Application Note

AN_177

User Guide For libMPSSE – I2C

Version 1.5

Issue Date: 2020-05-27

This application note is a guide to using the libMPSSE-I2C – a library which simplifies the design of firmware for interfacing to the FTDI MPSSE configured as an I2C 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 to enable 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.

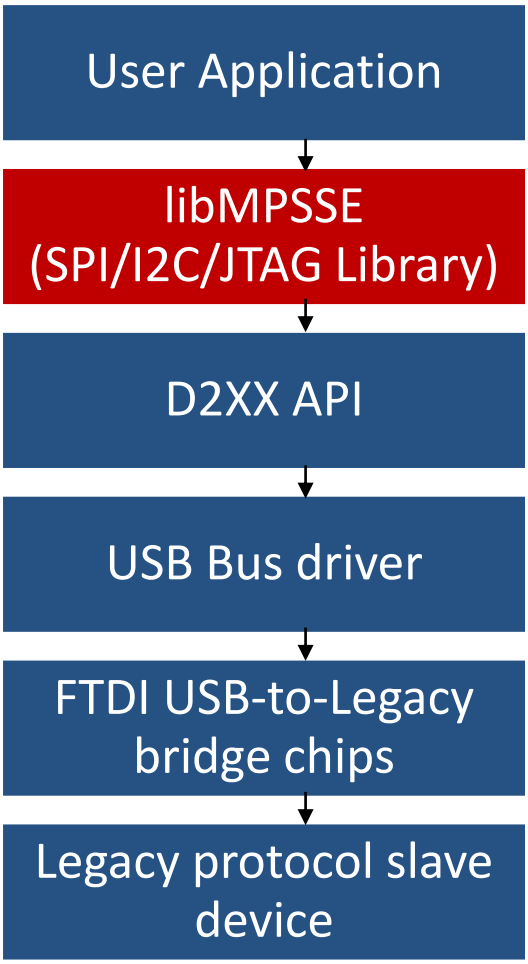

Figure 1 - The Software & Hardware Stack through which legacy protocol data flows

As shown in Figure 1, libMPSSE has three different APIs, one each for I2C, SPI and JTAG. This application note only describes the I2C section. The libMPSSE library (Linux or 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

