Application Note

AN_259

FT800 Example with 8-bit MCU

Version 1.0

Issue Date: 2013-10-09

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 SPI. 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^{2}C interface allowing interfacing to nearly any microcontroller with a SPI or I^{2}C 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 SPI 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 Software Architecture

This example code has three main sections:

- User Application
    - Initialization of MCU / FT800 / Display (see section 3.1)
    - Main Application (see section 3.2)
- FT800 SPI Functions (see section 4)
- Hardware-Specific Functions (see section 5)

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. Note that 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 SPI functions are called by the User Application, and translate the commands such as FT800_Write_16(data) into the actual byte data values which will be sent over SPI to the FT800.

The Hardware-Specific 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 SPI Functions will be needed.

For example, writing a 16-bit register:

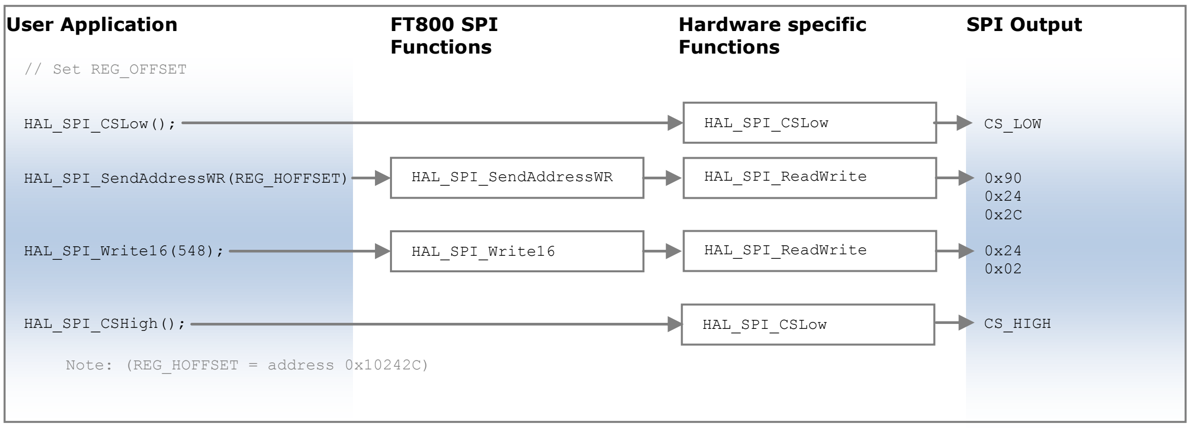


Figure 2.1 Example of using FT800 and Hardware-Specific functions

FT800 SPI Functions
void FT800_SPI_SendAddressWR(dword);
void FT800_SPI_SendAddressRD(dword);
dword FT800_SPI_Read32(void);
byte FT800_SPI_Read8(void);
void FT800_SPI_Write32(dword);
void FT800_SPI_Write16(unsigned int);
void FT800_SPI_Write8(byte);
void FT800_SPI_HostCommand(byte);
void FT800_SPI_HostCommandDummyRead(void);
unsigned int FT800_IncCMDOffset(unsigned int, byte);

Hardware-Specific Functions
void HAL_Configure_MCU(void);
byte HAL_SPI_ReadWrite(byte);
void HAL_SPI_CSLow(void);
void HAL_SPI_CSHigh(void);
void HAL_SPI_PDlow(void);
void HAL_SPI_PDhigh(void);
void Delay(void);

Table 2.1 Functions provided in the sample code

3 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 ways to create a screen to be displayed.

