Application Note

BRT_AN_006

FT81x Simple PIC Example Introduction

Version 1.0

Issue Date: 2017-03-24

This application note provides an example of creating a simple screen on the FT81x device using the PIC MCU. It explains the low level SPI communication and provides a basic framework for sending EVE commands which can be extended to form a full application on a variety of different MCU types.

Bridgetek Pte Ltd (BRT Chip)
178 Paya Lebar Road, #07-03, Singapore 409030
Tel: +65 6547 4827 Fax: +65 6841 6071
Web Site: http://www.brtchip.com

Table of Contents

1 Introduction

1.1 Overview

This application note provides a simple example for the FT81x using the Microchip PIC device as the SPI master. It is intended to illustrate the basic low-level SPI transfers to the FT81x which an MCU would use and can be ported to a variety of MCU platforms and extended to create more complex applications. This guide includes the following topics:

- An introduction to creating screens via Display Lists and via the Co-Processor
- A basic hardware circuit interfacing a PIC MCU to the FT81x development modules
- A software example illustrating the following layers:
    - Application layer: Configuring the FT81x and a simple flashing dot application
    - EVE layer: Translates application calls to the SPI transfers required by EVE's API
    - MCU layer: Passes the EVE layer requests to the MCU hardware allowing the layers above to be independent of the MCU hardware.

Note that an additional set of Application functions building upon this framework is provided in BRT_AN_007.

1.2 Scope

The scope of this application note is to explain the lower level SPI transfers used to communicate with the FT81x and to provide the library functions for implementing the SPI transfers within the main code.

These can be used as the starting point of an application controlled by a PIC and can also be ported relatively easily to other MCUs. The demo covers only a subset of the commands and features in the programmers guide, but the full set of commands could be added by following the principles shown here.

More complex examples demonstrating the full EVE feature set for FT900 and Visual Studio host platforms are also available for the FT81x series (see EVE Examples). Additional examples based on the library provided in this application note can be found in BRT_AN_007.

1.3 Compatibility

The code provided is primarily targeted at the FT81x series of devices but could be modified to run on the FT80x series too as they have similar APIs. Note however that FT81x has some new commands and features not present on the FT80x series. A similar application note AN_320 is available for the FT80x series.

This example is written for the PIC family of MCUs (PIC18F46K22) using MPLABX IDE and a PICKit3 debugger. It can be ported both to other PIC devices and to other MCU types without major modification. The main tasks would be to import the C source and header files into the project of the target MCU and to edit the MCU layer code so that it interacts with the correct registers on the chosen MCU.

2 EVE Display Lists and Co-Processor

This section briefly discusses the GPU's Display List RAM (RAM_DL) and Co-Processor FIFO (RAM_CMD) which are used in creating EVE screens.

In an EVE application, it is important to note that the Display List always defines what will actually be displayed on the screen. The co-processor is there to assist with creating a display list and for performing other tasks which the graphics engine cannot do itself.

2.1 Writing a Display List to RAM_DL

The display list consists of the list of GPU instructions beginning at offset 0 and can be up to 8K in size. This is a linear buffer and a display list always begins at the lower address (RAM_DL + 0).

The GPU parses the entire display list every time it renders a line on the screen and so it is continually parsing the display list during normal operation. The display list may reference image and font data stored in the RAM_G allowing these to be displayed.

The general structure of a display list, including typical starting and ending procedures are as shown below. The blue text denotes the items to be drawn on the screen.

```
Clear_Color_RGB(R,G,B)     // Set color to clear screen to
Clear(1,1,1)               //clear the screen
Color_RGB(0,0,255)
Point_Size(20)
Begin(Points)
Vertex2F(0,0)
End()
Display                    // Tell GPU that this is the end of the list
[Register write to REG_SWAP]
```

Address	Instruction
RAM_DL	CLEAR_COLOR_RGB(0,0,0)
RAM_DL + 4	CLEAR(1,1,1)
RAM_DL + 8	COLOR_RGB(0,0,255)
RAM_DL + 12	POINT_SIZE(20)
RAM_DL + 16	BEGIN(POINTS)
RAM_DL + 20	VERTEX2F(0,0)
RAM_DL + 24	END()
RAM_DL + 28	DISPLAY

Now do a Swap to make list active

Figure 1 - Creating a display list

After writing a display list into RAM_DL, a swap is performed by writing to REG_DLSWAP.

Because the GPU parses the RAM_DL on every display line, the RAM_DL is double buffered and so there are two 8K buffers mapped to the same address range. The application edits the 8K buffer mapped in the foreground to add the display instructions for the next screen, whilst the GPU utilises the background buffer to render the current screen. This ensures clean transitions between screens. The updated content will *only* appear after the swap is complete.

An example of writing a display list is shown in the APP_Init function (see source code provided and section 5.2)

2.2 Creating a Display List in RAM_DL via the Co-Processor

The Co-Processor does not directly render screen content but instead acts as an assistant to creating screen content within the RAM_DL. One of the main uses of the co-processor is to take more complex items such as widgets which the GPU cannot process directly and create display list entries in RAM_DL that the GPU can use. For example, a button command is turned into a series of primitive shapes which the GPU can understand.

The Co-Processor also performs tasks which involve processing such as running calibration, taking a compressed image and inflating it into an area of RAM_G in a form which the GPU can reference from a display list. The co-processor can also access registers (for example, a CMD_SWAP can be used which results in REG_SWAP being written).

The co-processor can accept both, commands (e.g. CMD_BUTTON) which must be used via the co-processor, and GPU primitives (e.g. COLOR_RGB()) which could have been written to the RAM_DL. In the latter case, it passes these GPU instructions directly through to the created display list. This allows an entire screen to be created via the co-processor FIFO rather than mixing writes to RAM_DL and RAM_CMD which requires very careful memory management.

RAM_CMD Handling

The co-processor is fed commands via a 4K circular FIFO (RAM_CMD). The application must note the following:

- Always treat the FIFO as a true circular buffer with read and write pointers and must *not* begin every new command or set of commands at RAM_CMD+0. The application should check for free space and then determine the current value of REG_CMD_WRITE and use this as the starting point for the next command or set of commands.

- Always write multiples of four bytes as each command must begin at an offset with multiple of 4. Some commands such as buttons, text and sliders have parameters which may make the overall length of command plus parameters non-multiple of 4. In these cases, dummy 0x00 bytes should be sent at the end of the last parameter to pad it to a multiple of 4 bytes. BRT_AN_007 has some examples of this.

- The co-processor indicates a fault condition by setting REG_CMD_READ to 0xFFF. This could occur for example where invalid image data is supplied after CMD_INFLATE or CMD_LOADIMAGE, or if an attempt is made to load more than 2048 instructions into RAM_DL via the co-processor. The FT81x programmers guide has further information on this.

- For widgets, the resulting number of display list instructions to create the shapes may be significantly more than the number of bytes in the co-processor command for that widget. Therefore, it cannot be assumed that 2048 co-processor commands will make only 2048 display list entries. REG_CMD_DL can be checked as mentioned further down in this section to keep track of the number of DL entries.

