Application Note

AN_178

User Guide For libMPSSE - SPI

Document Reference No.: FT_00492

Version 1.1

Issue Date: 2012-02-13

This application note is a guide to using the libMPSSE-SPI – a library which simplifies the design of firmware for interfacing to the FTDI MPSSE configured as an SPI interface. The library is available for Windows and for Linux.

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

Table of Contents

1 Introduction

The Multi Protocol Synchronous Serial Engine (MPSSE) is a generic hardware found in several FTDI chips that allows these chips to communicate with a synchronous serial device such an I2C device, a SPI device or a JTAG device. The MPSSE is currently available on the FT2232D, FT2232H, FT4232H and FT232H chips, which communicate with a PC (or an application processor) over the USB interface. Applications on a PC or on an embedded system communicate with the MPSSE in these chips using the D2XX USB drivers.

The MPSSE takes different commands to send out data from the chips in the different formats, namely I2C, SPI and JTAG. libMPSSE is a library that provides a user friendly API that enables users to write applications to communicate with the I2C/SPI/JTAG devices without needing to understand the MPSSE and its commands. However, if the user wishes then he/she may try to understand the working of the MPSSE and use it from their applications directly by calling D2XX functions.

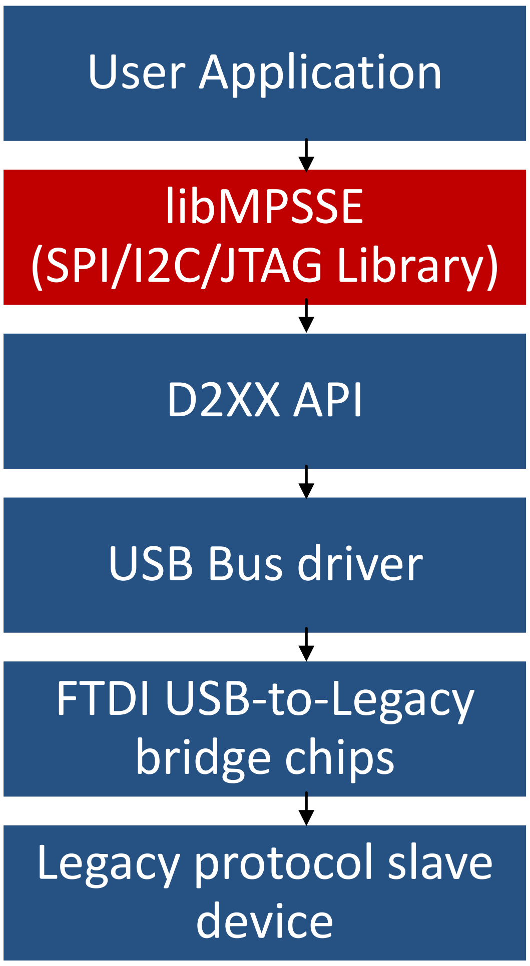


Figure 1: The software and hardware stack through which legacy protocol data flows

As shown in the above figure, libMPSSE has three different APIs, one each for I2C, SPI and JTAG. This document will only describe the SPI section.

The libMPSSE (Linux and Windows versions) sample code, release notes and all necessary files can be downloaded from the FTDI website at :

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

The sample source code contained in this application note is provided as an example and is neither guaranteed nor supported by FTDI.

2 System Overview

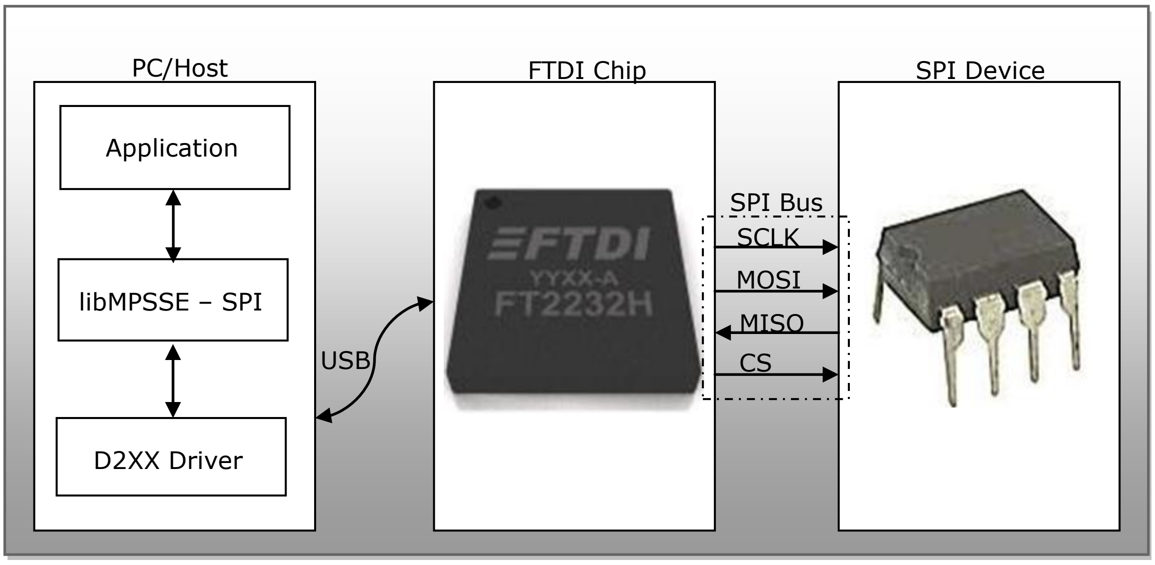


Figure 2: System organization

The above figure shows how the components of the system will typically be organized. The PC/Host may be desktop/laptop machine or an embedded system. The FTDI chip and the SPI device would usually be on the same PCB. Though only one SPI device is shown in the figure above, up to five SPI devices can actually be connected to each MPSSE.

3 Application Programming Interface (API)

The libMPSSE-SPI APIs can be divided into two broad sets. The first set consists of six control APIs and the second set consists of two data transferring APIs. All the APIs return an FT_STATUS. This is the same FT_STATUS that is defined in the D2XX driver.

