Application Note

AN_314

FT800 ADVANCED TECHNIQUES - WORKING WITH BITMAPS

Document Reference No.: FT_001017

Version: 1.0

Issue Date: 2013-03-17

Future Technology Devices International Limited (FTDI)
Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
Web Site: http://ftdichip.com

Contents

1 Introduction

This application note is written to explain the usage of bitmaps within an FT800 based design. After reading this application note the reader should be able to control the FT800 to display a bitmap image in a variety of formats and understand the difference between different formats. The app note will also give advice on bitmap transformations and when to use different formats.

1.1 Scope

The document will give the basic understanding about the FT800 and its use of bitmaps. Any non-FTDI tools that are used in this document will not be taught here.

2 Bitmaps

2.1 What is a Bitmap?

A bitmap is basically a digital picture. The raw data of the bitmap is stored as an array of bytes which define the colour of the pixel at each point in the image. Also stored in the bitmap file is the size and shape of the bitmap.

2.2 Where is a Bitmap Stored?

In the context of the FT800 the bitmap must be less than 512 x 512 pixels as that is the maximum resolution the device can handle. Also of importance is how the image is stored. The internal graphics RAM of the FT800 is 256k Bytes and as such any images stored on chip must fit into this memory size.

FTDI provide a tool for converting .png or .jpg images to the raw format that the FT800 requires to render the image. This raw data may be stored in your processor flash as a decompressed image that is copied to the FT800 graphics RAM at system initialisation for use in the BITMAPS primitive call.

Compressed images may also be stored on or off chip, but the decompressed version must always fit into the graphics RAM. The FT800 has an algorithm CMD_INFLATE to decompress such files from a ZLIB format.

3 Bitmap Formats

Figure 3-1 shows a standard 64 x64 pixel bitmap.

Figure 3-1 Jigsaw.png file

This means the image is 64 pixels wide and 64 pixels high.

The FT800 does not read .png files directly so they need to be converted to a compatible format with the free image conversion utility form the FTDI website.

Formats that the FT800 can accept are: ARGB1555, L1, L4, L8, RGB332, ARGB2, ARGB4, RGB565, PALETTED, TEXT8X8, TEXTVGA, BARGRAPH.

The different formats result in different levels of resolution / clarity of image. Lower resolution images save memory, but may not appear as clear as the higher resolution images.

This document will discuss the use of the ARGB1555, L1, L4, L8, RGB332, ARGB2, ARGB4, RGB565 and PALETTED output from the image conversion tool.

The bitmap data array is organised on a row by row layout.

3.1 ARGB1555

The converted image in ARGB1555 format appears as Figure 3-2 This image has 1 bit for Alpha data and 5 bits for each of the colours RGB.

Figure 3-2 ARGB1555

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: ARGB1555
Stride: 128
total size: 8192
```

This translates to needing 8192 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 2 bytes hence the stride is 64 x 2 = 128 bytes.

If the raw data is viewed then bytes 0 to 127 are the 64 pixels on line 1, bytes 128 to 255 are the 64 pixels on line 2 etc.

To decode the pixel information:

ARGB1555 format layout		Byte Order
A	Bit 15	Byte 1 Byte 0
R	Bit 14-10	
G	Bit 9- 5	
B	Bit 4-0	

Figure 3-3 ARGB1555 Pixel Data

3.2 L1

The converted image in L1 format appears as Figure 3-4. This format is monochrome.

Figure 3-4 L1

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: L1
Stride: 8
total size: 512
```

This translates to needing 512 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 1 bit hence the stride is 1 bit x 64 = 64 bits (8 bytes).

If the raw data is viewed then bytes 0 to 7 are the 64 pixels on line 1, bytes 8 to 15 are the 64 pixels on line 2 etc.

To decode the pixel information:

L1 format layout		Alignment
Pixel 0	Bit 7	Byte 0
Pixel 1	Bit 6	
.....		
Pixel 7	Bit 0	

Figure 3-5 L1 Pixel Data

This is the lowest resolution/quality format. Suitable for small or simple images. Note the lack of colour definition as each pixel is either on or off.

