



Application Note

AN_255

USB to I2C Example using the FT232H and FT201X devices

Version 1.0

Issue Date: 2013-09-30

FTDI have a range of devices which can be used to interface I²C devices to a host computer over USB. These include both I²C Master and I²C Slave solutions.

This application note provides information to assist engineers in designing USB to I²C interfaces with these devices, including sample code. It then illustrates their use by creating a simple I²C application with the FT232H connected to the FT201X over I²C.

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

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>

Copyright © 2013 Future Technology Devices International Limited

Table of Contents

1	Introduction	3
1.1	Overview	3
1.2	Selecting the correct FTDI chip for the I ² C Interface.....	4
2	FT232H USB-I ² C Master.....	5
2.1	Device Hardware settings	5
2.2	Device Initialization.....	6
2.2.1	Opening the Device.....	6
2.2.2	Configuring the Device.....	6
2.2.3	Verify MPSSE Mode.....	7
2.2.4	MPSSE Settings.....	7
2.3	I ² C Routines.....	10
2.3.1	SetI2CLinesIdle().....	10
2.3.2	SetI2CStart().....	11
2.3.3	SetI2CStop()	12
2.3.4	ReadByteAndSendNAK()	13
2.3.5	Read3BytesAndSendNAK().....	14
2.3.6	SendByteAndCheckACK()	15
2.3.7	SendAddrAndCheckACK().....	16
2.4	Other Considerations.....	17
3	Main Program - Routines for Interfacing to FT201X.....	20
3.1.1	Checking the USB State	20
3.1.2	Reading the I ² C IDs.....	22
3.1.3	MTP Operations	22
3.1.4	Buffer Flush	24
3.1.5	Reading and Writing Data	25
4	FT201X USB to I2C Slave Interface	27
4.1	Device Hardware settings	27
4.2	I ² C Interface.....	27
4.3	USB Interface	27



5	Hardware configurations	30
6	Running the Example Code	32
7	Conclusion.....	35
8	Contact Information.....	36
Appendix A – References		37
Document References.....		37
Acronyms and Abbreviations		37
Appendix B – List of Tables & Figures		39
List of Tables		39
List of Figures		39
Appendix C – Revision History		40

1 Introduction

This application demonstrates two different I²C solutions from FTDI, and gives an example of using their USB and I²C interfaces.

- It uses the FT232H as a USB to I²C Master interface
- It uses the FT201X as a USB to I²C Slave interface

This application note has three main aims:

- Show how to initialize the FT232H as an I²C Master and provide example I²C functions for using the FT232H as a generic USB to I²C Master interface. [The information in this chapter could be used as a starting point to make an I²C Master suitable for communicating with many types of I²C Slave \(e.g. ADCs, I/O Expander\)](#)
- Provide a main program which uses the above functions to implement the specific I²C transfers used to communicate with the FT201X, to demonstrate the FT201X's communication over I²C. [This could be modified to allow the FT232H I²C Master interface to be used with other types of I²C slaves.](#)
- Provide a description of the FT201X I²C Slave device from the USB side. [Provide background information on the communication between the PC and FT201X in D2XX and VCP modes.](#)

The document then details the hardware set-up used to connect the FT232H Master and FT201X Slave together and provides screen-shots showing how to run the demo code provided.

The full source code and supporting files are available from the link below:
https://ftdichip.com/an_255-files/

Note: This Application Note includes example code. FTDI accept no responsibility for any issues caused by the use of this example code. The customer is responsible for ensuring the correct, reliable and safe operation of any overall system created based on this code.

1.1 Overview

The two devices are connected as shown below, so that they can communicate with each other over I²C.

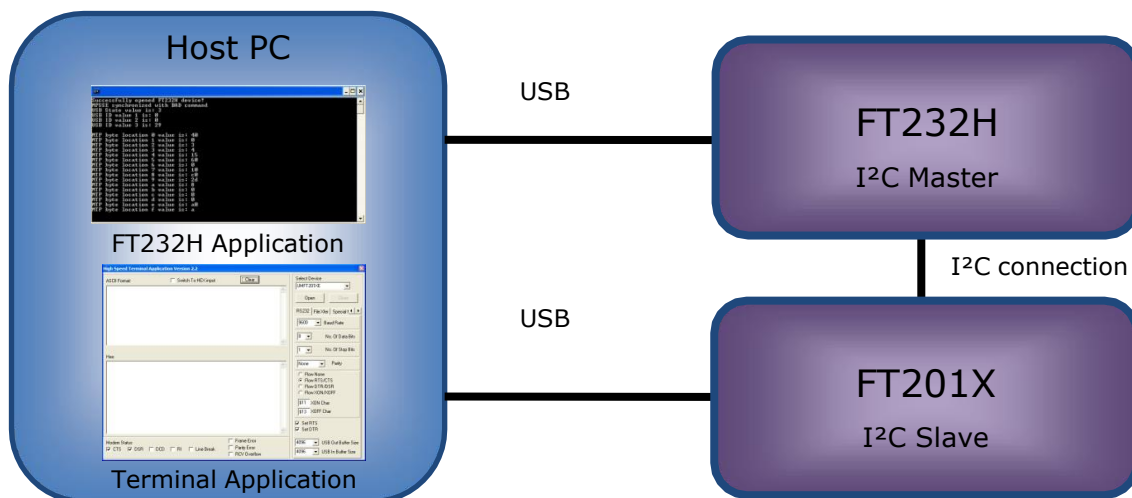


Figure 1.1 Overview of the I²C Demonstration

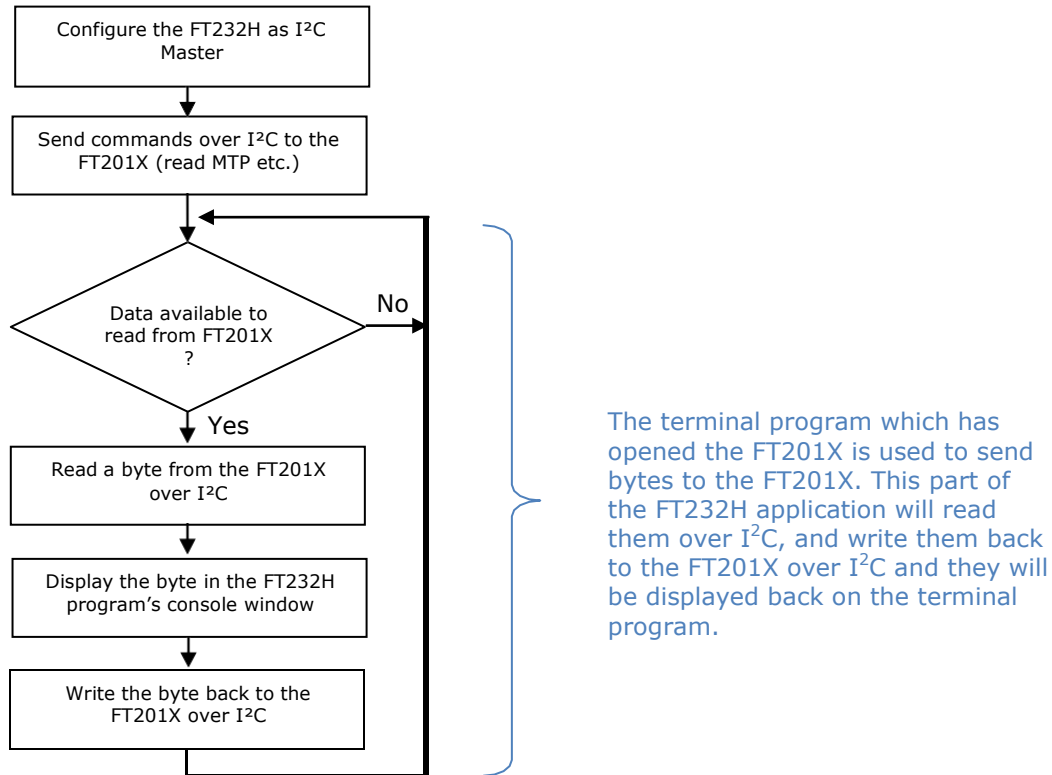


Figure 1.2 Overview of the FT232H I²C Master application

1.2 Selecting the correct FTDI chip for the I²C Interface

It is important to choose the correct FTDI device depending on which devices will be interfacing over I²C. The following devices are used in this application note:

- FT201X USB-to-I²C bridge as an I²C Slave.
 The FT201X would be used where an external microcontroller or FPGA needs to transfer data to and from a PC. In this case, the MCU/FPGA would be the I²C Master and the FT201X would be the I²C Slave.

Note: This example uses the FT201X device but the FT200XD device can be used with no software changes. The FT200XD is identical in operation from an I²C point of view, but with a reduced number of CBUS pins compared to the FT201X. The FT200XD is in a smaller package and is ideal for PCBs where space is limited.

- FT232H as a USB-to-I²C Master interface.
 The FT232H would be used where the PC application needs to communicate with one or more I²C slaves. For example, the application may be taking readings from Analog to Digital converters which have an I²C interface, or may be reading and writing to an EEPROM chip or I/O expander.

The FT2232H and FT4232H could also be used in place of the FT232H. Both the FT2232H and FT4232H have two MPSSE channels. However, please note that these devices do not feature the open-drain option used here in the FT232H and so the pins need to be manually tri-stated when not writing on the I²C bus. This requires a small change to the routines used to generate the I²C protocol.

Note: The MPSSE is not restricted to I²C and can be used for SPI and JTAG. However, this is beyond the scope of this document and is covered in other FTDI application notes.

2 FT232H USB-I²C Master

This section shows how the FT232H can be programmed to act as a USB to I²C master interface. These routines can be used as the basis for communicating with other I²C slave devices. The routines will be used in the following chapter to communicate with the FT201X I²C slave as an example.

The FT232H uses a combination of the on-chip hardware (Multi-Protocol Synchronous Serial Engine) and the software on the host computer to implement the I²C protocol. The MPSSE hardware assists in the clocking in and out of data with required bit timings without needing to clock the data bit-by-bit from the host computer application. Control of the MPSSE mode is achieved through sending buffers of commands including manually setting/reading the device pins and clocking of data in/out. The FT232H has several features geared towards the I²C protocol including three-phase clocking and open-drain modes for the I/O pins (selectable per pin).

As shown in Section 5, for I²C the Data_In and Data_Out lines are connected together to form the single SDA data line. This allows the MPSSE to both drive data onto the I²C SDA line and also read in data which is put onto the I²C SDA line by an I²C Slave device. The MPSSE's Clock_Out line is the I²C SCL.

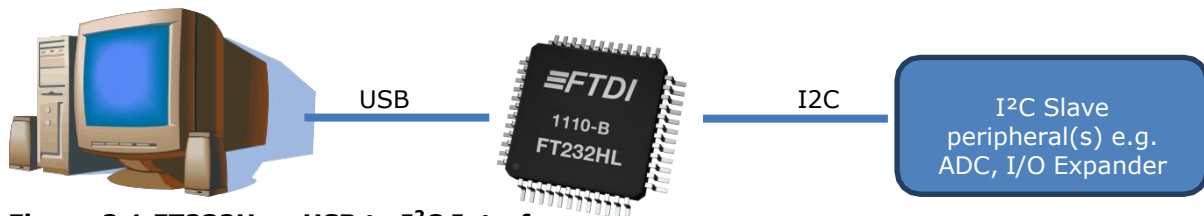


Figure 2.1 FT232H as USB to I²C Interface

2.1 Device Hardware settings

This example assumes that the following configuration settings have been applied to the FT232H EEPROM. The settings can be programmed using the FT_Prog utility available for free from the FTDI website (see Appendix A – References). The demonstration code provided with this application note may need modification if other settings have been used.

- The source code assumes that the UM232H module is used and has the default description "UM232H" in its EEPROM. If another module/cable such as the C232HM-EDHSL-0 (see Appendix A – References) is used, the string used to open the device (see section 2.2.1) will need modification.
- It is recommended to set the mode to '245 FIFO' in the EEPROM of the FT232H. The FIFO mode is recommended because it allows the ADbus lines to start as tristate. If the chip starts in the default UART mode, then the ADbus lines will be in the default UART idle states until the application opens the port and configures it as MPSSE. Care should also be taken that the RD# input on ACBUS is not asserted in this initial state as this can cause the FIFO lines to drive out. Consult the MPSSE and FIFO sections of the FT232H datasheet (see Appendix A – References) for full pin descriptions.
- The latest FTDI driver (2.08.30 at the time of writing) has been installed and the FT232H and FT201X devices are already connected to the PC's USB ports. The driver can be obtained from the FTDI driver page (see Appendix A – References). Note that even when the executable installer has been run, the devices must be connected to the PC in order to complete the installation.
- The FT201X has the default I²C address of 0x22. This can be changed in the MTP ROM inside the FT201X using FT_Prog if desired. The main code section of this application can easily be changed to communicate with other slave addresses if desired.
- The hardware is configured as shown in section 5.

