how to access user interface information and is shown in part in listing 1. `FxPEvent` is called whenever a user modifies a control on the user interface. For example, when the user

enters text to be transmitted by HappyTerminal, the text control is read using the `pOutCharsOK_20` macro which is defined computer-generated header file created by the Designer application. The same macro is used for clearing the text control, but in addition, an `FxPC_CtlUpdate` command is sent to the server to tell it that the contents have been modified. The server will then update the client. Processing of other control events is not shown in the listing but is provided in the project files.

Using the Terminal Emulator

I've had HappyTerminal in my lab for two weeks now and my natural reaction is to turn to it rather than HyperTerminal whenever I need a terminal emulator. So it passes the customer satisfaction test. It is much easier to set up than the RS232 signals levels required by HyperTerminal, which was the principle aim of the project. In addition, the 'debug mode' has also proved more useful than expected, particularly for finding out what's going on in circuits designed by other engineers.

The speed with which the HappyTerminal was developed using ToothPIC has also been a pleasant surprise. The ToothPIC Services firmware layer did much of the work for me in terms of creating a user interface and managing

Bluetooth communications. All I had to do was write the last ten percent that was specific to my application.

About the author

Richard Hoptroff is a development engineer for FlexiPanel Ltd of London, UK, and co-author of *Data Mining and Business Intelligence: A Guide to Productivity*. He holds a PhD in Physics from London University and his interests include travel, travel and more travel. He may be contacted at *rhoptroff@flexipanel.com.*

Project Files

Project files:

```
HappyTerminalRes.FxP
HappyTerminalRes.c
HappyTerminalRes.h
HappyTerminal.mcw
HappyTerminal.mcs
HappyTerminal.mcp
HappyTerminal.c
ToothPIC.c
ToothPIC.h
```

Field programming files:

`HappyTerminalWin.exe` (Windows)

`HappyTerminalPPC.exe` (Pocket PC)

Resources

`ToothPIC.pdf` documentation

`Designer.exe` user interface design tool

`Designer.pdf` documentation

References

[1] *Bluetooth & Remote Device User Interfaces*, R Hoptroff, Dr Dobb's Journal September 2004, pp61-65, CMP Media, San Francisco.

[2] *BASIC Stamp 2p24 Module Schematic*, Parallax Inc, http://www.parallax.com/dl/docs/prod/schem/Bs2p24revB.pdf.

Sources

ToothPIC distributor:

(Will ship to the USA)

R F Solutions Ltd
Unit 21, Cliffe Industrial Estate,
Lewes, E. Sussex BN8 6JL, United Kingdom
email: sales@rfsolutions.co.uk
http://www.rfsolutions.co.uk
Tel: +44 (0)1273 898 000, Fax: +44 (0)1273 480 661

MPLAB and C18 development tools:

Microchip Technlogy Inc
2355 W Chandler Blvd
Chandler, AZ 85224-6199, USA
http://www.microchip.com

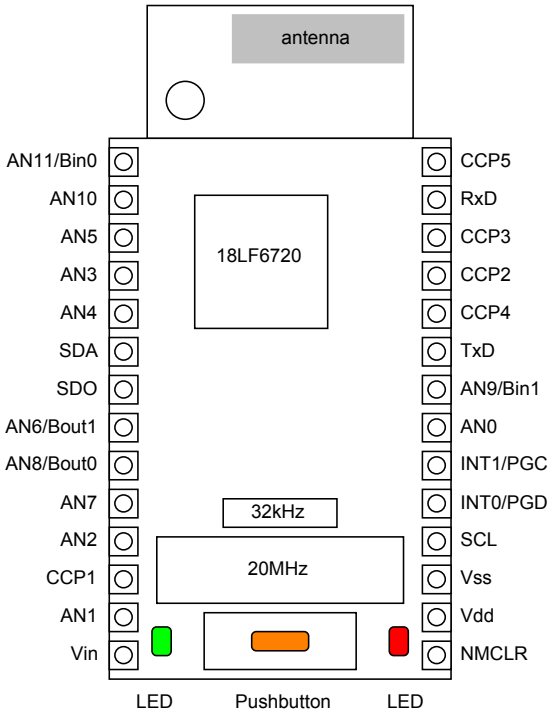

Figure 1. Pinout of the ToothPIC module. The FCC certified Bluetooth radio and the voltage regulator are mounted on the underside of the board.

