Application Note

BRT_AN_014

FT81X Simple PIC Library Examples

Version 1.0

Issue Date: 2018-06-11

This application note provides a set of simple examples for the FT81X using the PIC MCU library functions which were introduced in BRT_AN_008.

Use of Bridgetek devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold Bridgetek harmless from any and all damages, claims, suits or expense resulting from such use.

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

Table of Contents

Copyright © Bridgetek Pte Ltd

Copyright © Bridgetek Pte Ltd

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. The examples shown in this application note uses the library which was created in BRT_AN_008.

The application notes in this series so far cover the following topics:

- BRT_AN_006 Overview of the low level SPI transfers used to interface to EVE
- BRT_AN_007 Examples illustrating the use of the transfers described in BRT_AN_006
- BRT_AN_008 Creating a simple EVE library using the principles explained above
- BRT_AN_014 Examples showing the use of the library described in BRT_AN_008

BRT_AN_006 and BRT_AN_007 are provided as background information to the creation of the library in BRT_AN_008.

1.2 Scope

The scope of this application note is to illustrate how the library functions in BRT_AN_008 can be used to build up a full application. It presents a series of small examples illustrating common techniques which are used in making a larger application. Each demo is intentionally very basic and focuses only on one topic to aid readability but these techniques can be used together to form a comprehensive real-world application. Likewise, the demo code covers the use of only a subset of the commands and features in the programmers guide but the demos have been chosen to demonstrate key principles which also apply when using many of the other commands. This application note focuses on the main application itself and assumes familiarity with the lower level API, EVE and MCU layers which were covered in BRT_AN_008 and explained in the earlier application notes in the series.

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. However, note that the FT81X has some new commands and features that are 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. The code provided with BRT_AN_008 shows the slightly different sequence of host commands and GPIO writes when starting up the FT80X series as well as having defines included for both FT80X and FT81X in the header files.

This application is written for the PIC family of MCUs (in this example the PIC18F46K22 is used) 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 library layers from BRT_AN_008 and is ready to use without any need to import them. These layers may have additional functions added for the purposes of supporting the demos provided here.

Note: This code is intended to act as a starting point for developers to create their own application rather than being a complete library package. It is necessary that the developers of the final application incorporating this library review all layers of the code as part of their product validation. By using any part of this code, the customer agrees to accept full responsibility for ensuring that their final product operates correctly and complies with any safety requirements, and accepts full responsibility for any consequences resulting from its use.

Copyright © Bridgetek Pte Ltd

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.

This application note focuses on the Application Layer as the other layers were covered in BRT_AN_008.

2.2 Software Layers

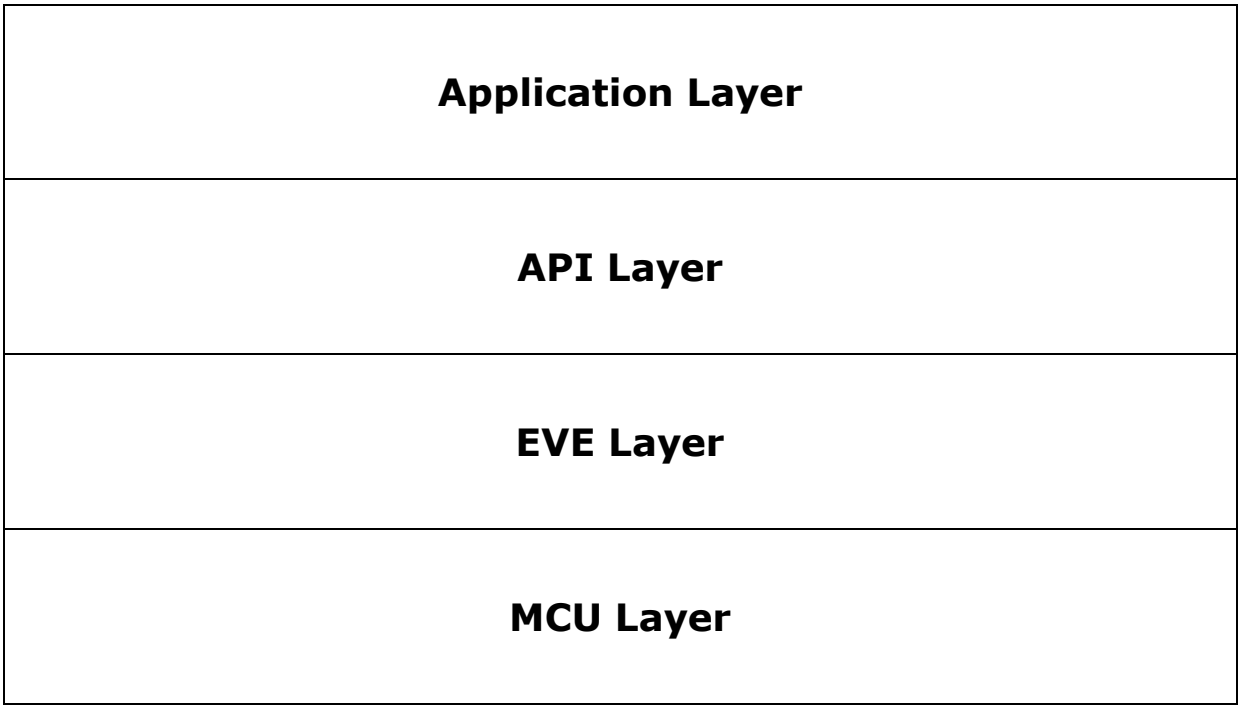


Figure 1 - Layers of the software example

2.3 MCU Memory Allocation

Table 1 summarises the addresses in the PIC MCU where the image and font data are loaded by the addresses defined in the header files. The application may also wish to keep a similar table for the RAM_G usage.

Asset	Start Address	Size	End Address	Comments
EVE_RGB565	0x200	8700 bytes	0x23FC	
EVE_PNG	0x3200	6285 bytes	0x4A8D	
LeaveItToEVE	0x5200	2272 bytes	0x5AE0	
DIGITfont	0x8200	7273 bytes	0x9E69	Size = 148 (metric block) + 7125 (data)

Table 1 – MCU memory

Copyright © Bridgetek Pte Ltd

2.4 Folder Structure

The project provided contains the following files:

Source Files

Main.c	Contains the Application layer
API_Layer.c	Contains API layer
EVE_Layer.c	Contains EVE layer
MCU_Layer.c	Contains MCU 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 sections for FT80X and FT81X and so FT_81X_ENABLE must be defined for this demo.
EVE_RGB565.h	Image data for the EVE bitmap demo
DIGITfont.h	Font data for the DS_DIGITS custom font demo
LeaveItToEVE.h	Compressed image data for Inflate demo
EVE_PNG.h	PNG image data for the LoadImagePNG demo

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

Supporting Files

APP_ConvertedBitmap	Contains original image and converted files for APP_ConvertedBitmap. The original image is also used in the PNG demo.
APP_DigitsFont	Contains original font and converted files used in APP_DigitsFont
APP_SnapShot2	Contains MTTTY_D2XX program and source code for Visual Basic NET uploader program. This is used in APP_SnapShot2PPM
APP_InflateImage	Contains original image and converted files for APP_InflateImage

Note: The library 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 the best practices are followed to, so that reliable operation of the final product can be assured.

If the information provided in this application note and accompanying code differs from the datasheet of the FT8XX, MCU or any other associated device, the datasheet of the device should take priority.

Copyright © Bridgetek Pte Ltd

3 Main Application

3.1 Overview

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
  APP_FlashingDot();            // Creating a basic screen via co-processor
  APP_VertexTranslate();        // Placing items beyond coordinates of 511
  APP_LineStrip();              // Splitting co-processor lists into sections
  APP_Text();                   // Simple text with built-in font
  APP_DigitsFont();             // Loading and using a custom font
  APP_ConvertedBitmap();        // Load and display a small bitmap
  APP_InflateImage();           // Inflating a compressed image
  APP_LoadImagePNG();     // Loading a PNG directly
  APP_SliderandButton(); // Touch tagging and tracking
  APP_ScreenRotate();     // Setting screen orientation to portrait
  APP_Append();           // Optimising screen updates using append
  APP_SnapShot2PPM();     // Take a snapshot line-by-line in PPM format
```

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 and APP_ScreenRotate. 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. The sample code then proceeds to call the selected demo. For ease of readability, the code was designed to have one demo routine un-commented at a time and it will then stay in that demo routine indefinitely. It could however be edited to run each for a certain amount of time etc.

The APP_SnapShotPPM functions can be called to take a snapshot. It would normally be called from within one of the other demo functions above, once a screen has been created which is to be captured. The call to these functions is commented out in each demo initially and section 10 should be consulted before using this feature.

The code must be re-compiled and programmed after changing the demo selection.

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

Copyright © Bridgetek Pte Ltd

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 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 of the FT81X may also be selected via a host command if required before sending the Active command. 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 800 x 480 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;
EVE_MemWrite32(ramDisplayList, CLEAR_COLOR_RGB(0,0,0));    // 0x02000000
ramDisplayList += 4;
EVE_MemWrite32(ramDisplayList, CLEAR(1,1,1));              // 0x26000007
ramDisplayList += 4;
EVE_MemWrite32(ramDisplayList, DISPLAY());                 // 0x00000000
EVE_MemWrite32(REG_DLSWAP, DLSWAP_FRAME);
```

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

Note: The MCU_Init and MCU_UART_Init are in the MCU layer and are covered in BRT_AN_008.

4.2 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.

Copyright © Bridgetek Pte Ltd

Overview

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.

An example co-processor list is shown below:

```
API_LIB_BeginCoProList();
  API_CMD_DLSTART();
  API_CLEAR_COLOR_RGB(0,0,0);
  API_CLEAR(1,1,1);
  API_CMD_TEXT(100, 100, 30, 0, "Calibration - Please tap the dots");
  API_CMD_CALIBRATE(0);
  API_DISPLAY();
  API_CMD_SWAP();
  API_LIB_EndCoProList();
  API_LIB_AwaitCoProEmpty();
```

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.

The calibration values are held in volatile memory and so are not retained. Therefore, 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 (for example after the writing of the display settings registers) instead of running the calibration.

The second case above would be commonly used in cases where the device is being powered on and off regularly so that the end user does not need to carry out calibration. Some additional considerations are discussed below.

Storing and Recalling the Values

When carrying out the calibration, the values are *only* valid after completion of a successful calibration command and can be read via 32-bit reads of the REG_TOUCH_TRANSFORM_A to _F. Completion is indicated by the REG_CMD_WRITE and REG_CMD_READ pointers becoming equal after the command, as tested by the API_LIB_AwaitCoProEmpty call above.

The values are not validated by the EVE device and so incorrect touch alignment can result if the user accidentally touches the screen in a different location whilst the calibration routine awaits a

Copyright © Bridgetek Pte Ltd

dot to be tapped, or if the user's tap is not accurately aligned on the dot. It is therefore suggested that a post-calibration test screen is generated by the user application asking the user to tap dots or symbols. If a touch is detected on the screen (for example REG_TOUCH_SCREEN_XY != 0x80008000) but is *not* aligned on the item then the values can be discarded and another calibration co-processor list run.

It is also recommended that a re-calibrate option is added by some means (via a menu system option, or restricted service menu, or physical jumper on an MCU GPIO pin) in case the LCD panel has to be replaced in the field etc.

The values require six 32-bit (or 24 bytes) storage locations in the MCU's non-volatile memory. The application may also wish to use an additional byte or 32-bit value stored alongside the values as a status flag which could indicate whether good calibration data is stored. If the location is blank when starting up, a calibration could be run instead of trying to read the stored values.

Once valid transform values have been stored, they can be retrieved by the MCU from its non-volatile memory and written back using six 32-bit register writes to REG_TOUCH_TRANSFORM_A to _F. This could be carried out after setting the display registers.

Copyright © Bridgetek Pte Ltd

5 Basic Graphics Operations

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.

