The FTDI Chip EVE graphics controller: Sophisticated graphics using only a modest 8-bit MCU

Chapter 1

Most electronics enthusiasts strive to make their projects as user-friendly and commercial in appearance as possible. I'm no exception, and lately I have been trying to use TFT colour displays with touch-screen capability whenever practical. Although they are still somewhat expensive, you should also consider the savings that you can obtain by eliminating many of the switches, potentiometers, etc. that the touch-screen can replace.

In late 2013, FTDI (the company that makes the USB-serial interface chips we all use) started advertising their new EVE display controller chips. Soon afterward, MikroElektronika started selling 4.3" TFT display boards based upon this new controller. I got quite intrigued at this point, and started to look into the EVE controllers more closely.

Basically, the EVE controller chip is a very intelligent TFT display controller which can handle TFT panels up to 512 x 512 pixels, in up to 18-bit colour depth, or resolution. They will interface to any MCU with an SPI port, which covers most all MCUs apart from a few low pin-count ones. The EVE SPI interface is high speed (up to 30 Mb/s). This, coupled with the fact that the EVE controller is an intelligent one, executing high-level graphics commands, means that you can achieve very impressive graphics displays, even if you are hosting it on a modest 8-bit MCU, as FTDI's advertisements claim. In my personal experience, it is possible to implement a very nice GUI using the EVE controller driven with the Atmel AVR Atmega328 (as found on the Arduino Uno board, for example.)

The EVE controller provides a comprehensive variety of low-level graphics commands such as those needed to clear all/part of the screen, draw lines, rectangles, circles and other basic block figures. In addition, it also contains a co-processor engine, which adds a whole series of widgets such as buttons, sliders, rotary controls, clocks, switches, progress bars, etc. These are generated quite easily by sending out the proper widget command, along with the parameters needed to customize it to your needs: i.e. size, orientation, full scale value, etc.

The EVE controller also handles the resistive touch-screen functionality. In addition to the normal touch screen routines, where the controller returns the X-Y value of the spot being pressed, the widgets mentioned earlier can be "tagged" with an ID number, and when the user touches those widgets, this distinctive "tag" ID is returned to your program. This makes a touch-enabled GUI quite easy to implement, even when using only a modest 8-bit MCU.

Finally, the EVE controller provides an audio output. I'll discuss this further, but at this point, let's just say that the EVE can "play" sound files of various compressed formats. Also, it implements a sound synthesizer function, which allows it to play musical notes, melodies, or provide sound effects.

Now that you have a basic idea of what EVE will do, let's step back a bit, and take a comparative look at the various other methods that are available to provide a colour TFT touch-screen capability to your MCU project. In a previous article, I outlined two basic approaches to adding a TFT display to your project. The first one involves the use of a “dumb” TFT display which employs an 8-bit (or 16-bit) parallel interface to your MCU of choice. These displays are very inexpensive on eBay (10-20 EUR) but they do have some disadvantages:

1) They require up to 26 digital I/O lines on your MCU, forcing you to use a higher pin-count MCU than you might otherwise employ. ***All TFT displays operate on 3.3V, so all of these I/O lines must be at 3.3V levels.***
2) You must find the proper driver firmware for your chosen MCU, and this driver often uses up a lot of Flash memory space that you could otherwise use for your own program. You should also be aware that while these displays come in a limited number of physical sizes, there are ***many*** different LCD driver chips used on the different panels, which makes finding the proper driver somewhat more difficult.

I should mention that some of these “dumb” TFT displays are now being designed with direct Arduino compatibility. That is, they are mounted on a PCB which will plug directly into an Arduino Mega 2560 (or they include a “transition” PCB that goes between the TFT display itself and the Arduino Mega 2560). The Mega2560 MCU has plenty of I/O capacity, which addresses concern #1 above, and some of these display modules come with Arduino drivers, covering concern #2. Such Arduino-targeted boards contain level-shifting chips to handle the 5V levels present on the Arduino Mega 2560.

I tried one of these “dumb” TFT display/Arduino combinations, and while it may not be typical, I found that while the display module that I received worked, it was so dim that it was unusable. I suspect that these Chinese vendors are selling “seconds”: TFT displays that don't pass the normal QC standards of the TFT panel manufacturer. I have seen feedback from customers on various websites regarding this shortcoming. You may want to think twice about going down this route.

The second approach involves a serially-interfaced TFT display which contains its own intelligent display controller MCU. Such displays contain a whole library of high-level graphics routines and touch-screen handling, all of which you can access by sending the appropriate commands to the display, over a high-speed serial data link. Such commands are pretty compact in relation to the complexity of the graphics objects that they generate, so a standard serial data link, at a high baud rate (i.e. 115,200) is adequate to produce quite useable graphics display.

I've had excellent results on several projects using the µLCD series of displays from 4D Systems in Australia. They come in many sizes from small mobile-phone sizes up to large 4.3” displays. The SE article that I referred to earlier covered these displays in detail, as well as including lots of hints and examples of Bascom/AVR code to use with them. I recently finished a personal project using 4D Systems 4.3” µLCD display: an IR remote control which controlled my flat-screen TV and the three peripheral units associated with it. In place of the myriad of small buttons present on four separate remote controls, this unit features a clear, easy to use graphics display containing only the commonly-used buttons on each of the individual IR remotes. The user can quickly switch amongst the four “screens” (one per remote unit) using a small push-button. An added advantage to this approach is that this unit is easy to see in the dark, which is not true of a standard commercial IR remote. Image 1 shows one of the two such units that I built recently.

4D System's new “Workshop” IDE program contains a very high-level method of designing the various graphics screens needed for such a project. If you are familiar with Visual Basic, you would find 4D Systems “Workshop” IDE very easy to use in designing a nice GUI for your application.

The disadvantage of the 4D Systems µLCD display is that they are relatively expensive. The 4.3” model, with a resistive touch-screen, cost about 100 EUR, when I bought them last year. They also need a µSD card to be inserted into an on-board socket- to hold the files containing the graphics images for the various “widgets” that are a part of the user's GUI design. This adds about another 5 EUR to the price of the display.

I happened to have 2 extra 4.3” µLCD displays left over after finishing a commercial project I designed/built recently, so it cost me basically nothing to use my “spares” for the IR remote controller project. But, since I felt that the average user would not likely be interested in such an expensive IR remote, I decided against writing an article about this project, at least in its present form (but I am working on an EVE version/article).

So, with this quick comparison of the various TFT display options out of the way, let's look in more detail at the EVE controller and the display panels that are currently available.

What Is So Great About EVE?

Since I am a fan of the 4D Systems intelligent µLCD modules, you might wonder how I became interested in the EVE display controller chips. Well, to start with, there was the issue of price. Right from the start, it was clear that EVE-based TFT display modules were going to be a lot less expensive than the 4D systems µLCD display modules. For example, the 4.3” size (which I find ideal for many of my projects) was available in MikroElektronika's Connect-EVE module, costing about 50 EUR. This is about ½ the cost of a comparable 4D Systems µLCD module.

Another consideration concerned the interface method. While I generally like using the serial port method used by µLCD modules, there can be some disadvantages to it. For any graphics applications requiring fast motion, or complex graphics operations, the speed of the serial port can be a limiting factor. Also, many AVR MCUs contain only a single serial port, so a problem exists if you have an additional peripheral device that also needs a serial port. Indeed, if your project needs a USB port capability, you will often use an FTDI USB-serial interface chip for this purpose, so two serial ports would be needed if you also use a µLCD display.

The EVE graphics controller chip instead uses an SPI interface, along with a couple of other control lines (*PD and interrupt). An SPI port is much faster than a serial port. In the case of the EVE chip, it is capable of running at up to 30 Mb/s. You won't be able to achieve this high a rate with common AVR chips, as their highest SPI rate is SYSCLK/2 (8 Mb/s when using the 16 MHz crystal common on Arduino boards). Still, this is 69 times faster than the 115,200 baud rate that you could use between an AVR MCU and a µLCD display.

The other advantage to the SPI protocol is that you can have many different devices sharing a single SPI port, so long as they all don't need to communicate simultaneously. So, even with a modest Atmega328 MCU, you can drive an EVE display along with several other SPI peripherals, as well as any other peripheral that needs a serial port.

A second advantage to the EVE display controller chip is that it uses an advanced method of generating all of the “Widgets” or graphics objects which you might need. These are all generated and stored within the EVE controller chip itself. In contrast, the µLCD displays form their widgets using bit-map images, which must be stored in the µSD card mounted on the µLCD board. This µSD card also holds any other images that you need to display, as well as any sound files. While the cost of a µSD card is low, it's important to note that one must download these bitmap files to the µSD card using the PC computer that is running the 4D Systems “workshop” IDE program. This usually requires the use of a USB card-reader module on most PCs (apart from laptops). To put this in another way:

1) All of the code needed to generate a GUI on an EVE display is contained in the firmware you write for your chosen MCU, which can be easily distributed.
2) The µLCD displays will require that you write firmware for your MCU, which you can easily distribute, but you (or the end-user) must also have access to the 4D Systems “Workshop” IDE (which runs only on a PC) to generate the necessary GUI bitmaps. These must then be downloaded to a µSD card, which is then inserted into the µLCD display's on-board card socket.

