Future Technology Devices International Ltd

Application Note AN_114

Interfacing FT2232H Hi-Speed Devices To SPI Bus

Document Reference No. FT_000149

Version 1.0

Issue Date: 2009-10-20

This application note introduces the SPI synchronous serial communication interface, and illustrates how to implement SPI with the FT2232H. The FT2232H will be used to write and read data to a SPI serial EEPROM.

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

The FT2232H and FT4232H are the FTDI's first USB 2.0 Hi-Speed (480Mbits/s) USB to UART/FIFO ICs. They have the capability of being configured in a variety of serial interfaces using the internal MPSSE (Multi-Protocol Synchronous Serial Engine). The FT2232H device has two independent ports, both of which can be configured using MPSSE while only Channel A and B of FT4232H can be configured using MPSSE.

Using MPSSE can simplify the synchronous serial protocol (USB to SPI, I^2C, JTAG, etc.) design. This application note illustrates how to use the MPSSE of the FT2232H to interface with the SPI bus. Users can use the example schematic (refer to Figure 3) and functional software code (section 3) to begin their design.

Note that the example software is for illustration and is neither guaranteed nor supported by FTDI.

1.1 Overview & Scope

This application note gives details of how to interface and configure the FT2232H to read and write data from a host PC to a serial EEPROM over the serial SPI interface bus. This note includes:

- Overview of SPI communications interface.
- Hardware example of a USB to a serial EEPROM SPI interface using the FT2232H.
- Code example in C++ showing how to configure the FT2232H in SPI mode.
- Oscilloscope plots showing example SPI read and write cycles.

1.2 Overview of SPI Interface

The SPI (Serial to Peripheral Interface) is a master/slave synchronous serial bus that consists of 4 signals. Both command signals and data are sent across the interface. The SPI master initiates all data transactions. Full duplex data transfers can be made up to 30 Mbits/sec with the FT2232H. There is no fixed bit length in SPI. A generic SPI system consists of the following signals and is illustrated in Figure 1.

- Serial Clock (SCLK) from master to slave.
- Serial Data Out (also called Master Out Slave In or MOSI) from master.
- Serial Data In (also called Master In Slave Out or MISO) from slave.
- Chip Select (CS) from master.

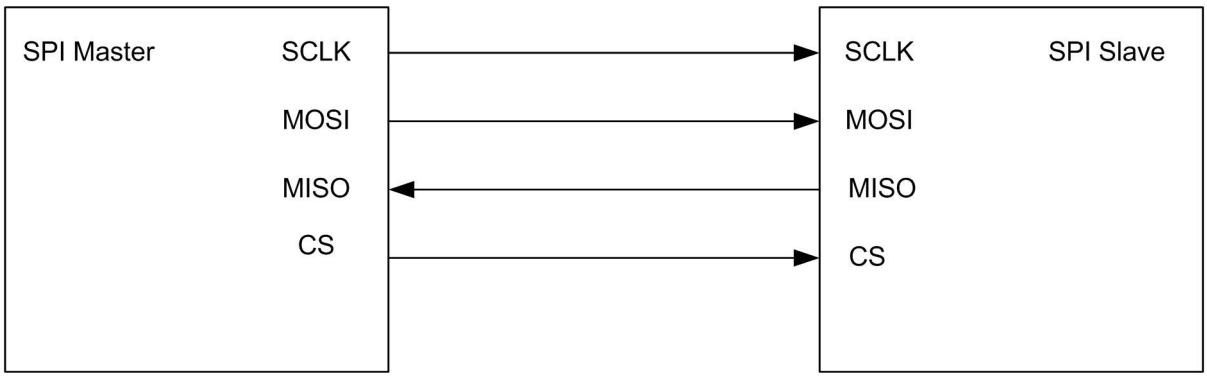

Figure 1 Generic SPI System

The FT2232H always acts as the SPI master. Multiple slave devices can be enabled by multiplexing the chip select line. As SPI data is shifted out of the master and in to a slave device, SPI data will also be shifted out from the slave and clocked in to the master. Depending on which type of slave device is being implemented, data can be shifted MSB first or LSB first. Slave devices can have active low or active high chip select inputs. Figure 2 shows an example SPI timing diagram.

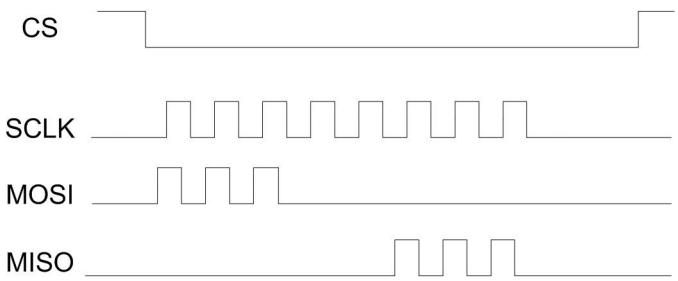

Figure 2 Example SPI Timing Diagram

This SPI device uses SPI Mode 0, with active low Chip Select

In addition, the SPI interface has 4 unique modes of clock phase (CPHA) and clock polarity (CPOL), known as Mode 0, Mode 1, Mode 2 and Mode 3. Table 1 summarizes these modes.

