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.

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.
# Table of Contents

1 Introduction ............................................................................................................. 4  
2 System Overview .................................................................................................... 5  
3 Application Programming Interface (API) .......................................................... 6  
    3.1 SPI Functions ..................................................................................................... 6  
        3.1.1 SPI_GetNumChannels ................................................................................. 6  
        3.1.2 SPI_GetChannelInfo ............................................................................... 6  
        3.1.3 SPI_OpenChannel .................................................................................. 7  
        3.1.4 SPI_InitChannel ................................................................................... 7  
        3.1.5 SPI_CloseChannel ................................................................................... 8  
        3.1.6 SPI_Read .................................................................................................... 8  
        3.1.7 SPI_Write .................................................................................................. 8  
        3.1.8 SPI_ReadWrite ......................................................................................... 9  
        3.1.9 SPI_IsBusy ............................................................................................. 10  
        3.1.10 SPI_ChangeCS ..................................................................................... 10  
    3.2 GPIO functions .................................................................................................. 11  
        3.2.1 FT_WriteGPIO ......................................................................................... 11  
        3.2.2 FT_ReadGPIO ......................................................................................... 11  
    3.3 Library Infrastructure Functions ....................................................................... 12  
        3.3.1 Init_libMPSSE ......................................................................................... 12  
        3.3.2 Cleanup_libMPSSE ................................................................................ 12  
    3.4 Data types ......................................................................................................... 12  
        3.4.1 ChannelConfig ....................................................................................... 12  
        3.4.2 Typedefs .................................................................................................. 14  
4 Example Circuit ........................................................................................................ 15  
5 Example Program ..................................................................................................... 16  
6 Contact Information ................................................................................................. 23  
Appendix A – References ............................................................................................ 24  
    Document References ........................................................................................... 24  
    Acronyms and Abbreviations ................................................................................. 24
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 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.

![The Software & Hardware Stack through which legacy protocol data flows](image)

As shown in Figure 1, 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](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

Figure 2 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 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 on 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 an 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:

<table>
<thead>
<tr>
<th>In</th>
<th>Index</th>
<th>Index of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>Handle</td>
<td>Pointer to the handle of type FT_HANDLE</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>In</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>Config</td>
<td>Pointer to ChannelConfig structure with the value of clock and latency timer updated</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>in</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

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 *sizeTransfer, uint32 transferOptions)

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

Parameters:

<table>
<thead>
<tr>
<th>In</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>Buffer</td>
<td>Pointer to the buffer where data is to be read</td>
</tr>
<tr>
<td>In</td>
<td>sizeToTransfer</td>
<td>Number of bytes or bits to be read</td>
</tr>
<tr>
<td>Out</td>
<td>*sizeTransfer</td>
<td>Pointer to variable containing the number of bytes or bits read</td>
</tr>
<tr>
<td>In</td>
<td>transferOptions</td>
<td>This parameter specifies data transfer options. The bit positions defined for each of these options are: BT0: 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 BT1: 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 BT2: 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 BT3 – BT31: reserved</td>
</tr>
</tbody>
</table>

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 *sizeTransfer, uint32 transferOptions)
This function writes the specified number of bits or bytes (depending on transferOptions parameter) to a SPI slave.

