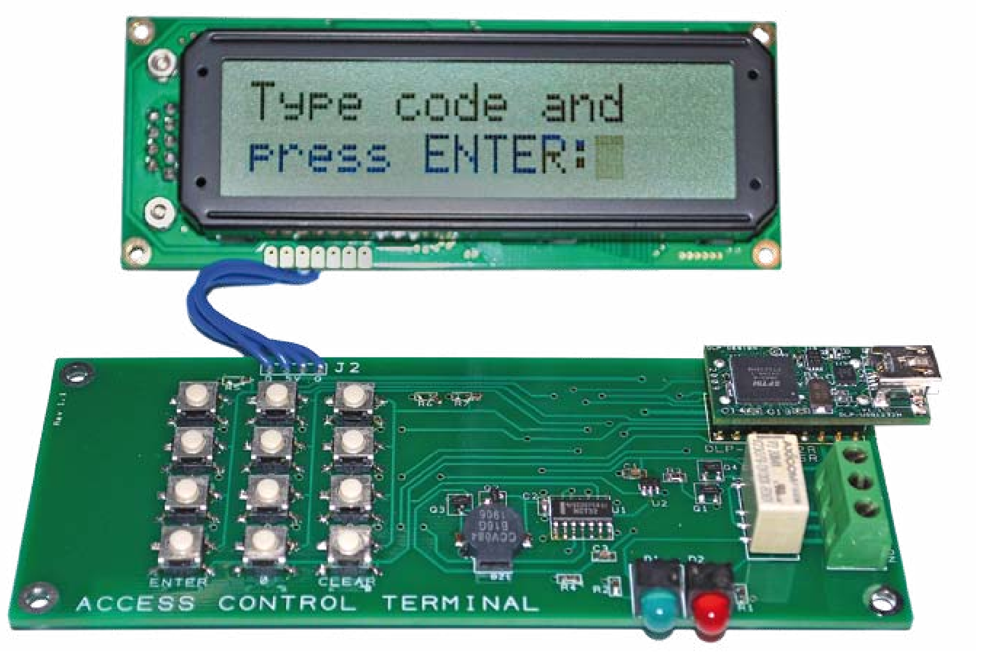
Bit-banging the FTDI-USB Module

Taking advantage of little-known features of FTDI's USB ICs

By Don Powrie (USA)



This article describes the electrical design and software requirements for a keyless entry control panel comprised of a numeric entry pad, an LCD display, relay contacts for unlocking a door and a USB interface. Even though this writing will delve into the inner workings of FTDI's FT2232H and its Bit-Bang Mode, understanding the technology will require neither an in-depth knowledge of USB nor the use of a microcontroller!

I'll begin with the assumption that the reader is already somewhat familiar with FTDI's line of easy-to-use USB ICs before diving into a couple of their lesser-known characteristics. If you haven't been exposed yet to these devices, then I might suggest boning up on their capabilities and applications by reviewing some of my earlier publications at [1].

Returning to the project, all user software will reside in a single application on the host PC, and the only ICs used in this design are the FT2232H and a couple simple logic gates. The FTDI module used is available for purchase from DigiKey, Mouser Electronics and FTDI's other distributors.

Bit-bang basics

Once the USB drivers have been loaded onto the PC and the port is open to the USB module (I used the DLP-USB1232H to make assembly easier), the Bit-Bang Mode can be enabled. The VC++ source code for this project is available for download from the project webpage [2]. The D2XX command for enabling the Bit-Bang Mode is FT_STATUS status = FT_SetBitMode(m_ftHandle, 0x01, 0x01) where the handle is returned from the call to open the port, the second parameter is used to select which of the eight data lines are inputs or outputs and the third parameter is the initial high/low state for the lines configured as outputs.

To read the high/low state of the IO lines that are configured as inputs, you would use the FT_GetBitMode(m_ftHandle, &data) function. The 'data' parameter points to the current state of the inputs. The important thing to keep in mind is that this function returns the instantaneous state of the inputs. Conversely, data that is written to the module (using the FT_Write() function) does not immediately appear on the output pins. Instead, the data appears at a preselected update rate. If the update (or baud) rate is currently set to 9600 and you send multiple bytes of data all at once, then each byte will automatically appear on the output Lines — one at a time — every 104 µs until all bytes have been issued.