3.1 SPI Functions

3.1.1 SPI_GetNumChannels

FT_STATUS **SPI_GetNumChannels** (uint32 **numChannels*)

This function gets the number of SPI channels that are connected to the host system. The number of ports available in each of these chips is different.

Parameters:

out	**numChannels*	The number of channels connected to the host

Returns:
Returns status code of type FT_STATUS

Note:
FTDI's USB-to-legacy bridge chips may have multiple channels in it but not all these channels can be configured to work as SPI masters. This function returns the total number of channels connected to the host system that has a MPSSE attached to it so that it may be configured as an SPI master.

For example, if an FT2232D (1 MPSSE port), an FT232H (1 MPSSE port), an FT2232H (2 MPSSE ports) and an FT4232H (2 MPSSE ports) are connected to a PC, then a call to SPI_GetNumChannels would return 6 in numChannels.

Warning:

This function should not be called from two applications or from two threads at the same time.

3.1.2 SPI_GetChannelInfo

FT_STATUS **SPI_GetChannelInfo** (uint32 *index*, FT_DEVICE_LIST_INFO_NODE **chanInfo*)

This function takes a channel index (valid values are from 0 to the value returned by SPI_GetNumChannels - 1) and provides information about the channel in the form of a populated FT_DEVICE_LIST_INFO_NODE structure.

Parameters:

in	*index*	Index of the channel
out	**chanInfo*	Pointer to FT_DEVICE_LIST_INFO_NODE structure

Returns:
Returns status code of type FT_STATUS

Note:

This API could be called only after calling SPI_GetNumChannels.

See also:
Structure definition of FT_DEVICE_LIST_INFO_NODE is in the D2XX Programmer's Guide.

Warning:

This function should not be called from two applications or from two threads at the same time.

3.1.3 SPI_OpenChannel

FT_STATUS **SPI_OpenChannel** (uint32 *index*, FT_HANDLE **handle*)

This function opens the indexed channel and provides a handle to it. Valid values for the index of channel can be from 0 to the value obtained using SPI_GetNumChannels - 1).

Parameters:

in	*index*	Index of the channel
out	*handle*	Pointer to the handle of type FT_HANDLE

Returns:
Returns status code of type FT_STATUS

Note:
Trying to open an already open channel will return an error code.

3.1.4 SPI_InitChannel

FT_STATUS **SPI_InitChannel** (FT_HANDLE *handle*, ChannelConfig **config*)

This function initializes the channel and the communication parameters associated with it.

Parameters:

in	*handle*	Handle of the channel
in	*config*	Pointer to ChannelConfig structure with the value of clock and latency timer updated

Returns:
Returns status code of type FT_STATUS

See also:
Structure definition of ChannelConfig

Note:

This function internally performs what is required to get the channel operational such as resetting and enabling the MPSSE.

3.1.5 SPI_CloseChannel

FT_STATUS **SPI_CloseChannel** (FT_HANDLE *handle*)

Closes a channel and frees all resources that were used by it

Parameters:

in	*handle*	Handle of the channel
out	*none*	

Returns:
Returns status code of type FT_STATUS

3.1.6 SPI_Read

FT_STATUS **SPI_Read**(FT_HANDLE *handle*, uint8 **buffer*, uint32 *sizeToTransfer*, uint32 **sizeTransferred*, uint32 *transferOptions*)

This function reads the specified number of bits or bytes (depending on *transferOptions* parameter) from an SPI slave.

Parameters:

in	*handle*	Handle of the channel
out	*buffer*	Pointer to the buffer where data is to be read
in	*sizeToTransfer*	Number of bytes or bits to be read
out	**sizeTransferred*	Pointer to variable containing the number of bytes or bits read
in	*transferOptions*	This parameter specifies data transfer options. The bit positions defined for each of these options are: BIT0: if set then *sizeToTransfer* is in bits, otherwise bytes. Bit masks defined for this bit are SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES and SPI_TRANSFER_OPTIONS_SIZE_IN_BITS BIT1: if set then the chip select line is asserted before beginning the transfer. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE BIT2: if set then the chip select line is disserted after the transfer ends. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE BIT3 – BIT31: reserved

Returns:
Returns status code of type FT_STATUS

Warning:

This is a blocking function and will not return until either the specified amounts of data are read or an error is encountered.

3.1.7 SPI_Write

FT_STATUS SPI_ Write(FT_HANDLE *handle*, uint8 **buffer*, uint32 *sizeToTransfer*, uint32 **sizeTransferred*, uint32 *transferOptions*)

This function writes the specified number of bits or bytes (depending on *transferOptions* parameter) to a SPI slave.

Parameters:

in	*handle*	Handle of the channel
out	*buffer*	Pointer to the buffer from where data is to be written
in	*sizeToTransfer*	Number of bytes or bits to write
out	**sizeTransferred*	Pointer to variable containing the number of bytes or bits written
in	*transferOptions*	This parameter specifies data transfer options. The bit positions defined for each of these options are: BIT0: if set then *sizeToTransfer* is in bits, otherwise bytes. Bit masks defined for this bit are SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES and SPI_TRANSFER_OPTIONS_SIZE_IN_BITS BIT1: if set then the chip select line is asserted before beginning the transfer. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE BIT2: if set then the chip select line is disserted after the transfer ends. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE BIT3 – BIT31: reserved

Returns:

Returns status code of type FT_STATUS

Warning:

This is a blocking function and will not return until either the specified amount of data is read or an error is encountered.

3.1.8 SPI_ReadWrite

FT_STATUS **SPI_ReadWrite**(FT_HANDLE *handle*, uint8 **inBuffer*, uint8 **outBuffer*, uint32 *sizeToTransfer*, uint32 **sizeTransferred*, uint32 *transferOptions*)

This function reads from and writes to the SPI slave simultaneously. Meaning that, one bit is clocked in and one bit is clocked out during every clock cycle.