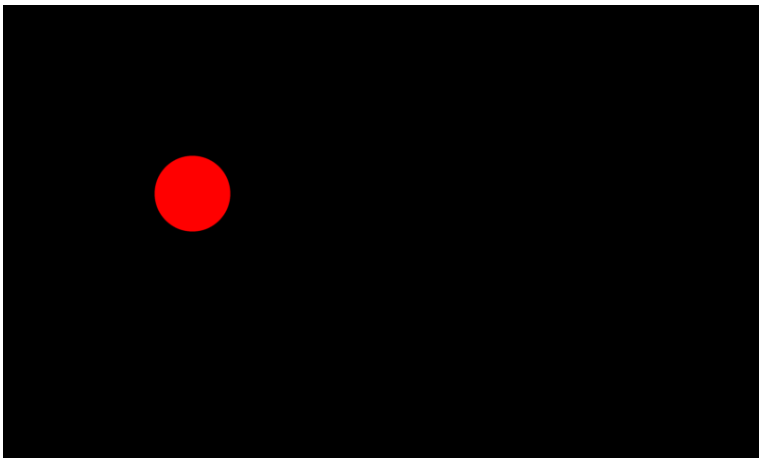


Figure 2 - Flashing Dot demo

The code runs in a constant while(1) loop. First, a variable is toggled which will be used to determine the colour of the dot on each screen update.

```
void APP_FlashingDot(void)
{
    uint8_t color = 0;

    while(1)
    {
        if(color == 0x00)
            color = 0xFF;
        else
            color = 0x00;
```

It then goes on to create the new co-processor list to generate the screen content. The API_LIB_BeginCoProList function is called to begin the new list.

API_LIB_BeginCoProList performs several functions. It first waits for the co-processor FIFO to be empty (whereby the write and read pointers are equal) via a call to API_LIB_AwaitCoProEmpty(). The address within the circular co-processor FIFO to which they currently point is also obtained. It uses this value as the starting address for the new co-processor list. The function will assert the CS# line and send the address thereby beginning a burst write cycle to the CMD_FIFO.

The first command (CMD_DLSTART) will tell the co-processor to make a new display list beginning at offset RAM_DL + 0. The first entries in this list are to set the colour to be used when clearing the screen and to clear the colour, stencil and tag buffers. In most cases, a new co-processor list will begin in a similar way to the four commands shown.

The API_ function call in each case will send the associated command over SPI and will update a variable to keep track of the number of bytes sent.

```
        API_LIB_BeginCoProList();
        API_CMD_DLSTART();
        API_CLEAR_COLOR_RGB(0,0,0);
        API_CLEAR(1,1,1);
```


Copyright © Bridgetek Pte Ltd

Now, the colour of the dot to be drawn is set and a point is drawn on the screen, using standard commands from the FT81X Programmers Guide.

```
        API_COLOR_RGB(color, 0, 0);
        API_BEGIN(FTPOINTS);
        API_POINT_SIZE(40*16);
        API_VERTEX2F(200*16, 200*16);
        API_END();
```

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.

```
        API_DISPLAY();
        API_CMD_SWAP();
```

The display creation finishes with two API_LIB function calls. The first one will bring CS# high to finish the burst write over SPI and will then perform a write to the REG_CMD_WRITE register to point it to the end of the new commands added. The call to API_LIB_AwaitCoProEmpty will then wait until the FT81X's internal REG_CMD_READ pointer has caught up with REG_CMD_WRITE and therefore until the co-processor has consumed all of the commands.

```
        API_LIB_EndCoProList();
        API_LIB_AwaitCoProEmpty();
```

Finally, a delay is provided so that the screen refreshes approximately every 500ms and the flashing of the dot is slow enough to be visible to the user.

```
        MCU_Delay_500ms();
    }
}
```

Note that the code here works in exactly the same way as the flashing dot illustrated in BRT_AN_006 but the main code is simpler due to the addition of the new API layer. The APIs such as `API_LIB_BeginCoProList();` takes care of awaiting the co-pro FIFO being empty, sending the address and handling chip select. Each API_ call which sends a GPU instruction or command takes care of keeping track of the new offset. This makes programming the screen content much more user-friendly. The code for the lower layers is however available in the API_Layer.c, EVE_Layer.c and MCU_Layer.c files should the developer wish to check or optimise the underlying code.

5.2 APP_VertexTranslate

This section gives a simple demonstration of the VERTEX_TRANSLATE_X and _Y commands.

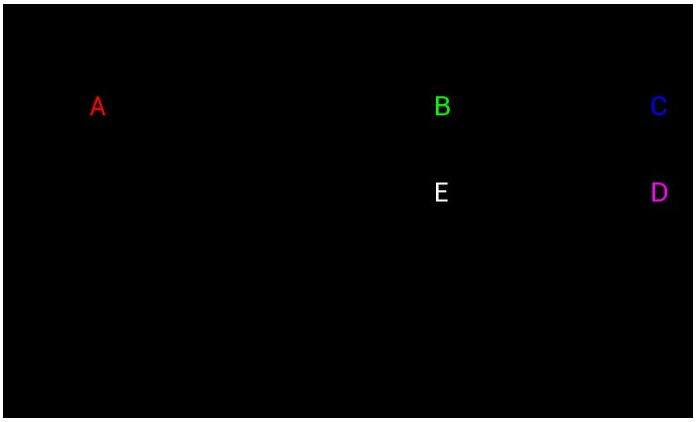


Figure 3 - Vertex Translate demo

Copyright © Bridgetek Pte Ltd

The VERTEX_TRANSLATE_X and VERTEX_TRANSLATE_Y instructions allow an offset to be added to the x coordinate and y coordinate respectively of a VERTEX command. One particular use case is with VERTEX2II where the item may need to be positioned beyond coordinates of 511 in order to reach the edge of an 800 pixel-wide screen.

When positioning objects on the screen of the FT81X (which supports 800x600 pixels) most commands and instructions, such as TEXT and VERTEX2F, already allow coordinates spanning the full screen (e.g. VERTEX2F(750,100)). These items can therefore be placed anywhere within the screen area without using VERTEX_TRANSLATE.

However, the VERTEX2II instruction can only accept coordinates from 0 to 511. VERTEX2II is necessary in cases where the bitmap cell must be specified in addition to the handle. VERTEX_TRANSLATE can be used to extend the range of coordinates by adding an offset.

To illustrate the use of VERTEX_TRANSLATE, one of the built in fonts is used to represent a bitmap with multiple cells which is positioned via the VERTEX2II instruction. Font handle 30 is used and there are 128 cells (characters) indexed from 0 to 127. The cell numbers were chosen to print the letters A – E (ASCII 0x41 - 0x45) in this case.

First, a red letter A is drawn at (100,100) and a green letter B is drawn at (500,100) both of which are within the bounds of the VERTEX2II instruction coordinates.

```
API_COLOR_RGB(255, 0, 0);
API_VERTEX2II(100,100,30,0x41); // Letter A at (100,100)

API_COLOR_RGB(0, 255, 0);
API_VERTEX2II(500,100,30,0x42); // Letter B at (500,100)
```

A blue letter C 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 VERTEX2II call.

```
API_COLOR_RGB(0,0,255);
API_VERTEX_TRANSLATE_X(250*16);
API_VERTEX2II(500,100,30,0x43); // Letter C at ((500+250),100)
```

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

```
API_COLOR_RGB(255, 0, 255);
API_VERTEX2II(500,200,30,0x44); // Letter D at ((500+250),200)
```

Finally, the VERTEX_TRANSLATE_X is set back to 0 again and so the white letter E appears at point (500,200) as per the values stated in the VERTEX2II call.

```
API_COLOR_RGB(255, 255, 255);
API_VERTEX_TRANSLATE_X(0*16);
API_VERTEX2II(500,200,30,0x45); // Letter E at (500,200)
```

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.

Copyright © Bridgetek Pte Ltd

5.3 APP_LineStrip

This example plots a simple line chart from data in an array. It demonstrates creating a list in several sections (for example when sending more than (4K-4) bytes of commands).

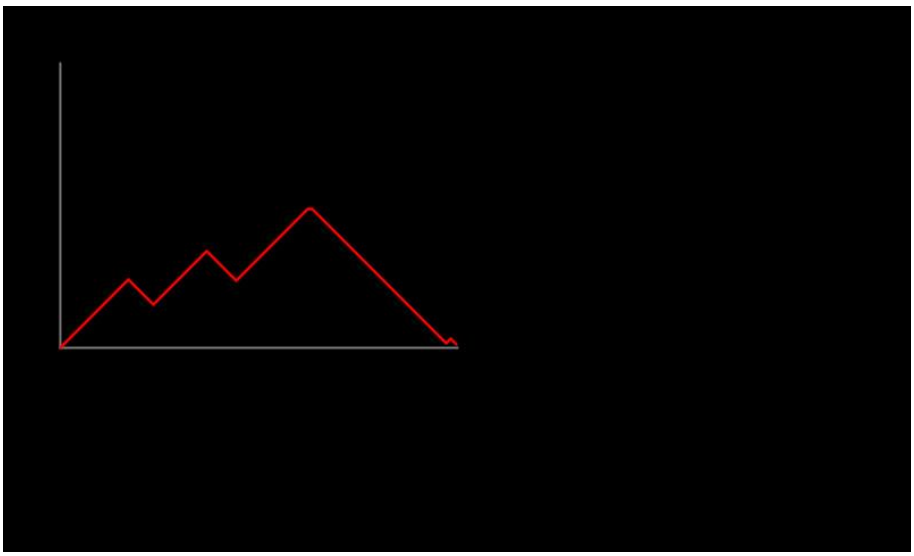


Figure 4 - LineStrip demo

As discussed in BRT_AN_008, the co-processor executes commands passed in via a 4K circular FIFO (RAM_CMD). For efficiency, commands are often written via an SPI burst write whereby the FT8XX increments its internal address pointer for each new data value written whilst chip select is held low. This allows writing of multiple commands without sending the register address each time (which is required after each assertion of chip select as this is a new SPI transaction). The APIs below are used throughout this document to implement burst writes.

```
API_LIB_BeginCoProList();     // CS# low and send first address
[commands]                    // one or more commands
API_LIB_EndCoProList();       // CS# high and update co-pro write pointer
API_LIB_AwaitCoProEmpty();    // await completion (read & write pointers equal)
```

Assuming that the FIFO is currently empty (REG_CMD_READ ==REG_CMD_WRITE) the burst write may send a maximum of (4k-4) bytes to RAM_CMD to avoid overlapping the first bytes. However, since the co-processor creates the Display List entries in RAM_DL and this can hold up to 8K of data, there may be cases where more than one burst write is required to complete the display list. In many cases an application may not need more than (4K-4) of commands to create the screen or to use most of the RAM_DL. Firstly, for some commands the number of RAM_DL bytes generated by the co-processor can be greater than the size of the command itself written to RAM_CMD (e.g. 3D buttons). Secondly, due to the object-oriented nature of the EVE family, screens can be created efficiently with relatively few instructions.

However, there are other cases where a screen does need more than (4K-4) commands in order to use the full RAM_G available. This may be due to the screen containing a lot of items, but is also influenced by the types of commands used. For example, some applications may use a lot of VERTEX commands (either directly when positioning shapes, bitmaps and line strips etc. or via the text command which generates VERTEX instructions in RAM_DL). In these cases, the solution is to send a section of the command list and then cause the co-processor to execute it, leaving space for the next section. Co-processor instructions can be sent in several sections or even to the extent of every command as a separate transaction. The main trade-off is in the efficiency.

Note: Despite the circular nature of RAM_CMD, the 8K limit on the resulting RAM_DL data remains the limiting factor and must not be overrun. During application development, overrun can be determined by a co-processor error condition (REG_CMD_READ == 0xFFF) and by REG_CMD_DL reaching 8K in value.

Copyright © Bridgetek Pte Ltd

Splitting co-processor lists into multiple bursts

This example uses a line strip to illustrate the technique. Whilst the number of chart points here could be accommodated in a single burst, this may not always be the case when creating a screen and so splitting the list into multiple sections may be required.

