Application Note

BRT_AN_007

FT81x Simple PIC Example Demo Functions

Version 1.0

Issue Date: 2017-03-24

This application note provides a set of simple examples for the FT81x using the PIC MCU library functions which were introduced in BRT_AN_006.

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 is part of a series providing some simple examples for the FT81x using the Microchip PIC microcontroller as the SPI master. These examples are 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. The application is based on the same MCU and EVE software layers as BRT_AN_006 but provides additional demo features in the Application Layer to demonstrate usage of various FT81x features such as text, bitmaps and touch controls.

1.2 Scope

The scope of this application note is to explain, via a collection of small examples; the lower level coding required using the FT81x features in an application. Each demo is intentionally very basic and covers only one topic to aid readability.

These can be used as the starting point for developing a full application based on a PIC and can also be ported relatively easily to other MCU`s. The demo code 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.

This application note focuses on the main application itself and assumes familiarity with the lower level EVE and MCU layers. These are covered in more detail in BRT_AN_006.

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 the FT81x has some new commands and features not present in the FT80x series, such as larger RAM_G, higher screen resolution capability and VERTEX_TRANSLATE commands. Application note AN_390 explains the considerations when migrating from the FT80x to the FT81x.

This example is written for the PIC family of MCUs (PIC18F46K22) using MPLABX IDE and a PICKit3 debugger. It should be able to be ported both to other PIC devices and to other MCU types without major modification. The main tasks would be to port 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.

Note that the MPLAB X project provided with this application note includes a copy of the MCU and EVE layers as covered in BRT_AN_006 and is ready to use without needing to import them. These layers may have additional functions added for the purposes of supporting the demos provided here.

2 Software

2.1 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.

2.2 Software Layers

Application Layer This layer implements the main application. It contains the code for 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 also 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 the EVE layer above.

Figure 1 - Layers of the software example

2.3 Folder Structure

The project provided contains the following 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.
EVE_RGB565.h	Image data for the EVE bitmap demo
DIGITfont.h	Font data for the DS_DIGITS custom font demo

The zip file supplied also has a folder containing various supporting files:

Supporting Files

APP_ConvertedBitmap	Contains original image and converted files
APP_DigitsFont	Contains original font and converted files
APP_SnapShot2	Contains MTTTY_D2XX program and source code for Visual Basic NET uploader program

3 Main Application

3.1 Overview

In the provided example code, this layer contains an initialisation section and then calls one of the demo routines.

```
MCU_Init();
APP_Init();
MCU_UART_Init();
//APP_Calibrate();

// Important Note: Enable only one demo below at a time
// Enable APP_Calibrate above if running any touch-related demos

APP_FlashingDot();            // Creating a basic screen via co-pro

//APP_VertexTranslate();      // Placing items beyond coords of 511

//APP_Text();                 // Simple text with built-in font

//APP_ConvertedBitmap();      // Load and display small bitmap

//APP_DigitsFont();           // Load and use a custom font

//APP_SliderandButton();      // Demonstrate slider and button with touch


// The APP_SnapShot2 function can be called to take a snapshot from within one of
the functions above. It is commented out in each demo initially and section 8.1
should be consulted before using the SnapShot feature.
```

The functions MCU_Init() and APP_Init() should always be run.

The UART configuration function is required if using the snapshot feature or if using UART for debugging. In the code provided, it is only used for the snapshot at this time.

The calibration function is required to be un-commented if using any touch-enabled demos such as APP_SliderAndButton. It will request the user to tap the three calibration dots on start-up if enabled. It can be left enabled even if running other non-touch demos however.

The sample code then proceeds to call the selected demo. The code was designed to have one demo routine enabled at a time and it will then stay in that demo routine indefinitely. This was primarily done to ensure readability of the code and to allow each key topic to be clearly demonstrated, and since the developer will wish to customise the overall application in any case to meet the needs of their own product.

The code must be re-compiled and programmed after changing the demo selection. It could however be edited to run each for a certain amount of time or even to add a menu to select the required demo.

The following sections now discuss each demo in turn and provide background information on them.

4 Initialisation

4.1 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 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 if required at this stage.

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
```

Note: The MCU_Init and MCU_UART_Init are in the MCU layer and are covered in BRT_AN_006.

5 Examples - General Features

5.1 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 2 - 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 bit 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 FT8xx.

```
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 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 processed 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();
```

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 will start 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: 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.

5.2 APP_VertexTranslate

This section gives a simple demonstration of the VERTEX_TRANSLATE_X and _Y commands. These commands allow the application to make use of the full screen.

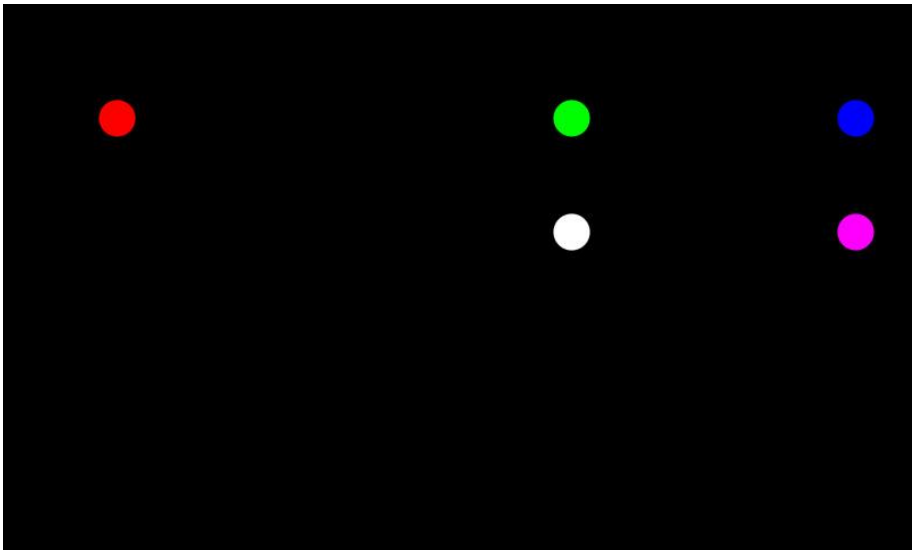


Figure 3 - Vertex Translate demo

Since the original FT80x series only supported screen coordinates up to 512, the VERTEX command parameters only allow values from 0 to 511. The new FT81x now supports up to 800 x 600 and so additional commands VERTEX_TRANSLATE_X and VERTEX_TRANSLATE_Y add an offset to any subsequent VERTEX commands to allow them to reach the full extent of the screen.

First, a red point is drawn at (100,100) which are within the limits of the standard VERTEX instruction. A green point is also drawn at (500,100) which are still within the bounds of the VERTEX command.

Red point @ (100, 100)

```
EVE_Write32 (COLOR_RGB(0xFF,0,0));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX2F(100*16,100*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4
```

Green point @ (500, 100)

```
EVE_Write32 (COLOR_RGB(0,0xFF,0));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX2F(500*16,100*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

A Blue point is then drawn at (750,100). This requires setting an offset of 250 in the X direction via VERTEX_TRANSLATE_X which will add to the 500 specified in the VERTEX call.

Blue point @ (750,100)

```
EVE_Write32 (COLOR_RGB(0,0,0xFF));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX_TRANSLATE_X(250*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX2F(500*16,100*16));
```

```
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

A further point coloured Purple is drawn at (750,200) demonstrating that the offset of 250 set above has been retained for this point.

Purple point @ (750,200)

```
EVE_Write32 (COLOR_RGB(0xFF,0,0xFF));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX2F(500*16,200*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

Finally, the VERTEX_TRANSLATE_X is set back to 0 again and so the white point appears at point (500,200) as per the values stated in the VERTEX call.

White point @ (500,200)

```
EVE_Write32 (COLOR_RGB(0xFF,0xFF,0xFF));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX_TRANSLATE_X(0*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
EVE_Write32(VERTEX2F(500*16,200*16));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

The same principle can be used if required for the Y direction, where VERTEX_TRANSLATE_Y specifies the offset in the Y direction. Note that many FT81x modules have screens with a resolution of 480 in the Y direction and so the Y translate may not be required.

5.3 APP_Text()

This demo writes a basic text string to the screen using the FT8xx Text feature.

Figure 4 - Text demo

The demo begins in the usual way by clearing the screen and starting a new display list. The color is then set to white for the text via the standard COLOR_RGB command.

The CMD_TEXT command is then used to write the text string.

Referring to the FT81x programmer's guide, the Text command has several parameters. The size of each value is also shown and this is reflected in the use of the Write8, Write16 and Write32 functions.

