Application Note

AN_308

FT800 I2C Example with 8-bit MCU

Version 1.0

Issue Date: 2014-03-03

The FTDI FT800 video controller offers a low cost solution for embedded graphics requirements. In addition to the graphics, resistive touch inputs and an audio output provide a complete human machine interface to the outside world.

This application note will provide a simple example of developing MCU code to control the FT800 over I^2C. The principles demonstrated can then be used to produce more complex applications.

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

Table of Contents

1 Introduction

The FT800 operates as a peripheral to the main system processor and provides graphic rendering, sensing of display touch stimulus, as well as audio capabilities.

The device is controlled over a low bandwidth SPI or I^2C interface allowing interfacing to nearly any microcontroller with a SPI or I^2C master port. Simple and low-pin-count microcontrollers can now have a high-end, human machine interface (HMI) by using the FT800.

This application note will demonstrate how a simple 8-bit MCU can be used to initialize the FT800 over I^2C and then easily generate a display.

In this case, the Freescale MC9S08QE8 microcontroller in 16-pin DIP package is used but the firmware can be easily ported over to other types of MCU by changing only the low level functions which access the hardware registers.

The example can be used as the basis for a larger project by changing the main application section which creates the command lists for the FT800. By doing so, a full application containing many graphics objects (lines, shapes, text etc.) and widgets (sliders, dials, buttons etc.) can be created.

In addition to providing a starting point for application development, this example also demonstrates that a low end 8-bit MCU can be used to drive a 5” color screen because the FT800’s internal graphics processor takes much of the hard work away from the MCU.

Note: This application note includes example source code. This code is provided as an example only and FTDI accept no responsibility for any issues resulting from its use. The developer of the final application is responsible for ensuring the correct and safe operation of the equipment incorporating any part of this sample code.

2 FT800 I²C Interface

This section gives an overview of the I²C communications required to interface the MCU to the FT800 and produce a simple display.

The FT800 device supports either SPI or I²C protocols. The required protocol should be selected before powering up via the mode pin on the FT800. Please refer to Section 8 for further details of the hardware. The remainder of this document assumes that the FT800 is being used in I²C mode. The application note AN_259 (see Appendix A – References) has a similar example for SPI.

It is also assumed that the FT800's I²C address is 0x23 (as the hardware configuration of the EVE Click PCB will select this address). The address is defined at the top of the provided source code and can be changed if the FT800 has a different I²C address.

2.1 Programming Model

It will be noted that the programming model for the FT800 is the same regardless of whether I²C or SPI is used and so the applications themselves are very similar. For example, writing the REG_HCYCLE register still involves sending the same sequence of bytes. The differences all exist in the lower level functions which sit between the main application and the MCU I²C or SPI peripheral module.

SPI uses the Chip Select line to select the FT800 device as the active slave and to separate individual read and write transactions. Because I²C has only one clock and one bi-directional data line shared between all slaves, the I²C protocol requires several additional features in its protocol such as Start, Stop and Repeated Start line conditions, an addressing phase with a Read/Write bit and a ninth bit in each transfer used as an acknowledgement.

2.2 Example I²C Transactions

This section gives examples of the typical I²C transactions used to communicate with the FT800. As with the SPI examples, the FT800 interfacing is almost entirely carried out via reading and writing memory addresses, corresponding to registers of FIFO locations, and sending a few specific host commands.

Note: The FT800's I²C address is 0x23 on the EVE Click (Section 8 has more details).

2.2.1 Writing a data value to a memory address

This method is used when writing to a register or FIFO location. The code section below writes a 16-bit value to the HCYCLE register:

```
// Write REG_HCYCLE to 548
FT800_I2C_SendAddressWR(REG_HCYCLE);
FT800_I2C_Write16(548);
```

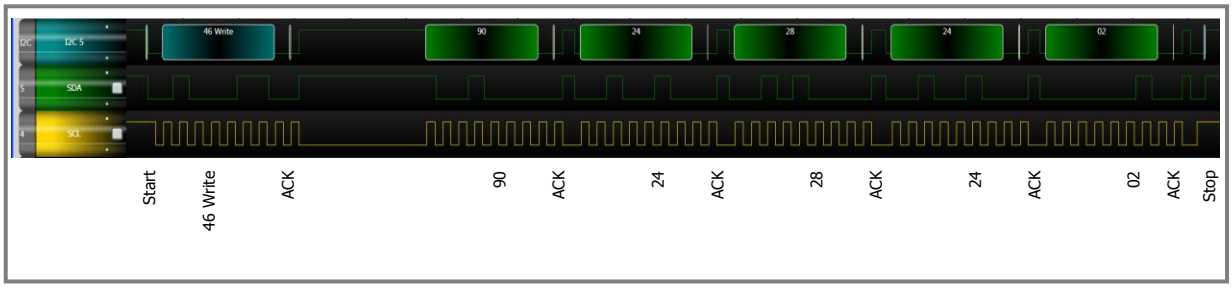


Figure 2.1 Writing a 16-bit register

1. Send the I²C Start condition
2. Send the FT800's I²C address with R/W bit = 0 (write)
   Check that the FT800 ACKs
   Note: 0x23 (00100011) plus R/W = 0 gives 01000110 which is 0x46
3. Send address of memory location to be written with upper 2 bits forced to '10'
   Check that the FT800 ACKs each byte
   Address 0x102428 (10010000 00100100 00101000) becomes 0x902428
4. Send the least significant byte of the data to be written (0x24)
   Check that the FT800 ACKs
   548dec = 0x0224
5. Send the next most significant byte of the data to be written (0x02)
   Check that the FT800 ACKs
   548dec = 0x0224
6. Send the I²C Stop Condition

Note that the same technique is used for writing other data sizes. For example, writing a 32-bit register would have four bytes written between steps 3 and 6. Some widgets have additional parameters (e.g. when writing the slider command to the co-processor FIFO, there are several parameters after the 4-byte command itself resulting in 18 bytes of data after the address).

2.2.2 Reading a data value from a memory address

This method is used when reading data from a memory address. The code section below reads a 32-bit value from REG_CMD_WRITE:

```
// Read the value of the REG_CMD_WRITE register
FT800_I2C_SendAddressRD(REG_CMD_WRITE);
cmdBufferWr = FT800_I2C_Read32();
```

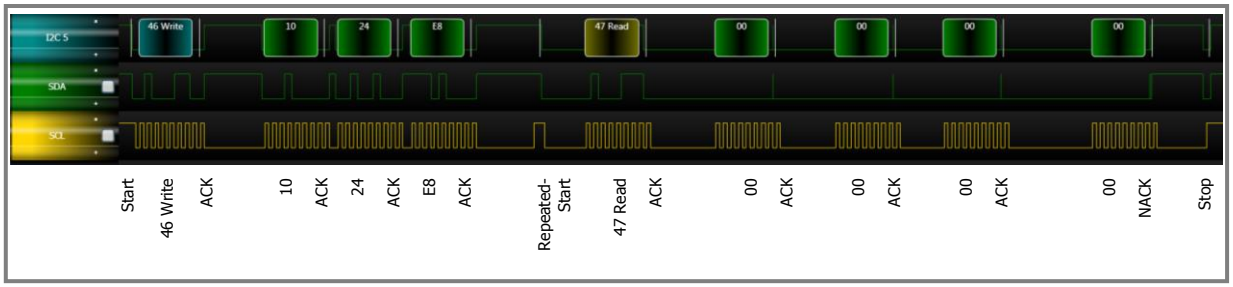


Figure 2.2 Reading a 32-bit register

1. Send the I^2C Start condition
2. Send the FT800's I^2C address with R/W bit = 0 (write) (writing register address first)
   Check that the FT800 ACK's
   0x23 (00100011) plus R/W = 0 gives 01000110 which is 0x46
3. Send address of memory location to be read (upper 2 bits forced '00' indicates read)
   Check that the FT800 ACKs each byte
   Address 0x1024E8 (00010000 00100100 11101000) remains 0x1024E8
4. Send a Repeated Start condition
5. Send the FT800's I^2C address with R/W bit = 1 (read) (now reading the register)
   Check that the FT800 ACK's
   0x23 (00100011) plus R/W = 1 gives 01000111 which is 0x47
6. Read the first byte from the FT800
   Send an ACK since this is a 32-bit read and there are three more bytes to read
   This will be the least significant byte of the 32-bit value
7. Read the second byte from the FT800
   Send an ACK since this is a 32-bit read and there are two more bytes to read
   This will be the 2nd least significant byte of the 32-bit value
8. Read the third byte from the FT800
   Send an ACK since this is a 32-bit read and there is one more byte to read
   This will be the 2nd most significant byte of the 32-bit value
9. Read the fourth byte from the FT800
   Send a NAK since this is the last byte to be read
   This will be the most significant byte of the 32-bit value
10. Send the I^2C Stop Condition

Note that same technique is used for reading 8 and 16-bit values. For example, to read an 8-bit value, step 6 above would read the byte and reply with a NAK. The Stop condition would then be sent.