Parameters:

in	*handle*	Handle of the channel
in	**inBuffer*	Pointer to buffer to which data read will be stored
out	*outBuffer*	Pointer to the buffer from where data is to be written
in	*sizeToTransfer*	Number of bytes or bits to write
out	**sizeTransferred*	Pointer to variable containing the number of bytes or bits written
in	*transferOptions*	This parameter specifies data transfer options. The bit positions defined for each of these options are: BIT0: if set then *sizeToTransfer* is in bits, otherwise bytes. Bit masks defined for this bit are SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES and SPI_TRANSFER_OPTIONS_SIZE_IN_BITS BIT1: if set then the chip select line is asserted before beginning the transfer. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE BIT2: if set then the chip select line is disserted after the transfer ends. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE BIT3 – BIT31: reserved

Returns:
Returns status code of type FT_STATUS

Warning:

This is a blocking function and will not return until either the specified amount of data is transferred or an error is encountered.

3.1.9 SPI_IsBusy

FT_STATUS **SPI_ IsBusy**(FT_HANDLE *handle*, bool **state*)

This function reads the state of the MISO line without clocking the SPI bus.

Some applications need the SPI master to poll the MISO line without clocking the bus to check if the SPI slave has completed previous operation and is ready for the next operation. This function is useful for such applications.

Parameters:

in	*handle*	Handle of the channel
out	**state*	Pointer to a variable to which the state of the MISO

		line will be read

Returns:
Returns status code of type FT_STATUS

3.1.10 SPI_ChangeCS

FT_STATUS **SPI_ChangeCS**(FT_HANDLE *handle*, uint32 *configOptions*)

This function changes the chip select line that is to be used to communicate to the SPI slave.

Parameters:

in	*handle*	Handle of the channel
in	*configOptions*	This parameter provides a way to select the chip select line and the slave's SPI mode. It is the same parameter as *ConfigChannel.configOptions* that is passed to function SPI_InitChannel and it is explained in section 3.4.1

Returns:

Returns status code of type FT_STATUS

3.2 GPIO functions

Each MPSSE channel in the FTDI chips are provided with a general purpose I/O port having 8 lines in addition to the port that is used for synchronous serial communication. For example, the FT232H has only one MPSSE channel with two 8-bit ports, ADBUS and ACBUS. Out of these, ADBUS is used for synchronous serial communications (I2C/SPI/JTAG) and ACBUS is free to be used as GPIO. The two functions described below have been provided to access these GPIO lines (also called the higher byte lines of MPSSE) that are available in various FTDI chips with MPSSEs.

3.2.1 FT_WriteGPIO

FT_STATUS **FT_WriteGPIO**(FT_HANDLE *handle*, uint8 *dir*, uint8 *value*)

This function writes to the 8 GPIO lines associated with the high byte of the MPSSE channel

Parameters:

in	*handle*	Handle of the channel
in	*dir*	Each bit of this byte represents the direction of the 8 respective GPIO lines. 0 for in and 1 for out
in	*value*	If the direction of a GPIO line is set to output, then each bit of this byte represent the output logic state of the 8 respective GPIO lines. 0 for logic low and 1 for logic high

Returns:

Returns status code of type FT_STATUS

3.2.2 FT_ReadGPIO

FT_STATUS **FT_ReadGPIO**(FT_HANDLE *handle*, uint8 **value*)

This function reads from the 8 GPIO lines associated with the high byte of the MPSSE channel

Parameters:

in	*handle*	Handle of the channel
out	**value*	If the direction of a GPIO line is set to input, then each bit of this byte represent the input logic state of the 8 respective GPIO lines. 0 for logic low and 1 for logic high

Returns:

Returns status code of type FT_STATUS

Note:
The direction of the GPIO line must first be set using FT_WriteGPIO function before this function is used.

3.3 Library Infrastructure Functions

The two functions described in this section typically do not need to be called from the user applications as they are automatically called during entry/exit time. However, these functions are not called automatically when linking the library statically using Microsoft Visual C++. It is then that they need to be called explicitly from the user applications. The static linking sample provided with this manual uses a macro which checks if the code is compiled using Microsoft toolchain, if so then it automatically calls these functions.

3.3.1 Init_libMPSSE

void **Init_libMPSSE**(void)

Initializes the library

Parameters:

in	none	
out	none	

Returns:
void

3.3.2 Cleanup_libMPSSE

void **Cleanup_libMPSSE**(void)

Cleans up resources used by the library

Parameters:

in	none	
out	none	

Returns:
void

3.4 Data types

3.4.1 ChannelConfig

ChannelConfig is a structure that holds the parameters used for initializing a channel. The following are members of the structure:

- uint32 **ClockRate**

This parameter takes the value of the clock rate of the SPI bus in hertz. Valid range for ClockRate is 0 to 30MHz.

- uint8 **LatencyTimer**

Required value, in milliseconds, of latency timer. Valid range is 0 – 255. However, FTDI recommend the following ranges of values for the latency timer:

Range for full speed devices (FT2232D):	Range 2 – 255
Range for Hi-speed devices (FT232H, FT2232H, FT4232H):	Range 1 - 255

- uint32 **configOptions**

Bits of this member are used in the way described below:

Bit number	Description	Value	Meaning of value	Defined macro(if any)
BIT1-BIT0	These bits specify to which of the standard SPI modes should the SPI master be configured to	00	SPI MODE0	SPI_CONFIG_OPTION_MODE0
		01	SPI MODE1	SPI_CONFIG_OPTION_MODE1 (Please refer to the release notes within the release package zip file for revision history and known limitations of this version)
		10	SPI MODE2	SPI_CONFIG_OPTION_MODE2
		11	SPI MODE3	SPI_CONFIG_OPTION_MODE3 (Please refer to the release notes within the release package zip file for revision history and known limitations of this version)
BIT4-BIT2	These bits specify which of the available lines should be used as chip	000	xDBUS3 of MPSSE is chip select	SPI_CONFIG_OPTION_CS_DBUS3
		001	xDBUS4 of MPSSE is	SPI_CONFIG_OPTION_CS_DBUS4

	select		chip select	
		010	xDBUS5 of MPSSE is chip select	SPI_CONFIG_OPTION_CS_DBUS5
		011	xDBUS6 of MPSSE is chip select	SPI_CONFIG_OPTION_CS_DBUS6
		100	xDBUS7 of MPSSE is chip select	SPI_CONFIG_OPTION_CS_DBUS7
BIT5	This bit specifies if the chip select line should be active low	0	Chip select is active high	
		1	Chip select is active low	SPI_CONFIG_OPTION_CS_ACTIVELOW
BIT6-BIT31	Reserved			

