

Brian's corner

The FTDI EVE graphics controller (2)



ELEKTRONIKAIKE svet



ISSN 1518-4679
letnik XXI
april 2014
števila 218
cena
4,50 €



RFduino



FTDI EVE
barvni TFT kontroler



Raspberry Pi in
Python za začetnike

Želite pospešiti razvoj analognih vezij?
Novi fotoelektrični senzorji OMRON
serije E3FA
Programiranje z Arduino (8) - varčni
Arduino
Elektrostatično občutljive elektronske
komponente (2)



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

Slovenian website

www.svet.el.si

English website

www.svet-el.si/english



www.svet-el.si/english

[Home](#)
[Blog](#)
[Inputs & Outputs](#)
[Data Displays](#)
[Data Measurement](#)
[Development Tools](#)
[Projects](#)
[Download](#)
[Brian's corner](#)

About the Book

- Table of Contents
- Authors
- Contact us
- Have your PCB
- My profile
- Shop
- Terms & conditions

Programming Blog

Brian is a well known author who uses different programming platforms and programs for his projects and articles. He regularly writes for different electronics magazines, including Svet elektronike (published in the Slovenian language).

Because Brian's articles are really great we have decided to publish them in English free of charge for all visitors of our web page.

Using NeoPixel LED's to Build a Unique Wall Clock

Written by Brian Miller
Friday, 27 May 2016 00:00



Like everyone interested in electronics, I'm hearing daily about the "Internet of Things" as well as "wearables" which I think is driven by an industry desperate to find the next "Great New Product" to sell us. I'm not sure how useful these new ideas will be.

[More...](#)


Last Updated on Tuesday, 31 May 2016 09:04

About the Book

- Table of Contents
- Authors
- Contact us
- Have your PCB
- My profile
- Shop
- Terms & conditions

Using NeoPixel LED's to Build a Unique Wall Clock

Written by Brian Miller
Friday, 27 May 2016 00:00



Like everyone interested in electronics, I'm hearing daily about the "Internet of Things" as well as "wearables" which I think is driven by an industry desperate to find the next "Great New Product" to sell us. I'm not sure how useful these new ideas will be.

However, I do feel that LEDs are products that will see an exponential growth in the future. There is no question that their efficiency and versatility make them the perfect choice for almost all general lighting needs. Although cost is an issue today, that will soon disappear because the cost of LEDs is on a steady downward trajectory.

You have likely seen large outdoor displays used for sporting events and concerts. These are very expensive, highly specialized displays, but a similar idea is now being found in smaller commercial displays used for promotional and advertising purposes. Such displays are usually a square meter and upwards in size, and are made up of thousands of individually-driven LEDs. Although displays like this used for traffic notifications are generally only one color, most of the units used for any form of advertising use RGB LEDs and can reproduce the whole color spectrum.

While the cost of such a large number of RGB LEDs is not insignificant, what is more of an issue is the problem of driving them in such a way that they can be individually controlled. You have probably built projects using multiplexed 7-segment LED displays, so you are familiar with that method of reducing the number of wires/drivers needed to handle multiple LEDs. Multiplexing of LEDs works because the human eye has a persistence of vision, which means it can't respond to changes in a light source at a frequency much beyond about 40 Hz. So, when we look at a 4-digit LED multiplexed display, our brain will make out the fact that, at any given moment, only one of the 4 LED digits is actually lit. While you can certainly use multiplexing techniques for large panels of LEDs, as you increase the multiplexing ratio, the "perceived" brightness will decrease as this ratio increases. Since brightness is a key factor for outdoor displays, this is definitely a limitation.

To facilitate the design of large LED display panels (and other related display products), the concept of the NeoPixel LED was invented. Initially, the idea was to develop a small IC that could control 3 LEDs (Red, Green, Blue) utilizing 8-bit PWM drive for each LED. Theoretically this gives "24-bit" color resolution: in practice the LEDs don't really yield this high a color resolution, but it is nonetheless very good. To further simplify the wiring, these LED driver ICs use a single-wire control protocol, and each IC device contains both a Data in and a Data out pin, allowing many such devices to be strung together in a "daisy-chain" configuration.

In practice, it's possible to string up to about 100 RGB LED driver ICs in this fashion, using only three lines (Vcc, ground and data). This is a tremendous reduction in the amount of wiring and circuitry needed to drive 300 completely independent LEDs (100 x Red, Green and Blue). While the original idea was to have this IC used with separate Red, Green and Blue LEDs (or a single RGB LED unit), it soon made sense to manufacture RGB LEDs with this driver IC built in. Such LEDs/drivers are called NeoPixel LEDs. These driver ICs were developed by World-Semi in Shenzhen, China, and, as far as I can tell, all of the NeoPixel modules come from the far-east. Let's look at some technical details of the most common NeoPixel IC driver chip, the WS2812.

| Parameter | Information | Maximum |
|------------------|------------------|---------|
| VCC | 5.0V to 5.5V | 5.5V |
| I _{LED} | 10.0mA to 15.0mA | 15.0mA |
| I _{DC} | 10.0mA to 15.0mA | 15.0mA |
| I _{PK} | 10.0mA to 15.0mA | 15.0mA |
| R _{LED} | 10Ω to 15Ω | 15Ω |

[Download program](#)

[Download article](#)

Using NeoPixel LED's to Build a Unique Wall Clock
2015_0226_31

Article on the web site

Download programs and
Download PDF of the article

WWW.SVET-EL.SI






Medio KIT - Svet elektronike magazine

www.svet-el.si/medeo_kit_2016.pdf

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

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

WWW.SVET-ME.SI






Presentation of Svet mehatronike magazine

www.svet-el.si/medeo_kit_2016.pdf

The FTDI EVE graphics controller (2)

In Part 1 of this series 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.

