Brian's corner

The FTDI EVE graphics controller (3)

svet ELEKTRONIKE

We at Svet elektronike are proud on what we do since 1994!

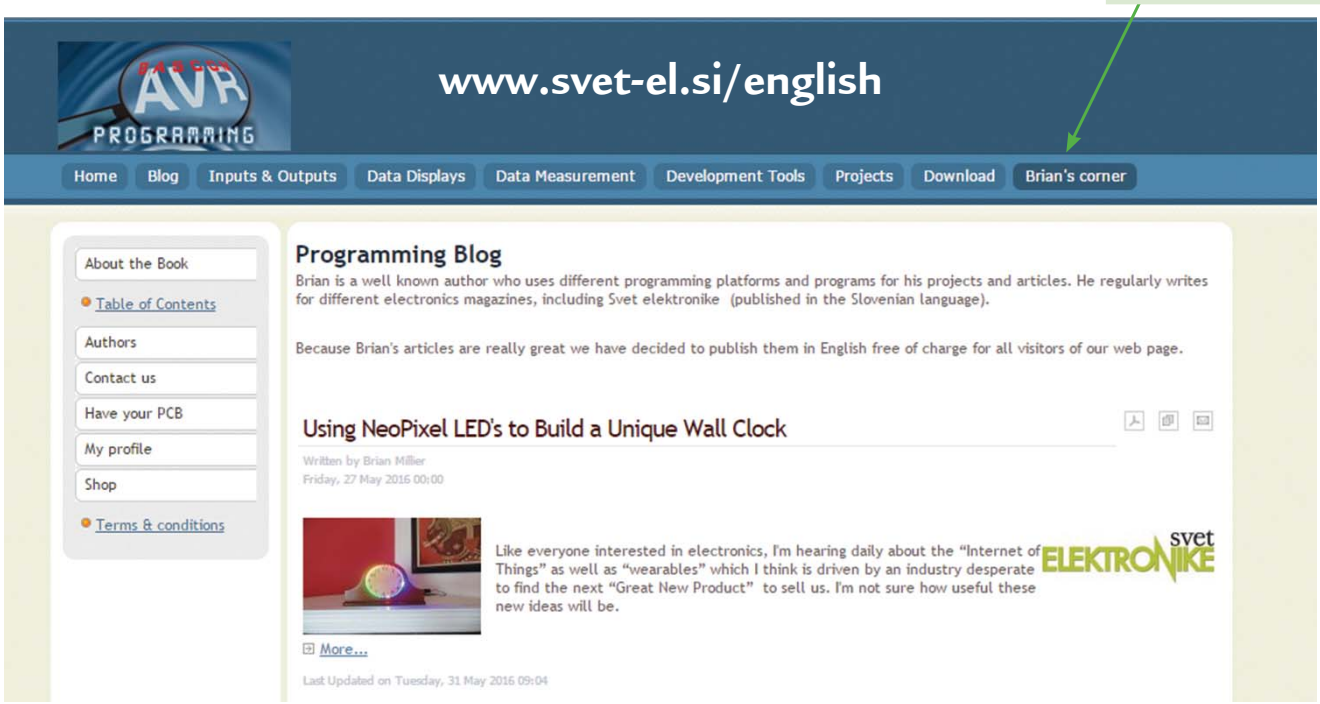
Slovenian website

www.svet.el.si

English website

www.svet-el.si/english

Brian's corner



Article on the web site

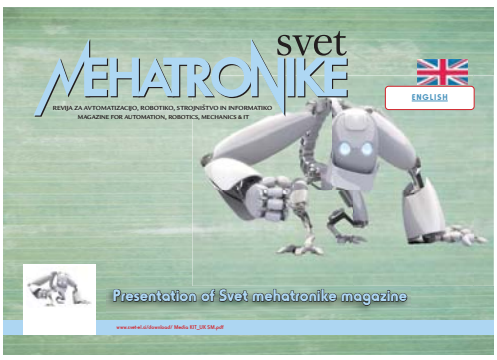
Download programs
and
Download PDF of the article

WWW.SVET-EL.SI

AX elektronika d.o.o.
Špruha 33
SI-1236 Trzin
Slovenia / Europe

00386 1 549 14 00
www.svet-el.si
stik@svet-el.si

WWW.SVET-ME.SI



Introducing some useful FT800 GUI Routines

At the end of last month's EVE article, I showed you the basic code needed to initialize the FT800 controller and to produce a simple start-up screen.