For CPOL = 0, the base (inactive) level of SCLK is 0.

In this mode:

- When CPHA = 0, data will be read in on the rising edge of SCLK, and data will be clocked out on the falling edge of SCLK.
- When CPHA = 1, data will be read in on the falling edge of SCLK, and data will clocked out on the rising edge of SCLK

For CPOL =1, the base (inactive) level of SCLK is 1.

In this mode:

- When CPHA = 0, data will be read in on the falling edge of SCLK, and data will clocked out on the rising edge of SCLK
- When CPHA =1, data will be read in on the rising edge of SCLK, and data will be clocked out on the falling edge of SCLK.

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Table 1 Clock Phase/Polarity Modes

It is worth noting that the SPI slave interface can be implemented in various ways. The FT2232H can be configured to handle these different implementations.

It is recommended that designers review the SPI Slave data sheet to determine the SPI mode implementation.

1.3 FT2232H/FT4232H SPI Pinout

These tables show the location and function of the SPI signal pins on Channel A and B of the FT2232H and FT4232H devices.

Channel A

FT2232H Pin#	FT4232H Pin#	Pin Name	MPSEE Function	Type	Description
16	16	ADBUS0	SCLK	Output	Serial Clock
17	17	ADBUS1	DO (MOSI)	Output	Master Out
18	18	ADBUS2	DI (MISO)	Input	Master In
19	19	ADBUS3	CS	Output	Chip Select

Channel B

FT2232H Pin#	FT4232H Pin#	Pin Name	MPSEE Function	Type	Description
38	26	BDBUS0	SCLK	Output	Serial Clock
39	27	BDBUS1	DO (MOSI)	Output	Master Out
40	28	BDBUS2	DI (MISO)	Input	Master In
41	29	BDBUS3	CS	Output	Chip Select

Table 2 FT2232H/4232H SPI Pinout

2 SPI Design Example

The following design, using the FT2232H, demonstrates how to configure the SPI communication with a Microchip 93LC56 Serial SPI EEPROM. A simplified diagram, Figure 3, illustrates the connections.

For clarity, only the Channel A SPI Pins are shown in figure 3.

Please refer to the FT2232H datasheet & the 93LC56 datasheet for interface details.

The sections which follow provide the user with some example code for setting up the SPI interface.