Co-Processor writes to RAM_DL

The co-processor list begins with CMD_DL_START which sets the REG_CMD_DL register to 0, as this register defines the address in RAM_DL at which the co-processor will write the next display instruction to. The REG_CMD_DL value is updated by the co-processor as it writes entries in RAM_DL and so advanced applications may read this value *after* executing commands if required to know the position in RAM_DL which the co-processor will write next. One example is when copying sections of display list to RAM_G for recall via the Append command (see AN_340)

Co-Processor list structure

The general structure of a co-processor list when generating a screen is as shown below denoting the typical starting and ending procedures.

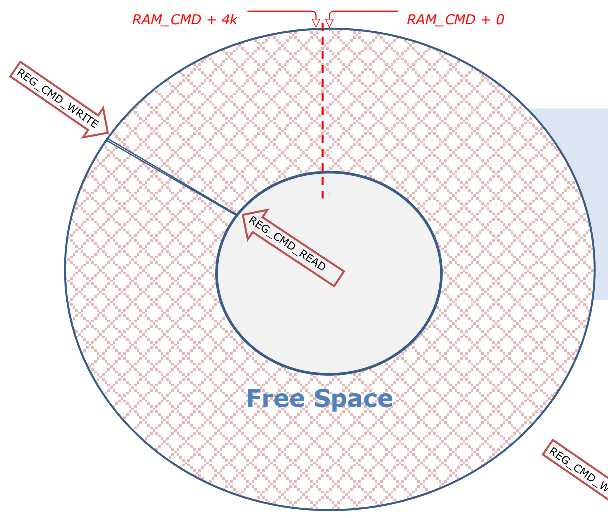
```
[Await REG_CMD_READ == REG_CMD_WRITE and read value of these registers]
CMD_DL_START        // Tells co-pro to begin writing DL at RAM_DL+0
CLEAR_COLOR_RGB     // Co-processor writes this instruction to the DL
CLEAR(1,1,1)        // Co-processor writes this instruction to the DL
Color_RGB(0,0,255)
Point_Size(20)
Begin(Points)
Vertex2F(0,0)
End()
DISPLAY             // Co-processor writes this instruction to the DL
CMD_SWAP            // Co-processor writes REG_DL_SWAP
[update REG_CMD_WRITE to point to end of new commands]
[Await REG_CMD_READ == REG_CMD_WRITE]
```

Note that the co-processor will not process any of the commands until the second-last step (writing REG_CMD_WRITE). The processing is complete and the screen will update only after the last condition (read and write pointers equal) becomes true.

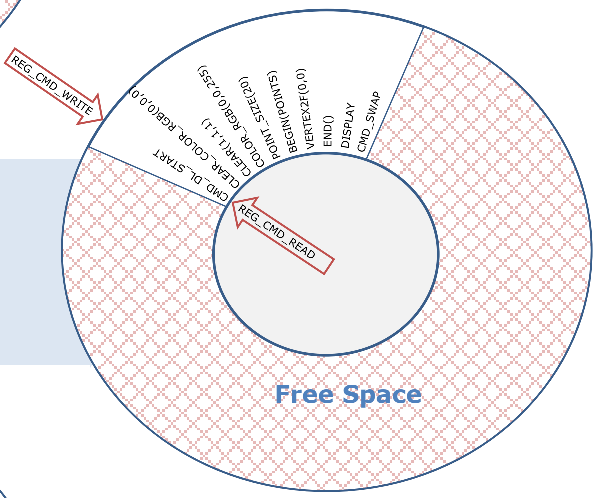
An illustration of writing this list to the co-processor can be found in Figure 2. The main applications within the sample code provided use the co-processor in this way.

FT81x features

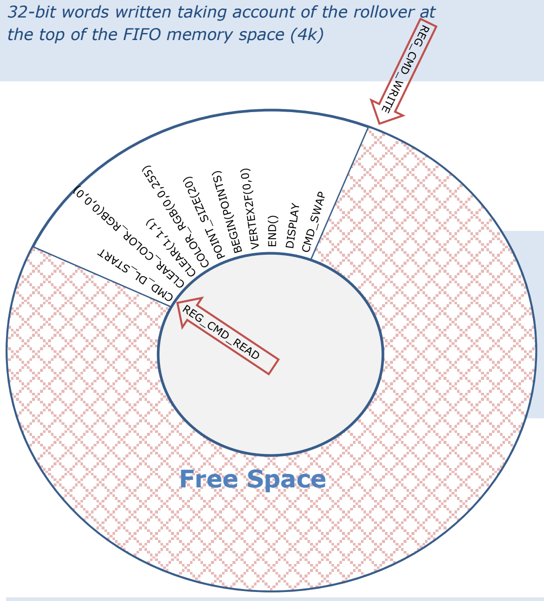
The FT81x series have an additional Bulk writing feature. The REG_CMDB_SPACE register indicates the amount of free space in the buffer. This can be used instead of awaiting the read and write pointers becoming equal. So long as the FIFO has enough free space for the commands to be sent, these commands can be written to register REG_CMDB_WRITE. Checking free space instead of awaiting the FIFO empty is especially useful if writing the compressed data to the FIFO following a CMD_INFLATE for example.



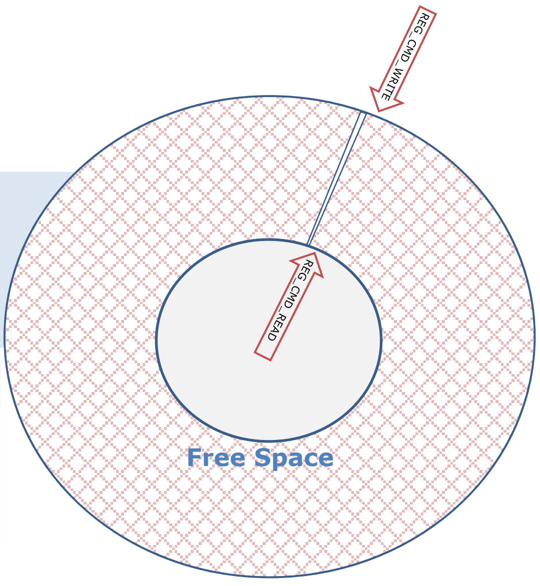
Step 1: The read and write pointers are initially equal and so the FIFO is empty. The application must use the current value of the REG_CMD_WRITE (starting offset) pointer as the first location of the next co-processor list.



Step 2: The application writes the new commands to the FIFO. The FT81x increments its internal address pointer automatically for each 32-bit command written and takes account of the rollover at 4K. The MCU must also keep a local count of the number of 32-bit words written taking account of the rollover at the top of the FIFO memory space (4k)



Step 3: The application now updates REG_CMD_WRITE to point to the end of the new list of commands based on the MCU's count (starting offset + local count). Since the FIFO's write pointer is now greater than the read pointer, the co-processor recognises that there are commands to be executed.