3 Software Architecture

This example code has three main sections:

- User Application
  - Initialization of MCU / FT800 / Display (see section 4.1)
  - Main Application (see section 4.2)
- FT800 I^2C Functions (see section 5)
- Hardware-Specific Functions (see section 6)

The User Application is responsible for performing the initial configuration of the MCU and FT800 (through the Hardware Specific layer) and is then responsible for running the main MCU application, where it can create the lists of commands to send to the FT800 to draw the different screens. The creation of the specific graphics for the users' applications is the element that will change the most from the examples given in this application note. FTDI Chip has developed a Programming Guide that is dedicated to the explanation of this graphic creation operation.

The FT800_I2C_ functions are called by the User Application, and translate the commands such as FT800_I2C_Write32(…) into the actual byte data values which will be sent to the FT800. They then call the HAL_ functions to send/receive these data bytes to/from the FT800.

The Hardware-Specific (HAL_) functions are the only part of the code which access the specific registers of the MCU. These functions can be replaced with equivalent ones written for a different type of MCU (e.g. PIC Microcontroller). By doing so, little or no changes to the User Application and FT800 I^2C Functions will be needed.

In the examples throughout this application note, the data highlighted in green is coming from the MCU to the FT800 and data highlighted in purple is from FT800 to MCU. Red is used to highlight particular bits relevant to the surrounding discussion.

The diagram below shows an example of writing a 16-bit register and reading an 8-bit register. It demonstrates the way in which the FT800_ functions call the HAL_ functions which in turn put the values and line conditions on the I²C bus.

User Application	Hardware specific Functions	I²C Bus
`// Write REG_HCYCLE to 548`		
`FT800_I2C_SendAddressWR(REG_HCYCLE);` `(0x102428)`	`HAL_I2C_AddrFT800wr(FT800_I2C_Slave_Address);` `(0x23)`	START 0x46 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x90)` `(false)`	0x90 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x24)` `(false)`	0x24 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x28)` `(false)`	0x28 ACK
`FT800_I2C_Write16(548, TRUE);`	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x24)` `(false)`	0x24 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x02)` `(true)`	0x02 ACK STOP
`// Reading the value of REG_ID`		
`FT800_I2C_SendAddressRD(REG_ID);` `(0x102400)`	`HAL_I2C_AddrFT800wr(FT800_I2C_Slave_Address);` `(0x23)`	START 0x46 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x10)` `(false)`	0x10 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x24)` `(false)`	0x24 ACK
	`HAL_I2C_Write(I2C_Writebyte, LastByte);` `(0x00)` `(false)`	0x00 ACK
	`HAL_I2C_RepeatStart();`	REPEAT_START
	`HAL_I2C_AddrFT800rd(FT800_I2C_Slave_Address);` `(0x23)`	0x47 ACK
`chipid = FT800_I2C_Read8();`	`HAL_I2C_Read(FirstByte, LastByte);` `(true)` `(true)`	0x7C NAK STOP
Note: REG_ID has value 0x7C		

Figure 3.1 Example of function calls and actual I²C bus values

The functions used in the sample code provided are listed below.

FT800 I²C Functions
void FT800_I2C_SendAddressWR(dword);
void FT800_I2C_SendAddressRD(dword);
dword FT800_I2C_Read32(void);
byte FT800_I2C_Read8(void);
void FT800_I2C_Write32(dword, Bool);
void FT800_I2C_Write16(word, Bool);
void FT800_I2C_Write8(byte, Bool);
void FT800_I2C_HostCommand(byte);
void FT800_I2C_HostCommandDummyRead(void);
word FT800_IncCMDOffset(word, byte);

Hardware-Specific Functions
void HAL_Configure_MCU(void);
void HAL_I2C_AddrFT800wr(byte);
void HAL_I2C_AddrFT800rd(byte);
void HAL_I2C_Write(byte, Bool);
byte HAL_I2C_Read(Bool, Bool);
void HAL_I2C_RepeatStart(void);
void HAL_PDlow(void);
void HAL_PDhigh(void);
void Delay(void);
void DelayShort(void);

Table 3.1 Functions provided in the sample code

Section 7 further discusses the way in which a final application might use these functions.

4 User Application

The main section of the sample program will first initialize the FT800 and then idle in a main loop which will be used to display the graphics required by the application.

There are two methods to create a screen to be displayed.

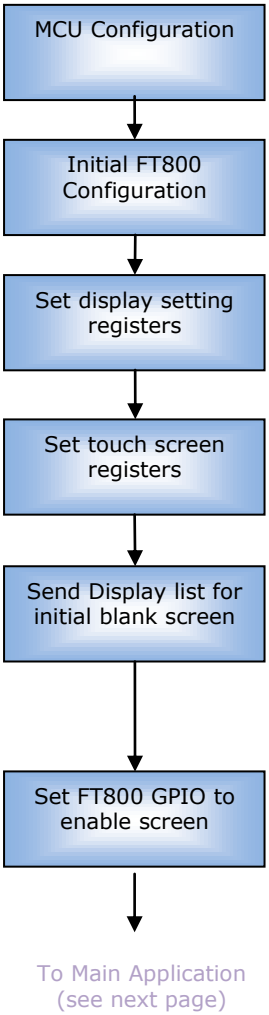
- The first method is to write display list commands directly to the RAM_DL (Display List RAM). Note: This method is illustrated when blanking the screen as part of the initialization of the FT800 as discussed in section 4.1.

- The second method is to write a series of Co-Processor commands or display list commands to the RAM_CMD (Command FIFO). The Co-Processor then creates the display list in RAM_DL based on the commands which it is given in the RAM_CMD FIFO. This method makes it easier to combine the drawing of primitive graphics objects (lines etc.) and Widgets (slider etc.) on the same screen.
  Note: This method is illustrated when creating the main screens in the Main Application in section 4.2.

Although it is in theory possible to mix both methods when creating a new display list (screen), this requires care on the MCU side since both the MCU and the co-processor will be writing to RAM_DL and the MCU must ensure that no unintended overlap occurs. This application note and associated sample code are intended to focus on the I^2C communication and so use only Display List or Co-Processor list for each screen.

4.1 Initialization of MCU / FT800 / Display

This code carries out the initialization of the MCU and the FT800; including setting the FT800's display setting registers to match the LCD used.



Configures the MCU I/O Ports and I^2C Interface by writing directly to the registers. Also disables the watchdog timer in this application. All configuration is carried out within the HAL_Configure_MCU function

Powers up the FT800 and then sends commands over I^2C to configure the FT800's oscillator settings and reset the FT800. This is followed by reading the FT800's ID register – reading the expected value of 0x7C confirms that the FT800 is ready and responding correctly.

Write the display registers of the FT800 to configure it for the particular display which is attached. The code supplied configures the FT800 for the 5" display attached to the EVE Click in this case. Each register is configured with a write of a 16-bit value to its address.