FT2232H Datasheet

93LC56 Datasheet

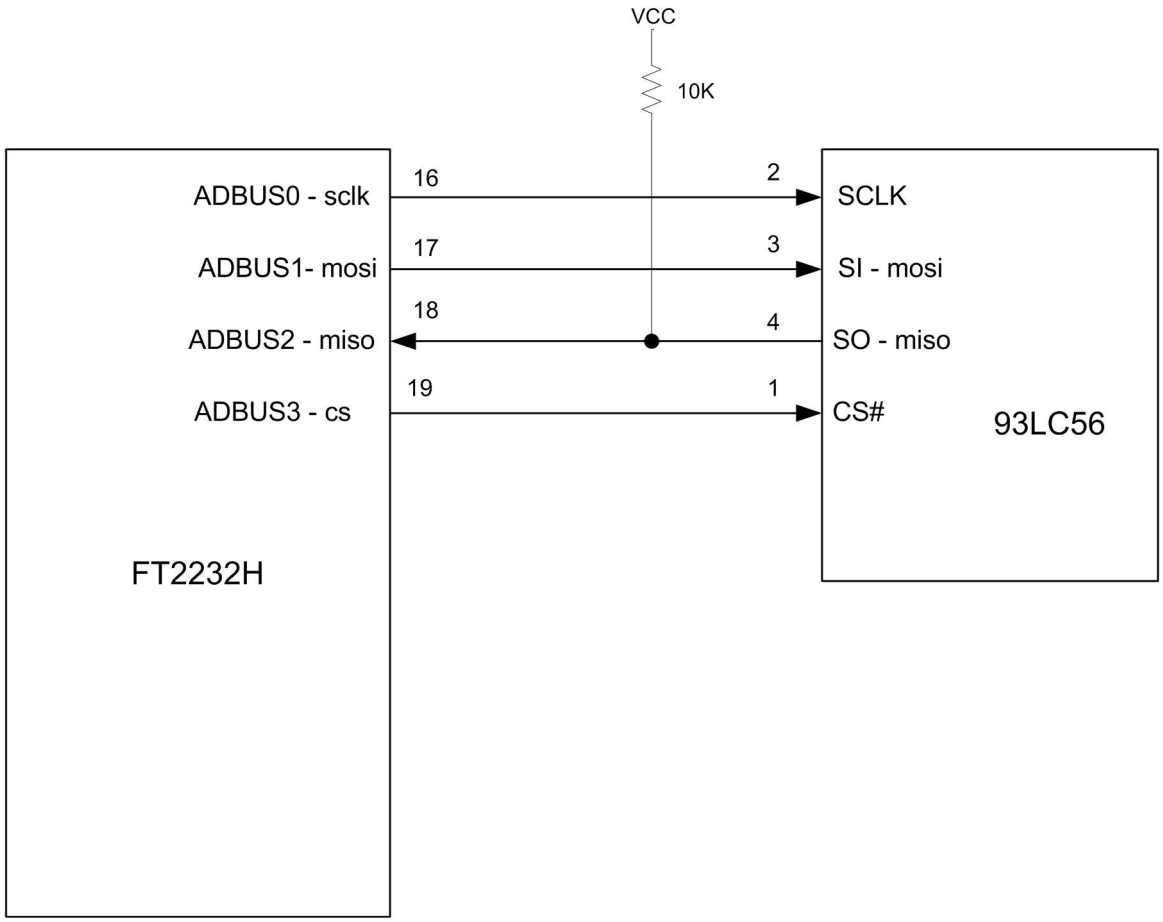

Figure 3 FT2232H to 93LC56 EEPROM

3 Sample SPI Program Code Overview

The sample code provided in the following section configures the FT2232H to function in SPI mode.

The code first verifies that a FTDI Hi Speed device is plugged in, and then identifies the device as either a FT2232H or FT4232H.

Next, the SPI application sends a command to erase the entire EEPROM, and then writes a 16 bit word to the EEPROM.

After each write cycle, the address counter and the data counter are incremented. The new address and data are written to the next memory location.

After 128 bytes of data is written, the application reads back the data and verifies that a successful write of 128 addresses has taken place.

The 128 byte write/read process can be repeated by changing the ”LoopCntr” parameter on page 14.

The code example uses the FTCSPI dll, and requires the use of Microsoft Visual Studio 2008 C++.

The source code for this project can be downloaded from:

http://www.ftdichip.com/Projects/MPSSE/FT2232HS_SPI.zip

The code requires the FTDI D2XX driver:

D2XX Driver link

More information on the D2XX API can be found in the D2XX Programmer’s Guide:

D2XX Programmer's Guide

3.1 C++ Code Listing

```
// Copyright (c) 2009  Future Technology Devices International Ltd.
//
// Module Name
//
//     FT2232HSPIC++NETTestApp.cpp
//
// Abstract:
//
//    This is a C++ console program, which uses FTCSPI.DLL. The FTCSPI.DLL is used to control the FT2232H dual hi-speed device
//    and the FT4232H quad hi-speed device to simulate the Serial Peripheral Interface(SPI) synchronous serial protocol
//    to communicate with SPI devices connected to FT2232H dual hi-speed devices and FT4232H quad hi-speed devices.
//
// Revision History:
//
//   26/08/08    Rev 1
//   21/08/09    Rev 2
//
// FT2232HSPIC++NETTestApp.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

#include <windows.h>

#include "FTCSPI.h"

#define MAX_SPI_93LC56B_CHIP_SIZE_IN_WORDS 128

// To communicate with the 93LC56(2048 word) EEPROM, the maximum frequency the clock can be set is 1MHz

#define MAX_FREQ_93LC56B_CLOCK_DIVISOR 29 //  Sets SCLK to 1 Mhz.

const BYTE SPI_EWEN_CMD = '\x9F'; // set up write enable command
const BYTE SPI_EWDS_CMD = '\x87'; // set up write disable command
const BYTE SPI_ERAL_CMD =  '\x97'; // set up erase all command

#define NUM_93LC56B_CMD_CONTOL_BITS 11    //Number of bits required for 93C56 Control operations
#define NUM_93LC56B_CMD_CONTOL_BYTES 2  // There are 2 control bytes

#define NUM_93LC56B_CMD_DATA_BITS 16      // Number of bits required for 93C56 Data operations
#define NUM_93LC56B_CMD_DATA_BYTES 2

#define MAX_READ_DATA_WORDS_BUFFER_SIZE 65536  // Maximum Number of words to write

typedef WORD ReadDataWordBuffer[MAX_READ_DATA_WORDS_BUFFER_SIZE];
typedef ReadDataWordBuffer *PReadDataWordBuffer;

int _tmain(int argc, _TCHAR* argv[])
{
  FTC_STATUS Status = FTC_SUCCESS;

// Initialize Buffers and Variables

  DWORD dwNumHiSpeedDevices = 0;
  char szDeviceName[100];
  char szChannel[5];
  DWORD dwLocationID = 0;
  DWORD dwHiSpeedDeviceType = 0;
  DWORD dwHiSpeedDeviceIndex = 0;
  FTC_HANDLE ftHandle = 0;
  char szDeviceDetails[150];
  BYTE timerValue = 0;
  DWORD dwClockFrequencyHz = 0;
  BOOL bPerformCommandSequence = false;
  FTC_CHIP_SELECT_PINS ChipSelectsDisableStates;
  FTH_INPUT_OUTPUT_PINS HighInputOutputPins;
  FTH_LOW_HIGH_PINS HighPinsInputData;

FTC_INIT_CONDITION WriteStartCondition;
  WriteControlByteBuffer WriteControlBuffer;
```

```
WriteDataByteBuffer WriteDataBuffer;
  DWORD dwNumDataBytesToWrite = 0;
  FTC_WAIT_DATA_WRITE WaitDataWriteComplete;
  FTH_HIGHER_OUTPUT_PINS HighPinsWriteActiveStates;
  FTC_CLOSE_FINAL_STATE_PINS CloseFinalStatePinsData;
  DWORD dwDataWordInitValue = 0;
  DWORD dwDataWordValue = 0;
  DWORD dwWriteDataWordAddress = 0;
  DWORD dwControlLocAddress1 = 0;
  DWORD dwControlLocAddress2 = 0;
  FTC_INIT_CONDITION ReadStartCondition;
  DWORD dwReadDataWordAddress = 0;
  DWORD dwReadWordValue = 0;
  ReadDataWordBuffer ReadWordData;
  DWORD dwNumDataBytesReturned = 0;
  DWORD dwDataWordWritten = 0;
  DWORD dwCharCntr = 0;
  ReadDataByteBuffer ReadDataBuffer;
  DWORD dwReadDataIndex = 0;
  ReadCmdSequenceDataByteBuffer ReadCmdSequenceDataBuffer;
  int MsgBoxKeyPressed = 0;
  DWORD dwLoopCntr = 0;

char szDllVersion[10];
char szTitleErrorMessage[100];
char szStatusErrorMessage[100];
char szErrorMessage[200];
char szMismatchMessage[100];

for (dwCharCntr = 0; (dwCharCntr < 100); dwCharCntr++)
szMismatchMessage[dwCharCntr] = '\0';

Status = SPI_GetDllVersion(szDllVersion, 10);          // Get version of DLL file
MessageBox(NULL, szDllVersion, "FTCSPI DLL Version", MB_OK);

Status = SPI_GetNumHiSpeedDevices(&dwNumHiSpeedDevices);

if ((Status == FTC_SUCCESS) && (dwNumHiSpeedDevices > 0))
{
do
{

// This gets the name and Location identifier for any FTDI HI Speed device connected to the host.

 Status = SPI_GetHiSpeedDeviceNameLocIDChannel(dwHiSpeedDeviceIndex, szDeviceName, 100, &dwLocationID, szChannel, 5,
&dwHiSpeedDeviceType);

dwHiSpeedDeviceIndex = dwHiSpeedDeviceIndex + 1;   // Increment to look for other HiSpeed devices connected to host.
}
 while ((Status == FTC_SUCCESS) && (dwHiSpeedDeviceIndex < dwNumHiSpeedDevices) && (strcmp(szChannel, "A") != 0));

if (Status == FTC_SUCCESS)
{
if (strcmp(szChannel, "A") != 0)
Status = FTC_DEVICE_IN_USE;
}

 if (Status == FTC_SUCCESS) {

// Opens a handle to the Hi Speed device and initializes the device to default state.

Status = SPI_OpenHiSpeedDevice(szDeviceName, dwLocationID, szChannel, &ftHandle);
 if (Status == FTC_SUCCESS) {

// Returns the device type, either FT2232H or FT4232H

 Status = SPI_GetHiSpeedDeviceType(ftHandle, &dwHiSpeedDeviceType);