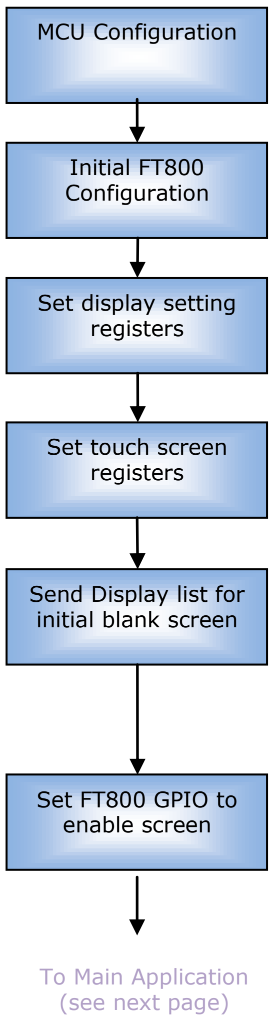
- The first way is to write display list commands directly to the RAM_DL (Display List RAM). Note: This method is illustrated when blanking the screen as part of the initialization of the FT800 as discussed in section 3.1.

- The second way 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 graphics objects (lines etc.) and Widgets (slider etc.) on the same screen.
  Note: This method is illustrated when creating the two main screens in the Main Application in section 3.2.

Although it is in theory possible to mix both methods when creating a new display list (screen) it is recommended that only one of the two methods is used in any given screen. This is because the RAM_DL would be written by both the MCU and the Co-Processor within the FT800.

3.1 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.


MCU Configuration
Initial FT800 Configuration
Set display setting registers
Set touch screen registers
Send Display list for initial blank screen
Set FT800 GPIO to enable screen
To Main Application (see next page)

Configures the MCU I/O Ports and SPI 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 SPI 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 supplied with the Credit Card module VM800C50A-D. 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 SPI. This part of the code writes to these registers to assert the display's enable pin which is connected to the FT800 GPIO.

Figure 3.1 Initialization flow chart

3.2 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 producing animation by moving or gradually changing the color of the circle.

As discussed previously, in this example, both of 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.