Step 4: The co-processor will now consume and execute each command in turn and will update REG_CMD_READ as it does so, until the pointers become equal. The display list will be generated in RAM_DL as the co-processor works through the commands and the swap will be carried out. The display list now appears as shown below:

Address	Instruction
RAM_DL	CLEAR_COLOR_RGB(0,0,0)
RAM_DL + 4	CLEAR(1,1,1)
RAM_DL + 8	COLOR_RGB(0,0,255)
RAM_DL + 12	POINT_SIZE(20)
RAM_DL + 16	BEGIN(POINTS)
RAM_DL + 20	VERTEX2F(0,0)
RAM_DL + 24	END()
RAM_DL + 28	DISPLAY

Execute swap

Figure 2 - Co-Processor usage to create display list

3 Hardware

The hardware is based around a PIC MCU and an FT81x module. The schematic is shown below. It includes an optional UART interface.

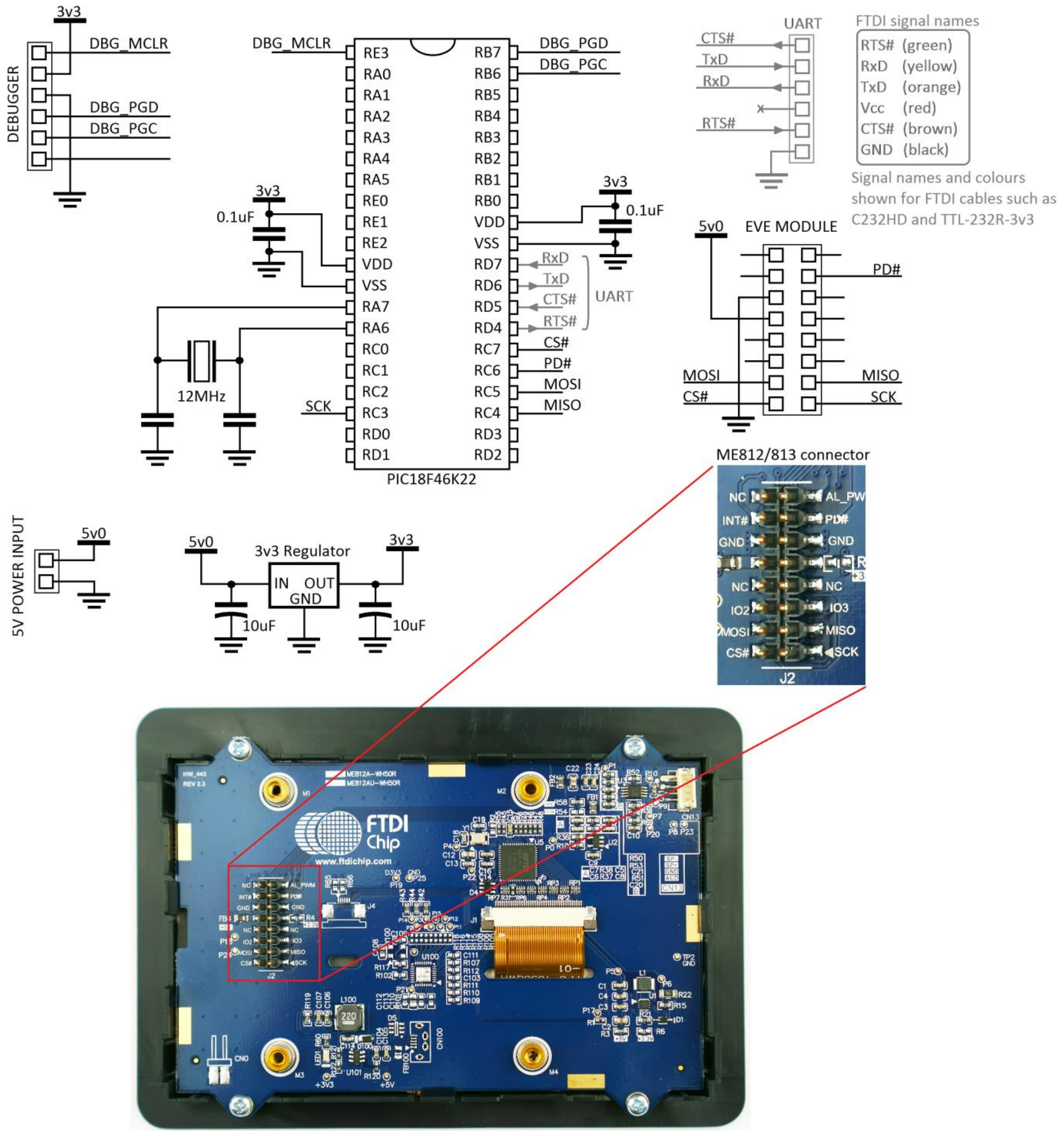


Figure 3 - Schematic of the PIC MCU with ME812-WH50R

3.1 FT81x Module

At the time of writing, the following modules were available from FTDI and are compatible with the code. The schematic illustrates the connection of an ME812 module in this case.

- ME813-WH50C
- ME812-WH50R
- VM810C50A-D

These modules contain the FT81x device along with all supporting circuitry and the LCD in order to easily get started with developing an FT81x application. In the case of the ME812/813, a pin header is provided to allow direct connection to one of the FTDI MM900EVxA MCU boards but can also be used to access the SPI signals to allow connection to other MCU boards. The SPI signals should use 3v3 levels. The VM810C has a single row pin header providing power and SPI signals.

3rd party modules with the FT81x are also currently available and should be compatible provided that they have connections for the SPI lines CS#, MOSI, MISO, SCK and the PD# line.

Note that the datasheet of any module selected should always be consulted to determine the pinout, power supply voltages, signal voltage levels and screen settings.

3.2 PIC MCU

The PIC18F46K22 device was selected in this case as it is available in several different IC packages and should allow easy porting to the other PIC18F series devices.

Since the FT81x carries out most of the work in creating the screen content, the requirement on the MCU in terms of speed and resources is normally low. The code will also work with smaller PIC MCUs running from their internal clock sources for example. Or likewise in many cases the EVE support can be added to a PIC which is running an existing application without requiring too much resource.

The PIC was run at 3v3 in this case via a small 3v3 regulator so that the signal levels on the SPI and GPIO lines to the FT81x module were at 3v3.

A 12MHz crystal was used and the internal x4 PLL enabled to run the device at 48MHz. The code was also tested successfully using the internal oscillator of the PIC. Note that when changing clock sources/settings or if porting to another PIC family, the delays used in the code may need updating so that they are maintained at the correct delay timing. This is especially important for the start-up delay of 500ms. It is important to consult the datasheet for the selected crystal and to calculate and select the appropriate load capacitor values as this can be crucial in ensuring reliable start-up and running in the final product or application.

The UART was also initialised although it is not utilised in this particular example. It can be used for debug purposes, or part of the main application allowing the PIC to communicate with other devices. It was also used in this case to upload the data for the screenshot shown in the main application. This topic is covered in application note BRT_AN_007. FTDI TTL cables such as the C232HD or TTL-232R-3v3 are ideal for connecting this UART port to the PC and have the added benefit of supporting non-standard baud rates and therefore allowing greater flexibility in selection of crystals. The Tx/Rx indicator LEDs in the C232HD cables make these especially well suited.

A PICkit3 debugger was used to allow programming and debug from MPLAB X on the PC.

4 Software

4.1 Overview

This application note is provided with a sample code project for MPLAB X IDE and was developed on version 3.15. The software is organised in several layers which are detailed below.