2.2 Device Initialization

2.2.1 Opening the Device

The example code opens the device by description, using the FT_OpenEX function. Therefore, the entry "UM232H" will need to be changed if the FT232H used has a different description programmed (or the description in the chip could be changed to UM232H with FT_Prog. The D2xx Programmers Guide (see Appendix A – References) has further details and also shows other options for the FT_OpenEX function, allowing an application to easily find FTDI devices connected and open the desired one.

```
// Open the UM232H module by it's description in the EEPROM
ftStatus = FT_OpenEx("UM232H", FT_OPEN_BY_DESCRIPTION, &ftHandle);

if (ftStatus != FT_OK)           // Check if Open was successful
{
    printf("Can't open FT232H device! \n");
    getchar();
    return 1;
}
```

2.2.2 Configuring the Device

The next step is to configure the general device parameters and put it into MPSSE mode.

The timeouts are set to 5 seconds to ensure that the reads and writes will return if the device does not respond with the expected data within this time period.

This application uses a 'Send Immediate' command at the end of each set of commands sent to the MPSSE which involve reading data. This ensures that any data read by the MPSSE will be sent back to the PC almost immediately. It is not therefore required to set the latency timer to a low value and the latency timer is set to the default of 16ms.

Note: The latency timer sends any data which the device has received back to the PC after the time interval specified, so that a small amount of data will not sit indefinitely in the chip's on-board buffer. However, setting it to low values such as 1ms causes a lot of USB traffic because it sends an empty buffer even when the chip has no useful data to send back. When using the MPSSE, the Send Immediate allows the data which was read to be sent back to the PC quickly but without this unnecessary traffic.

The code calls the FT_SetBitMode to select the interface mode. The first call sets bit mode to 0x00 which resets the mode to that which is stored in the EEPROM (in this application, 245 FIFO mode was selected in EEPROM). This ensures that it is then ready to go into MPSSE mode. The second call sets the mode to 0x40 which is MPSSE mode. The first parameter in both of the SetBitMode calls '0x0' is setting the initial directions of the port AD to inputs.

```
// Reset the FT232H
ftStatus |= FT_ResetDevice(ftHandle);

// Purge USB receive buffer ... Get the number of bytes in the FT232H receive buffer and then read them
ftStatus |= FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
if ((ftStatus == FT_OK) && (dwNumInputBuffer > 0))
{
    FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);
}

ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535); // Set USB request transfer sizes
ftStatus |= FT_SetChars(ftHandle, false, 0, false, 0); // Disable event/error characters
ftStatus |= FT_SetTimeouts(ftHandle, 5000, 5000); // Set rd/wr timeouts to 5 sec
ftStatus |= FT_SetLatencyTimer(ftHandle, 16); // Latency timer at default 16ms
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00); // Reset mode to setting in EEPROM
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x02); // Enable MPSSE mode

// Inform the user if any errors were encountered
if (ftStatus != FT_OK)
{
    printf("failure to initialize FT232H device! \n");
    getchar();
    return 1;
}
```

2.2.3 Verify MPSSE Mode

To make sure that the MPSSE mode has successfully been selected, the application then sends an invalid command (0xAA) to the MPSSE. The MPSSE should reply with two bytes; the error code 0xFA followed by a copy of the invalid command which had been received (0xAA)

```
#####  
// Synchronise the MPSSE by sending bad command AA to it  
#####  
  
dwNumBytesToSend = 0; // Used as an index to the buffer  
OutputBuffer[dwNumBytesToSend++] = 0xAA; // Add an invalid command 0xAA  
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent); // Send to FT232H  
  
// Check if the bytes were sent off OK  
if(dwNumBytesToSend != dwNumBytesSent)  
{  
    printf("Write timed out! \n");  
    getchar();  
    return 1;  
}  
  
// Now read the response from the FT232H. It should return error code 0xFA  
// followed by the actual bad command 0xAA  
  
dwNumInputBuffer = 0;  
ReadTimeoutCounter = 0;  
  
ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); // Get # bytes in input buffer  
  
while ((dwNumInputBuffer < 2) && (ftStatus == FT_OK) && (ReadTimeoutCounter < 500))  
{  
    // Sit in this loop until  
    // (1) we receive the two bytes expected  
    // or (2) a hardware error occurs causing the GetQueueStatus to return an error code  
    // or (3) we have checked 500 times and the expected byte is not coming  
    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); // Check queue status  
    ReadTimeoutCounter ++;  
    Sleep(1); // short delay  
}  
  
// If the loop above exited due to the byte coming back (not an error code and not a timeout)  
// then read the bytes available and check for the error code followed by the invalid character  
if ((ftStatus == FT_OK) && (ReadTimeoutCounter < 500))  
{  
    ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead); // Now read data  
  
    // Check if we have two consecutive bytes in the buffer with value 0xFA and 0xAA  
    bCommandEchod = false;  
    for (dwCount = 0; dwCount < dwNumBytesRead - 1; dwCount++)  
    {  
        if ((InputBuffer[dwCount] == BYTE(0xFA)) && (InputBuffer[dwCount+1] == BYTE(0xAA)))  
        {  
            bCommandEchod = true;  
            break;  
        }  
    }  
}  
  
// If the device did not respond correctly, display error message and exit.  
if (bCommandEchod == false)  
{  
    printf("fail to synchronize MPSSE with command 0xAA \n");  
    getchar();  
    return 1;  
}
```

This is then repeated a second time with invalid instruction 0xAB. The code is not shown in this application note but can be found in the full source code file.

2.2.4 MPSSE Settings

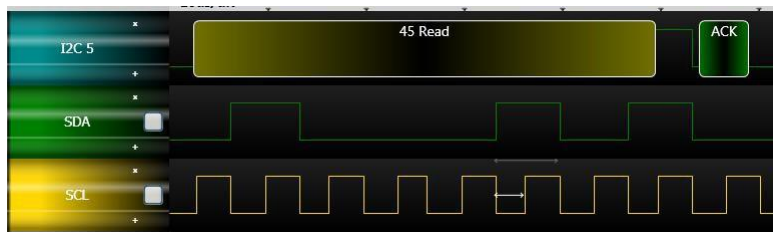
After enabling MPSSE mode, the next step is to initialize the MPSSE settings. The software builds up a buffer containing several MPSSE commands, and then sends them to the chip using the FT_Write D2xx function.

The first setting sent to the device is to disable the internal divide-by-5 clock divider so that the MPSSE receives a 60MHz clock from the FT232H's internal clock circuit.

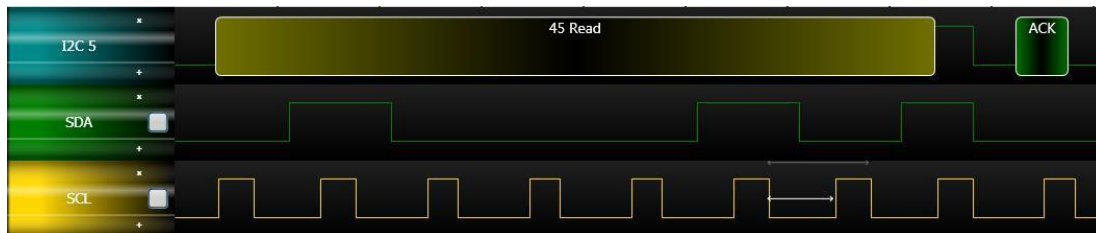
The MPSSE also includes a mode called three-phase-clocking, which is ideal for I²C implementation. This means that each bit time actually uses three clock half-cycles. With the usual two-phase clock mode, data is put onto the line for ½ clock period, and the clock then changes state in the middle of the cycle. In three-phase mode, the data stays on the line for the equivalent of three half-clock periods instead, and the clock toggles twice. The result is that the data is stable on both clock edges, which is a requirement for I²C. The figure below illustrates this.

The only configuration difference between the two figures below is that three-phase-clocking is enabled in the lower waveform. Two main implications of this are:

- There is now both a rising and a falling edge whilst the data is stable, as required by the I²C protocol.
- The three-phase clocking extends each clock cycle by 50%, which results in the period increasing by 1/3rd and the frequency (for a given clock divider value) reducing by 1/3rd.



Two-phase clocking: Clock Period: 6.5µs, Clock Frequency: 153.846KHz, Duty Cycle 50%



Three-phase clocking: Clock Period: 10µs, Clock Frequency: 100KHz, Duty Cycle 66.6%

Figure 2.2 Two- and Three-Phase Clocking Comparison

The FT232H has a drive-only-zero feature which can be enabled individually on any of the 16 ACBUS and ADBUS pins. This is effectively an open-drain mode for the selected pins and is again ideal for I²C where the lines are pulled down for logic 0 but released (pulled up by external resistors rather than driven high) for logic 1, thereby allowing many devices to share the same clock and data lines.

Finally, the clock divider is set to give the required I²C clock rate. The I²C clock frequency has been set to 100KHz in this application. This is created by dividing down the 60MHz clock which is supplied internally to the MPSSE.

MPSSE Clock Source = 60MHz

$$\text{Clock} = \frac{60\text{MHz}}{(1 + \text{CLOCKDIVIDER}) + 2}$$

$$100,000 = \frac{60,000,000}{(1 + \text{CLOCKDIVIDER}) + 2}$$

$$100,000 = (1 + \text{CLOCKDIVIDER}) + 2 = 60,000,00$$

$$1 + \text{CLOCKDIVIDER} = 300$$

CLOCKDIVIDER = 299

Before calculating the final value for the divider however, the effect of the 3-phase-clocking mode must also be considered. The standard calculation above assumes the normal two-phase clock mode. As shown above a bit cycle in three-phase mode is actually 1/3rd longer than the two-phase cycle. Therefore, the frequency should be increased by 1/3 so that the actual I²C frequency is as required. The CLOCKDIVIDER value should be reduced by 1/3 to provide a resulting 1/3 increase in the clock frequency. The Hex value for the divider is therefore 0xC8.

CLOCKDIVIDER = 299 + 2/3 ≈ 200 ≈ 0xC8

```
#####  
// Configure the MPSSE settings  
#####  
  
dwNumBytesToSend = 0; // Clear index to zero  
OutputBuffer[dwNumBytesToSend++] = 0x8A; // Disable clock divide-by-5 for 60Mhz master clock  
OutputBuffer[dwNumBytesToSend++] = 0x97; // Ensure adaptive clocking is off  
OutputBuffer[dwNumBytesToSend++] = 0x8C; // Enable 3 phase data clocking, data valid on both clock edges  
for I2C  
  
OutputBuffer[dwNumBytesToSend++] = 0x9E; // Enable drive-zero mode on the lines used for I2C ...  
OutputBuffer[dwNumBytesToSend++] = 0x07; // ... on the bits AD0, 1 and 2 of the lower port...  
OutputBuffer[dwNumBytesToSend++] = 0x00; // ...not required on the upper port AC 0-7  
  
OutputBuffer[dwNumBytesToSend++] = 0x85; // Ensure internal loopback is off  
  
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent); // Send off the  
commands  
  
// Now configure the dividers to set the SCLK frequency which we will use  
// The SCLK clock frequency can be worked out by the algorithm (when divide-by-5 is off)  
// ClockDivisor has been defined as 0xC8 at the top of the source code.  
  
dwNumBytesToSend = 0;  
  
// Clear index to zero  
OutputBuffer[dwNumBytesToSend++] = 0x86;  
// Command to set clock divisor  
OutputBuffer[dwNumBytesToSend++] = dwClockDivisor & 0xFF;  
// Set 0xValueL of clock divisor  
OutputBuffer[dwNumBytesToSend++] = (dwClockDivisor >> 8) & 0xFF;  
// Set 0xValueH of clock divisor  
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);  
// Send off the commands
```

Finally, the lines are set to their initial states for I²C and for any of the other pins used as GPIO. This involves calling the `SetI2CLinesIdle()` function to set the I²C lines to their idle state. This function is detailed in section 2.3 I2C Routines.

The function configures the I²C lines on port AD to their default states and also configures bit 6 of ACBUS to be an output to drive the status LED. The function could be customized to configure the pins not used by the I²C communication (and which can therefore be used as GPIO in the same way as AC6 here) to any states required by the application.

```
#####  
// Configure the I/O pins of the MPSSE  
#####  
  
// Call the I2C function to set the lines of port AD to their required states  
SetI2CLinesIdle();
```

2.3 I²C Routines

This section covers the general I²C functions which can be used by the main application code to send and receive I²C data. It aims to explain the way in which these work so that the reader can create their own routines which are optimized for their application. The functions provided are:

Function	Description
SetI2CLinesIdle()	Sets I ² C related pins (AD0/AD1/AD2) to their idle state. The other pins on AD3 – AD7 and AC0 – AC7 can also be configured here as per application requirements.
SetI2CStart()	Sets the Start condition on the I ² C Lines. Also puts AC6 low in this application to turn on an LED.
SetI2CStop()	Sets the Stop condition on the I ² C Lines. Also puts AC6 high in this application to turn off the LED.
ReadByteAndSendNAK()	Clocks one byte in from the I ² C Slave and replies with a NAK.
Read3BytesAndSendNAK()	Clocks three bytes in from the I ² C Slave, replies with ACK for the first two and with a NAK for the last one.
SendByteAndCheckACK()	Writes a byte onto the I ² C bus. The byte is passed in when calling the function.
SendAddrAndCheckACK()	Writes a byte onto the I ² C bus, consisting of 7 bits combined with the R/W bit. Typically used for sending an address of a slave before reading/writing data to the slave. The byte (Address) is passed in along with a flag showing whether it is a read or write.

