Future Technology Devices International Ltd.

Application Note

AN_135

FTDI MPSSE Basics

Document Reference No.: FT_000208

Version 1.1

Issue Date: 2010-03-12

The FTDI Multi-Protocol Synchronous Serial Engine (MPSSE) provides a flexible means of interfacing synchronous serial devices to a USB port. This document provides a basic discussion of the FTDI MPSSE, how to configure it for use and establish the signalling required for synchronous communication.

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
E-Mail (Support): **support1@ftdichip.com** Web: http://www.ftdichip.com

Table of Contents

1 Introduction 4
1.1 Required Components 4
1.2 Master/Slave Association 4
2 Hardware Connections 5
2.1 Suggested Connections 7
2.1.1 SPI – Single Slave 7
2.1.2 SPI – Multiple Slaves 7
2.1.3 I^2C – Single Slave 8
2.1.4 I^2C – Multiple Slaves 8
2.1.5 JTAG 8
3 Serial Protocol Configuration 9
3.1 MSB/LSB & Data Length 9
3.2 Clocks 9
3.2.1 Divisors 9
3.2.2 Edges 10
3.2.3 Special Cases 10
3.3 Miscellaneous Details 11
3.3.1 Initial Pin States 11
3.3.2 Buffer Sizes 11
4 Software Configuration 12
4.1 Confirm device existence and open handle 12
4.2 Configure FTDI Port For MPSSE Use 12
4.3 Configure MPSSE 13
4.3.1 Synchronization & Bad Command Detection 13
4.3.2 MPSSE Setup 13
4.4 Serial Communications 13
4.5 GPIO Access 14
4.6 Close handle 14
5 Example Program 15
5.1 Confirm device existence and open handle 15
5.2 Configure FTDI Port For MPSSE Use 16
5.3 Configure the FTDI MPSSE 16
5.3.1 Synchronization & Bad Command Detection 16
5.3.2 MPSSE Setup 18

5.4 Serial Communications 19
5.5 GPIO Access..... 22
5.6 Close handle 23
6 Summary..... 24
7 Contact Information..... 25
Appendix A – References 26
Document References 26
Acronyms and Abbreviations..... 26
Appendix B – List of Tables & Figures 27
Appendix C – Revision History 28

1 Introduction

FTDI's Multi-Protocol Synchronous Serial Engine (MPSSE) provides a flexible means of interfacing synchronous serial devices to a USB port. By being "Multi-Protocol", the MPSSE allows communication with many different types of synchronous devices, the most popular being SPI, I^2C and JTAG. Data formatting and clock synchronization can be configured in a variety of ways to satisfy almost any requirement. In addition to the serial data pins, additional GPIO signals are available. This document outlines the basics in configuring the MPSSE for use and demonstrates some of the available modes of operation.

1.1 Required Components

Use of the MPSSE requires certain components be in place, both software and hardware:

1) FTDI FT-series device with the MPSSE – At the time of publication, FTDI manufactures three devices with the MPSSE block:
    a. FT2232D – USB 2.0 Full-Speed Dual UART/FIFO with a single MPSSE (6Mbps, maximum)
    b. FT2232H – USB 2.0 Hi-Speed Dual UART/FIFO with two MPSSEs (30Mbps each, maximum)
    c. FT4232H – USB 2.0 Hi-Speed Quad UART with two MPSSEs (30Mbps each, maximum)
2) FTDI D2XX Device Drivers
    a. The latest D2XX device drivers are required. Multiple operating systems are supported. See http://ftdichip.com/Drivers/D2XX.htm for the latest downloads. Installation guides for various operating systems are available at the FTDI Website.
3) Documentation
    a. Datasheet for the FTDI FT-series device with the MPSSE
    b. D2XX Programmers Guide
    c. AN_108 Command Processor for MPSSE and MCU Host Bus Emulation Modes

Although there are programming examples and libraries on the FTDI web site specific to SPI, I^2C and JTAG, it is often easier to access the MPSSE directly with the D2XX calls. This direct access will be covered through the examples presented here.

The code examples contained within this document are for demonstration purposes only and FTDI extend no responsibility or guarantees regarding the correctness of this code.

1.2 Master/Slave Association

The MPSSE is always a master controller for the selected synchronous interface. As such, it generates the clock and any required interface select / chip-select signals. The MPSSE does not operate as a slave.

2 Hardware Connections

There are four defined pins for each MPSSE channel coupled with a selection of GPIO pins. Table 2.1 below indicates the various pin assignments.

Signal Name	FT2232H Channel A	FT2232H Channel B	FT4232H Channel A	FT4232H Channel B	FT2232D Channel A
TDI/DO	17	39	17	27	23
TDO/DI	18	40	18	28	22
TCK/SK	16	38	16	26	24
TMS/CS	19	41	19	29	21
GPIOL0	21	43	21	30	20
GPIOL1	22	44	22	32	19
GPIOL2	23	45	23	33	17
GPIOL3	24	46	24	34	16
GPIOH0	26	48			15
GPIOH1	27	52			13
GPIOH2	28	53			12
GPIOH3	29	54			11
GPIOH4	30	55			
GPIOH5	32	57			
GPIOH6	33	58			
GPIOH7	34	59			

Table 2.1 MPSSE Pin Assignments

The MPSSE can be configured to handle nearly any synchronous interface. Table 2.2 below indicates the signal assignments for the more popular interfaces of SPI, I^2C and JTAG, as well as GPIO signals that have a pre-defined function.

MPSSE Signal	SPI Assignment	I^2C Assignment	JTAG Assignment
Data Out (TDI/DO)	MOSI	SDA(1)	TDI(2)
Data In (TDO/DI)	MISO	SDA(1)	TDO(2)
Clock (TCK/CK)	SCLK	SCK	TCK
Chip Select (TMS/CS)	CS	Unused	TMS
GPIOL0	<available>	<available>	<available>
GPIOL1	WAIT(4) or STOPCLK(3)(4)	WAIT(4) or STOPCLK(3)(4)	WAIT(4) or STOPCLK(3)(4)
GPIOL2	<available>	<available>	<available>
GPIOL3	<available>	<available>	RTCK(3)(4)
GPIOH0	<available>	<available>	<available>
GPIOH1	<available>	<available>	<available>
GPIOH2	<available>	<available>	<available>
GPIOH3	<available>	<available>	<available>
GPIOH4	<available>	<available>	<available>
GPIOH5	<available>	<available>	<available>
GPIOH6	<available>	<available>	<available>
GPIOH7	<available>	<available>	<available>

Table 2.2 Popular Synchronous Bus Signal Assignments

Notes:

1) The DI and DO pins need connected together in order to create the full SDA signal for I^2C. The DO pin requires configuration as an input except when transmitting in order to avoid driver contention during a slave transmission.
2) The signal name assignments are with respect to the JTAG chain. TDI is the input to the first device in the JTAG chain; the MPSSE DO signal will connect to TDI. TDO is the output from the last device in the JTAG chain; the MPSSE DI signal will connect to TDO.
3) FT2232H and FT4232H Only.
4) These pins are available for GPIO if the MPSSE commands that use them are *not* used.