As you can see, the EVE method is more straight-forward, particularly if others need to duplicate your project.

As I mentioned earlier, both EVE-based and µLCD modules handle all interaction with the touch screen using internal, high-level routines. That is to say, your program is relieved of the task of constantly scanning the resistive touch screen display for presses, and then doing a lot of calculations with X,Y co-ordinates, to determine which of the buttons, widgets, etc. was actually touched (or adjusted). On both of these displays, your program merely polls the display controller periodically, and it returns a code which identifies which widget on the screen was touched. This is really nice!

The last major feature of both of these display modules involves sound generation capabilities. The µLCD display's sound capability involves playing various types of compressed sound files, at a fairly low bit-rate and resolution. This file(s) must be downloaded to a µSD card, which is then mounted into the display's on-board socket. When I used µLCD displays for my most recent project, this was the only way to generate sound. In other words, if you needed something as simple as a “beep” or a click to indicate a screen touch, you had to download a compressed audio file to the µSD card. I think they should have allowed for a simple routine which generates a square-wave tone, the frequency and duration of which you could pass to the display as command parameters. Something along the lines of Bascom/AVR's SOUND statement would have been fine. The larger µLCD displays contain an audio amplifier and a very small, 12 mm speaker.

In the case of the EVE display controller, the sound functions are somewhat more versatile. It can play sound files in various formats (8-bit PCM, µLAW, 4-bit ADPCM) like the µLCD displays. However it also contains the equivalent of a MIDI synthesizer (something like the one I described in my previous article) which can either play simple musical melodies, or be used to provide simple beeps and clicks as needed for user interaction with the GUI. Playing a note consists of just a few short commands specifying the MIDI instrument, musical note and note duration. The MikroElektronika Connect-EVE modules that I am using have an audio output pin, but no amplifier or speaker on-board. I haven't needed or tried out the sound capability of the EVE controller yet.

What's your preference: 5V or 3.3V Displays?

If you are using an AVR MCU, you are probably running it at a Vcc 5 volts. You get full speed operation that way, and many of the common peripheral IC devices operate on 5V. The most commonly-used Arduino boards also operate on 5V, although this is gradually changing with the advent of the Due, as well as numerous “clones” from other manufacturers that operate on either 3.3V, or both 3.3V and 5V.

Regardless of what TFT display module you choose, they all operate internally at 3.3V. However, the power supply voltage that you must provide to the module will vary from one manufacturer to the next. Also the necessary logic levels that are required will also vary. It's critical to note that supplying a module that requires a 3.3V power source with 5V will likely destroy it, as will the application of 5V logic level signals to a module that calls for 3.3V (maximum) logic levels.

Table 1 shows some important features of the three 4.3” EVE-based display modules that were available when I wrote this article. Here you can see that only FTDI's own modules are capable of operating with either a 3.3 or 5 volt power supply. Related to this, you can see that they are also the only modules that will interface to either 3.3 or 5V logic-level signals. If you are using an MCU running on 5V, like an Arduino Uno for example, it is probably best to choose the FTDI module rather than worrying about adding your own level-shifting circuitry. Incidentally, FTDI also sell similar EVE modules that contain a smaller board (for the EVE) that connects up to the TFT display with a flex cable. These models don't come with a mounting bezel for the display however, so you might find them harder to use.

Feature	FTDI VM800B	Mikroelektronika Connect-EVE	4D Systems 4DLCD-FT843
Power supply	3.3V or 5V	3.3V	3.3V
Logic levels	3.3V or 5V	3.3V (not 5V Tolerant)	3.3V (not 5V tolerant)
Ease of Panel mount	Easy (bezel included)	Mounting holes but no bezel	Difficult without Bezel and Breakout board kit (available separately)
Interface connector	10 pin 0.1” header	10 pin 0.1” header & 2X5 0.1” header	10 pin 0.5mm FPC flex. ribbon
Audio capability	Amplifier & speaker	Audio output pin	Audio output pin
Price	64 EUR	50 EUR	43 EUR (includes Bezel and breakout board)

Table 1: Comparison of unique features amongst three currently-available 4.3” EVE-based display modules. All modules feature a QVGA resolution of 480 X 272 pixels, and 16-bit colour depth.

I started working with EVE-based displays after purchasing a few MikroElektronika Connect-EVE modules, which I chose because they were the first EVE display modules available. These displays operate on 3.3V only, and the first few projects that I had in mind for them called for a 5V AVR MCU, both for speed considerations, and because most of the other peripheral devices needed were 5V devices.

Although I powered the EVE display itself with a separate 3.3V regulated supply, I also found that the required logic-level conversion was easily accomplished using only a simple resistive divider network (1K and 470 ohm resistors) on the MCU's MOSI,SCK and -PD output pins. The Connect-EVE's 3.3V logic-level MISO output signal was sufficient to drive the ATMega328's MISO pin directly.

The Connect-EVE's -INT pin would also have interfaced to the ATMega328 directly, but I did not need that pin for my design. That being said, I should add that I was using an SPI clock rate of only 4 MHz, due to limitations of some of the other SPI devices used in the project. But I am doubtful that the EVE display module would operate at much higher SPI rates (it's rated up to 30 Mb/s maximum), using this simple resistive level-shifting method (shown in Figure 1).

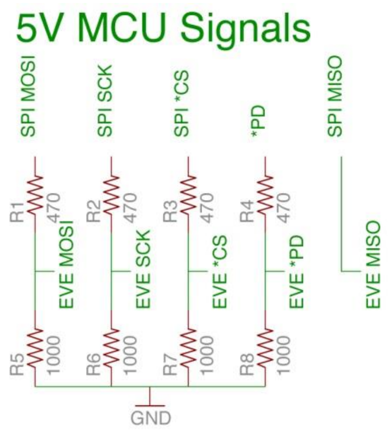

Figure 1: If you are using the MikroElektronika Connect EVE module (3.3V Logic and power supply) with a 5V MCU, you can match signal levels using this resistive voltage divider.

I didn't make use of the audio capabilities of the EVE controller, but it should be noted that the FTDI VM800B modules contain both an amplifier/filter and a small speaker, which the other two modules do not.

When studying such exciting new TFT display modules, it is quite easy to overlook the matter of mounting it in a cabinet or on a panel. Like most readers, I am an electronics enthusiast without access to customized plastic enclosures with all the cut-outs and mounting tabs/brackets already present. Unlike most readers, I do have a home-built CNC milling machine available, so I can easily cut out rectangular holes in aluminum or plastic enclosures. So, personally, it was fairly easy to mount the MikroElektronika Connect-EVE modules by merely cutting the proper-sized rectangular hole in an aluminum enclosure, and fastening it in place using 4 spacers and screws/nuts. If you don't have an easy way of cutting out a “clean” rectangular hole, the FTDI VM800B modules may be your best choice, as they come with a bezel which can hide any rough edges of the hole that you cut out. Alternately, if you like the low price of 4D System's 4DLCD-FT843 module, you can order the optional Bezel/Breakout board kit (which I included in the price shown in Table 1). This kit takes care of any mounting complications, as well as providing a small breakout PCB which mates up with the 10-way 0.5mm FPC ribbon cable, and provides a 0.1” header connection. I should add that using the MikroElektronika Connect-EVE modules has the advantage that it takes up a bit less space on your front panel than using either of the other two models with their respective bezels.

EVE's Basic Architecture

Before discussing the graphics features of the EVE controller, I want to mention a couple of basic architectural details in which EVE differs from most other graphics controllers. Generally, graphics display controllers contain a fairly large RAM memory device, which is used to store whatever image is being displayed at the time. For the 4.3” TFT displays that we're talking about, the pixel resolution is 480 x 272, and the colour bit depth or resolution is 18-bits. This would theoretically require a RAM chip capacity of *480 X 272 X 3 bytes/pixel or 391,680 bytes.*

If you want to allow for smooth screen updates (like when quick motion must be displayed) you generally have to double the size of this RAM. This allows for two screen buffers: one acting as the active display buffer, with the other one being the one that the host MCU “fills up” with the content of the next image “frame”. Then, you can just swap the pointers to the buffers, to provide an instantly updated screen, i.e. preventing the display of a screen which visually “morphs” as its screen buffer is updated by the host MCU. So, you can see that close to 800K bytes of RAM is needed in this scenario.

The EVE controller doesn't work in this manner at all. Instead, it keeps track of all of the visual items needed on the active screen (lines, text, widgets, etc.) in a “display list”. Then, custom-designed, high- speed logic examines all of the aspects of the various objects in the display list, and, on a ***line-by-line basis***, determines what pixel data has to be sent to the screen. So, while EVE still has to store the display list in RAM, you don't need to design in a RAM array that is capable of storing 2X a whole screen's worth of information at a time. Like the traditional graphic controller that contains double the amount of RAM needed for a given size screen and swaps it between consecutive frames, so too does the EVE controller maintain two separate display object lists, which it swaps between, for instant display updating.