The first burst write begins in the same way as the examples above. API_Lib_BeginCoProList asserts Chip Select and sends the address of the first register to be written. DL_Start then causes the co-processor to begin writing the Display List at RAM_DL+0. It does this by setting REG_CMD_DL (which points to the next location in RAM_DL where the co-processor will write generated instructions) to 0. The commands used to create the screen then follow. These do not include a DISPLAY or SWAP command as this is not the end of the overall list. The burst write ends with API_LIB_EndCoProList which causes the co-processor to execute the commands and create the corresponding display list entries. The call API_LIB_AwaitCoProEmpty waits for the read and write pointers to be equal and at this point the RAM_CMD is empty.

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();
API_CLEAR_COLOR_RGB(0, 0, 0);
API_CLEAR(1,1,1);
API_COLOR_RGB(128, 128, 128);
API_BEGIN(LINES);
API_LINE_WIDTH(16);
API_VERTEX2F(Chart_TopLeftX*16, Chart_TopLeftY*16);       // Draw Y axis
API_VERTEX2F(Chart_TopLeftX*16, Chart_BottomRightY*16);
API_VERTEX2F(Chart_TopLeftX*16, Chart_BottomRightY*16);   // Draw X axis
API_VERTEX2F(Chart_BottomRightX*16, Chart_BottomRightY*16);
API_END();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

The next section of the list can now be created. API_Lib_BeginCoProList asserts Chip Select and sends the address of the first register to be written. There is no DL_START in this case so that the co-processor will begin writing the resulting entries to RAM_DL at the end of the first part created above. The instructions then follow. In this case, a chart line is created using a large number of 4-byte VERTEX commands which will be passed to the RAM_DL by the co-processor. The burst write again ends with API_LIB_EndCoProList which causes the co-processor to execute the commands and create the corresponding display list entries. The call API_LIB_AwaitCoProEmpty waits for the read and write pointers to be equal and at this point the RAM_CMD is empty.

```
API_LIB_BeginCoProList();
API_COLOR_RGB(255,0,0);
API_BEGIN(LINE_STRIP);
API_LINE_WIDTH(16);
for (ChartCounter = 0; ChartCounter < 350; ChartCounter++)
{
    API_VERTEX2F(   ((ChartCounter + Chart_TopLeftX)*16), ((Chart_TopLeftY +
         Chart_SizeY - ChartData[ChartCounter])*16 )   );
}
API_END();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

The final section of the list in this application repeats the above process to add a third section of the Display List onto the end of the preceding section. In this case, the DISPLAY and SWAP commands are used.

```
API_LIB_BeginCoProList();
API_DISPLAY();
```


Copyright © Bridgetek Pte Ltd

```
API_CMD_SWAP();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

The overall result of this process is equivalent to creating a display list with a single burst write except that more than (4K-4) of co-processor commands have been able to be used.

Chart

The chart itself was created using the LINE STRIP command followed by a series of VERTEX2F commands.

```
API_BEGIN(LINE_STRIP);
API_LINE_WIDTH(16);
for (ChartCounter = 0; ChartCounter < 350; ChartCounter++)
{
    API_VERTEX2F(   ((ChartCounter + Chart_TopLeftX)*16), ((Chart_TopLeftY +
         Chart_SizeY - ChartData[ChartCounter])*16 )   );
}
API_END();
```

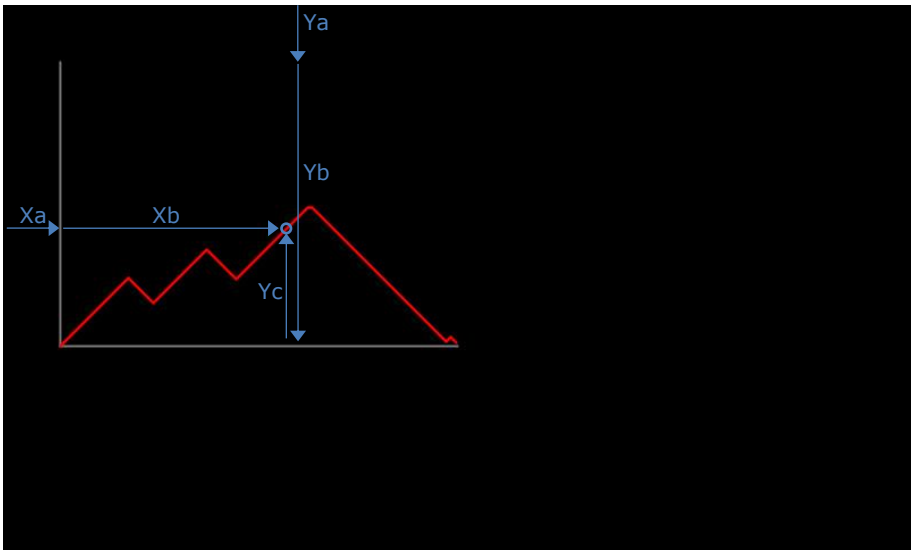


Figure 5 – Chart using Line Strip

Each VERTEX2F command sent specifies the coordinates of a point on the line. The code plots a series of 350 points.

For the X direction, the coordinate system on the FT8XX increases from left to right and so the X value for each subsequent coordinate (Xb) is incremented by 1. An offset (Xa) is added corresponding to the starting X offset of the chart.

For the Y direction, the coordinate system on the FT8XX increments from top to bottom and so the calculation adds the Y value of the top of the chart (Ya) to the Y dimension of the chart (Yb) and then subtracts the actual value to be displayed (Yc) to derive the final value for the VERTEX2F coordinate.

Copyright © Bridgetek Pte Ltd

6 Displaying Text

6.1 APP_Text()

This demo writes a basic text string to the screen using the FT8XX Text feature.

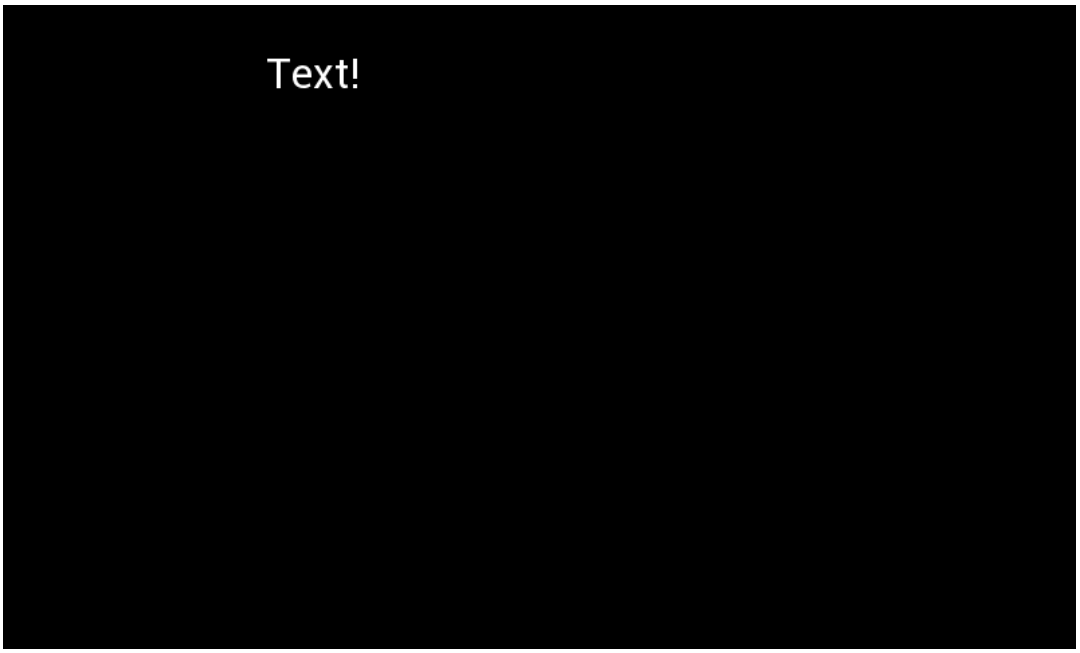


Figure 6 - Text demo

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.

This uses the same technique as the text example in BRT_AN_007 but the additional API layer in the BRT_AN_008 library used here handles the sending of the command, parameters, and associated string data and makes the main application much easier to develop.

```
API_LIB_BeginCoProList
API_CMD_DLSTART();
API_CLEAR_COLOR_RGB(0, 0, 0);
API_CLEAR(1,1,1);
API_COLOR_RGB(255, 255, 255);
API_CMD_TEXT(196, 33, 30, 0, "Text!");
API_DISPLAY();
API_CMD_SWAP();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```


Copyright © Bridgetek Pte Ltd

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!
```

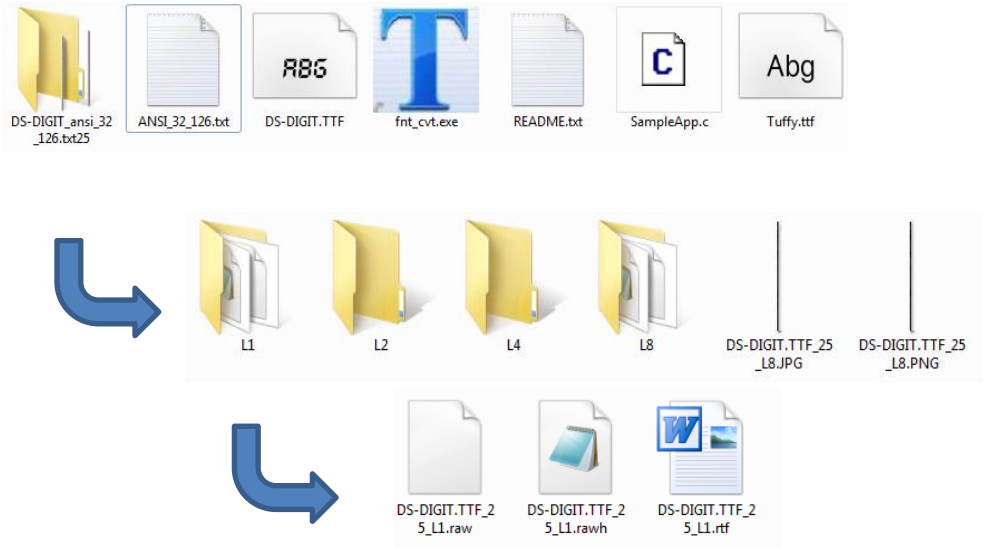


Figure 8 - Digits custom font folders

Copyright © Bridgetek Pte Ltd

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 0x8200 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 0x8200 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 DigitsFont [] @ 0x8200 = {

/* 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]

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

The first part of the APP_DigitFont() demo will load the data byte-by-byte from the array into the RAM_G of the FT81X using an SPI burst write. The function takes as parameters the location of the array containing the data in the MCU's memory and the size of the data to be transferred. The third parameter specifies the starting address within the FT81X where the data will be copied. This line of code is all that is required to load the data into the FT81X. As this is raw data written direct to RAM_G, the size written will be the same as the size of the source data (including the metric block at the beginning). Care should be taken to organise the use of the RAM_G to avoid accidentally overwriting other data.

```
API_LIB_WriteDataRAMG(DigitsFont, sizeof(DigitsFont), 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

Copyright © Bridgetek Pte Ltd

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.

The raw data address is calculated as an offset which points to where the 0th character of the font data *would* have been. The actual data loaded begins with the first printable character but since this would be the 32nd character in an ASCII data set, the virtual value of the raw data address of the 0th character would have been at:

*(Address of data in RAM_G) + (Metric Block size) – (32 characters worth of (stride * height))*
1000 + 148 – (32(3*25))*
= 1000 + 148 – 2400
= -1252

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

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();

API_CLEAR_COLOR_RGB(0, 0, 0);
API_CLEAR(1,1,1);
API_BITMAP_HANDLE(14);
API_BITMAP_SOURCE(-1252);
API_BITMAP_LAYOUT(L1,3,25);
API_BITMAP_SIZE(NEAREST, BORDER, BORDER, 18,25);
```