Table 2.1 Example I²C Master functions presented in this application note

These functions are intended to be a starting point for the development of functions and applications to communicate with many types of I²C slave. They can be customized to suit the needs of the intended I²C Slaves and additional functions can be added if required.

Note that these routines are designed for the FT232H with its open drain (drive only zero) mode. The routines require a change to tri-state the Data_Out line of the FT2232H/FT4232H if used with these chips as mentioned in section 2.4.

2.3.1 SetI2CLinesIdle()

This function can be used whenever the lines need to be assured to be put back to their idle states.

It sets the I²C clock and data lines (AD0/AD1/AD2) to their idle states for I²C communication (both SCL and SDA released and pulled up by the external resistor). It also sets the states of the ACBUS port (referred to as the High Byte of the port in the AN_108 MPSSE command set document) so that an LED can be driven by AC6. This line is set as output driving high so that the LED is off initially. The other lines AD3/4/5/6/7 and AC0/1/2/3/4/5/7 are unused and configured as inputs.

The function builds up a buffer of six bytes in total, consisting of a 'Set Data Bits LowByte' command followed by the desired data and directions of the ADBUS lines, and a 'Set Data Bits HighByte' command followed by the desired data and direction of the ACBUS lines. The six bytes are then sent to the MPSSE.

Note that the GPIO commands for ADBUS and ACBUS specify the directions each time. These have been set to correspond to the hardware directions shown in Section 5 and in the comments below.

```
// #####
// Function to set all lines to idle states
// For I2C lines, it releases the I2C clock and data lines to be pulled high externally
// For remainder of port AD, it sets AD3/4/5/6/7 as inputs as they're unused in this application
// For the LED control, it sets AC6 as an output with initial state high (LED off)
// For remainder of port AC, it sets AC0/1/2/3/4/5/7 as inputs as they're unused in this application
// #####
```

```
void SetI2CLinesIdle(void)
```

```

{
    dwNumBytesToSend = 0; //Clear output buffer

    // Set the idle states for the AD lines
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction and data
    OutputBuffer[dwNumBytesToSend++] = 0xFF; // Set all 8 lines to high level
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set all pins as o/p except bit 2 (data_in)

    // IDLE line states are ...
    // AD0 (SCL) is output high (open drain, pulled up externally)
    // AD1 (DATA OUT) is output high (open drain, pulled up externally)
    // AD2 (DATA IN) is input (therefore the output value specified is ignored)
    // AD3 to AD7 are inputs (not used in this application)

    // Set the idle states for the AC lines
    OutputBuffer[dwNumBytesToSend++] = 0x82; // Command to set ACbus direction and data
    OutputBuffer[dwNumBytesToSend++] = 0xFF; // Set all 8 lines to high level
    OutputBuffer[dwNumBytesToSend++] = 0x40; // Only bit 6 is output
    // IDLE line states are ...
    // AC6 (LED) is output driving high
    // AC0/1/2/3/4/5/7 are inputs (not used in this application)

    //Send off the commands
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
}
  
```

2.3.2 SetI2CStart()

The SetI2CStart function generates the I²C Start condition on the lines. It uses the 'Set Data Bits LowByte' command to put the data_out (port AD bit 1) low so that the SDA line goes low whilst leaving the SCL line (port AD bit 0) high. As mentioned previously, this requires three bytes; the command, the data values and the direction for each bit.

The 'for' loop actually adds four sets of this sequence to the buffer (12 bytes in total) so that the MPSSE will write the same values and direction to the pins four times. This holds the value on the pins for a longer time to ensure the slave has time to recognize it.

The second 'for' loop then writes the GPIO command another four times, this time pulling the SCK line (AD bit 0) low. Both SDA and SCK are now low.

The code then sends the 'Set Data Bits HighByte' to change bit 6 of the ACbus port to the low state, turning on the LED.

Finally, it uses FT_Write to send off the commands (27 bytes in total) to the FT232H



Figure 2.3 I²C Start Condition

```

// #####
// Function to set the I2C Start state on the I2C clock and data lines
// It pulls the data line low and then pulls the clock line low to produce the start condition
// It also sends a GPIO command to set bit 6 of ACbus low to turn on the LED.
// This acts as an activity indicator. It turns on (low) during the I2C Start and off (high) during // the
// I2C stop condition, giving a short blink.
// #####

void SetI2CStart(void)
{
    dwNumBytesToSend = 0; //Clear output buffer

    DWORD dwCount;
    for(dwCount=0; dwCount < 4; dwCount++) // Repeat commands to ensure the minimum period
        // of the start hold time is achieved
    {
  
```

```

    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction/ data
    OutputBuffer[dwNumBytesToSend++] = 0xFD; // Bring data out low (bit 1)
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)
}

for(dwCount=0; dwCount < 4; dwCount++) // Repeat commands to ensure the minimum period
                                        // of the start setup time is achieved
{
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction/ data
    OutputBuffer[dwNumBytesToSend++] = 0xFC; // Bring clock line low too
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)
}

// Turn the LED on by setting port AC6 low
OutputBuffer[dwNumBytesToSend++] = 0x82; // Command to set ACbus direction and data
OutputBuffer[dwNumBytesToSend++] = 0xBF; // Bit 6 is going low
OutputBuffer[dwNumBytesToSend++] = 0x40; // Only bit 6 is output

//Send off the commands
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
}

```

2.3.3 SetI2CStop()

The SetI2CStop function works in a very similar way to the SetI2CStart function. The first set of writes to port AD will ensure that Data is low in addition to the Clock which should already be low. The second set will release the clock line (open drain, pulled up externally), and the third set of GPIO writes then release the data line so that both lines are now high (open drain, pulled up externally). The write to the High Byte (ACbus) sets the bit AC6 high so that the LED turns off.

Again, each port AD write is repeated four times which, along with the single write to the port AC, gives a total of 39 bytes to be written to the FT232H.

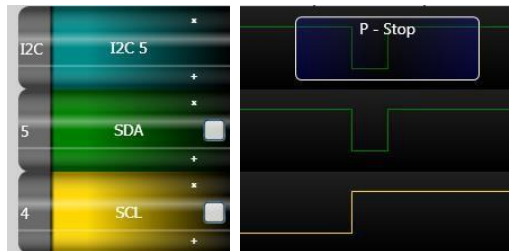


Figure 2.4 I2C Stop Condition

```

// #####
// Function to set the I2C Stop state on the I2C clock and data lines
// It takes the clock line high whilst keeping data low, and then takes both lines high
// It also sends a GPIO command to set bit 6 of ACbus high to turn off the LED, as an activity indicator
// Turns on (low) during the I2C Start and off (high) during the I2C stop condition, giving a short blink.
// #####

void SetI2CStop(void)
{
    dwNumBytesToSend = 0; //Clear output buffer
    DWORD dwCount;

    // Initial condition for the I2C Stop - Pull data low (Clock will already be low and is kept low)
    for(dwCount=0; dwCount<4; dwCount++) // Repeat commands to ensure the minimum period
                                        // of the stop setup time is achieved
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction/data
        OutputBuffer[dwNumBytesToSend++] = 0xFC; // put data and clock low
        OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)
    }

    // Clock now goes high (open drain)
    for(dwCount=0; dwCount<4; dwCount++) // Repeat commands to ensure the minimum period
                                        // of the stop setup time is achieved
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction/data
        OutputBuffer[dwNumBytesToSend++] = 0xFD; // put data low, clock remains high
        OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)
    }
}

```

```

// Data now goes high too (both clock and data now high / open drain)
for(dwCount=0; dwCount<4; dwCount++) // Repeat commands to ensure the minimum period
// of the stop hold time is achieved
{
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set Adbus direction/data
    OutputBuffer[dwNumBytesToSend++] = 0xFF; // both clock and data now high
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)
}

// Turn the LED off by setting port AC6 high.
OutputBuffer[dwNumBytesToSend++] = 0x82; // Command to set ACbus direction/data
OutputBuffer[dwNumBytesToSend++] = 0xFF; // All lines high (including bit 6 LED)
OutputBuffer[dwNumBytesToSend++] = 0x40; // Only bit 6 is output

ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
//Send off the commands
}

```

2.3.4 ReadByteAndSendNAK()

This function is used to read a byte from the I²C Slave and reply to the I²C Slave with a NAK. This function would often be preceded by a write of the I²C Slave devices address or the General Call address (refer to section 3.1.1 for a full example).

As before, the function builds up a buffer with the required MPSSE commands and then sends them to the FT232H.

The first command is 0x20 which causes the MPSSE to clock bytes in, sampling on the rising edge of the clock. This command is always followed by two bytes which define the number of bytes which the MPSSE should clock in. In this case, the function sets the length to 0x0000 because this corresponds to clocking in one byte. This byte will be placed in the FT232H USB buffer ready to be sent back to the host PC over USB.

The next command is 0x13, which clocks out bits on the falling edge of the clock. This command is followed by a byte defining the number of bits to be clocked out (0x00 means clock one bit out) and then a byte containing the value to be clocked out. Since only one bit is being clocked out in this case, the actual bit to be clocked out is the MSB (bit 7) of the value. The value is set to 0xFF so that a '1' is clocked out, which is interpreted as a NAK by the I²C Slave.

The next command 0x80 is a GPIO write to the pins, which ensures that the I²C lines are left in a consistent state regardless of what the last data value was. The SDA line goes back to an idle high state with the clock line remaining low, ready for the next transfer.

The final command is 0x87, which tells the FT232H to send the byte of data which was clocked in by the MPSSE and currently in the buffer back to the PC as soon as possible. Otherwise, the byte would sit in the buffer until the latency timer reaches its pre-configured value (default 16ms), and this would result in delays in getting the data back. The byte will go back to the PC over USB and the FTDI driver will make this byte available to the application when it calls the FT_Read command.

The buffer of commands is then written to the FT232H and it will run the sequence of commands.

The FT_GetQueueStatus function is used to check how many bytes the driver has received back from the FT232H. If no bytes have been received, a while loop waits until the byte arrives. As an additional error checking step, to avoid sitting in an infinite loop and the application hanging, the loop also checks the return value of GetQueue status for non-zero values (indicating an error) and also uses a software counter to provide a timeout if the byte never arrives back.

If the byte did come back as expected, then the program reads it using FT_Read, stores it in a global array and returns a True value to indicate success. Otherwise, it returns False indicating to the calling function that no data was read back from the Slave.

```

// #####
// Function to read 1 byte from the I2C slave (e.g. FT201X chip)
//   Clock in one byte from the I2C Slave which is the actual data to be read
//   Clock out one bit to the I2C Slave which is the ACK/NAK bit
//   Put lines back to the idle state (idle between start and stop is clock low, data high)
// This function reads only one byte from the I2C Slave. It therefore sends a '1' as the ACK/NAK
// bit. This is NAKing the first byte of data, to tell the slave we dont want to read more bytes.
// The one byte of data read from the I2C Slave is put into ByteDataRead[0]

```



```
// #####  
BOOL ReadByteAndSendNAK(void)  
{  
    dwNumBytesToSend = 0; // Clear output buffer  
  
    // Clock one byte of data in...  
    OutputBuffer[dwNumBytesToSend++] = 0x20; // Command: clock data byte in on clk rising edge  
    OutputBuffer[dwNumBytesToSend++] = 0x00; // Length  
    OutputBuffer[dwNumBytesToSend++] = 0x00; // Length 0x0000 means clock ONE byte in  
  
    // Now clock out one bit (ACK/NAK). This bit has value '1' to send a NAK to the I2C Slave  
    OutputBuffer[dwNumBytesToSend++] = 0x13; // Command: clock data bits out on clk falling edge  
    OutputBuffer[dwNumBytesToSend++] = 0x00; // Length of 0x00 means clock out ONE bit  
    OutputBuffer[dwNumBytesToSend++] = 0xFF; // Command will send bit 7 of this byte (= '1')  
  
    // Put I2C line back to idle (during transfer) state... Clock line low, Data line high  
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction/ data  
    OutputBuffer[dwNumBytesToSend++] = 0xFE; // Set the value of the pins  
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)  
    // AD0 (SCL) is output driven low  
    // AD1 (DATA OUT) is output high (open drain)  
    // AD2 (DATA IN) is input (therefore the output value specified is ignored)  
    // AD3 to AD7 are inputs driven high (not used in this application)  
  
    // This command then tells the MPSSE to send any results gathered back immediately  
    OutputBuffer[dwNumBytesToSend++] = 0x87; // Send answer back immediate command  
  
    // Send off the commands to the FT232H  
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);  
  
    // =====  
    // Now wait for the byte which we read to come back to the host PC  
    // =====  
    dwNumInputBuffer = 0;  
    ReadTimeoutCounter = 0;  
  
    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);  
    // Get number of bytes in the input buffer  
  
    while ((dwNumInputBuffer < 1) && (ftStatus == FT_OK) && (ReadTimeoutCounter < 500))  
    {  
        // Sit in this loop until  
        // (1) we receive the one byte expected  
        // or (2) a hardware error occurs causing the GetQueueStatus to return an error code  
        // or (3) we have checked 500 times and the expected byte is not coming  
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); // Get # bytes in buffer  
        ReadTimeoutCounter ++;  
        Sleep(1); // short delay  
    }  
  
    // If loop above exited due to the byte coming back (not an error code and not a timeout)  
    // then read the byte available and return True to indicate success  
    if ((ftStatus == FT_OK) && (ReadTimeoutCounter < 500))  
    {  
        ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);  
        ByteDataRead[0] = InputBuffer[0]; // store the data read  
        return True; // Indicate success  
    }  
    else  
    {  
        return False; // Failed to get any data back or got an error code back  
    }  
}
```

2.3.5 Read3BytesAndSendNAK()

In some cases, it is required to read multiple values from an I²C device after addressing the device. Some I²C Slaves support burst reads. For a normal single byte read, the Master would reply with a NAK after the byte has been read, to tell the slave that the read is complete. If however, the Master ACKs the byte being read and continues to do I²C reads, the Slave will send the next data value on each subsequent read until the Master NAKs the read. Examples include reading multiple consecutive addresses in an I²C EEPROM, or reading the device IDs from the FT201X (see section 3.1.2).