I mentioned that you must do some form of touchscreen calibration if you wish to use the touchscreen capability. FTDI makes this process simple by defining a high-level graphics co-processor command that does all of the necessary work. You need to include the following line in your program to start this calibration routine:

```
Ft_Gpu_CoCmd_Calibrate(phost,0);
```

You should note that you must call this command from within a routine that first sets up the graphics co-processor ring buffer properly, and afterwards initiates the actual transfer between the ring buffer and the FT800 co-processor engine. Here is how this is accomplished in the FTDI sample programs, ***program 1.***

Photo 1

LISTING 1

Most of the commands have already been covered in part 2, so I won't explain them again. The calibration routine places three small, flashing circles on the screen, one at a time, and you have to accurately tap each one of them as close to the center of the circle as possible. The routine makes use of the FT800's audio functions, so if you are using an FTDI VM800B43A evaluation module which contains an audio amplifier/speaker, you will hear a beep after you tap each circle.

What this routine does is to measure the touchscreen's ADC readings at the three pre-defined co-ordinates and then do some math to generate six 32-bit calibration constants. These six constants are automatically stored in the FT800's register space in the following locations, ***program 2*** **.**

These registers are part of the FT800's SRAM array, so they are volatile: i.e. they do not survive after the power is removed. Normally this would mean that you would have to do this calibration every time that you powered up your project, which would be quite inconvenient. To get around this, I added the following code right after the calibration routine shown in Listing 1, ***program 3.***

The second and third statements are repeated six times, once for each of the REG_TOUCH_TRANSFORM_ A

Program 1

```
Ft_Gpu_CoCmd_Dlstart(phost);
Ft_App_WrCoCmd_Buffer(phost,CLEAR_COLOR_RGB(64,64,64));
Ft_App_WrCoCmd_Buffer(phost,CLEAR(1,1,1));
Ft_App_WrCoCmd_Buffer(phost,COLOR_RGB(0xff,0xff,0xff));
Ft_Gpu_CoCmd_Text(phost,(FT_DispWidth/2), (FT_DispHeight/2), 27, OPT_CENTER,"Please Tap on the dot");
Ft_Gpu_CoCmd_Calibrate(phost,0);
Ft_Gpu_CoCmd_Swap(phost);
/* Wait till coprocessor completes the operation */
Ft_Gpu_Hal_WaitCmdfifo_empty(phost);
```

Program 2

```
REG_TOUCH_TRANSFORM_A
        through
           REG_TOUCH_TRANSFORM_F
```

through F registers. I then copied down the 6 constants that were displayed on the Arduino's Serial monitor window. After that, whenever I wrote a new program using the FT800 controller, I just added the following routine to restore these constants to the proper FT800 registers, ***program 4.***

LISTING 2

When I wrote parts 1 & 2 of this article series, I had been strictly dealing with two Mikroelektronika's Connect EVE 4.3" display modules. Both displays worked nicely with the constants that you will find in the program listing associated with part 2 of the article series (on the SE website).

However, I have now started using the FTDI VM800B43A 4.3" evaluation modules, and have found that the six constants that you get when you run the calibration routine on these modules are much different. So different in fact, that none of my previously written routines worked properly when using the constants collected from my Connect-EVE displays! The constants that you see in Listing 2 above, are those that I recently obtained after running the calibration routine on an FTDI VM800BC43A module that I am currently using. I should mention that these calibration constants are defined by FTDI to be 32-bit fixed point numbers,with the implied decimal point situated between bit 15 and 16. When you read these variables using the following statement, you can either define the variable calreg as a long or an unsigned long integer, ***program 5*** **.**

It doesn't matter which method you use- in the program that I supplied with part 2 of the series, I happened to choose an unsigned long variable, so I ended up with some very large numbers corresponding to the negative values shown for REG_TOUCH_TRANSFORM_C and E above.

Now that I know there are big differences in the calibration constants obtained using Connect-Eve and VM800BC43A modules, I am going to change my approach to handling calibration to the following:

- At program start, I will allow the user the option of running the calibration routine (Listing 1 above)
- If that option is taken, I will then retrieve the 6 calibration constants, as shown above, and store them in the AVR MCU's EEPROM.
- If the calibration option is NOT chosen, I will run a routine that retrieves the six 32-bit constants from the AVR MCU's EEPROM, and store them in the REG_TOUCH_TRANSFORM_A through F registers, in a similar way to that shown in LISTING 2.