```
EVE_Write32 (CMD_TEXT);          // command text 0xFFFFFF0C
EVE_Write16 (100);               // x
EVE_Write16 (200);               // y
EVE_Write16 (25);                // font
EVE_Write16 (0);                 // options
EVE_Write8 (0x48);               // string H
EVE_Write8 (0x45);               // string E
EVE_Write8 (0x4C);               // string L
EVE_Write8 (0x4C);               // string L
EVE_Write8 (0x4F);               // string O
EVE_Write8 (0);                  // null
EVE_Write8 (0);                  // padding
EVE_Write8 (0);                  // padding
cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);
```

First, the x and y coordinates of the text string are specified. The font value follows which selects one of the built-in EVE fonts. An "Options" value follows allowing the text to be centred horizontally and vertically, effectively adjusting where the text sits relative to the specified coordinate. In this case, '0' results in the coordinate being the top-left of the text string.

The characters themselves now follow as ASCII byte values, in this case the word HELLO. The string is terminated by one further 0x00 byte.

One important point to note is that since the co-processor list is being written as a burst write and the FT8xx is keeping track of the count of bytes written, dummy bytes need to be written if the overall TEXT command including parameters is not a multiple of 4 bytes, so that the next command sent after TEXT will begin at a 4 byte boundary. Failure to do this will result in the next command being processed incorrectly and an error occurring. The total number of bytes shown in green below is 18 and so two additional dummy 00 bytes are sent as shown in red to take the total to 20. The MCU variable keeping track of the number of bytes is therefore also incremented by 20.

```
EVE_Write32 (CMD_TEXT);          // 4 bytes
EVE_Write16 (100);               // 2 bytes
EVE_Write16 (200);               // 2 bytes
EVE_Write16 (25);                // 2 bytes
EVE_Write16 (0);                 // 2 bytes
EVE_Write8 (0x48);               // 1 bytes
EVE_Write8 (0x45);               // 1 bytes
EVE_Write8 (0x4C);               // 1 bytes
EVE_Write8 (0x4C);               // 1 bytes
EVE_Write8 (0x4F);               // 1 bytes
EVE_Write8 (0);                  // 1 bytes
EVE_Write8 (0);                  // 1
EVE_Write8 (0);                  // 1
cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);
```

This could be used as the basis of a text function which can make use of the compiler's data types to take a string data type and convert it to the required series of EVE_Write transfers.

6 Examples - Loading Bitmaps and Custom Fonts

6.1 APP_ConvertedBitmap()

This example loads a small bitmap of Eve and displays it on the screen.

Figure 5 - Bitmap demo

The original PNG file can be found within the zip file package provided with this application note. The EVE Image Converter was used to create a raw data file in RGB565 format which was then loaded into the PIC's memory.

After downloading and un-zipping the image converter to a folder in C:\ the EVE.png file was copied to this folder. Then, the following command line was run to convert the file into format RGB565 which corresponds to 7 in the list of formats provided by the tool.

```
Img_cvt –I EVE.png –f 7
```

A new folder will appear in the Image Converter program folder which contains the converted data; in this case the folder is called EVE_RGB565. The resulting files are shown in the second screenshot below.

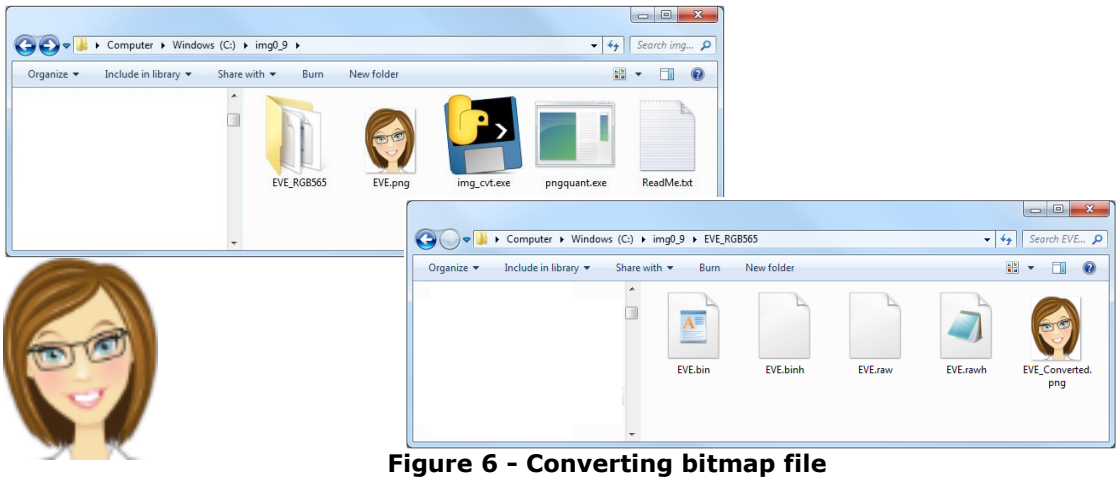


Figure 6 - Converting bitmap file

The text-readable file EVE.rawh was used in this case and the contents were copied to an array in a header file EVE_RGB565.h in the source code.

In MPLAB, the EVE_RGB565.h header file contains the following array:

```
const char rawData[8838] @ 0x200 = {/*('file properties: ', 'resolution ', 58, 'x',
75, 'format ', 'RGB565', 'stride ', 116, ' total size ', 8700)*/
255,255,255,255, [remainder of file],
}
```

The @ 0x200 specifies the memory address in the PIC and in this case the image data is loaded to the Flash memory of the PIC. Please check the documentation of the selected MCU for information on the memory map and data storage locations available. The output from the image conversion is also included in green text as a comment to remind the developer of the image properties.

Since this is a bitmap file in a format supported by the FT8xx graphics engine, the image data can be loaded directly to an empty area of RAM_G (in this case beginning at address 0) where the bitmap commands can subsequently be pointed to in order to retrieve and display the image.

A simple burst write of the 8-bit values of the image data can be used. Chip select goes low followed by specifying the address with the bits set to indicate a write transaction. A while loop then streams the bytes of the image data array until all 8700 bytes have been written. An additional if statement checks if the data size is a multiple of 4 bytes and if not pads the data with 0x00 bytes to make it a multiple of 4.

```
DataPointer = 0;

MCU_CSlow();
EVE_AddrForWr(RAM_G);

while(DataPointer < DataSize)
{
    EVE_Write8(rawData[DataPointer]);
    DataPointer ++;
}

BitmapDataSize = DataSize - DataPointer;
BitmapDataSize = BitmapDataSize & 0x03;  // Mask off the bottom 2 bits

if (BitmapDataSize == 0x03)
{
    EVE_Write8(0x00);  // 1 extra dummy byte
}
else if (BitmapDataSize == 0x02)
{
    EVE_Write8(0x00);  // 2 extra dummy bytes
    EVE_Write8(0x00);
}
else if (BitmapDataSize == 0x01)
{
    EVE_Write8(0x00);  // 3 extra dummy bytes
    EVE_Write8(0x00);
    EVE_Write8(0x00);
}

MCU_CShigh();
```

The demo then displays this image by beginning a new co-processor list in the usual way (refer to the Flashing Dot example in section 5.1).

The Bitmap Source specifies the starting point of the image data in RAM_G. A bitmap handle can be specified so that the bitmap can be referenced via its handle.

The Bitmap Layout then specifies the file format, line stride and height which instructs the FT8xx graphics engine how the data in the file should be interpreted with regards to the actual image. The Bitmap Size is also specified which tells the FT8xx how the image is to be displayed on-screen such as width, height and, filtering and wrapping.

Note that the values for width, height and stride required by these commands can be obtained from the comments at the start of the raw data file output by the converter.