While I don't pretend to understand the fine points of EVE's design, it must be a lot more efficient for the EVE controller to manipulate compact display lists than it is to try and manipulate bits in a large display RAM array. This would account for the fact that EVE-based displays are much less expensive than competing modules using conventional controller technology.

I should mention that if you are using an intelligent display module such as the 4D Systems µLCD displays or EVE, which of the above two methods is actually used in your display module, is not overly important to you, the programmer of the Host MCU. In either case you are basically sending high-level graphics commands from the host MCU to the display controller serially, and how the display controller actually renders the video display is somewhat transparent to the host MCU programmer.

In my opinion, it's quite a bit easier for the MCU programmer to learn the high-level graphics commands needed to successfully use the 4D Systems µLCD displays, than it is to use EVE-based modules. With only a few simple commands (and parameters), you can clear the screen and put up some text or a box very easily on a µLCD display. There are a ***lot*** more commands and parameters needed to use EVE-based display modules even for modest applications. However, once you advance to more complex GUI applications, I believe that the code complexity of either of these display options is comparable.

I think it's fair to say that if the average MCU programmer was not provided with a comprehensive EVE graphics library (written for his/her chosen MCU family), along with some reasonable demo programs, he/she would likely give up in despair if they had to “start from scratch”. Once you get going however, the low price of the EVE displays, along with its very powerful architecture, makes it well worth the effort.

My Introduction to EVE-based Displays

At the start, I carefully read the “preliminary” datasheet [1] that FTDI published at the same time as their early advertisements. This datasheet made everything look very simple. However, all of their examples appeared to be written in a “pseudo-code” of sorts: i.e. all commands seemed to be simple “English” phrases. They also referred to various function calls which seemed to be taken from a C driver library, which was not yet available to the public. So, I delayed purchasing any actual display modules for a few months until the more comprehensive Programmer's Guide became available [2] . At the same time that this guide (with a “draft” watermark) was published, FTDI also released a software package for the Arduino, with drivers and example programs. While I am much more comfortable using Bascom/AVR for Atmel AVR MCUs, I did have some experience writing/understanding Arduino “sketches” (which the Arduino IDE basically surrounds with a “wrapper” to simplify things for newbie users, and then passes on to a C/C++ compiler) . With at least some software available, I decided it was time to order some actual hardware. At that time, I chose the only modules available: MikroElektronika's 4.3” Connect-EVE displays.

Once the display modules arrived, I had a few choices on how to use them with the AVR MCUs that I customarily use in my projects. One obvious choice was to further investigate the Arduino software package created by FTDI themselves, as referred to in their datasheet and application notes.

Another choice was to use MikroElektronika's own software which supports EVE-based displays. They sell compilers of various types (C, Basic and Pascal) targeted at several different MCU families (AVR, PIC and ARM). Associated with all of these compilers is their Visual TFT software package, which provides a high-level interface to many types of TFT displays, including their own Connect-EVE display module. MikroElektronika had been very generous in providing me (being an electronics author) with a compiler of my choosing, at no charge. I had chosen their AVR Basic compiler as well as the Visual TFT package. So, I decided to try this first.

The basic concept behind the Visual TFT package is that you select the MCU development board that you have, as well as the type of TFT display module that you wish to use. Then you start out with a “clean slate”, so to speak, and drag/drop the various graphics elements you need onto a “virtual screen” contained within Visual TFT's IDE. If you have used Microsoft Visual Basic or Visual C++, then this GUI-based drag-drop method of programming will be familiar to you. Once you have done this, Visual TFT will generate the source code needed to implement your screen (or multiple screens if needed). Next you will be transferred to whatever MikroElektronika compiler you are using. Here you would add your own code to handle all non-display aspects of your program. Also, the code needed to handle the various touch screen actions can be done either in Visual TFT, or later in the compiler itself. Then you “build” the program, and upload it to your MCU.

Although I did not own any MikroElektronika MCU development boards, I picked the Xmega development board from the list, since I had an Atmel XMEGA Xplained board on hand. Having had prior experience with the Xmega Xplained board, I knew which of the 4 SPI ports available on the XMEGA devices, was accessible on the Xplained board's header socket. It turns out that MikroElektronika's Xmega board used a different SPI port. After quite a bit of searching, I found the spot in the Visual TFT source code where the SPI port was defined and initialized, and made the necessary changes to some global definitions. After doing this, I was able to upload a very simple program to the Xmega Xplained board/Connect-EVE display, and everything worked fine. So far so good!

The next thing I did was to examine the source code generated by Visual TFT, to see if I could understand it enough to be able to integrate their code into whatever code I would be writing myself. Also, I had to determine how to make Visual TFT/MikroElektronika AVR Basic compiler generate code that would work on the actual AVR devices that I commonly use (Mega 88, 328, 644, 1284 etc.). This is where I hit “a brick wall”.

It seems like MikroElektronika first developed C compilers for the various MCU families and then went on to expand into Pascal and Basic. However, being so familiar with both Bascom/AVR and Visual Basic (PC), I found the syntax and conventions used by MikroElektronika's Basic compiler to be quite different and confusing. To me, the code generated by Visual TFT looked more like C++ than Basic, and I struggled to follow it.

The Visual TFT program has to be able to generate code for both “intelligent” TFT displays (i.e. EVE) and many different types of “dumb” displays (as mentioned at, the start of the article). As such, I believe it is generating non-optimal (and hard-to decipher) code. I think this is particularly true in the case of EVE-based displays. The fact that it is called upon to generate code in three different compiler languages, for a large number of MCUs in the PIC, AVR and ARM families contributes to making the code much more complex than it need be, in my opinion.

To back up this observation, I found that a very simple Visual TFT demo program containing only a few buttons on the screen (with NO code in the handling routines for those button presses), generated a 40 Kilobyte AVR program. This is far too big to fit into the flash memory of an Arduino Uno's Mega328. In contrast, I have since written a pretty complicated Arduino sketch, containing a fairly sophisticated GUI (keypads, buttons & X-Y graphs etc.) and lots of code to handle several other peripheral chips. This sketch takes up only about 26K of Flash memory on a Mega328.

I don't mean to paint an unflattering picture of MikroElektronika's software products, based solely on my own personal experience. If C++ is your main MCU development language, then you would probably be happy with Visual TFT and MikroElektronika's

C compilers. Since most of the EVE demo programs supplied by MikroElektronika were written in C for the PIC MCU family, I did not find it very helpful personally.

FTDI Drivers and Demos

Given the problems described in the previous section, I next decided to try FTDI's own Arduino driver/demo code for the Arduino [3]. I was able to compile FTDI's Arduino demo program easily enough using V1.5 of the Arduino IDE that I use (I'm sure V1.05 would work as well). It would have been nice had I been able to load the code directly into either the Arduino Uno or Mega2560 boards that I had on hand. However, since the Connect-EVE display module will only work on a 3.3V power supply, using 3.3V logic levels, neither of these 5V logic-level Arduino boards would work. So I wired the Connect-EVE up to a home-built circuit board containing a Atmega328. While this board ran the 'Mega328 at 5V, there was plenty of space on the board to add both a 3.3V regulator, and the necessary level-shifting circuitry. The simple resistive level-shifter that I used is shown in Figure 1.

After I loaded the FTDI Arduino demo program into the Mega328, I did not initially see anything appearing on the display. I was a bit surprised, as I had connected the Connect-EVE module up exactly as shown in FTDI AN 246 [3], which is the application note which accompanies this demo program. A lesson: don't stop reading this application note after you get to the wiring diagram, like I did! It turns out that later on in the Application Note; it instructs you to define your screen size in the program. The Connect-EVE's 4.3" screen was the default, so I was OK there. But more importantly, it tells you to un-comment ***one*** of the five lines which define which ***set*** of demo routines that you want to run. This demo program contains a **lot** of different demo functions, and an Arduino Uno (with a 'mega328) would not have nearly enough flash memory to hold this program if more than one of these sets of routines was included. By default, **all** of these 5 lines are commented-out, so, until you choose one line to un-comment, your program will compile OK, but nothing will appear on the screen! After my oversight was corrected, the program ran as designed, and the Connect-EVE display went through a set of demo routines. It was quite impressive. Success at last!

Of course, I had already gotten the Connect-EVE to work using a simple program compiled by the MikroElektronika compiler and Visual TFT, only to find I couldn't follow the code it generated. To be honest, when I first examined the code in the FTDI Arduino sample program, I was similarly unable to make much sense of it. To begin with, it wasn't written using the normal syntax of an Arduino sketch, but rather as a C++ program. Also, where an Arduino sketch is generally quite easy-to-follow, with all of the complexities of the hardware driver hidden in an Arduino "class" library, this demo program did not define an "EVE" class at all.

To make things even more confusing (to a newbie), the C++ demo program was written to work with either

1) an Arduino board

2) a PC computer interfaced to the EVE's SPI interface via an FTDI USB interface chip programmed in the MPSSE mode (Multi-protocol Synchronous Serial Engine)

As a result, the program was full of compiler directives meant to instruct the compiler to generate code for whichever of the above options was chosen. This basically made the listing at least twice as long as it would have been for just the Arduino alone. That, combined with the fact that the demo program contained such a large number of different demo functions, made it hard for me (with C++ being my “third language”) to follow.