4.2 Software Layers

Application Layer

This layer implements the main application. It contains the code for initialising the display and then building up each application screen by creating co-processor lists. It will also include header files where required for image and font data.

This code can be edited to create the screen content required by the final application.

EVE Layer

This layer is called by the application layer functions and translates the display content-oriented calls from the main application into a series of SPI transfers formatted for the protocol used by the FT8xx. This allows the application layer above to focus on the screen content rather than the raw SPI values.

The project includes a header file which contains the definitions for the registers and also has bit-manipulation statements defined. Many commands have both the command and parameters combined into a single 32-bit command value and these bit manipulation definitions help to make the calls from the main code simpler.

MCU Layer

This layer provides an interface to the MCU hardware. It takes the SPI transfers from the EVE layer and translates them into the register level operations needed to control the MCU’s SPI Master peripheral as well as GPIO operations for chip select and power down.

If porting the provided code to another type of MCU or other SPI host platform, the code in this layer would be modified to suit the register map of the intended SPI Master whilst keeping the same syntax for calls to this layer from EVE layer above.

Figure 4 - Layers of the software example

4.3 Folder Structure

The project provided contains the following files in addition to any MPLAB-specific files:

Source Files

MCU_Layer.c	Contains MCU layer
EVE_Layer.c	Contains EVE layer
Main.c	Contains the Application layer

Header Files

Library.h	Contains function definitions etc. for the MCU/EVE layers
FT8xx.h	Contains the EVE register defines and bit-shifting functions to combine commands with parameters. It has defines for both FT80x and FT81x and so FT_81X_ENABLE must be defined for this demonstration.

Note that the project is provided with each layer in a different .c file as detailed above, for ease of readability and when porting to other MCU platforms. However, the content from EVE_Layer.c and MCU_Layer.c can be copied and pasted into the main.c file to make a single source file if preferred.

4.4 Usage Examples

The main application uses the following syntax when performing these common actions:

Setting PD# low or high using the dedicated functions from the MCU layer, which in turn perform a GPIO operation on the MCU port pin.

```
MCU_PDlow();

MCU_PDhigh();
```

Writing a register with a 16-bit data value. This function handles chip select, address to write, and data. Similar functions are available for writing 8-bit and 32-bit data sizes.

```
EVE_MemWrite16(REG_HOFFSET, lcdHoffset);
```

32-bit writes are also often used for writing to RAM_DL, for example the first two entries of a display list.

```
EVE_MemWrite32(RAM_DL+0, 0x02FF0000);                 // Clear Color RGB
EVE_MemWrite32(RAM_DL+4, (0x26000000 | 0x00000007)); // Clear(1,1,1)
```

Reading an 8-bit value from a register. This function performs the chip select operation, writing of the address and reading of the value. The value read is returned in variable FT81x_GPIO.

```
FT81x_GPIO = EVE_MemRead8(REG_GPIO);
```

Writing a burst of data to a series of sequential memory locations. A similar sequence is often used for loading bitmap data to RAM_G. This principle can also be used as part of a burst write of 32-bit values via EVE_Write32 to the co-processor (see the provided sample application for full details).

```
MCU_CSlow();                              // start a transaction
EVE_AddrForWr(FirstAddressToWrite);       // Write starting address
EVE_Write8(DataByte0);                    // stream data
EVE_Write8(DataByte1);
EVE_Write8(DataByte2);
EVE_Write8(DataByte3);
MCU_CShigh();                             // end transaction
```

5 Software - Application Layer

5.1 Overview

This layer is implemented in main.c. In the provided example code, this layer contains an initialisation section and then calls one of the demo routines.

```
    MCU_Init();
    APP_Init();
    APP_FlashingDot();
```

The functions MCU_Init() and APP_Init() initialise the MCU and FT81x respectively, and the code then remains in the flashing dot application which represents a very basic main application.

5.2 APP_Init()

This function performs the application's configuration of the FT81x including starting up and writing the display settings registers in the FT81x. It finishes by writing a short display list to clear the screen.

First, the PD line is asserted for 20msec and then de-asserted. This resets the FT81x and provides a clean start-up. The Active host command is then sent to wake up the FT81x. Note that the external oscillator mode may also be selected via a host command if required at this stage. Some modules use the internal oscillator and others may have an external crystal.

The FT81x requires a delay of at least 300ms to perform housekeeping actions including configuring the font/bitmap handles. This delay must be observed to ensure correct operation of the device. The example code uses a 500ms delay at this point.

After this, a read of the Chip ID register is performed and this must return the expected 0x7C value before proceeding. Failure to read this value could indicate an issue with the SPI connections or power to the EVE circuit for example.

A read of REG_CPURESET is also performed and must read value 0x00 before proceeding, which confirms that the FT81x is ready.

The display registers are then written to set the display parameters to match the connected LCD. The values provided are for 800x480 screens and will work with the ME812-WH50R, ME813-WH50C and VM810C50A-D modules but can be changed to suit other screens.

The GPIO lines are configured to enable the display, along with the touch threshold for resistive screens. The audio is not used here and so the volume is turned down. Note that the writing of the PCLK register and the PWM of the backlight can be done after the first display list to provide a cleaner start-up appearance to the user.

Finally, a short display list is created which clears the screen. Note that the commands begin at RAM_DL + 0 and are added to each sequential 4-byte offset. In this case, the Clear Color RGB specifies a black color and then Clear(1,1,1) clears the color, stencil and tag buffers. The Display command marks the end of the list, and the Swap will result in this display list becoming active. It is only after execution of the Swap that any change will be apparent on the screen.

```
    ramDisplayList = RAM_DL;                             // Start of Display List
    EVE_MemWrite32(ramDisplayList, 0x02000000);          // Clear Color RGB

    ramDisplayList += 4;                                 // point to next location
    EVE_MemWrite32(ramDisplayList, (0x26000000 | 0x00000007));    // Clear(1,1,1)

    ramDisplayList += 4;                                 // point to next location
    EVE_MemWrite32(ramDisplayList, 0x00000000);          // DISPLAY command

    EVE_MemWrite32(REG_DLSWAP, DLSWAP_FRAME);            // Swap display list
```

At this point, the SPI clock rate may be increased above 11MHz up to a maximum of 30MHz if required.

5.3 APP_FlashingDot()

This example draws a very simple dot on the screen which alternates in color between red and black, thereby appearing to flash red against the black background.

Figure 5 - Flashing Dot demo

The code begins by waiting for the co-processor FIFO to be empty whereby the write and read pointers are equal. The place within the circular co-processor FIFO to which they currently point is also obtained and is used as the starting point for the new co-processor list.

```
cmdOffset = EVE_WaitCmdFifoEmpty();
```

A while loop is used to run the remainder of the demo. A toggle variable is used to alternate the color each time round the loop.

The CS# line is brought low to begin an SPI transaction. The address (as obtained above) is then sent with the write bits set to indicate that this is the beginning of a data write transaction. Whilst the CS line is held low, data can be written in a burst write to the FT81x.

```
MCU_CSlow();
EVE_AddrForWr(RAM_CMD + cmdOffset);
```

Since this code is writing to the co-processor, the first command tells the co-processor to begin creating a new display list at location 0 of the display list RAM. All screen updates should begin with a CLEAR to clear the device buffers. This also results in clearing the screen to the color specified in the optional CLEAR_COLOR_RGB beforehand. This can be used as a convenient way to create a background color for the subsequent screen items. A counter variable cmdOffset keeps track of the number of bytes being used in the co-processor FIFO.