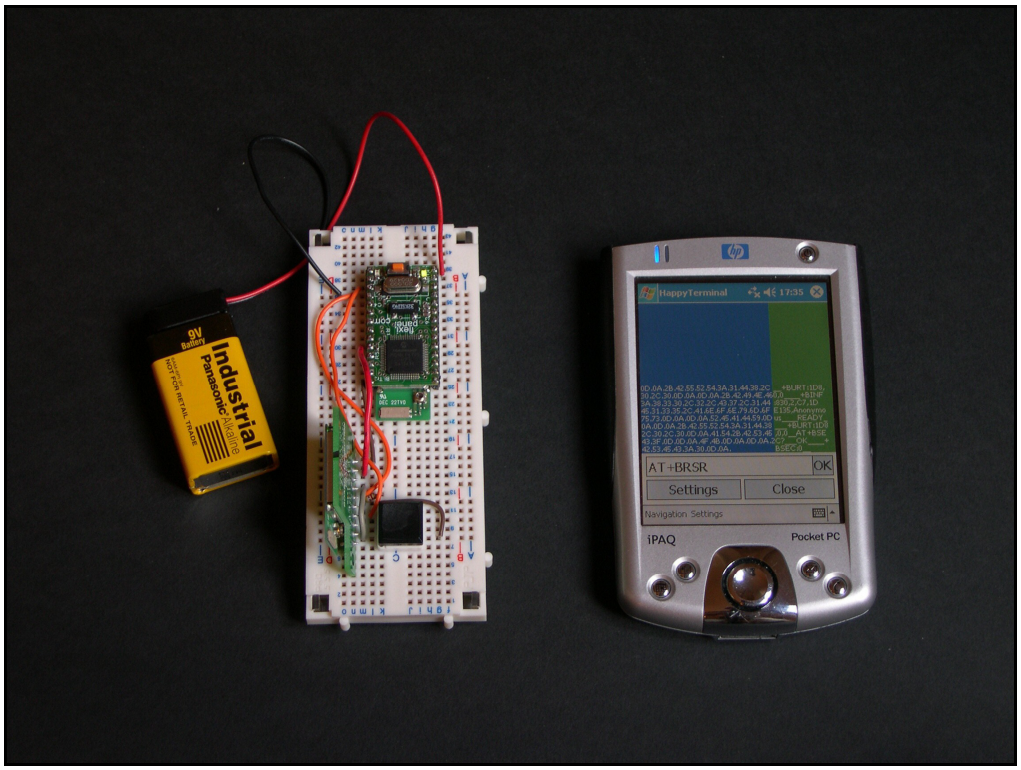

Figure 2. The ToothPIC terminal emulator (top) with serial connection to a microcontroller module under test (bottom). The serial terminal is 'emulated' on a Pocket PC or Windows PC using the Bluetooth link.

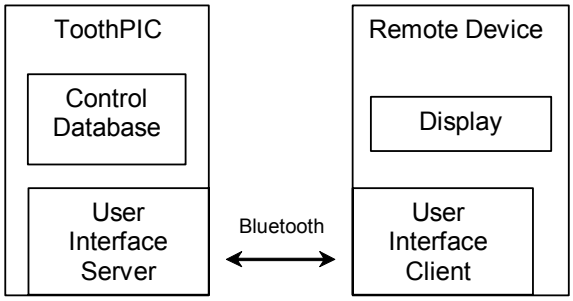

Figure 3. The user interface is managed by a server on the ToothPIC and displayed by a client such as a Pocket PC or Windows PC.