 if (Status == FTC_SUCCESS) {

strcpy_s(szDeviceDetails, "Type = ");
```

```
// Is the device a FT4232H ?

if (dwHiSpeedDeviceType == FT4232H_DEVICE_TYPE)  strcat_s(szDeviceDetails, "FT4232H");

// Is the device a FT2232H ?

else if (dwHiSpeedDeviceType == FT2232H_DEVICE_TYPE)   strcat_s(szDeviceDetails, "FT2232H");

strcat_s(szDeviceDetails, ", Name = ");
strcat_s(szDeviceDetails, szDeviceName);

MessageBox(NULL, szDeviceDetails, "Hi Speed Device", MB_OK);
}
}
}
}

if ((Status == FTC_SUCCESS) && (ftHandle != 0))
  {

// Resets the device, initialize the MPSSE, clear buffers, set clock frequency for data in and data out.
Status = SPI_InitDevice(ftHandle, MAX_FREQ_93LC56B_CLOCK_DIVISOR);

 if (Status == FTC_SUCCESS) {
 if ((Status = SPI_GetDeviceLatencyTimer(ftHandle, &timerValue)) == FTC_SUCCESS) {
 if ((Status = SPI_SetDeviceLatencyTimer(ftHandle, 50)) == FTC_SUCCESS) {
 Status = SPI_GetDeviceLatencyTimer(ftHandle, &timerValue);

Status = SPI_SetDeviceLatencyTimer(ftHandle, 16);  // Set Latency Timer to 16 mSec

Status = SPI_GetDeviceLatencyTimer(ftHandle, &timerValue);
}
}
}

if (Status == FTC_SUCCESS)
{
if ((Status = SPI_GetHiSpeedDeviceClock(0, &dwClockFrequencyHz)) == FTC_SUCCESS)
{
if ((Status = SPI_TurnOnDivideByFiveClockingHiSpeedDevice(ftHandle)) == FTC_SUCCESS)
{
Status = SPI_GetHiSpeedDeviceClock(0, &dwClockFrequencyHz);

//  Sets frequency of SCLK by applying clock divisor
if ((Status = SPI_SetClock(ftHandle, MAX_FREQ_93LC56B_CLOCK_DIVISOR, &dwClockFrequencyHz)) == FTC_SUCCESS)
{
//  Turning off divide by five clocking allows SCLK to be set up to 30 Mhz for the FT2232H and FT4232H
if ((Status = SPI_TurnOffDivideByFiveClockingHiSpeedDevice(ftHandle)) == FTC_SUCCESS)
Status = SPI_SetClock(ftHandle, MAX_FREQ_93LC56B_CLOCK_DIVISOR, &dwClockFrequencyHz);
}
}
}
}

if (Status == FTC_SUCCESS)
{
Status = SPI_SetLoopback(ftHandle, false);

if (Status == FTC_SUCCESS)
{
bPerformCommandSequence = false;

if (bPerformCommandSequence == true)
{
if (Status == FTC_SUCCESS)
Status = SPI_ClearDeviceCmdSequence(ftHandle);
}

if (Status == FTC_SUCCESS)

{
// Chip select disable states need to be defined for all the SPI devices connected to a FT2232H hi-speed dual
```

```
// device or FT4332H hi-speed quad device

ChipSelectsDisableStates.bADBUS3ChipSelectPinState = false; // This makes CS active high


ChipSelectsDisableStates.bADBUS4GPIOL1PinState = false;
ChipSelectsDisableStates.bADBUS5GPIOL2PinState = false;
ChipSelectsDisableStates.bADBUS6GPIOL3PinState = false;
ChipSelectsDisableStates.bADBUS7GPIOL4PinState = false;


// Configure the logic states of the  8 GPIO Pins

HighInputOutputPins.bPin1InputOutputState = true;
HighInputOutputPins.bPin1LowHighState = false;
HighInputOutputPins.bPin2InputOutputState = true;
HighInputOutputPins.bPin2LowHighState = false;
HighInputOutputPins.bPin3InputOutputState = true;
HighInputOutputPins.bPin3LowHighState = false;
HighInputOutputPins.bPin4InputOutputState = true;
HighInputOutputPins.bPin4LowHighState = false;

HighInputOutputPins.bPin5InputOutputState = true;
HighInputOutputPins.bPin5LowHighState = false;
HighInputOutputPins.bPin6InputOutputState = true;
HighInputOutputPins.bPin6LowHighState = false;
HighInputOutputPins.bPin7InputOutputState = true;
HighInputOutputPins.bPin7LowHighState = false;
HighInputOutputPins.bPin8InputOutputState = true;
HighInputOutputPins.bPin8LowHighState = false;

Status = SPI_SetHiSpeedDeviceGPIOs(ftHandle, &ChipSelectsDisableStates, &HighInputOutputPins); //NULL

if (Status == FTC_SUCCESS)
{
Sleep(200);

Status = SPI_GetHiSpeedDeviceGPIOs(ftHandle, &HighPinsInputData); //NULL);

Sleep(200);
}
}

do
{
if (Status == FTC_SUCCESS)
{
WriteStartCondition.bClockPinState = false;
WriteStartCondition.bDataOutPinState = false;
WriteStartCondition.bChipSelectPinState = false;  // Makes Chip Select Active High
WriteStartCondition.dwChipSelectPin = ADBUS3ChipSelect;  // Assign ADBUS3 as Chip Select

WaitDataWriteComplete.bWaitDataWriteComplete = false;

// Set the startup states of the 8 general GPIO pins

HighPinsWriteActiveStates.bPin1ActiveState = false;
HighPinsWriteActiveStates.bPin1State = false;
HighPinsWriteActiveStates.bPin2ActiveState = false;
HighPinsWriteActiveStates.bPin2State = false;
HighPinsWriteActiveStates.bPin3State = false;
HighPinsWriteActiveStates.bPin4ActiveState = false;
HighPinsWriteActiveStates.bPin4State = false;
HighPinsWriteActiveStates.bPin5ActiveState = false;
HighPinsWriteActiveStates.bPin5State = false;
HighPinsWriteActiveStates.bPin6ActiveState = false;