Note: The terms xDBUS0 – xDBUS7 corresponds to lines ADBUS0 – ADBUS7 if the first MPSSE channel is used, otherwise it corresponds to lines BDBUS0 – BDBUS7 if the second MPSSE channel(i.e. if available in the chip) is used.

The SPI modes are:

SPI MODE0 - data are captured on rising edge and propagated on falling edge

SPI MODE1 - data are captured on falling edge and propagated on rising edge

SPI MODE2 - data are captured on falling edge and propagated on rising edge

SPI MODE3 - data are captured on rising edge and propagated on falling edge

uint32 Pins

This member specifies the directions and values of the lines associated with the lower byte of the MPSSE channel after SPI_InitChannel and SPI_CloseChannel functions are called.

Bit number	Description	Comment
BIT7-BIT0	Direction of the lines after SPI_InitChannel is called	A 1 corresponds to output and a 0 corresponds to input
BIT15-BIT8	Value of the lines after SPI_InitChannel is called	A 1 corresponds to logic high and a 0 corresponds to logic low
BIT23-BIT16	Direction of the lines after SPI_CloseChannel is called	A 1 corresponds to output and a 0 corresponds to input
BIT31-BIT24	Value of the lines after SPI_CloseChannel is called	A 1 corresponds to logic high and a 0 corresponds to logic low

Note that the directions of the SCLK, MOSI and the specified chip select line will be overwritten to 1 and the direction of the MISO like will be overwritten to 0 irrespective of the values passed by the user application. Other than these 4 lines, the rest of the lines will be an a state that is specified via this parameter.

- uint16 **reserved**

This parameter is reserved and should not be used.

3.4.2 Typedefs

Following are the typedefs that have been defined keeping cross platform portability in view:

- typedef unsigned char **uint8**
- typedef unsigned short **uint16**
- typedef unsigned long **uint32**
- typedef signed char **int8**
- typedef signed short **int16**
- typedef signed long **int32**
- typedef unsigned char **bool**

4 Example Circuit

This example will demonstrate how to connect a MPSSE chip (FT2232H) to an SPI device (93LC56B – EEPROM) and program it using libMPSSE-SPI library.

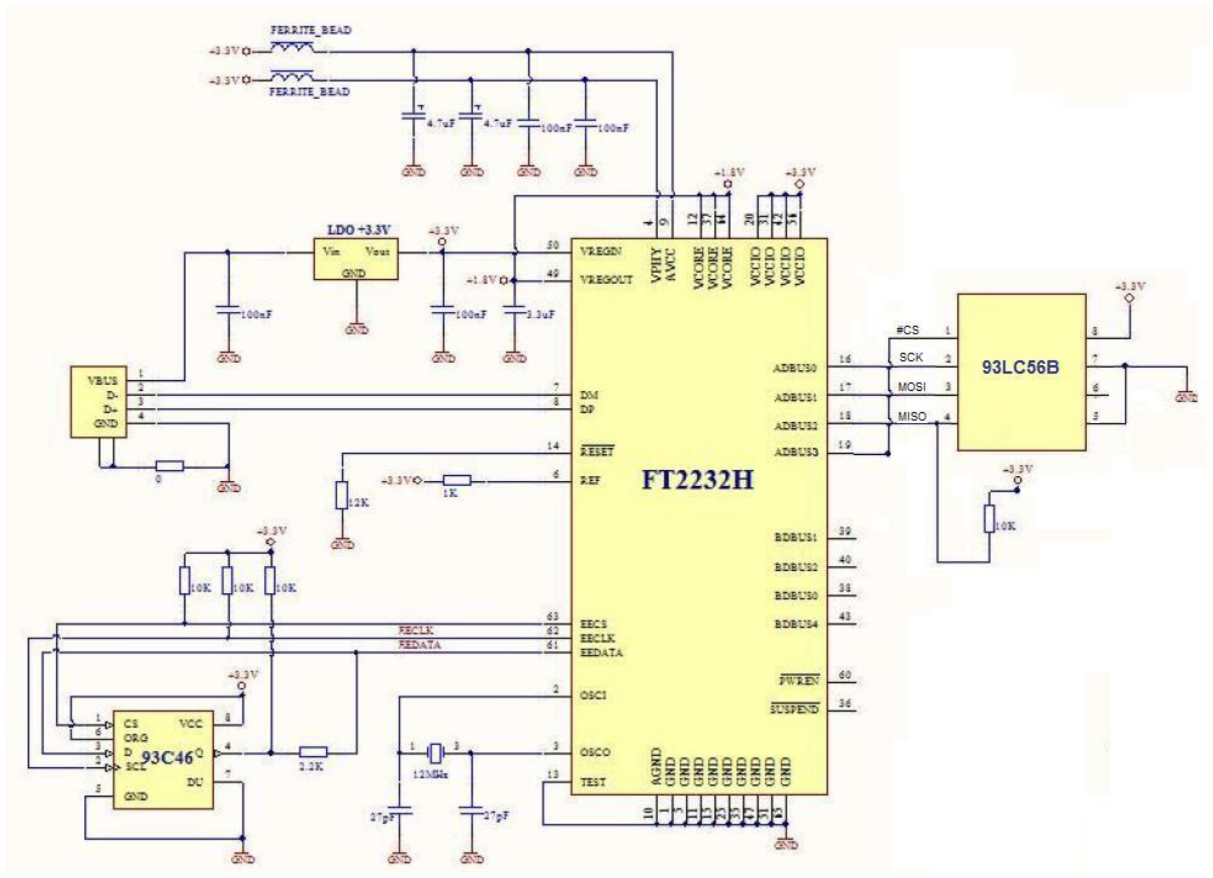


Figure 3: Schematic for connecting FT2232H to SPI EEPROM device (93LC56B)

The above schematic shows how to connect a FT2232H chip to an SPI EEPROM. Please note that the FT2232H chip is also available as a module which contains all the components shown in the above schematic (except the 93LC56B and the pull-up resistors connected to it). This module is called *FT2232H Mini Module* and details about it can be found in the device datasheet. The FT2232H chip acts as the SPI master here and is connected to a PC running using USB interface.