The CLEAR command and the DLSTART command mean that the previous screen contents will be cleared. In many cases, an application screen may consist mainly of items which do not change and the update is to only a few values etc. In this case, the techniques discussed in AN_340 may be used to minimise overheads.

```
        EVE_Write32(CMD_DLSTART);
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(CLEAR_COLOR_RGB(0,0,0));
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(CLEAR(1,1,1);
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

The demo now draws the point on the screen after setting the color and the point size.

```
        EVE_Write32 (COLOR_RGB(color,0,0));
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(BEGIN(FTPOINTS));
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(POINT_SIZE(point_size));
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(VERTEX2F(100*16,100*16));
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(END);
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

The co-processor list finishes with a DISPLAY command which, when actioned by the co-processor and added to the display list, tells the FT8xx that this is the end of the set of display items. The SWAP command performs the same task as writing to the swap register; once the display list has been written to the RAM_DL, this command will swap the foreground and background display list memory so that the newly written display list is now active on the screen.

```
        EVE_Write32(DISPLAY());
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        EVE_Write32(CMD_SWAP);
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

The Chip Select is now brought high to end the burst write cycle.

```
        MCU_CShigh();
```

As discussed in section 2 above, the commands have now been written to the co-processor FIFO but have not yet been executed and so the new screen content will not be visible. The co-processor is now instructed to execute them by setting its REG_CMD_WRITE write pointer to the end of the new items. In response, the co-processor will work its way through the new commands, updating its REG_CMD_READ read pointer as it does so.

```
        EVE_MemWrite32(REG_CMD_WRITE, (cmdOffset));
        cmdOffset = EVE_WaitCmdFifoEmpty();
```

Once the READ pointer becomes again equal to the WRITE pointer, the co-processor has executed all commands and will have started a new display list at location RAM_DL + 0, created the entries in RAM_DL, and swapped the display list. The new screen will now be visible on the LCD.

A short delay is added at the bottom of the While loop so that the user can visualise the alternating of the point between black and red in a blinking effect.

Note that in comparison to AN_320, this application uses burst writes for its co-processor writes instead of writing each 32-bit command with a separate address transaction each time, making it more efficient. The same technique can be used for all EVE family devices.

6 Software - EVE Layer

The EVE functions can be found in EVE_Layer.c. They provide the interface between the Application layer and the MCU layer, allowing the application to use a simpler syntax so that the developer can focus on the screen content. At the same time, the EVE functions are independent of MCU type as they use the MCU layer for all MCU-specific accesses.

6.1 Separate Address and Data functions

These functions send the address which is to be written or read. They take an unsigned long parameter which should have the address in the lower three bytes. The function configures the upper two bits of this 24-bit address to indicate to the FT8xx whether this is a write or a read operation and then sends the resulting three bytes.

6.1.1 Addressing

void EVE_AddrForWr (unsigned long ftAddress)

This function sends the 24-bit register address. It forces the MSB of the address to '10' which tells the FT8xx that this is a write operation and the byte(s) following the address will be written to that address. Since the address is 3 bytes, the upper byte passed in is ignored. The function then sends the address MS byte first.

e.g. EVE_AddrForWr (0x102428);

Value passed in:	00000000 00010000 00100100 00101000	(unsigned long)
Value written out of MOSI:	10010000 00100100 00101000	
Value returned:	N/A	

void EVE_AddrForRd(unsigned long ftAddress)

This function is similar to the one above but instead sets the upper two bits of the address to '00', which tells the FT8xx that this is a read of the location being addressed.

e.g. EVE_AddrForRd (0x102428);

Value passed in:	00000000 00010000 00100100 00101000	(unsigned long)
Value written out of MOSI:	00010000 00100100 00101000	
Value returned:	N/A	

6.1.2 Writing Data

These functions can be used to either write a value to a previously addressed 32-bit/16-bit/8-bit register or can be used to write values to a display or command list.

Before writing data, the address should be specified by using the Send Address WR functions in the previous section.

void EVE_Write32(unsigned long ftData32)

This function sends a 32-bit data value to the FT8xx. It does not make any change to the 32-bit data passed in, and it sends the data one byte at a time starting with the least significant byte.

e.g. EVE_Write32 (0x12345678);

Value passed in:	00010010 00110100 01010110 01111000	(unsigned long)
Value written out of MOSI:	01111000 01010110 00110100 00010010	
Value returned:	N/A	

void EVE_Write16(unsigned int ftData16)

This function is similar to the EVE_Write32 but it accepts a 16-bit value and sends two bytes out of the SPI interface.

e.g. EVE_Write16 (0x1234);

Value passed in:	00010010 00110100	(unsigned int)
Value written out of MOSI:	00110100 00010010	
Value returned:	N/A	

void EVE_Write8(unsigned char ftData8)

This function is similar to the EVE_Write32 and EVE_Write16 but it accepts an 8-bit value and sends one bytes out of the SPI interface.

e.g. EVE_Write8 (0x12);

Value passed in:	00010010	(unsigned char)
Value written out of MOSI:	00010010	
Value returned:	N/A	

6.1.3 Reading Data

These functions can be used to either read a value from a previously addressed 32-bit/16-bit/8-bit register. Before calling these, the address should be specified by using the Send Address RD functions in the previous section. The MCU's SPI hardware always writes a byte whenever it reads a byte and so dummy zero bytes are sent.

unsigned long EVE_Read32(void)

This function reads a 32-bit data value from a previously addressed register in the FT8xx.

e.g. MyValue = EVE_Read32()
(assume register being read contains 0x87654321 == 10000111 01100101 01000011 00100001)

Value passed in:	N/A		
Value written out of MOSI:	00000000 00000000 00000000 00000000	Value read in from MISO:	00100001 01000011 01100101 10000111
Value returned:	10000111 01100101 01000011 00100001		(unsigned long)

unsigned int EVE_Read16(void)

This function reads a 16-bit data value from a previously addressed register in the FT8xx.

e.g. MyValue = EVE_Read16()
(assume register being read contains 0x4321 == 01000011 00100001)

Value passed in:	N/A		
Value written out of MOSI:	00000000 00000000	Value read in from MISO:	00100001 01000011
Value returned:	01000011 00100001		(unsigned int)

unsigned char EVE_Read8(void)

This function reads an 8-bit data value from a previously addressed register in the FT8xx.

e.g. MyValue = EVE_Read8()
(assume register being read contains 0x21 == 00100001)

Value passed in:	N/A		
Value written out of MOSI:	00000000	Value read in from MISO:	00100001
Value returned:	00100001		(unsigned char)

6.2 Combined Address and Data functions

These functions combine the chip select, addressing and data operations to provide a single call which can write a value to a specified address or read a value from a specified address. These are ideal for writing/reading a single value to/from a register in the FT8xx.

Note: Chip select is asserted before the address and de-asserted after last data byte in all of these functions. For burst writes or reads where data is streamed to/from memory the individual addressing and data functions from the previous section can be used instead.

6.2.1 Writing a register

void EVE_MemWrite32(unsigned long ftAddress, unsigned long ftData32)

e.g. EVE_MemWrite32 (0x102428, 0x12345678);

Value passed in:	Addr	00000000 00010000 00100100 00101000
	Data	00010010 00110100 01010110 01111000
Value written out of MOSI:	10010000	
	00100100	
	00101000	
	01111000	
	01010110	
	00110100	
	00010010	
Value returned:	N/A	

void EVE_MemWrite16(unsigned long ftAddress, unsigned int ftData16)

e.g. EVE_MemWrite16 (0x102428, 0x1234);

Value passed in:	Addr	00000000 00010000 00100100 00101000
	Data	00010010 00110100
Value written out of MOSI:	10010000	
	00100100	
	00101000	
	00110100	
	00010010	
Value returned:	N/A	

void EVE_MemWrite8(unsigned long ftAddress, unsigned char ftData8)

e.g. EVE_MemWrite8 (0x102428, 0x12);

Value passed in:	Addr	00000000 00010000 00100100 00101000
	Data	00010010
Value written out of MOSI:	10010000	
	00100100	
	00101000	
	00010010	
Value returned:	N/A	

6.2.2 Reading a register

unsigned long EVE_MemRead32(unsigned long ftAddress)

e.g. MyValue = EVE_MemRead32 (0x102428);
(assume register being read contains 0x87654321 == 10000111 01100101 01000011 00100001)