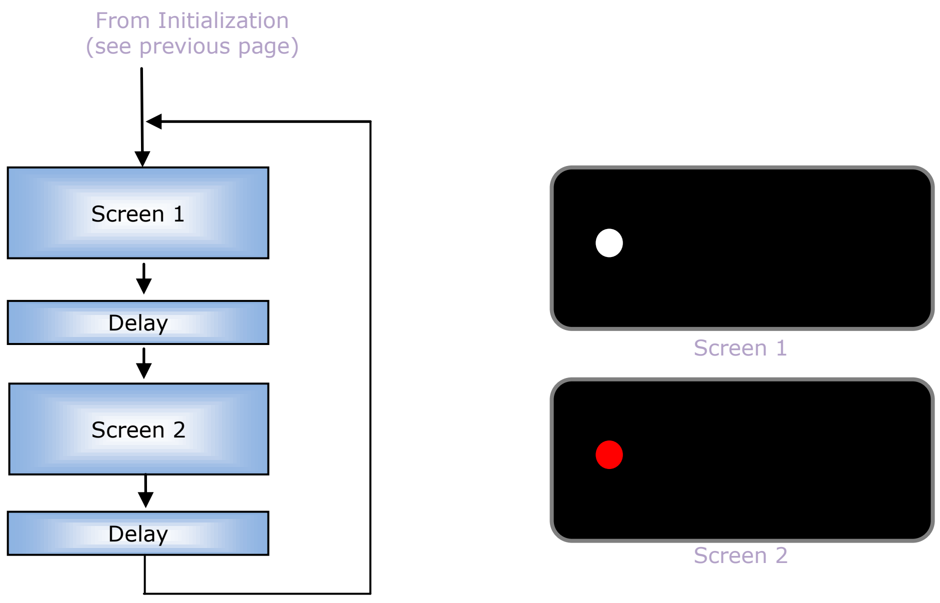


Figure 3.2 Main Application loop flow chart

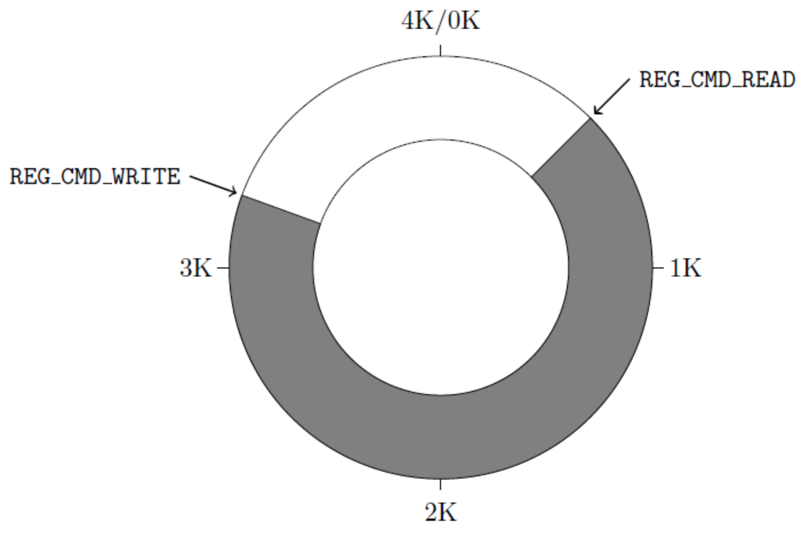
3.2.1 Creating a Coprocessor Command List

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 3.3 Co-Processor Command FIFO

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

```
do
{
        HAL_SPI_CSLow();
        HAL_SPI_SendAddressRD(REG_CMD_WRITE);    //
        cmdBufferWr = FT800_SPI_Read32();        //
        HAL_SPI_CSHigh();

        HAL_SPI_CSLow();
        HAL_SPI_SendAddressRD(REG_CMD_READ);     //
        cmdBufferRd = FT800_SPI_Read32();        //
        HAL_SPI_CSHigh();
} 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 Chip Select is asserted, and the 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). Commands are <u>always</u> multiples of 4 bytes for the FT800. The Chip_Select line is then de-asserted, marking the end of this command.

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.

```
HAL_SPI_CSLow();
HAL_SPI_SendAddressWR(RAM_CMD + CMD_Offset);// Writing to next location in FIFO
FT800_SPI_Write32(0xFFFFFF00);              // Write the DL_START command
HAL_SPI_CSHigh();

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

The next command now starts from the offset determined above

```
HAL_SPI_CSLow();
HAL_SPI_SendAddressWR(RAM_CMD + CMD_Offset;// Writing to next location in FIFO
FT800_SPI_Write32(0x02000000);             // Clear Color RGB to black
HAL_SPI_CSHigh();

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 3.2.2.

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 4096 bytes 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(dec). 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.

```
HAL_SPI_CSLow();
HAL_SPI_SendAddressWR(REG_CMD_WRITE);    //
FT800_SPI_Write16(CMD_Offset);           //
HAL_SPI_CSHigh();
```

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 SPI Traffic.

3.2.2 Drawing the first application screen

To draw the first screen, the process shown in section 3.2.1 is used to send the following ten commands.

Note: For clarity, only the actual command values are shown here. The actual code for creating the command list will require all steps shown in section 3.2.1. The full code listing can be found in the source code file provided with this application note (see Appendix A – References)

```
…
FT800_SPI_Write32(0xFFFFFF00);          // Write the DL_START command
…
FT800_SPI_Write32(0x02000000);          // Clear Color RGB
…
FT800_SPI_Write32(0x26000007);          // Clear
…
FT800_SPI_Write32(0x04FFFFFF);          // Color RGB (FFFFFF = white)
…
FT800_SPI_Write32(0x0D0000FF);          // Point Size
…
FT800_SPI_Write32(0x1F000002);          // Begin
…
FT800_SPI_Write32(0x43000880);          // Vertex 2F
…
FT800_SPI_Write32(0x21000000);          // End
…
FT800_SPI_Write32(0x00000000);          // Display
…
FT800_SPI_Write32(0xFFFFFF01);          // Swap
…
```

3.2.3 Drawing the second application screen

The example application then creates a second Command List which is almost identical to the first one, but uses a different color for the circle.

Note: As in section 3.2.2, only the actual commands are listed here. The full code listing showing all steps required to create the command list can be found in the source code file provided with this application note (see Appendix A – References)