3.3 L4

The converted image in L4 format appears as Figure 3-6. This format is monochrome.

Figure 3-6 L4

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: L4
Stride: 32
total size: 2048
```

This translates to needing 2048 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 4 bits hence the stride is 0.5 byte x 64 = 32 bytes.

If the raw data is viewed then bytes 0 to 31 are the 64 pixels on line 1, bytes 32 to 63 are the 64 pixels on line 2 etc.

To decode the pixel information:

L4 format layout		Alignment
Pixel 0	Bit 7-4	Byte 0
Pixel 1	Bit 3-0	

Figure 3-7 L4 Pixel Data

3.4 L8

The converted image in L8 format appears as Figure 3-8. This format is monochrome.

Figure 3-8 L4

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: L8
Stride: 64
total size: 4096
```

This translates to needing 4096 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 1 byte hence the stride is 1 byte x 64 = 64 bytes.

If the raw data is viewed then bytes 0 to 63 are the 64 pixels on line 1, bytes 64 to 127 are the 64 pixels on line 2 etc.

To decode the pixel information:

L8 format layout		Alignment
Pixel 0	Bit 0-7	Byte 0
pixel 1	Bit 15-8	Byte 1
pixel 2	Bit 23-16	Byte 2

Figure 3-9 L8 Pixel Data

3.5 RGB332

The converted image in RGB332 format appears as Figure 3-10. This image has 3 bits for Red and Green and 2 bits for blue data

Figure 3-10 RGB332

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: RGB332
Stride: 64
total size: 4096
```

This translates to needing 4096 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 1 byte hence the stride is 1 byte x 64 = 64 bytes.

If the raw data is viewed then bytes 0 to 63 are the 64 pixels on line 1, bytes 64 to 127 are the 64 pixels on line 2 etc.

To decode the pixel information:

RGB332 pixel layout		Byte Order
R	Bit 7-5	Byte 0
G	Bit 4-2	
B	Bit 1-0	

Figure 3-11 RGB332 Pixel Data

3.6 ARGB2

The converted image in ARGB2 format appears as Figure 3-12. This image has 2 bit for Alpha data and for each of the colours RGB.

Figure 3-12 ARGB2

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: ARGB2
Stride: 64
total size: 4096
```

This translates to needing 4096 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 1 byte hence the stride is 1 byte x 64 = 64 bytes.

If the raw data is viewed then bytes 0 to 63 are the 64 pixels on line 1, bytes 64 to 127 are the 64 pixels on line 2 etc.

To decode the pixel information:

ARGB2 format layout		Byte Order
A	Bit 6-7	Byte 0
R	Bit 4-5	
G	Bit 2-3	
B	Bit 0-1	

Figure 3-13 ARGB2 Pixel Data

3.7 ARGB4

The converted image in ARGB4 format appears as Figure 3-14. . This image has 4 bits for Alpha data and for each of the colours RGB.

Figure 3-14 ARGB4

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: ARGB4
Stride: 128
total size: 8192
```

This translates to needing 8192 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 2 bytes hence the stride is 2 bytes x 64 = 64 bytes.

If the raw data is viewed then bytes 0 to 127 are the 64 pixels on line 1, bytes 128 to 255 are the 64 pixels on line 2 etc.

To decode the pixel information:

ARGB4 format layout		Byte Order
A	Bit 15-12	Byte 1
R	Bit 11-8	
G	Bit 7-4	Byte 0
B	Bit 3-0	

Figure 3-15 ARGB4 Pixel Data

3.8 RGB565

The converted image in ARGB2 format appears as Figure 3-16. . This image has 5 bits for red and blue data and 6 bits for green.

Figure 3-16 RGB565

As raw binary data the output is structured as an array of bytes:

```
Resolution: 64 'x' 64
Format: ARGB2
Stride: 128
total size: 8192
```

This translates to needing 8192 bytes of free space in RAM to store the data.

The height of the image is 64 pixels and the width is 64 pixels.

Each pixel requires 2 bytes hence the stride is 2 bytes x 64 = 64 bytes.

If the raw data is viewed then bytes 0 to 127 are the 64 pixels on line 1, bytes 128 to 255 are the 64 pixels on line 2 etc.

To decode the pixel information:

RGB565 format layout		Byte Order
R	Bit 15-11	Byte 1 Byte 0
G	Bit 10-5	
B	Bit 4-0	

Figure 3-17 RGB565 Pixel Data

Note this format includes alpha data as well as colour data.

4 How the FT800 Displays a Bitmap

The FT800 uses a display list to create displays. For a bit map the key elements are:

`BITMAP_SOURCE()`
This is the location where the bitmap is stored. Typically in graphics RAM

`BITMAP_LAYOUT(Format,stride,height)`
This is used by the FT800 to read the image data array correctly.
The format allows the FT800 to decode each byte correctly while the stride and height help to define where a line begins and ends in the array.

`BITMAP_SIZE(filter, wrapx, wrapy, width, height)`
The Bitmap size helps define the size of the image on the display. Note it does not scale the image but essentially creates a window to determine how much is visible.

Using the FTDI Hardware Abstraction Layer (HAL) code to display the ARGB1555 image at coordinates (10, 10) would be of the form.

```
ft_void_t Jigsaw_ARGB1555_demo()
{
	/*Jigsaw ARGB1555*/
	/*('file properties: ', 'resolution ', 64, 'x', 64, 'format ', 'ARGB1555',
	'stride ', 128, ' total size ', 8192)*/

	/* Copy raw data into address 0 followed by generation of bitmap */
	Ft_Gpu_Hal_WrMemFromFlash(phost, RAM_G,&JigsawARGB1555, 8192);

	Ft_Gpu_CoCmd_Dlstart(phost);
	Ft_App_WrCoCmd_Buffer(phost, CLEAR(1, 1, 1)); // clear screen
	Ft_App_WrCoCmd_Buffer(phost,COLOR_RGB(255,255,255));
	Ft_App_WrCoCmd_Buffer(phost,BITMAP_SOURCE(RAM_G));
	Ft_App_WrCoCmd_Buffer(phost,BITMAP_LAYOUT(ARGB1555, 128, 64));
	Ft_App_WrCoCmd_Buffer(phost,BITMAP_SIZE(NEAREST, BORDER, BORDER, 64, 64));
	Ft_App_WrCoCmd_Buffer(phost,BEGIN(BITMAPS)); // start drawing bitmaps
	Ft_App_WrCoCmd_Buffer(phost,VERTEX2II(10, 10, 0, 0));
	Ft_App_WrCoCmd_Buffer(phost,END());
	Ft_App_WrCoCmd_Buffer(phost, DISPLAY() );

	Ft_Gpu_CoCmd_Swap(phost);

	/* Download the commands into fifo */
	Ft_App_Flush_Co_Buffer(phost);

	/* Wait till coprocessor completes the operation */
	Ft_Gpu_Hal_WaitCmdfifo_empty(phost);

	Ft_Gpu_Hal_Sleep(3000);//timeout for snapshot to be performed by coprocessor

}
```

4.1 Filter

Typically when drawing a textured shape on the screen, the texture is not displayed exactly as it is stored, without any distortion. Filtering can improve the appearance of the final display and the FT800 can perform this filtering based on 2 different algorithms.

The filter element in the Bitmap size command has two options, NEAREST or BILINEAR. This will improve the appearance of an image especially if it is displayed smaller or larger than the stored image e.g. after it has been scaled.

If the NEAREST option is selected this is the simplest filtering algorithm based on the nearest pixel.

If the BILINEAR option is selected then a bilinear filtering algorithm is applied to points between pixels to smooth the display. This filtering has a greater smoothing affect than NEAREST.

4.2 Wrap

The FT800 will allow for wrapping an image in either the x (Wrapx) or the y (Wrapy) direction.

There are two settings for this parameter, either BORDER or REPEAT.

If the BORDER setting is applied then the wrapping is simply confined to the single image as defined by the bitmap height and width.

If the REPEAT setting is applied it allows the image to be repeated. Suitable for creating a tiled view with a single bitmap image.

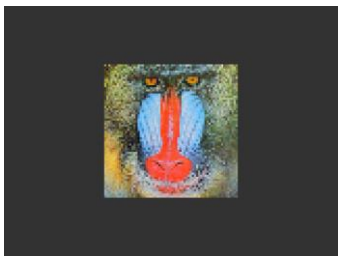
The image below is a 64 x 64 pixel image, but the drawn area size is 160 x 120. With the REPEAT wrapping the image is repeated to fill a 160 x 120 area