**Parameters:**

<table>
<thead>
<tr>
<th>In</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>Buffer</td>
<td>Pointer to the buffer from where data is to be written</td>
</tr>
<tr>
<td>In</td>
<td>sizeToTransfer</td>
<td>Number of bytes or bits to write</td>
</tr>
<tr>
<td>out</td>
<td>*sizeTransferred</td>
<td>Pointer to variable containing the number of bytes or bits written</td>
</tr>
<tr>
<td>In</td>
<td>transferOptions</td>
<td>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 asserted after the transfer ends. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE. BIT3 – BIT31: reserved</td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>In</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>*inBuffer</td>
<td>Pointer to buffer to which data read will be stored</td>
</tr>
<tr>
<td>out</td>
<td>outBuffer</td>
<td>Pointer to the buffer from where data is to be written</td>
</tr>
<tr>
<td>in</td>
<td>sizeToTransfer</td>
<td>Number of bytes or bits to write</td>
</tr>
<tr>
<td>out</td>
<td>*sizeTransferred</td>
<td>Pointer to variable containing the number of bytes or bits written</td>
</tr>
<tr>
<td>in</td>
<td>transferOptions</td>
<td>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 asserted after the transfer ends. Bit mask defined for this bit is SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE. BIT3 – BIT31: reserved</td>
</tr>
</tbody>
</table>
this bit is
SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE
BIT2: if set then the chip select line is dossed 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 are ready for the next operation. This function
is useful for such applications.

Parameters:

<table>
<thead>
<tr>
<th>in</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>*state</td>
<td>Pointer to a variable to which the state of the MISO line will be read</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>in</th>
<th>handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>configOptions</td>
<td>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</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>in</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Dir</td>
<td>Each bit of this byte represents the direction of the 8 respective GPIO lines. 0 for in and 1 for out</td>
</tr>
<tr>
<td>in</td>
<td>value</td>
<td>If the direction of a GPIO line is set to output, then each bit of this byte represents the output logic state of the 8 respective GPIO lines. 0 for logic low and 1 for logic high</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>in</th>
<th>Handle</th>
<th>Handle of the channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>*value</td>
<td>If the direction of a GPIO line is set to input, then each bit of this byte represents the input logic state of the 8 respective GPIO lines. 0 for logic low and 1 for logic high</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>in</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>none</td>
</tr>
</tbody>
</table>

Returns:

void

3.3.2 Cleanup_libMPSSE

void Cleanup_libMPSSE(void)

Cleans up resources used by the library.

Parameters:

<table>
<thead>
<tr>
<th>in</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>None</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Description</th>
<th>Value</th>
<th>Meaning of value</th>
<th>Defined macro (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT1-BIT0</td>
<td>These bits specify to which of the standard SPI modes should the SPI master be configured to</td>
<td>00</td>
<td>SPI MODE0</td>
<td>SPI_CONFIG_OPTION_MODE0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>SPI MODE1</td>
<td>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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>SPI MODE2</td>
<td>SPI_CONFIG_OPTION_MODE2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>SPI MODE3</td>
<td>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)</td>
</tr>
<tr>
<td>BIT4-BIT2</td>
<td>These bits specify which of the available lines should be used as chip select</td>
<td>000</td>
<td>xDBUS3 of MPSSE is chip select</td>
<td>SPI_CONFIG_OPTION_CS_DBUS3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001</td>
<td>xDBUS4 of MPSSE is chip select</td>
<td>SPI_CONFIG_OPTION_CS_DBUS4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010</td>
<td>xDBUS5 of MPSSE is chip select</td>
<td>SPI_CONFIG_OPTION_CS_DBUS5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011</td>
<td>xDBUS6 of MPSSE is chip select</td>
<td>SPI_CONFIG_OPTION_CS_DBUS6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>xDBUS7 of MPSSE is chip select</td>
<td>SPI_CONFIG_OPTION_CS_DBUS7</td>
</tr>
<tr>
<td>BIT5</td>
<td>This bit specifies if the chip select line should be active low</td>
<td>0</td>
<td>Chip select is active high</td>
<td>SPI_CONFIG_OPTION_CS_ACTIVEHIGH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Chip select is active low</td>
<td>SPI_CONFIG_OPTION_CS_ACTIVELOW</td>
</tr>
<tr>
<td>BIT6-BIT31</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT7-BIT0</td>
<td>Direction of the lines after SPI_InitChannel is called</td>
<td>A 1 corresponds to output and a 0 corresponds to input</td>
</tr>
<tr>
<td>BIT15-BIT8</td>
<td>Value of the lines after SPI_InitChannel is called</td>
<td>A 1 corresponds to logic high and a 0 corresponds to logic low</td>
</tr>
<tr>
<td>BIT23-BIT16</td>
<td>Direction of the lines after SPI_CloseChannel is called</td>
<td>A 1 corresponds to output and a 0 corresponds to input</td>
</tr>
<tr>
<td>BIT31-BIT24</td>
<td>Value of the lines after SPI_CloseChannel is called</td>
<td>A 1 corresponds to logic high and a 0 corresponds to logic low</td>
</tr>
</tbody>
</table>