```
…
FT800_SPI_Write32(0xFFFFFF00);          // Write the DL_START command
…
FT800_SPI_Write32(0x02000000);          // Clear Color RGB
…
FT800_SPI_Write32(0x26000007);          // Clear
…
FT800_SPI_Write32(0x040000FF);          // Color RGB (0000FF = red)
…
FT800_SPI_Write32(0x0D0000FF);          // Point Size
…
FT800_SPI_Write32(0x1F000002);          // Begin(POINTS)
…
FT800_SPI_Write32(0x43000880);          // Vertex 2F
…
FT800_SPI_Write32(0x21000000);          // End
…
FT800_SPI_Write32(0x00000000);          // Display
…
FT800_SPI_Write32(0xFFFFFF01);          // Swap
…
```

By continually drawing the first and second screens in turn (with a delay between), the display will show a circle which changes between red and white repeatedly.

4 FT800 SPI 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 SPI communication.

To form a complete transaction, the main application will normally perform the following actions:

• Assert Chip Select	(see section 5)
• Send Address	(see section 4.1)
• Write or read data	(see sections 4.2 and 4.3)
• De-Assert Chip Select	(see section 5)

Some examples include:

Reading the 8-bit REG_ID register…

```
        HAL_SPI_CSLow();                        // CS low
        HAL_SPI_SendAddressRD(REG_ID);          // Send the address with RD bits
        chipid = FT800_SPI_Read8();             // Read the actual value
        HAL_SPI_CSHigh();                       // CS high to complete transaction
```

Writing a 16-bit value to the HCYCLE register…

```
        HAL_SPI_CSLow();                        // CS low
        HAL_SPI_SendAddressWR(REG_HCYCLE);      // Send the address
        FT800_SPI_Write16(548);                 // Send the 16-bit value
        HAL_SPI_CSHigh();                       // CS high
```

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 at current position in FIFO
                                                // (read from FT800 by earlier code)
        // Add the DL_Start command to the FIFO
        HAL_SPI_CSLow();                                 //
        HAL_SPI_SendAddressWR(RAM_CMD + CMD_Offset);     // First location in CMD FIFO
        FT800_SPI_Write32(0xFFFFFF00);                   // Write the DL_START command
        HAL_SPI_CSHigh();                                //

        CMD_Offset = UpdateCmdFifo(4);                   // Add 4 to the offset

        // Add the Clear Color RGB command to the FIFO
        HAL_SPI_CSLow();                                 //
        HAL_SPI_SendAddressWR(RAM_CMD + CMD_Offset);     // Next location in CMD FIFO
        FT800_SPI_Write32(0x02000000);                   // Clear Color RGB
        HAL_SPI_CSHigh();                                //

        CMD_Offset = UpdateCmdFifo(4);                   // Add 4 to the offset

        …
```

Note: This sample code uses separate functions to send the address and the data itself. This allows the same code to be used to specify the address, regardless of the number of data bytes being written to that address. Although each command used here is 4 bytes long, the commands for widgets such as sliders need larger numbers of bytes to be sent whilst CS is asserted.

4.1 Send Address functions

These functions send the address which is to be written or read. They take a dword parameter which should have the address in the lower three bytes. They configure the upper two bits of this 24-bit address to indicate to the FT800 whether this is a write or a read operation and then use the HAL_SPI_ReadWrite function to send the resulting three bytes.

void HAL_SPI_SendAddressWR(dword Memory_Address)

This function sends the 24-bit register address. It forces the MSB of the address to '10' which tells the FT800 that this is a write operation and the byte(s) following the address will be written to that address. Since the address is 3 bytes but a dword was passed into the function, the upper byte is ignored. The function then sends the address MS byte first.

e.g. HAL_SPI_SendAddressWR(0x102428);

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

void HAL_SPI_SendAddressRD(dword Memory_Address)

This function performs the same task as HAL_SPI_SendAddressWR but it instead sets the upper two bits of the address to '00', which tells the FT800 that this is a read of the location being addressed.

e.g. HAL_SPI_SendAddressRD(0x102428);

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

4.2 Write 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 or command list.

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

void FT800_SPI_Write32(dword SPIValue32)

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_SPI_ReadWrite function one byte at a time starting with the least significant byte.