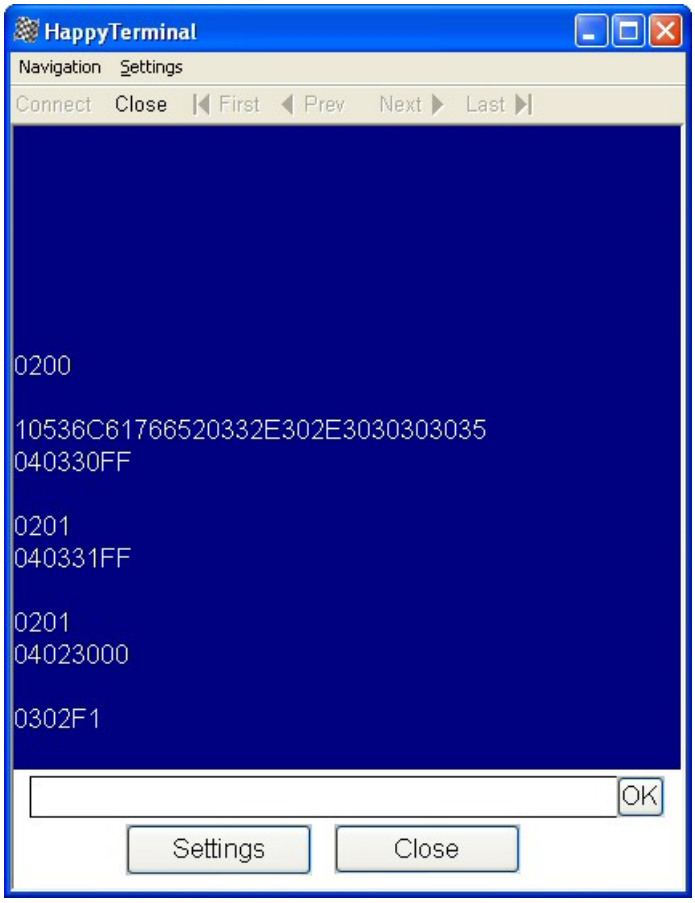

Figure 4. Main terminal emulator screen as it appears on a Windows PC.

The equivalent appearance on a Pocket PC is shown in figure 2.

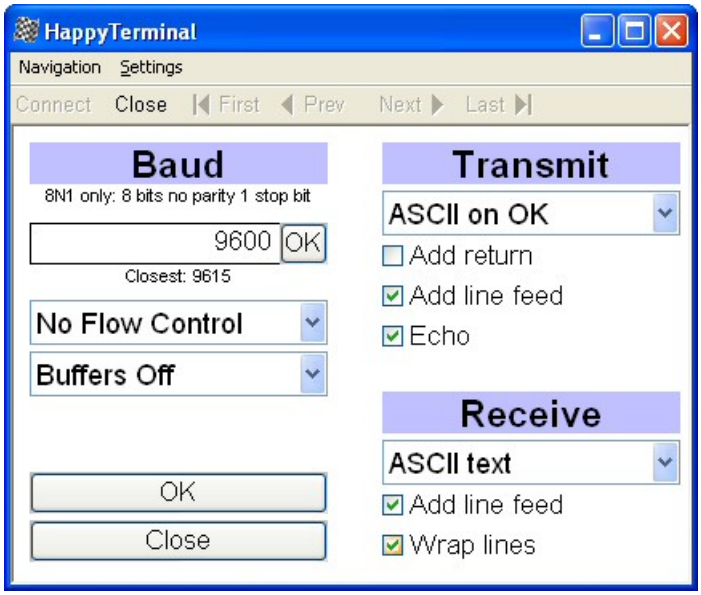

Figure 5. The settings dialog as it appears on a Windows PC.