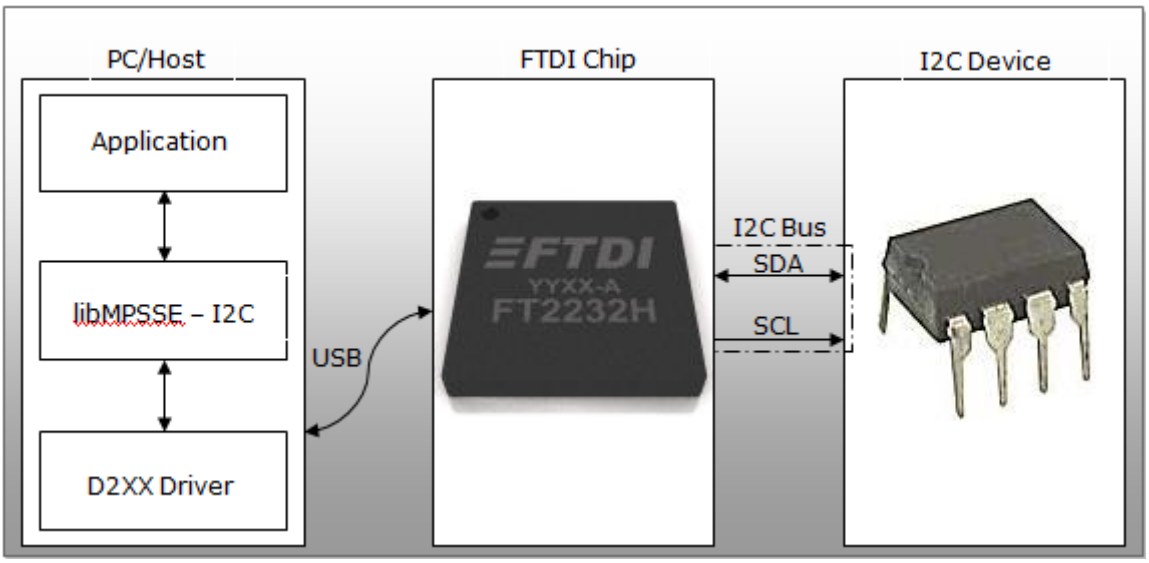

Figure 2 - System Organization

Figure 2 shows how the components of the system are typically organised. The PC/Host may be desktop/laptop machine or an embedded system. The FTDI chip and the I2C device would usually be on the same PCB. Though only one I2C device is shown in the diagram above, many devices can actually be connected to the bus if each device has a different I2C address. I2C devices that support configurable addresses will have pins which can be hardwired to give a device an appropriate address; this information may be found in the datasheet of the I2C device chip.

3 Application Programming Interface (API)

The libMPSSE-I2C APIs can be divided into two broad sets. The first set consists of five 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 I2C Functions

3.1.1 I2C_GetNumChannels

FT_STATUS **I2C_GetNumChannels** (uint32 **numChannels*)

This function gets the number of I2C 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 them but not all these channels can be configured to work as I2C masters. This function returns the total number of channels connected to the host system that has a MPSSE attached to it so that they may be configured as I2C masters.

For example, if an FT2232D (1 MPSSE port), a FT232H (1 MPSSE port), a FT2232H (2 MPSSE port) and a FT4232H (2 MPSSE ports) are connected to a PC, then a call to I2C_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 I2C_GetChannelInfo

FT_STATUS **I2C_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 I2C_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 I2C_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 I2C_OpenChannel

FT_STATUS **I2C_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 I2C_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 returns an error code.

3.1.4 I2C_InitChannel

FT_STATUS **I2C_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. Members of ChannelConfig structure contains the values for I2C master clock, latency timer and Options
out	*None*	

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 I2C_CloseChannel

FT_STATUS **I2C_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 I2C_DeviceRead

FT_STATUS **I2C_DeviceRead**(FT_HANDLE *handle*, uint32 *deviceAddress*, uint32 *sizeToTransfer*, uint8 **buffer,* uint32 **sizeTransferred,* uint32 *options*).

This function reads the specified number of bytes from an addressed I2C slave.

Parameters:

in	*Handle*	Handle of the channel
in	*DeviceAddress*	Address of the I2C slave. This is a 7bit value and it should not contain the data direction bit, i.e. the decimal value passed should be always less than 128
In	*SizeToTransfer*	Number of bytes to be read
out	*Buffer*	Pointer to the buffer where data is to be read
out	*SizeTransferred*	Pointer to variable containing the number of bytes read
in	*options*	This parameter specifies data transfer options. The bit positions defined for each of these options are: BIT0: if set then a start condition is generated in the I2C bus before the transfer begins. A bit mask is defined for this options in file ftdi_i2c.h as I2C_TRANSFER_OPTIONS_START_BIT BIT1: if set then a stop condition is generated in the I2C bus after the transfer ends. A bit mask is defined for this options in file ftdi_i2c.h as I2C_TRANSFER_OPTIONS_STOP_BIT BIT2: reserved (only used in I2C_DeviceWrite) BIT3: some I2C slaves require the I2C master to generate a NAK for the last data byte read. Setting this bit enables working with such I2C slaves. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_NACK_LAST_BYTE BIT4: setting this bit will invoke a multi byte I2C transfer without having delays between the START, ADDRESS, DATA and STOP phases. Size of the transfer in parameters sizeToTransfer and sizeTransferred are in bytes. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES* BIT5: setting this bit would invoke a multi bit transfer without having delays between the START, ADDRESS, DATA and STOP phases. Size of the transfer in parameters sizeToTransfer and sizeTransferred are in bytes. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS* BIT6: the deviceAddress parameter is ignored if this bit is set. This feature may be useful in generating a special I2C bus conditions that do not require any address to be passed. Setting this bit is effective only when either

		I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES or I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS is set. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_NO_ADDRESS* BIT7 – BIT31: reserved

*The I2C_DeviceRead and I2C_DeviceWrite functions send commands to the MPSSE, reads the response and based on the response sends further commands. Delays between START, ADDRESS, DATA and STOP conditions are seen on the I2C bus as a result of waiting for command responses, and also because these commands are sent over different USB transfers. I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES is introduced to minimize these delays by sending multiple MPSSE commands and I2C data over fewer (or possibly just one) USB transfers, without waiting for I2C ack bits to be read into the PC/host. Also, sometimes some I2C devices may require a special non-I2C frame to be sent to it over the I2C bus which may have not have an address phase and may have either more or less than 8 bits in the frame.