2.1 Suggested Connections

Each type of interface has some unique circuitry needs. With such a diverse selection of peripheral devices, not every interface can be covered here. The implementations below will serve as a starting point for each of the popular interfaces.

For the FTDI USB Hi-Speed devices (FT2232H and FT4232H), the I/O interface operates at 3.3V. The pins are 5V-tolerant, so it is possible to directly connect 5V devices to the interface. For the FTDI USB Full-Speed device (FT2232D), the I/O interface operates at the voltage applied to VCCIO.

Care should be taken to review all datasheets for the devices to ensure I/O threshold and maximum voltages are met.

2.1.1 SPI – Single Slave

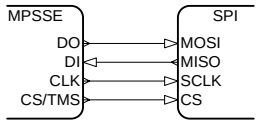

Figure 2.1 SPI - Single Slave Example Circuit

With a single SPI slave device, it is connected with a 1:1 relationship between signals. Some SPI devices do not have both data in and data out signals. For example an analogue to digital converter may not have a MOSI data input, and a digital to analogue converter may not have a MISO data output. The signals on the FTx232D/H chips have internal pull-ups, so they may be left unconnected if they are not used.

The chip select (CS) signal is used to enable the slave device's interface. With only one device, it may be acceptable to tie the chip select to an always-active state.

2.1.2 SPI – Multiple Slaves

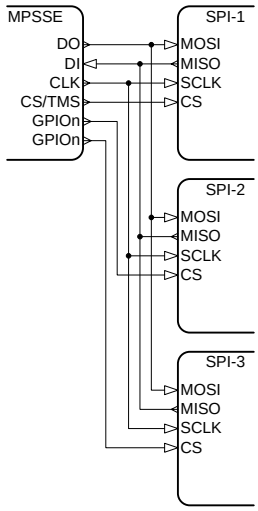

Figure 2.2 SPI - Multiple Slaves Example Circuit

Multiple SPI slaves share data in, data out and clock signals; however, they each require a unique chip select (CS) signal. Any of the available GPIO signals, in addition to the MPSSE CS signal can be used as additional chip selects. Only one slave device can be active at a time. The application program must keep track of which SPI slave device is enabled.

As with the single-slave connection, unused DO and DI signals can be left unconnected.

2.1.3 I²C – Single Slave

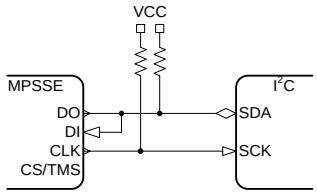

Figure 2.3 I²C Single Slave Example Circuit

I²C is a bidirectional, half-duplex communication scheme. Although the full specification allows for multiple-masters, the MPSSE can only interface with I²C slave devices.

The I²C interface can be implemented with the connection shown in Figure 2.3. In addition, the application software will need to include steps to change the direction of the MPSSE DO signal in order to eliminate any bus contention.

Other connection schemes are possible, but are outside the scope of this application note.

2.1.4 I²C – Multiple Slaves

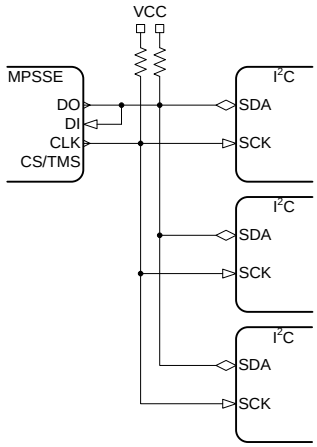

Figure 2.4 I²C Multiple Slaves Example Circuit

As an extension of the single-slave connection above, multiple I²C slave devices can be connected in parallel. As before, only I²C slave devices can be used and the MPSSE DO signal will require changes in direction to eliminate bus contention.

2.1.5 JTAG

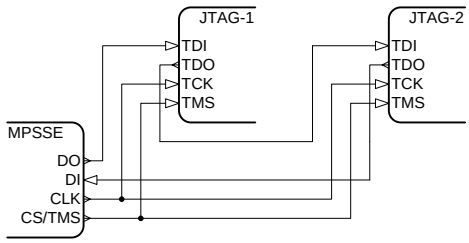

Figure 2.5 JTAG - Multiple TAP Example Circuit

JTAG implementations are well defined in the IEEE 1149.1 specification. DO is connected to the first JTAG TAP TDI signal. The first JTAG TDO is connected to the second JTAG TDI. This continues for all devices in the chain. The last TDO will be connected back to the MPSSE DI input. TCK and TMS are connected in parallel for all JTAG TAP devices.

There are other connection schemes commonly used to break up large JTAG chains. Those are outside the scope of this application note.

3 Serial Protocol Configuration

All MPSSE commands as well as general communications with the FTDI MPSSE is accomplished over the USB bus from the host computer to the FT-series part. Use of the D2XX API is required, and all MPSSE commands are formulated through its read and write calls. See Section 1.1 Required Components for links to the device driver and documentation.

Now that the hardware connections have been established, one must determine how to configure the MPSSE to actually communicate with the target device(s). The following sections are a discussion of the various parameters that need to be set.

3.1 MSB/LSB & Data Length

The MPSSE commands configure communications to send and receive data with either the least significant bit or most significant bit first. Consult the datasheet of the target device to determine which is needed. It is important to recognize that the choice of MPSSE command determines in which direction the data will be transmitted.

The MPSSE commands operate on bytes, regardless of the orientation of the target device. For example, if a 93C46D EEPROM configured for 16-bit operation, byte 0 is transmitted first from the MPSSE queue and will be the most significant byte of the 16-bit word. Byte 1 is transmitted next, becoming the least significant byte.

Both byte- and bit-based MPSSE commands exist to accommodate devices that may not operate on 8-bit increments.

3.2 Clocks

As with data transmission direction, it is important to determine not only the speed that the target device will communicate, but also on which edges the FTx232H/D device will transmit and receive data.

3.2.1 Divisors

The FT2232D is based around a 12MHz clock. A 16-bit divisor is used to program the data transmission speed according the following formula:

$$Data\ Speed = \frac{12MHz}{\left(\left(1 + Divisor\right) * 2\right)}$$

Equation 3.1 FT2232D MPSSE Data Rate

The Divisor is a 16-bit hex value between 0x0000 and 0xFFFF, yielding possible data rates between 6MHz and 92Hz.

The FT2232H and FT4232H are based around a 60MHz clock, 5x faster than the FT2232D. The FTx232H data transmission formula is:

$$Data\ Speed = \frac{60MHz}{\left(\left(1 + Divisor\right) * 2\right)}$$

Equation 3.2 FTx232H MPSSE Data Rate

As with the FT2232D, the FT2232H Divisor is a 16-bit hex value between 0x0000 and 0xFFFF. With the faster base clock, data rates range between 30MHz and ~460Hz.