The font can now be used throughout the application to produce a simple text string by setting the color desired and then using the CMD_TEXT command with the font set to 14. The same font can be used for the CMD_NUMBER command too.

```
API_CMD_SETFONT(14, 1000);
API_COLOR_RGB(255,255,255);
API_CMD_TEXT(100,100,14,0,"HELLO");
API_CMD_NUMBER(100, 300, 14, 0, 12345);
API_DISPLAY();
API_CMD_SWAP();

API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

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. The FT81X also offers the CMD_SETFONT2 command which allows the ASCII value of the first character in the font to be specified.

Copyright © Bridgetek Pte Ltd

7 Images

7.1 APP_ConvertedBitmap()

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

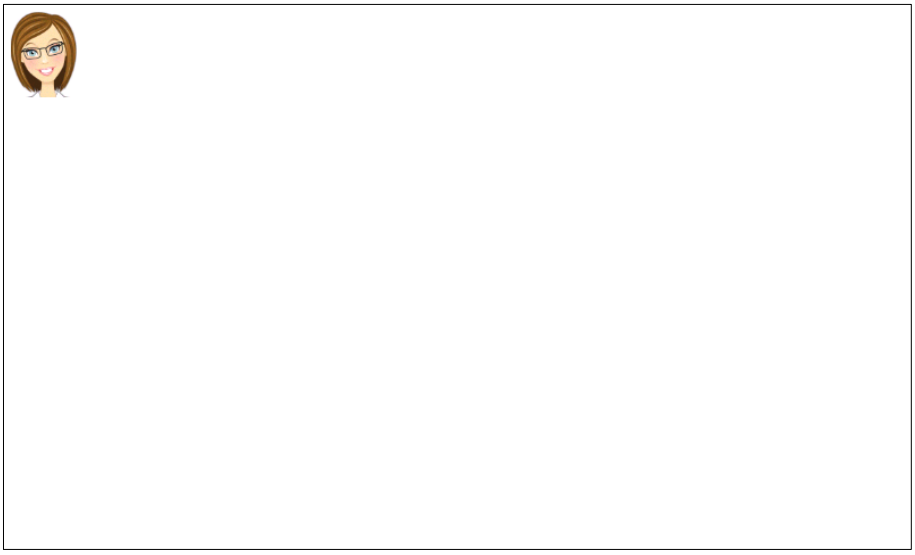


Figure 9 - Bitmap demo

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 (EVE_RGB565 folder). 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.

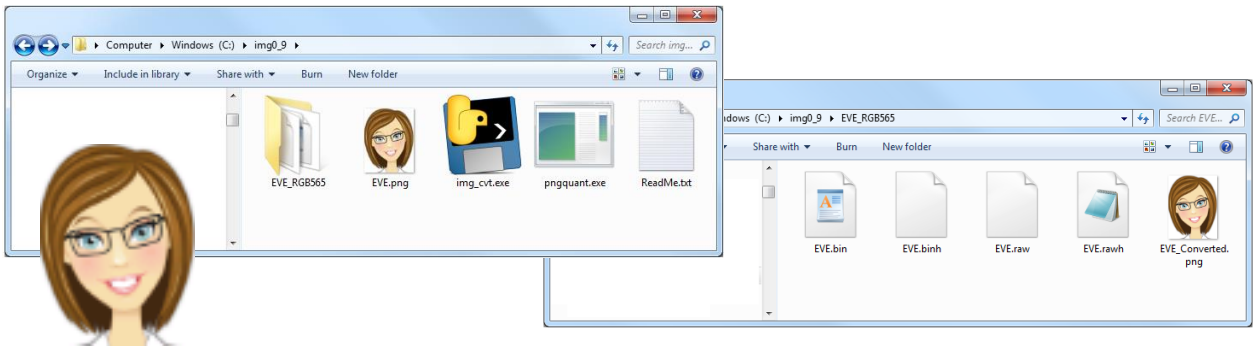


Figure 10 - Converting bitmap file

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

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


Copyright © Bridgetek Pte Ltd

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 is provided by the call below. This loads the image data from array rawData[] into RAM_G beginning at address RAM_G+0.

```
API_LIB_WriteDataRAMG(rawData, sizeof(rawData), 0);
```

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 and used many times throughout the application. The Bitmap Layout then specifies the file format, line stride and height which instruct the FT8XX graphics engine on 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 can be obtained from the comments at the start of the raw data file output by the converter.

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();
API_CLEAR_COLOR_RGB(255, 255, 255);
API_CLEAR(1,1,1);
API_BITMAP_HANDLE(0);
API_BITMAP_SOURCE(0);
API_BITMAP_LAYOUT(RGB565, 116, 75);
API_BITMAP_SIZE(NEAREST, BORDER, BORDER, 58, 75);
API_COLOR_RGB(255, 255, 255);          // White
API_BEGIN(BITMAPS);
API_VERTEX2F(0,0);                     // Display the image
API_END();
API_DISPLAY();
API_CMD_SWAP();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

Note that the last colour set (by COLOR_RGB) before the VERTEX which displays the image will set the colour of the bitmap (in a similar way to looking through a coloured glass at the image). To display in its original colours, set COLOR_RGB to white as demonstrated above.

The FT81X has a new command CMD_SETBITMAP which can replace the lines highlighted in blue above with a single command `API_CMD_SETBITMAP(0,RGB565,58,75);`

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.

Copyright © Bridgetek Pte Ltd

7.2 APP_InflateImage()

This example inflates an image and displays it on the screen. One key advantage is that the file stored on the MCU is smaller compared to storing the image directly. The end address of the image is also displayed in Decimal and Hex.

Figure 11 – Inflating an image

The EVE Image Converter was used to create a compressed data file from a PNG image in RGB332 format which was then loaded into the PIC's memory. The original PNG file can be found within the zip file package provided with this application note.

After downloading and un-zipping the image converter to a folder in C:\ the LeaveItToEVE.png file was copied to this folder. Then, the following command line was run to convert the file into format RGB332 which corresponds to 4 in the list of formats provided by the tool.

```
Img_cvt -I LeaveItToEVE.png -f 4
```

A new folder will appear in the Image Converter program folder which contains the converted data (LeaveItToEVE_RGB332 folder). The resulting files are shown in the second screenshot below.

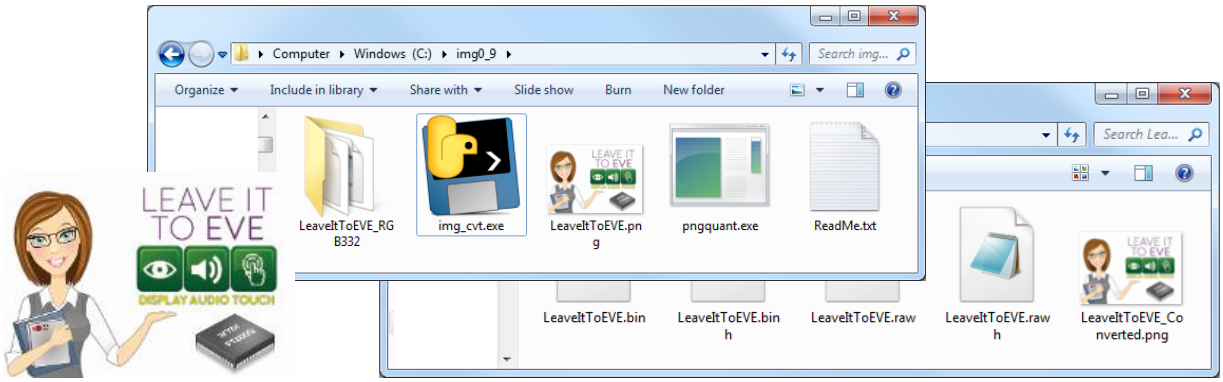


Figure 12 - Converting to a compressed file

A Hex editor was used to copy the contents of the compressed LeaveItToEVE.bin file into a C array. Some editors have a useful 'copy as C' feature which copies the data and allows it to be pasted into a blank header file. The data can be found in LeaveItToEVE.h in the provided project. The hex tool adds the brackets etc. for the array and so after pasting, all that is required is to add any MCU-specific aspects to the array name. In this case, the @ 0x5200 specifies the address within the PIC where the data will be stored. Please check the documentation of the selected MCU for information on the memory map and data storage locations available.

Copyright © Bridgetek Pte Ltd

```
const uint8_t LeaveItToEVE[] @ 0x5200 =
```

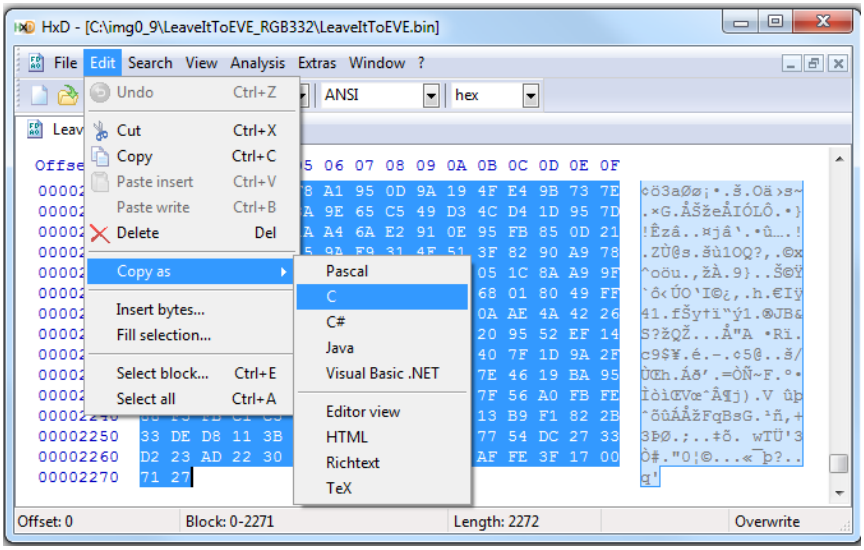


Figure 13 – Copying data from the compressed file

Since this is a compressed file, the co-processor must be used to decompress the data. Therefore, instead of writing this array to the RAM_G (as carried out in the ConvertedBitmap demo) the data is written to the co-processor. The co-processor will then inflate the data into a format in RAM_G on the user's behalf which can be used directly by the GPU.

The library sends the CMD_INFLATE command to the co-processor as a separate command (using the BeginCoProList and EndCoProList to do this). The parameter 0 indicates that the co-processor will start outputting the resulting data to address 0 which is the beginning of RAM_G. The co-processor now awaits the data to be inflated. The data follows immediately afterwards via the WriteDataToCmd function which sends the array of data created above. This also takes care of managing the co-processor read and write pointers and splitting the data into smaller chunks of necessary. Finally, the code checks for completion by ensuring the buffer write and read pointers are equal. The co-processor will give an error condition (REG_CMD_READ = 0xFFF) if the extracted data would overrun the available RAM_G or if the input file is in a non-supported format. The code could be extended to handle this error.

```
API_LIB_BeginCoProList();
API_CMD_INFLATE(0);
API_LIB_EndCoProList();
API_LIB_WriteDataToCMD(LeaveItToEVE, sizeof(LeaveItToEVE));
API_LIB_AwaitCoProEmpty();
```

An optional step then follows to allow the end of the inflated data to be determined, which will be useful when loading data into subsequent RAM_G locations to make efficient use of the memory. Whilst the commands above define that the data will begin at 0, the end address is not known as the inflated data will be larger than the compressed data written to the CMD_INFLATE command. The CMD_GETPTR can be used to check the ending address. This command actually returns its value via the co-processor FIFO itself. The command (4 bytes) is sent with a dummy (4-byte) value as a parameter and so occupies two 32-bit locations in the command FIFO. On completion, the co-processor replaces the dummy value with the ending address. The application must then perform a read of this location (which can be obtained by checking the current REG_CMD_WRITE pointer and subtracting 4 taking account of any possible rollover at (RAM_CMD+0).