The return value from HAL_SPI_ReadWrite is ignored as it is not required in this case. Some compilers will display a warning because of this.

e.g. FT800_SPI_Write32(0x12345678);

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

void FT800_SPI_Write16(unsigned int SPIValue16)

This function is similar to the FT800_SPI_Write32 but it accepts a 16-bit value and sends two bytes out of the SPI interface.

e.g. FT800_SPI_Write16(0x1234);

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

void FT800_SPI_Write8(byte SPIValue8)

This function is similar to the FT800_SPI_Write32 and FT800_SPI_Write16 but it accepts an 8-bit value and sends one bytes out of the SPI interface.

e.g. FT800_SPI_Write8(0x12);

```
Value passed in:           00010010                                      (byte)
Value written out of MOSI: 00010010
Value returned:            N/A
```

4.3 Read Functions

These functions can be used to read a value from a 32-bit or 8-bit register. Before calling these functions, the address should be specified by using the Send Address RD functions in the previous section.

dword FT800_SPI_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.

The HAL_SPI_ReadWrite routine always writs a byte whenever it reads a byte and so dummy zero bytes are passed when calling HAL_SPI_ReadWrite.

e.g. MyReadDword = FT800_SPI_Read32(); *(assume register being read has value 0x87654321)*

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

byte FT800_SPI_Read8()

This function is similar to the FT800_SPI_Read32 function above but reads only 8 bits from the register.

e.g. MyReadByte = FT800_SPI_Read8(); *(when the register being read has value 0x21)*

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

4.4 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_SPI_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 writing three bytes over SPI to the FT800. The first 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.

Because the host commands have a fixed number of bytes, the chip select assert and de-assert are also performed within the function itself.

e.g. FT800_SPI_HostCommand (FT_GPU_PLL_48M); *(FT_GPU_PLL_48M defined as 0x62)*

```
Value passed in:           01100010                                        (byte)
Value written out of MOSI: 01100010
                           00000000
                           00000000
Value returned:            N/A
```

void FT800_SPI_HostCommandDummyRead(void)

This function performs a dummy read of memory location 0x000000 which is used to wake up the FT800. The function simply sends three bytes of 0x00. It therefore takes no parameters and does not return a value.

e.g. FT800_SPI_HostCommandDummyRead();

```
Value passed in:           N/A
Value written out of MOSI: 00000000
                           00000000
                           00000000
Value returned:            N/A
```

unsigned int FT800_IncCMDOffset(unsigned int 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

```
HAL_SPI_CSLow();
HAL_SPI_SendAddressWR(RAM_CMD + CMD_Offset);// Writing to next location in FIFO
FT800_SPI_Write32(0xFFFFFF00);              // Write DL_START (4-byte command)
HAL_SPI_CSHigh();
CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Update Offset
```

CMD_Offset now equals 4

5 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.

5.1 Function Descriptions

Each function is described below:

5.1.1 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 SPI peripheral on the MCU to be Master using SPI Mode 0 and set the desired SPI clock frequency.

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. As noted, the MCU's SPI module will take over control of bits 4, 3 and 2 once enabled. They are pre-set to their idle values initially. The SPI Slave select (port B5) is controlled by the code itself and not by the SPI module. It is initially set high (de-asserted). The Power-Down signal (port B0) is initially set high so that the FT800 is powered down.