The touch screen threshold is set here. The touch screen is not used in this application note but some of the later code examples use the touch feature.

Create an initial display list which simply blanks the screen. This code writes three 4-byte commands to successive locations in the Display list RAM.

- [RAM_DL + 0] Specify the colour which will be used when the screen is cleared
- [RAM_DL + 4] Specify that the Colour, Stencil and Tag buffers should be cleared
- [RAM_DL + 8] Display command which signifies the end of the Display List
- Writing to the DL_Swap register then tells the FT800 to render the above display

The FT800 has its own GPIO port which can be controlled by writing to the FT800's GPIO_DIR and GPIO registers over I^2C. This part of the code writes to these registers to assert the display's enable pin which is connected to the FT800 GPIO. Please consult the EVE Click datasheet for further details of the GPIO connections on this module

Figure 4.1 Initialization flow chart

4.2 Main Application

In this example, the main loop simply cycles between two different static screens (display lists), one with a small white circle drawn on a back background and another with the same circle but coloured red. This could be expanded by drawing more shapes/text/widgets or by adding animation by moving or gradually changing the color of the circle.

As discussed previously, the main application screens are created using the Co-Processor (RAM_CMD) method. For each screen, the MCU creates a list of commands for the Co-Processor inside the FT800 and then tells the Co-Processor to execute them. The Co-Processor creates the Display List in RAM_DL.

A variable in the main loop is toggled each time round, so that each time the screen is drawn the colour of the circle can alternate between Red and White.

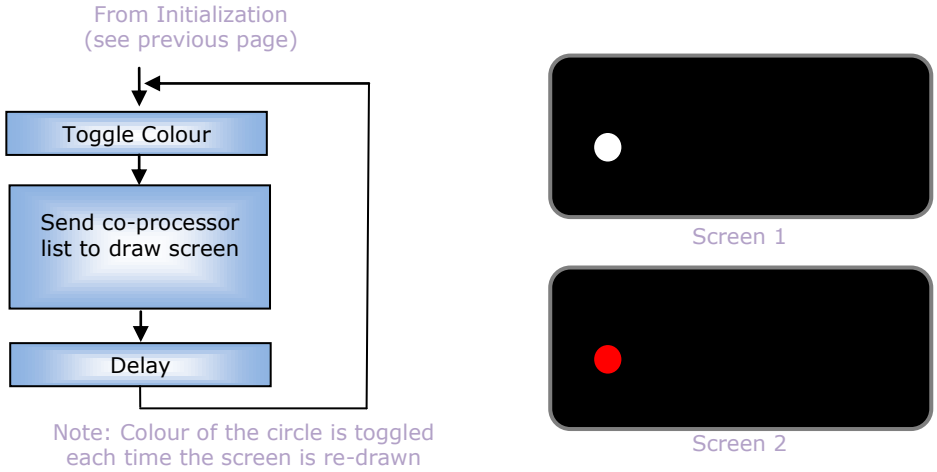


Figure 4.2 Main Application loop flow chart

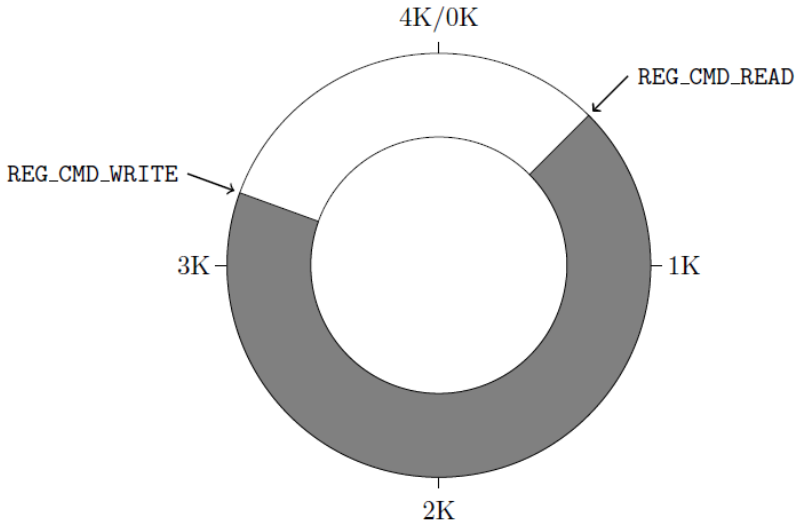
4.2.1 Creating a Coprocessor Command List

The code used to create a Co-Processor command list uses the following steps:

Step 1

The first step is to wait for the Co-Processor to finish executing the previous (if any) command list it was given. The FT800 provides two registers which help to monitor the FIFO status. Both registers show an offset with respect to the starting address of the Command FIFO as opposed to an absolute address.

- REG_CMD_READ is updated by the Co-Processor as it executes commands stored in the Command FIFO to indicate the address (offset) at which it is currently sitting.
- REG_CMD_WRITE is updated by the MCU to tell the Co-Processor where the last valid instruction ends.



Note: This FIFO is mapped at FT800 memory addresses 108000h (RAM_CMD) to 108FFFh (RAM_CMD + 4095)

Figure 4.3 Co-Processor Command FIFO

When REG_CMD_READ = REG_CMD_WRITE, the Co-Processor has executed all commands from the command FIFO.

```
do
{
       FT800_I2C_SendAddressRD(REG_CMD_WRITE);        //
       cmdBufferWr = FT800_I2C_Read32();              //

      FT800_I2C_SendAddressRD(REG_CMD_READ);          //
      cmdBufferRd = FT800_I2C_Read32();               //
} while(cmdBufferWr != cmdBufferRd);
```

Step 2

The FT800 uses a circular buffer / FIFO to hold its command list. When a new list is to be created, the MCU will write the commands starting from the next available location (i.e. the current value of REG_CMD_WRITE). This has already been read from the FT800 by the code above and so the value of REG_CMD_WRITE is copied into a variable as the starting index.

```
CMD_Offset = cmdBufferWr;       // Get current value of the CMD_WRITE pointer
```

Step 3

The first command in the Command List can now be written to this offset in the RAM_CMD.

The FT800_I2C_SendAddressWR function is used to send the address which the following data will be written to. In this case, it is the starting address of the FIFO (RAM_CMD) plus the offset within the FIFO. The command itself is then written, which is in this case the DL_START command (CMD_DLSTART in the FT800 Programmers Guide), by calling FT800_I2C_Write32. Commands are always multiples of 4 bytes for the FT800.

This operation has written four bytes into the FIFO. A command list would typically consist of many commands and so the MCU must now update its own offset value so that it knows the offset at which it will start writing the next command. The FT800_IncCMDOffset function does this. In most cases, it just adds the length of the last command to the previous offset but also handles the case

where the index reaches 4095 and must be wrapped back to 0 due to the circular nature of the FIFO.

```
FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset); // Writing to next location in FIFO
FT800_I2C_Write32(0xFFFFFF00, TRUE);           // Write the DL_START command

CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Move the CMD Offset
```

The next command now starts from the offset determined above

```
FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset); // Writing to next location in FIFO
FT800_I2C_Write32(0x02000000, TRUE);           // Clear Color RGB to black

CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Move the CMD Offset
```

This can be repeated to create the full command list. Only two commands are shown here because the other eight commands use exactly the same technique. The full list of commands used to draw the sample screen is listed in section 4.2.2.

Note: The code shown in this step is creating the actual objects to be drawn on the screen. Additional commands can easily be added to draw shapes, text, and widgets such as sliders.

Because at the start of this command list, the code waited until the READ and WRITE pointers were equal, the circular FIFO is effectively empty. Therefore, up to 4062 bytes (4096 – 4 to avoid reaching back to the start) worth of commands can be added.

Step 4

It is important to note that the REG_CMD_READ and REG_CMD_WRITE are still unchanged from their values in Step 1. The MCU has simply written commands into successive FIFO locations.

For example, the REG_CMD_READ and REG_CMD_WRITE may both have been 1000(decimal) in Step 1. The code in Step 3 has now added 10 x 4-byte instructions = 40 bytes.