```
API_LIB_BeginCoProList();
API_CMD_GETPTR(0);
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
Reg_Cmd_Write_Offset = EVE_MemRead16(REG_CMD_WRITE);
```


Copyright © Bridgetek Pte Ltd

```
Reg_Cmd_Write_Offset = ((Reg_Cmd_Write_Offset - 4) & 4095);
End_Address = EVE_MemRead32((RAM_CMD+Reg_Cmd_Write_Offset));
```

Since the co-processor has created a bitmap file in RAM_G which the GPU can process, the image can now be used like any other bitmap (for example, like the ConvertedBitmap demo provided with this application note).

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();
API_CLEAR_COLOR_RGB(0, 0, 0);
API_CLEAR(1,1,1);
API_COLOR_RGB(255,255,255);
API_BITMAP_HANDLE(5);
API_BITMAP_SOURCE(0);
API_BITMAP_LAYOUT(RGB332, 200, 141); // FMT, STR, H
API_BITMAP_SIZE(BILINEAR, BORDER, BORDER, 200, 141);  // W, H
API_BEGIN(BITMAPS);
API_VERTEX2II(100,100,5,0);
API_END();
```

For the purposes of the demo, the code also displays the address of the end of the data which was obtained by the optional GETPTR step earlier in both decimal and hex values. The FT81X's SETBASE command is useful here for setting the number base.

```
API_COLOR_RGB(255,255,255);
API_CMD_NUMBER(300, 260, 30, 0, End_Address);
API_CMD_SETBASE(16);
API_CMD_NUMBER(300, 300, 30, 0, End_Address);
```

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.

Note: Consult the FT81X Programmers Guide (see Appendix A– References) for details of the Inflate command and the supported file formats as only some types of compressed image file are supported. Using compressed files means that the data stored on the MCU is smaller compared to storing the image directly. The space required in the RAM_G of the FT81X will be equivalent to the un-compressed version of the file however and so the usage of RAM_G will be larger than the data array which is loaded to the co-processor.

Copyright © Bridgetek Pte Ltd

7.3 APP_LoadImagePNG()

The EVE devices can also decompress some PNG images directly without using the EVE Image converter. In this case, the same PNG image used in the ConvertedBitmap demo is loaded directly.

Figure 14 – Loading a PNG image

The image file can be found in the APP_LoadImagePNG folder of the provided zip file. There are a variety of ways of storing the file on the MCU but an easy way is to open the PNG file in a hex editor and copy the data as a C array and paste into an empty header file in the MCU project.

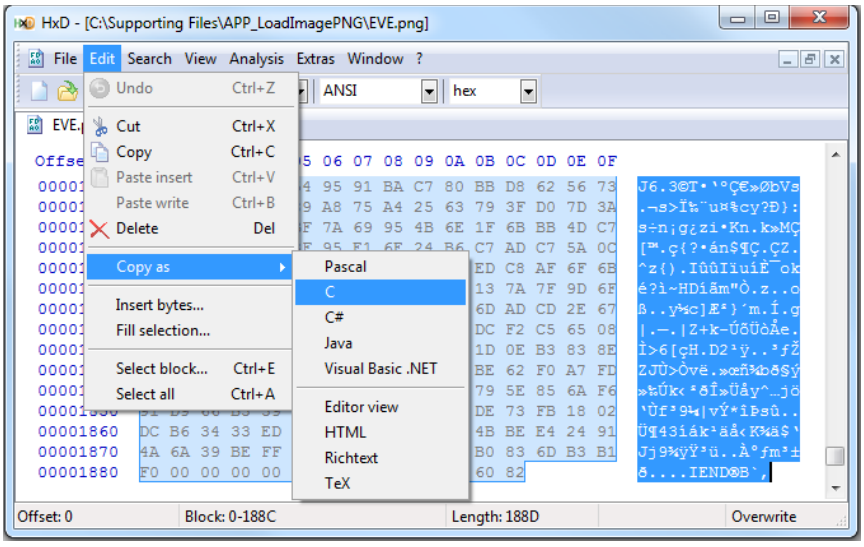


Figure 15 – Copying data from the PNG file

As with the earlier examples in this application note, the array name can be edited to suit the MCU / compiler syntax. The EVE_PNG.h file contains the pasted data and has been configured in MPLABX syntax to store the data at location 0x3200 in the PIC.

```
const uint8_t EVE_PNG[] @ 0x3200 = {
```

The CMD_LOADIMAGE command is used here. This is used in a similar way to the CMD_INFLATE in the previous example. The command is sent to the co-processor followed by two parameters which

Copyright © Bridgetek Pte Ltd

define the address to which the data will be decompressed in RAM_G and also an options parameter. In this case, the image data will be decompressed to RAM_G + 0 and no special options are selected (see the FT81X Programmers Guide for full details).

As with the INFLATE command, the data follows immediately after the command and is written to the co-processor FIFO. The WriteDataToCMD function provides all of the actions necessary to load the data. The co-processor will give an error condition (REG_CMD_READ = 0xFFF) if the extracted data would overrun the available RAM_G or if the input file is in a non-supported format. The code could be extended to handle this error.

```
API_LIB_BeginCoProList();
API_CMD_LOADIMAGE(StartAddress,0);
API_LIB_EndCoProList();
API_LIB_WriteDataToCMD(EVE_PNG, sizeof(EVE_PNG));
API_LIB_AwaitCoProEmpty();
```

The properties of the decompressed image can then be obtained. This uses a similar technique to that described in the APP_InflateImage() demo. The CMD_GETPROPS is used instead of CMD_GETPTR when determining the properties of an image loaded via CMD_LOADIMAGE. It provides the ending address and the width and height.

```
API_LIB_BeginCoProList();
API_CMD_GETPROPS(0, 0, 0);                    // 3 dummy 32-bit values
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();

Reg_Cmd_Write_Offset = EVE_MemRead16(REG_CMD_WRITE);

ParameterAddr = ((Reg_Cmd_Write_Offset - 12) & 4095); // 12 bytes back
End_Address = EVE_MemRead32((RAM_CMD+ParameterAddr));

ParameterAddr = ((Reg_Cmd_Write_Offset - 8) & 4095); // 8 bytes back
Width = EVE_MemRead32((RAM_CMD+ParameterAddr));

ParameterAddr = ((Reg_Cmd_Write_Offset - 4) & 4095); // 4 bytes back
Height = EVE_MemRead32((RAM_CMD+ParameterAddr));
```

After this, the image can be displayed as a standard bitmap. The properties can be set using the End_Address, Width and Height values obtained above. Although the LOADIMAGE command can generate display list entries for the image properties, obtaining the values and storing them as shown above allows them to be used later on in the application as many applications may load and decompress the image and then use it later on.

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();
API_CLEAR_COLOR_RGB(0, 0, 0);
API_CLEAR(1,1,1);
API_COLOR_RGB(255,255,255);

API_BITMAP_HANDLE(0);
API_BITMAP_SOURCE(StartAddress);
API_BITMAP_LAYOUT(RGB565, Width*2, Height); // Format, Stride, Height
API_BITMAP_SIZE(BILINEAR, BORDER, BORDER, Width, Height);
API_BITMAP_LAYOUT_H((Width * 2) >> 10, Height >> 9);
API_BITMAP_SIZE_H(Width >> 9, Width >> 9);

API_BEGIN(BITMAPS);
API_VERTEX2II(200,200,0,0);
API_END();
```


Copyright © Bridgetek Pte Ltd

An optional set of instructions (for the purposes of this demo) displays the end address in decimal and hex bases, and the width and height.

```
API_COLOR_RGB(255,255,255);
API_CMD_TEXT(300, 200, 28, 0, "End Addr");
API_CMD_TEXT (300, 240, 28, 0, "Width");
API_CMD_TEXT (300, 280, 28, 0, "Height");

API_CMD_NUMBER(400, 200, 28, 0, End_Address);
API_CMD_NUMBER(400, 240, 28, 0, Width);
API_CMD_NUMBER(400, 280, 28, 0, Height);
API_CMD_SETBASE(16);
API_CMD_NUMBER(500, 200, 28, 0, End_Address);
```

The co-processor list then finishes in the usual way resulting in the screen being displayed.

```
API_DISPLAY();
API_CMD_SWAP();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

Note: Consult the FT81X Programmers Guide (see Appendix A– References) for details of the CMD_LOADIMAGE command and the supported file formats as only some types of compressed image file are supported. The CMD_LOADIMAGE command will decompress the data to RAM_G in order for the GPU to display the image. The decompressed data may be significantly larger than the data which is fed into the co-processor FIFO and so care should be taken that the image will fit in the available RAM_G.

Copyright © Bridgetek Pte Ltd

8 Touch Examples

8.1 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.

Note: Calibration is required for touch-based demos. Refer to section 4.2 for details.

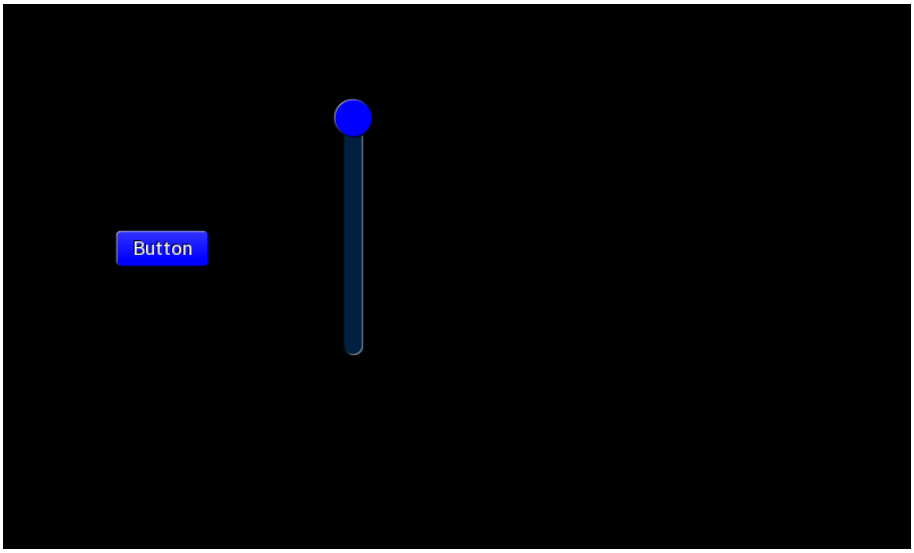


Figure 16 - Slider and Button demo

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 can be applied which can be any number from 1 to 255. The tag 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 un-masked, 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.

Copyright © Bridgetek Pte Ltd

```
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 17 - 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).

```
API_TAG_MASK(1);
API_TAG(2);
API_CMD_FGCOLOR(0x0000FF);
API_CMD_BUTTON(100, 200, 80, 30, 27, Button3D, "Button");
```

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.

Copyright © Bridgetek Pte Ltd

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.

```
API_TAG(5);
API_CMD_SLIDER(300, 100, 16, 200, 0, SlideVal, 255);
API_CMD_TRACK(300, 100, 16, 200, 5);
```

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

```
API_TAG_MASK(0);
```

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);
}
```

8.2 APP_ScreenRotate()

This demo uses the same code as APP_SliderAndButton()but has an additional command to rotate the screen. Calibration is required for touch-based demos. Refer to section 4.2 for details.

Copyright © Bridgetek Pte Ltd

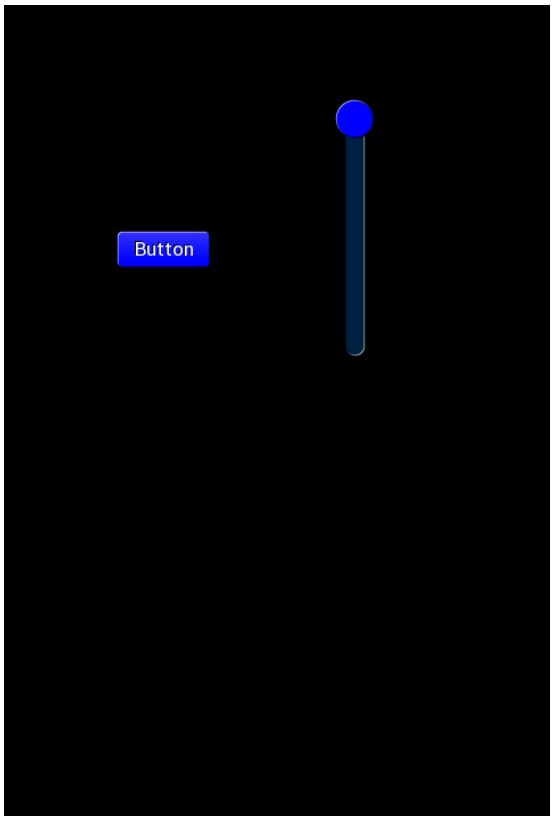


Figure 18 - Slider and Button demo on rotated screen

As with the App_SliderAndButton demo, calibration of the screen is required and will have been completed already by APP_Calibrate before this demo routine is called.