```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT(RGB565, 128, 64) );
dl(  BITMAP_SIZE(NEAREST,  REPEAT,
REPEAT, 160, 120) );
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(0, 0, 0, 0) );
```

Figure 4-1 REPEAT wrapping to create a tiled display

Note: If wrapx or wrapy is set to REPEAT then the corresponding memory layout dimension (BITMAP_LAYOUT line stride or height) must be power of two, otherwise the result is undefined

5 Bitmap Manipulation



Some display solutions require the bitmap to be scaled or rotated. The FT800 provides transform functions to assist with this. The transforms are based on matrix transforms as used in Open GL. This can be achieved with sending commands to the FT800 main graphics processor or via the graphics co-processor.

There are six basic transforms each effectively an element in a matrix array.

Figure 5-1 Original image

$$m = \begin{bmatrix} BITMAP\_TRANSFORM\_A & BITMAP\_TRANSFORM\_B & BITMAP\_TRANSFORM\_C \\ BITMAP\_TRANSFORM\_D & BITMAP\_TRANSFORM\_E & BITMAP\_TRANSFORM\_F \end{bmatrix}$$

These can be accessed directly via the BITMAP_TRANSFORM_x display list command, where x is a letter for A-F, or indirectly by making use of co-processor commands such as CMD_SCALE or CMD_ROTATE.

5.1 Rotating

Rotation of a bitmap may be achieved with the co-processor command CMD_ROTATE and this is the simplest mechanism to code with. Rotation is by default centred around the top left corner of the image. And has a resolution of 1/65536 of a circle.

To use CMD_ROTATE the identity matrix must first be loaded. CMD_ROTATE then updates the bitmap transform matrix, before CMD_SETMATRIX loads the new parameters into the display list for the graphics engine to action.

```
BEGIN(BITMAPS)
cmd_loadidentity();
cmd_rotate();
cmd_setmatrix();
```

This is essentially updating BITMAP_TRANSFORM A-F at a higher level of programming.

To rotate the bitmap counter clockwise by 33 degrees wrt top left of the bitmap:

```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotate(-33 * 65536 / 360);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

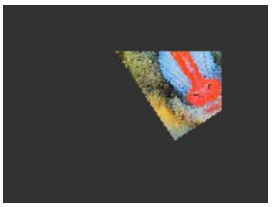


Figure 5-2 Rotated around top left image

Note the effect of rotating around a corner looks like the image is in a picture frame which has vertical and horizontal edges. Only the parts of the rotated image inside the picture frame are visible.

If a bitmap is to be rotated around the centre of the original image then a translate function must be used before the rotate. The translation should be 0.5 x width in the x direction and 0.5x height in the y-direction.

e.g. Rotating a 64 x 64 bitmap around its center:

```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(65536 * 32, 65536 * 32);
cmd_rotate(90 * 65536 / 360);
cmd_translate(65536 * -32, 65536 * -32);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

Figure 5-3 Rotated around centre image

5.2 Resizing

To resize or scale an image to zoom in or out, it is recommended to use the CMD_SCALE instruction.

This is again a higher level mechanism for accessing the bitmap transform matrix elements A-F, making coding simpler.

As with the rotating, the function operates around the top left corner of the image by default. To scale around the centre the image must first be translated by be 0.5 x width in the x direction and 0.5x height in the y-direction.

To zoom in by a factor of 2 around the centre requires an instruction set as below:

```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(65536 * 32, 65536 * 32);
cmd_scale(2 * 65536, 2 * 65536);
cmd_translate(65536 * -32, 65536 * -32);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