In Step 4, the MCU writes REG_CMD_WRITE to be 1040(decimal). The Co-Processor detects that REG_CMD_WRITE > REG_CMD_READ and now reads (and executes) instructions from the FIFO until REG_CMD_READ reaches REG_CMD_WRITE again.

```
FT800_I2C_SendAddressWR(REG_CMD_WRITE);        //
FT800_I2C_Write16(CMD_Offset, TRUE);           //
```

The commands are only now being read and executed and therefore the display will not show this new screen until this command in Step 4 has been sent.

Note: It is also possible to update REG_CMD_WRITE after every individual command is written to the FIFO in Step 3 and in this case the Co-Processor will execute each command in turn. The Co-Processor updates the value of REG_CMD_READ as it works its way through the locations in the FIFO. However, updating REG_CMD_WRITE between writing each command and / or reading the new value of REG_CMD_READ will increase the amount of I^2C Traffic. A full discussion of the various ways to load and execute the commands is beyond the scope of this application note.

4.2.2 Drawing the application screen

To draw the screen, the process shown in section 4.2.1 is used to send the following ten commands.

Note: For clarity, only the actual command values are shown here. The complete code for creating the command list will require all steps shown in section 4.2.1. The full code listing can be found in the source code file provided with this application note (see Appendix A – References)

```
…
FT800_I2C_Write32(0xFFFFFF00);          // Write the DL_START command
…
FT800_I2C_Write32(0x02000000);          // Clear Color RGB
…
FT800_I2C_Write32(0x26000007);          // Clear
…
FT800_I2C_Write32(0x04FFFFFF);          // Color RGB (FFFFFF = white)
…
FT800_I2C_Write32(0x0D0000FF);          // Point Size
…
FT800_I2C_Write32(0x1F000002);          // Begin
…
FT800_I2C_Write32(0x43000880);          // Vertex 2F
…
FT800_I2C_Write32(0x21000000);          // End
…
FT800_I2C_Write32(0x00000000);          // Display
…
FT800_I2C_Write32(0xFFFFFF01);          // Swap
…
```

The same set of commands, as shown above, is repeatedly sent as a new co-processor list to the FT800 but the parameters of the ColorRGB command are toggled between 0xFFFFFF (white) and 0xFF0000 (red) each time.

A delay was also added between each write of the new co-processor list so that the toggling does not happen too quickly for the user to see.

Note: The hex values have been intentionally used here so that byte values can be related those on the I2C bus but a larger application would normally define the values e.g. Display can be defined as 0x00000000.

5 FT800 I²C Functions (FT800_)

This section describes the functions which read or write values to the FT800's registers. They handle the formatting of the data bytes etc. They do not contain any MCU-specific code, which allows them to be used on a variety of MCUs. Instead, they call the lower level HAL_ functions for the actual I²C communication.

To form a complete I²C transaction (e.g. a read from or a write to a register), the main application uses two separate types of functions: addressing phase and data phase. These have intentionally been kept separate in this example for ease of reading and because having functions which do the addressing and data phase together would require a large number of functions to cover all combinations of read/write and 8/16/32-bit. Some examples include:

Reading the 8-bit REG_ID register…

```
FT800_I2C_SendAddressRD(REG_ID);          // Send the address of the REG_ID
chipid = FT800_I2C_Read8();               // Read the value of the REG_ID
```

Writing a 16-bit value to the HCYCLE register…

```
FT800_I2C_SendAddressWR(REG_HCYCLE);      // Send the register address
FT800_I2C_Write16(548, TRUE);             // Send the 16-bit value
```

When adding commands to the Co-Processor FIFO, the MCU uses a variable (CMD_Offset) to keep track of the current position in the FIFO. This determines the address in the FIFO at which the next command should be written. After adding all of the commands for a Co-Processor list to the FIFO, this value is also written to the REG_CMD_WRITE register in the FT800 to indicate the end address of the current list to the Co-Processor.

```
CMD_Offset = cmdBufferWr;  // Start @ current FIFO position read from FT800 earlier

FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);  // First loc in CMD FIFO
FT800_I2C_Write32(0xFFFFFF00, TRUE);            // Write DL_START command

CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Add 4 to the offset

FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);  // Next loc in CMD FIFO
FT800_I2C_Write32(0x02000000, TRUE);            // Clear Color RGB

CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Add 4 to the offset
…
```

Some widgets such as sliders have parameters of more than four bytes. The use of separate Address and Read/Write functions allows various sizes of data to be read and written without having functions for every specific data size. For example, adding a slider command to the co-processor FIFO below:

```
…
FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);   // Next loc in CMD FIFO
FT800_I2C_Write32(0xFFFFFF10, FALSE);            // CMD Slider
FT800_I2C_Write16(0x0080, FALSE);                // Coordinates
FT800_I2C_Write16(0x0028, FALSE);                // Coordinates
FT800_I2C_Write16(0x000F, FALSE);                // Width
FT800_I2C_Write16(0x00B0, FALSE);                // Height
FT800_I2C_Write16(0x0000, FALSE);                // Options
FT800_I2C_Write16(0x8000, FALSE);                // Value
FT800_I2C_Write16(0xFFFF, TRUE);                 // Range
CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 20); // Add 20 to the offset
…
```

Note: The TRUE parameter causes the Write function to send an I²C Stop after the write. The size has been rounded up from 18 to 20 so that the next command starts at a multiple of 4 bytes

5.1 Send Address functions

These functions send the address which is to be written or read. They take a dword parameter which should have the memory address in the lower three bytes.

Note that they send an I^2C Start condition since they will be the first operation in each I^2C transaction but they do not include an I^2C Stop condition. The Stop condition will be sent at the end of the subsequent data read or write operation.

void FT800_I2C_SendAddressWR(dword Memory_Address)

This function is called to address the FT800 and specify the register in the FT800 to which a subsequent write will take place.

It first calls the HAL_I^2C_AddrFT800wr function, to send the Start condition followed by the FT800's I^2C address. The R/W bit is 0 since the first step is to *write* the register address. Assuming the FT800 has I^2C address 0x23, the result will be (00100011 << 1) | 0x00 which is 01000110.

The function then takes the register address passed in, and configures the upper two bits of this 24-bit address to '10' to indicate to the FT800 that a *write of data* to this address will follow. Since the address is 3 bytes but a dword was passed into the function, the upper byte is ignored. This modified register address is then sent to the FT800 one byte at a time using the HAL_I2C_Write function (Most Significant Byte first).

e.g. FT800_I2C_SendAddressWR(0x102428); where FT800's I2C address is 0x23

```
Value passed in:        00000000 00010000 00100100 00101000          (dword)
I2C Bus values:         S
                        0100011 0 ACK
                        10010000 ACK 00100100 ACK 00101000 ACK
Value returned:         N/A
```

The application code can now call the data write function detailed in section 5.2 to send the data for the data phase.

void FT800_I2C_SendAddressRD(dword Memory_Address)

This function is called to address the FT800 and specify the register in the FT800 to which a subsequent read will take place.

This function first calls the HAL_I2C_AddrFT800wr function, to send the Start condition followed by the FT800's I^2C address with R/W = 0 for a write. The R/W is 0 since the first step is to *write* the register address. Assuming the FT800 has I^2C address 0x23, the result will be (00100011 << 1) | 0x00 which is 01000110.

The function then takes the register address passed in, and configures the upper two bits of this 24-bit address to '00' to indicate to the FT800 that a *read of data* from this address will follow. Since the address is 3 bytes but a dword was passed into the function, the upper byte is ignored. This modified register address is then sent to the FT800 one byte at a time using the HAL_I2C_Write function (Most Significant Byte first).

A Repeated Start condition is now sent onto the I^2C bus so that an I^2C read phase can begin, where the FT800 is sending data and the MCU is the receiving node.