The FT81X supports rotation of the screen into 8 different orientations. Full details of these can be obtained in the FT81X Programmers Guide (see Appendix A– References) under sections 2.5.3 (Screen Rotation) and 5.53 (CMD_SETROTATE).

It is recommended to use the co-processor command CMD_SETROTATE. This will provide both screen rotation and touch rotation. It will write the value to REG_ROTATE for the screen rotation and will adjust the touch transform matrix so that the touch is aligned to the rotated screen.
Writing the REG_ROTATE directly would rotate only the screen but not the touch.

When sending individual commands such as CMD_SETROTATE() using this library, the following technique is recommended:

```
API_LIB_BeginCoProList();    // Set up an SPI transfer to the co-pro FIFO
API_CMD_SETROTATE(2);        // Command(s) to be sent
API_LIB_EndCoProList();      // Finish SPI transaction
API_LIB_AwaitCoProEmpty();   // Await completion by the co-pro
```


Copyright © Bridgetek Pte Ltd

9 Optimising Screen Updates

9.1 APP_Append()

This demo provides an example of optimising the screen updates in cases where the majority of items on the screen are static and only a few values or parameters change. This is explained further in AN_340.

The text, line border and background are all stored in the RAM_G and the dynamic parts (gauge, toggle switch and numerical counter) are added on each refresh.

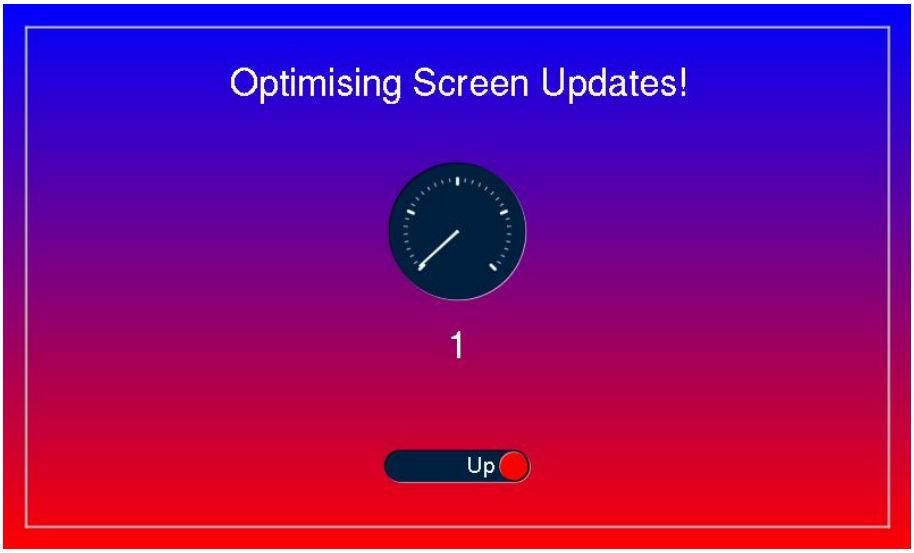


Figure 19 – Optimising screen updates with Append

Creating the Static Content

The application begins by creating the static parts which will not change. This takes the form of a standard screen creation sequence with the CMD_DLSTART causing a new display list to be created at RAM_DL + 0.

Note however that the clearing of the screen is not carried out. In a bigger application, this static section could be one of many building blocks used to create a screen and so clearing of the screen in each block would erase content from previous blocks when they are all put together. These will be added by the main code routine which builds up the display with these blocks.

In addition, the Display and Swap should be commented out as there will be a single instance of these added by the main code creating the final display.

```
API_LIB_BeginCoProList();
API_CMD_DLSTART();
// API_CLEAR_COLOR_RGB(255, 255, 255);
// API_CLEAR(1,1,1);
API_CMD_GRADIENT(300, 0, 0x0000FF, 300, 480, 0xFF0000);
API_COLOR_RGB(255, 255, 255);
API_CMD_TEXT(200, 50, 25, 0, "Optimising Screen Updates!");
API_COLOR_RGB(200, 200, 200);
API_BEGIN(LINE_STRIP);
API_VERTEX2F(20*16, 20*16);
API_VERTEX2F(780*16, 20*16);
API_VERTEX2F(780*16, 460*16);
API_VERTEX2F(20*16, 460*16);
```


Copyright © Bridgetek Pte Ltd

```
API_VERTEX2F(20*16, 20*16);
API_END();
// API_DISPLAY();
// API_CMD_SWAP();
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

Storing the Static Content

When the code above completes, the co-processor will have generated a display list in RAM_DL beginning at RAM_DL + 0. The application must now store these items in RAM_G so that they can be recalled later. Note that this technique is storing the resulting display list items and not the co-processor commands themselves.

REG_CMD_DL is a register which the co-processor updates as it adds items to RAM_DL and points to the next available location in RAM_DL. Since the display list began at RAM_DL + 0, on completion of the command above, the value in REG_CMD_DL represents the size of the list.

A MEMCPY is used to copy the list to RAM_G. The 1000 is the start address of the destination and so the data will be copied to RAM_G + 1000. The RAM_DL is the start address of the source and so the list will be copied from RAM_DL + 0. The dlOffset is the number of bytes to copy.

Note that the destination address 1000 was chosen arbitrarily here. The application should however keep a note of the starting address of the stored section (1000), the length (dlOffset) and ending address (1000 + dlOffset) so that other items do not accidentally overwrite this list.

```
dlOffset = EVE_MemRead32(REG_CMD_DL);
API_LIB_BeginCoProList();
API_CMD_MEMCPY(1000, RAM_DL, dlOffset);
API_LIB_EndCoProList();
API_LIB_AwaitCoProEmpty();
```

Using the Static Content

Now, the building block above can be used in the final screen. The first part of the code is simply to adjust the variable for the gauge pointer and toggle position so that the gauge constantly ramps up and down in value and the toggle variable also changes with the up/down direction. A short delay (not shown) prevents the screen updating too quickly for the user to observe.

```
while(1)
{
    if (GaugeVal == 100)
    {
        GaugeDir = 0;
        Toggle = 0;
    }
    else if (GaugeVal == 0)
    {
        GaugeDir = 0xFF;
        Toggle = 65535;
    }
    if(GaugeDir == 0)
        GaugeVal --;
    else
        GaugeVal ++;
```

The main screen is now created. Note that the CLEAR_COLOR_RGB and CLEAR commands are present here.

Copyright © Bridgetek Pte Ltd

The static part is added via the CMD_APPEND call which specifies the beginning address of the stored section and the length of the stored list, both of which were recorded above. Note that several CMD_APPEND commands could be used to add various building blocks to make up a full screen as shown in AN_356. The dynamic items are then added to the co-processor list in the usual way.

Finally, the DISPLAY and SWAP commands (which again were not present in the static part) are added and the final screen will be displayed.

```
        API_LIB_BeginCoProList();
        API_CMD_DLSTART();
        API_CLEAR_COLOR_RGB(255, 255, 255);
        API_CLEAR(1,1,1);

       ////////////////// Static Section ////////////////////

        // parameters 1000 and dlOffset were determined when storing the data above
        API_CMD_APPEND(1000, dlOffset);        // starting address and length

       ////////////////// Dynamic Section ////////////////////

        API_COLOR_RGB(255, 255, 255);
        API_CMD_GAUGE(400, 200, 60, 0, 4, 8, GaugeVal, 100);
        API_CMD_FGCOLOR(0xFF0000);
        API_CMD_TOGGLE(350, 400, 100, 23, 0, Toggle, "Down" "\xff" "Up");
        API_CMD_NUMBER(400, 300, 25, OPT_CENTER, Gauge Val);

        API_DISPLAY();
        API_CMD_SWAP();
        API_LIB_EndCoProList();
        API_LIB_AwaitCoProEmpty();
    }
```

Additional Information

It is important to note that the RAM_DL still defines the final screen content in exactly the same way as when creating screens in the other demos shown here. The Append command is copying sections of the stored display list into RAM_DL and it is still RAM_DL which is parsed by the GPU to define the screen. Any stored sections which are appended in a given screen must still be taken into account with regards to the overall display list size.

Due to the object oriented programming model of the FT8XX, the creation of screens is already very efficient and there are cases where refreshing the entire screen is as good as partial refreshing. Examples include screens where the majority of items are bitmaps (as these are already stored in RAM_G) and cases where the majority of the screen content changes with any screen update. However, the technique described in this demo can be very effective in screens with a lot of GPU-generated items where only a few values change and can reduce the workload of the MCU's SPI peripheral even further.

A more detailed description can be found in AN_340 and an example of utilising this technique in AN_356

Copyright © Bridgetek Pte Ltd

10 Taking Snapshots

10.1 APP_SnapShot2PPM

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 or module, to a PC where it can be converted and viewed and used in documents.

FTDI have a wide range of suitable USB-TTL cables (including the TTL-232R cables, FT234X cables and C232H cables) as well as modules such as the UM232H. The FTDI range of USB-UART ICs can also be mounted directly on the application PCB. The C232HD cable was used for uploading the images in this document.

It is necessary to have the MCU's UART initialised. In this case, this is achieved via a call to MCU_UART_Init which is un-commented in the main() function.

This function is a variation of the technique originally described in BRT_AN_007 but has two advantages; firstly, the snapshot is taken one line at a time and therefore requires a much smaller area of RAM_G. This helps to avoid conflicts between the snapshot image data and the other assets stored in RAM_G. Secondly; the code presented here formats the image into PPM format and sends over the UART. This format is supported by graphics tools such as GIMP directly and so running a conversion script is not necessary. Note that the full-screen snapshot technique can also be used with this library.

Each of the other demos have a call to APP_SnapShot2PPM() after they have completed rendering of the screen. This call is initially commented out but can be un-commented to take a snapshot.

The code first waits for a starting command from the host which is the ASCII value for capital S. This ensures that the PC is ready to receive the streamed data.

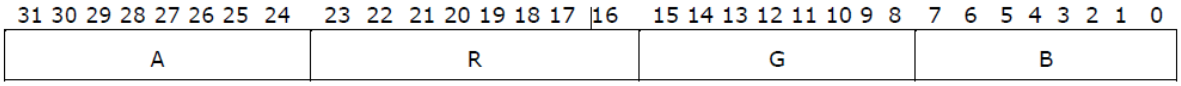
```
//Wait here until we receive starting code 0x53 over UART
StartCode = 0x00;
while(StartCode != 0x53)
{
    StartCode = MCU_UART_Rx();
}
```

A PPM file begins with a header which specifies the file type (P6) as well as the width and height of the image and a max colour value (8-bit resolution). These are separate by a whitespace.