```
Value passed in:              00000000 00010000 00100100 00101000     (unsigned long)
Value written out of MOSI:    00010000
                              00100100
                              00101000
                              00000000
                              00000000      Value read in from MISO:  00100001
                              00000000                                01000011
                              00000000                                01100101
                              00000000                                10000111
Value returned:         10000111 01100101 01000011 00100001           (unsigned long)
```

unsigned int EVE_MemRead16(unsigned long ftAddress)

e.g. MyValue = EVE_MemRead16 (0x102428);
(assume register being read contains 0x4321 == 01000011 00100001)

```
Value passed in:              00000000 00010000 00100100 00101000     (unsigned long)
Value written out of MOSI:    00010000
                              00100100
                              00101000
                              00000000
                              00000000      Value read in from MISO:  00100001
                              00000000                                01000011

Value returned:               01000011 00100001                       (unsigned int)
```

unsigned char EVE_MemRead8(unsigned long ftAddress)

e.g. MyValue = EVE_MemRead8 (0x102428);
(assume register being read contains 0x21 == 00100001)

```
Value passed in:              00000000 00010000 00100100 00101000     (unsigned long)
Value written out of MOSI:    00010000
                              00100100
                              00101000
                              00000000
                              00000000      Value read in from MISO:  00100001

Value returned:               00100001                                (unsigned char)
```

6.3 Host Command

void EVE_CmdWrite(unsigned char EVECmd, unsigned char Param)

This function sends the specified Host Command to the FT8xx. The command itself is passed into the function as an unsigned char. The FT81x uses the second unsigned char as a parameter for the command. A third byte, of value 0x00, is also sent to complete the host command.

For example, CMD_ACTIVE is 0x00 0x00 0x00

Value passed in:	Command	00000000	(unsigned char)
	Parameter	00000000	(unsigned char)
Value written out of MOSI:	00000000		
	00000000		
	00000000		
Value returned:	N/A		

6.4 Co-Processor FIFO Supporting Functions

unsigned int EVE_IncCMDOffset(unsigned int currentOffset, unsigned char commandSize)

This function is used when adding commands to the Command FIFO of the Co-Processor and handles the wraparound of the circular buffer. It takes in the current offset and the size of the last command. It returns the offset at which the next command will be written.

When a command is added to the FIFO, the MCU must calculate the offset at which the next command will be written. Normally, if a 4-byte command was written at (RAM_CMD + Offset), then the next command would start at (RAM_CMD + Offset + 4). However, since the FT8xx uses a circular buffer of 4096 bytes, the offset also needs to wrap around when offset 4095 is reached.

Note: This function allows the MCU to keep track of the FIFO write pointer. The FT8xx also keeps track internally. If performing a burst write of a co-processor list, the FT8xx will keep count of the bytes received including taking account of the internal rollover of the FIFO whilst CS# is held low. If the MCU uses this function to keep track of its internal counter cmdOffset, the FT8xx's internal pointer and variable cmdOffset will remain in sync even with FIFO rollover in the middle of a burst.

unsigned int EVE_WaitCmdFifoEmpty(void)

This function reads the values of the REG_CMD_WRITE and REG_CMD_READ pointers and waits for them to be equal. This indicates that the co-processor has consumed and processed all of the commands given to it.

When the pointers become equal, the function returns the current offset value of the pointers and this can be used as the starting offset (RAM_CMD + offset) for the next command, so that the RAM_CMD is used as a true circular FIFO.

6.5 Header file

The header file FT81x.h was copied from the main EVE Sample application.

This file provides definitions for the memory map, register names and for the EVE primitives and commands. In addition, since many EVE commands include not only the command opcode but also some parameters mapped into a single 32-bit value, the definitions help to make the main application code more readable. E.g. the call EVE_Write32 (COLOR_RGB(255,0,0)); is manipulated as follows:

```
#define COLOR_RGB(red,green,blue)
((4UL<<24) | (((red)&255UL)<<16) | (((green)&255UL)<<8) | (((blue)&255UL)<<0))
```

6.6 Command Implementation

It would be possible to create a middle layer which sits between the Application and EVE layers in order to provide a single line call to each co-processor command, including updating the offset for the command FIFO. This would allow the library to be made compatible with the text output from the EVE Screen Editor for example.

For example, in the application:

```
        MID_Begin_CoProList();            // see example below
        MID_DLStart();
        MID_ClearColorRGB(0,0,0);  // see example below
        MID_Clear(1,1,1);
        MID_Display();
        MID_Swap();
        MID_Finish_CoProList();           // see example below
```

With supporting functions in the MID layer such as:

```
        MID_Begin_CoProList()
        {
                cmdOffset = EVE_WaitCmdFifoEmpty();
                MCU_CSlow();
                EVE_AddrForWr(RAM_CMD + cmdOffset);
        }

        MID_ClearColorRGB(unsigned char r, unsigned char g, unsigned char b)
        {
                EVE_Write32 (COLOR_RGB(color,0,0));
                cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
        }