Port	Direction	Initial Value	Notes
Port B7	IN	1	Not used
Port B6	IN	1	Not used
Port B5	OUT	1	SPI Slave Select to FT800
Port B4	IN	1	Will become SPI MISO once SPI module enabled
Port B3	OUT	1	Will become SPI MOSI once SPI module enabled
Port B2	OUT	0	Will become SPI Clock once SPI module enabled
Port B1	IN	1	Not used
Port B0	OUT	1	Power Down signal to FT800

Table 5.1 Port pin configuration

5.1.2 Data Transfer Functions

byte HAL_SPI_ReadWrite(byte MCU_Writebyte)

This function will send the specified byte over SPI and will return the byte which was simultaneously received. The SPI module on the MCU always sends and receives simultaneously.

- Reads the SPI Status register in the MCU and waits for the 'Transmit Buffer Empty' flag to be set. This indicates that the SPI module is ready to send the next byte.
- Writes the byte to be sent (which was passed in when calling this function) to the SPI Data register
- The SPI module will now clock out the byte on MISO and will clock a byte back in on MOSI
- Reads the SPI Status register in the MCU and waits for the 'SPI Transmit Buffer Full' flag to be set, indicating that the transaction is complete and the byte received on MISO is now ready to read.
- Read the SPI Data register to read the received byte.

Note that on this MCU the same SPID register address is used for both reading and writing - the MCU automatically detects whether SPID is being read or written. A byte written to SPID will go to the SPI Tx register and a byte read from SPID will return the byte in the SPI Rx register.

5.1.3 I/O Functions

void HAL_SPI_CSLow(void)

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

void HAL_SPI_CSHigh(void)

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

void HAL_SPI_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_SPI_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.

5.1.4 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.

5.2 Data Types

The sample code uses the following data types. These are used throughout the sample code. These may need to be replaced 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	unsigned int
Unsigned 32-bit value	Dword

Table 5.2 Data Types used in the Sample Code

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.

6 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 SPI 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 whilst the 5V from the external power supply is provided directly to the FT800 Credit Card module through the 10-way pin header. Note that the FT800 Credit Card module could also be powered directly from 3v3, allowing a 3v3 supply to be used and allowing the regulator to be removed.

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 10-way header on the VM800C Credit Card 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.