5 Example Program

The required D2XX driver should be installed into the system depending on the OS that is already installed in the PC/host. If a linux PC is used then the default drivers usbserial and ftdi_sio must be removed (using rmmod command).

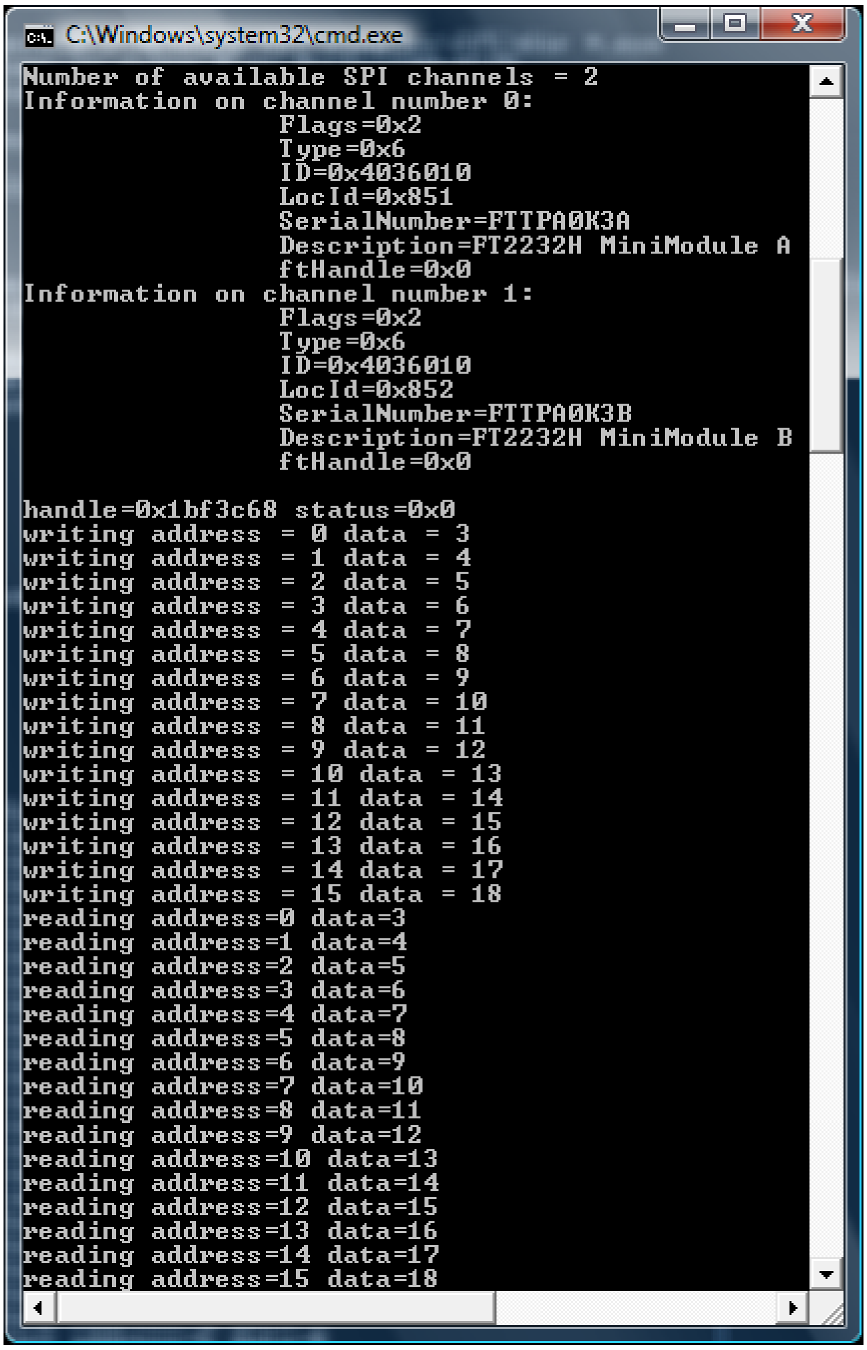
Once the hardware shown above is connected to a PC and the drivers are installed, we can place the following sample code (sample-static.c), D2XX.h, libMPSSE_spi.h and libMPSSE.a into one folder, compile the sample and run it.

```
/*!
 * \file sample-static.c
 *
 * \author FTDI
 * \date 20110512
 *
 * Copyright © 2011 Future Technology Devices International Limited
 * Company Confidential
 *
 * Project: libMPSSE
 * Module: SPI Sample Application - Interfacing 94LC56B SPI EEPROM
 *
 * Rivision History:
 * 0.1 - 20110512 - Initial version
 * 0.2 - 20110801 - Changed LatencyTimer to 255
 *                                   Attempt to open channel only if available
 *                                   Added & modified macros
 *                                   Included stdlib.h
 * 0.3 - 20111212 - Added comments
 */

/******************************************************************************/
/*                                                          Include files
                                                     */
/******************************************************************************/
/* Standard C libraries */
#include<stdio.h>
#include<stdlib.h>
/* OS specific libraries */
#ifdef _WIN32
#include<windows.h>
#endif

/* Include D2XX header*/
#include "ftd2xx.h"

/* Include libMPSSE header */
#include "libMPSSE_spi.h"

/******************************************************************************/
/*                                                                Macro and type defines
                                               */
/******************************************************************************/
/* Helper macros */

#define APP_CHECK_STATUS(exp) {if(exp!=FT_OK){printf("%s:%d:%s(): status(0x%x) \
!= FT_OK\n",__FILE__, __LINE__, __FUNCTION__,exp);exit(1);}else{;}};
#define CHECK_NULL(exp){if(exp==NULL){printf("%s:%d:%s():  NULL expression \
encountered \n",__FILE__, __LINE__, __FUNCTION__);exit(1);}else{;}};

/* Application specific macro definations */
#define SPI_DEVICE_BUFFER_SIZE                      256
```

```
#define SPI_WRITE_COMPLETION_RETRY                10
#define START_ADDRESS_EEPROM             0x00 /*read/write start address inside the EEPROM*/
#define END_ADDRESS_EEPROM               0x10
#define RETRY_COUNT_EEPROM               10         /* number of retries if read/write fails */
#define CHANNEL_TO_OPEN                           0            /*0 for first available channel, 1 for next... */
#define SPI_SLAVE_0                               0
#define SPI_SLAVE_1                               1
#define SPI_SLAVE_2                               2
#define DATA_OFFSET                                            3

/******************************************************************************/
/*                                                                  Global variables
                                                          */
/******************************************************************************/
uint32 channels;
FT_HANDLE ftHandle;
ChannelConfig channelConf;
uint8 buffer[SPI_DEVICE_BUFFER_SIZE];

/******************************************************************************/
/*                                                       Public function definitions
                                                */
/******************************************************************************/
/*!
 * \brief Writes to EEPROM
 *
 * This function writes a byte to a specified address within the 93LC56B EEPROM
 *
 * \param[in] slaveAddress Address of the I2C slave (EEPROM)
 * \param[in] registerAddress Address of the memory location inside the slave to where the byte
 *                                  is to be written
 * \param[in] data The byte that is to be written
 * \return Returns status code of type FT_STATUS(see D2XX Programmer's Guide)
 * \sa Datasheet of 93LC56B http://ww1.microchip.com/downloads/en/DeviceDoc/21794F.pdf
 * \note
 * \warning
 */
FT_STATUS read_byte(uint8 slaveAddress, uint8 address, uint16 *data)
{
        uint32 sizeToTransfer = 0;
        uint32 sizeTransfered;
        bool writeComplete=0;
        uint32 retry=0;
        bool state;
        FT_STATUS status;

        /* CS_High + Write command + Address */
        sizeToTransfer=1;
        sizeTransfered=0;
        buffer[0] = 0xC0;/* Write command (3bits)*/
        buffer[0] = buffer[0] | ( ( address >> 3) & 0x0F );/*5 most significant add bits*/
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES|
                SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE);
        APP_CHECK_STATUS(status);

        /*Write partial address bits */
        sizeToTransfer=4;