HighPinsWriteActiveStates.bPin6State = false;
HighPinsWriteActiveStates.bPin7ActiveState = false;
HighPinsWriteActiveStates.bPin7State = false;
HighPinsWriteActiveStates.bPin8ActiveState = false;
HighPinsWriteActiveStates.bPin8State = false;

}
```

```
// Setup Erase Opcode
WriteControlBuffer[0] = SPI_ERAL_CMD;  // This is the opcode to "Erase All" for the 93C56
WriteControlBuffer[1] = '\FF';


/* Erase the entire 93C56 EEPROM.  This is a control command. No data is sent. The WriteDataBuffer boolean is set to false.
WriteStartCondition true,false indicates data is clocked MSB first, with SPI Mode 0 selected. */

Status = SPI_WriteHiSpeedDevice(ftHandle, &WriteStartCondition, true, false,
NUM_93LC56B_CMD_CONTOL_BITS, &WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES,
false, 0, &WriteDataBuffer, 0, &WaitDataWriteComplete, &HighPinsWriteActiveStates);

Sleep(700);  // Delay for 700 mSec to give erase all command time to execute.


{
WaitDataWriteComplete.bWaitDataWriteComplete = true;
WaitDataWriteComplete.dwWaitDataWritePin = ADBUS2DataIn;
WaitDataWriteComplete.bDataWriteCompleteState = true;
WaitDataWriteComplete.dwDataWriteTimeoutmSecs = 5000;  //def is 5000

dwWriteDataWordAddress = 0;

do
{
// Set up Write Enable Opcode
WriteControlBuffer[0] = SPI_EWEN_CMD;


/* Enable writing. Since this is a control command, the boolean for &WriteDataBuffer is set to false – no data is sent, just the mode
command at the start of the data stream. WriteStartCondition true,false indicates data is clocked MSB first with SPI Mode 0  selected.
*/

Status = SPI_WriteHiSpeedDevice(ftHandle, &WriteStartCondition, true, false,
NUM_93LC56B_CMD_CONTOL_BITS, &WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES,
false, 0, &WriteDataBuffer, 0, &WaitDataWriteComplete, &HighPinsWriteActiveStates);

// Set up write command opcode and address
dwControlLocAddress1 = 160; //'\xA0';
dwControlLocAddress1 = (dwControlLocAddress1 | ((dwWriteDataWordAddress / 8) & '\x0F'));

// Shift left 5 bits ie make bottom 3 bits the 3 MSB's
dwControlLocAddress2 = ((dwWriteDataWordAddress & '\x07') * 32);

WriteControlBuffer[0] = (dwControlLocAddress1 & '\xFF');
WriteControlBuffer[1] = (dwControlLocAddress2 & '\xFF');

if (dwDataWordInitValue == 0)
dwDataWordValue = dwWriteDataWordAddress;
else
dwDataWordValue = dwDataWordInitValue;

// write data
WriteDataBuffer[0] = (dwDataWordValue & '\xFF');
WriteDataBuffer[1] = ((dwDataWordValue / 256) & '\xFF');

/* The SPI write hi speed command includes both a command section and a data section.The Write Data boolean is set true.
WriteStartCondition true,false indicates data is clocked MSB first with SPI Mode 0 selected.  See Figure 3.1 for details. */

Status = SPI_WriteHiSpeedDevice(ftHandle, &WriteStartCondition, true, false,
NUM_93LC56B_CMD_CONTOL_BITS, &WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES,
true, NUM_93LC56B_CMD_DATA_BITS, &WriteDataBuffer, NUM_93LC56B_CMD_DATA_BYTES,
 &WaitDataWriteComplete, &HighPinsWriteActiveStates);

// Disable Writing until next cycle.  This is recommended to ensure data integrity after a write.

WriteControlBuffer[0] = SPI_EWDS_CMD;  // Write Disable opcode for 93C56

/* Disable writing.  This is a control command.  No data is sent. WriteStartCondition true,false indicates data is clocked MSB first with
SPI Mode 0 selected.  */


Status = SPI_WriteHiSpeedDevice(ftHandle, &WriteStartCondition, true, false,
```

```
NUM_93LC56B_CMD_CONTOL_BITS, &WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES,

false, 0, &WriteDataBuffer, 0, &WaitDataWriteComplete, &HighPinsWriteActiveStates);



dwWriteDataWordAddress++;   // Increment data address.

// Write data until the chip size is met. (128 words for 93C56)

while ((dwWriteDataWordAddress < MAX_SPI_93LC56B_CHIP_SIZE_IN_WORDS) && (Status == FTC_SUCCESS));

Sleep(200);

if (Status == FTC_SUCCESS)
{
ReadStartCondition.bClockPinState = false;
ReadStartCondition.bDataOutPinState = false;
ReadStartCondition.bChipSelectPinState = false;  // Chip Select is active high
ReadStartCondition.dwChipSelectPin = ADBUS3ChipSelect;

dwReadDataWordAddress = 0;

do
{
// set up read command and address
dwControlLocAddress1 = 192; //'\xC0';  Read Opcode for 93C56.
dwControlLocAddress1 = (dwControlLocAddress1 | ((dwReadDataWordAddress / 8) & '\x0F'));

 // shift left 5 bits ie make bottom 3 bits the 3 MSB's
dwControlLocAddress2 = ((dwReadDataWordAddress & '\x07') * 32);

WriteControlBuffer[0] = (dwControlLocAddress1 & '\xFF');
WriteControlBuffer[1] = (dwControlLocAddress2 & '\xFF');