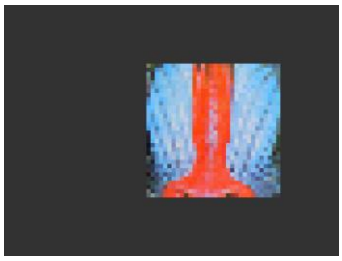


Figure 5-4 scaled by x2 from centre image

6 Animation and Bitmap Cells

Animation in its simplest form is simply a matter of redefining the xy coordinates that an object will be placed at. However with bitmaps you may want to modify the image displayed e.g. a walking man or a flickering flame.

To simplify coding it is possible to join more than one bitmap together and treat each one as a cell within one bitmap.

To be able to use this technique each of the individual bitmaps must be of the same shape and size. It is also important that the individual images are joined vertically.

e.g.

Each of the fire bin images are 22 pixels wide and 52 pixels high. The final composite bitmap that is stored in memory is 22 pixels wide and 4 x 52 (208) pixels high.

When specifying the layout:

```
BITMAP_LAYOUT(format,Bitmap_Stride,height)
```

The stride is defined by the width and the image format.
The height is for one of the images (52 pixels).

When specifying the size:

```
BITMAP_SIZE(filter, wrapx, wrapy, height)
```

Figure 6-1 Fire image strip

The height must be for the entire bitmap (208 pixels)

It the bitmap is now given a handle it is possible to treat each of the four bitmaps as a cell within the main image.

```
BITMAP_HANDLE(FIRE_HANDLE)

BEGIN(BITMAPS));
VERTEX2II( x, y, FIRE_HANDLE, 0)
VERTEX2II( x, y, FIRE_HANDLE, 1)
VERTEX2II(x, y, FIRE_HANDLE, 2)
VERTEX2II(x, y, FIRE_HANDLE, 3)
END(BITMAPS)
```

X and y are the coordinates where the image is displayed. The values can be varied in the case of an object walking across the screen or use a time delay to update the image if the same location is used.

The last element is the cell 0,1,2,3

7 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1,2 Seaward Place, Centurion Business Park
Glasgow G411HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com
Web Site URL http://www.ftdichip.com
Web Shop URL http://www.ftdichip.com

Branch Office – Tigard, Oregon, USA

7130 SW Fir Loop
Tigard, OR972123
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales) us.sales@ftdichip.com
E-Mail (Support) us.support@ftdichip.com
E-Mail (General Enquiries) us.admin@ftdichip.com
Web Site URL http://www.ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan , R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

E-mail (Sales) tw.sales1@ftdichip.com
E-mail (Support) tw.support1@ftdichip.com
E-mail (General Enquiries) tw.admin1@ftdichip.com
Web Site URL http://www.ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited (China)
Room 408, 317Xianxia Road,
Shanghai, 200051
China
Tel: +86 2162351596
Fax: +86 2162351595

E-mail (Sales) cn.sales@ftdichip.com
E-mail (Support) cn.support@ftdichip.com
E-mail (General Enquiries) cn.admin@ftdichip.com

Distributor and Sales Representatives

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

8 Appendix A– References

8.1 Document References

1. datasheet for VM800C
2. datasheet for VM800B
3. FT800 programmer guide FT_000793.
4. FT800 Embedded Video Engine Datasheet FT_000792

8.2 Acronyms and Abbreviations

Terms	Description
Arduino Pro	The open source platform variety based on ATMEL's ATMEGA chipset
EVE	Embedded Video Engine
SPI	Serial Peripheral Interface
UI	User Interface
USB	Universal Serial Bus

9 Appendix B – List of Tables & Figures

List of Figures

Appendix C– Revision History

Document Title: AN_314 FT800 Advanced Techniques - Working with Bitmaps

Document Reference No.: FT_001017

Clearance No.: FTDI# 392

Product Page: http://www.ftdichip.com/FTProducts.htm

Document Feedback: Send Feedback

Revision	Changes	Date
1.0	First Release	2014-06-09