An Important Correction

When I was trying out the FTDI-supplied sample programs mentioned in this article, I came across an inconsistency in FTDI's documentation/sample applications. If you follow the FT800 hardware signal connection shown in Table 1 of application note "AN_246 VM800CB_SampleApp_Arduino_Introduction", then the FT800's CS# signal is shown connected to Arduino Digital pin 10 (M328 PB2 -SS). I used this wiring when I first started using FT800 displays-even though I was using a MikroElektronika Connect-EVE display. I found that the FTDI-supplied sample programs didn't work. The reason is that all FTDI-supplied sample programs contain an FT_Platform.h file, in which there is the definition:

```
#ifdef  FT_ATMEGA_328P
#define FT800_CS (9)
```

This incorrectly assumes that the FT800's CS# line is connected to Arduino Digital pin 9, not pin 10 as dictated in the AN246 applicaiton note.

You have to change this to (10) in order for these sample programs to work!

I corrected the FT_Platform.h file early in my FT800 development cycle, and used this corrected file when writing my own applications, so I didn't think to mention FTDI's error until I started loading a few of their sample applications later on, for this article.

A Numeric Keyboard Routine

One of the reasons it makes sense

Program 3

```
long calreg;
calreg= Ft_Gpu_Hal_Rd32(phost,REG_TOUCH_TRANSFORM_A);
  Serial.println(calreg);
```

Program 4

```
void  storeTouchscreenCals()
{
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_A,33915);
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_B,337);
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_C,-1768013);
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_D,266);
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_E,-19237);
Ft_Gpu_Hal_Wr32(phost,REG_TOUCH_TRANSFORM_F,18768537);
}
```

Program 5

```
calreg= Ft_Gpu_Hal_Rd32(phost,REG_TOUCH_TRANSFORM_A);
```

to spend some extra money on a touchscreen display is that you can often eliminate other types of user-interface hardware such as numeric keyboards, potentiometers and switches. Good quality keypads, pots and switches are only getting more expensive while technology is driving down the cost of things like touchscreen display modules.

In their sample applications, FTDI provide the FT_App_ Keyboard program. This program provides a full QWERTY (upper/lower case) keyboard including numbers and punctuation. It includes a modest display region to hold a few lines of text, and some basic editing functionality. It you need all of this functionality, then I suggest you try to incorporate this program into your own application code. This FTDI program is quite a complicated program and somewhat hard to comprehend. Photo 1 shows this QWERTY keyboard.

For most of my projects, a full QWERTY keyboard is not necessary. Rather, I generally need only a numeric keypad, with the ability to enter numbers that may include a decimal point and may be positive or negative. The only "editing" capability I need is to be able to "back-space" in order to erase one or more incorrectly entered characters. The function I wrote to do this uses only about 50 lines of code and is reasonably easy to understand.

The basic FT800 "Widget" that you need to generate this keyboard is the Co-processor command CMD_KEYS, which draws a row of key buttons- the number of keys drawn is determined by the user, by setting the parameters properly. Here is a simple example, ***program 6*.**

Lets look at the parameter list. The leading "phost" is a "handle" which must be present when using any of the Co-processor "C" routines that are supplied in the FTDI FT800 driver code. Following that are the X,Y co-ordinates (in pixels) of the upper-left corner of the row of keys. The "180,50" specify the length and height of the row of keys (in pixels). The "27" is the font number- 27 is a good font size for a numeric keypad that takes up about 1/2 of the

Photo 2

4.3" display, and which can easily be typed on with a man's large fingers. The "0" following the font designates options: a zero provides the default 3D style keypad, However you can substitute the OPT_FLAT constant for a simpler 2D key style. There is also a constant OPT_CENTER which can be used to provide the smallest possible keys, which can hold the selected font, and fitting into the designated size (180W x 50H in this example). Absent the OPT_CENTER option, the keys will be drawn as large as possible while still fitting into the designated size.

The final parameters are enclosed in quotation marks and specify the alphanumeric legends that you want displayed on the row of keys, from left to right.

In my routine, I use 4 such statements to generate a 12-key numeric keypad with the bottom row containing the "-" symbol, the "0" and the decimal point symbol. Although it doesn't mention it in the FT800 Programmer's Guide, when you press one of these keys, the TAG value that is stored by the FT800's touchscreen engine is the ASCII value of the symbol you have designated for that key. So, for example, if you press the "0" key, the TAG value stored will be 0x30, corresponding to the ASCII value of a "0" symbol. If you don't remember what TAG refers to, look back at part 2 of the article series, where I explain the concept.