I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS & I2C_TRANSFER_OPTIONS_NO_ADDRESS options are introduced to address such needs. For example, some I2C EEPROM chips need a 9bit frame without address to be sent to it to perform a software reset. These bits may be set to implement such features.

I2C_TRANSFER_OPTIONS_START_BIT and I2C_TRANSFER_OPTIONS_STOP_BIT have their usual meanings when used with I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES or I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS, however I2C_TRANSFER_OPTIONS_BREAK_ON_NACK and I2C_TRANSFER_OPTIONS_NACK_LAST_BYTE are not meant to be used with them.

Returns:

Returns status code of type FT_STATUS

Following are the special meanings of the FT_STATUS code when returned from this function:

Return code FT_DEVICE_NOT_FOUND would mean that the I2C slave didn't respond when it was addressed and so the function returned before even beginning any data transfer. Typically this would mean that the address passed to the function was incorrect, or the address of the I2C slave has been configured incorrectly (i.e. if the slave allows it), or the I2C master and the I2C slave isn't connected properly.

Return code FT_INVALID_PARAMETER would mean that the *deviceAddress* that is greater than 127.

Return code FT_IO_ERROR would mean that the transfer failed while actually transferring data

Note:

This function internally performs the following operations:

- Write START bit (if BIT0 of *options* flag is set)
- Write device address
- Get ACK from device
- LOOP until *sizeToTransfer*
  - Read byte to buffer
  - Give ACK
- Write STOP bit(if BIT1 of *options* flag is set)

Warning:

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

3.1.7 I2C_DeviceWrite

FT_STATUS **I2C_DeviceWrite** (FT_HANDLE *handle*, uint32 *deviceAddress*, uint32 *sizeToTransfer*, uint8 **buffer,* uint32 **sizeTransferred,* uint32 *options*)

This function writes the specified number of bytes to an addressed I2C slave.

Parameters:

In	*Handle*	Handle of the channel
In	*deviceAddress*	Address of the I2C slave. This is a 7bit value and it should not contain the data direction bit, i.e. the decimal value passed should be always less than 128
In	*sizeToTransfer*	Number of bytes to be written
out	*Buffer*	Pointer to the buffer from where data is to be written
out	*sizeTransferred*	Pointer to variable containing the number of bytes written
In	*transferOptions*	This parameter specifies data transfer options. The bit positions defined for each of these options are: BIT0: if set then a start condition is generated in the I2C bus before the transfer begins. A bit mask is defined for this options in file ftdi_i2c.h as I2C_TRANSFER_OPTIONS_START_BIT BIT1: if set then a stop condition is generated in the I2C bus after the transfer ends. A bit mask is defined for this options in file ftdi_i2c.h as I2C_TRANSFER_OPTIONS_STOP_BIT BIT2: if set then the function will return when a device nAcks after a byte has been transferred. If not set then the function will continue transferring the stream of bytes even if the device nAcks. A bit mask is defined for this options in file ftdi_i2c.h as I2C_TRANSFER_OPTIONS_BREAK_ON_NACK BIT3: reserved (only used in I2C_DeviceRead) BIT4: setting this bit will invoke a multi byte I2C transfer without having delays between the START, ADDRESS, DATA and STOP phases. Size of the transfer in parameters sizeToTransfer and sizeTransferred are in bytes. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES* BIT5: setting this bit would invoke a multi bit transfer without having delays between the START, ADDRESS, DATA and STOP phases. Size of the transfer in parameters sizeToTransfer and sizeTransferred are in bytes. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS* BIT6: the deviceAddress parameter is ignored if this bit is set. This feature may be useful in generating a special I2C bus conditions that do not require any address to be passed. Setting this bit is effective only when either I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES or I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS is set. The bit mask defined for this bit is I2C_TRANSFER_OPTIONS_NO_ADDRESS* BIT7 – BIT31: reserved