Note: The datasheet for the I²C slave must be consulted to determine if and how burst reads will work on that device.

This function is very similar to the ReadByteAndSendNAK() function above, but it reads three bytes instead of one. The ReadByteAndSendNAK() function above reads one byte and terminates the read at that point by NAKing the byte. The Read3BytesAndSendNAK() function effectively uses the same routine but calls it three times, once for each byte to be read. The important difference is that it responds with an ACK for the first two bytes and then a NAK for the final byte, so that three bytes are read in total.

For the first two byte reads, the 0x13 command is sending a '0' as its third parameter instead of 0xFF, which means that the SDA line is pulled low during the acknowledgement phase, and the Slave will interpret this as an ACK.

```
// Clock out one bit...send ack bit as '0'
OutputBuffer[dwNumBytesToSend++] = 0x13; // Command: clock data bits out on clk falling edge
OutputBuffer[dwNumBytesToSend++] = 0x00; // Length of 0x00 means clock out ONE bit
OutputBuffer[dwNumBytesToSend++] = 0x00; // Command will send bit 7 of this byte (= '0')
```

The other code in this function is not shown here but can be found in the full source code file.

2.3.6 SendByteAndCheckACK()

This function sends a byte passed to it, and then checks whether the I²C Slave responds with an ACK (driving SDA low during acknowledgement phase) or a NAK (SDA still pulled high during the acknowledgement phase).

The function is used for sending 8-bit data values. For sending the 7-bit address (or other times where the data is 7 bits plus R/W bit), refer to the function SendAddrAndCheckACK below.

The first command 0x11 causes the MPSSE to clock bytes out MSB first. The data is put on the line during the falling edge. The next two bytes added to the buffer are the length (number of bytes to be clocked out) which is set to 0x0000 in this application, which corresponds to clocking one byte out. The actual data to be clocked out follows. In this case only one byte is being sent, which is the value which was passed in when calling the SendByteAndCheckACK function.

A GPIO write then follows which ensures that the SDA line goes back to the idle (pulled up) state whilst SCK remains low. This allows the Slave device to control the SDA line during the acknowledgement phase.

The slave device will now be driving the SDA line to indicate a NAK (SDA high) or ACK (SDA low). The 0x22 command, which clocks in bits, is used to check this value. This command has a single parameter following it, which indicates the number of bits to clock in, in this case set to 0x00 meaning one bit. Because USB works in bytes, the MPSSE will send an entire byte back with the single bit which has been read in the LSB.

The 0x87 command tells the MPSSE to send this result back to the host PC over USB as soon as possible.

As with the Read Byte functions above, FT_GetQueueStatus is used to check for the byte coming back containing the ACK bit.

The function returns TRUE to the calling function if the I²C Slave has sent back an ACK (if the ACK bit is 0). It returns FALSE if the Slave has NAKed the transfer or if the byte containing the ACK bit failed to return from the FT232H. Additional error checking could be added to tell the calling function the reason for the FALSE value being returned.

It is important to note that this function sends the byte value passed to it without taking account of the Read/Write bit. When sending I²C addresses or where the data is 7 bits with R/W, the following function SendAddrAndCheckACK() function should be used (or the function which calls SendByteAndCheckACK must combine the 7-bit data value and the R/W bit and pass the resulting value to this function)

```
// #####
// Function to write 1 byte, and check if it returns an ACK or NACK by clocking in one bit
// We clock one byte out to the I2C Slave
// We then clock in one bit from the Slave which is the ACK/NAK bit
// Put lines back to the idle state (idle between start and stop is clock low, data
// Returns TRUE if the write was ACKed
```



```
// #####
BOOL SendByteAndCheckACK(BYTE dwDataSend)
{
    dwNumBytesToSend = 0;                // Clear output buffer
    FT_STATUS ftStatus = FT_OK;

    OutputBuffer[dwNumBytesToSend++] = 0x11; // command: clock bytes out MSB first on
                                           // clock falling edge
    OutputBuffer[dwNumBytesToSend++] = 0x00; //
    OutputBuffer[dwNumBytesToSend++] = 0x00; // Data length of 0x0000 means clock out 1 byte
    OutputBuffer[dwNumBytesToSend++] = dwDataSend; // Actual byte to clock out

    // Put I2C line back to idle (during transfer) state... Clock line low, Data line high
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADBus direction/ data
    OutputBuffer[dwNumBytesToSend++] = 0xFE; // Set the value of the pins
    OutputBuffer[dwNumBytesToSend++] = 0xFB; // Set pins o/p except bit 2 (data_in)

    // AD0 (SCL) is output driven low
    // AD1 (DATA OUT) is output high (open drain)
    // AD2 (DATA IN) is input (therefore the output value specified is ignored)
    // AD3 to AD7 are inputs driven high (not used in this application)

    OutputBuffer[dwNumBytesToSend++] = 0x22; // Command to clock in bits MSB first
                                           // on rising edge
    OutputBuffer[dwNumBytesToSend++] = 0x00; // Length of 0x00 means to scan in 1 bit

    // This command then tells the MPSSE to send any results gathered back immediately
    OutputBuffer[dwNumBytesToSend++] = 0x87; //Send answer back immediate command

    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
    //Send off the commands

    // =====
    // Now wait for the byte which we read to come back to the host PC
    // =====

    dwNumInputBuffer = 0;
    ReadTimeoutCounter = 0;

    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
    // Get number of bytes in the input buffer

    while ((dwNumInputBuffer < 1) && (ftStatus == FT_OK) && (ReadTimeoutCounter < 500))
    {
        // Sit in this loop until
        // (1) we receive the one byte expected
        // or (2) a hardware error occurs causing the GetQueueStatus to return an error code
        // or (3) we have checked 500 times and the expected byte is not coming
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); // Get # bytes in buffer
        ReadTimeoutCounter ++;
        Sleep(1); // short delay
    }

    // If loop above exited due to the byte coming back (not an error code and not a timeout)
    if ((ftStatus == FT_OK) && (ReadTimeoutCounter < 500))
    {
        ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);
        // Check if the returned bit was ack or nak
        if (((InputBuffer[0] & 0x01) == 0x00)) //Check ACK bit 0 on data byte read out
        {
            return TRUE; // Return True if the ACK was received
        }
        else
        {
            //printf("Failed to get ACK from I2C Slave \n");
            return FALSE; //Error, can't get the ACK bit
        }
    }
    else
    {
        return False; // Failed to get any data back or got an error code back
    }
}

```

2.3.7 SendAddrAndCheckACK()

This function is almost identical to the SendByteAndCheckACK() function above. The difference is that it takes two parameters, the address byte value and also the R/W bit value. It then calculates the combined byte.

I2C Address phase - Writing Example

When writing a data byte to the FT201X (assuming I2C address 0x22), the R/W bit will be '0'. The combined value will therefore be:

```
AddressByte = (I2CAddress << 1) AND 0xFE
             = (0010 0010 << 1) AND 1111 1110
             = 0100 0100 AND 1111 1110
             = 0100 0100
             = 0x44
```

SendAddrAndCheckACK(0x22,FALSE) will send the hex value 0x44 on the I2C bus.

I2C Address phase - Reading Example

When reading a data byte from the FT201X (assuming I2C address 0x22), the R/W bit will be '1'. The combined value will therefore be:

```
AddressByte = (I2CAddress << 1) OR 0x1
             = (0010 0010 << 1) OR 0000 0001
             = 0100 0100 OR 0000 0001
             = 0100 0101
             = 0x45
```

SendAddrAndCheckACK(0x22,TRUE) will send the hex value 0x45 on the I2C bus.

It can be noted that SendByteAndCheckACK() above would have to send 0x22 instead of the 0x44 and 0x45 shown here.

In terms of the code itself, this function differs only in the first few lines where the byte value to be sent is modified to combine the data and R/W value.

```
BOOL SendAddrAndCheckACK(BYTE dwDataSend, BOOL Read)
{
    dwNumBytesToSend = 0; // Clear output buffer
    FT_STATUS ftStatus = FT_OK;

    if(Read == True)
    {
        dwDataSend = ((dwDataSend << 1) | 0x01);
    }
    else
    {
        dwDataSend = ((dwDataSend << 1) & 0xFE);
    }
}
```

2.4 Other Considerations

This section briefly covers some other considerations when using the MPSSE for I²C.

Using other ADbus (Low Byte) lines as GPIO

When using the MPSSE for I²C, only lines AD0/1/2 are required. As demonstrated in this application note, the lines on the ACbus can be used as GPIO by using the 'Set Data Bits HighByte' command. It is also possible to use the ADbus lines AD3/4/5/6/7 as GPIO, provided that any writes to them retain the directions and levels used by the I2C lines.

One method would be to modify the functions given in section 2.3 to use a value passed into them instead of setting the upper 5 bits of port AD to inputs and leaving them unused.

The desired values and directions of the bits AD7 – AD3 would be set by the main application.

```
// Set the desired values and directions in the upper 5 bits
GPIO_Direction = 0xFF // Upper 5 bits of this contain the desired directions.
// Lower 3 bits will be ignored
GPIO_Value = 0x55 // Upper 5 bits of this contain the desired pin values
// Lower 3 bits will be ignored
```

Each I²C function from section 2.3 could then be modified to take these values as an input parameter. Some additional code in each function would need to mask off the lower three bits of the direction, and then combine the upper 5 GPIO bits with the lower 3 I²C bits to make a final value to write to the port.

As an example, the SetI2CLinesIdle function would be as shown below. Changes from the original function are shown in red.

```
void SetI2CLinesIdle(BYTE MyGPIO_Direction, BYTE MyGPIO_Value)
{
    dwNumBytesToSend = 0; //Clear output buffer

    PinDirection = MyGPIO_Direction | 0x03 & 0xFB; // Ensures xxxx x011
    PinData = MyGPIO_Value | 0x07; // Ensures xxxx x111

    // Set the idle states for the AD lines
    OutputBuffer[dwNumBytesToSend++] = 0x80; // Command to set ADbus direction and data
    OutputBuffer[dwNumBytesToSend++] = PinData; // Set data value
    OutputBuffer[dwNumBytesToSend++] = PinDirection; // Set directions
    ...
}
```

If any pins had been set as input, then a separate function would be needed to read their state. This would consist of sending a buffer with the Read Data Bits LowByte command (0x81) followed by the Send_Immediate command (0x87) and then reading the byte returned by the FT232H containing the pin values for all 8 bits of port AD.

It is important to note that all of the I²C functions shown in this application note would need to be modified in a similar way and with their particular values for the lower 3 bits, as otherwise they will change the values set above when they use the 0x80 command to set the I²C line states.

For this reason, it is easier to use Port AC as GPIO and to use port AD only for the I²C lines themselves unless the use of ACBUS is not possible due to the need for additional I/O line or if ACBUS is not available in the hardware.

FT2232H/FT4232H Code

When using the FT2232H and FT4232H, the drive-only-zero (open drain) feature is not available. Therefore, an additional GPIO command is needed before and after each I2C read/write so that the I2C Clock and Data_Out pins are set to input when not driving the lines. The application note AN_113 demonstrates this technique (see Appendix A – References).