        sizeTransfered=0;
        buffer[0] = ( address & 0x07 ) << 5; /* least significant 3 address bits */
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                SPI_TRANSFER_OPTIONS_SIZE_IN_BITS);
        APP_CHECK_STATUS(status);

        /*Read 2 bytes*/
        sizeToTransfer=2;
        sizeTransfered=0;
        status = SPI_Read(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                        SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES|
                        SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE);
        APP_CHECK_STATUS(status);

        *data = (uint16)(buffer[1]<<8);
        *data = (*data & 0xFF00) | (0x00FF & (uint16)buffer[0]);

        return status;
}

/*!
 * \brief Reads from EEPROM
 *
 * This function reads a byte from a specified address within the 93LC56B EEPROM
 *
 * \param[in] slaveAddress Address of the I2C slave (EEPROM)
 * \param[in] registerAddress Address of the memory location inside the slave from where the
 *                          byte is to be read
 * \param[in] *data Address to where the byte is to be read
 * \return Returns status code of type FT_STATUS(see D2XX Programmer's Guide)
 * \sa Datasheet of 93LC56B http://ww1.microchip.com/downloads/en/DeviceDoc/21794F.pdf
 * \note
 * \warning
 */
FT_STATUS write_byte(uint8 slaveAddress, uint8 address, uint16 data)
{
        uint32 sizeToTransfer = 0;
        uint32 sizeTransfered=0;
        bool writeComplete=0;
        uint32 retry=0;
        bool state;
        FT_STATUS status;

        /* Write command EWEN(with CS_High -> CS_Low) */
        sizeToTransfer=11;
        sizeTransfered=0;
        buffer[0]=0x9F;/* SPI_EWEN -> binary 10011xxxxxx (11bits) */
        buffer[1]=0xFF;
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                        SPI_TRANSFER_OPTIONS_SIZE_IN_BITS|
                        SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE|
                        SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE);
        APP_CHECK_STATUS(status);

        /* CS_High + Write command + Address */
        sizeToTransfer=1;
        sizeTransfered=0;
        buffer[0] = 0xA0;/* Write command (3bits) */
        buffer[0] = buffer[0] | ( ( address >> 3) & 0x0F );/*5 most significant add bits*/
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                        SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES|
                        SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE);
        APP_CHECK_STATUS(status);

        /*Write 3 least sig address bits */
        sizeToTransfer=3;
        sizeTransfered=0;
        buffer[0] = ( address & 0x07 ) << 5; /* least significant 3 address bits */
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
                        SPI_TRANSFER_OPTIONS_SIZE_IN_BITS);
        APP_CHECK_STATUS(status);

        /* Write 2 byte data + CS_Low */
        sizeToTransfer=2;
        sizeTransfered=0;
        buffer[0] = (uint8)(data & 0xFF);
        buffer[1] = (uint8)((data & 0xFF00)>>8);
        status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
					SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES|
					SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE);
		APP_CHECK_STATUS(status);

		/* Wait until D0 is high */
#if 1
		/* Strobe Chip Select */
		sizeToTransfer=0;
		sizeTransfered=0;
		status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
					SPI_TRANSFER_OPTIONS_SIZE_IN_BITS|
					SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE);
		APP_CHECK_STATUS(status);
#ifndef __linux__
		Sleep(10);
#endif
		sizeToTransfer=0;
		sizeTransfered=0;
		status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
					SPI_TRANSFER_OPTIONS_SIZE_IN_BITS|
					SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE);
		APP_CHECK_STATUS(status);
#else
		retry=0;
		state=FALSE;
		SPI_IsBusy(ftHandle,&state);
		while((FALSE==state) && (retry<SPI_WRITE_COMPLETION_RETRY))
		{
					printf("SPI device is busy(%u)\n",(unsigned)retry);
					SPI_IsBusy(ftHandle,&state);
					retry++;
		}
#endif
		/* Write command EWEN(with CS_High -> CS_Low) */
		sizeToTransfer=11;
		sizeTransfered=0;
		buffer[0]=0x8F;/* SPI_EWEN -> binary 10011xxxxxx (11bits) */
		buffer[1]=0xFF;
		status = SPI_Write(ftHandle, buffer, sizeToTransfer, &sizeTransfered,
					SPI_TRANSFER_OPTIONS_SIZE_IN_BITS|
					SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE|
					SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE);
		APP_CHECK_STATUS(status);
		return status;
}

/*!
 * \brief Main function / Entry point to the sample application
 *
 * This function is the entry point to the sample application. It opens the channel, writes to the
 * EEPROM and reads back.
 *
 * \param[in] none
 * \return Returns 0 for success
 * \sa
 * \note
 * \warning
 */
int main()
{
		FT_STATUS status;
		FT_DEVICE_LIST_INFO_NODE devList;
		uint8 address=0;
		uint16 data;
		int i,j;
#ifdef _MSC_VER
		Init_libMPSSE();
#endif
```