After the Repeated Start, the FT800's I^2C address is sent again in preparation for the reading of the data from the previously addressed register. This time, the I^2C address has R/W configured for reading and so has a '1' value. Assuming the FT800 has I^2C address 0x23, the result will be (00100011 << 1) | 0x01 which is 01000111.

e.g. FT800_I2C_SendAddressRD(0x102428); where FT800's I²C address is 0x23

```
Value passed in:          00000000 00010000 00100100 00101000          (dword)
I2C Bus values:           S
                          0100011 0 ACK
                          00010000 ACK 00100100 ACK 00101000 ACK
                          SR
                          0100011 1 ACK
Value returned:           N/A
```

The application code can now call the data read function detailed in section 5.3 to send the data for the data phase.

5.2 Write Data Functions

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

They do not send a Start condition as they follow on immediately from the preceding address function, which must be called first to tell the FT800 where the data is to be written to.

In addition to the data value to be written, the write functions take a second parameter which determines whether an I^2C Stop condition will be sent after the last byte. This is normally set 'true' in the example code provided since the example is just writing 8/16/32 bit values. However, when writing co-processor lists which involve drawing widgets it is sometimes necessary to write more than 4 bytes of data before the stop condition (e.g. a slider has 18 bytes including command and parameters).

Prerequisites: Before writing data, the address must be specified by using the FT800_I2C_SendAddressWR function in the previous section.

void FT800_I2C_Write32(dword I2CValue32, bool SendStop)

This function sends a 32-bit data value to the FT800. It does not make any change to the 32-bit data passed in, and it passes the data to the HAL_I2C_Write function one byte at a time starting with the least significant byte. If the SendStop parameter is 'TRUE', then the last (fourth) call to HAL_I2C_Write will also request an I^2C Stop condition.

As detailed in the HAL_I2C_Write section later, the HAL_I2C_Write function takes two parameters; the byte to be written and a Boolean value indicating whether this is the last data byte of the current transaction. This allows the HAL function to put the Stop condition on the I^2C line after the last byte to tell the FT800 that this write operation has ended.

e.g. FT800_I2C_Write32(0x12345678, TRUE);

```
Value passed in:          00010010 00110100 01010110 01111000          (dword)
I2C Bus values:           01111000 ACK
                          01010110 ACK
                          00110100 ACK
                          00010010 ACK
                          S                          (only if SendStop is TRUE)
Value returned:           N/A
```

void FT800_I2C_Write16(word I2CValue16, bool SendStop)

This function is very similar to the FT800_I2C_Write32 function, but writes a 16-bit value instead.

e.g. FT800_I2C_Write16(0x1234, TRUE);

```
Value passed in:          00010010 00110100                            (word)
I2C Bus values:           00110100 ACK
                          00010010 ACK
                          S                          (only if SendStop is TRUE)
Value returned:           N/A
```

void FT800_I2C_Write8(byte I2CValue8, byte SendStop)

This function is very similar to the FT800_I2C_Write32 and FT800_I2C_Write16 functions above, but writes an 8-bit value instead.

e.g. FT800_I2C_Write16(0x12, TRUE);

```
Value passed in:          00010010                                      (byte)
I2C Bus values:           00010010 ACK
                          S                          (only if SendStop is TRUE)
Value returned:           N/A
```

5.3 Read Data Functions

These functions can be used to read a value from a 32-bit or 8-bit register.

They do not send a Start condition as they follow on immediately from the preceding address function. Since they mark the second half of the transaction (i.e. the data phase) they do however send a Stop condition once the required number of bytes has been read.

Because the MCU is reading the data in these cases, the FT800 is the device sending the data and the MCU is the device generating the ACK/NAK bit. The MCU will respond to a byte with an ACK if it wants to read another byte next, or will respond with an ACK if it does not want to read any other byte in this transaction.

Prerequisites: Before reading data, the address should be specified by using the FT800_I2C_SendAddressRD function in the previous section.

dword FT800_I2C_Read32()

This function reads a 32-bit data value from the FT800. During a read of a register, the FT800 sends the least-significant byte first. This function reads all four bytes and returns them to the calling function with the most significant byte of the register in the most significant position in the returned dword. By taking care of the little-endian format of the FT800, this function avoids the need for the main application to reverse the bytes.

This function calls the HAL_I2C_Read routine for each byte to be read. The HAL_I2C_Read returns the byte value read. HAL_I2C_Read also takes two Boolean parameters which specify if this is the first or last byte to be read. This allows the HAL_I2C_Read function to configure the MCU's I^2C peripheral accordingly. Further details can be found in section 6.2.

e.g. MyReadDword = FT800_I2C_Read32(); *(assume register being read has value 0x87654321)*

```
Value passed in:          N/A
I2C Bus values:           00100001 ACK
`                         01000011 ACK
                          01100101 ACK
                          10000111 NAK
                          S
Value returned:           10000111 01100101 01000011 00100001          (dword)
```

byte FT800_I2C_Read8()

This function is similar to the FT800_I2C_Read32 function above but reads only 8 bits from the register.

e.g. MyReadByte = FT800_I2C_Read8(); *(assume register being read has value 0x21)*

```
Value passed in:          N/A
I2C Bus values:           00100001 NAK
                          S
Value returned:           00100001                                      (byte)
```

5.4 Host Command Functions

The FT800 also has a set of specific commands which are used during the start-up / configuration of the device. These have a slightly different format to the more addressing and data functions above and so separate functions were created to avoid making the ones above more complicated.

void FT800_I2C_HostCommand(byte Host_Command)

This function sends the specified Host Command to the FT800. The command itself is passed into the function as a byte.

A host command consists of an address phase (calling HAL_I2C_AddrFT800wr) followed by three data bytes (which constitute the command itself).

The first data byte has bits 7:6 set to '01' to indicate that this is a host command. The bits 5:0 contain the command itself. The second and third bytes are always zero. The function forces bits 7:6 of the upper byte to '01' in case the value passed in does not already have these set correctly.

e.g. FT800_I2C_HostCommand (FT_GPU_PLL_48M); *(FT_GPU_PLL_48M defined as 0x62)*
(FT800's I2C address assumed 0x23)

```
Value passed in:        01100010                                    (byte)
I2C Bus values:         S
                        0100011 0 ACK
                        01100010 ACK 00000000 ACK 00000000 ACK
                        P
Value returned:         N/A
```

void FT800_I2C_HostCommandDummyRead(void)

This function performs a dummy read of memory location 0x000000 which is used to wake up the FT800. After addressing the FT800 over I^2C, the function simply sends three bytes of 0x00. It therefore takes no parameters and does not return a value.

e.g. FT800_I2C_HostCommandDummyRead();

```
Value passed in:        N/A
I2C Bus values:         S
                        0100011 0 ACK
                        00000000 ACK 00000000 ACK 00000000 ACK
                        P
Value returned:         N/A
```

5.5 FIFO Management Functions

word FT800_IncCMDOffset(word Current_Offset, byte Command_Size)

This function is used when adding commands to the Command FIFO of the Co-Processor.

When a command is added to the FIFO at (RAM_CMD + Offset), the MCU must calculate the offset at which the next command would be written. Normally, if a 4-byte command was written at (RAM_CMD + Offset), then the next command would start at (RAM_CMD + Offset + 4).

However, since the FT800 uses a circular buffer of 4096 bytes, the offset also needs to wrap around when offset 4095 is reached. This is the reason for carrying out the increment in a function as opposed to simple adding a value in the main code.