The FTx232H devices also have a divide by 5 option. It is enabled by default to maintain compatibility with FT2232D. With the divide by 5 option enabled, the FT2232D divisor formula is used.

3.2.2 Edges

Data is typically clocked in and out on clock edges. Either the rising or falling edge can be used on transmit or receive. This allows for six possibilities as outlined in Table 3.1.

Clock edge for transmit	Clock edge for receive	Idle Clock State
Rising	Rising	Not Valid
Rising	Falling	Low
Falling	Rising	High
Falling	Falling	Not Valid
Rising	No data transfer	Low
Falling	No data transfer	High
No data transfer	Rising	Low
No data transfer	Falling	High

Table 3.1 Data Transfer on Clock Edges

Note that both bidirectional and unidirectional data transfer is possible.

Based on these available options, refer to the datasheet of the target device to determine which edge is required for each direction. In the 93C46D example noted above, the EEPROM clocks data in and out on the rising edge. In this case, the MPSSE should be configured for data transfer on falling edges for both transmit and receive. This allows the data out from both the MPSSE and the target device to stabilize before being clocked in on the next edge.

3.2.3 Special Cases

3.2.3.1 Clocks without data transfer

MPSSE commands can be issued to generate the clock signal without any data transfer. Options are available to halt the clock generation through the use of a GPIO signal, or to simply generate a given number of clock cycles.

3.2.3.2 3-phase clocking (FT2232H & FT4232H only)

For I^2C, data is available to the target on both the rising and falling edge of a clock signal. Data transitions should only occur while the clock line is low.

3.2.3.3 JTAG Details

In addition to data in/out and clock, JTAG requires a fourth signal, TMS, to navigate through the IEEE 1149.1 state machine. Within the MPSSE, TMS is treated as a sort of secondary data output. The command chain sent to the MPSSE consists of both data input/output and TMS output instructions.

All JTAG communications are LSB first. Data is shifted into the Test Access Port (TAP) on the rising edge, so the MPSSE must clock data out on the prior falling edge. Data is shifted out of the TAP on the falling edge, so the MPSSE must clock data in on the rising edge. The initial clock should idle high.

Adaptive Clocking is a means of synchronizing the generation of TCK for some target MCUs. RTCK (return clock) is an input to the MPSSE that adjusts the TCK output to match the internal clock of the JTAG TAP device, usually ARM-based microcontrollers.

See AN_129_FTDI_Hi_Speed_USB_To_JTAG_Example for additional JTAG details.

3.3 Miscellaneous Details

3.3.1 Initial Pin States

It is important to note that clock generation depends on the initial state of the SK output. Clock generation is performed according to Table 3.2 below.

Initial State of SK/TCK	Generated Clock Pulse
Low	Low – High – Low
High	High – Low – High

Table 3.2 Clock Pulse Generation

Prior to sending or receiving any data, the four dedicated MPSSE pins and any GPIO signals in use must be configured for the correct direction and, if an output, the initial state. See "AN_108 Command Processor for MPSSE and MCU Host Bus Emulation Modes", Section 2.2 for additional data clocking details.

A more detailed discussion of the programming steps required to configure and communicate through the MPSSE is in Section 4.

3.3.2 Buffer Sizes

MPSSE data and commands are mixed in a single buffer as shown in Table 3.3 below. Multiple commands can be sent to the MPSSE with a single call to FT_Write. The application's buffers must be of sufficient size to handle the largest combination of commands and data used in a single call.

Data Type	Length (bytes)
Command	1
Data length	2
Data payload	1 to 65536
Total	**4 to 65539**

Table 3.3 Application Buffer Size Allocation

4 Software Configuration

All communication with the MPSSE is accomplished through the FTDI D2XX API, which is defined in the FTDI D2XX Programmer's Guide. See Section 1.1 Required Components for links to the device driver and documentation.

The following sections outline the required steps to configure and communicate through the MPSSE.

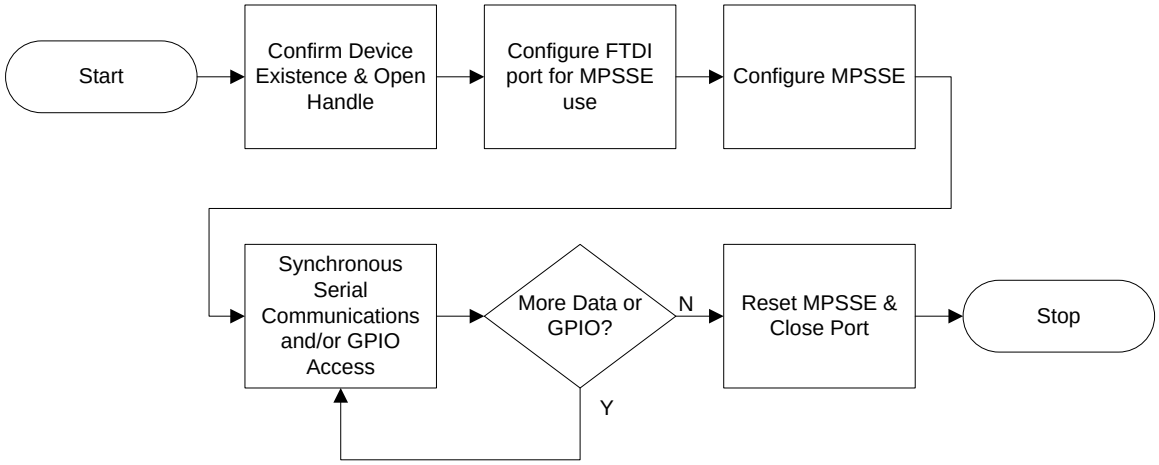

Figure 4.1 MPSSE Usage Flow Chart

4.1 Confirm device existence and open handle

Prior to utilizing the MPSSE, an application program needs to find out how many FT-series devices are connected to the host system and to select the correct one. This is done with the following D2XX API calls:

1) FT_CreateDeviceInfoList – This call returns the number of available FT-series devices that are available on a particular system. It is important to note that each port of a multi-port chip is included in this number.
2) FT_GetDeviceInfoList or FT_GetDeviceInfoListDetail – Depending on how it is used, this call returns information about each of the available devices, such as the device name, which port it is (e.g. "FT2232H A" or "FT2232H B"), the USB Location ID, the USB Serial Number, and most importantly the USB Handle.
3) FT_Open or FT_OpenEx – Once the port information is determined, the application next opens the port based on the handle obtained in step 2.

4.2 Configure FTDI Port For MPSSE Use

After opening the port, a few parameters need configured before the MPSSE can be enabled. This consists of the following steps:

1) FT_ResetDevice – Reset the peripheral side of FTDI port.
2) FT_SetUSBParameters – Configure the maximum USB transfer sizes. This value can be set from 64 bytes to 64Kbytes and is dependent on the amount of data that needs to be transmitted or received. Separate input and output transfer sizes can be set.
3) FT_SetChars –Configure the event and error characters. Most applications disable any event or error characters.
4) FT_SetTimeouts – Configures the read and write timeouts in milliseconds. These timeouts are disabled by default, so it is common to give the device driver a means of breaking out of an errant transfer.
5) FT_SetLatencyTimer – This configures the amount of time to wait before sending an incomplete USB packet from the peripheral back to the host. For applications that require quick responses from the peripheral, set the latency timer to a lower value.