Clock Stretching

Note that the MPSSE does not automatically support clock stretching for I²C. The MPSSE does provide instructions for reading the state of the ADBUS and this could be used to check the state of the I²C lines between transfers but clock stretching is not officially supported by the MPSSE.

Delays over USB

The routines presented here are used for reading and writing small amounts of data. One of the limiting factors is the need to write a byte out and then read in the ACK bit and return it to the PC to be checked. This example code reduces unnecessary delays by using the Send Immediate command to ensure that any data or ACK bit read is returned with minimal delay, instead of waiting for the latency timer.

If it is not required to check the ACK/NAK on every byte when writing to the slave, several writes could be buffered up before sending to the FT232H and so the FT232H will keep sending the data regardless of whether the slave ACKs or NAKs the last one. This depends on the I2C Slave device used and whether it is acceptable within the overall application to ignore the ACK/NAK bits.

Other Platforms

As shown throughout this section, the I²C routines involve using standard D2xx functions to send buffers of commands and read bytes back from the FT232H. The routines can therefore be ported over to any other operating systems which support the D2xx drivers. Likewise, the routines can be used in other languages such as Visual Basic and C# by modifying them to use the correct syntax for the required language.

This is a big advantage to controlling the MPSSE directly as opposed to using compiled DLLs, because the code is compatible with any OS/language which supports D2xx and can be easily ported into any code project which can use the basic D2xx read and write functions.

3 Main Program - Routines for Interfacing to FT201X

This section covers the main program loop. It uses the general I²C functions created above to communicate with the FT201X device. The FT201X device datasheet shows the sequences of I²C transfers which are required (see Appendix A – References).

When communicating with other types of I²C slave (such as I/O expanders and ADCs), this main program loop can be changed to reflect the protocol commands detailed by the datasheet for the I²C slave.

In this example, the main code does the following:

- Read the USB State from the FT201X
- Read the I²C IDs from the FT201X
- Read the MTP user area from 0x0000 to 0x007C
- Program an MTP location: Check the value in user area address 0x0024
 - o If address 0x0024 has value 0x00, program it to 0x55
 - o If address 0x0024 has value 0x55, program it to 0x00
- Read the MTP user area again to confirm that location 0x0024 changed
- Flush the buffers
- Wait for user to press a key
- Loop continuously:
 - o Use the Data Available command to determine if there is data to read
 - o If FT201X has any data to be read over I²C (i.e. sent from the PC to the FT201X)
 - Read one byte from it's buffer over I²C
 - Then write this byte back to the FT201X (which the FT201X will send back to the PC)
 - o 1 second delay to avoid the screen scrolling too quickly

Note that to improve readability, these routines do not perform the full set of error checking which would be recommended for final applications. For example, the return value from the functions used to send I²C data should be checked to determine whether the I²C acknowledged the byte sent or not.

3.1.1 Checking the USB State

This section of the code demonstrates the USB State command detailed in the FT201X device datasheet.

In general, this application begins an I²C transfer by ensuring the lines are set to their idle state and then setting the start condition on the lines.

Because the following byte will be a command to the FT201X and not data, the General Call Address (0x00) is used instead of the device's own address (0x22). The USB State command (0x16) is then sent which requests the USB State information.

A repeated start is created on the lines by a combination of the SetI2CLinesIdle and SetI2CStart functions.

The SendAddrAndCheckAck() function is used to send the device address (0x22) with the R/W bit set for reading. Note that as described in section 2.3.7 the resulting value sent over I²C is 0x45.

The ReadByteAndSendNAK() function will now clock in the 8 bits containing the USB state from the FT201X and will reply with a '1' for the ACK bit (i.e. NAKing the byte). This tells the FT201X that the FT232H does not wish to read any further bytes.

Finally, the Stop condition is written to the I²C lines to end this I²C transfer.

Further details on the USB States can be found in the FT201X datasheet and in Chapter 9 of the USB specification. In general, the value returned by this command will indicate the device in one of

the following states. The Configured state is the normal operating state and indicates that the FT201X is enumerated and ready.

Value	USB State
0x00	Suspended
0x01	Default
0x02	Addressed
0x03	Configured

Table 3.1 USB States

The waveforms of this I²C transfer are shown below:

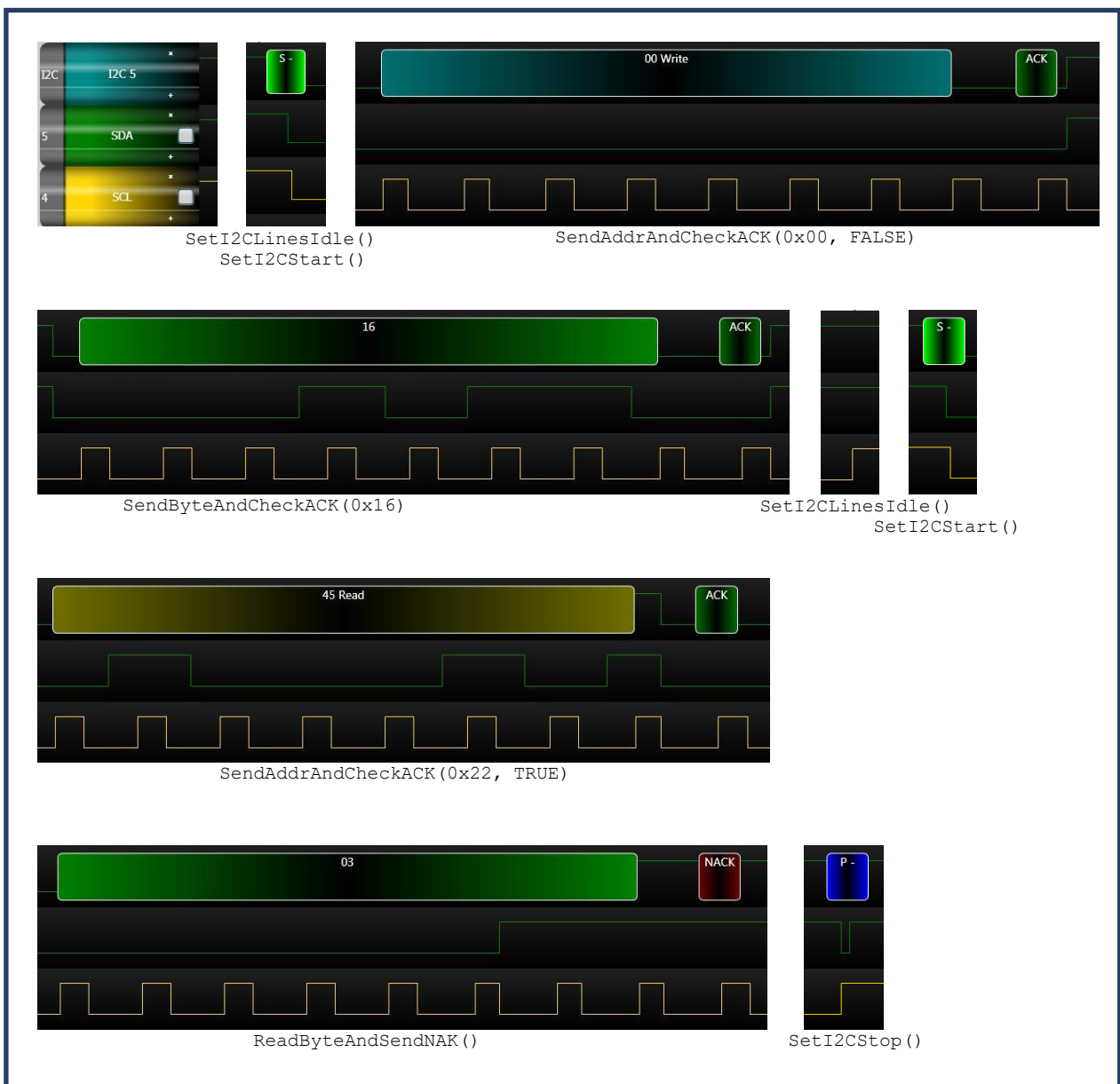


Figure 3.1 I²C Waveforms for Checking USB State

```
// #####
// Check the USB State
// #####

SetI2CLinesIdle();           // Set idle line condition
SetI2CStart();              // Set the start condition on the lines

bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send the general call address 0x00 wr (I2C = 0x00)
bSucceed = SendByteAndCheckACK(0x16);       // Send the USB State command

SetI2CLinesIdle();           // Set idle line condition as part of repeated start
SetI2CStart();              // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x22, TRUE); // Send the device address 0x22 rd (I2C = 0x45)
bSucceed = ReadByteAndSendNAK();           // Read 1 byte from the device, and send NAK

SetI2CStop();               // Send the stop condition

printf("USB State value is: %x\n", InputBuffer[0]); // Print the value returned
```

3.1.2 Reading the I²C IDs

This routine is almost identical to the USB State routine above, but sends the Device ID command. This causes the FT201X to return three bytes which contain the I²C IDs programmed into it.

Since the FT201X will have three bytes of data, the Read3BytesAndSendACK() function is used to read the IDs. It will send an ACK back to the FT201X for the first two bytes read, to indicate that more data is requested, and sends a NAK after the third byte to instruct the FT201X that no more bytes are requested. In this example, the data is returned in the first three elements of a global array called InputBuffer.

Note that if the code were to keep ACKing the bytes and doing further reads, the FT201X will wrap around and keep sending the three ID bytes back on subsequent reads.

```
// #####
// Read the I2C IDs
// #####

SetI2CLinesIdle();           // Set idle line condition
SetI2CStart();              // Send the start condition

bSucceed = SendAddrAndCheckACK(0x7C, FALSE); // Send the Device ID command 0x7C wr (I2C = 0xF8)
bSucceed = SendAddrAndCheckACK(0x22, TRUE);  // followed by the device address 0x22 rd (i2C = 0x45)

SetI2CLinesIdle();           // Set idle line condition as part of repeated start
SetI2CStart();              // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x7C, TRUE); // Send the Device ID command 0x7C rd (I2C = 0xF9)
Read3BytesAndSendNAK();       // Read 3 bytes, ACK first two and then NAK last one

SetI2CStop();               // Send the stop condition

printf("USB ID value 1 is: %x\n", InputBuffer[0]); // Print the result
printf("USB ID value 2 is: %x\n", InputBuffer[1]);
printf("USB ID value 3 is: %x\n", InputBuffer[2]);

printf("\n");
Sleep(100);
```

3.1.3 MTP Operations

This section demonstrates the MTP commands detailed in the FT201X datasheet. These commands allow an external device to read and write locations in the MTP memory in the device. This example shows the storing of application related data in a location in the user area of the MTP. This could allow an application to store data without the need to have an extra EEPROM chip on the board. In other cases, the chip configuration (e.g. serial number, strings, CBUS settings) could be set by writing to the appropriate memory locations as detailed in AN_201 (see Appendix A – References). This could be used instead of programming the MTP over USB with FT_Prog, if the programming is being carried out at a step in the board assembly where no computer is present.

It is important to note that the device must be awake (not in suspend) when carrying out MTP operations. If there is no PC connected to the USB side of the FT201X, then a CBUS pin can be configured as `Keep_Awake#` and pulled down to prevent the FT201X from going into suspend whilst it is being programmed.

Reading MTP Locations

This part of the example reads the MTP from locations 0x0000 to 0x007C and prints the value of each location on the host PC screen. This includes some of the configuration data (as detailed in AN201) and also one of the user areas. The actual sequence of I²C commands is taken from the FT201X datasheet, chapter 10 (see Appendix A – References).