```
MCU_UART_Tx(0x50); // P
MCU_UART_Tx(0x36); // 6
MCU_UART_Tx(0x0A); // LineFeed
MCU_UART_Tx(0x38); // 8
MCU_UART_Tx(0x30); // 0
MCU_UART_Tx(0x30); // 0
MCU_UART_Tx(0x0A); // LineFeed
MCU_UART_Tx(0x34); // 4
MCU_UART_Tx(0x38); // 8
MCU_UART_Tx(0x30); // 0
MCU_UART_Tx(0x0A); // LineFeed
MCU_UART_Tx(0x32); // 2
MCU_UART_Tx(0x35); // 5
MCU_UART_Tx(0x35); // 5
MCU_UART_Tx(0x0A); // LineFeed
```


Copyright © Bridgetek Pte Ltd

The snapshot2 command allows a window to be specified for the snapshot. Here, the application takes a snapshot of each row of the display in turn. For each row, it takes the snapshot with the format set to 0x20 for ARGB8 which offers 8-bits per colour via the following format.



The code sets a pointer value (0x104000) to near the end of RAM_G to avoid the snapshot data interfering with other assets loaded in the lower parts of RAM_G. The snapshot2 command will place the line of data beginning at this location, consisting of the above format for each of the 800 pixels in this case. The value must be far enough away from the end of the RAM_G to avoid overrunning the RAM.

A counter is then used to send the R, G and then B bytes to the UART for each pixel in turn. Therefore, each time around the (`While(RowNum < ScreenHeight)`) loop the routine will send 800 sets of RGB values in the order RGBRGBRGB ….

This process is repeated for each row on the display (480 times in this example)

```
Pointer =  1040000;    // Near end of RAM_G
RowNum = 0;
while(RowNum < ScreenHeight)  // < 480
{
    API_LIB_BeginCoProList();
    // 0x20 = ARGB8
    API_CMD_SNAPSHOT2(0x20, Pointer, 0, RowNum, (ScreenWidth*2), 1);
    API_LIB_EndCoProList();
    API_LIB_AwaitCoProEmpty();

    MCU_Delay_20ms();       // ensure co-pro has finished taking snapshot

    SerialCounter = 0;
    while(SerialCounter < (ScreenWidth*4))
    {
        ColorData = EVE_MemRead32(Pointer + SerialCounter);

        SerByteR = (uint8_t) ((ColorData >> 16));
        SerByteG = (uint8_t) (ColorData >>8);
        SerByteB = (uint8_t) (ColorData);

        MCU_UART_Tx(SerByteR);
        MCU_UART_Tx(SerByteG);
        MCU_UART_Tx(SerByteB);

        SerialCounter = SerialCounter + 4; // increment address
    }
    RowNum ++;
}
```

The resulting serial data will contain the 15 bytes of the PPM header followed by 3 * ScreenWidth * ScreenHeight bytes. In this case, it will be 15 + (3 *800 * 480) = 1152015 bytes.

Copyright © Bridgetek Pte Ltd

This demo uses 57600 baud as it is a common rate available on most terminals but if using a C232H 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 Hardware (RTS/CTS) flow control is strongly recommended to ensure that no data is lost.

To use hardware flow control, the application on the PC should set RTS/CTS flow control mode. If using an existing terminal program, this option will be available under the flow control settings menu. If writing an application using the D2xx driver, the FT_SetFlowControl call can be used to set this mode when setting the port up after opening.

On the MCU side, the implementation of flow control varies depending on the MCU and may be a feature of the UART block or may need to be implemented by GPIO. It generally uses two additional pins which connect to the RTS and CTS lines of the C232HM cable. Example connections are shown in the schematic in BRT_AN_008. In its simplest form, the lines can be used in the following way.

- The MCU will have a pin configured as an input, connected to the RTS output of the C232HM. The MCU should check this line before sending each byte and should only send a byte when the line is low. It should pause if the line is high as the C232HM buffer cannot accept further characters. Since a high volume of data flows in this direction in this particular application, the flow control will ensure that data is not lost if the PC is delayed in reading the data over USB and therefore the C232HM buffer fills.

- The MCU will have a pin configured as an output, connected to the CTS input of the C232HM. The MCU should set this line low when it can accept data from the C232HM. In this case, the only data is the single character "S" for the start command. The line must be low in order for the C232HM to send this character to the MCU as a high level will cause the C232HM to stop sending.

Refer to the following sections for details of how to receive this data on the PC using a terminal or a custom application.

10.2 APP_SnapShot2PPM - 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_008. 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 is entirely responsible for determining suitability of the software, for compliance with any licensing requirements, and for any consequences resulting from its use.

Copyright © Bridgetek Pte Ltd

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_Snapshot2PPM function. It will now be waiting for a start character.

Open the terminal program such as MTTTY_D2XX and 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".

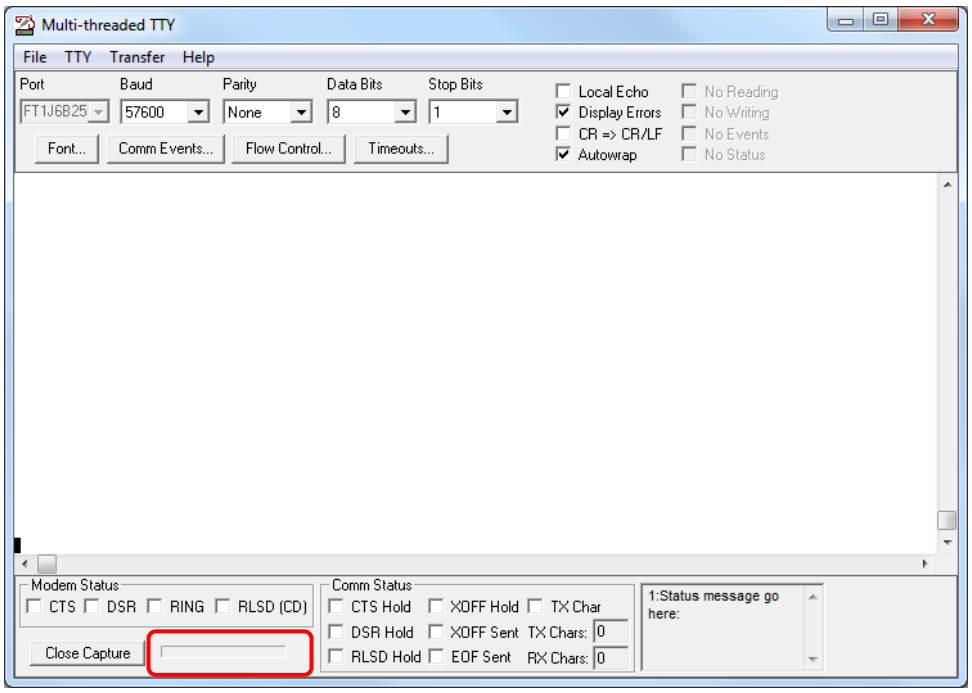


Figure 20 - MTTTY_D2XX Terminal

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

The bar graph (circled above) 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 .ppm. For example, "MySnapshot.ppm"

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 10.4 for details of how to use the file.

Copyright © Bridgetek Pte Ltd

10.3 APP_SnapShot2PPM – VB.Net Uploader

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 10.2.

First, the program FT81X SnapShot2 Reader.exe can be copied onto a convenient place such as the PCs desktop.

The steps in section 10.1 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.

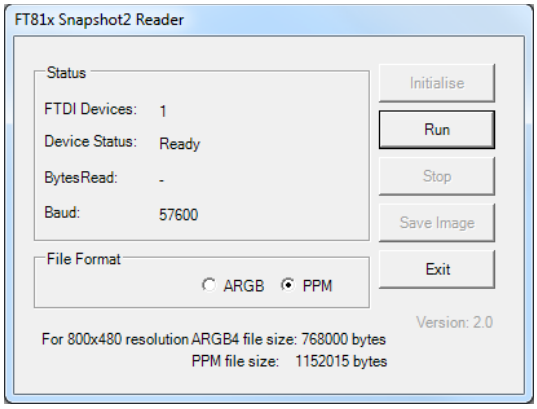


Figure 21 - Application after the attached cable is initialised

Ensure the PPM button is selected in the File Format window. 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.

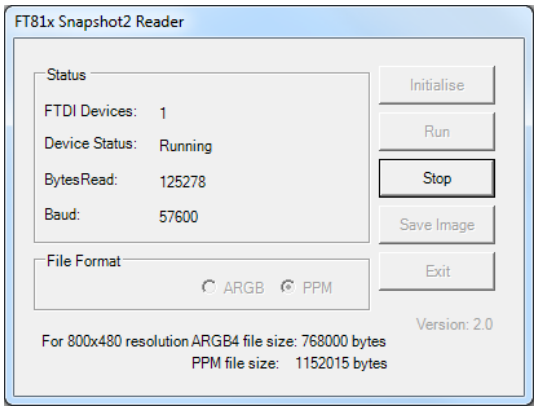


Figure 22 - Application running and collecting data

The Bytes Read counter will update as the bytes are received and can be compared to the total number expected.

As discussed in section 10.1 (and as noted on the program window itself as a reminder) the expected size is 1152015 at which point the value should stop incrementing. This will be different if the resolution or screen area or file format is changed. If the counter stops incrementing at an unexpected value (either too high or too low), check that the MCU is providing the correct baud rate. Once all bytes are received, the counter will stop incrementing and the Stop button can be pressed.

Copyright © Bridgetek Pte Ltd

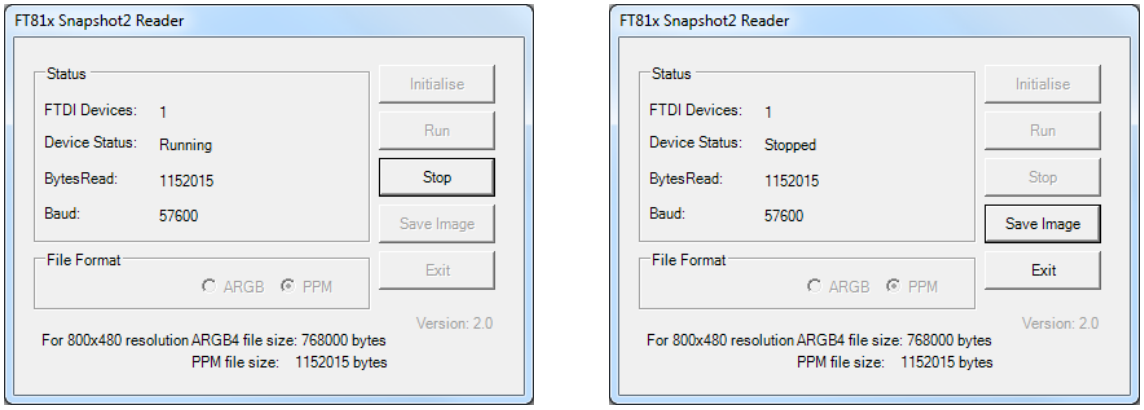


Figure 23 - Application being stopped after data collection

The Save Image button opens the Save As dialog allowing the location and name to be specified. The program automatically puts the .ppm extension onto the file.

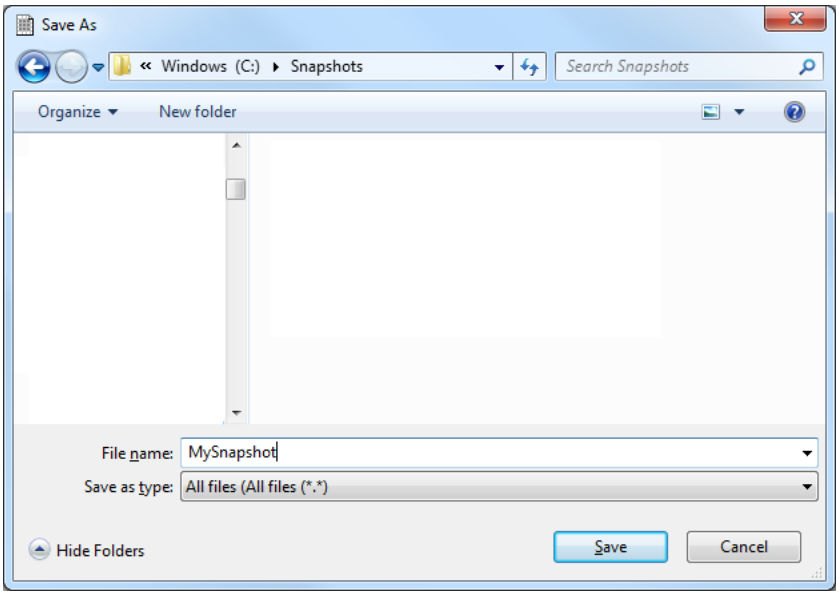


Figure 24 - Saving the .ppm file

The file can be opened shown in section 10.4.

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.

10.4 Using the PPM files

Once the PPM file has been uploaded using either of the methods shown above, it can be opened directly with a photo editing program. One program which supports PPM files and also offers a wide range of editing and exporting options is GIMP. The image can be opened by starting the GIMP application and then dragging the .PPM file into the window. Once opened, the file can be copied and pasted into a document in Microsoft Word etc. or the File -> Export option can be used to save the file as a format such as JPEG which will be accepted by most other applications.

Copyright © Bridgetek Pte Ltd

Note that if the screen is set to a different orientation using CMD_SETROTATE, the image may require to be rotated and flipped after uploading using GIMP or a similar tool.

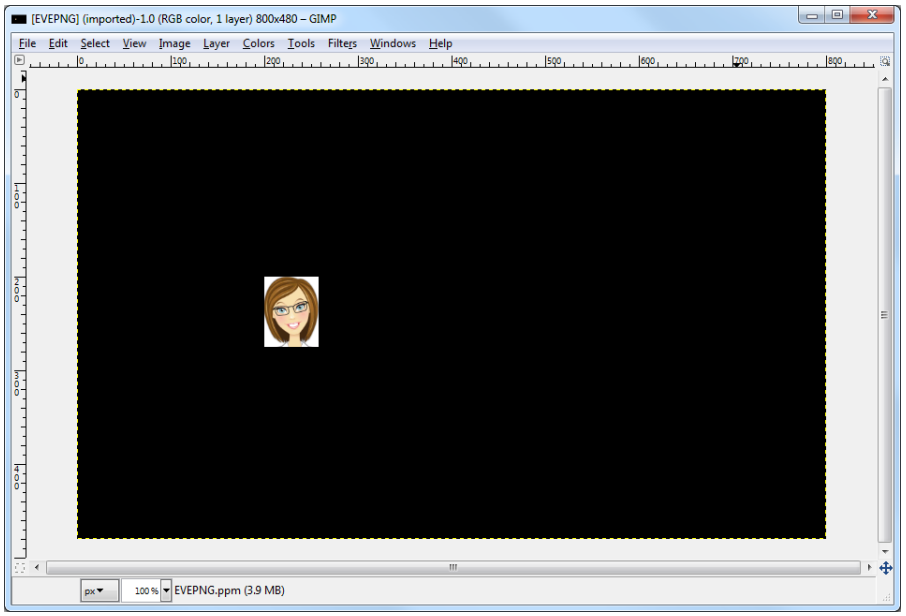


Figure 25 – Opening the PPM file

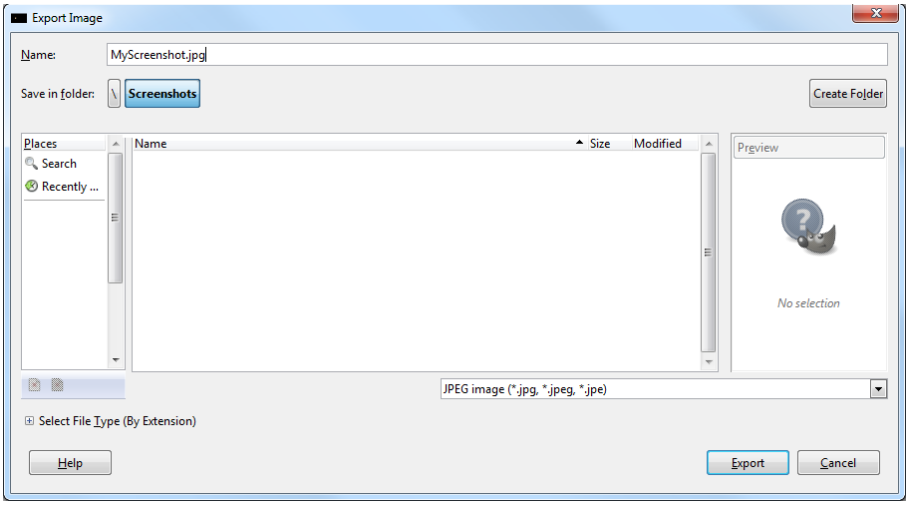


Figure 26 – Saving as JPEG

Note: FTDI/BridgeTek cannot accept responsibility for, or provide technical support for, 3rd party software. The user is entirely responsible for determining suitability of the software, for compliance with any licensing requirements, and for any consequences resulting from its use.

Copyright © Bridgetek Pte Ltd

11 Using the Demo Application

The provided zip file contains the full MPLABx project, including the C and header files from BRT_AN_008.

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_014_Source.zip (see Appendix A– References) and browse to the MPLAB folder. This can be copied to the user's normal project workspace directory. This is often c:\Users\[username]\MPLABXProjects\

Then go to File -> Open Project and select BRT_AN_014_Source. The project should now appear in the Projects window.

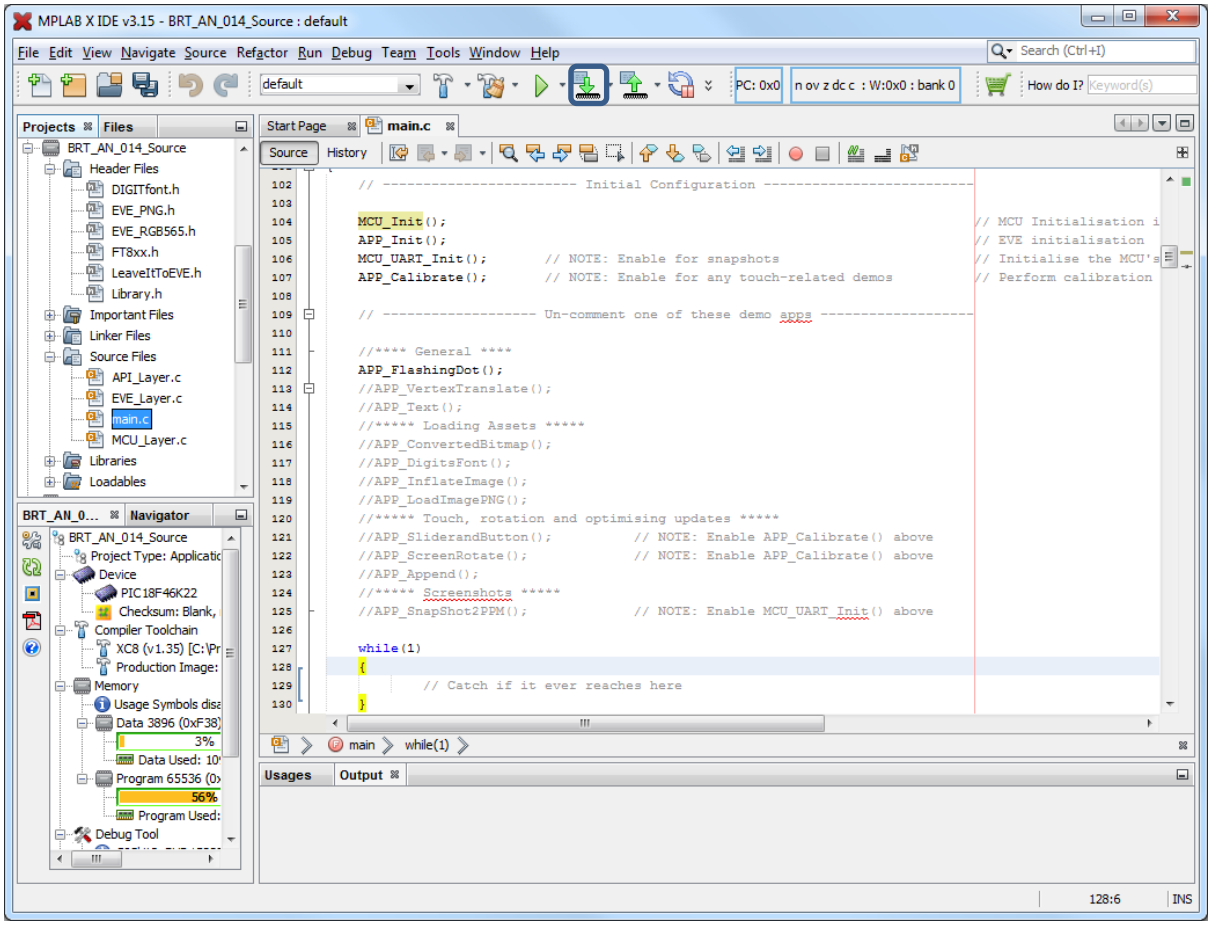


Figure 27 - MPLABX screenshot

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(). In the example below, APP_FlashingDot is enabled.

The comments beside the demo function indicate if the UART or Calibration routines are needed. It is recommended to enable all four functions shown below from the Set-up section and then select one of the functions from the Demo section.