The function takes in the current offset (starting offset where the last command had been written) and the size of the last command. It returns the offset at which the next command will be written.

e.g. if CMD_Offset = 0 beforehand,

```
FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next location in FIFO
FT800_I2C_Write32(0xFFFFFF00, TRUE);            // Write DL_START (4-byte command)
CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Update Offset
```

CMD_Offset now equals 4

As discussed previously, each command in the FIFO always starts at an address offset which is a multiple of 4. When using a command such as Slider, which has a total size of 18 bytes, the offset is incremented by 20 so that the address returned will start the subsequent command at an offset which is a multiple of 4.

```
…
FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);   // Next loc in CMD FIFO
FT800_I2C_Write32(0xFFFFFF10, FALSE);            // CMD Slider
FT800_I2C_Write16(0x0080, FALSE);                // Coordinates
FT800_I2C_Write16(0x0028, FALSE);                // Coordinates
FT800_I2C_Write16(0x000F, FALSE);                // Width
FT800_I2C_Write16(0x00B0, FALSE);                // Height
FT800_I2C_Write16(0x0000, FALSE);                // Options
FT800_I2C_Write16(0x8000, FALSE);                // Value
FT800_I2C_Write16(0xFFFF, TRUE);                 // Range
CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 20); // Add 20 to the offset
…
```

6 Hardware-Specific Functions (HAL_)

The example code provided separates the MCU-specific parts into a small set of functions. These functions access the actual registers in the MCU. This allows the example to be used on almost any MCU, by creating functions which perform the same task on the particular MCU chosen to replace the ones provided.

The code provided was developed for the MC9S08QE8 MCU from Freescale and should also be compatible with other Freescale MCUs, particularly those from the MC9S08 family.

When porting the code to an MCU from a different manufacturer, the main areas which require attention are these MCU-specific functions and also the data types used which could differ depending on the development environment for the chosen MCU.

The MCU-specific functions provided here all begin with HAL_ as they form a hardware abstraction layer which makes the other parts of the code independent of the actual hardware used.

Note: The I^2C communication has been implemented using the polling method to ease readability. Many MCUs including the MC9S08 can use interrupts and this may result in better efficiency. Some error checking/handling has also been omitted for code readability. The MCU may already have an I^2C library available which could be used instead of this lower level code.

Note: The functions in section 6.2 have some additional delays (no-operation/nop instructions and DelayShort calls) between I^2C operations which may not be required in other MCUs. It was also necessary to disable and re-enable the I^2C peripheral in between each individual read and write transaction but this may not be required on other MCUs.

6.1 Configuration Functions

void HAL_Configure_MCU(void)

This function is responsible for the following tasks:

- disabling the Watchdog timer (which was not required in this simple demonstration but can be enabled if required)
- Configuring the MCU port pins as described below
- Configuring the I^2C peripheral on the MCU

Note that port A is not used in this example, and are set to all inputs. The only connections to port A are the reset and background debug lines which share the port A pins.

Port B is configured as shown below. The MCU's I^2C module will take over control of bits 7 and 6 once enabled. The Power-Down signal (port B0) is initially set low so that the FT800 is powered down.

Port	Direction	Initial Value	Notes
Port B7	OUT	1	SCL
Port B6	OUT	1	SDA
Port B5	IN	1	Not used
Port B4	IN	1	Not used
Port B3	IN	1	Not used
Port B2	IN	1	Not used
Port B1	IN	1	Not used
Port B0	OUT	0	Power Down signal to FT800

Table 6.1 Port pin configuration

6.2 Address and Data Transfer Functions

void HAL_I2C_AddrFT800wr(byte FT800_I2C_Address)

This function is used to address the FT800 where the subsequent data will be written from the MCU to the FT800.

Prerequisites: None

This function does the following:

- Takes the FT800's I^2C address which has been passed in and calculates the 7-bit address with W/R value set to '0'. e.g. 0x23 = (00100011 << 1) | 0 = 01000110 = 0x46
- Enables the MCU's I^2C module
- Sets the MCU's I^2C module to be a Master and Transmit data. This generates an I^2C Start condition on the bus.
- Writes the address with R/W to the I^2C Data register (e.g. 0x46)
- Waits for the I^2C write to complete
- Checks if the I^2C module received an ACK from the FT800

void HAL_I2C_AddrFT800rd(byte FT800_I2C_Address)

This function is used to address the FT800 where the subsequent data will be read by the MCU from the FT800.

Prerequisites: Call HAL_I2C_AddrFT800wr to address FT800
Send register address with HAL_I2C_Write
Repeated Start

This function does the following:

- Takes the FT800's I^2C address which has been passed in and calculates the 7-bit address with W/R value set to '1'. e.g. 0x23 = (00100011 << 1) | 1 = 01000110 = 0x47
- Sets the MCU's I^2C module to be a Master and Transmit data. This generates an I^2C Start condition on the bus.
- Writes the address with R/W to the I^2C Data register (e.g. 0x47)
- Waits for the I^2C write to complete
- Checks if the I^2C module received an ACK from the FT800

Note that unlike the HAL_I2C_AddrFT800wr function above, this function does not enable the MCU's I^2C module. Since there has always been a call to HAL_I2C_AddrFT800wr and then sending of the register address before the HAL_I2C_AddrFT800rd has been called (so that the FT800 knows which register is to be read), the MCU's I^2C module has therefore already been enabled.

void HAL_I2C_RepeatStart(void)

This function will simply write a '1' to the RSTA bit of the I^2C module which triggers a repeated start condition to be put on the line.

Note that the bit does not need to be written back to '0' afterwards on this particular MCU.

void HAL_I2C_Write(byte MCU_Writebyte, Bool LastByte)

This function sends the actual data or the register address.

Prerequisites: Call HAL_I2C_AddrFT800wr to address FT800

Therefore, the I^2C module is already enabled and the FT800 has already been addressed and had the register address to be written sent to it.

The data byte to be written is passed to this function along with a Boolean value indicating whether this is the last byte of the I^2C transaction (in which case, it finished with an I^2C Stop).

This function does the following:

- Writes the data byte to the I^2C Data register and the MCU's I^2C module will send it out
- Waits for the I^2C write to complete
- Checks if the I^2C module received an ACK from the FT800
- If LastByte == TRUE, disables the I^2C module Master mode which generates the Stop and Disables the MCU's I^2C module

byte HAL_I2C_Read(Bool FirstByte, Bool LastByte)

This function reads a data byte from the FT800.

Prerequisites: Call HAL_I2C_AddrFT800wr to address FT800
Send register address with HAL_I2C_Write
Repeated Start
Call HAL_I2C_AddrFT800rd to address FT800

Therefore, the I^2C module is already enabled and the FT800 has already been addressed and had the register address to be read sent to it.

This function will then do the following:

- If this is the first byte to be read, put the I^2C module into 'receiving' mode and enable the 'send acknowledge' option
- If this is the last byte, to be read, disable the 'send acknowledge' option as the MCU should NAK the last byte
- If this is the first byte
  - Read the I^2C data register to start a dummy read which will clock in the actual first byte
  - Wait for the I^2C read to complete. The byte just clocked in will now be in the I^2C data register.
- If this is the last byte to be read, disable Master mode now so that the subsequent reading of the I^2C data register does not trigger the clocking in of another byte
- Read the I^2C data register to get the most recent byte clocked in. Note: If the master mode was not disabled in the previous step, this will begin the clocking in of the *next* byte over I^2C.
- If this is the last byte, disable the I^2C module.

Note: In the MC9S08 MCU, the I^2C module (when configured as a Master in read mode) will start an I^2C read whenever the I^2C data register is read. When carrying out a read (e.g. a 4-byte read)

For the first byte, a dummy read is carried out first to trigger the clocking in of the first byte over I^2C. When this completes, the first byte read will be available in the I^2C data register. The application then reads this byte from the I^2C data register. This in turn begins the next read cycle which will read the second byte.

For the last of the four bytes, the I^2C Master mode bit must be de-selected before reading the data register, to avoid clocking in an extra (fifth) byte at the end.

6.3 I/O Functions

void HAL_I2C_PDlow(void)

This function will simply set the port pin assigned to the Power Down pin of the FT800 to the low state. In this example, it does this by a read-modify-write operation.

void HAL_I2C_PDhigh(void)

This function will simply set the port pin assigned to the Power Down pin of the FT800 to the high state. In this example, it does this by a read-modify-write operation.