For each location to be read, the following steps are used:

1. Ensure the lines are Idle and then set the start condition
2. Send the General Call Address (since a command to the FT201X will follow)
3. Send the MTP Address command
4. Repeated Start - Ensure the lines are Idle and then set the start condition
5. Send the device address 0x22 write (R/W = '0') (about to write the MTP memory address)
6. Send the Most Significant Byte of the address (always 0x00 in this example)
7. Send the Least Significant Byte of the address (changed each time through the loop)

8. Repeated Start - Ensure the lines are Idle and then set the start condition
9. Send the General Call Address (since a command to the FT201X will follow)
10. Send the MTP Read command
11. Repeated Start - Ensure the lines are Idle and then set the start condition
12. Send the device address with read (R/W = '1') (about to read the memory content)
13. Read one byte from the FT201X and NAK this byte (to tell FT201X that no more bytes requested)
14. Set the Stop condition on the lines

15. Store the byte read in an array.
16. Print the byte to the screen

```
// #####
// Reading values from the MTP Memory locations 0x00 to 0x7C (byte addressing)
// #####

unsigned char MTP_ReadLoop = 0;
unsigned char MemBuffer[0x7C];
unsigned char ValueToWrite = 0x00;

for(MTP_ReadLoop = 0x00; MTP_ReadLoop <0x7C; MTP_ReadLoop ++)
{
    SetI2CLinesIdle(); // Set idle line condition
    SetI2CStart(); // Send the start condition

    bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send general call addr 0x00 wr (I2C = 0x00)
    bSucceed = SendByteAndCheckACK(0x10); // Send the MTP address command 0x10

    SetI2CLinesIdle(); // Set idle line condition as part of repeated start
    SetI2CStart(); // Set the start condition as part of repeated start

    bSucceed = SendAddrAndCheckACK(0x22, FALSE); // Send the device address 0x22 wr (I2C = 0x44)
    bSucceed = SendByteAndCheckACK(0x00); // Send the Most Significant byte of the MTP address
    bSucceed = SendByteAndCheckACK(MTP_ReadLoop); // Send the LS byte of the MTP address

    SetI2CLinesIdle(); // Set idle line condition as part of repeated start
    SetI2CStart(); // Send the start condition as part of repeated start

    bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send general call addr 0x00 wr (I2C = 0x00)
    bSucceed = SendByteAndCheckACK(0x14); // Send the MTP read command 0x14

    SetI2CLinesIdle(); // Set idle line condition as part of repeated start
    SetI2CStart(); // Send the start condition as part of repeated start

    bSucceed = SendAddrAndCheckACK(0x22, TRUE); // Send the device address 0x22 rd (I2C = 0x45)
    ReadByteAndSendNAK(); // Read 1 byte from the device, and send NAK

    SetI2CStop(); // Send the stop condition

    MemBuffer[MTP_ReadLoop] = InputBuffer[0];
}
```



```

Sleep(20);
printf("MTP byte location %x value is: %x\n", MTP_ReadLoop, MemBuffer[MTP_ReadLoop]);
}

```

Writing MTP Locations

The code which writes location 0x0024 is very similar to the reading code above. The differences are:

- In Step 10, send the MTP Write command instead of the MTP Read command
- In Step 12, send the device address with the write bit (R/W = '0') since it will be writing a value instead of reading
- In Step 13, write one byte to the FT201X instead of reading a byte

```

// #####
// Write a value to the MTP Memory
// #####

SetI2CLinesIdle();           // Set idle line condition
SetI2CStart();              // Send the start condition

bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send the general call address 0x00 wr (I2C = 0x00)
bSucceed = SendByteAndCheckACK(0x10);       // Send the MTP address command 0x10

SetI2CLinesIdle();           // Set idle line condition as part of repeated start
SetI2CStart();              // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x22, FALSE); // Send the device address 0x22 wr (I2C = 0x44)
bSucceed = SendByteAndCheckACK(0x00);       // Send the Most Significant byte of the MTP address
bSucceed = SendByteAndCheckACK(0x24);       // Send the LS byte (writing to 0x24 here)

SetI2CLinesIdle();           // Set idle line condition as part of repeated start
SetI2CStart();              // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send the general call address 0x00 wr (I2C = 0x00)
bSucceed = SendByteAndCheckACK(0x12);       // Send the MTP Write command 0x12

SetI2CLinesIdle();           // Set idle line condition as part of repeated start
SetI2CStart();              // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x22, FALSE); // Send the device address 0x22 wr (I2C = 0x44)

if(MemBuffer[0x24] == 0x00) // Toggle the byte to be written so that we alternate
    ValueToWrite = 0x55;    // between 0x00 and 0x55 each time the program is run
else
    ValueToWrite = 0x00;

bSucceed = SendByteAndCheckACK(ValueToWrite); // Write out the byte (0x00 or 0x55) to the MTP

SetI2CStop();              // Send the stop condition

```

The Reading routine is then called again in the example code, to list the contents to the screen once more. Comparing the two sets of reads will show that location 0x0024 has changed from 0x00 to 0x55 or from 0x55 to 0x00 (depending on whether it was 0x55 or 0x00 initially)

3.1.4 Buffer Flush

This command uses the General Call Address since it is a command to the FT201X chip. The command has no parameters and so is simply a write of the General Call Address followed by a write of the command itself 0x0E.

```

// #####
// Flush the buffer
// #####

SetI2CLinesIdle();           // Set idle line condition
SetI2CStart();              // Set the start condition on the lines

```

```
bSucceed = SendAddrAndCheckACK(0x00, FALSE); // Send the general call address 0x00 wr (I2C = 0x00)
bSucceed = SendByteAndCheckACK(0x0E); // Send the Flush Buffer command

SetI2CStop(); // Send the stop condition

printf("Buffer flushed\n");

Sleep(100);
```

3.1.5 Reading and Writing Data

The code now sits in an infinite loop, checking for any data using the Read Data Available command. If the FT201X reports that it has one or more bytes in the buffer, the application will do a read over I²C to read the first byte from the buffer. It will then write that byte back to the FT201X and the FT201X will subsequently send that byte back to it's host computer over USB. The code for the FT232H is now therefore behaving as a loopback – any data which a user types into the FT201X's terminal will be echoed back to the terminal.



- Read Data Available
- If data available > 0
 - Read one byte from the FT201X
 - Print the byte to the screen
 - Write that byte back to the FT201X

It can be noted that this code reads one byte from the buffer each time round the loop. If the user types 'Hello' into the FT201X's terminal window, the FT232H will take five cycles of the loop to read and echo all five characters. This method has been used to ease the readability of the code, but in a real application the I²C Master could read multiple bytes at a time.

Checking the number of bytes available:

The Data Available command returns the number of bytes in the buffer of the FT201X chip. When the host computer sends any data to the FT201X chip (e.g. a user typing a character in a terminal program), the FT201X will hold this data in it's buffer ready for the I²C Master to read over I²C.

The main steps are:

- Ensure the lines are Idle and then set the start condition
- Send the General Call Address ([since a command to the FT201X will follow](#))
- Send the Data Available command
- Repeated Start - Ensure the lines are Idle and then set the start condition
- Send the device address 0x22 with read (R/W = '1')
- Read one byte from the FT201X and NAK this byte.
- Print the number of bytes available on the console screen
- Set a software flag to tell the following read routine whether any bytes are available

```
while(1)
{
// *****
// Reading the Data Available register of an FT201X device on the I2C bus
// *****

SetI2CLinesIdle(); // Set idle line condition
SetI2CStart(); // Send the start condition

bSucceed = SendAddrAndCheckACK(0x00, FALSE); // General call address 0x00 wr (I2C = 0x00)
bSucceed = SendByteAndCheckACK(0x0C); // Send the DataAvailable command 0x0C

SetI2CLinesIdle(); // Set idle line condition as part of repeated start
SetI2CStart(); // Send the start condition as part of repeated start

bSucceed = SendAddrAndCheckACK(0x22, TRUE); // Send the device address 0x22 rd (I2C = 0x45)
ReadByteAndSendNAK(); // Read 1 byte from the device, and send NAK

SetI2CStop(); // Send the stop condition
```

```
printf("ReadDataAvail value is: %x\n", InputBuffer[0]);
printf("\n");

// Set a flag to show whether there were any bytes available so that we can read them next
if(InputBuffer[0] > 0)
    DataInBuffer = TRUE;
else
    DataInBuffer = FALSE;
```

Reading a byte from the FT201X:

If any data is available to read, now read one byte from the buffer. The main steps are:

- Ensure the lines are Idle and then set the start condition
- Send the device address with read (R/W = '1')
- Read one byte from the FT201X and NAK this byte.
- Store the byte read in an array.

```
// #####
// If data is available, now read a byte from the buffer
// #####

if(DataInBuffer == TRUE)
{
    SetI2CLinesIdle();           // Set idle line condition
    SetI2CStart();              // Send the start condition

    bSucceed = SendAddrAndCheckACK(0x22, TRUE); // Send the device address 0x22 rd (I2C = 0x45)
    ReadByteAndSendNAK();       // Read 1 byte from the device, and send NAK

    SetI2CStop();               // Send the stop condition

    DataByte = InputBuffer[0];

    printf("Data value (in hex ascii) is: %x\n", DataByte);
    printf("Data value (as character) is: %c\n", DataByte);
    printf("\n");
}
}
```

Writing a byte to the FT201X:

Write the last received byte back to the FT201X over I²C. The main steps are:

- Ensure the lines are Idle and then set the start condition
- Send the device address with write (R/W = '0')
- Write one byte to the FT201X

```
// #####
// Now send the character which we just read back to the FT201X
// #####

if(DataInBuffer == TRUE)
{
    SetI2CLinesIdle();           // Set idle line condition
    SetI2CStart();              // Send the start condition

    bSucceed = SendAddrAndCheckACK(0x22, FALSE); // Send the device address 0x22 wr (I2C = 0x44)
    bSucceed = SendByteAndCheckACK(byte(DataByte)); // Send the byte (which has just been read
                                                    // above) back to the FT201X

    SetI2CStop();               // Send the stop condition

    printf("Sent the received character back to the FT201X\n");
    printf("\n");
}
}
```

4 FT201X USB to I²C Slave Interface

The FT201X is a dedicated USB to I²C Slave bridge. One common application is to transfer data between an MCU and a computer over USB, for example as shown below:

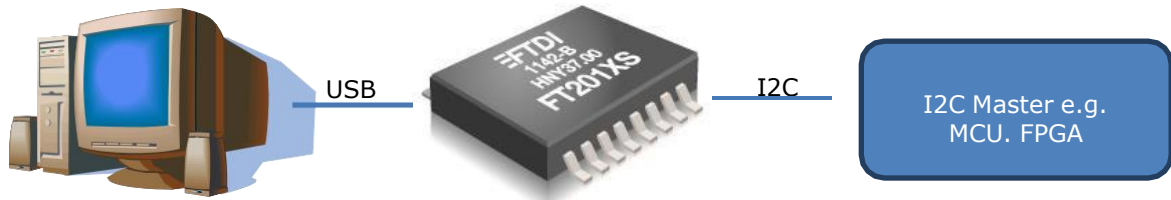


Figure 4.1 FT201X USB to I²C Slave Interface

As a true hardware bridge, the FT201X does not require the PC application to assist in generating the I²C protocol and the IC does this completely in hardware.

4.1 Device Hardware settings

The FT201X is ready for use as a USB to I²C Slave interface without any configuration. The device does not need any special configuration in order to use in this demonstration. If the module has been re-programmed or re-configured in any way, the default settings may have changed. The main assumptions are:

- The I²C address is assumed to be the default of 0x22 in the demonstration provided with this application note.
- The CBUS settings are not critical for this example but if using the FT201X as a self-powered design which needs to support access to MTP over I²C whilst not connected to a USB host, please see the note below.

It is important to note that the FT201X device must be awake (not in suspend) when carrying out MTP operations. If there is no PC connected to the USB side of the FT201X, it would normally be in suspend. In this case, a CBUS pin can be configured as Keep_Awake# and pulled down to prevent the FT-201X from going into suspend, allowing the external I²C Master to read and write the MTP over I²C. For example, an MCU can pull this line down using one of its GPIO pins.

4.2 I²C Interface

For the I²C Interface, Chapter 9 of the FT201X datasheet (see Appendix A – References) gives details of both data transferring and also the commands used to check the device status (Read Data Available etc.). Chapter 10 of the datasheet also details the sequence used when programming the MTP over I²C.

In general, a read or write to the device's I²C address is used to read and write data. There are several specific sequences in chapter 9 which use the General Call Address (0x00) to send commands instead of data to the FT201X. These are demonstrated in the example code provided with this application note and are also covered in section 3 of this application note.

4.3 USB Interface

From the PC side, the device can communicate in two ways:

- Virtual COM Port
- D2XX Interface

In either case, any bytes sent to the FT201X over USB (i.e. written to the virtual com port or sent by the D2xx FT_Write command) will simply appear in the FT201X's I²C buffer for the I²C Master to read. As an I²C Slave, the FT201X requires the I²C Master to initiate the read of the data.

Likewise, if the I²C Master writes data to the FT201X over I²C, then the FT201X will send this over USB to the host computer where the FTDI driver will buffer it, ready for the application to read. There is no I²C related protocol to strip out as the chip handles this itself and the data read from the FTDI driver will be the actual byte values which had been written over I2C.

Virtual COM Port

The FT201X and FT200XD devices may be supplied with the MTP programmed for D2xx only. If it is desired to use the VCP, a setting can be changed to enable it.

First of all, check the Device Manager. The device will always appear under the Universal Serial Bus Controllers once installed. If it does not appear under Ports (COM & LPT) then the VCP layer must be enabled by either ticking the 'Load VCP box' in the device's advanced properties (which will enable it on this computer only) or using FT_Prog to program this setting in the actual device's MPT memory (which will cause that device to load the VCP on any computer to which it is connected).