6) FT_SetFlowControl – Configure for RTS/CTS flow control to ensure that the driver will not issue IN requests if the buffer is unable to accept data.
7) FT_SetBitMode – mode = 0, mask = 0 – Reset the MPSSE controller. Perform a general reset on the MPSSE, not the port itself.
8) FT_SetBitMode – mode = 2, mask = 0 – Enable the MPSSE controller. Pin directions are set later through the MPSSE commands.

4.3 Configure MPSSE

At this point, the MPSSE is ready to accept commands. MPSSE commands consist of an op-code followed by one or more parameters. Op-codes are defined in "AN_108 Command Processor for MPSSE and MCU Host Bus Emulation Modes". FT_Write is used to send the command and parameters to the MPSSE. Responses from the MPSSE are read by the application with FT_Read.

4.3.1 Synchronization & Bad Command Detection

If a bad command is detected, the MPSSE returns the value 0xFA, followed by the byte that caused the bad command.

Use of the bad command detection is the recommended method of determining whether the MPSSE is in sync with the application program. By sending a bad command on purpose and looking for 0xFA, the application can determine whether communication with the MPSSE is possible.

4.3.2 MPSSE Setup

With communications established, the MPSSE must now be configured for the clock speed, pin directions and initial pin states. The MPSSE in the FT2232H and FT4232H Hi-Speed USB parts have additional parameters that need set: Divide clock by 5, 3-phase data clocking and JTAG adaptive clocking. Although the default settings may be appropriate for a particular application, it is always a good practice to explicitly send all of the op-codes to enable or disable each of these features.

4.4 Serial Communications

Once all of the parameters are configured, communication with the peripheral may occur.

The MPSSE can be placed in loop-back mode for diagnostic purposes. In addition to data being transmitted out of the DO pin, it is also connected internally to the DI pin.

In both normal and loop-back modes, there are 32 choices of how data is transmitted received or both transmitted and received. The choice of op-code depends on the following:

- *Most significant bit first or least significant bit first.* Note that each byte is transmitted in order. If data is more than 8-bits wide, care must be taken to place the data into the buffer in the correct order.
- *Transmit data only, receive data only, or both transmit and receive data.*
- *Transmit on rising or falling edge; receive on rising or falling edge.*

Commands in a buffer used with FT_Write would be followed by FT_GetStatus and FT_Read to read back the response from the peripheral.

4.5 GPIO Access

Each of the FTDI chips with the MPSSE has several pins that can be used for general purpose input and output as outlined in Table 2.2. As with the serial communications, FT_Write is used to set the direction and output values and to prompt the MPSSE to return the actual pin states. After issuing the MPSSE "read GPIO" command, FT_Read is then used to retrieve the data containing the pin states.

4.6 Close handle

When the application has completed all communications with the peripheral, the handle to the FTDI device must be closed. Although it is not explicitly necessary, it is also a good idea to reset the MPSSE first, placing the port in an idle state before closing.

5 Example Program

The example code listed below will follow Section 4 Software Configuration. While Section 4 covers the FTDI MPSSE in general, this example focuses on a single set of parameters. This example program utilizes the FTDI D2XX device driver. It is purposely written in a linear fashion to demonstrate the actual bytes being sent to the MPSSE and the resultant data read from the MPSSE. A single call to FT_Write can actually contain a number of MPSSE commands and arguments.

The first section establishes several of the variables that will be used throughout the program.

```
int _tmain(int argc, _TCHAR* argv[])
{
        // ------------------------------------------------------------
        // Variables
        // ------------------------------------------------------------

        FT_HANDLE ftHandle;                     // Handle of the FTDI device
        FT_STATUS ftStatus;                     // Result of each D2XX call

        DWORD dwNumDevs;                        // The number of devices
        unsigned int uiDevIndex = 0xF;          // The device in the list that we'll use

        BYTE byOutputBuffer[8];                 // Buffer to hold MPSSE commands and data
                                                //     to be sent to the FT2232H
        BYTE byInputBuffer[8];                  // Buffer to hold data read from the FT2232H

        DWORD dwCount = 0;                      // General loop index
        DWORD dwNumBytesToSend = 0;             // Index to the output buffer
        DWORD dwNumBytesSent = 0;               // Count of actual bytes sent - used with FT_Write
        DWORD dwNumBytesToRead = 0;             // Number of bytes available to read
                                                //     in the driver's input buffer
        DWORD dwNumBytesRead = 0;               // Count of actual bytes read - used with FT_Read

        DWORD dwClockDivisor = 0x05DB;          // Value of clock divisor, SCL Frequency =
                                                //     60/((1+0x05DB)*2) (MHz) = 1Mhz
```

5.1 Confirm device existence and open handle

```
        // ------------------------------------------------------------
        // Does an FTDI device exist?
        // ------------------------------------------------------------

        printf("Checking for FTDI devices...\n");

        ftStatus = FT_CreateDeviceInfoList(&dwNumDevs);
                                                // Get the number of FTDI devices
        if (ftStatus != FT_OK)                  // Did the command execute OK?
        {
                printf("Error in getting the number of devices\n");
                return 1;                       // Exit with error
        }

        if (dwNumDevs < 1)                      // Exit if we don't see any
        {
                printf("There are no FTDI devices installed\n");
                return 1;                       // Exit with error
        }

        printf("%d FTDI devices found \
                - the count includes individual ports on a single chip\n", dwNumDevs);


        // ------------------------------------------------------------
        // Open the port - For this application note, we'll assume the first device is a
        //      FT2232H or FT4232H.  Further checks can be made against the device
        //      descriptions, locations, serial numbers, etc. before opening the port.
        // ------------------------------------------------------------


        printf("\nAssume first device has the MPSSE and open it...\n");
```

```
ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK)
{
        printf("Open Failed with error %d\n", ftStatus);
        return 1;                       // Exit with error
}
```

5.2 Configure FTDI Port For MPSSE Use

```
// Configure port parameters

printf("\nConfiguring port for MPSSE use...\n");

ftStatus |= FT_ResetDevice(ftHandle);
                                        //Reset USB device

//Purge USB receive buffer first by reading out all old data from FT2232H receive buffer
ftStatus |= FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in the FT2232H
                                        //      receive buffer
if ((ftStatus == FT_OK) && (dwNumBytesToRead > 0))
        FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);
                                        //Read out the data from FT2232H receive buffer

ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535);
                                        //Set USB request transfer sizes to 64K
ftStatus |= FT_SetChars(ftHandle, false, 0, false, 0);
                                        //Disable event and error characters
ftStatus |= FT_SetTimeouts(ftHandle, 0, 5000);
                                        //Sets the read and write timeouts in milliseconds
ftStatus |= FT_SetLatencyTimer(ftHandle, 1);
                                        //Set the latency timer to 1mS (default is 16mS)
ftStatus |= FT_SetFlowControl(ftHandle, FT_FLOW_RTS_CTS, 0x00, 0x00);
                                        //Turn on flow control to synchronize IN requests
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        //Reset controller
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x02);
                                        //Enable MPSSE mode
if (ftStatus != FT_OK)
{
        printf("Error in initializing the MPSSE %d\n", ftStatus);
        FT_Close(ftHandle);
        return 1;                       // Exit with error
}

Sleep(50); // Wait for all the USB stuff to complete and work
```

The D2XX return value of ftStatus can be checked individually, or as an aggregate result of a number of API calls as shown above. Throughout the rest of this example program, ftStatus is not always checked in order maintain focus on each task being discussed.