```
// ------------------- Set-up functions -------------------

MCU_Init();             // MCU Initialisation including I/O, oscillator and SPI
APP_Init();             // EVE initialisation
MCU_UART_Init();        // NOTE: Enable for snapshots
```


Copyright © Bridgetek Pte Ltd

```
APP_Calibrate();       // NOTE: Enable for any touch-related demos

// ------------------- Demo Functions-------------------

//**** General ****
APP_FlashingDot();
//APP_VertexTranslate();
//APP_LineStrip();
//APP_Text();
//APP_DigitsFont();
//APP_ConvertedBitmap();
//APP_InflateImage();
//APP_LoadImagePNG();
//APP_SliderandButton();          // NOTE: Enable APP_Calibrate() above
//APP_ScreenRotate();             // NOTE: Enable APP_Calibrate() above
//APP_Append();
```

Note: The demo `APP_SnapShot2PPM();` is normally called from within another demo function rather than being enabled here, because the demo would initially create screen content which was to be captured in a 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 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.

Copyright © Bridgetek Pte Ltd

12 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_008 can be extended to use many of the features of the device. The code can be developed in a variety of ways.

If porting to another type of MCU, it is suggested to port the BRT_AN_008 code which includes a simple flashing dot example first. Once this is running, the code from main.c provided in this application note can be added. The header files for the image and font data from this application note can also be added, with any necessary change to the address in MCU Flash depending on where the data Flash resides in the MCU's memory map.

In the main code, the simple examples provided can be used along with the other commands available in the FT81X Programmers Guide (see Appendix A– References) to create a full application.

The full source of the library layers is also provided allowing them to be optimised for a particular MCU or additional features added.

Copyright © Bridgetek Pte Ltd

13 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.

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Bridgetek Pte Ltd (BRT Chip) devices incorporated in their systems, meet all applicable safety, regulatory and system-level performance requirements. All application-related information in this document (including application descriptions, suggested Bridgetek devices and other materials) is provided for reference only. While Bridgetek has taken care to assure it is accurate, this information is subject to customer confirmation, and Bridgetek disclaims all liability for system designs and for any applications assistance provided by Bridgetek. Use of Bridgetek devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless Bridgetek from any and all damages, claims, suits or expense resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder. Bridgetek Pte Ltd, 178 Paya Lebar Road, #07-03, Singapore 409030. Singapore Registered Company Number: 201542387H.

Copyright © Bridgetek Pte Ltd

Appendix A– References

Document References

BRT_AN_014_Source Code

BRT_AN_008

BRT_AN_006

BRT_AN_007

FT81X Product Page

FT81X Series Programmer Guide

FT81X Datasheet

ME813-WH50C Datasheet

ME812-WH50R Datasheet

VM810C50A-D

EVE Examples

PIC18F46K22

PICKIT3

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

Copyright © Bridgetek Pte Ltd

Appendix B – List of Tables & Figures

List of Tables

NA

List of Figures

Copyright © Bridgetek Pte Ltd

Appendix C– Revision History

Document Title: BRT_AN_014 FT81X Simple PIC Library Examples
Document Reference No.: BRT_000162
Clearance No.: BRT#128
Product Page: http://brtchip.com/i-ft8/
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial release	2018-06-11

Copyright © Bridgetek Pte Ltd