```
EVE_Write32(BITMAP_HANDLE(0));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(BITMAP_SOURCE(0));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(BITMAP_LAYOUT(RGB565,116,75));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(BITMAP_SIZE(NEAREST, BORDER, BORDER, 58,75));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

The "Begin" command is then called, with the parameter set to begin drawing bitmaps. Subsequent Vertex commands will draw the image at the required location(s). The End command will end the drawing of bitmaps.

```
EVE_Write32(BEGIN(BITMAPS));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(VERTEX2F(0x100,0x100));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(END());
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

Note that the FT81x has BITMAP_LAYOUT_H and BITMAP_SIZE_H instructions which allow the upper bits of these values to be set. This is for backward compatibility as the FT81x can use higher values than were available on the FT80x due to the larger resolutions available.

The image data was loaded into MCU flash via the header file in this example to avoid reliance on additional libraries which may be specific to the MCU (e.g. an SD card interface). However, the data can be loaded from other sources if the chosen MCU platform has other storage media available such as external data flash memory, SD cards etc.

6.2 APP_DigitsFont

This example loads a custom font and then prints some text using the font.

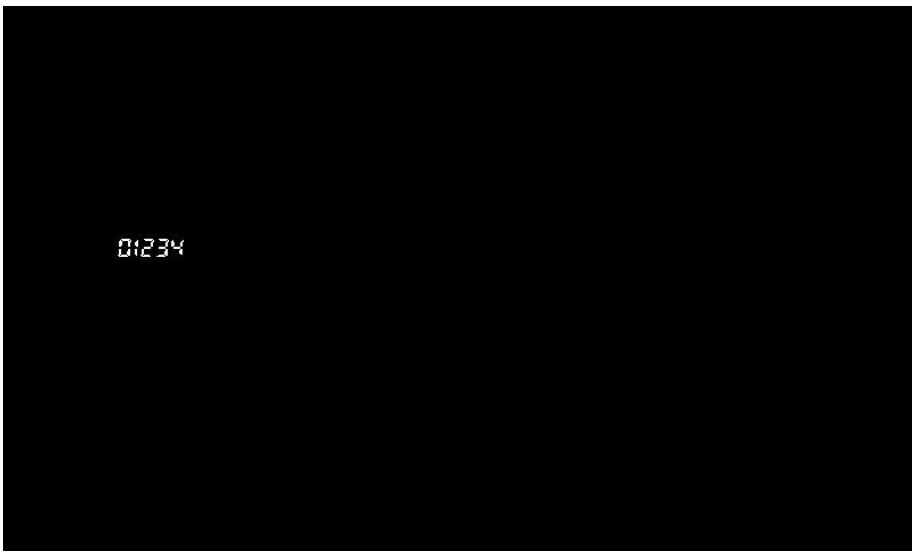


Figure 7 - Digits custom font demo

The font creation is based on the technique described in AN_277. The first step is to download the latest version of the font converter and extract it to a folder (for example a new folder within C:\)

The font file DS-DIGIT.ttf was obtained from the Windows font folder as described in AN_277 and was copied into the same folder as the fnt_cvt.exe file.

The font converter was then run to convert the files. In this case, all characters were converted and so the command line was as shown below. A size of 25 was selected and the data was set to be loaded at RAM_G address 1000. The output folders are also shown.

```
C:\Font_Convert>fnt_cvt.exe -i DS-DIGIT.TTF -s 25 -a -d 1000
converting ASCII coding file ansi_32_126.txt
Font Conversion Complete!
```

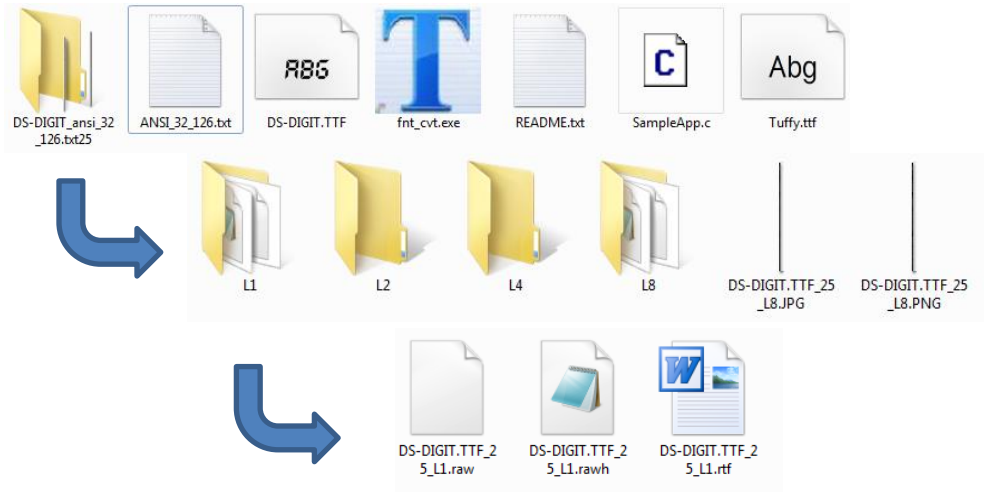


Figure 8 - Digits custom font folders

The entire contents of the DS-DIGIT.TTF_25_L1.rawh file were copied and pasted into a new .h file in the MPLABX and this was named DIGITfont.h. This file must be included in the project.

Two edits as circled in red below were made to turn the pasted data into an array suitable for MPLAB X. The first edit will result in the array being placed at address 0x3000 in the PIC. This may need to be adjusted depending on the memory map of the selected MCU. It is important to note that this address is not related to the "-1000" specified in the fnt_cvt.exe command line call. The "-1000" refers to the location in the FT81x's RAM_G where the data will be loaded in the next steps.

```
/*Command Line: fnt_cvt.exe -i DS-DIGIT.TTF -s 25 -a -d 1000*/

/*95 characters has been converted */

/* 148 Metric Block Begin +++  */
/*('file properties ', 'format ', 'L1', ' stride ', 3, ' width ', 18, 'height',
25)*/

const char MetricBlock[] @ 0x3000 = {

/* Widths */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,7,9,15,13,17,15,5
,9,8,13,12,5,12,5,14,13,7,13,13,13,13,13,13,13,13,6,6,9,12,9,13,16,13,13,13,13,13,1
3,13,13,7,13,13,13,13,13,13,13,13,13,13,14,13,13,13,13,13,13,9,12,9,12,13,7,13,13,1
3,13,13,13,13,13,7,13,13,13,13,13,13,13,13,13,13,14,13,13,13,13,13,13,11,8,10,11,0,
/* Format */
1,0,0,0,
/* Stride */
3,0,0,0,
/* Max Width */
18,0,0,0,
/* Max Height */
25,0,0,0,
/* Raw Data Address in Decimal: <-1252> */
28,251,255,255,
/* 148 Metric Block End ---  */

[remainder of file]

0,0,0,0,0,0,0,0,0,0,0,0,0,28,64,0,63,192,0,39,192,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
/*Bitmap Raw Data end ---*/
}
;
```

The first part of the APP_DigitFont() function will load the data byte-by-byte from the array into the RAM_G of the FT81x using an SPI burst write, beginning at address 1000.

A new screen is created in the usual way beginning with the DL_Start and Clear commands. The application must then tell the FT81x about the font data to be loaded. Here, font handle 14 was selected. The FT81x also needs to know how to interpret the block of data which has been written to it, including where to find the first printable character, which format the font is in, and how many bytes make up each line of each character. The code must also specify how the resulting bitmap of each character is to be displayed including its size.

Note that the values for width, height, stride and raw data address required by these commands can be obtained from the metric block area (see code above) of the raw data file output by the converter.

Finally, the SetFont command applies this font beginning at 1000 in RAM_G to font handle 14.

```
EVE_Write32(BITMAP_HANDLE(14));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4;

EVE_Write32(BITMAP_SOURCE(-1252));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(BITMAP_LAYOUT(L1,3,25));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(BITMAP_SIZE(NEAREST, BORDER, BORDER, 18,25));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(CMD_SETFONT);
EVE_Write32(14);
EVE_Write32 (1000);
cmdOffset = EVE_IncCMDOffset(cmdOffset, 12);
```

The font can now be used to produce a simple text string by setting the color desired and then using the Text command with the font set to 14.