5.3 Configure the FTDI MPSSE

At this point, the MPSSE is ready for commands. Each command consists of an op-code followed by any necessary parameters or data. For clarity, each command is sent to the MPSSE with an individual FT_Write call. In practice, it may be desirable to combine several commands into a single FT_Write call.

5.3.1 Synchronization & Bad Command Detection

```
// Enable internal loop-back

byOutputBuffer[dwNumBytesToSend++] = 0x84;
                                        // Enable loopback
ftStatus = FT_Write(ftHandle, byOutputBuffer, \
        dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the loopback command
dwNumBytesToSend = 0;                   // Reset output buffer pointer

// Check the receive buffer - it should be empty
ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
```

```
                                        // Get the number of bytes in
                                        // the FT2232H receive buffer
if (dwNumBytesToRead != 0)
{
        printf("Error - MPSSE receive buffer should be empty\n", ftStatus);
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
        FT_Close(ftHandle);             // Close the USB port
        return 1;                       // Exit with error
}

// ------------------------------------------------------------
// Synchronize the MPSSE by sending a bogus opcode (0xAB),
//      The MPSSE will respond with "Bad Command" (0xFA) followed by
//      the bogus opcode itself.
// ------------------------------------------------------------

byOutputBuffer[dwNumBytesToSend++] = 0xAB;
                                        //Add bogus command '0xAB' to the queue
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the BAD command
dwNumBytesToSend = 0;                   // Reset output buffer pointer


do
{
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in the device input buffer
} while ((dwNumBytesToRead == 0) && (ftStatus == FT_OK));
                                        //or Timeout

bool bCommandEchod = false;

ftStatus = FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);
                                        //Read out the data from input buffer

for (dwCount = 0; dwCount < dwNumBytesRead - 1; dwCount++)
                                        //Check if Bad command and echo command are received
{
        if ((byInputBuffer[dwCount] == 0xFA) && (byInputBuffer[dwCount+1] == 0xAB))
        {
                bCommandEchod = true;
                break;
        }
}
if (bCommandEchod == false)
{
        printf("Error in synchronizing the MPSSE\n");
        FT_Close(ftHandle);
        return 1;                       // Exit with error
}

// Disable internal loop-back

byOutputBuffer[dwNumBytesToSend++] = 0x85;
                                        // Disable loopback
ftStatus = FT_Write(ftHandle, byOutputBuffer,\
        dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the loopback command
dwNumBytesToSend = 0;                   // Reset output buffer pointer

// Check the receive buffer - it should be empty
ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in
                                        // the FT2232H receive buffer

if (dwNumBytesToRead != 0)
{
        printf("Error - MPSSE receive buffer should be empty\n", ftStatus);
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
        FT_Close(ftHandle);             // Close the USB port
        return 1;                       // Exit with error
}
```


5.3.2 MPSSE Setup

```
// ---------------------------------------------------------
// Configure the MPSSE settings for JTAG
//      Multple commands can be sent to the MPSSE with one FT_Write
// ---------------------------------------------------------

dwNumBytesToSend = 0;                // Start with a fresh index

// Set up the Hi-Speed specific commands for the FTx232H

byOutputBuffer[dwNumBytesToSend++] = 0x8A;
                                     // Use 60MHz master clock (disable divide by 5)
byOutputBuffer[dwNumBytesToSend++] = 0x97;
                                     // Turn off adaptive clocking (may be needed for ARM)
byOutputBuffer[dwNumBytesToSend++] = 0x8D;
                                     // Disable three-phase clocking
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                     // Send off the HS-specific commands
dwNumBytesToSend = 0;                // Reset output buffer pointer

// Set TCK frequency
// TCK = 60MHz /((1 + [(1 +0xValueH*256) OR 0xValueL])*2)

byOutputBuffer[dwNumBytesToSend++] = '\x86';
                                     // Command to set clock divisor
byOutputBuffer[dwNumBytesToSend++] = dwClockDivisor & 0xFF;
                                     // Set 0xValueL of clock divisor
byOutputBuffer[dwNumBytesToSend++] = (dwClockDivisor >> 8) & 0xFF;
                                     // Set 0xValueH of clock divisor
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                     // Send off the clock divisor commands
dwNumBytesToSend = 0;                // Reset output buffer pointer

// Set initial states of the MPSSE interface
//      - low byte, both pin directions and output values
//      Pin name        Signal  Direction       Config  Initial State   Config
//      ADBUS0          TCK/SK  output          1       high            1
//      ADBUS1          TDI/DO  output          1       low             0
//      ADBUS2          TDO/DI  input           0                       0
//      ADBUS3          TMS/CS  output          1       high            1
//      ADBUS4          GPIOL0  output          1       low             0
//      ADBUS5          GPIOL1  output          1       low             0
//      ADBUS6          GPIOL2  output          1       high            1
//      ADBUS7          GPIOL3  output          1       high            1

byOutputBuffer[dwNumBytesToSend++] = 0x80;
                                     // Configure data bits low-byte of MPSSE port
byOutputBuffer[dwNumBytesToSend++] = 0xC9;
                                     // Initial state config above
byOutputBuffer[dwNumBytesToSend++] = 0xFB;
                                     // Direction config above
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                     // Send off the low GPIO config commands
dwNumBytesToSend = 0;                // Reset output buffer pointer


// Note that since the data in subsequent sections will be clocked on the rising edge, the
//      inital clock state of high is selected.  Clocks will be generated as high-low-high.

//      For example, in this case, data changes on the rising edge to give it enough time
//      to have it available at the device, which will accept data *into* the target device
//      on the falling edge.

// Set initial states of the MPSSE interface
//      - high byte, both pin directions and output values
//      Pin name        Signal  Direction       Config  Initial State   Config
//      ACBUS0          GPIOH0  input           0                       0
//      ACBUS1          GPIOH1  input           0                       0
//      ACBUS2          GPIOH2  input           0                       0
//      ACBUS3          GPIOH3  input           0                       0
//      ACBUS4          GPIOH4  input           0                       0
//      ACBUS5          GPIOH5  input           0                       0
//      ACBUS6          GPIOH6  input           0                       0
//      ACBUS7          GPIOH7  input           0                       0

byOutputBuffer[dwNumBytesToSend++] = 0x82;
```

```
                                        // Configure data bits low-byte of MPSSE port
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                                        // Initial state config above
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                                        // Direction config above
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the high GPIO config commands
dwNumBytesToSend = 0;                   // Reset output buffer pointer


for(dwCount = 0; dwCount < 8; dwCount++)
{                                       // Clear out the input and output buffers
        byInputBuffer[dwCount] = 0x00;
        byOutputBuffer[dwCount] = 0x00;
}
```