*The I2C_DeviceRead and I2C_DeviceWrite functions send commands to the MPSSE, reads the response and based on the response sends further commands. Delays between START, ADDRESS, DATA and STOP conditions are seen on the I2C bus as a result of waiting for

command responses, and also because these commands are sent over different USB transfers. I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES is introduced to minimize these delays by sending multiple MPSSE commands and I2C data over fewer (or possibly just one) USB transfers, without waiting for I2C ack bits to be read into the PC/host. Also, sometimes some I2C devices may require a special non-I2C frame to be sent to it over the I2C bus which may have not have an address phase and may have either more or less than 8 bits in the frame. I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS & I2C_TRANSFER_OPTIONS_NO_ADDRESS options are introduced to address such needs. For example, some I2C EEPROM chips need a 9bit frame without address to be sent to it to perform a software reset. These bits may be set to implement such features.

I2C_TRANSFER_OPTIONS_START_BIT and I2C_TRANSFER_OPTIONS_STOP_BIT have their usual meanings when used with I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES or I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS, however I2C_TRANSFER_OPTIONS_BREAK_ON_NACK & I2C_TRANSFER_OPTIONS_NACK_LAST_BYTE are not meant to be used with them.

Returns:

Returns status code of type FT_STATUS

Following are the special meanings of the FT_STATUS code returned in the context of I2C:

Return code FT_DEVICE_NOT_FOUND would mean that the I2C slave didn't respond when it was addressed and so the function returned before beginning data transfer. Typically this would mean that the address passed to the function was incorrect, or the device of the I2C slave has been configured incorrectly (i.e. if the slave allows it), or the I2C master and the I2C slave isn't connected properly.

Return code FT_INVALID_PARAMETER would mean that the *deviceAddress* that is greater than 127.

Return code FT_IO_ERROR would mean that the transfer failed while transferring data

Return code FT_FAILED_TO_WRITE_DEVICE would either mean that the I2C slave NAKed

Note:

This function internally performs the following operations:

- Write START bit (if BIT0 of *options* flag is set)
- Write device address
- Get ACK
- LOOP until *sizeToTransfer* (or until device NAK, if BIT2 in *options* is set)
    - Write byte from buffer
    - Get ACK
- Write STOP bit(if BIT1 of *options* flag is set)

Warning:

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

3.2 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 FT223H has only one MPSSE channel with two 8-bit busses, 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:

- I2C_CLOCKRATE **ClockRate**

  Valid range for clock divisor is from 0 to 3400000

  The user can pass either I2C_CLOCK_STANDARD_MODE, I2C_CLOCK_FAST_MODE, I2C_CLOCK_FAST_MODE_PLUS or I2C_CLOCK_HIGH_SPEED_MODE for the standard clock rates; alternatively a value for a non-standard clock rate may be passed directly.

- 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:

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

- uint32 **Options**

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

Bit number	Description	Value	Meaning of value	Defined macro(if any)
BIT0	These bit specify if 3-phase-clocking is enabled or disabled	0	3-phase-clocking enabled*	
		1	3-phase-clocking is disabled*	I2C_DISABLE_3PHASE_CLOCKING
BIT1	Setting this bit will enable Drive-Only-Zero feature	0	Drive-Only-Zero disabled**	
		1	Enable Drive-Only-Zero**	I2C_ENABLE_DRIVE_ONLY_ZERO
BIT2 – BIT31	Reserved			

*Please note that 3-phase-clocking is available only on the hi-speed devices and not on the FT2232D.

**Enabling Drive-Only-Zero ensures that the SDA line is driven by the I2C master only when it is supposed to be driven LOW, and tristate it when it is supposed to be driven HIGH. This feature is available only in FT232H chip. Trying to enable this feature using function I2C_Init will have no effect on chips other than FT232H.