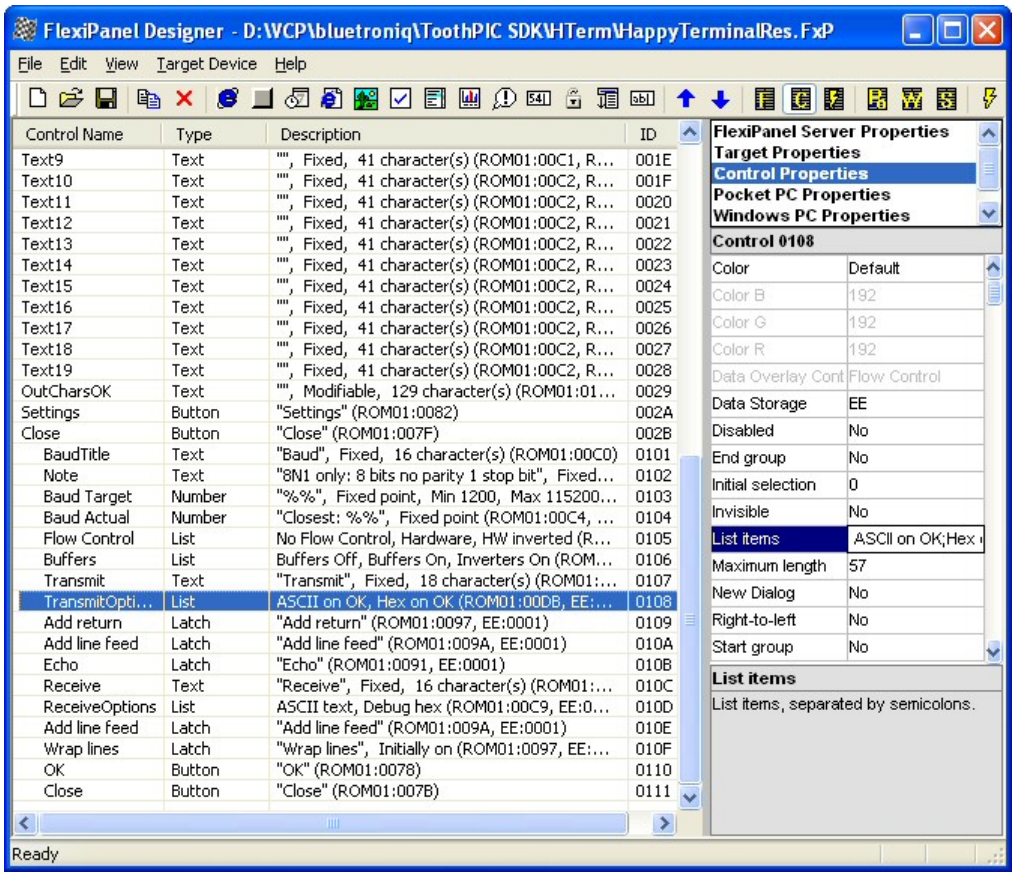

Figure 6. The user interface controls being created using the Designer software.