5.4 Serial Communications

```
// Data Transmit, no receive

byOutputBuffer[dwNumBytesToSend++] = 0x10;
                                        // Output on rising clock, no input
                                        // MSB first, clock a number of bytes out
byOutputBuffer[dwNumBytesToSend++] = 0x01;   // Length L
byOutputBuffer[dwNumBytesToSend++] = 0x00;   // Length H
                                        // Length = 0x0001 + 1
byOutputBuffer[dwNumBytesToSend++] = 0xA5;
byOutputBuffer[dwNumBytesToSend++] = 0x0F;
                                        // Data = 0xA50F

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the command
dwNumBytesToSend = 0;                   // Reset output buffer pointer

Sleep(2);                               // Wait for data to be transmitted and status
                                        //      to be returned by the device driver
                                        //      - see latency timer above




// Check the receive buffer - it should be empty
ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in
                                        //      the FT2232H receive buffer
                                        // It should be zero since there was
                                        //      no data clocked *in*

FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);

if (dwNumBytesToRead != 0)
{
        printf("Error - MPSSE receive buffer should be empty\n", ftStatus);
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
        FT_Close(ftHandle);             // Close the USB port
        return 1;                       // Exit with error
}
```


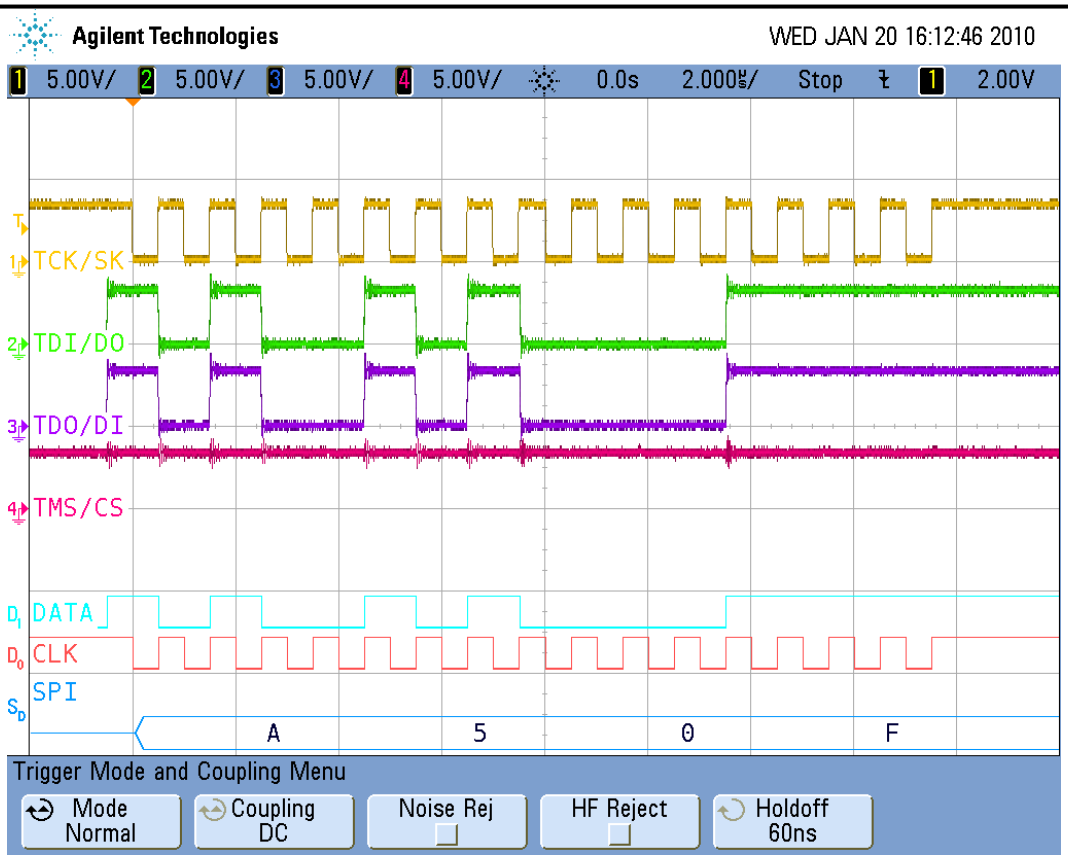

Figure 5.1 Oscilloscope Result of Data OUT only

In Figure 5.1, the four main signals of the FTDI MPSSE match the expected value. A FT2232H module was used with TDI/DO connected to TDO/DI and approproate connections to configure it for USB bus power. The DATA and CLK signals shown above are duplicates of TDI/DO and TCK/SK, respectively. These two are used as inputs to the SPI decode function within the oscilloscope.

By inspecting the TCK/SK and TDI/DO, one can observe that data is being clocked out on the rising edge of the clock signal. This makes it available to the peripheral on the next falling edge of the clock.

```
printf("Press <Enter> to continue\n");
getchar();                              // wait for a carriage return

// Now repeat the transmission with the send and receive op-code in place of transmit-only
// Data Transmit, with receive

byOutputBuffer[dwNumBytesToSend++] = 0x34;
                                        // Output on rising clock, input on falling clock
                                        // MSB first, clock a number of bytes out
byOutputBuffer[dwNumBytesToSend++] = 0x01;   // Length L
byOutputBuffer[dwNumBytesToSend++] = 0x00;   // Length H
                                        // Length = 0x0001 + 1
byOutputBuffer[dwNumBytesToSend++] = 0xA5;
byOutputBuffer[dwNumBytesToSend++] = 0x0F;
                                        // Data = 0xA50F

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                        // Send off the command
dwNumBytesToSend = 0;                   // Reset output buffer pointer

Sleep(2);                               // Wait for data to be transmitted and status
                                        //      to be returned by the device driver
                                        //      - see latency timer above


// Check the receive buffer - it should contain the looped-back data

ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in
                                        //      the FT2232H receive buffer
```

```
                                        // It should be zero since there was
                                        //      no data clocked *in*

FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);


// The input buffer should contain the same number of bytes as those output

if (dwNumBytesToRead != 2)
{
        printf("Error - MPSSE receive buffer should have the looped-back data\n");
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
        FT_Close(ftHandle);             // Close the USB port
        return 1;                       // Exit with error
}
printf("The correct number of bytes have been received\n");


// Check to be sure it's the same.

for(dwCount = 0; dwCount <= dwNumBytesRead - 1; dwCount++)
{
        if (byInputBuffer[dwCount] != byOutputBuffer[dwCount + 3])
                                        // Output data begins at location 3,
                                        //      after the opcode and length
        {
                printf("Error - Data received does not match data output\n");
                FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
                FT_Close(ftHandle);     // Close the USB port
                return 1;               // Exit with error
        }
}

printf("The input data matches the output data\n");
```