```
EVE_Write32 (COLOR_RGB(255,255,255));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32 (CMD_TEXT);
EVE_Write16 (100);                    // x
EVE_Write16 (200);                    // y
EVE_Write16 (14);                     // font
EVE_Write16 (0);                      // options
EVE_Write8 (0x30);                    // string 0
EVE_Write8 (0x31);                    // string 1
EVE_Write8 (0x32);                    // string 2
EVE_Write8 (0x33);                    // string 3
EVE_Write8 (0x34);                    // string 4
EVE_Write8 (0);                       // null
EVE_Write8 (0);                       // padding
EVE_Write8 (0);                       // padding
cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);
```

The font data was loaded into MCU flash via the header file in this example to avoid reliance on additional libraries which may be specific to the MCU (e.g. an SD card interface). However, the data can be loaded from other sources if the chosen MCU platform has other storage media available such as external data flash memory, SD cards etc.

7 Examples - Touch Features

7.1 APP_Calibrate

The calibration function is provided to run the calibrate feature of the FT81x. In the demo code provided with this application note, the Calibrate function should be un-commented so that it runs just after the initialisation if any of the touch-enabled demos are used.

Calibration is necessary in any touch-enabled applications in order to ensure that the touch detection is aligned with the LCD panel underneath. This avoids the user experiencing any offset between where they touch the screen and where the application detects the touch.

The calibration feature of the FT8xx presents the user with three dots on the screen which they tap in turn, allowing it to calculate six touch transform values which relate the position of the touch to the real position on the screen. It would typically be run once during initialisation of the application on start-up and (assuming the alignment of touch panel vs screen did not change) will apply throughout the remainder of the application.

The calibration is run via the co-processor and will block until the user taps the three dots. At this point, when the read and write pointers of the co-processor indicate completion of the screen containing the calibration, the six transform values will have been loaded into their respective REG_TOUCH_TRANSFORM_A to REG_TOUCH_TRANSFORM_F registers and will be applying them to the touch inputs. Therefore, after completion of the routine the FT8xx can be considered calibrated.

The calibration values are held in volatile memory and so most applications will either:

- Run calibration once during each start-up of the application so that the transform values are calculated and populated in their registers
- Run calibration during factory test of the product and store the six transform values in the MCU's non-volatile memory. The values can then be written back by the MCU on subsequent power-ups instead of running the calibration.

In the second case, the values are only valid after completion of a successful calibration and can be read via 32-bit reads of the REG_TOUCH_TRANSFORM_A to _F. When powering up the application the next time, the values can be written back during initialisation using six 32-bit register writes to REG_TOUCH_TRANSFORM_A to _F. It is recommended that a re-calibrate option is added (via a menu system or restricted service menu) in case the LCD panel had to be replaced in the field etc.

The function itself uses the normal method of creating a screen via a co-processor list. The screen is cleared and the CALIBRATE command is run. Note that as per some of the EVE sample applications, it is possible to display a text string via a standard text command before the calibrate command to ask the user to tap the dots.

```
cmdOffset = EVE_WaitCmdFifoEmpty();

MCU_CSlow();
EVE_AddrForWr(RAM_CMD + cmdOffset);

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);

EVE_Write32(CMD_CALIBRATE);
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

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

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

MCU_CShigh();

EVE_MemWrite32(REG_CMD_WRITE, (cmdOffset));

cmdOffset = EVE_WaitCmdFifoEmpty();
```

Once the REG_CMD_WRITE is updated in the second-last line above, the co-processor will begin to work through the set of commands. The calibrate command is a blocking call and so the code will not return from the EVE_WaitCmdFifoEmpty call until the user has tapped the three dots.

7.2 APP_SliderandButton()

This demo gives a basic illustration of using the touch tag and tracker features. It displays a button and a slider on the screen. If the user touches the button, a red dot will be illuminated above the button. If the user touches the slider, they can drag the handle up and down due to the tracker applied to the slider.

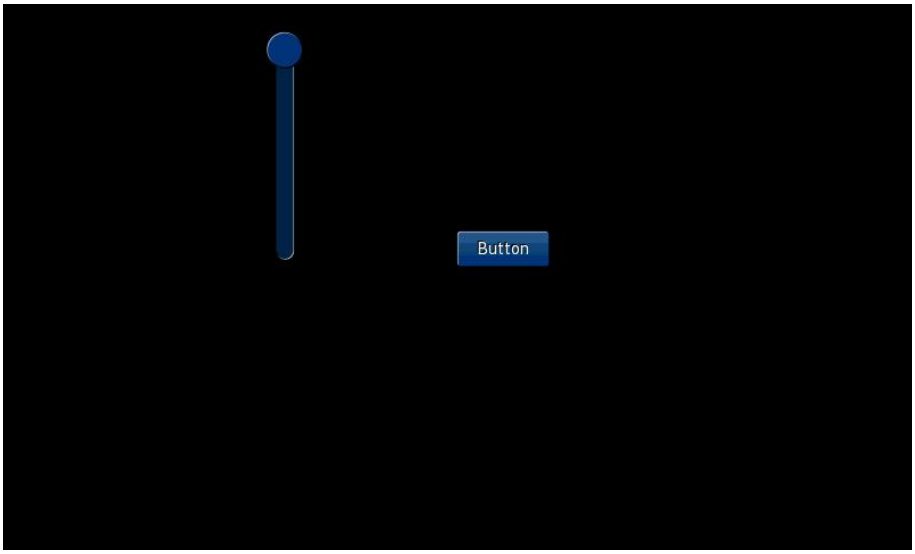


Figure 9 - Slider and Button demo

The FT8xx offers tagging and tracking features to help make development of touch control applications easier. In particular, they avoid the need for an application to read raw touch coordinates and then work out whether these coordinates lie within the area of an item on the screen (e.g. if touching within a button)

Tagging

Tagging is used to detect when an item is touched, for example a button widget or a bitmap of a custom button etc. When drawing objects on the screen, a tag command is used to set the tag number which will be applied to any subsequent objects. This can be any number from 1 to 255 and the numbers in this demo were chosen arbitrarily.

The EVE devices have a Tag() command, which specifies the tag number to be given to subsequent items in the list, and a Tag_Mask() command, which acts as an enable for that tag to be applied to those items. As a general rule, when adding an item to the co-processor list, if tagging is currently enabled, the last tag specified with the Tag() command will be applied to that item.

The following example gives a very simple illustration of how these commands can be used to apply tags to only the desired items. It uses buttons and points but can also be applied in the same way to widgets and bitmaps etc.

```
Tag_Mask(1)   // Enable Tagging

Tag(4)        // Assign Tag 4 to following items

Button_A      // Button A is given Tag 4
Point_A       // Point A is given Tag 4
Button_B      // Button B is given Tag 4

Tag(8)        // Assign Tag 8 to following items

Point_B       // Point B is given Tag 8
Button_C      // Button C is given Tag 8

Tag_Mask(0)   // Disable Tagging

Button_D      // Button D is not tagged as tagging is masked
Point_C       // Point C is not tagged as tagging is masked

Tag_Mask(1)   // Enable Tagging

Button_E      // Button E is given Tag 8 since it was last tag set above

Tag(20)       // Assign Tag 20 to following items

Point_D       // Point D is given Tag 20


With this code, reading REG_TOUCH_TAG will give a result of 0 when touching any
blank areas of the screen and will give the following values when touching the items
detailed above:
Touch: Button_A or Point_A or Button_B         REG_TOUCH_TAG = 4
Touch: Point_B or Button_C                     REG_TOUCH_TAG = 8
Touch: Button_D or Point_C                     REG_TOUCH_TAG = 0
Touch: Button_E                                REG_TOUCH_TAG = 8
Touch: Point_D                                 REG_TOUCH_TAG = 20
```

Figure 10 - Tag() and Tag_Mask() illustration

In the Slider and Button example, the tag is set to 2 and a single button is drawn. The TAG_MASK(1) before the button ensures that tagging is enabled and therefore the tag 2 will be

applied to the button. The button uses variable Button3D which will be 256 for a 3D effect (button not pressed) and 0 for a flat effect (as if the button is pressed).

```
EVE_Write32(TAG_MASK(1));              // Enable tagging
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32(TAG(2));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32 (CMD_BUTTON);              // command button 0xFFFFFF0D
EVE_Write16 (400);                     // x
EVE_Write16 (200);                     // y
EVE_Write16 (80);                      // w
EVE_Write16 (30);                      // h
EVE_Write16 (26);                      // font
EVE_Write16 (Button3D);                // options
EVE_Write8 (0x42);                     // string B
EVE_Write8 (0x75);                     // string u
EVE_Write8 (0x74);                     // string t
EVE_Write8 (0x74);                     // string t
EVE_Write8 (0x6F);                     // string o
EVE_Write8 (0x6E);                     // string n
EVE_Write8 (0);                        // null terminates string
EVE_Write8 (0);                        // pad with extra zero
cmdOffset = EVE_IncCMDOffset(cmdOffset, 24);
```

Tracking

The code keeps tagging enabled after drawing the button but now sets a different tag number for the slider. Whereas tagging allows the application to determine if an object is touched, tracking allows the position within a defined range to be detected. Like tagging, the tracking feature takes a lot of the work away from the MCU in creating sliding and rotary controls.

The slider is drawn with a range of values 0x00 to 0xFF and a variable SlideVal for the field which defines the handle position.

A tracker is then defined with the area set to the same size as the slider in this case. The tracker will report a value in the range 0 to 65535 depending on the position of the users touch relative to the tracked area. The tracking is carried out in relation to the longest side of the tracked area and so the area defined here results in vertical tracking.