if (bPerformCommandSequence == false)

/* For SPI Read, read command and memory address are applied to data stream. WriteControlBuffer contains
dwReadDataWordAddress, which is shifted to format the data stream.  ReadStartCondition true,false indicates data is clocked MSB
first with SPI Mode 0 selected.  Refer to figure 3.2 for details. */

Status = SPI_ReadHiSpeedDevice(ftHandle, &ReadStartCondition, true, false, NUM_93LC56B_CMD_CONTOL_BITS,
&WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES, true, false,  NUM_93LC56B_CMD_DATA_BITS, ReadDataBuffer,
&dwNumDataBytesReturned, &HighPinsWriteActiveStates);

if (Status == FTC_SUCCESS)
 {
dwReadWordValue = ((ReadDataBuffer[1] * 256) | ReadDataBuffer[0]);

ReadWordData[dwReadDataWordAddress] = dwReadWordValue;

// Returns value of address and data to console window.

printf("For Address = %d , Data Read = %d\n ", dwReadDataWordAddress, dwReadWordValue);
}
}
Else
//  AddHiSpeedDeviceRead allows multiple read commands to be staged
//  The remainder of the code is for error detection and reporting.

Status = SPI_AddHiSpeedDeviceReadCmd(ftHandle, &ReadStartCondition, true, false, NUM_93LC56B_CMD_CONTOL_BITS,
&WriteControlBuffer, NUM_93LC56B_CMD_CONTOL_BYTES, true, false, NUM_93LC56B_CMD_DATA_BITS,
&HighPinsWriteActiveStates);

dwReadDataWordAddress++;
}
while ((dwReadDataWordAddress < MAX_SPI_93LC56B_CHIP_SIZE_IN_WORDS) && (Status == FTC_SUCCESS));

MsgBoxKeyPressed = IDOK;
```

```
if (Status == FTC_SUCCESS)
{
if (bPerformCommandSequence == true)
{
Status = SPI_ExecuteDeviceCmdSequence(ftHandle, &ReadCmdSequenceDataBuffer, &dwNumDataBytesReturned);




if (Status == FTC_SUCCESS)
{
if (dwNumDataBytesReturned > 0)
{
dwReadDataWordAddress = 0;

dwReadDataIndex = 0;

do
{

// Scroll through read data buffer
dwReadWordValue = ((ReadCmdSequenceDataBuffer[(dwReadDataIndex + 1)] * 256) |
ReadCmdSequenceDataBuffer[dwReadDataIndex]);

ReadWordData[dwReadDataWordAddress] = dwReadWordValue;

dwReadDataIndex = (dwReadDataIndex + 2);
// Increment Address Counter
dwReadDataWordAddress++;
}
while (dwReadDataIndex < dwNumDataBytesReturned);
}
Else
//  Useful for debugging code. (when configuring opcodes for different format EEPROMs)
MsgBoxKeyPressed = MessageBox(NULL, "No data returned.", "SPI Execute Command Sequence", MB_OK | MB_OKCANCEL);
}
}

if ((Status == FTC_SUCCESS) && (MsgBoxKeyPressed == IDOK))
{
for (dwReadDataIndex = 0; ((dwReadDataIndex < MAX_SPI_93LC56B_CHIP_SIZE_IN_WORDS) && (MsgBoxKeyPressed == IDOK));
dwReadDataIndex++)
{
if (dwDataWordInitValue != 0)
dwDataWordWritten = dwDataWordInitValue;
else
dwDataWordWritten = (dwReadDataIndex & '\xFF');

if (((ReadWordData[dwReadDataIndex] != dwDataWordWritten) && (dwDataWordValue != 0)) ||
((ReadWordData[dwReadDataIndex] != dwDataWordWritten) && (dwDataWordValue == 0)))
{
for (dwCharCntr = 0; (dwCharCntr < 100); dwCharCntr++)
   szMismatchMessage[dwCharCntr] = '\0';

//  If read data does not equal address counter value, error has occurred.

sprintf(szMismatchMessage, "SPI 93LC56B Read/Write mismatch, Loop = %i, Address = %i, Write Value = %X, Read Value = %X",
dwLoopCntr, dwReadDataIndex, dwDataWordWritten, ReadWordData[dwReadDataIndex]);

MsgBoxKeyPressed = MessageBox(NULL, szMismatchMessage, "SPI Read Error Message", MB_OK | MB_OKCANCEL);
}
}
}
}
}
}
dwLoopCntr++;
}
```

```
// Number of iterations - ie: one pass equals 128 read/write cycles.  Multiple iterations are useful for showing SPI signals on scope.


while ((dwLoopCntr < 10) && (Status == FTC_SUCCESS) && (MsgBoxKeyPressed == IDOK));
}
}
}



if (ftHandle != 0)
{

SPI_Close(ftHandle);
ftHandle = 0;
}

if ((Status != FTC_SUCCESS) || (dwNumHiSpeedDevices == 0) || (strlen(szMismatchMessage) > 0))
{
if (Status != FTC_SUCCESS)

// Detect and indicate miscellaneous errors.
{
sprintf_s(szErrorMessage, "Status Code(%u) - ", Status);

Status = SPI_GetErrorCodeString("EN", Status, szStatusErrorMessage, 100);

strcat_s(szErrorMessage, szStatusErrorMessage);

MessageBox(NULL, szErrorMessage, "FTCSPI DLL Error Status Message", MB_OK);
} else {
if (dwNumHiSpeedDevices == 0)
strcpy_s(szErrorMessage, "There are no devices connected.");
else
strcpy_s(szErrorMessage, szMismatchMessage);

MessageBox(NULL, szErrorMessage, "FTCSPI DLL Error Message", MB_OK);
}
}

Else

//  No errors detected, you pass !
{
Status = SPI_GetDllVersion(szDllVersion, 10);

strcpy_s(szErrorMessage, "You have passed the EEPROM test");

MessageBox(NULL, szErrorMessage, "FTCSPI DLL Message", MB_OK);
}

        return 0;
}
```