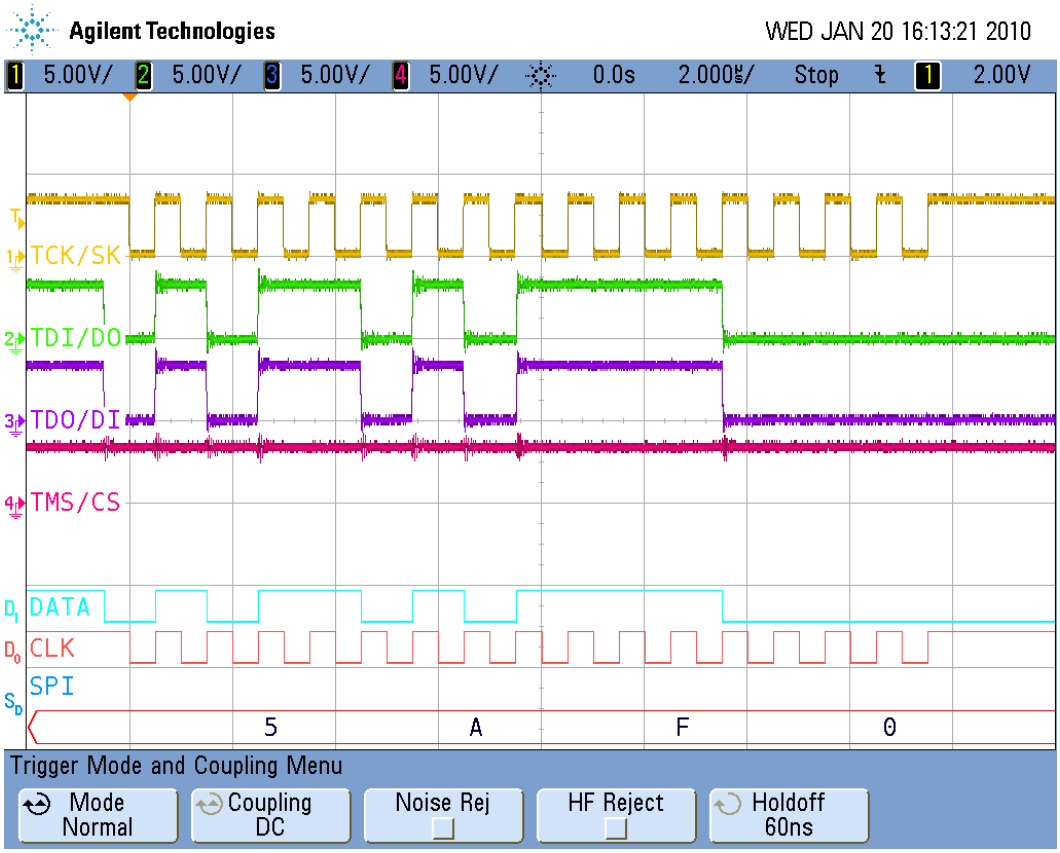

Figure 5.2 Oscilloscope Result of Data OUT and IN

Figure 5.2 uses the same oscilloscope setup as in Figure 5.1. With TDI/DO connected to TDO/DI, data is clocked into the MPSSE on the falling edge of clock. Again, one can confirm the data being transmitted, and that it matches when read from the MPSSE receive buffer.

```
printf("Press <Enter> to continue\n");
getchar();                              // wait for a carriage return


// Clear the buffers
for(dwCount = 0; dwCount < 8; dwCount++)
{
        byInputBuffer[dwCount] = 0x00;
        byOutputBuffer[dwCount] = 0x00;
}
```

5.5 GPIO Access

The non-dedicated pins of the FTDI MPSSE port can be used as GPIO. This includes any chip select (CS) signals needed for SPI operation. Setting and clearing any of these pins is done in the same fashion. If sent individually, as demonstrated below, the speed at which the GPIO signals can be changed is limited to the USB bus sending each FT_Write call in a separate transaction. This is one of the cases where it is desirable to chain multiple MPSSE commands into a single FT_Write call. The CS can be set, data transferred and CS cleared with one call.

The code below performs a simple read-modify-write sequence to ensure the states of the other bits are kept intact.

```
// Read From GPIO low byte
// ***********************************************
//
// Change scope trigger to channel 4 (TMS/CS) falling edge
//
// ***********************************************

byOutputBuffer[dwNumBytesToSend++] = 0x81;
                                        // Get data bits - returns state of pins,
                                        //      either input or output
                                        //      on low byte of MPSSE

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                        // Read the low GPIO byte

dwNumBytesToSend = 0;                   // Reset output buffer pointer

Sleep(2);                               // Wait for data to be transmitted and status
                                        //      to be returned by the device driver
                                        //      - see latency timer above

// Check the receive buffer - there should be one byte

ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                                        // Get the number of bytes in the
                                        //      FT2232H receive buffer

ftStatus |= FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);

if ((ftStatus != FT_OK) & (dwNumBytesToRead != 1))
{
        printf("Error - GPIO cannot be read\n");
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                        // Reset the port to disable MPSSE
        FT_Close(ftHandle);             // Close the USB port
        return 1;                       // Exit with error
}
printf("The GPIO low-byte = 0x%X\n", byInputBuffer[0]);
                                        // The inpute buffer only contains one
                                        //      valid byte at location 0

printf("Press <Enter> to continue\n");
getchar();                              // wait for a carriage return

// Modify the GPIO data (TMS/CS only) and write it back
```

```
        byOutputBuffer[dwNumBytesToSend++] = 0x80;
                                            // Set data bits low-byte of MPSSE port
        byOutputBuffer[dwNumBytesToSend++] = byInputBuffer[0] & 0xF7;
                                            // Only change TMS/CS bit to zero
        byOutputBuffer[dwNumBytesToSend++] = 0xFB;
                                            // Direction config is still needed for each GPIO write
        ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                                            // Send off the low GPIO config commands
        dwNumBytesToSend = 0;               // Reset output buffer pointer

        Sleep(2);                           // Wait for data to be transmitted and status
                                            //      to be returned by the device driver
                                            //      - see latency timer above
```