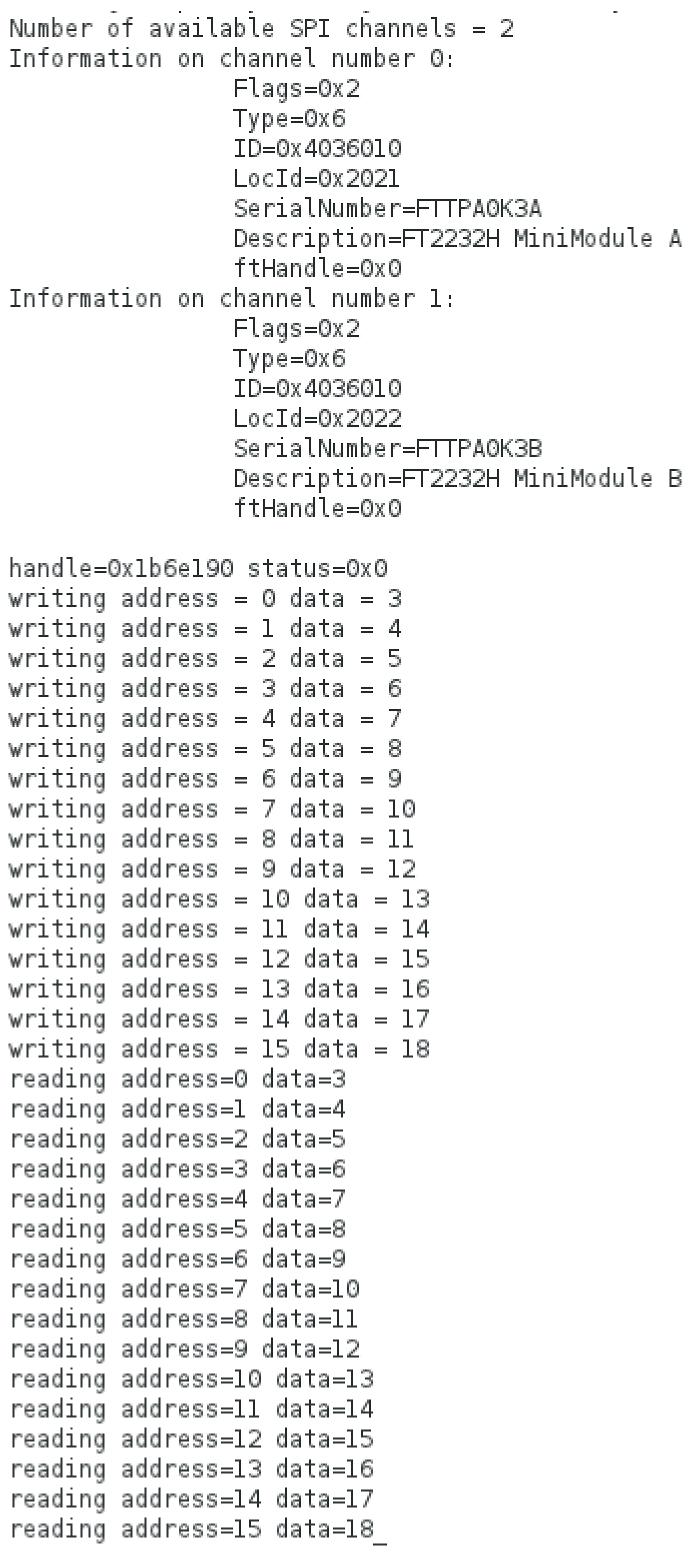
```
	channelConf.ClockRate = 5000;
	channelConf.LatencyTimer= 255;
	channelConf.configOptions = \
		SPI_CONFIG_OPTION_MODE0 | SPI_CONFIG_OPTION_CS_DBUS3;
	channelConf.Pin = 0x00000000;/*FinalVal-FinalDir-InitVal-InitDir (for dir 0=in, 1=out)*/

	status = SPI_GetNumChannels(&channels);
	APP_CHECK_STATUS(status);
	printf("Number of available SPI channels = %d\n",channels);

	if(channels>0)
	{
		for(i=0;i<channels;i++)
		{
			status = SPI_GetChannelInfo(i,&devList);
			APP_CHECK_STATUS(status);
			printf("Information on channel number %d:\n",i);
			/* print the dev info */
			printf("		Flags=0x%x\n",devList.Flags);
			printf("		Type=0x%x\n",devList.Type);
			printf("		ID=0x%x\n",devList.ID);
			printf("		LocId=0x%x\n",devList.LocId);
			printf("		SerialNumber=%s\n",devList.SerialNumber);
			printf("		Description=%s\n",devList.Description);
			printf("		ftHandle=0x%x\n",devList.ftHandle);/*is 0 unless open*/
		}

		/* Open the first available channel */
		status = SPI_OpenChannel(CHANNEL_TO_OPEN,&ftHandle);
		APP_CHECK_STATUS(status);
		printf("\nhandle=0x%x status=0x%x\n",ftHandle,status);
		status = SPI_InitChannel(ftHandle,&channelConf);
		APP_CHECK_STATUS(status);

		for(address=START_ADDRESS_EEPROM;address<END_ADDRESS_EEPROM;address++)
		{
			printf("writing address = %d data = %d\n", address, \
				address+DATA_OFFSET);
			write_byte(SPI_SLAVE_0, address, (uint16)address+DATA_OFFSET);
		}

		for(address=START_ADDRESS_EEPROM;address<END_ADDRESS_EEPROM;address++)
		{
			read_byte(SPI_SLAVE_0, address,&data);
			read_byte(SPI_SLAVE_0, address,&data);
			printf("reading address=%d data=%d\n",address,data);
		}

		status = SPI_CloseChannel(ftHandle);
	}

#ifdef _MSC_VER
	Cleanup_libMPSSE();
#endif

	return 0;
}
```