```
EVE_Write32(TAG(5));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);

EVE_Write32 (CMD_SLIDER);              // command slider 0xFFFFFF10
EVE_Write16 (0x00F0);                  // x
EVE_Write16 (0x0028);                  // y
EVE_Write16 (0x000F);                  // width
EVE_Write16 (0x00B0);                  // height
EVE_Write16 (0x0000);                  // options
EVE_Write16 (SlideVal);                // value (i.e. position of handle)
EVE_Write16 (0x00FF);                  // range of slider
EVE_Write16 (0x0000);                  // dummy
cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);

EVE_Write32 (CMD_TRACK);               // command track 0xFFFFFF2C
EVE_Write16 (0x00F0);                  // x
```

```
EVE_Write16 (0x0028);                    // y
EVE_Write16 (0x000F);                    // width
EVE_Write16 (0x00B0);                    // height
EVE_Write16 (0x0005);                    // tag the tracked area with 5
EVE_Write16 (0x0000);                    // dummy
cmdOffset = EVE_IncCMDOffset(cmdOffset, 16
```

Tagging is masked after the tracker has been set so that any subsequent items (if present) would not be tagged.

```
EVE_Write32(TAG_MASK(0));
cmdOffset = EVE_IncCMDOffset(cmdOffset, 4);
```

Checking the values

Once the objects have been drawn on the screen with their tags and tracking set, the next step is to check if any tagged or tracked areas were touched. The register REG_TOUCH_TAG can be checked to determine if a touch is taking place (if value non-zero) and will report the tag value. The application can check this register to see if the value matches the tag of the button or slider.

If the value matches the button, a variable is set which will cause the point drawn to be coloured red the next time round the while() loop when the screen is updated. A second variable selects either 3D effect or Flat effect for the button to indicate de-pressed or pressed respectively.

If the slider is touched, a read of REG_TRACKER allows the tracking value to be determined. The upper 16 bits contain the value but since the slider was set for a 0-255 range then only the upper 8 bits are taken by shifting down 24 places. The resulting 8 bit value is stored as SliderVal and is used when drawing the slider the next time around the while() loop. In addition to updating the slider handle position to provide feedback for the user, this value could be passed to the PWM of the PIC for example to control the intensity of an LED or used for another purpose within the application.

```
TagVal = EVE_MemRead8(REG_TOUCH_TAG);
TrackerVal = EVE_MemRead32(REG_TRACKER);

if(TagVal == 2)
{
      color = 0xFF;
      Button3D = 256;
}
else
{
      color = 0x00;
      Button3D = 0;
}

if(TagVal == 5)
{
      SlideVal = (TrackerVal >> 24);
}
```

8 Examples - Snapshot

8.1 APP_SnapShot2

This function is used to take Snapshots of the screen via the Snapshot2 command and was used to take the screenshots throughout this document. The data is transferred via UART, through an FTDI USB-TTL cable, to a PC where it can be converted and viewed and used in documents.

Each of the other demos have a call to APP_SnapShot2() after they have completed rendering of the screen. This call is initially commented out but can be un-commented to take a snapshot.

```
unsigned long Pointer = 10000;          // Where to start image in RAM_G
unsigned long SerialCounter = 0;
unsigned int ARGB = 0;
unsigned char SerByteAR = 0;
unsigned char SerByteGB = 0;
unsigned char StartCode = 0;
```

The code then waits for the command FIFO to be empty and then writes the SnapShot2 command to the co-processor buffer. The Snapshot2 command allows selection of the image format and in this case ARGB4 is chosen. The pointer specifies where in RAM_G the image data will be written.

The next parameters specify the top-left coordinate and the x and y dimensions. This allows the code to select a window on the screen to be taken as a snapshot instead of the entire screen if preferred. For example, only the chart area of a display may be required to be copied.

```
cmdOffset = EVE_WaitCmdFifoEmpty();        // Await co-processor buffer empty

MCU_CSlow();                                // Begin transaction
EVE_AddrForWr(RAM_CMD + cmdOffset);         // Starting address in FIFO
EVE_Write32 (CMD_SNAPSHOT2);
EVE_Write32 (6);                           // format argb4
EVE_Write32 (Pointer);                     // Where image data will be stored
EVE_Write16 (0);                            // x coord top-left
EVE_Write16 (0);                            // y coord top-left
EVE_Write16 (800);                          // x size
EVE_Write16 (480);                          // y size

cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);

MCU_CShigh();

EVE_MemWrite32(REG_CMD_WRITE, (cmdOffset)); // execute command

// the snapshot will now be taken

cmdOffset = EVE_WaitCmdFifoEmpty();

// 2 * 800 * 480 = 768000
```

ARGB4 format uses four bits for Alpha, Red, Green and Blue and so requires two bytes for each pixel. The resulting data size will therefore be 2 x 800 x 480 in this case which is 768000 bytes.

It is vital when selecting a format and screen area to snapshot that the size of the image is considered. The developer must consider the total size of RAM_G (1Mbytes) as well as the starting point selected for the image data and any other data being stored in RAM_G by the application.

If starting at offset RAM_G + 0 and using a full 800x480 area with ARGB4 format, the RAM_G from 0 to 768000 will be used and overwritten. A snapshot should *not* overwrite content in RAM_G which is displayed on the present screen.

At this point, the snapshot data will be in RAM_G at the selected location. The demo now waits for a byte with value 0x53 (ASCII for capital S) from the UART which is sent by the application which

will receive the data. This tells it that the PC application is ready. Awaiting a specific character also avoids spurious transitions on the UART line during power-up from triggering the data to begin.

```
StartCode = 0x00;

while(StartCode != 0x53)
{
    StartCode = MCU_UART_Rx();
}
```

The code now performs a 16-bit burst read of 768000 bytes from the selected starting RAM_G address (10000) onwards, and arranges each pair of bytes so that the resulting format sent over UART will be ARGB. It then sends each byte to the UART. It is also possible to increase the baud rate to make the transfer faster. This demo uses 57600 baud as it is a common rate available on most terminals but if using an FTDI USB-TTL cable, other rates are achievable. For example 250Kbaud is possible with the same crystal and PLL settings used in this application note, and are supported by the FTDI chipset. The use of RTS/CTS flow control is strongly recommended to ensure that no data is lost.

```
SerialCounter = 0;
while(SerialCounter != 768000)
{
    ARGB = EVE_MemRead16(SerialCounter + Pointer);

    SerByteAR = (unsigned char)(ARGB >> 8);
    SerByteGB = (unsigned char)(ARGB);

    MCU_UART_Tx(SerByteAR);
    MCU_UART_Tx(SerByteGB);