What I have done is go through all of the code and remove the entire conditionally-compiled code specific to the PC environment (#2 above). Then I further removed all of the “fancy” screen demos that were not necessary for a simple application. The resulting code is easier to follow, and makes it somewhat easier for a newbie to get started. For reference, this simple program, which initializes the screen, puts up a few buttons, text, etc. takes up about 6K of AVR code. As I mentioned earlier, a pretty complex GUI program that I've subsequently written, takes only about 26K of Flash memory on a 'mega328. You can certainly do some complex programs with this EVE display, even with an MCU as modest as that used in an Arduino Uno. This basic program will be included with Part 2 of this multi-part EVE article series.

In the next part of the article, I'll be covering some of the basic EVE functions that you will want to use, such as Text, lines, boxes, etc. I'll also cover some of the most useful widgets, such as the buttons, and how to draw graphs, etc. I'll also look at the MikroElektronika Connect-EVE display module in more depth.

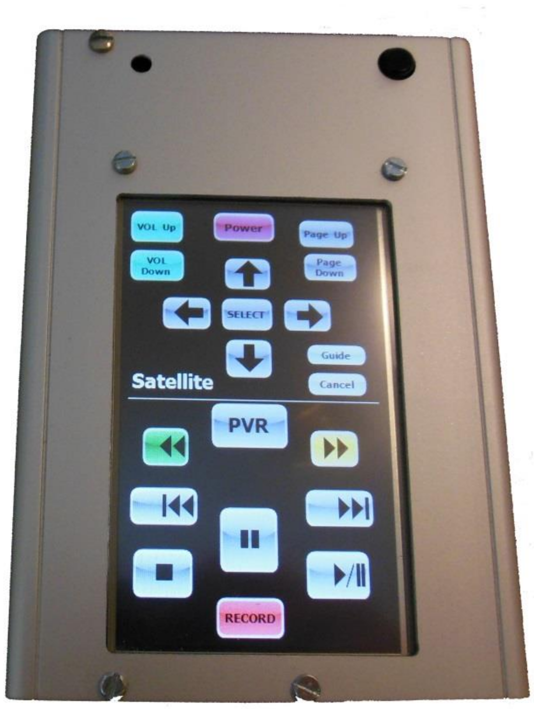

Image 1: This is a touch-screen multi-function IR remote control I built a while back, using 4D Systems uLCD-43PT intelligent TFT touchscreen module. I am now designing a similar device using the much less expensive EVE display modules.

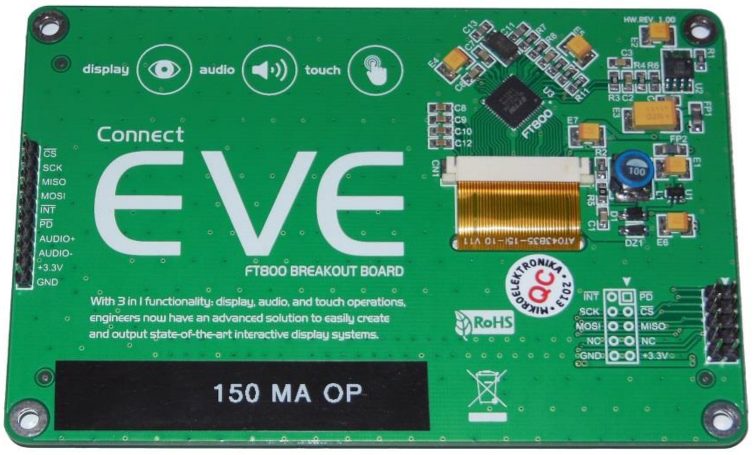
display
audio
touch
Connect
EVE
FT800 BREAKOUT BOARD
With 3 in 1 functionality: display, audio, and touch operations, engineers now have an advanced solution to easily create and output state-of-the-art interactive display systems.
150 MA OP
CS
SCK
MISO
MOSI
INT
PD
AUDIO+
AUDIO-
+3.3V
GND
RoHS
MIKROELEKTRONIKA QC 2013
FT800

Image 2: This is the control PCB side of the Connect-EVE 4.3" display module. Note that it contains both a 10 pin interface and a 2X5 header, with all necessary signals available on both. The "150 MA OP" label is one that I added to remind me that the module draws 150 milliamps when operating, which is important since I am using it in battery-powered projects.

Chapter 2 - Basic Hook-up and Programming

In Chapter 1, I covered the basic features and advantages of the FTDI FT800 EVE graphics controller chip. In doing so, I mentioned several other methods of obtaining color TFT touch-screen functionality, including both “dumb” TFT displays that interface to your MCU via an 8/16 bit parallel port, as well as “intelligent” TFT display modules such as those sold by 4D Systems. I feel that FTDI's EVE solution is both cost-effective and lends itself well to being interfaced with modest 8-bit MCUs, like the Atmel Mega328P that is found on many of the popular Arduino boards (i.e. the Uno). This month I'm going to show the reader how to get started using some of the EVE-powered TFT touch-screen modules that are currently available.

Before choosing which display module that you wish to purchase, you should look back at Table 1 in Chapter 1 to determine which of the three company's modules best serve your requirements. When I first read about the FTDI EVE controller, MikroElektronika made the only EVE-powered TFT touchscreen modules available: the 4.3” ConnectEVE. So, I started out by purchasing a few of these and designing projects around them. Shortly thereafter, FTDI themselves started selling two different series of evaluation modules: the VM800B series that comes with a mounting bezel and full-size control PCB, and the VM800C series which contain a credit card-sized controller PCB and can be purchased with or without a TFT display. The VM800C modules come without a bezel, and the TFT display has no mounting tabs either.

In this article I will use one of the FTDI 4.3” VM800B modules to demonstrate the operation of the EVE controller. Later in the series, I'll describe one of the projects I designed/built using the somewhat more compact MikroElektronika ConnectEVE modules (the first ones I purchased).

Module Hook-up

If you think like me, the physical hook-up of TFT display modules to your favourite MCU is probably the least of your worries. My prime consideration is how difficult it will be to write or obtain the necessary software drivers to interface such display modules with the Atmel AVR MCU family that I generally use. This consideration is further complicated by the fact that I generally write programs using Bascom/AVR and not C/C++. While I hate to admit it, it appears that the majority of drivers for peripheral devices/chips are supplied by the vendors in the form of C/C++ libraries. Part of the reason for this is that AVR-based Arduino boards are extremely popular and the Arduino is programmed in C/C++. Arduino programs or “sketches” are basically C/C++ code which has been “wrapped” by a user-friendly IDE which hides a lot of the complexity of C/C++.

In the case of the FT800 EVE controller, FTDI decided to write both their drivers and example programs in C/C++. The Arduino compiler will handle this just as well as it will handle “sketches” written in the Arduino simplified C format. If you are accustomed to writing Arduino “sketches” but are not a C/C++ expert, you will find the FTDI drivers and example programs rather hard to follow. As a long-time Bascom/AVR programmer, this C/C++ code was even more difficult for me to follow! Therefore, I decided it would be too daunting a task to convert all of this code to Basic, and instead decided to join the large world-wide Arduino “club”, and write code for my FTDI EVE-based projects as Arduino “sketches”.

To save you from having to “re-invent the wheel”, what I have done is carefully go through FTDI's example code, remove as much extra code as possible, and simplify what remained as much as possible. If a Bascom/AVR fan like me can successfully use these FTDI EVE modules in projects, you should be able to do so as well!

So, let's see how we can hook up our EVE-based display to an AVR Mega328. The interface is just a standard SPI interface, with a couple of extra lines to handle the board Reset/Power-down and an optional interrupt. Depending upon which type of module you have, the logic levels required will be either strictly 3.3V or 3.3/5V switchable. See Table 1 in Chapter 1 to determine which is applicable to your board. If you have a 3.3V only module, but are using an MCU powered by 5V (i.e. an Arduino Uno), then you will have to use some level-shifting circuitry. I described a simple resistive logic level converter in Figure 1 of Chapter 1, which is simple and works well for this purpose.

In either case, the wiring between the display module and your MCU is shown in Table 1 in FTDI's AN246 Application note entitled ‘VM800CB_Sample_App_Arduino_Introduction’ [4]

The MCU pin definitions shown here are the Arduino definitions- not the same as the actual port designations used by Atmel (i.e. PortB.5). If you are using an Arduino board, this table will be easy to follow. If you are using some other board with a Mega328 on it, and want to know the connections based upon Atmel's designations, then check out Table 2 below.

EVE Controller module	Mega328 MCU board
SCK	PortB.5
MOSI	PortB.3
MISO	PortB.4
*CS	PortB.2
*PD	PortD.4
*INT	PortD.3
GND	GND

Table 2: EVE controller hookup using Atmel pin designations

It is obvious that EVE's SPI lines should interconnect with the AVR's SPI lines. In the case of the *PD and optional *INT lines, these could connect to any other available I/O port lines, but the FTDI drivers and example programs are configured to use the specific I/O pins shown in Figure `. When I discuss programming a bit later on, I'll show you where you can find these definitions in the program code itself, in case you wish to use other I/O pins rather than the defaults.