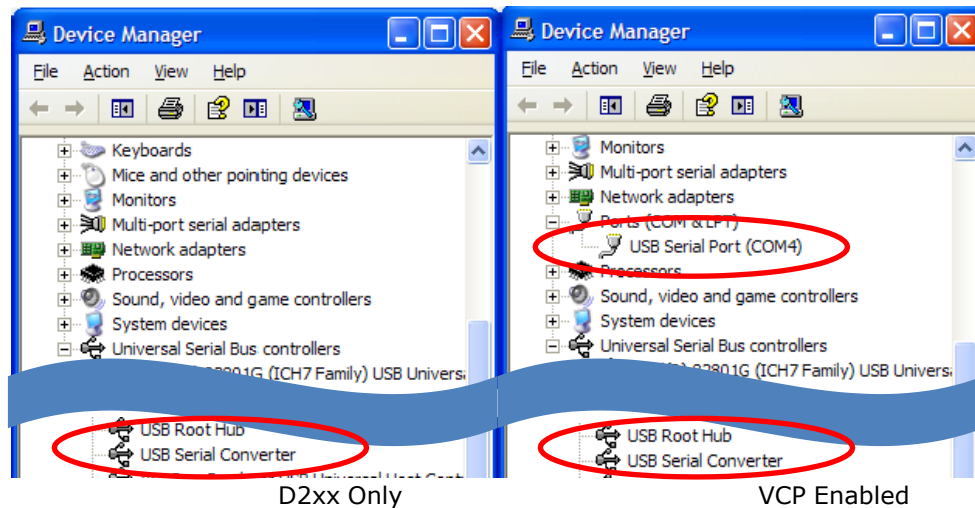


Figure 4.2 Device Manager showing D2xx and VCP configurations

The COM port associated with the device can then be opened using a terminal program or an application written using standard COM port functions.

D2XX

The application can use D2xx functions instead of the Virtual COM Port to interface with the FT201X. A small example is provided below for this. Note that the demonstration provided in this application note uses a D2xx terminal program (which can be found in the AN_255 Files zip file provided) because the focus of the demonstration is the FT232H code. However, a D2xx program similar to the one below could be used instead.

```
int main(int argc, char* argv[])
{
    FT_HANDLE fthandle;
    FT_STATUS status;
    DWORD numdev = 0;

    // Check how many FTDI devices are connected and installed. If one or more connected, open the first one
    status = FT_CreateDeviceInfoList(&numdev);

    if ((status == FT_OK) && (numdev > 0) )
    {
```

```
// Open the device now
status = FT_Open(0, &fthandle);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Set the In transfer size. Ideally, use a small value if receiving a few bytes at a time or a larger
// value (up to 64K) if transferring large amounts of data
status = FT_SetUSBParameters(fthandle, 256, 0);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Reset the device
status = FT_ResetDevice(fthandle);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Set the handshaking mode in the driver, for I2C chips this has no affect on the external I2C
// interface as it does not have handshake lines but this enables internal handshake in the driver
status = FT_SetFlowControl(fthandle, FT_FLOW_RTS_CTS, FT_STOP_BITS_1, FT_PARITY_NONE);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Set Timeouts to 5 seconds to ensure a Read or Write will return if unable to be completed
status = FT_SetTimeouts(fthandle, 5000, 5000);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Set Latency Timer (keeping it at default of 16ms here)
status = FT_SetLatencyTimer(fthandle, 16);
if(status != FT_OK)
    printf("status not ok %d\n", status);

// Now write some data to the chips buffer
char data_out[12] = "HELLO WORLD";
DWORD w_data_len = 12;
DWORD data_written;

status = FT_Write(fthandle, data_out, w_data_len, &data_written);
if(status != FT_OK)
    printf("status not ok %d\n", status);
else
    printf("12 Bytes Sent, waiting for bytes to come back\n");

// The I2C Master should now be able to read these 12 bytes from the FT201X over I2C.
// This example expects the I2C Master to send the bytes back over I2C to the FT201X

// Now read the data which the I2C master has written to the FT201X
char data_in[12];
DWORD data_read;
DWORD MyBytesReceived = 0;
DWORD SoftwareTimeout = 0;

// Wait for the FT201X to send our 12 bytes back to the PC
while((MyBytesReceived < 12) && (SoftwareTimeout < 500))
{
    FT_GetQueueStatus(fthandle, &MyBytesReceived);
    Sleep(1);
    SoftwareTimeout++;
}

// Check if the loop exited due to timing out or receiving 12 bytes
if(SoftwareTimeout == 500)
{
    printf("Timed out waiting for data\n");
}
else
{
    // Now read the received bytes
    status = FT_Read(fthandle, data_in, MyBytesReceived, &data_read);
    if(status != FT_OK)
        printf("status not ok %d\n", status);
    else
        printf("data read %s\n", data_in);
}

// Close the device
status = FT_Close(fthandle);
}
else
{
    printf("No FTDI devices connected to the computer \n");
}
}
...

```


5 Hardware configurations

FTDI have a variety of development modules and cables available for both the FT232H and the FT201X. This example uses the UM232H and the UM201XE dual-in-line modules and needs only the addition of two pull-up resistors for the I2C bus and an optional LED with series resistor. The hardware can be easily constructed on a small breadboard.

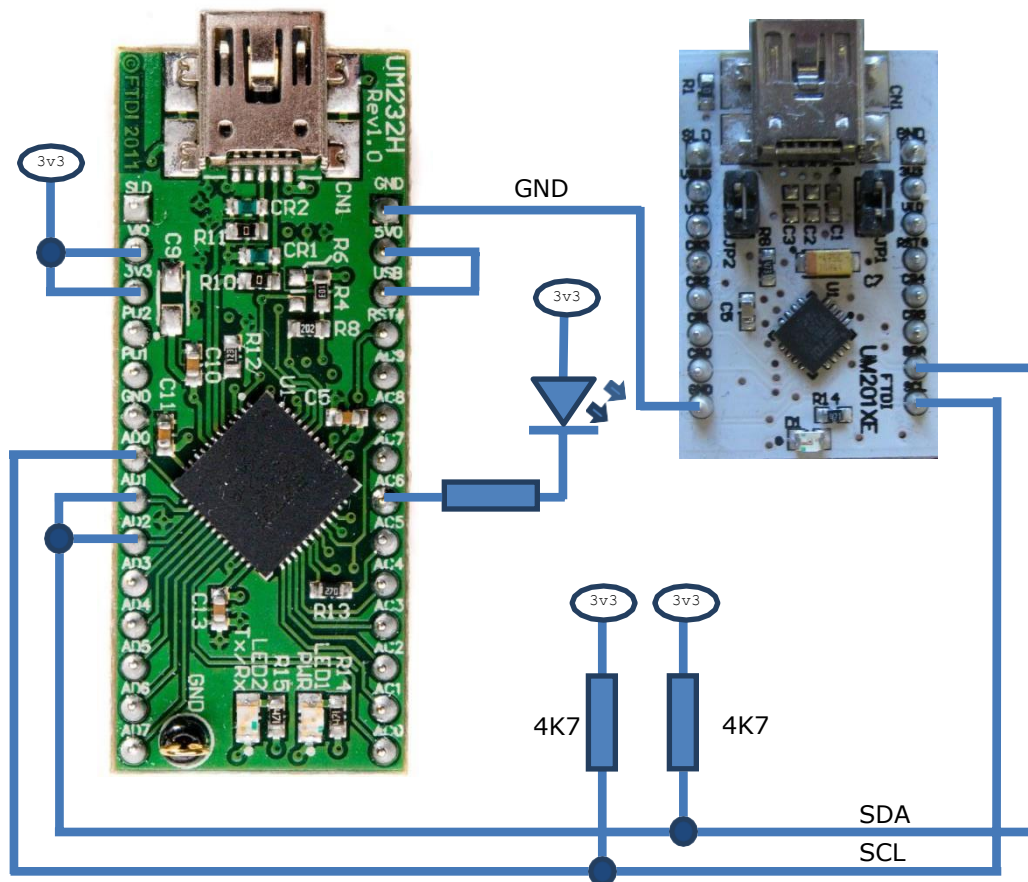


Figure 5.1 Schematic using UM232H and UM201XE

Note that FTDI have several other modules which feature the FT232H and FT201X, and these could also be used. Examples include the C232HM-DDHSL-0, UM232H-B and UM201XB below:



Figure 5.2 Some of the FT232H and FT201X development modules available

UM232H Connections

The FT232H has fixed assignments for the MPSSE Clock, Data Out and Data In. These pins are configured as open-drain in the device by the application code as required by the I²C protocol but the I²C SDA and SCL require an external pull-up resistor.

For the UM232H, the following lines are used in this example:

Pin	Software configuration	Description
AD0	Output, drive-only-zero enabled by commands from PC software	Clock Output from MPSSE
AD1	Output, drive-only-zero enabled by commands from PC software	Data_Out from MPSSE
AD2	Set to Input by command from PC software	Data_In line of MPSSE
AC6	Output enabled by commands from PC software	Drives the status LED which is controlled by the GPIO instructions in the I ² C Routines

Note: AD1 and AD2 are connected together to form a combined In/Out SDA line

VIO (VCCIO input) is linked to the 3v3 output from the FT232H's internal regulator, to power the I/O supply of the FT232H. This supply is also used externally to the module for the I2C pull-up resistors and LED power.

The USB supply is also linked to the modules 5V0 input pin so that the module gets its power from the USB supply.

UM201XE Connections

For the UM201XE, the following lines are used:

Pin	Pin Configuration	Description
SCL	I2C bus clock pin	Note: This pin has fixed assignment as SCL in hardware
SDA	I2C bus data pin	Note: This pin has fixed assignment as SDA in hardware
CB0	Set in MTP to the Keep_Awake# mode	CBUS pin, set to Keep_Awake function by programming the MTP ROM. Pull to ground if communicating with the FT201X over I2C whilst it is powered but not connected to a PC.

Note: CB0 is programmed as Keep Awake# as detailed in section 4.

It is assumed that the on-board jumpers JP1 and JP2 are set so that VCC is powered from USB Vbus and VCCIO is powered from the devices internal 3v3 regulator. If these jumpers are not populated, then additional connections are required to those shown in the figure above, to power the device.

6 Running the Example Code

This section covers the running of the provided demo code. The following steps assume that the UM232H and UM201X modules are connected to the PC and the drivers are loaded.

1. Open the FT232H to FT201X I2C Example project in Visual Studio 2008 or later.

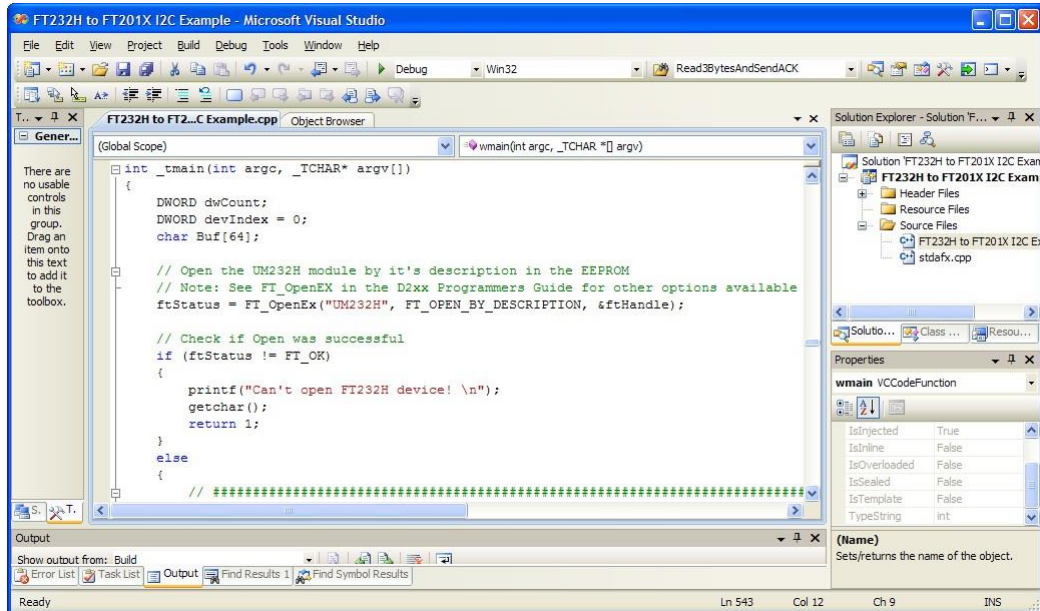


Figure 6.1 Opening the sample code in Visual Studio (FT232H I2C Master application)

2. Open the D2xx TTY program provided in the zip file.
3. Select the FT201X module from the 'Select Device' drop-down and click Open.