3.2 FT2232H to 93LC56 Read/Write Timing on Scope

The following screenshots show examples of the Read and write waveforms on the SPI interface. These are provided to illustrate the operational details of the SPI write and read commands sent from the FT2232H to the 93LC56 EEPROM.

Figure 4 Write 3 to Address 3

The Figure 4 screen capture shows a write command (101) applied to MOSI, followed by a 3 written to address 3.

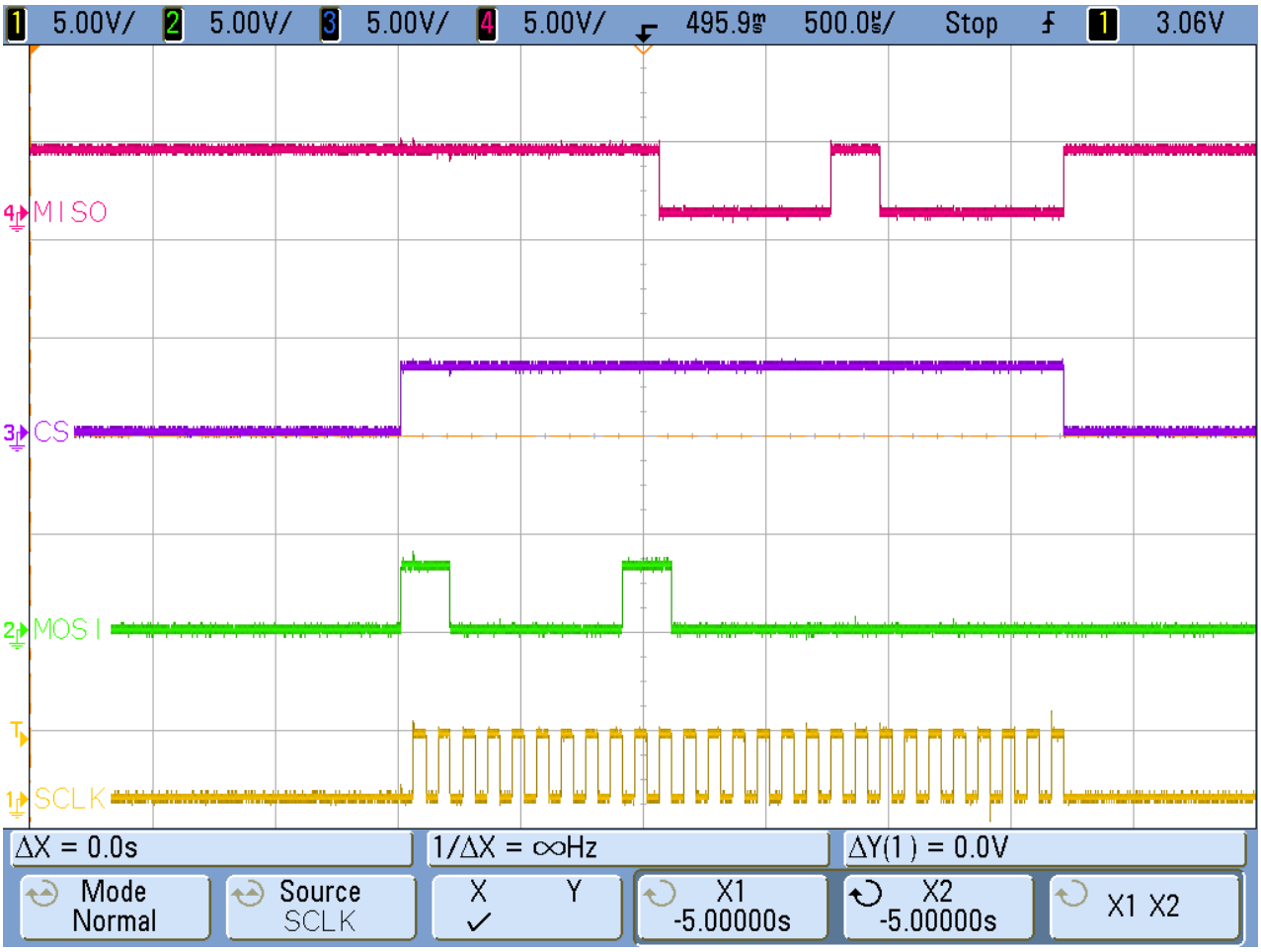

Figure 5 Read 3 from Address 3

The Figure 5 screen capture shows a Read command applied to MOSI (110), followed by the address 3. The MOSI output shows the data read (3).

4 Acronyms and Abbreviations

Terms	Description
MPSSE	Multi Purpose Synchronous Serial Engine
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
JTAG	Joint Test Action Group
USB	Universal Serial bus
Serial EEPROM	A programmable memory chip that uses a bitwise serial interface such as I2C or SPI.
SPI Master	A SPI device that initiates and manages serial communication to all devices connected to its SPI bus.
SPI Slave	A SPI device that responds to commands sent to it by the SPI master.
MISO	Master In, Slave Out
MOSI	Master Out, Slave In

Table 3 Acronyms and Abbreviations

5 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 – 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 – 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.admin@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

Distributors and Sales Representatives

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

Appendix A - References

FTCSPI Programmer's Guide

D2XX Programmer's Guide

Datasheet for FT2232H

Datasheet for 93C56 Serial EEPROM

Appendix B - List of Figures and Tables

List of Figures

List of Tables

Appendix C - Revision History

Revision History

Version 1.0	Initial Release	20th October, 2009