There are a few lines of code used to draw a rectangle around the keypad, as well as designate a small "window" at the top, to display the keys that have been pressed. I also needed an ENTER key as well as a Backspace key. I wanted these to be significantly larger than the numeric keys, and somewhat physically removed from the main keypad matrix. Therefore I generated them separately, using the button commands, as follows, ***program 7*.**

I explained the Button command in part 2, so I won't repeat the explanation of the parameters here.

Program 6

```
Ft_Gpu_CoCmd_Keys(phost,0,55,180,50,27, 0, "123");
```

Program 7

```
// TAG (identify the following button with the TAG "E"
Ft_App_WrCoCmd_Buffer(phost,TAG('E'));
 // place ENTER  Button
Ft_Gpu_CoCmd_Button(phost,200,220, 50,50,26,0,"ENTER");
  // TAG (identify the following key with the TAG "B"
 Ft_App_WrCoCmd_Buffer(phost,TAG('B'));
  // place Backspace button
Ft_Gpu_CoCmd_Button(phost,200,165, 50,50,27,0,"<-");
```

Program 8

```
Ft_Gpu_CoCmd_Text(phost,50,10,27,0,Entry_String);
```

When it comes to converting a series of characters into a floating point number, there are a number of ways to do it in "C", some of which are not part of the default Arduino syntax. Some of these methods are harder to understand than others (for a person like myself who is not a pro in "C" programming) . What I chose to do was to collect each key press and place it sequentially into the "Entry_String" character array. If the backspace key is pressed, I decremented the pointer into this array, and "erase" the last character entered, by replacing it with a value of 0 (the ASCII string terminator in "C"). When the "Enter" key is detected, the loop exits, and the "Entry_String" array is processed before returning from the keypad function.

The keypad routine operates as a loop in which the display is constantly being re-drawn with the keypad buttons and rectangular frame as well as the contents of the "Entry_String" character array, which changes as the user presses keys. The numeric entry contained in the "Entry_String" array is displayed in the top entry display window, using the following command, ***program 8*.**

The only other routine needed in this loop is the function call which checks to see if the user has touched the screen in the areas occupied by the keys. This is accomplished as follows:

```
tmp=Read_keys();
```

The Read_keys() function is one provided by FTDI as part of their sample programs. This function returns the TAG value of whatever key (or button, in the case of ENTER and BACKSPACE) has been pressed and released. I test the value of tmp, and if it is NOT a null (0), it means that a key has been pressed, and I store this TAG value, which is the ASCII value of the key symbol itself, into the "Entry_String" character array.

Whenever the ENTER key is detected, I exit the loop described above. Then I convert the character array into a floating point number and return it as the keypad function's value, as follows:

```
return atof(Entry_String);
```

The cryptic "atof" is the "C" ASCII to Float conversion function.

The Arduino demo program for this numeric keypad can be found on the SE website [1]. Photo 2 shows the keypad demo in action.

A Bar-graph to display Battery Voltage

Sometimes it is useful to be able to display a bar-graph to display battery voltage in a portable instrument. For this purpose it is nice to have the "bar" part of the graph change color, instantly giving you a rough idea of the battery condition. I use a color scheme like a traffic signal: green means the battery is fine, red means it is discharged and yellow means you have to be careful as the battery is getting close to being completely discharged. Of course, the length of the bar itself will give you a more quantitative idea of the battery's condition.

The widget that we will use for this is the progress bar. A photo of this, taken from the FTDI Programmer Guide, is shown in Photo 3. This is a standard progress bar using white for the bar.

Following is a code snippet showing how we can display a variable-color Progress Bar for battery reading, as described earlier, ***program 9*.**

LISTING 3

In the first line we call a function to retrieve the battery voltage (v), which I'll describe in more detail later. For now, all you need to know is that it returns a value between 0 and 100, representing the battery's charge on a percentage basis. Then there is a series of three conditional tests, which divide the range into three parts. For each of these ranges, I define a set of r1,g1,b1 values- these correspond to the RGB values that I will later assign to the "bar" section of the Progress Bar, using the Co-processor command "BgColor". After that is done all we have to do is issue the Co-processor command "Progress" with the proper parameters. In the example above, I place the progress bar at an X,Y co-ordinate of 350,200 and make the progress bar 80 pixels wide and 20 pixels high. The next parameter, a "0" draws the progress bar in a 3-D style. We then pass this routine the battery value (v), and define the full scale value of the bar as 100. That is all that is necessary to display a bar-graph of the battery voltage, with a dynamic color scheme, as described earlier.