3.4.2 I2C_CLOCKRATE

I2C_CLOCKRATE is an enumerated data type that is defined as follows –

- enum I2C_ClockRate_t { I2C_CLOCK_STANDARD_MODE = 100000,
- I2C_CLOCK_FAST_MODE = 400000,
- I2C_CLOCK_FAST_MODE_PLUS = 1000000,
- I2C_CLOCK_HIGH_SPEED_MODE = 3400000 }

3.4.3 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 demonstrates how to connect a MPSSE chip (FT2232H) to an I2C device (24LC024H – EEPROM) and how to program it using libMPSSE-I2C library.

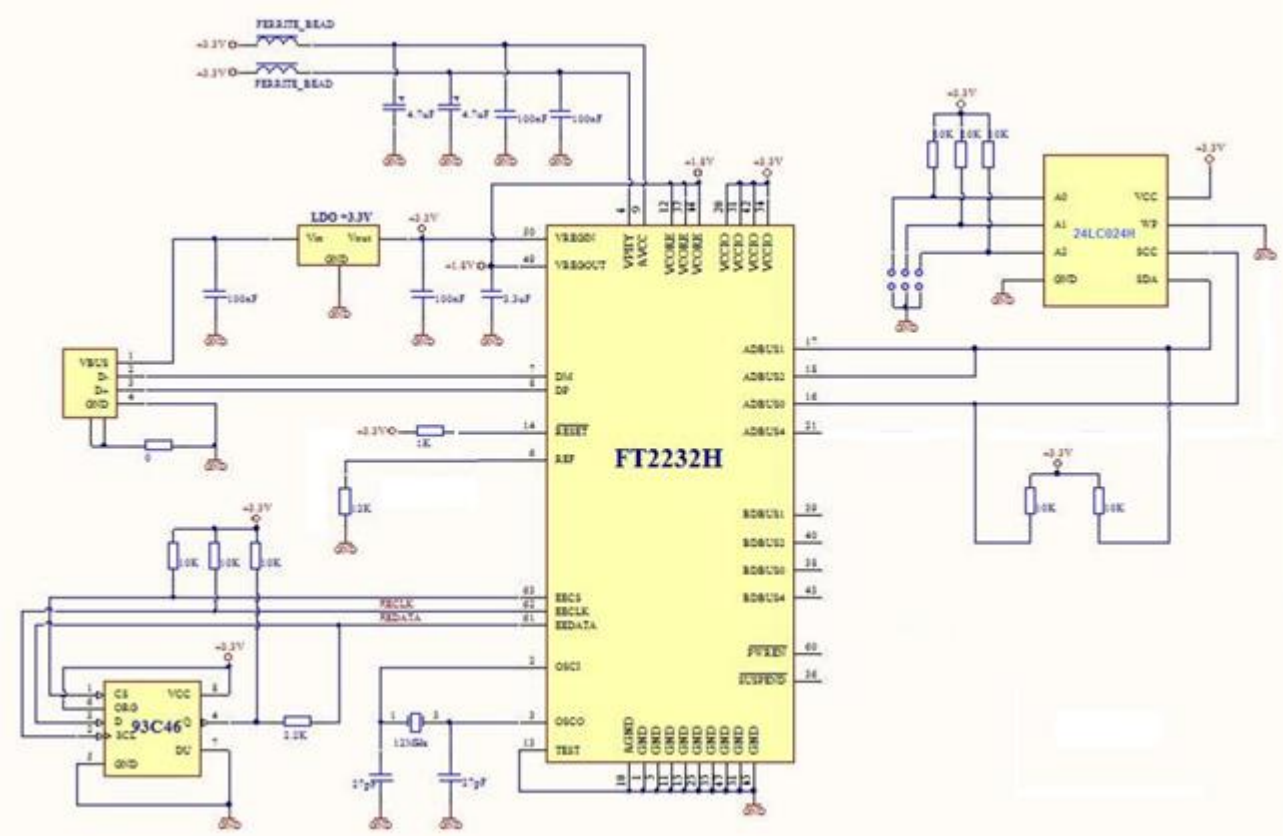

Figure 3 - Schematic for connecting FT2232H to I2C EEPROM device (24LC024H)

The above schematic shows how to connect a FT2232H chip to an I2C EEPROM. Please note that the FT2232 chip is also available as a module which contains all the components shown in the above schematic (except the 24LC024H and its address line pull-up resistors). This module is called *FT2232H Mini Module* and details about it can be found in the device datasheet. The FT2232H chip acts as the I2C master here and is connected to a PC using USB interface. For the example we connected lines A0, A1 and A2 of 24LC024H chip to logic HIGH (using the 10K pull-up resistors), this gave the chip an I2C device address of 0x57.