If you have a MikroElektronika ConnectEVE display board, you will have to supply it with a regulated 3.3 volt power supply. I measured the current that this board draws, and found it to be about 150 mA under normal operating conditions. A Microchip MCP1700-3302 is an ideal LDO regulator to use for this, so long as you don't exceed its 6V maximum input voltage limit. For higher input voltages you could use a MCP1702-3302 instead, but you should make sure that you don't exceed its power dissipation limits if your input voltage is too high. Of course there are many other LDO regulators you could choose, but I like these as they are available in a TO-92 package, which is easy to handle/solder (in other words it is not a tiny SMT part that I can barely see/handle!)

If you have an FTDI VM800B module, you can supply it with either 5V or 3.3V. Usually you would choose the same voltage that you are using for your MCU. If you choose 5V, you can supply 5V to this board in 3 ways:

1) Via the micro USB socket provided on the board and jumper 2 & 3 of SW1
2) Via CN1, (which is a 2 pin JST connector) and jumper 1 & 2 of SW1
3) Via pin 7 of the J5 interface header (SW1 jumper position doesn't matter)

If you choose to supply the VM800B with 3.3V, you can do it in the following ways:

1) Via CN1 (the 2 pin JST connector) and short pins 7&8 of J5 and jumper pins 1&2 of SW1.
2) Via pins 7&8 of the J5 interface header. (SW1 jumper position doesn't matter in this case)

If you are using the MikroElektronika ConnectEVE display module, there is a warning I'd like to mention. This module contains both an inline 10 pin header as well as a 2X5 header. I recommend that you use the 10 pin header. The 2X5 header contains the same signals, which are silk-screened on the board. However, if you look closely at Image 1, at the location of the arrow which normally designates Pin 1, you will see that it is actually pointing to Pin 2. If you were to wire up your host MCU board expecting the *PD signal to be on Pin1, etc., all of your connections would be wrong!

When I contacted MikroElektronika about this, they claimed that their drawing was correct- if you placed the 2X5 header on the front side of the module. However, as I pointed out to them, this is impossible as there is not enough depth to allow a ribbon cable plug to be mounted on the front- assuming that the module is mounted on a panel of some sort, which would normally be the case. Luckily, I used the 10 Pin header when I first tried out the module, and on my next project, when I decided to use the 2X5 header, I checked out my interconnect wiring with an ohmmeter, and noticed the discrepancy before I powered things up.

At this point you are ready to load the Mega328P with an example program to test out the display. Image 2 shows my interconnect wiring between the FTDI VM800B display and an Arduino Uno.

Initial Test Program

At this point you have a few choices about which program to load to test out the display. Assuming that you are using an Atmel Mega328P (whether on an Arduino board or not), you can try the following:

1) The program example supplied by FTDI as part of their application note AN246 [3], mentioned earlier. A link to this can be found in the reference section of this article.
2) A simple program which I have written which is based upon FTDI's code, but highly simplified to make it easier for the beginner to follow. This can be found on SE's website and is an Arduino sketch titled “FT800_Basic_Setup” and includes the necessary FTDI C support library files.

If you decide to go with option 1, and use the FTDI example program, it is very important that you read section 2.2.1 and 2.2.2 of AN246 where it instructs you how to modify the source code to:

A) Match your display size
B) Pick a set of demo routines to run. Note that, by default, ***none*** of the demo routines are pre-selected, so ***nothing will show up on the screen if you fail to do this!***

If you choose option 2, my program will simply put up a “splash” screen without you having to make any customizations to the code. You just compile and download it from within the Arduino IDE. Of course you have to be somewhat familiar with the Arduino, and have set up the IDE for the Arduino Uno board and have chosen the serial port number that its USB interface has been assigned to.

Apart from putting up a simple “splash” screen, this program also opens the Mega328P's serial port (at 9600 baud) and sends out messages to indicate that the program has started and that the start screen has been displayed. It also places two button “widgets” on the screen, and stays in a polling loop waiting to see if the user has pressed either of the two buttons. When either button is pressed, a message will be sent out the serial port indicating which button was pressed.

If you don't see the splash screen shown in Image 5:

Image 5

at power-up, but you can see the serial port messages mentioned above, then you have either:

1) An interconnection wiring problem
2) A problem with the resistive level-shifter circuit which must be used if you are using a Mega328P at a Vcc of 5V and an EVE module that requires 3.3V logic level signals. (i.e. the MikroElektronika ConnectEVE or 4D Systems FT843)
3) Improper power supply to the EVE display module
4) A bad display module (unlikely)

It all has gone smoothly so far, you will probably want to start looking closely at the program code to see what you need to understand in order to use these displays. If you take a look at the FTDI program that comes with the AN246, one of two things will happen. If you are an expert C/C++ programmer and have read the 238 pages of the FT800 Programmer's Guide, you will probably be able to follow it reasonably well. Otherwise, you will probably not be able to make much sense out of it. Even if you are reasonably experienced writing Arduino sketches, this will probably still be daunting to you, as the driver and sample code is:

1) Written using C/C++ conventions rather than the simpler Arduino language syntax.
2) Not written as a “class”, as are most Arduino libraries. Therefore it is more difficult for Arduino programmers, who are used to simply “including” a library and accessing the device via various class methods.
3) Written to work with either an Arduino board or a PC (outfitted with FTDI's VA800A-SPI MPSSE USB adapter, model, available from the FTDI online store [5].

While there wasn't a lot I could easily do about 1) and 2), what I decided was to do the following:

1) Remove all of the conditional-compilation source code needed to operate the EVE display using an MPSSE-equipped PC. This made the code easier to follow.
2) Remove all of the complex routines that were contained in the FTDI sample program which, although they produced a fancy demo program, made it very hard for a beginner with basic requirements, to follow.

Basic EVE Initialization

The FT800 EVE chip is designed to work with TFT panels with pixel counts up to 512 vertically by 512 horizontally. The actual pixel count of the display must be specified as part of the initialization routine, as well as a fair number of other timing parameters etc. We don't have to worry much about this, apart from examining the FTDI-supplied driver routines and making sure that the settings match our display. Both the FTDI VM800B and MikroElektronika ConnectEVE display modules that I have on hand were WQVGA (Wide 1/4 VGA resolution). The resolution of these modules is 480H X 272V. The FTDI SampleApp1.0 program that is described in AN246 includes the proper definitions for this display, starting at line 14 of the “sampleApp.cpp” file. In my program, the same definitions are found at line 17 of the “FT800_Basic_Setup” file.

I could not find the pixel resolution of either the 3.5” or 5” VM800 modules on FTDI's website. However, if you wanted to use the EVE controller with a QVGA TFT module, the correct display definitions for it, as well as the WQVGA resolutions, can be found in FTDI application note AN_240 on page 16.

To start up the EVE controller chip, you must first drive the *PD pin low for 20ms and then wait 20ms after this line is returned high again. After this, there is a prescribed series of commands that must be issued to the EVE controller. You can see this command sequence in my program starting at line 180 (close to the start of the standard “setup” routine present in all Arduino sketches). I won't go into any detail here, except to say that the sequence is pretty well-explained in section 4.2.2 of Application note AN_240.

FTDI mentions the fact that the maximum SPI rate that the FT800 EVE will accept is less than/equal to 10 MHz during the early phases of the initialization. However, once the internal PLL is set to 48 MHz, you can increase the SPI rate of your MCU up to 30MHz. This is not really a concern with Mega328 Arduino boards, as the maximum SPI rate that they can achieve is SysCLK/2 or 8 MHz. However, if you are using the resistive level-shifter circuitry shown in Figure 1 of Part 1, you should limit the SPI rate to SysCLK/4, because this form of level shifter degrades the SPI signals somewhat, and an 8 MHz SPI rate is too high to work properly, in my experience. The M328's SPI rate is set in file “FT_GPU_Hal.cpp” at line 17 of my program (line 42 of FTDI's sampleApp1 program):

SPI.setClockDivider(SPI_CLOCK_DIV4);   // SPI rate =4 MHZ for 16 MHz clock

or
SPI.setClockDivider(SPI_CLOCK_DIV2);   // SPI rate = 8Mhz for 16Mhz clock

You may wonder what “Hal” means as part of some of the FTDI-supplied driver files. Hal is an abbreviation for Hardware Abstraction Layer. What this means is that FTDI has broken up its EVE driver structure into two layers. There is an upper, generic layer of the driver that implements all of the necessary commands needed to use the EVE controller. As well, there is the Hardware Abstraction Layer, which translates these generic operations into the MCU code that is needed to access the physical registers and I/O ports for the chosen target device. It is in this layer that you would find both the code needed for the Arduino Mega328P MCU, as well as the routines needed if you instead wanted to communicate with EVE via an MPSSE-equipped PC (as mentioned earlier). It is in these files that I went to the trouble of removing all of the MPSSE-equipped PC driver code, to make the program easier to follow for the majority of readers that would not be using this option.