Note that you must surround this code snippet with the "standard" lines of code which prepare the FT800's Co-processor ring buffer to accept data, and then swap and display the widget, like the examples described earlier.

The routine to read the battery voltage is completely independent of the FT800 controller routines, but I will briefly explain it. Before I describe the code, I should explain that in the project that this code was used, I powered the Mega328 AVR MCU using a 3.6V volt LiPo cell. These cells can provide up to 4.2 volts when fully charged, but nominally put out 3.6 volts for most of their discharge cycle. I fed the battery voltage to a low-dropout 3.3V regulator, which feeds both the Mega328 and the FT800 display module. When setting up the ADC on your AVR MCU, you should use its Internal reference and NOT AVcc, ***program 10.***

If you were to configure the ADC to use AVcc as its reference, when the battery got low in charge, the LDO regulator would no longer be able to supply the Mega328 with 3.3V. While the Mega328 would continue

to run, the ADC would now have a dropping reference voltage (AVcc) and would therefore report the battery voltage incorrectly (too high). Using the internal 1.1V reference avoids this. You must also use a voltage divider across the LiPo cell to reduce its 4.3V fully-charged value to something a bit less than the 1.1V reference. I used common 100K and 470K resistors, which divided the battery voltage by a ratio of 5.7.

Following is the code that I use to read the battery voltage, ***program 11.***

LISTING 4

To improve the ADC resolution, I take 16 readings and sum them. I am really only interested in battery voltages between 3.3 and 4.2 volts, since voltages less than 3.3V will not operate the circuit reliably. Given the ADC reference,the 5.7:1 resistive divider and the accumulation of 16 readings, this 3.3- 4.2 volt range will result in readings of 8000 to 11000. The C language has a Map command which will "map" this range into a 0 to 100 range, when used as shown above. So, the progress bar will have an active range of 3.3 to 4.2 volts, which is the useful battery range in this application. While probably unnecessary, I also used the Constrain function to insure that the readings are always in the 0-100 range.

In my project, I was somewhat limited in the amount of screen space I had available to display the battery voltage, which prompted me to use a small Progress Bar. However, as you can see in Photo 4, the FT800 also has a Gauge widget, which is similarly easy to use, and might be more attractive in your application.

X-Y Graphs

Much of the work I've done for my job at Dalhousie University involves reading sensors and plotting the data. Display modules based upon the FT800 are very well-suited for plotting data in graphical format. The FT800 controller is very fast, as is the SPI data pathway between your MCU and the FT800, so it is possible to plot graphs that change quite quickly with respect to time. In general, the FT800

Program 9

```
int v =readBatteryVoltage();
byte r1; byte g1;byte b1;
if (v <30) {                          // <30% make the bar RED
  r1=255;g1=0;b1=0;
}
if ((v >30) && ( v <50)) {            // 30 to 50% make the bar yellow
  r1=255;g1=255;b1=0;
}
if (v>50) {                           // 60 to 100% make bar green
  r1=0;b1=0;g1=255;
}
  // draw battery guage
Ft_Gpu_CoCmd_BgColor(phost,0xffffff);                    // white
Ft_App_WrCoCmd_Buffer(phost,COLOR_RGB(r1,g1,b1));        // bar color tracks voltage from
                                                         // red to yellow to green
Ft_Gpu_CoCmd_Progress(phost,350,200,80,20,0,v,100);
```

Program 10

```
analogReference(INTERNAL);
// use M328's 1.1Volt internal reference.
```

Program 11

```
int readBatteryVoltage()
 {
   int BatteryVoltage=0;
   for (int k=0;k<16;k++) {
     BatteryVoltage=BatteryVoltage+analogRead(A0);
     }
   int v=map(BatteryVoltage,8000, 11000,0,100);          // map 3.3 through 4.2 volts
                                                         // as 0-100% on the meter
   constrain (v,0,100);
   return v;
   }
```

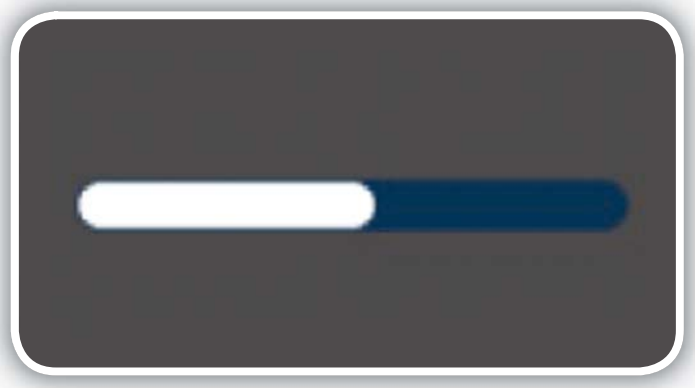


Photo 3

will plot a dynamically changing graph as fast as your eyes can register it. Graphs can get quite complicated in terms of axis marking, legends, the number of traces displayed. What I will show you is a way to produce a simple X-Y graph with one trace, simple axis marking, and without fancy legends.