FTDI's USB chips have always been able to do this. However, with the introduction of their new high-speed chips, the update rate can now be accurately controlled, and up to 8 serial streams can now be generated at precise baud rates to drive serial devices at stable baud rates. For example, the following code will set the update rate to the baud rate required by the LCD module and the TTL serial interface that I utilized in this project:

```
div = 0x8c30;//35888 decimal for 19200 baud to LCD with 0.6% error
status = FT_SetDivisor(m_ftHandle, div);
```

Note that the serial data can only be clocked **out** at a controlled rate. Unfortunately, no serial reply data can be clocked back in on an input line. You would have to use the second channel of the USB IC to receive return data; but that's OK for this project since we are only driving an LCD display (Crystalfontz America part # CFA632-YFB-KS) with TTL serial data, and we don't care about return data.

Now that we have eight controllable I/O lines that can also clock out TTL serial data at controlled baud rates, the platform is set for our project.

One 8-Bit variable

The host app keeps track of all inputs and outputs, including the serial data stream to the LCD, using a single 8-bit variable. To read the high/low state of an IO line configured as an input, you would call the FT_GetBitMode() function and mask the return variable so that you can look at a single bit. To change the high/low state of an output, you should first update the state of the bit in question in the 8-bit variable and then write out the byte.

So far so good... but what if you want to send a serial data stream of 200 bytes on