After the above initialization routines have been performed, the EVE controller is ready to accept the user's specific commands.

Doing Something Useful

Now let's take a look at what we have to do in order to display a useful Graphical User Interface, or GUI. To begin with, I have to say that the EVE controller is a very complex device, with a very involved set of instructions and ***many*** registers. FTDI has supplied more than 300 pages of technical literature in the form of a Programmer's Guide, FT800 EVE data sheet and numerous applications notes (with AN_240 and AN246 containing the most useful information for newcomers).

That being said, a newbie who just wants to program the EVE to provide a display with some touch-sensitive buttons and controls, some simple readouts like gauges, dials, clock displays etc., and simple text (in many font sizes), doesn't need to fully understand all that is going on “underneath the hood” of the EVE chip. As long as you have a general idea of how the EVE chip works, as well as some sample code to “tweak” to your own requirements, you should manage OK.

The basic concepts that you need to understand are as follows. The video display section of the EVE controller consists of two discrete graphics “engines” that work independently. These are the ***main graphics engine*** and the ***graphics coprocessor***. The main graphics engine performs basic graphics operations such as drawing points, lines, rectangles, bitmaps and what FTDI call Edge Strips.

The graphics co-processor engine performs many higher-level functions such as drawing text, buttons, gauges, rows of keys, bitmap images, progress and scroll bars, etc. In addition to generating these high-level “widgets”, this co-processor also performs several very high-level functions such as:

1) An FTDI banner screen with a moving logo (more useful to FTDI than you or I)
2) A complete touch-screen calibration routine, which prompts the user to touch 3 different circles that are displayed consecutively on the screen, and then performs all of the calculations necessary to calibrate the resistive touch-screen to the EVE controller.

Any application that makes use of the touch-screen requires a calibration to be done, at least once when you first use the module. Having a built-in routine to handle this without any programming on your part, is a real advantage.

As I mentioned in Chapter 1, the EVE controller is unique in that it does not contain a video frame buffer- that is to say, a large SRAM array to hold all of the video pixel data. Instead it maintains a display list, in internal SRAM, of the attributes of all of the graphics elements that the user has asked to be displayed on the screen. So, to "paint" a screen with the required text, lines, and widgets, the programmer fills up this display list with the necessary commands needed to generate all of the necessary graphics elements.

Actually it is a bit more complicated than this because both the graphics engine and the co-processor engine each have their own display list. So, you would fill the graphics display list with the low-level graphics elements- basically lines, circles and edge strips. The higher-level "widgets", as well as the "FTDI logo" and "calibrate" functions are loaded into the co-processor's display list.

Although the EVE controller actually maintains the two discrete display lists mentioned in the last paragraph, there is a nice feature which I haven't yet mentioned. It turns out that the co-processor engine is able to process the lower-level graphics commands as well as its own higher-level commands (such as widgets). So, this means that if your graphics demands are not too high, you can ignore the graphics engine display list, and just load both your high and low-level commands into the co-processor's display list (ignoring the other list). This "shortcut" is used in FTDI's example programs, and is the method that I use in my own programs. Were you doing complex graphics, you might need to use both of the two engines simultaneously, and then you would have to fill both display lists as necessary.

What I haven't mentioned yet is that the EVE controller actually breaks these display lists into two sections. First you populate the display list with the commands to clear the screen and generate all of the necessary graphics elements needed for the current screen. Then you perform what is called a "swap": this allows the appropriate graphics engine to start rendering those graphics elements onto the display screen. At the same time, it points to a second area of RAM which acts as the storage space for a "new" display list. You are then free to fill up this "new" display list with graphics commands- without those commands actually interfering with whatever is currently being sent to the TFT display. The effect of this is that you get virtually instantaneous screen updates, regardless of the complexity of the graphics that you want rendered.

Another way of looking at this process is as follows. Once your host MCU has filled up a display list, and performed the “Swap” command, the SPI data transfer will stop, and your host MCU can proceed to do any other non-video related tasks required- until such time as you wish to either:

1) Change something on the display.
2) Check to see if the user has touched the screen.
3) Access the Audio engine present on the EVE controller.

I have over-simplified this process somewhat. In the case of the graphics engine display list, all commands are 32-bits long (4 bytes). So, it is relatively easy to sequentially fill up this list, checking to make sure that you haven't reached the maximum size of the list. The graphics display buffer size is 8K bytes so it can handle 2048 4-byte commands. I personally use the graphics co-processor to render all of the graphics elements that I use, so I don't actually use the 8K display buffer assigned to the graphics engine. Therefore, I am not sure if you can only use half of the 8K bytes of RAM for the list you are populating (with the other half used for the “swap” function).

In the case of the co-processor, the handling of the display list is somewhat different. This list is implemented as a 4K byte ring buffer. So, you start at address 0 and start filling the buffer up with co-processor commands (using the write pointer). There is also a read pointer for this ring buffer. While you are filling up the ring buffer, the graphics co-processor is concurrently reading from the buffer (using the read pointer) and is processing those commands. When the co-processor “sees” the CMD_SWAP message, it will transfer the graphics elements which it has processed to the actual display screen. The co-processor is fast, but it does take a finite amount of time to process each command that it receives. Therefore, as you are writing the co-processor commands to the ring buffer, you have to check to see whether the co-processor has gotten so far behind that you are “catching up with it from behind” in the ring buffer. This check is performed automatically by routines in the FTDI-supplied driver code, and I have not had to worry about it in my programs, as my graphics demands were not too high.

Actual Code

Let's look at a snippet of actual code found in my ***FT800_Basic_Setup*** demo program:

```
ft_void_t StartupScreen()
{
ft_uint16_t dloffset = 0,z;
Ft_CmdBuffer_Index = 0;
// Touch Screen Calibration -actual routine was removed, as proper values
have
// been found and are loaded into the
// proper registers by the routine  'storeTouchscreenCals()'
```

```
storeTouchscreenCals();  // Take values derived from an earlier-performed
Calibrate
// routine and store them in Touchscreen transform registers
// prepare startup Screen
Serial.println("Startup screen displayed");
Ft_Gpu_CoCmd_Dlstart(phost);
Ft_App_WrCoCmd_Buffer(phost,CLEAR_COLOR_RGB(219,180,150));
Ft_App_WrCoCmd_Buffer(phost,CLEAR(1,1,1));
// INFORMATION - puts up 2 lines of text, centered horizontally
Ft_Gpu_CoCmd_Text(phost,FT_DispWidth/2,20,30,OPT_CENTERX|OPT_C
ENTERY,              (char*)pgm_read_word(&info[0]));    // the 30 is the font
size
Ft_Gpu_CoCmd_Text(phost,FT_DispWidth/2,60,28,OPT_CENTERX|OPT_C
ENTERY,      (char*)pgm_read_word(&info[1]));
Ft_App_WrCoCmd_Buffer(phost,COLOR_RGB(255,255,255));

// place 2 option buttons
Ft_App_WrCoCmd_Buffer(phost,TAG('C'));     // TAG (identify the following
button with the        //TAG "C"
Ft_Gpu_CoCmd_Button(phost,20,100, 120,70,24,0,"Calibrate");   // place
calibrate Button
Ft_App_WrCoCmd_Buffer(phost,TAG('O'));     // TAG (identify the following
key with the   //TAG "O"
Ft_Gpu_CoCmd_Button(phost,200,100, 120,70,24,0,"Operate");      // place
operate button
Ft_App_WrCoCmd_Buffer(phost,DISPLAY());
Ft_Gpu_CoCmd_Swap(phost);
Ft_Gpu_Hal_WaitCmdfifo_empty(phost);
sk=0;
bool q= false;
do {
tmp=Read_keys();
if (tmp == 'C') {
Serial.println("Calibrate button pushed");
}
if (tmp == 'O') {
Serial.println("Operate button pushed");
}

} while (1);
}
```

Listing 1: FT800 Basic Setup Demo Code

What you see in Listing 1 is the code needed to compose the splash screen described earlier. This routine follows the generic set-up code needed to initialize an FT800 controller for ***any*** operation (as discussed earlier). In my program, this code is contained in the ***StartupScreen*** function. Note that since long lines that had to be split here, I have the 2nd part of the split line indented to make this splitting more obvious to the reader.

The first thing to notice, a few lines into the listing, is that I mention having removed the standard FTDI touchscreen Calibration function. All FTDI-supplied example programs contain this routine early in their programs, forcing you to do a calibration every time you run the program. This gets pretty boring when you are developing code, as you are repeating it every time you restart your program. After I tired of doing this, I decided to implement a work-around. I found out where these calibration values were being stored within FT800 registers. I then used the M328's serial port and the "C" Serial.println routine to send them out to a PC terminal program, where I wrote them down. Next I wrote a routine (storeTouchscreenCals) that took these values (expressed as constants) and sent them to the proper FT800 registers. The end effect is that I have a calibrated touchscreen without running a calibration routine at each M328 startup.