Photo 5 shows a picture of the graph routine in action, with a simple sawtooth waveform being plotted. For simplicity, I just divided each axis into 10 intervals and placed ticks at the proper places on the axis. I left a bit of room on the right of the screen to allow for some numeric windows or for buttons to allow you to manipulate the display or to exit to another part of the program.

This routine does not use any fancy Co-processor commands or widgets. However, we will be sending lower-level graphics commands to the Co-processor as it is somewhat easier to send the commands this way. As mentioned earlier, the Co-processor is happy to handle these lower level commands as well as the more complex widgets commands.

To draw the enclosing rectangle we use the following code, ***program 12.***

LISTING 5

The first line tells the Co-processor that we want the lines to have a POINT_SIZE or thickness of 32. It is important to note that when the FT800 is drawing primatives such as lines and points, it uses a scale of 16 units per actual pixel on the screen. So a POINT_SIZE of 32 is actually 2 pixels wide. Although you could specify a POINT_SIZE of 24, you wouldn't actually see a line which is 1.5 pixels wide, as that would be physically impossible. So, you might wonder- what is the point of using a scale of 16? It turns out that once you start drawing diagonal lines or curves, having the graphics engine do to all of its calculations in units 16x larger than a pixel results in smoother lines (or anti-aliased as they say in the graphics business). All you have to remember is to multiply all of our drawing co-ordinates by a factor of 16.

The second line tells the Co-processor that the instructions/ co-ordinates following are meant to be drawn as lines. For every line we wish to draw, there must be two commands sent- one each for the X,Y pairs defining the two endpoints of the line. Unlike Visual Basic for example, where you can draw rectangles or polygons using a command that draws a line from its last-defined location to some new one, the FT800 requires that you specify both ends of a line, even when it is just a continuation of a previously-drawn line. So, for the graph's outer rectangle, we need 3 more pairs of statements similar to the last two statements, but defining the end-points of the other three lines needed for this rectangle.

The axes are easy to draw- for example here is the code for the horizontal axis, ***program 13.***

Program 12

```
      Ft_App_WrCoCmd_Buffer(phost,POINT_SIZE(32));
      Ft_App_WrCoCmd_Buffer(phost,BEGIN(LINES));
      Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(0,0));
      Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(400*16,0));
```

Program 13

```
 // draw the horizontal axis with 10 ticks
  for (j=640;j<6400;j+=640)
  {
    Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(j,245*16));
    Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(j,250*16));
  }
```

Program 14

```
for (j=0;j<k;  j++)
   {
  Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(Data_X[j], Data_Y[j]));
  Ft_App_WrCoCmd_Buffer(phost,VERTEX2F(Data_X[j+1], Data_Y[j+1]));
   }
```

LISTING 6

Note the large numbers- we have to multiply all of our actual pixel values by 16, as mentioned before. To plot the actual data points, we use the following code, ***program 14.***

LISTING 7

In this program I have defined X and Y arrays with 250 elements (and filled them with the sawtooth waveform), so we have to set the value of k=249 before executing the code above.

As mentioned before, these code fragments have to be surrounded by the "standard" lines of code which prepare the FT800's Co-processor ring buffer to accept data, and then swap and display the widget, like the other examples described earlier. The code for an Arduino sketch which demonstrates this graphing routine can be found on the SE website [2]

Generating Sound using the FT800

The FT800 has an impressive sound-generating capability which I briefly mentioned in Part 1 of the article series. Until recently, I was using only Mikroelektronika Connect-EVE display modules, and these particular modules contain neither an audio amplifier nor a small loudspeaker. Sound was not important in my first FT800 projects, so I didn't bother to investigate the sound capability.

Once I received an FTDI VM800BC43A module containing those missing audio components, I decided to give the audio commands a try. Actually, the touchscreen calibration routine, built into the FT800 firmware, uses the audio engine to produce little beeps, when you touch the screen in response to the calibration routine prompts. Hearing them reminded me to try out the audio functions.