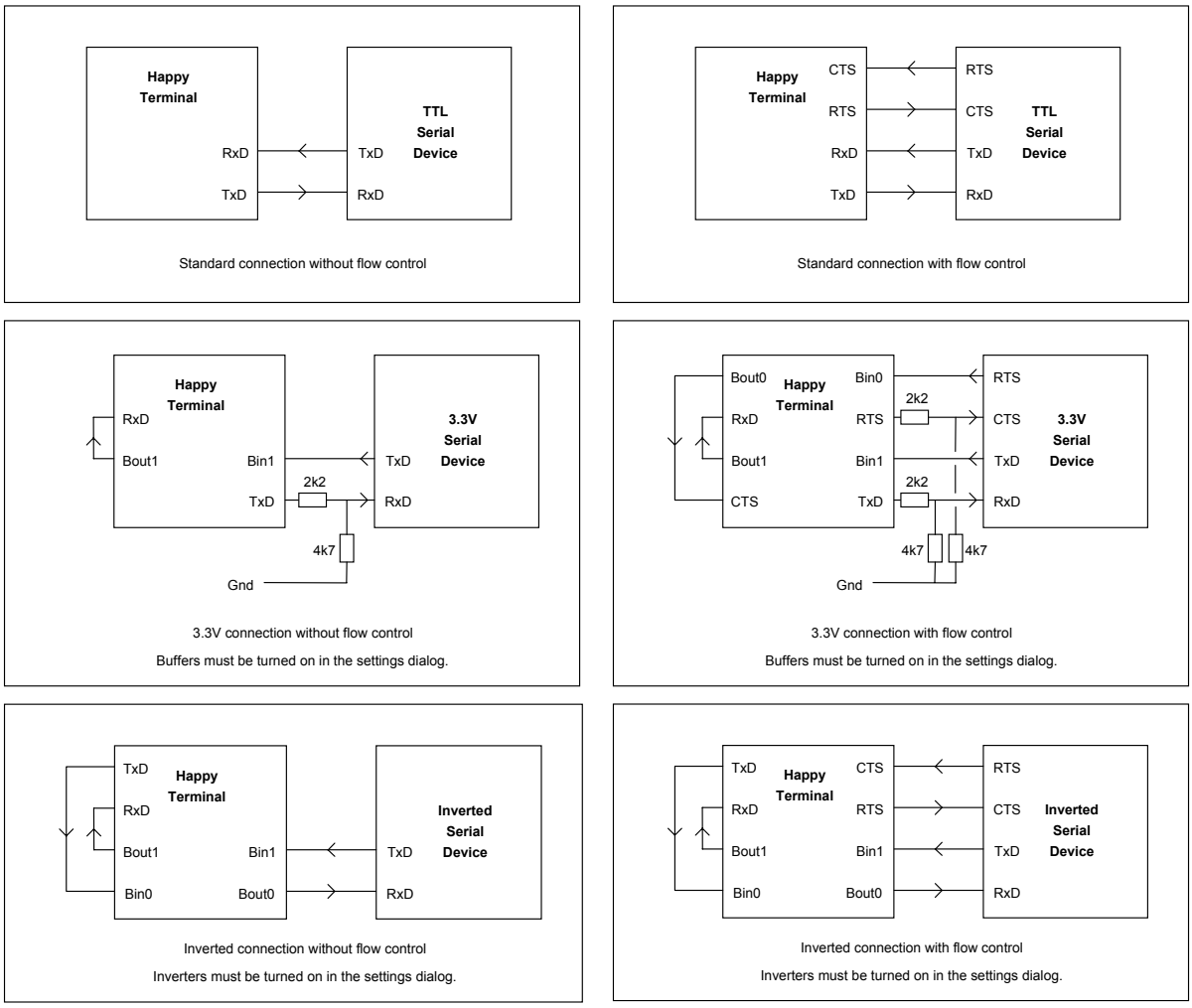

Figure 7. Various pin configurations for the terminal emulator.

```
void FxPEvent( unsigned char EventID, void *pData )
{
    if ( EventID==FxPE_ClntUpdate )
    {
        if (*((unsigned short*) pData) == ID_OutCharsOK_20)
        {
            // transmit text OK button pressed
            unsigned long bHex;
            unsigned char bEcho;
            unsigned char bCR;
            unsigned char bLF;
            unsigned char ChLoc;
            unsigned char bError = 0;

            // Read control setting using Designer-generated macros
            Get_TransmitOptions_26( &bHex );
            Get_Add_return_28( &bCR );
            Get_Add_line_feed_6F( &bLF );
            Get_Echo_29( &bEcho );

            // read data and transmit it
            for (ChLoc = 0; pOutCharsOK_20[ChLoc]!=0 && ! bError; ChLoc++)
            {
                unsigned char byte;
                // get byte
                if (bHex)
                {
                    byte = ConvertToASCII( ChLoc );
                    ChLoc+=2;
                }
                else
                {
                    byte = pOutCharsOK_20[ChLoc];
                }
                // transmit character
                TxChar( byte, bEcho );
            }

            if (bCR) TxChar( '\r', bEcho );
            if (bLF) TxChar( '\n', bEcho );

            // clear edit text
            if (bError)
                Set_OutCharsOK_20( szError, 0 );
            else
                pOutCharsOK_20[0] = 0;

            // update client
            FxPCommand( FxPC_CtlUpdate, ID_OutCharsOK_20, 0 );
        }

        else
        {
            // process other control events
        }
        return;
    }
}
```

Listing 1. `FxPEvent()` reports when a user modifies a control. (Not all control events are shown.) `ConvertToASCII()` converts hexadecimal characters into a binary byte; `TxChar()` transmits a byte to the serial UART; these routines are detailed in the project files.