6.4 General Functions

void Delay(void)

A delay function is also provided here. A delay should be used after powering up the FT800 and after changing the oscillator frequency. This function is also used in the example applications to provide a general delay. The delay does not use any MCU-specific registers but is included in the MCU_Specific functions section because some processors/compilers may have existing routines or have hardware timers which could be used instead.

void DelayShort(void)

This is a short delay used between some I^2C module operations and may not be required on other MCUs

6.5 Data Types

The sample code uses the following data types. These are used throughout the sample code. These may need to be replaced or be defined in the source code if the chosen compiler uses different data types to represent these sizes of unsigned data.

Size	Data Type
Unsigned 8-bit value	Byte
Unsigned 16-bit value	Word (== Unsigned Int)
Unsigned 32-bit value	Dword
Boolean value	Bool (== Byte)

Table 6.2 Data Types used in the Sample Code

Note: In the source code, Word is defined as type Unsigned Int. Bool is defined as type Byte.

The code also has #include definitions at the top of main.c which would need changed if a different compiler/processor type was used.

```
#include "derivative.h" /* include peripheral declarations */
#include "FT_Gpu.h"
```

The FT_Gpu.h can be used with any processor type as it contains the FT800-specific register addresses etc.

The derivative.h contains definitions specific to Freescale MCUs and in particular the MC9S08QE8. These definitions would be updated to include the definitions etc. for the specific MCU model used. The development tools for the selected MCU will normally include an equivalent library file.

7 Final Application Architecture

The example which accompanies this application note intentionally uses individual address and data functions, to show the actual bytes which are sent and received over the I^2C interface. This allows developers to determine the byte-level transfers required, so that they can create applications for platforms where I^2C libraries are not available and where the existing sample code cannot be directly ported.

Most final applications would create an additional layer on top of the FT800_ functions shown in this application note so that the top-level application code would use more friendly function calls. This will make the main application code more readable and maintainable.

In each case below, the new function call can be seen to perform the same tasks as several lines of code in the sample project provided with this application note.

Note: The source code provided with this application note does not use this additional layer discussed below so that the low level transfers used when interfacing the FT800 over I2C can be easily demonstrated.

Creating a Write8 function which writes an 8-bit register:

```
Example function call: Write8(REG_GPIO_DIR, 0x83)


Function:              // Writes the 8-bit value to the address
                       Write8(dword address, byte value)
                       {
                              FT800_I2C_SendAddressWR(address);
                              FT800_I2C_Write8(value, TRUE);
                       }
```

Creating a Read8 function which reads the value from an 8-bit register:

```
Example function call: chipID = Read8(REG_ID)


Function:              // Returns 8-bit value read from specified address
                       byte Read8(dword address)
                       {
                              Byte data = 0x00;
                              FT800_I2C_SendAddressRD(address);
                              data = FT800_I2C_Read8();
                              Return data;
                       }
```

Creating a co-processor command list in the FIFO and executing the commands:

```
Example function call: dword CMD_Offset = 0; // keeps track of FIFO offset
                                             // global variable used here for simplicity

                       // ### New command list###

                       WaitFifoReady()               // Wait for FIFO

                       Add_CoPro__DL_START()                 // Add new commands
                       Add_CoPro__CLEAR_COLOR_RGB(0x00, 0x00, 0xFF);
                       … add other commands here …

                       Execute_Commands()            // FT800 executes them
```

Functions:

```
// Waits for the read and write pointers to be equal
// Also updates global variable CMD_Offset with current write pointer
// value to be used as starting point for writing next command list
WaitFifoReady()
{
        Do
        {
                Dword cmdBufferWr = 0x00;
                Dword cmdBufferRd = 0x00;

                FT800_I2C_SendAddressRD(REG_CMD_WRITE);
                cmdBufferWr = FT800_I2C_Read32();
                FT800_I2C_SendAddressRD(REG_CMD_READ);
                cmdBufferRd = FT800_I2C_Read32();
        } while(cmdBufferWr != cmdBufferRd);
        CMD_Offset = cmdBufferWr;
}


// Updates the write pointer to the end of the new list
// global variable CMD_Offset has tracked this
Execute_Commands()
{
        FT800_I2C_SendAddressWR(REG_CMD_WRITE);
        FT800_I2C_Write16(CMD_Offset, TRUE);
}


// Functions for each co-processor command

// Adds the DL_Start command (0xFFFFFF00) and increments
// variable CMD_Offset in the main application by 4
Add_CoPro__DL_START()
{
        FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);
        FT800_I2C_Write32(0xFFFFFF00, TRUE);
        CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4);
}


// Adds the CLEAR_COLOR_RGB command (0x02BBGGRR) and increments
// variable CMD_Offset in in the main application by 4
Add_CoPro__CLEAR_COLOR_RGB(byte RED, byte GREEN, byte BLUE)
{
        Dword COLORcommand = 0x00000000;

        COLORcommand =    (0x02000000 || (RED << 16) || (GREEN << 8)
                                || (BLUE));

        FT800_I2C_SendAddressWR(RAM_CMD + CMD_Offset);
        FT800_I2C_Write32(COLORcommand, TRUE);
        CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4);
}
```

Other functions could be created for the other co-processor commands in addition to those shown above. The FT800 example application which is designed for MPSSE and Arduino devices can be used as an example (see the 'src' and 'hdr' directories) along with the FT800 Programmers guide. Refer to Appendix A – References for links to the Programmers Guide and Sample App.

In addition, since this application note is intended to describe the basics of the low-level I^2C communication between MCU and FT800, it sends the entire co-processor list each time and also sends the FIFO address for every individual co-processor command. There are several ways to optimize the creation of co-processor lists to reduce the amount of traffic on the SPI or I^2C bus, especially where only particular parts of the screen are changing. These are out with the scope of this application note.

8 Hardware

This example uses the MC9S08QE8 MCU from Freescale. This low-cost MCU is available in a 16-pin Dual In-Line package and has an internal oscillator and I^2C module allowing the circuit to be created with minimal external components, as demonstrated in the circuit diagram below.

The sample code provided will work directly with higher pin count members of the MC9S08QE8 family (available in surface-mount packages) and on other members of the MC9S08 family with only minor changes to the initial MCU configuration routine. It can also be ported to MCUs from other manufacturers by changing the hardware-specific routines as explained in the previous sections of this application note.

An additional voltage regulator provides the 3.3V supply to the MCU. Note that in the EVE Click board, the 3v3 supply also powers the entire display and backlight and so this must be considered when specifying the regulator circuit for power dissipation etc.

An EVE Click board is used in this example, as it allows access to the MODE pin of the FT800 so that it can be put into I^2C mode. Most FT800 evaluation modules were intended for SPI operation and so have the MODE pin permanently connected to GND on the PCB.

The prototype circuit shown in the schematic below has connectors for the 5V power input, the BDM debug interface for the MCU and for connection to the EVE Click board.

The firmware for the MC9S08QE8 MCU was created using the Freescale CodeWarrior Development Studio for Microcontrollers V6.3. The application is based on a standard project created by the New Project Wizard. The MC9S08QE8 processor type was selected when creating the project. All application code was added to the main.c source file and an additional library FT_Gpu.h was also added which contains the register definitions for the FT800.

A P&E USB Multilink was used as the Programming and In Circuit Debug interface between the PC and the MCU's Background Debug port.