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 line 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 on 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 an MPSSE chip (FT2232H) to an SPI device (93LC56B – EEPROM) and program it using libMPSSE-SPI library.

![Schematic](image)

**Figure 3 - Schematics 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.

```c
/*!
 * \file sample-static.c
 * \author FTDI
 * \date 20131002
 * Copyright © 2013 Future Technology Devices International Limited
 * Company Confidential
 * Project: libMPSSE
 * Module: SPI Sample Application - Interfacing 94LC56B SPI EEPROM
 * Revision 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
 * 0.4 - 20131002 - 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{;}};

#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 CHANNEL_SLAVE_0 0
#define CHANNEL_SLAVE_1 1
#define CHANNEL_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

FT_STATUS read_byte(uint8 slaveAddress, uint8 address, uint16 *data)
{
    uint32 sizeToTransfer = 0;
    uint32 sizeTransfered = 0;
    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 address 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_BYTES |
                       SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE);
    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; 
} 

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_BITS| 
      SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE| 
      SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE); 
   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]=0x0F; /* 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;
  #endif

int main()
{
  FT_STATUS status;
  FT_DEVICE_LIST_INFO_NODE devList;
  uint8 address=0;
  
  //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
  //\returns Returns 0 for success
  //\sa
  //\warning
  int main()
  {
    FT_STATUS status;
    FT_DEVICE_LIST_INFO_NODE devList;
    uint8 address=0;
The sample program shown above writes to address 0 through 15 in the EEPROM chip. The value that is written is \( \text{address} + 3 \), 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:
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=FTTFA0K3A
   Description=FT2232H MiniModule A
   ftHandle=0x0

Information on channel number 1:
   Flags=0x2
   Type=0x6
   ID=0x4036010
   LocId=0x2022
   SerialNumber=FTTFA0K3B
   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 – 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

http://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.

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

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

<table>
<thead>
<tr>
<th>Terms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO</td>
<td>General Purpose Input/output</td>
</tr>
<tr>
<td>MPSSE</td>
<td>Multi-Protocol Synchronous Serial Engine</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interconnect</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>
Appendix B – List of Tables & Figures

**List of Tables**

NA

**List of Figures**

- Figure 1 - The Software & Hardware Stack through which legacy protocol data flows .................. 4
- Figure 2 - System Organization ........................................................................................................ 5
- Figure 3 - Schematics for connecting FT2232H to SPI EEPROM device (93LC56B) .................. 15
- Figure 4 - Sample Output on Windows ................................................................................................ 21
- Figure 5 - Sample Output on Linux .................................................................................................... 22
### Appendix C – Revision History

Document Title: AN_178 User Guide For libMPSSE - SPI  
Document Reference No.: FT_000492  
Clearance No.: FTDI# 215  
Document Feedback: Send Feedback

<table>
<thead>
<tr>
<th>Revision</th>
<th>Changes</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>First release</td>
<td>01-08-2011</td>
</tr>
<tr>
<td>1.1</td>
<td>New function SPI_ReadWrite clocks data in and out simultaneously</td>
<td>12-12-2011</td>
</tr>
<tr>
<td></td>
<td>Read/Write multiple bytes per USB frame when</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPI_TRANSFER_OPTION_SIZE_IN_BYTES is enabled</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Updated US and CN office addresses</td>
<td>19-05-2020</td>
</tr>
<tr>
<td>1.2</td>
<td>Reviewed for release</td>
<td>18-08-2023</td>
</tr>
</tbody>
</table>