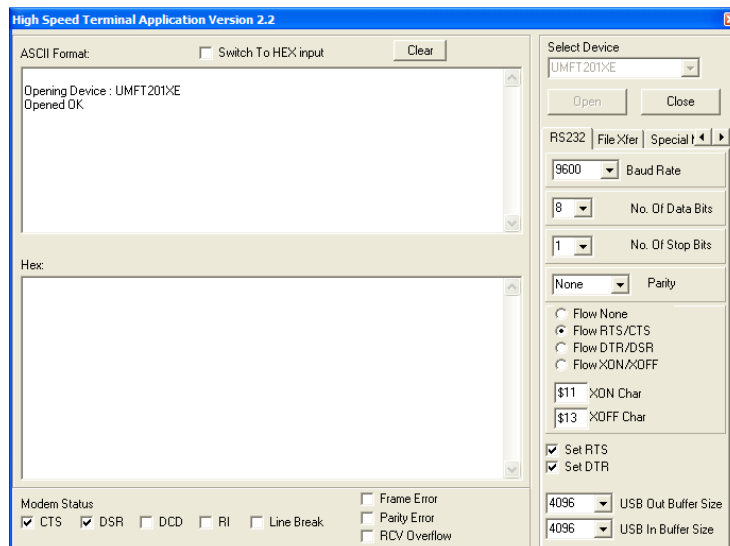


Figure 6.2 Terminal used for the FT201X I2C Slave

4. Go back to the Visual Studio project and start debugging. The console window screen will display the USB State, USB ID and MTP memory values as they are read.

```
Successfully opened FT232H device!
MPSSE synchronized with BAD command
USB State value is: 3
USB ID value 1 is: 0
USB ID value 2 is: 0
USB ID value 3 is: 29

MTP byte location 0 value is: 40
MTP byte location 1 value is: 0
MTP byte location 2 value is: 3
MTP byte location 3 value is: 4
MTP byte location 4 value is: 15
MTP byte location 5 value is: 60
```

Figure 6.3 The FT232H application displays information read from the FT201X over I²C

- In the MTP section, the code will read the memory contents twice, and will have toggled location 0x24 from 0x00 to 0x55 in between (or 0x55 to 0x00 if it was 0x55 initially).

MTP byte location 21 value is: 0	MTP byte location 21 value is: 0
MTP byte location 22 value is: 0	MTP byte location 22 value is: 0
MTP byte location 23 value is: 0	MTP byte location 23 value is: 0
MTP byte location 24 value is: 0	MTP byte location 24 value is: 55
MTP byte location 25 value is: 0	MTP byte location 25 value is: 0
MTP byte location 26 value is: 0	MTP byte location 26 value is: 0
MTP byte location 27 value is: 0	MTP byte location 27 value is: 0
MTP byte location 28 value is: 0	MTP byte location 28 value is: 0

Figure 6.4 The MTP memory location 0x24 before and after programming

- After the second MTP read, the program will wait for a keypress before proceeding to the section which reads and writes data.

```
MTP byte location 7a value is: 0
MTP byte location 7b value is: 0
Buffer flushed
Press a key to continue to the reading/writing data mode
_
```

Figure 6.5 Entering the loop which will read data from the FT201X and write it back

- The device will now sit in a loop, asking the FT201X if it has any data available to read.

```
MTP byte location 7b value is: 0
Buffer flushed
Press a key to continue to the reading/writing data mode

ReadDataAvail value is: 0
-----
ReadDataAvail value is: 0
```

Figure 6.6 FT232H checking for data available in the FT201X

- Now go back to the Terminal which is linked to the FT201X. Type a few characters into the terminal and press enter. For example, 'Hello' was typed below.

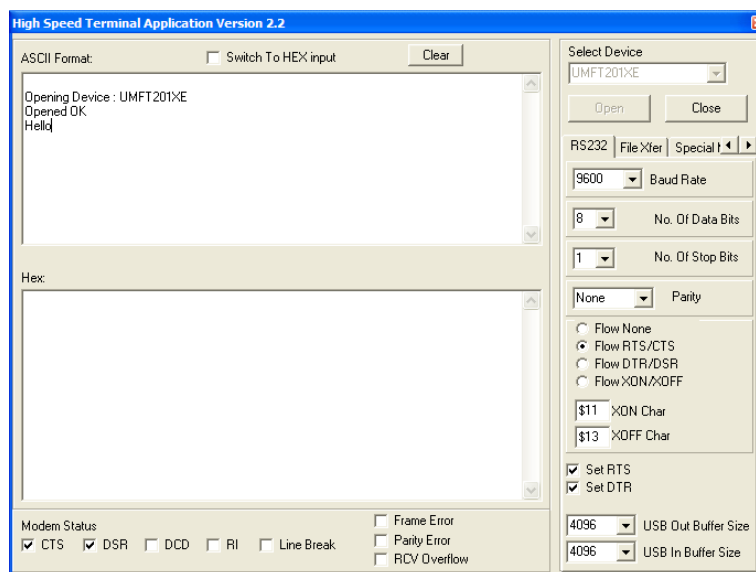


Figure 6.7 Sending some data to the FT201X over USB

9. Looking at the console window, it can be seen that the FT201X is now reporting 6 bytes available (Hello plus a carriage return). The FT232H will read one byte and display it, 0x48 or ASCII 'H' below, and then send it back to the FT201X. The character 'H' will also now appear on the Terminal window. The FT232H program in the console window will continue to read one byte at a time from the FT201X, display it and return it to the FT201X.

The console window of the FT232H application is shown alongside the Terminal window of the FT201X below:

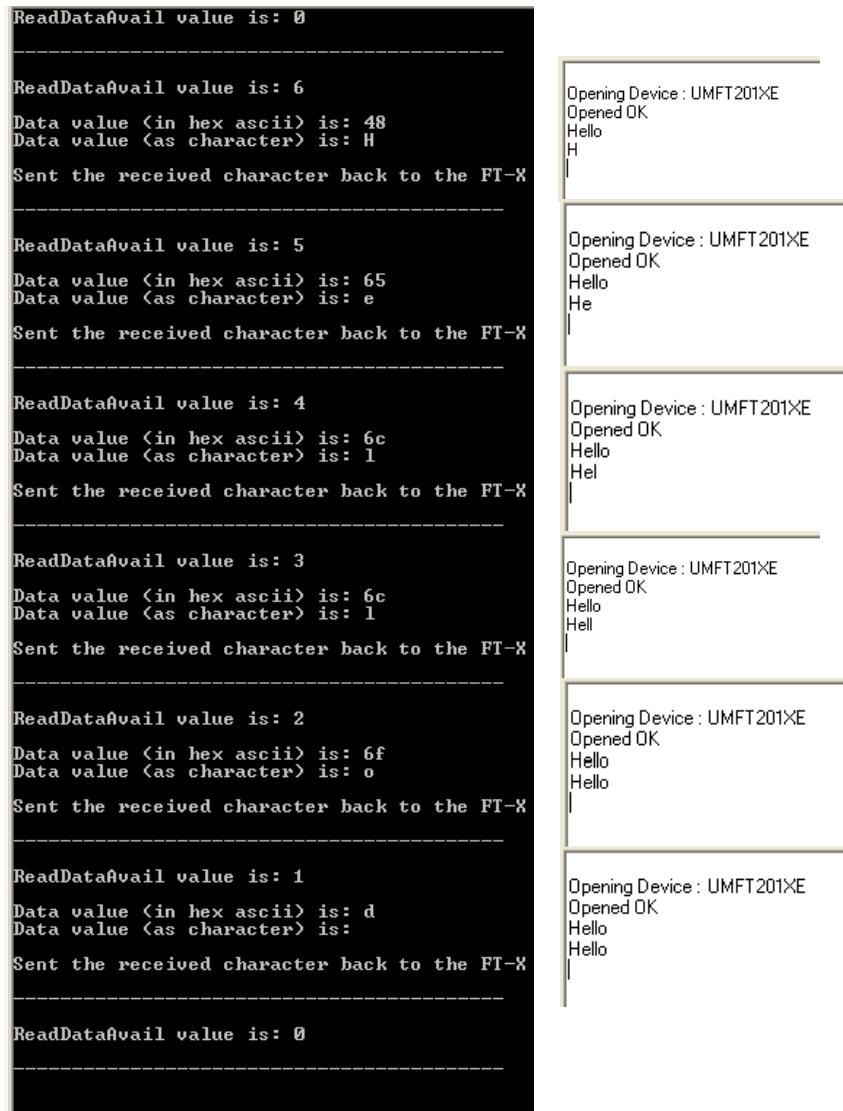


Figure 6.8 FT232H reads the data from the FT201X one byte at a time in this demo

10. Finally, close both devices. Close the FT201X port in the terminal program using the 'Close' button. Use the Stop Debugging option in Visual Studio to terminate the FT232H I²C Master program.

7 Conclusion

This application note has demonstrated the use of the FT232H as an I²C Master and provided some example code which can be used to develop applications for communicating with a variety of I²C slaves.

It has also given a practical example of how an I2C Master would use the command set of the FT201X over I²C to perform tasks such as data transfer and status commands.

It has provided some background information on using the FT201X I²C slave device from the host PC perspective.

Finally, it has shown how the three elements above can be brought together to produce a working loop-back demonstration.

The same principles can be applied to create many other real-world applications using either the FTx232H in a USB-to-I²C-Master design or the FT200/201X in a USB-to-I²C-Slave design.

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

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Future Technology Devices International Ltd (FTDI) devices incorporated in their systems, meet all applicable safety, regulatory and system-level performance requirements. All application-related information in this document (including application descriptions, suggested FTDI devices and other materials) is provided for reference only. While FTDI has taken care to assure it is accurate, this information is subject to customer confirmation, and FTDI disclaims all liability for system designs and for any applications assistance provided by FTDI. Use of FTDI devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless FTDI from any and all damages, claims, suits or expense resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder. Future Technology Devices International Ltd, Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow G41 1HH, United Kingdom. Scotland Registered Company Number: SC136640

Appendix A – References

Document References

Sample code

http://www.ftdichip.com/Support/SoftwareExamples/MPSSE.htm/AN_255_Files.zip

UM232H Datasheet

http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_UM232H.pdf

FT232H IC Datasheet

http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232H.pdf

Command Set for the MPSSE

http://www.ftdichip.com/Support/Documents/AppNotes/AN_108_Command_Processor_for_MPSSE_and_MCU_Host_Bus_Emulation_Modes.pdf

I2C Example for FT2232H and FT4232H

http://www.ftdichip.com/Support/Documents/AppNotes/AN_113_FTDI_Hi_Speed_USB_To_I2C_Example.pdf

C232HM-EDHSL-0 MPSSE Cables (alternative to UM232H)

http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_C232HM_MPSSE_CABLE.pdf

UM201XE Datasheet

http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_UMFT201XE.pdf

FT201X Datasheet

http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT201X.pdf

FT201X MTP Configuration

http://www.ftdichip.com/Support/Documents/AppNotes/AN_201_FT-X%20MTP%20Memory%20Configuration.pdf

D2xx Programmers Guide

[http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide\(FT_00071\).pdf](http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide(FT_00071).pdf)

Latest FTDI Drivers

<http://www.ftdichip.com/Drivers/D2XX.htm>

FT_Prog Utility

http://www.ftdichip.com/Support/Utilities.htm#FT_Prog

Acronyms and Abbreviations

Terms	Description
GPIO	General Purpose Input / Output
I ² C	Inter-Integrated Circuit
MCU	Microcontroller
SPI	Serial Peripheral Interface
USB	Universal Serial Bus



Appendix B – List of Tables & Figures

List of Tables

Table 2.1 Example I ² C Master functions presented in this application note.....	10
Table 3.1 USB States.....	21

List of Figures

Figure 1.1 Overview of the I ² C Demonstration	3
Figure 1.2 Overview of the FT232H I ² C Master application.....	4
Figure 2.1 FT232H as USB to I ² C Interface	5
Figure 2.2 Two- and Three-Phase Clocking Comparison.....	8
Figure 2.3 I ² C Start Condition	11
Figure 2.4 I2C Stop Condition.....	12
Figure 3.1 I ² C Waveforms for Checking USB State.....	21
Figure 4.1 FT201X USB to I ² C Slave Interface.....	27
Figure 4.2 Device Manager showing D2xx and VCP configurations	28
Figure 5.1 Schematic using UM232H and UM201XE	30
Figure 5.2 Some of the FT232H and FT201X development modules available	30
Figure 6.1 Opening the sample code in Visual Studio (FT232H I2C Master application).....	32
Figure 6.2 Terminal used for the FT201X I2C Slave	32
Figure 6.3 The FT232H application displays information read from the FT201X over I ² C	33
Figure 6.4 The MTP memory location 0x24 before and after programming.....	33
Figure 6.5 Entering the loop which will read data from the FT201X and write it back.....	33
Figure 6.6 FT232H checking for data available in the FT201X.....	33
Figure 6.7 Sending some data to the FT201X over USB.....	33
Figure 6.8 FT232H reads the data from the FT201X one byte at a time in this demo	34

Appendix C – Revision History

Document Title: AN_255 USB to I2C Example using the FT232H and FT201X devices
Document Reference No.: FT_000883
Clearance No.: FTDI# 352
Product Page: <http://www.ftdichip.com/FTProducts.htm>
Document Feedback: [Send Feedback](#)

Revision	Changes	Date
1.0	Initial Release	2013-09-30