After my calibration-store routine, the next FT800 command found in this function is:

Ft_Gpu_CoCmd_Dlstart(phost);

This is a function telling the co-processor that we want to start a new display list. Like all of the FT800 functions, this command passes "phost" as a parameter. Don't worry about the meaning of this- as far as I know it is a "handle" that is defined elsewhere in the driver routines, but it does need to be specified (excuse my vagueness: I am not an expert C/C++ programmer!)

While it would be nice (i.e. consistent) if all FT800 co-processor commands were structured as above, they are not. The other common method of sending commands to the co-processor is demonstrated in the 2nd command:

Ft_App_WrCoCmd_Buffer(phost,CLEAR_COLOR_RGB(219,180,150));

Here we are writing to the co-processor command Ring buffer, the command CLEAR_COLOR_RGB(219,180,150). If you refer to the FT800 Programmer's Guide, you will find this command on page 112. Right away you will notice that this is NOT a co-processor command, but rather a graphics engine command. This demonstrates the feature that I mentioned earlier, where both graphics engine and co-processor engine commands can be inter-mingled, and sent to the co-processor display list.

This command specifies what RGB color is used for the background: i.e. when the screen is cleared. (the RGB values 219,180,150 produce a light pink/tan color).

Ft_App_WrCoCmd_Buffer(phost,CLEAR(1,1,1));

If you refer to the FT800 Programmer's Guide, you will find this command on page 109. When I first read over the Programmer's Guide, it seemed to me that, in order to perform this CLEAR function, that you were actually sending the FT800 the ASCII string "CLEAR 1,1,1)". It even looks this way in the program code itself. However this is not the case!

All graphics engine commands are 4-byte numbers. For this command, which can clear any/all of the color, stencil and tag buffers, the values of the parameters (the 1,1,1) are encoded into various bits of this 4-byte value. This is all looked after by routines in the driver library, as well as a huge list of #define statements (in file FT_GPU.h) which translate these “user-friendly” command names into 4-byte values, for use by the FT800. I'll go into the use of some of the color and tag buffers later on, but for now, all we need to know is that we should clear all three of them.

```
Ft_App_WrCoCmd_Buffer(phost,COLOR_RGB(255,255,255));
```

sets the foreground or drawing color to 255 for each of the red, green and blue phosphors (i.e. White).

Following this command is a long (2 line) command used to put some text on the screen. It is a somewhat complex statement, but centers the text horizontally, using font #30 (medium size). While it is possible to specify the text literally as an ASCII string like “sample text”, FTDI generally uses another method:

```
(char*)pgm_read_word(&info[0])
```

with the corresponding line which defines the string(s):

```
PROGMEM char *info[] = { "Basic Screen setup","Brian Millier"};
```

For either method of defining the string constant, this string constant is stored in program memory (FLASH). However, using the second method (shown above), when it's time to actually send the string to the FT800, the whole string is not copied into M328 SRAM and then transferred to the FT800. Instead, on a character by character basis, it is copied from FLASH to SRAM and then on to the FT800.Using this method, you save the amount of SRAM needed to hold the whole string, and this saving is repeated for every string constant that you need to display. Since the M328 SRAM is only 2K bytes, these savings can be important!

The next section of the code places the 2 button widgets on the screen. First, you issue   a command COLOR_RGB(255,255,255) to make the foreground (drawing) color WHITE. The color of the text within the button is what is being defined here.

In the case of the button widgets, you can change the default color (BLUE) of the button itself using the FG_COLOR command, but I did not do this here.

Before you actually send the command to place the button widget on the screen, you want to define this button with a TAG. What is a TAG? It is a byte (or character) that you associate with the area on the screen taken up by whatever graphics items you place on the screen ***after*** this TAG command and ***before*** the next TAG command. Then, whenever you touch the screen in this defined area, the touchscreen “engine” will report this TAG byte in the

FT800 register “REG_TOUCH_TAG”. So, here you can see that the first button that I place will be tagged “C”, which is a convenient abbreviation for the “Calibrate” function of the button. Note that you can use any byte value (or ASCII character value) for the TAG- apart from zero. Zero is the value that is stored in the “REG_TOUCH_TAG” resister when the touchscreen is NOT being touched. With this taken care of, you actually place the button using the following line:

```
Ft_Gpu_CoCmd_Button(phost,20,100, 120,70,24,0,"Calibrate");
```

See the programmer's Guide page 161 for the explanation of the 7 parameters passed to this routine. Note that in the case of button widgets, you are not passing “Cmd_Button” to the co-processor display list using the “Ft_App_WrCoCmd_Buffer” function, but are instead invoking the dedicated “Ft_Gpu_CoCmd_Button” function to accomplish this. In general, the graphic engine commands are sent using the “Ft_App_WrCoCmd_Buffer” function, whereas the graphics co-processor commands each have their own dedicated function call. If you want to use any of the co-processor commands listed in that section of the Programmer's Guide, please take a look in the file “FT_CoPro_Cmds.h” to see the exact function call definition used in the driver software. You also have to remain aware of the fact that all C/C++ variables, function names etc. are case-sensitive.
Aside from putting up another button on the screen, we are finished composing our start-up screen. All that remains is to execute the following few statements:

```
Ft_App_WrCoCmd_Buffer(phost,DISPLAY());
Ft_Gpu_CoCmd_Swap(phost);
Ft_Gpu_Hal_WaitCmdfifo_empty(phost);
```

These lines tell the FT800 to display the preceding graphics elements. The “Ft_Gpu_CoCmd_Swap” command tells the FT800 to swap the contents of this display list into another buffer which the co-processor uses to do the actual screen update, thereby freeing the 4K Ring buffer up for filling with the next display list. Since this takes a bit of time, the “Ft_Gpu_Hal_WaitCmdfifo_empty” function is called to wait until this transfer has finished.

At this point, we have an EVE display looking like Image 5. The “textured” appearance of the screen background is an “artifact”: likely the fault of my camera (or my photo technique) and does not show up on the actual screen.

All that remains is to handle the touchscreen itself. The last 11 lines of Listing 1 consist of a simple DO-WHILE loop where we call the Read_keys function, which returns the value stored in “REG_TOUCH_TAG” register. As long as the screen is not being touched, this function returns a zero, otherwise it returns the TAG byte that you have associated with the object on the screen that is being touched at the time. In this simple sample program, all I do is check for the character values for “C” or “O”, and print messages out the serial port indicating which button was pressed.

Next month, in the third part of the series, we'll look at some routines that we can use to provide things such as:

1) A numeric keyboard that can be used to allow the user to enter numeric data into your program.
2) A routine that allows you to plot X-Y data to a graph.
3) A Bar display which can be used to indicate battery voltage, and which changes bar color from green to yellow to red as the battery voltage decreases.
4) A simple function to provide some audible feedback to the user, via the FT800's Audio engine.

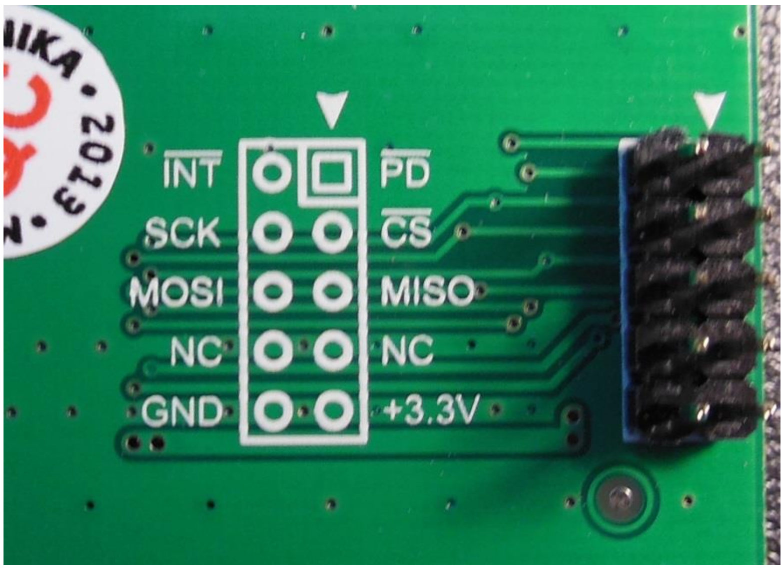

Image 3

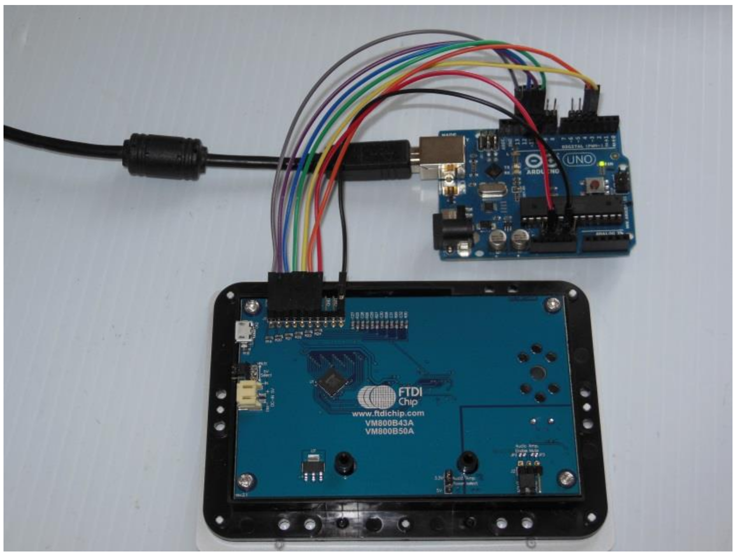

Image 4

Chapter 3 – Introducing some useful FT800 GUI Routines

At the end of Chapter 2, 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:

```
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);
```

Listing 2

Most of the commands have already been covered in Chapter 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:

REG_TOUCH_TRANSFORM_A

through
REG_TOUCH_TRANSFORM_F

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 2:

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

Listing 2

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

```
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);
}
```

Listing 3

When I wrote Chapters 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 web-site).