Basic Hook-up and Programming

Before choosing which display module that you wish to purchase, you should look back at Table 1 in Part 1 of the series, 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.

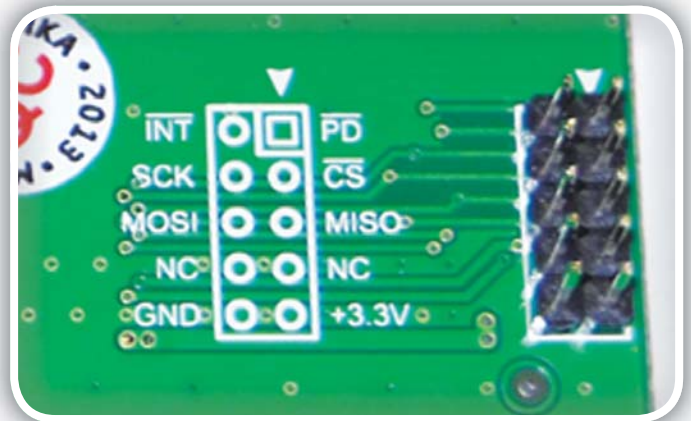


Photo 1

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 myself 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 Part 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 Part 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 titled “VM800CB_Sample_App_Arduino_Introduction” REF. 1

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 1 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 1: 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 1. 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:

- via the micro USB socket provided on the board and jumper 2 & 3 of SW1
- via CN1, (which is a 2 pin JST connector) and jumper 1 & 2 of SW1
- 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:

- via CN1 (the 2 pin JST connector) and short pins 7&8 of J5 and jumper pins 1&2 of SW1.
- 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 Photo 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. Photo 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:

- The program example supplied by FTDI as part of their application note AN246, mentioned earlier. A link to this can be found as REF2 in the reference section of this article.
- 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.

PROGRAMMING

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 Photo 3 at power-up, but you can see the serial port messages mentioned above, then you have either:
- a interconnection wiring problem
- 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)
- improper power supply to the EVE display module
- 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:

- written using C/C++ conventions rather than the simpler Arduino language syntax.
- 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.
- 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 REF. 3)

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

- 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.
- 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

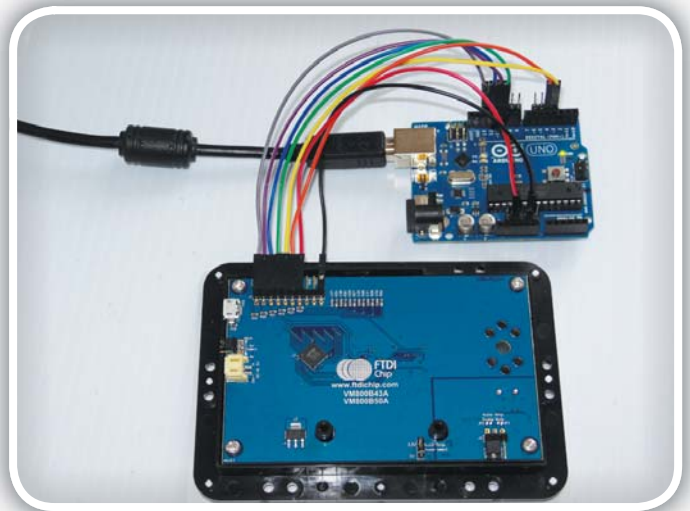


Photo 2

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 20 ms and then wait 20 ms 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 30 MHz. 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), **program 1**.

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:

- An FTDI banner screen with a moving logo (more useful to FTDI than you or I)
- 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 Part 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

PROGRAMMING

Program 1

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

or

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

Program 2

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

Program 3

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

Program 4

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

Program 5

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

Program 6

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

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 displaylist 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:

- Change something on the display.
- Check to see if the user has touched the screen.
- 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 this 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.

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.

QUIZ

Using the FT800 Programmer's Guide, find out what FT800 registers are used to store these calibration constants (hint: there are 6 of them). Compose a routine that restores these 6 values, as constants, to the proper FT800 registers.)

Next month in Part 3, I will give you the answer to this quiz,

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, but they are not. The other common method of sending commands to the co-processor is demonstrated in the 2nd command, **program 2**.

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, **program 3**, 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), **program 4**.

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`” register when the touchscreen is NOT being touched. With this taken care of, you actually place the button using the following line, **program 5**.

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, **program 6**.

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 Photo 3. 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:

- a numeric keyboard that can be used to allow the user to enter numeric data into your program.
- A routine that allows you to plot X-Y data to a graph.
- 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.
- A simple function to provide some audible feedback to the user, via the FT800's Audio engine.

I hope that you have been enjoying the FT800 EVE series so far, and hope to see you back again next month for the next part of the series.

References

- REF.1 FTDI AN_246 Application note:
 - ◇ http://www.ftdichip.com/Support/Documents/AppNotes/AN_246%20VM800CB_SampleApp_Arduino_Introduction.pdf
- REF. 2 FTDI Sample Application for the Arduino.
 - ◇ http://www.ftdichip.com/Support/SoftwareExamples/EVE/FT800_SampleApp_1.0.zip
- REF 3 FTDI MPSSE module available at their online store.
 - ◇ http://apple.clickandbuild.com/cnb/shop/ftdichip?productID=418&op=catalogue-product_info-null&prodCategoryID=199

www.svet-el.si/english