The sample program shown above writes to address 0 through 15 in the EEPROM chip. The value that is written is *address+3,* i.e. if the address is 5 then a value 8 is written to that address. When this sample program is compiled and run, we should see an output like the one shown below:



```
Number of available SPI channels = 2
Information on channel number 0:
                Flags=0x2
                Type=0x6
                ID=0x4036010
                LocId=0x851
                SerialNumber=FTTPA0K3A
                Description=FT2232H MiniModule A
                ftHandle=0x0
Information on channel number 1:
                Flags=0x2
                Type=0x6
                ID=0x4036010
                LocId=0x852
                SerialNumber=FTTPA0K3B
                Description=FT2232H MiniModule B
                ftHandle=0x0

handle=0x1bf3c68 status=0x0
writing address = 0 data = 3
writing address = 1 data = 4
writing address = 2 data = 5
writing address = 3 data = 6
writing address = 4 data = 7
writing address = 5 data = 8
writing address = 6 data = 9
writing address = 7 data = 10
writing address = 8 data = 11
writing address = 9 data = 12
writing address = 10 data = 13
writing address = 11 data = 14
writing address = 12 data = 15
writing address = 13 data = 16
writing address = 14 data = 17
writing address = 15 data = 18
reading address=0 data=3
reading address=1 data=4
reading address=2 data=5
reading address=3 data=6
reading address=4 data=7
reading address=5 data=8
reading address=6 data=9
reading address=7 data=10
reading address=8 data=11
reading address=9 data=12
reading address=10 data=13
reading address=11 data=14
reading address=12 data=15
reading address=13 data=16
reading address=14 data=17
reading address=15 data=18
```

Figure 4: Sample output on windows



```
Number of available SPI channels = 2
Information on channel number 0:
                Flags=0x2
                Type=0x6
                ID=0x4036010
                LocId=0x2021
                SerialNumber=FTTPA0K3A
                Description=FT2232H MiniModule A
                ftHandle=0x0
Information on channel number 1:
                Flags=0x2
                Type=0x6
                ID=0x4036010
                LocId=0x2022
                SerialNumber=FTTPA0K3B
                Description=FT2232H MiniModule B
                ftHandle=0x0

handle=0x1b6e190 status=0x0
writing address = 0 data = 3
writing address = 1 data = 4
writing address = 2 data = 5
writing address = 3 data = 6
writing address = 4 data = 7
writing address = 5 data = 8
writing address = 6 data = 9
writing address = 7 data = 10
writing address = 8 data = 11
writing address = 9 data = 12
writing address = 10 data = 13
writing address = 11 data = 14
writing address = 12 data = 15
writing address = 13 data = 16
writing address = 14 data = 17
writing address = 15 data = 18
reading address=0 data=3
reading address=1 data=4
reading address=2 data=5
reading address=3 data=6
reading address=4 data=7
reading address=5 data=8
reading address=6 data=9
reading address=7 data=10
reading address=8 data=11
reading address=9 data=12
reading address=10 data=13
reading address=11 data=14
reading address=12 data=15
reading address=13 data=16
reading address=14 data=17
reading address=15 data=18_
```

Figure 5: Sample output on linux

6 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 – 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

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 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

http://ftdichip.com

Appendix A – References

Document References

MPSSE Basics

Command Processor For MPSSE and MCU Host Bus Emulation Modes

D2XX Programmers Guide

D2XX Drivers

FT2232 – Dual Channel MPSSE IC

MPSSE cables

Acronyms and Abbreviations

Terms	Description
GPIO	General Purpose Input/Output
MPSSE	Multi Protocol Synchronous Serial Engine
SPI	Serial Peripheral Interconnect
USB	Universal Serial Bus

Appendix C – Revision History

Document Title:	AN_178 Programming Guide for libMPSSE - SPI
Document Reference No.:	FT_000492
Clearance No.:	FTDI #215
Product Page:	http://www.ftdichip.com/FTProducts.htm
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	First release	2011-08-01
1.1	New function SPI_ReadWrite clocks data in and out simultaneously Read/Write multiple bytes per USB frame when SPI_TRANSFER_OPTION_SIZE_IN_BYTES is enabled	2011-12-12