In Part 2 of the the series, I described how to hook up both the Connect-EVE and the FTDI VM800BC43A modules, including any necessary jumper settings. If you want to use the audio capability of the VM800BC43A, there are a few more jumpers to configure. To see what needs to be done, download the VM800BC43A data sheet [3] and check out Figure 3-5 for the locations of JP1 and JP2. JP1 selects whether you want to power the amplifier from 3.3V or 5V. I was using a 5V supply to power the module, so I selected 5V, but you will have to match whatever power source you are using. However, more importantly, you will have to place a solder bridge or small piece of wire across the JP2 land pattern (located just above JP1). Even before I needed the audio capabilities of this module and had configured these jumpers, I could still hear a weak "beep" sound when the FT800 touchscreen was touched during the calibration routines. You will hear weak sound even if the audio amplifier is NOT powered up via JP2, but to get anything louder than this, you need to insure that the audio amplifier is receiving power.

Photo 4

I will admit that my sound requirements are very modest- generally just some "beeps" for user feedback, for example. Both the FT800 display modules, and the 4D Systems µLCD modules that I've used in the past, were able to play various formats of wavefiles. However in both cases you needed to supply these wavefiles from some form of external memory such as an SD card. Although the 4D Systems modules includes a µSD card socket, the VM800BC43A does not. I didn't want to go to the trouble of adding an SD card socket and also add the Arduino SD card library, just to get simple sound effects for user feedback.

In this respect, the FT800 controller is very versatile, as it contains a mini sound synthesizer engine, which is great for producing user-feedback "beeps" and also capable of generating "low-quality" tones emulating a number of musical instruments. It can also produce standard DTMF tones used in telephone dialing etc.

The information regarding the FT800's audio capabilities are spread out amongst several different FTDI documents. Therefore it is not straightforward figuring out how to get things going at the start. The sound synthesizer capability is covered briefly on page 19 of the FT800 Programmer's Guide[4]. However this short example is

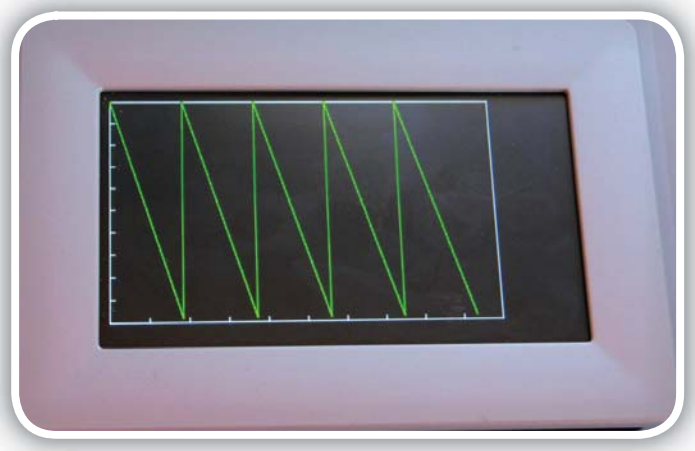


Photo 5

Photo 6

written in pseudocode, not the actual code needed to call the FT800 routines in the FTDi-supplied C routine library. Also, the information relating the different "voices" to numbers in the FT800 registers is not given anywhere in the Programmer's Guide. FTDI provides a sample application titled "FT_App_Music" [5] as shown in Photo 6. This application provides you with a quick way to listen to the musical voices and percussive sounds (the keys numbered 1-9 above piano keyboard) that are available.

Due to the "fancy" graphics in the program, as well as its versatility, this application is somewhat complicated to follow, so I will illustrate some basic code needed to provide such things as "beeps" for user-feedback in your own programs. The first important piece of information that you need is Table 2-1 in application note AN252 [6] . This table gives you the voice number of the various musical sounds, percussive effects, DTMF tones, etc. that are available. Table 2-2 gives you the midi note numbers corresponding to the various musical notes, as well as their frequency.

With this information, along with the short example given on page 19 of the FT800 Programmers Guide, I figured out the following short code snippet needed to produce a short "beep", useful as a GUI alert, ***program 15.***

LISTING 8

Line 1 is self-explanatory. In Line 2 we load REG_SOUND with a 16-bit number containing the midi note # in the upper byte and the voice in the lower byte. I have chosen to use the sawtooth wave (voice #3) with a midi note of 64 (0x40) which is a frequency of 329.6 Hz. If you look at the Sawtooth voice in Table 2-1 you will see that, unlike the musical voices, this one is a continuous sound. So, instead of waiting for the sound to end, as suggested in the FTDI example on page 19 of the Programmer's Guide, I just use a delay(300) statement to allow the tone to sound for 300 mS.