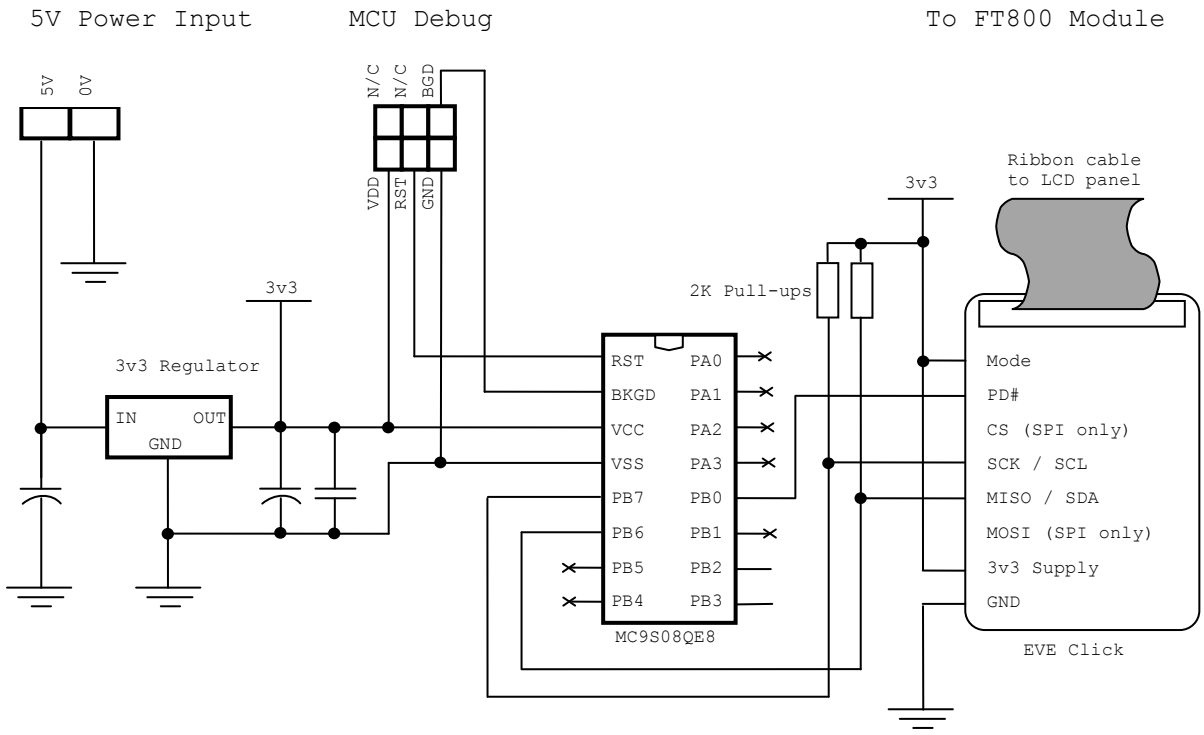


Figure 8.1 Schematic of the MCU Circuit

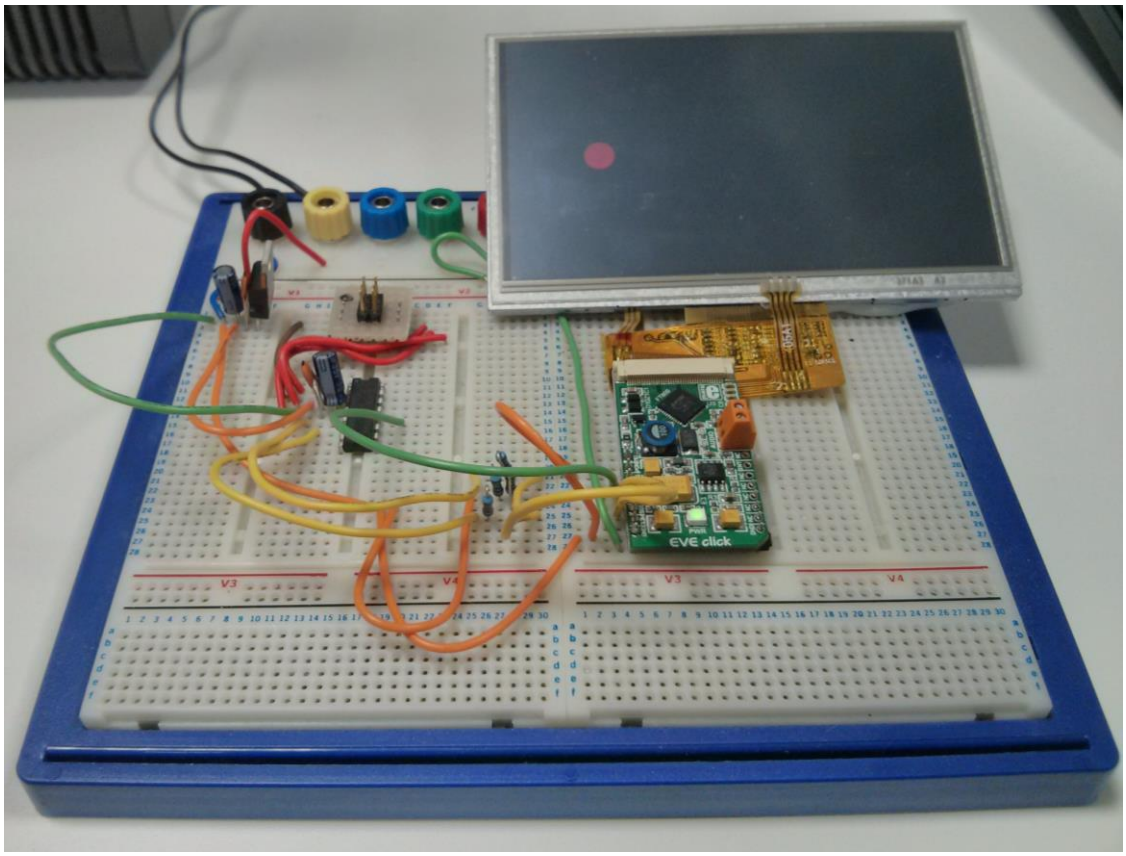


Figure 8.2 MCU board with EVE Click board and screen

9 Conclusion

This application note has presented a simple example of initializing the FT800 and then creating displays using command lists from a low-cost MCU over an I^2C interface. The sample code provided has intentionally been kept simple to demonstrate the low-level I^2C communication between the MCU and FT800 but can be expanded to produce more comprehensive displays for real applications.

The code can be extended to display screens containing many other graphics objects and widgets by changing the command list within the User Application - Main Application section of the example code.

The code can also be used on other MCU types or I^2C hosts. For most types of MCU, the functions shown in the User Application and FT800 I^2C Functions sections of the code can be used unchanged. The only changes are in the areas specific to the MCU and Compiler. These are normally the Hardware-Specific Functions, the data types and the include files for the MCU.

10 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
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) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com

Branch Office – Tigard, Oregon, USA

Future Technology Devices International Limited
(USA)
7130 SW Fir Loop
Tigard, OR 97223-8160
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

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

Branch Office – Shanghai, China

Future Technology Devices International Limited
(China)
Room 1103, No. 666 West Huaihai Road,
Shanghai, 200052
China
Tel: +86 21 62351596
Fax: +86 21 62351595

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

Web Site

www.ftdichip.com

Appendix A – References

The Freescale CodeWarrior Project for the MC9S08QE8 firmware can be found at the link below:
http://www.ftdichip.com/Support/SoftwareExamples/EVE/AN_308_Source_Code.zip

Document References

DS_FT800	FT800 Datasheet
PG_FT800	FT800 Programmer Guide
EVE Click	EVE Click Information (links to MikroElektronika site)
MC9S08QE8 Datasheet	MCU Datasheet (links to Freescale site)
MC9S08QE8 Reference Manual	MCU Reference Manual (links to Freescale site)
AN_240	FT800 From the Ground Up
AN_259	FT800 SPI Example with 8-bit Freescale MCU
AN_296	FT800 I2C Example with Arduino
FT800 Software Examples	FT800 software examples page

Acronyms and Abbreviations

Terms	Description
EVE	Embedded Video Engine
GPIO	General Purpose Input / Output
HAL	Hardware Abstraction Layer
IC	Integrated Circuit
I^2C	Inter-Integrated Circuit
MCU	Microcontroller
SPI	Serial Peripheral Interface
TFT	Thin-Film Transistor
VGA	Video Graphics Array
WQVGA	Wide Quarter VGA (480 x 272 pixel display size)

Appendix B – List of Tables & Figures

List of Figures

Appendix C – Revision History

Document Title: AN_308 FT800 I2C Example with 8-bit MCU

Document Reference No.: FT_001012

Clearance No.: FTDI#391

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

Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2014-06-09