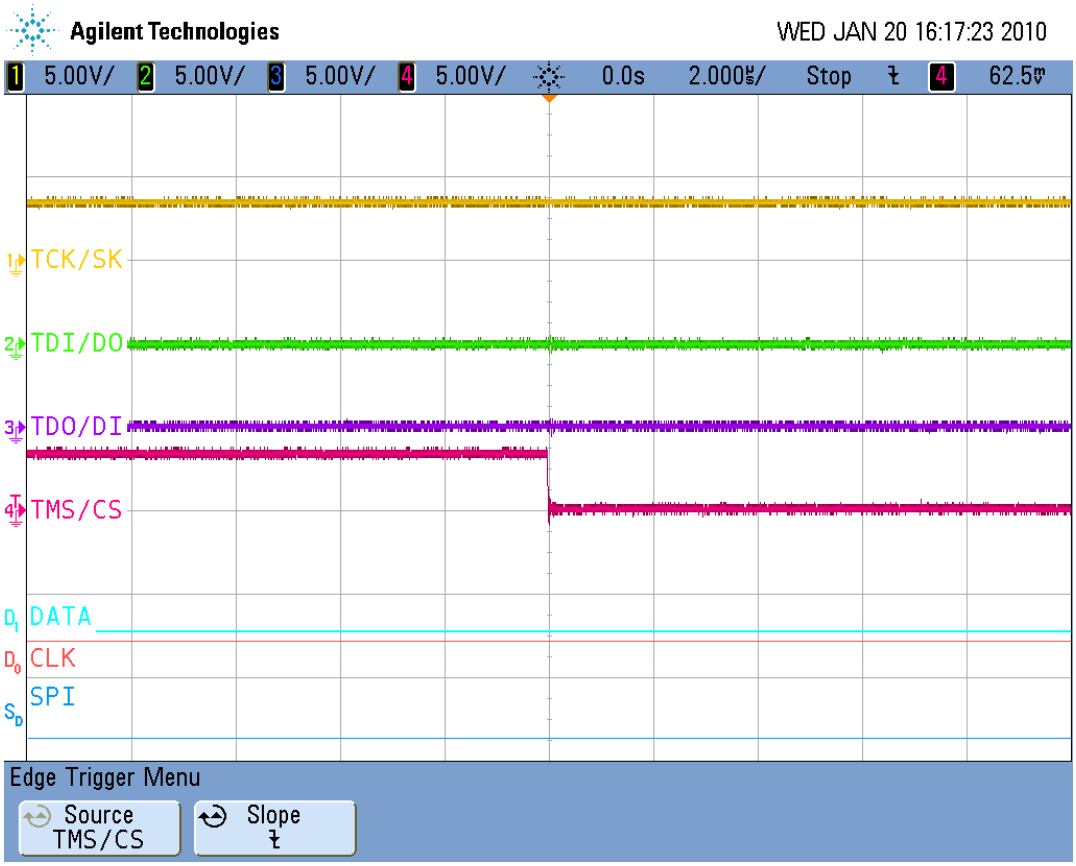

Figure 5.3 Oscilloscope Result of GPIO Output Change

Figure 5.3 shows the effect of reading the low GPIO byte, setting bit 3 (TMS/CS) to zero and writing that value back to the GPIO port.

5.6 Close handle

Once all functions are completed, the FTDI MPSSE should be reset and disabled. This is followed by closing the handle to the FT2232H port, freeing it for use by another application.

```
        // ------------------------------------------------------------
        // Start closing everything down
        // ------------------------------------------------------------
        printf("\nAN_135 example program executed successfully.\n");
        printf("Press <Enter> to continue\n");
        getchar();                          // wait for a carriage return
        FT_SetBitMode(ftHandle, 0x0, 0x00);
                                            // Reset MPSSE
        FT_Close(ftHandle);                 // Close the port

        return 0;                           // Exit with success
}
```

6 Summary

This application note demonstrates the basics of communicating with the FTDI devices with the Multi-Protocol Synchronous Serial Engine (MPSSE). By getting a few fundamental practices configured, one can utilize the flexibility of the MPSSE through implementing many synchronous serial protocols, including SPI, I^2C and JTAG. See Appendix A – References for additional application notes that investigate the specifics of each of these protocols.

7 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
Web Site URL http://www.ftdichip.com
Web Shop URL http://www.ftdichip.com

Branch Office – Hillsboro, Oregon, USA

Future Technology Devices International Limited
(USA)
7235 NW Evergreen Parkway, Suite 600
Hillsboro, OR 97123-5803
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

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

Branch Office – Taipei, Taiwan

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

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

Branch Office – Shanghai, China

Future Technology Devices International Limited
(China)
Room 408, 317 Xianxia Road,
Shanghai, 200051
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 URL http://www.ftdichip.com

Distributor and Sales Representatives

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

Appendix A – References

Document References

FT2232H Datasheet

FT4232H Datasheet

FT2232D Datasheet

FTDI D2XX Programmer's Guide

AN_108 Command Processor For MPSSE and MCU Host Bus Emulation Modes

AN_113 Interfacing FT2232H Hi-Speed Devices To I^2C Bus

AN_114 Interfacing FT2232H Hi-Speed Devices To SPI Bus

AN_129 Interfacing FT2232H Hi-Speed Devices to a JTAG TAP

TN_109 Instructions On Including The FTD2xx DLL In A VS2008 Project

Acronyms and Abbreviations

Terms	Description
API	Application Programming Interface
D2XX	FTDI "Direct Driver" – USB device driver
GPIO	General Purpose Input/Output
I^2C	Inter-Integrated Circuit – synchronous serial bus
IEEE	Institute of Electrical and Electronics Engineers
IEEE 1149.1	Synchronous serial bus for integrated circuit and printed circuit test purposes
JTAG	Joint Test Action Group
MPSSE	Multi-Protocol Synchronous Serial Engine
SPI	Serial Peripheral Interface – synchronous serial bus
TAP	Test Access Point – a device within an IEEE 1149.1 bus
USB	Universal Serial Bus

Table A.1 Acronyms and Abbreviations

Appendix B – List of Tables & Figures

List of Tables

Table 2.1 MPSSE Pin Assignments 5

Table 2.2 Popular Synchronous Bus Signal Assignments 6

Table 3.1 Data Transfer on Clock Edges 10

Table 3.2 Clock Pulse Generation 11

Table 3.3 Application Buffer Size Allocation 11

Table A.1 Acronyms and Abbreviations 26

List of Figures

Figure 2.1 SPI - Single Slave Example Circuit 7

Figure 2.2 SPI - Multiple Slaves Example Circuit 7

Figure 2.3 I^2C Single Slave Example Circuit 8

Figure 2.4 I^2C Multiple Slaves Example Circuit 8

Figure 2.5 JTAG - Multiple TAP Example Circuit 8

Figure 4.1 MPSSE Usage Flow Chart 12

Figure 5.1 Oscilloscope Result of Data OUT only 20

Figure 5.2 Oscilloscope Result of Data OUT and IN 21

Figure 5.3 Oscilloscope Result of GPIO Output Change 23

Appendix C – Revision History

Revision	Changes	Date
1.0	Initial Release	2010-02-12
1.1	Added instructions to enable RTS/CTS flow control – pp 13,16	2010-03-11