        MID_Finish_CoProList()
        {
                MCU_CShigh();
                EVE_MemWrite32(REG_CMD_WRITE,(cmdOffset));
                cmdOffset = EVE_WaitCmdFifoEmpty();
        }
```

The EVE sample apps available at EVE Samples (for platforms FT900 and Visual Studio) have additional layers in the c and header files which could be used as the basis for this e.g. FT_CoPro_Cmds.c. The naming of the calls could be tailored to match for example a text output copied and pasted from the EVE Screen Editor.

7 Software - MCU Layer

The MCU layer can be found in MCU_Layer.c. This layer contains all of the code which directly accesses the MCU's register map for configuration, GPIO and SPI. The code in this section can be changed to suit a different PIC microcontroller or another type of microcontroller so that the layers above can access the relevant registers and peripherals. This example uses the standard SPI interface found on most MCUs and so should be portable across a wide variety of devices without significant changes.

This code uses the following data types, which may need to be converted depending on the types supported by the target MCU's development tools:

unsigned long	32-bit unsigned number
unsigned int	16-bit unsigned number
unsigned char	8-bit unsigned number

Note: These functions are intended to perform a basic set-up of the PIC so that the EVE functionality can be demonstrated. The designer must consult the product documentation provided by the manufacturer of their selected MCU to confirm the correct set-up and best practices are followed so that reliable operation of the final product can be assured. The latest documentation for the selected EVE device and module should also be consulted. In the case of differences between the information provided in this document/code and the datasheets of the devices used, the product datasheet should take precedence.

7.1 Initialisation

void MCU_Init(void)

This function, along with the definitions at the top of the main.c file, performs a relatively basic configuration of the MCU to set the GPIO port pins used for the EVE signals and to configure the SPI peripheral. The definitions at the top of the code also set the oscillator for use with a 12MHz crystal and x4 PLL which runs the PIC at 48MHz.

The code was also tested and ran well with only the internal oscillator of the PIC. Note that when changing clock settings, the delays specified (especially the at-least-500ms delay on start-up) must be maintained. Also, the SPI clock rate must be kept below 11MHz during initial start-up and until after the MCU_Init and EVE_Init functions have been called. If running on a faster PIC, adjustments to delays and prescalers may be required.

7.2 GPIO Functions

These functions set or clear a GPIO line on the MCU for PD# and CS#

void MCU_CSlow(void)

This function will put the port pin assigned to the Chip Select of the FT81x to the low state.

void MCU_CShigh(void)

This function will put the port pin assigned to the Chip Select of the FT81x to the high state.

void MCU_PDlow(void)

This function will put the port pin assigned to the Power Down pin of the FT81x to the low state.

void MCU_PDhigh(void)

This function will put the port pin assigned to the Power Down pin of the FT81x to the high state.

7.3 SPI Functions

unsigned char MCU_SPIReadWrite(unsigned char DataToWrite)

This function writes a byte to the PIC's SPI peripheral which will be clocked out of MOSI. The peripheral will simultaneously clock in a byte on MISO which is returned by the function.

7.4 UART Functions

These functions provide a basic UART configuration for debug purposes or outputting data such as screenshot data to the PC. The reader must consult the documentation for their selected PIC in order to confirm the required settings for their application and should also adjust the baud rate divisors based on their crystal and MCU bus frequency to achieve the required baud rate.

The FTDI C232HD or TTL-232R-3v3 cables are ideal for interfacing the UART to a PC via its USB port and can be connected directly to the PIC. 5V signal versions of the TTL-232R cables are also available. The voltage chosen should match that of the PIC VCCIO.

void UART_Init(void)

This function sets the UART up for approximately 57600 baud based on a 12MHz crystal with x4 PLL enabled. It configures UART2 which can be found on pins RD6 (TxD) and RD7 (RxD). RD5 and RD4 can be used as GPIO lines controlled by these UART functions to provide CTS# and RTS# for flow control.

void UART_Tx(unsigned char SerialTxByte)

This function transmits one byte via the UART.

unsigned char UART_Rx(void)

This function receives one byte via the UART.

7.5 Delay Functions

Functions are provided for 20ms and 500ms delays. These help to improve readability of the main application by having a single call for each delay. They may require adjustment depending on the bus frequency of the MCU selected.

void Delay_20ms(void)

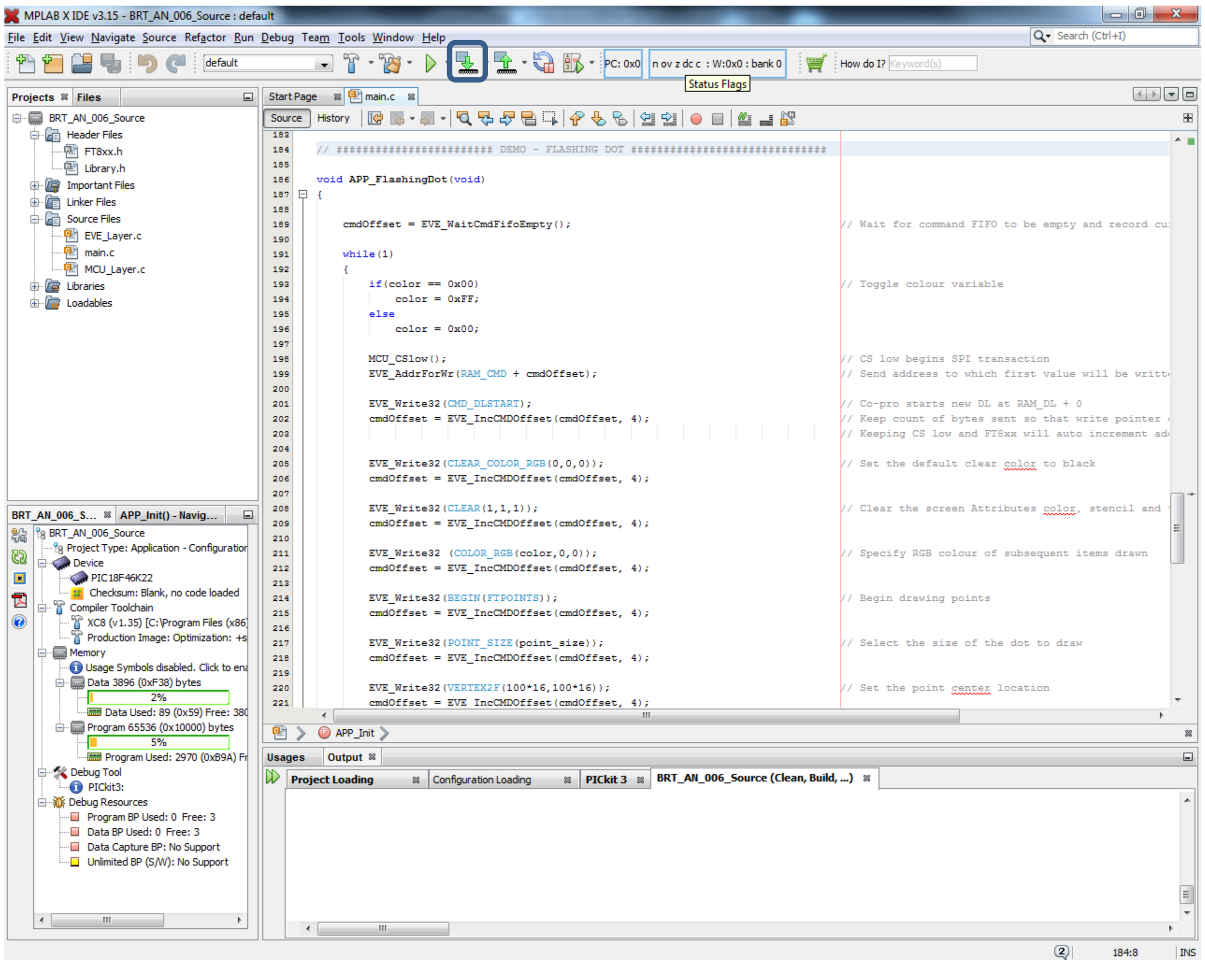
void Delay_500ms(void)

8 Using the Demo Application

To load the project, ensure that MPLAB X is installed on the PC. The latest download can be obtained from Microchip http://www.microchip.com/mplab/mplab-x-ide

The provided file BRT_AN_006_Source.zip (see Appendix A– References) can be un-zipped and the resulting folder copied to the user's normal project workspace directory. E.g. this is often c:\Users\[username]\MPLABXProjects\

Then go to File -> Open Project and select BRT_AN_006_Source. The project should now appear in the Projects window.


MPLAB X IDE v3.15 - BRT_AN_006_Source : default
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
default
PC: 0x0
n ov z dc c : W:0x0 : bank 0
Status Flags
How do I?
Search (Ctrl+I)
Projects
Files
Start Page
main.c
Source
History
APP_Init
Usages
Output
Project Loading
Configuration Loading
PICkit 3
BRT_AN_006_Source (Clean, Build, ...)
184:8
INS

BRT_AN_006_Source
- Header Files
  - FT8xx.h
  - Library.h
- Important Files
- Linker Files
- Source Files
  - EVE_Layer.c
  - main.c
  - MCU_Layer.c
- Libraries
- Loadables

BRT_AN_006_S... APP_Init() - Navig...
- BRT_AN_006_Source
  - Project Type: Application - Configuration
  - Device
    - PIC18F46K22
    - Checksum: Blank, no code loaded
  - Compiler Toolchain
    - XC8 (v1.35) [C:\Program Files (x86)
    - Production Image: Optimization: +s
  - Memory
    - Usage Symbols disabled. Click to ena
    - Data 3896 (0xF38) bytes
      - 2%
      - Data Used: 89 (0x59) Free: 380
    - Program 65536 (0x10000) bytes
      - 5%
      - Program Used: 2970 (0xB9A) Fr
  - Debug Tool
    - PICkit3:
  - Debug Resources
    - Program BP Used: 0 Free: 3
    - Data BP Used: 0 Free: 3
    - Data Capture BP: No Support
    - Unlimited BP (S/W): No Support