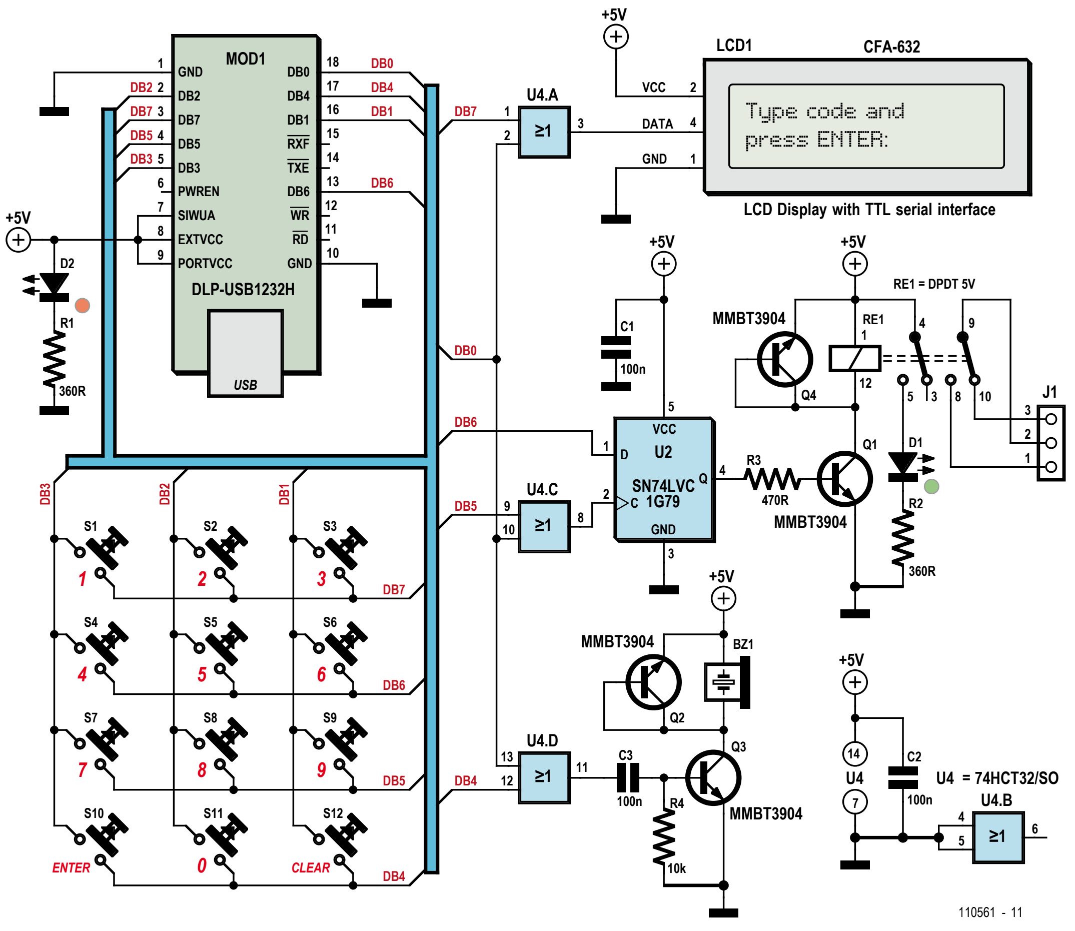


Figure 1. The DLP-USB1232H module after being suitably bit-banged acts as the controlling element of a code lock.

one of the eight I/O lines without affecting the other seven? That’s right; you build a 1,600-byte buffer where each byte has only one bit that gets changed according to the next bit that is to be clocked out serially at the next timer tick, and then you send the entire buffer with the FT_Write() function all at once. Tedious? Yes! But computers love doing tedious tasks, and you only have to write the software once for clocking out long serial strings.

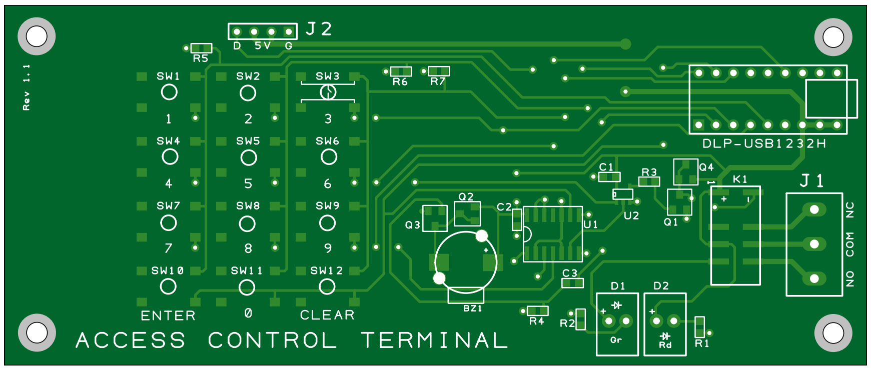


Figure 2. Component side layout of the circuit board designed for the code lock (here at 80% of its true size). The Gerber files may be downloaded from the Elektor website [2].

Hardware

For the following, refer to the electrical schematic shown in **Figure 1**. To scan the 12 keys in the numeric entry pad using seven of the eight available I/O lines; you just drive the DB4, DB5, DB6 or DB7 ‘row’ lines low (one at a time) and look at the state of the three ‘column’ lines connected to DB1, DB2 and DB3. If a switch is pressed, then the corresponding column reports a low level on its I/O input line.

DB0 controls whether the host is reading the keyboard or driving the LCD display, relay or beeper ‘devices’. When DB0 is logic High, the OR gates all block data from driving these devices.
When Low, the keyboard is ignored, and data can be written to these devices via DB4 through DB7.

By now you have probably surmised that holding a keyboard switch down will disable the host’s ability to write to one or more of the devices. You can get around this somewhat by waiting in the host app for each key press to be released before proceeding. There is almost always a way to break a design if you go looking for one, but then this system is designed to keep someone out of a locked area. If they hold a key pressed, then they’re definitely not getting in.

The Gerber files for making the circuit board for the project may be downloaded free from [2]. The component mounting plan appears in **Figure 2**.

Bit-Bang++...+?

At first I was tempted to present a project in which the hardware was comprised of eight TTL serial LCD displays all connected to a host PC using just the eight I/O lines and the Bit-Bang Mode. That would have worked fine, but it really didn’t present much of a challenge. It would also have been more expensive. The Bit-Bang Mode can also be used for more mundane tasks like controlling eight relays or simple digital I/O. More adventurous types can try controlling multiple SPI devices such as A/D and DACs.

I guess the primary take-away from this article is that you don’t necessarily need a microcontroller — and its associated firmware development — in order to use the USB interface to control the world around you. The Bit-Bang Mode can be a perfect low-cost solution for systems requiring only host-side software to connect to the environment outside of a PC.

(110561)

Internet Links

[1] www.dlpdesign.com/pub.shtml

[2] www.elektor.com/110561