    SerialCounter = SerialCounter + 2; // increment address to read by 2
}
while(1)
{
}
```

Refer to the following section (section 8.2) for details of how to receive this data on the PC.

8.2 APP_SnapShot2 - Uploading Using Terminal

The section above discusses the code running on the PIC itself which will send the image data over UART. On the PC, an application is needed which can receive and store these bytes and eventually convert them to a format such as JPG which is supported by word processor programs etc.

This section describes how to use a terminal program to receive the data. It requires the following:

A Windows PC running Windows 7 up to Windows 10

Terminal program capable of storing received data to a file. The MTTTY_D2XX program is provided for example in the supplied zip file. It should be copied to a convenient place such as the desktop.

FTDI USB-TTL cable such as C232HD or TTL-232R-3v3 connected to a USB port on the PC and to the PIC MCU as shown in the hardware section of BRT_AN_006. Before connecting to the PC, download the latest FTDI driver (if not already installed) to the PCs desktop from FTDI Drivers. Right-click on the .exe and choose run as administrator. Follow the steps of the install wizard until finished and connect the cable. Verify that the cable now shows under Universal Serial Bus Controllers in the device manager. It should also appear

under Ports where the COM port number can be determined if using a com port based terminal.

Note: FTDI/BridgeTek cannot accept responsibility for, or provide technical support for, 3rd party software. The user must determine suitability of the software and must comply with any licensing requirements.

For example, enable the Flashing dot demo in the supplied code and un-comment the call to APP_SnapShot2 within the APP_FlashingDot demo code function.

Allow the PIC to run its application and to reach the point where it has rendered the screen and entered the APP_Snapshot2 function. It will now be waiting for a start character.

Open the terminal program such as MTTTY_D2XX

Select the FTDI cable under the Port menu (listed by FTDI serial number in MTTTY_D2XX)

Select baud rate as 57600

Select File -> Connect

Select Transfer -> Receive File (text)

In the file dialog, specify a name and location. For example, select the desktop and name the file "MySnapshot".

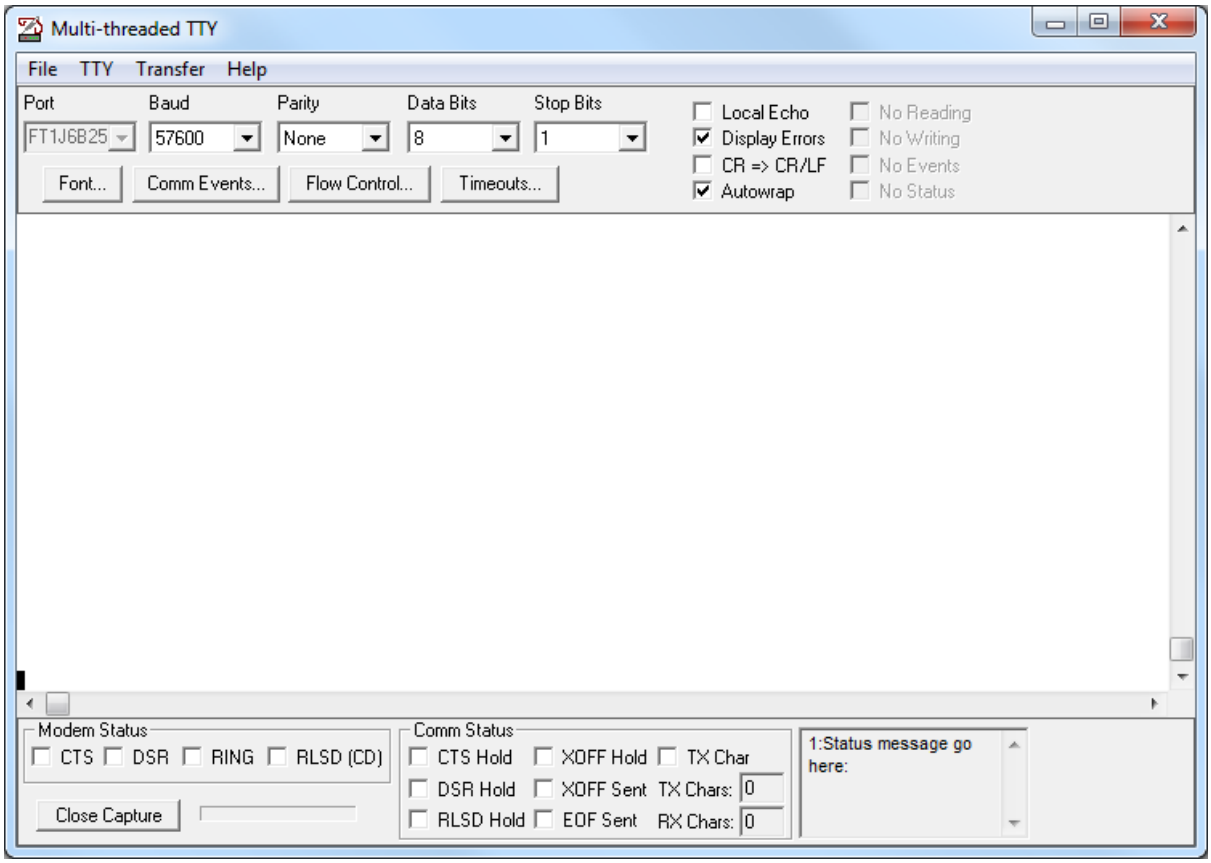


Figure 11 - MTTTY_D2XX Terminal

Type a capital S which will be sent to the UART to tell the PIC to start the transfer

The bar graph beside the 'Close Capture' button will show activity as the data is received. One advantage of cable such as the C232HD is that the traffic indication LEDs will also provide an indication of the transfer. A brief flash of the Tx LED confirms the sending of the S character and a prolonged blinking of the Rx LED will indicate the data coming back.

Leave running until the activity bar remains static (note that it does not always go back to 0) and (if fitted) the Rx LED on the cable stops blinking.

Click 'Close Capture' to close the file.

Find the file in the location specified when creating the transfer and re-name to .argb. For example, "MySnapshot.argb"

Note that the supplied code will now wait in an infinite while() loop after the data has all been sent but could return to the normal application if preferred.

Refer to section 8.3 for details of how to convert the image to a viewable format.

8.3 APP_SnapShot2 – Converting the image

Once the data has been uploaded and saved as an ARGB file, it may require conversion if a program is not available which can open this file directly.

This section requires Image Conversion software which can convert ARGB images to other formats. The software used here is ImageMagick which can be found at the following link: https://www.imagemagick.org/script/index.php. This should be installed as per the instructions provided on the downloads page.

It is often convenient at this stage to make a folder in C drive called Snapshots and copy the .argb file created in the section above into this folder. This allows the paths on the command line to be kept short. It is worth checking that the file is of the expected size. In the 800x480 ARGB4 format used here, the resulting MySnapshot.argb file size shows as 750KB in Windows.

Now open a command window by searching for CMD.

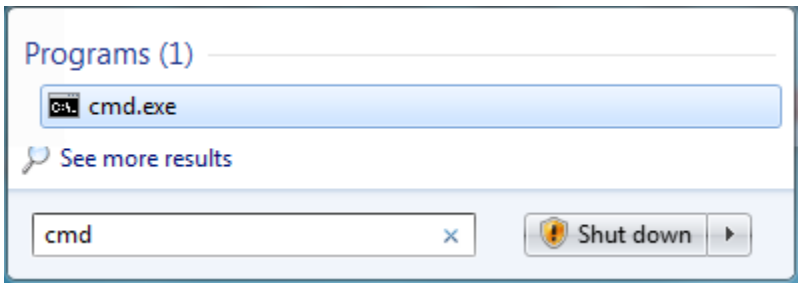


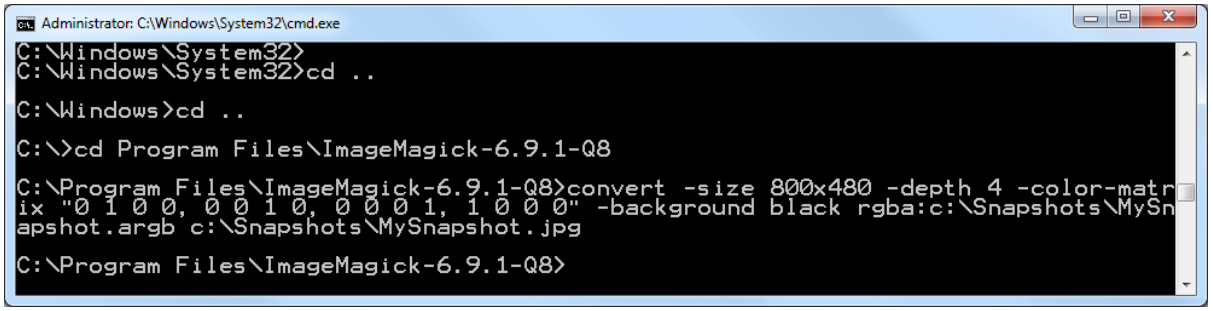
Figure 12 - Opening CMD

Note: On some systems it may be necessary to right-click on the cmd.exe entry above and run as administrator instead of just left-clicking. If permissions issues are encountered in the following stages below, it may be that the terminal does not have sufficient privileges by default to access files in the c:\ drive.

Now browse to the folder containing the ImageMagick conversion tools as shown below. The command line shown is used to carry out the conversion.

The parameters include a color matrix which specifies how the alpha, red, green and blue components of the source file are mapped. The data from the PIC has been stored in the format AAAARRRR GGGGBBBB whereas the conversion uses format RRRRGGGGBBBBAAAA.

It also specifies the input and output files. The paths should be changed from C:\Snapshots if the source ARGB file is in a different location. The screenshot of the entire conversion, a text version of the commands, and an image of the input and output files are provided below. Note that whilst the input (.argb) file size will be the same for an 800x480 image in ARGB4 format, the output JPG file may differ in size compared to the one in the screenshot.