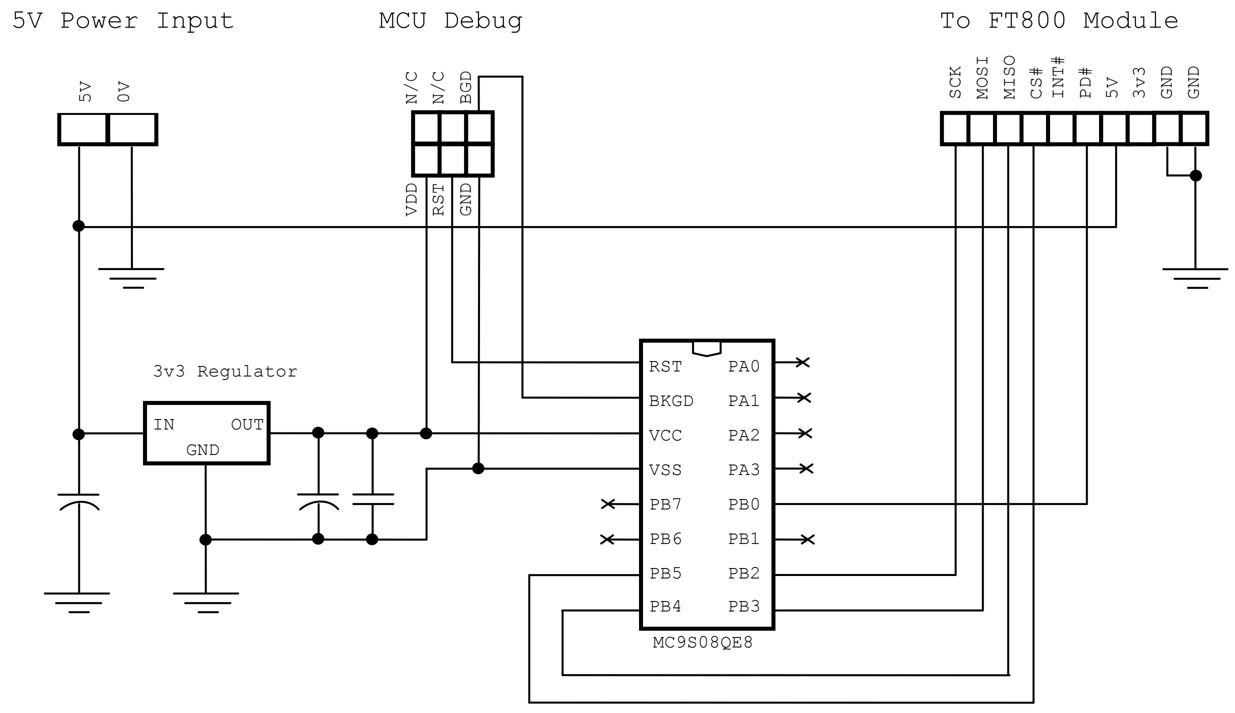


Figure 6.1 Schematic of the MCU Circuit

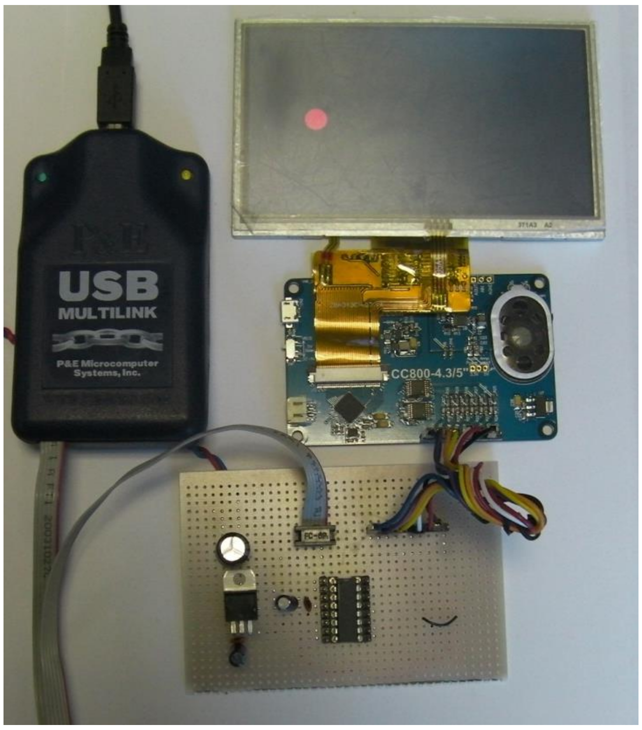


Figure 6.2 MCU board with FT800 Credit Card board and Debugger

7 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 SPI interface. The sample code provided has intentionally been kept simple to demonstrate the low-level SPI 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 SPI hosts. For most types of MCU, the functions shown in the User Application and FT800 SPI 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.

8 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_259 Source Code.zip

Document References

DS_FT800	FT800 Datasheet
PG_FT800	FT800 Programmer Guide
DS_VM800C	FT800 Credit Card Module Datasheet
MC9S08QE8 Datasheet	MCU Datasheet
MC9S08QE8 Reference Manual	MCU Reference Manual
AN_240	FT800 From the Ground Up

Acronyms and Abbreviations

Terms	Description
EVE	Embedded Video Engine
GPIO	General Purpose Input / Output
HAL	Hardware Abstraction Layer
IC	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_259 FT800 Example with 8-bit MCU
Document Reference No.:	FT_000897
Clearance No.:	FTDI# 353
Product Page:	http://www.ftdichip.com/EVE.htm
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	Initial Release	2013-10-09