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:

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

It doesn't matter which method you use- in the program that I supplied with Chapter 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:

1) At program start, I will allow the user the option of running the calibration routine (Listing 3 above)
2) 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.
3) 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 3.

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 application 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 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. Image 6 shows this QWERTY keyboard:

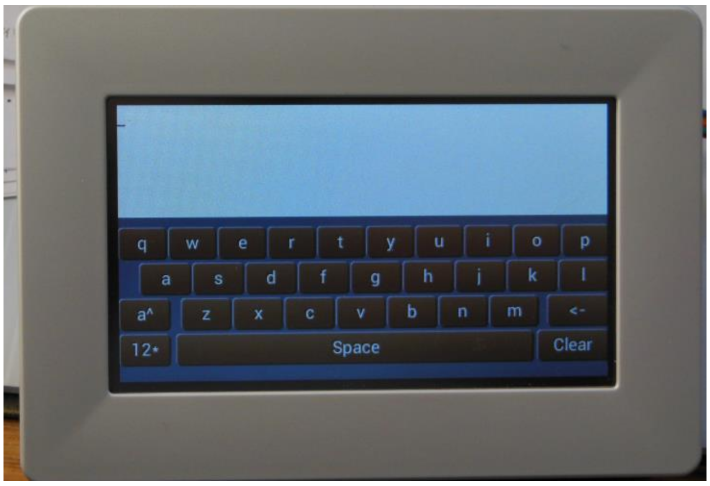

Image 6

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:

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

Let’s 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 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 Chapter 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:

```
// 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,"<-");
```

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

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:

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

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. Image 7 shows the keypad demo in action:

Image 7

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 Image 8:

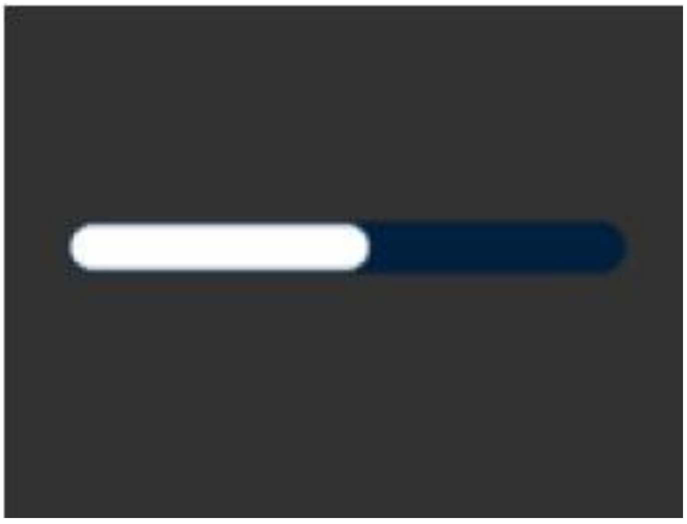

Image 8

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:

```
 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);
```

Listing 4

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:

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

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:

```
  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;
  }
```

Listing 5

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 Image 9, the FT800 also has a Gauge widget, which is similarly easy to use, and might be more attractive in your application:

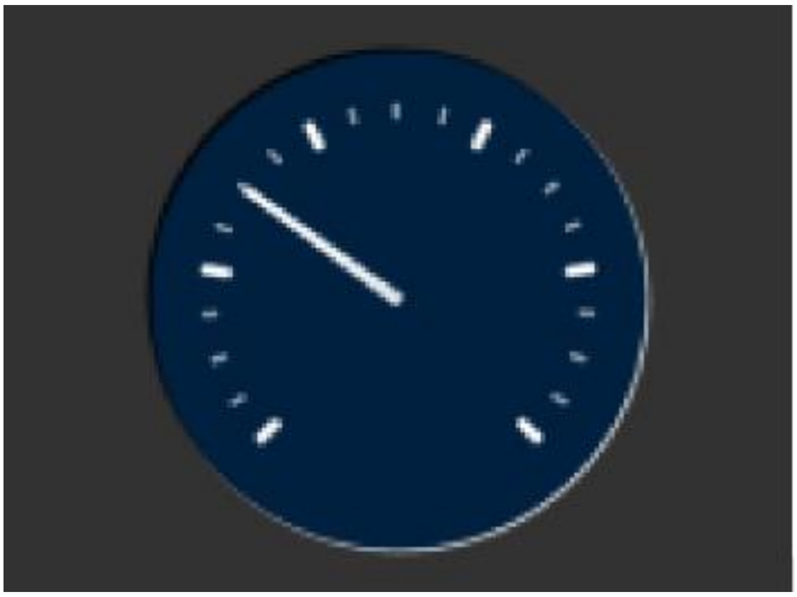

Image 9

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 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.

Image 10 shows a picture of the graph routine in action, with a simple saw tooth waveform being plotted:

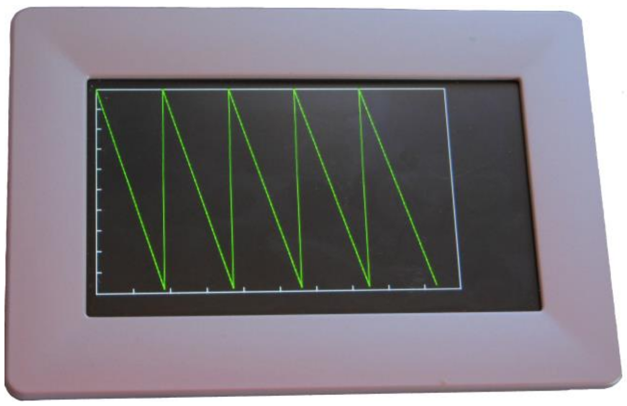

Image 10

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:

```
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));
```

Listing 6

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:

```
 // 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));
```

```
        }
```

Listing 7

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:

```
        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 8

In this program I have defined X and Y arrays with 250 elements (and filled them with the saw tooth 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.

Generating Sound using the FT800

The FT800 has an impressive sound-generating capability which I briefly mentioned in Chapter 1. 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 Chapter 2, 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 [9] 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.

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 wave files. However in both cases you needed to supply these wave files 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 [2]. However this short example is written in pseudo code, 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” [10] as shown in Image 11:

Image 11

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 [11]. 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
a short “beep”, useful as a GUI alert:

```
 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
```

Listing 9

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 saw tooth wave (voice #3) with a midi note of 64 (0x40) which is a frequency of 329.6 Hz. If you look at the saw tooth 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 300mS.

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:

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

```
        } while (sound_status ==1);
```

Listing 10

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.

Conclusions

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

[1] EVE FT800 Datasheet
http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT800.pdf

[2] EVE FT800 Programmers Guide
http://www.ftdichip.com/Support/Documents/ProgramGuides/FT800%20Programmers%20Guide.pdf

[3] AN_246 VM800CB SampleApp_Arduino_Introduction
http://www.ftdichip.com/Support/Documents/AppNotes/AN_246%20VM800CB_SampleApp_Arduino_Introduction.pdf

[4] FTDI Chip Sample Application for the Arduino:
http://www.ftdichip.com/Support/SoftwareExamples/EVE/FT800_SampleApp_1.0.zip

[5] FTDI Chip MPSSE module *(available via the FTDI Chip shop):*
http://apple.clickandbuild.com/cnb/shop/ftdichip?productID=418&op=catalogue-product_info-null&prodCategoryID=199

[6] Numeric Keypad demo program (Arduino sketch) *This zipped file contains my demo code along with all of the FTDI-supplied driver files.*

[7] Draw Graph demo program (Arduino sketch) *This zipped file contains my demo code along with all of the FTDI-supplied driver files.*
http://www.ftdichip.com/Support/Documents/ProgramGuides/FT800%20Programmers%20Guide.pdf

*[*8] Datasheet for the FTDI VM800BC43A:
http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_VM800B.pdf

[9] FTDI Musical instrument demo program
http://www.ftdichip.com/Support/Documents/AppNotes/AN_246%20VM800CB_SampleApp_Arduino_Introduction.pdf

[10] FTDI Chip Sample Application for the Arduino:
http://www.ftdichip.com/Support/SoftwareExamples/EVE/FT_App_Music.zip

[11] FTDI Application note AN_252 Audio Primer:
http://www.ftdichip.com/Support/Documents/AppNotes/AN_252%20FT800%20Audio%20Primer.pdf