```
C:\windows\system32>cd ..

C:\Windows>cd ..

C:\>cd Program Files\ImageMagick-6.9.1-Q8

C:\Program Files\ImageMagick-6.9.1-Q8>convert -size 800x480 -depth 4 -color-matr
ix "0 1 0 0, 0 0 1 0, 0 0 0 1, 1 0 0 0" -background black rgba:c:\Snapshots\MySn
apshot.argb c:\Snapshots\MySnapshot.jpg

C:\Program Files\ImageMagick-6.9.1-Q8>
```

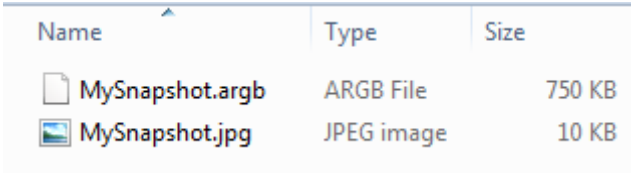


Figure 13 - Running ImageMagick

The MySnapshot.jpg file can now be copied and used in documentation such as the user guide for the product.

8.4 Snapshot Uploader – VB-Net application

The sample package supplied with this document also includes a very simple program written in VB.NET which performs the same function of uploading and saving the data as shown in section 8.2. Note that the program as supplied will only replace the uploading step and does not perform image conversion.

First, the program FT81x SnapShot2 Reader.exe can be copied onto a convenient place such as the PCs desktop.

The steps in section 8.1 should be followed now in order to set the PIC up for taking the snapshot.

The Uploader program can then be run by double-clicking the exe. Once open, click the Initialise button and this will check for connected FTDI cables and open the first port found. It is recommended that the USB-UART cable is the only FTDI device connected at this time to ensure the correct one is opened although the program could be extended to open by description etc.

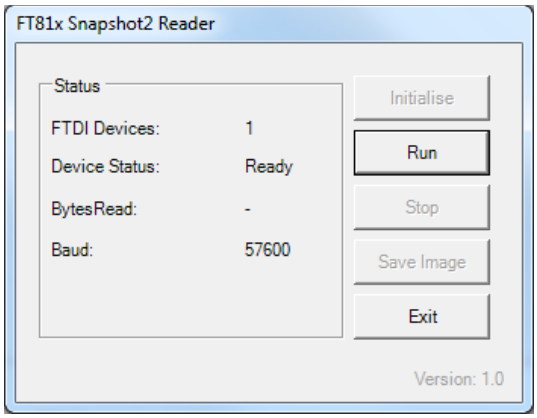


Figure 14 - Application after the attached cable is initialised

Now, click the Run button. This will send a byte "S" to the UART which will in turn trigger the PIC to send the image data.

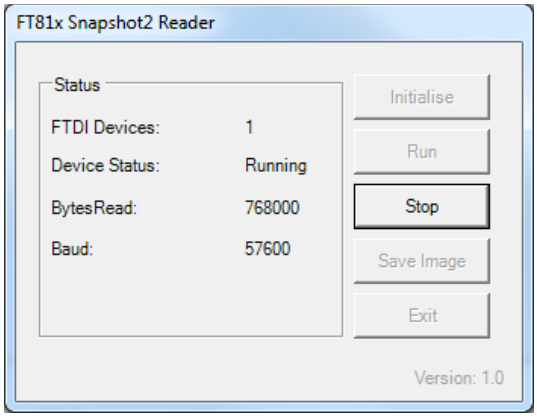


Figure 15 - Application running and collecting data

The Bytes Read counter will update as the bytes are received and can be compared to the total number expected. As discussed in section 8.1 it is expected to receive 768000 bytes in this case.

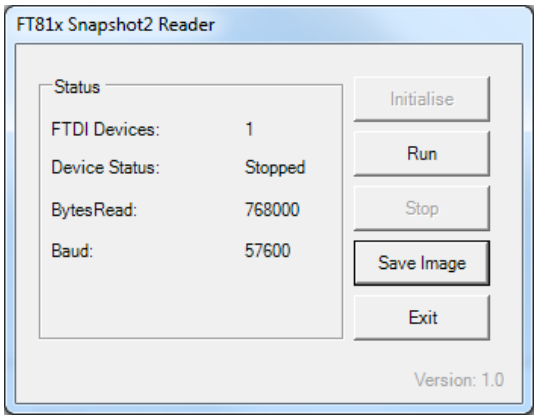


Figure 16 - Application being stopped after data collection

Once all bytes are received, the counter will stop incrementing and the Stop button can be pressed. The Save Image button opens the Save As dialog allowing the location and name to be specified. The program automatically puts the .argb extension onto the file.

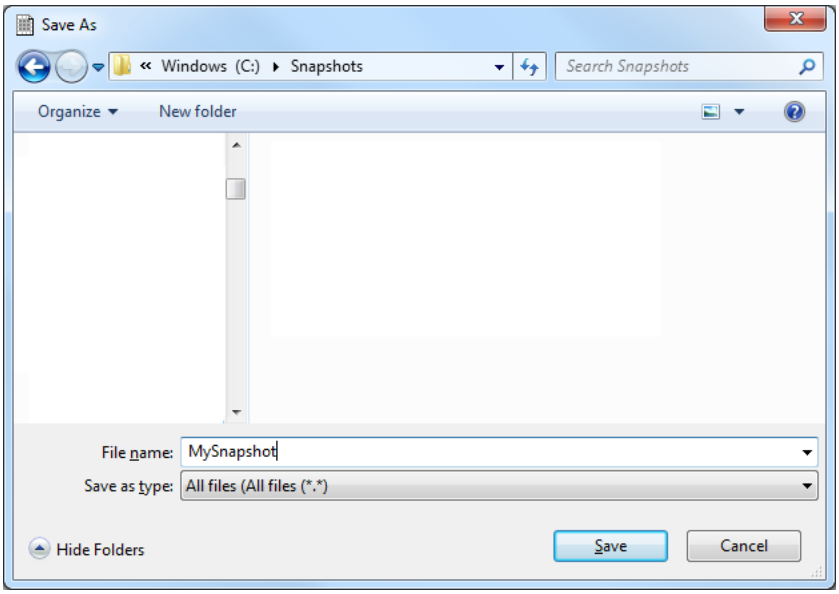


Figure 17 - Saving the .argb file

The file can be processed as shown in section 8.3 to produce a JPG image.

Note: This application is provided as a basic example for further development and does not perform comprehensive error checking and handling. It is not intended to be a fully robust end-user application. FTDI and Bridgetek accept no responsibility for any consequences resulting from the use of this application. The user must determine suitability for their intended application.

The developer may wish to enhance the program via the supplied source code or may wish to perform a similar operation via scripting in other languages such as PERL. In some cases it may be possible to automate the conversion by calling Image Magick via the same script.

9 Software Library layers

This section lists the functions in the EVE and MCU libraries used. Further details on them can be found in BRT_AN_006.

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 that best practices are followed to so that reliable operation of the final product can be assured.

9.1 EVE Layer

The EVE functions provide the interface between the Application layer and the MCU layer, allowing the application to use a simpler syntax. At the same time, the EVE functions are independent of MCU type as they use the MCU layer for all MCU-specific accesses.

Separate Address and Data functions

void EVE_AddrForWr (unsigned long ftAddress)

void EVE_AddrForRd(unsigned long ftAddress)

void EVE_Write32(unsigned long ftData32)

void EVE_Write16(unsigned int ftData16)

void EVE_Write8(unsigned char ftData8)

unsigned long EVE_Read32(void)

unsigned int EVE_Read16(void)

unsigned char EVE_Read8(void)

Combined Address and Data functions

void EVE_MemWrite32(unsigned long ftAddress, unsigned long ftData32)

void EVE_MemWrite16(unsigned long ftAddress, unsigned int ftData16)

void EVE_MemWrite8(unsigned long ftAddress, unsigned char ftData8)

unsigned long EVE_MemRead32(unsigned long ftAddress)

unsigned int EVE_MemRead16(unsigned long ftAddress)

unsigned char EVE_MemRead8(unsigned long ftAddress)

Host Command

void EVE_CmdWrite(unsigned char EVECmd, unsigned char Param)

Co-Processor FIFO Supporting Functions

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

unsigned int EVE_WaitCmdFifoEmpty(void)

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.

9.2 MCU Layer

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.

Initialisation

void MCU_Init(void)

GPIO Functions

void MCU_CSlow(void)

void MCU_CShigh(void)

void MCU_PDlow(void)

void MCU_PDhigh(void)

SPI Functions

unsigned char MCU_SPIReadWrite(unsigned char DataToWrite)

UART Functions

void UART_Init(void)

void UART_Tx(unsigned char SerialTxByte)

unsigned char UART_Rx(void)

Delay Functions

void Delay_20ms(void)

void Delay_500ms(void)

9.3 Additional Layers

It may also be beneficial to add another layer between the Application and EVE layers which takes a single function call with parameters and maps them into a series of SPI transfers. This can include adding the offset for the command FIFO.

The EVE sample apps available at EVE Samples (for FT900 and Visual Studio platforms) use this technique and provide these calls within the FT_CoPro_Cmds.c file. For example, the code below from FT_CoPro_Cmds.c implements the slider allowing it to be used in the main application with the following single line of code.