Because I didn't pay enough attention to the example, when I first wrote my code, I assumed that if you send a "1" to the REG_PLAY register to start the sound, then you would send a "0" to stop the sound. No Luck! If you look at Lines 4 and 5 you will see that you have to start playing a voice called "Silence" when you want to silence any sound that is currently playing.

If you choose to use a "musical" voice, it will not sound continuously but will play for a set amount of time. If you want to wait until that sound has finished, you can use the following code, ***program 16.***

LISTING 9

If you study the FT800 Programmer Guide carefully, you will notice that the FT800 Audio registers are 32-bits wide (as are most if not all of its registers). However, in the code snippets above there are commands that read/write either 8-bits or 16-bits. It turns out that if the upper word/bytes of a register are not used, then it is OK to just write/read the 8 or 16 bits that are needed, using the Wr8, Wr16, Rd8 commands.

You might also note that while the code examples that FTDI includes in the Programmers Guide shows wr8,wr16,wr32,rd8 etc., this is incorrect: the FTDI-supplied library routines use Wr8, Wr16, Wr32,Rd8 instead, so your program won't compile if you don't get the case of the first letter correct.

Program 15

```
Ft_Gpu_Hal_Wr8(phost,REG_VOL_SOUND,0x40); //set the volume to 1/4
Ft_Gpu_Hal_Wr16(phost,REG_SOUND, (0x40<< 8) | 0x03); //  MIDI note #64 sawtooth wave
Ft_Gpu_Hal_Wr8(phost,REG_PLAY, 1); // play the sound
delay(300);
 Ft_Gpu_Hal_Wr16(phost,REG_SOUND, (0x40<< 8) | 0x0); // 0 is "Silence" voice
 Ft_Gpu_Hal_Wr8(phost,REG_PLAY, 1); // play the sound
```

Program 16

```
int sound_status;
do {
  sound_status = Ft_Gpu_Hal_Rd8(phost,REG_PLAY);
  } while (sound_status ==1);
```

Conclusions

This is the final part of my article series on the FT800 display controller. The more I use these displays, the more I am impressed with their graphics capability and the speed at which the screens are drawn. Basically you can't perceive any "re-drawing" occurring, such as I've seen when using other LCD display panels that were being driven by an AVR MCU directly (although I admit that this is not a fair comparison).

While I did find the "learning-curve" to be a bit steep, that can be explained by the fact that I am a long-term Bascom/AVR programmer, and am not as well-versed in "C". The Arduino drivers and demo programs that FTDI supply use standard "C" conventions- they are not written as "standard" Arduino sketches. Unlike most peripheral chip drivers written for the Arduino, FTDI did not provide their driver as an "FT800" class, which are easily integrated into Arduino sketches by Arduino programmers that are not too familiar with real "C" programming.

What I tried to do in this series, was to simplify the process as much as possible for Arduino programmers. For example, I removed a lot of "extra" code in FTDI's Arduino samples,that were included to allow the sample programs to also run on a PC computer, via a USB adapter module. Removal of a lot of this conditional compiler code makes it a little less intimidating for a newbie to these displays. I also provided a few functional demo programs that contain only the code needed to do a specific function, like the numeric keypad, for example. If you look through most, if not all, of the FTDI sample applications, you will find a whole lot of "leftover" code- i.e. code that was part of one or more of their other sample programs, but totally unnecessary for that particular demo.

I am discussing with SE's Jure Mikeln the possibility of writing further articles on projects that I have built lately using these FT800 display modules. So, if you have been intrigued by these devices, stay tuned to SE!

References

- Numeric Keypad demo program (Arduino sketch) This zipped file contains my demo code along with all of the FTDI-supplied driver files.
- Draw Graph demo program.(Arduino sketch) This zipped file contains my demo code along with all of the FTDI-supplied driver files.
- DataSheet for the FTDI VM800BC43A:
  - ◊ http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_VM800B.pdf
- FT800 Programmers Guide:
  - ◊ http://www.ftdichip.com/Support/Documents/ProgramGuides/FT800%20Programmers%20Guide.pdf
- FTDI Musical instrument demo program:
  - ◊ http://www.ftdichip.com/Support/SoftwareExamples/EVE/FT_App_Music.zip
- FT800 Application note AN_252 AudioPrimer:
  - ◊ http://www.ftdichip.com/Support/Documents/AppNotes/AN_252%20FT800%20Audio%20Primer.pdf