```

// ######################## DEMO - FLASHING DOT ##############################

void APP_FlashingDot(void)
{

    cmdOffset = EVE_WaitCmdFifoEmpty();                     // Wait for command FIFO to be empty and record cu

    while(1)
    {
        if(color == 0x00)                                   // Toggle colour variable
            color = 0xFF;
        else
            color = 0x00;

        MCU_CSlow();                                        // CS low begins SPI transaction
        EVE_AddrForWr(RAM_CMD + cmdOffset);                 // Send address to which first value will be writt

        EVE_Write32(CMD_DLSTART);                           // Co-pro starts new DL at RAM_DL + 0
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);         // Keep count of bytes sent so that write pointer
                                                            // Keeping CS low and FT8xx will auto increment ad

        EVE_Write32(CLEAR_COLOR_RGB(0,0,0));                // Set the default clear color to black
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

        EVE_Write32(CLEAR(1,1,1));                          // Clear the screen Attributes color, stencil and
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

        EVE_Write32 (COLOR_RGB(color,0,0));                 // Specify RGB colour of subsequent items drawn
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

        EVE_Write32(BEGIN(FTPOINTS));                       // Begin drawing points
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

        EVE_Write32(POINT_SIZE(point_size));                // Select the size of the dot to draw
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

        EVE_Write32(VERTEX2F(100*16,100*16));               // Set the point center location
        cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

Figure 6 - MPLABX screenshot

Ensure that the debugger is connected to the PC and to the PIC circuit and is showing up correctly under the debug tools section. Select the Run -> Clean and Build option to build the project. The highlighted button 'Make and program Device' can then be used to download the code to the PIC.

After programming is complete, the PIC will reset and begin running the code. The screen of the FT81x module should show a black background with a small red dot flashing as shown in Figure 5.

Note: The instructions above for the Microchip MPLABX tools and debugger tools are correct at the time of writing but may change in the future independent of Bridgetek. Please refer to the MPLABX documentation from Microchip for the latest information on loading/configuring projects and configuration of the debugger tools.

9 Conclusion

This application note has presented a simple example of driving the FT81x series from a PIC microcontroller. It has illustrated the way in which the calls from the main application are translated to the API used by the EVE devices and then on to the low level SPI transfers used by the MCU. This can form the basis of a more comprehensive FT81x library and application on a variety of different microcontrollers and other devices which have an SPI Master.

The code can be developed in a variety of ways including:

- Porting to other MCU platforms by editing the MCU_Layer
- Extending the code to provide a middle layer which accepts commands in other syntaxes to be compatible with tools such as EVE Screen Editor
- Editing the main application to create a final application with greater functionality

After gaining familiarity with this application note and code, the reader is encouraged to refer to application note BRT_AN_007 which contains some examples of extending the main application.

10 Contact Information

Head Quarters – Singapore

Bridgetek Pte Ltd
178 Paya Lebar Road, #07-03
Singapore 409030
Tel: +65 6547 4827
Fax: +65 6841 6071

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Branch Office – Taipei, Taiwan

Bridgetek Pte Ltd, Taiwan Branch
2 Floor, No. 516, Sec. 1, Nei Hu Road, Nei Hu District
Taipei 114
Taiwan , R.O.C.
Tel: +886 (2) 8797 5691
Fax: +886 (2) 8751 9737

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Branch Office - Glasgow, United Kingdom

Bridgetek Pte. Ltd.
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales)	sales.emea@brtichip.com
E-mail (Support)	support.emea@brtchip.com

Branch Office – Vietnam

Bridgetek VietNam Company Limited
Lutaco Tower Building, 5th Floor, 173A Nguyen Van Troi,
Ward 11, Phu Nhuan District,
Ho Chi Minh City, Vietnam
Tel : 08 38453222
Fax : 08 38455222

E-mail (Sales)	sales.apac@brtchip.com
E-mail (Support)	support.apac@brtchip.com

Web Site

http://brtchip.com/

Distributor and Sales Representatives

Please visit the Sales Network page of the Bridgetek Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A– References

Document References

FT81x

ME813-WH50C Datasheet

ME812-WH50R Datasheet

VM810C50A-D

FT81x Programmer Guide

FT81x Datasheet

EVE Examples (including BRT_AN_006.zip)

PIC18F46K22

PICKIT3

BRT_AN_007

Acronyms and Abbreviations

Terms	Description
EVE	Embedded Video Engine
FT81x	Latest version of the EVE family with enhanced feature set
LCD	Liquid Crystal Display
MPLAB X	Development environment software for PIC MCUs
PIC	PIC Microcontroller family from Microchip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter for serial data transfer

Appendix B – List of Tables & Figures

List of Figures

List of Tables

NA

Appendix C– Revision History

Document Title:	BRT_AN_006 FT81x Simple PIC Example Introduction
Document Reference No.:	BRT_000082
Clearance No.:	BRT#079
Product Page:	http://brtchip.com/i-ft8/
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	Initial version	2017-03-24