```
Ft_Gpu_CoCmd_Slider(phost, 20, 20, 10, 100, 0, 10, 100);
```

```
ft_void_t Ft_Gpu_CoCmd_Slider(Ft_Gpu_Hal_Context_t *phost,ft_int16_t x, ft_int16_t
y, ft_int16_t w, ft_int16_t h, ft_uint16_t options, ft_uint16_t val, ft_uint16_t
range)
{
  Ft_Gpu_CoCmd_StartFunc(phost,FT_CMD_SIZE*5);
  Ft_Gpu_Copro_SendCmd(phost, CMD_SLIDER);
  Ft_Gpu_Copro_SendCmd(phost, (((ft_uint32_t)y<<16)|(x & 0xffff)));
  Ft_Gpu_Copro_SendCmd(phost, (((ft_uint32_t)h<<16)|(w&0xffff)));
  Ft_Gpu_Copro_SendCmd(phost, (((ft_uint32_t)val<<16)|(options&0xffff)));
  Ft_Gpu_Copro_SendCmd(phost, range);
  Ft_Gpu_CoCmd_EndFunc(phost,(FT_CMD_SIZE*5));
}
```

A similar layer could be created in this PIC application, for example as shown below. This could be utilised along with the other functions mentioned in BRT_AN_006 section **'Command Implementation'** to create a wrapper which accepts commands in a syntax exported from other tools such as EVE Screen Editor.

```
void Ft_Gpu_CoCmd_Slider(unsigned int x, unsigned int y, unsigned int w, unsigned
int h, unsigned int options, unsigned int val, unsigned int range)
{
  EVE_Write32 (CMD_SLIDER);
  EVE_Write32 (((ft_uint32_t)y<<16)|(x & 0xffff));
  EVE_Write32 (((ft_uint32_t)h<<16)|(w&0xffff));
  EVE_Write32 (((unsigned long _t)val<<16)|(options&0xffff));
  EVE_Write32 ((range));
  cmdOffset = EVE_IncCMDOffset(cmdOffset, 20);
}
```

10 Using the Demo Application

The provided zip file contains the full MPLABx project.

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

Un-zip the provided BRT_AN_007_Source.zip and browse to the MPLAB folder. This can be 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_007_Source. The project should now appear in the Projects window.

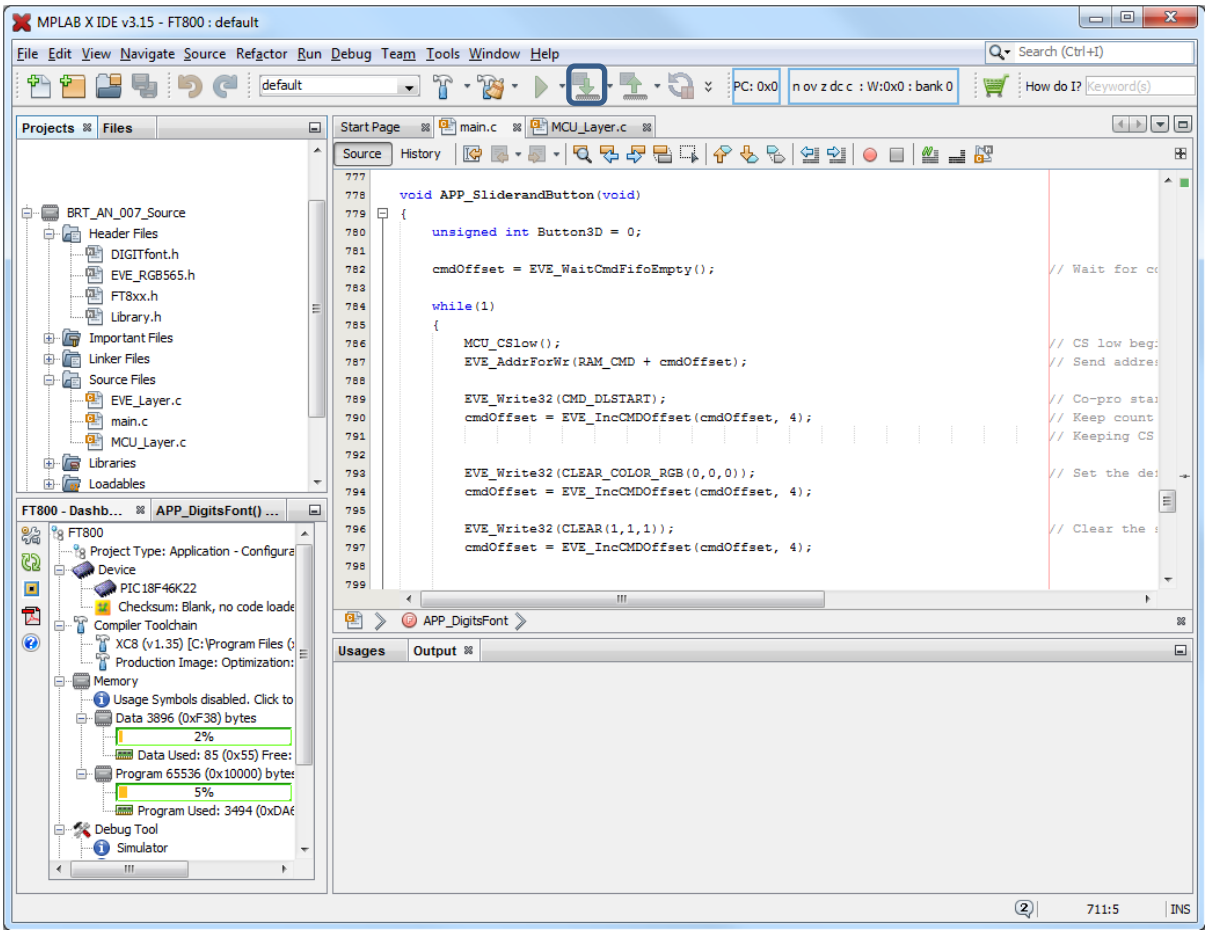


Figure 18 - MPLABX screenshot

Locate the void main(void) function near the top of the main.c file and select a demo by un-commenting the relevant line in the main()

```
void main(void)
{
    MCU_Init();
    APP_Init();
    MCU_UART_Init();
    //APP_Calibrate();                                          // NOTE:  Enable if using any touch demos

    // Important Note: Enable only one demo below at a time.
    // If running any of the following demos, un-comment the APP_Calibrate function above
    //     * APP_SliderAndButton

    APP_FlashingDot();

    //APP_VertexTranslate();

    //APP_Text();

    //APP_ConvertedBitmap();

    //APP_DigitsFont();

    //APP_SliderandButton();                                    // APP_Calibrate() must be un-commented

    while(1)
    {
        // Catch if it ever reaches here
    }
}
```

Figure 19 - Selecting the demo

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 section 5.1.

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

11 Conclusion

This application note has presented a series of simple examples of using the features of the EVE devices when controlling from a PIC microcontroller. It has illustrated the way in which the framework provided in BRT_AN_006 can be extended to use many of the features of the device.

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 use the same principles to create a final application with greater functionality

12 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

PIC18F46K22

PICKIT3

BRT_AN_006

Acronyms and Abbreviations

Terms	Description
EVE	Embedded Video Engine
MCU	Microcontroller
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_007 FT81x Simple PIC Example Demo Functions
Document Reference No.: BRT_000083
Clearance No.: BRT#080
Product Page: http://brtchip.com/i-ft8/
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial release	2017-03-24