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, the user can place the following code (sample-win32-static.c), D2XX.h, libMPSSE_i2c.h and libMPSSE.a into one folder, compile the sample and run it.

```
/*!
 * \file sample-static.c
 *
 * \author FTDI
 * \date 20131002
 *
 * Copyright © 2013 Future Technology Devices International Limited
 * Company Confidential
 *
 * Project: libMPSSE
 * Module: I2C Sample Application - Interfacing 24LC024H I2C EEPROM
 *
 * Rivision History:
 * 0.1 - 20110513 - initial version
 * 0.2 - 20110801 - Changed LatencyTimer to 255
 *                  Attempt to open channel only if available
 *                  Added & modified macros
 *                  Change I2C_GetChannelInfo & OpenChannel to start indexing from 0
 * 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_i2c.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 definitions */
#define I2C_DEVICE_ADDRESS_EEPROM       0x57
#define I2C_DEVICE_BUFFER_SIZE          256
#define I2C_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                     1       /*0 for first available channel, 1 for next... */
#define DATA_OFFSET                         1


/*****************************************************************************/
/*                                                                           Global variables
                                                */
/*****************************************************************************/
uint32 channels;
FT_HANDLE ftHandle;
ChannelConfig channelConf;
FT_STATUS status;
uint8 buffer[I2C_DEVICE_BUFFER_SIZE];

/*****************************************************************************/
/*                                                  Public function definitions
                                        */
/*****************************************************************************/
/*!
 * \brief Writes to EEPROM
 *
 * This function writes a byte to a specified address within the 24LC024H 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 24LC024H http://ww1.microchip.com/downloads/en/devicedoc/22102a.pdf
 * \note
 * \warning
 */
FT_STATUS write_byte(uint8 slaveAddress, uint8 registerAddress, uint8 data)
{
        uint32 bytesToTransfer = 0;
        uint32 bytesTransfered;
        bool writeComplete=0;
        uint32 retry=0;

        bytesToTransfer=0;
        bytesTransfered=0;
        buffer[bytesToTransfer++]=registerAddress; /* Byte addressed inside EEPROM */
        buffer[bytesToTransfer++]=data;
        status = I2C_DeviceWrite(ftHandle, slaveAddress, bytesToTransfer, buffer, \
&bytesTransfered, I2C_TRANSFER_OPTIONS_START_BIT|I2C_TRANSFER_OPTIONS_STOP_BIT);

        /* poll to check completion */
        while((writeComplete==0) && (retry<I2C_WRITE_COMPLETION_RETRY))
        {
                bytesToTransfer=0;
                bytesTransfered=0;
                buffer[bytesToTransfer++]=registerAddress; /* Addressed inside EEPROM  */
                status = I2C_DeviceWrite(ftHandle, slaveAddress, bytesToTransfer,\
                        buffer, &bytesTransfered, \

        I2C_TRANSFER_OPTIONS_START_BIT|I2C_TRANSFER_OPTIONS_BREAK_ON_NACK);
                if((FT_OK == status) && (bytesToTransfer == bytesTransfered))
                {
                        writeComplete=1;
                        printf("  ... Write done\n");
                }
                retry++;
```

```
        }
        return status;
}

/*!
 * \brief Reads from EEPROM
 *
 * This function reads a byte from a specified address within the 24LC024H 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 24LC024H http://ww1.microchip.com/downloads/en/devicedoc/22102a.pdf
 * \note
 * \warning
 */
FT_STATUS read_byte(uint8 slaveAddress, uint8 registerAddress, uint8 *data)
{
        FT_STATUS status;
        uint32 bytesToTransfer = 0;
        uint32 bytesTransfered;

        bytesToTransfer=0;
        bytesTransfered=0;
        buffer[bytesToTransfer++]=registerAddress; /*Byte addressed inside EEPROM */
        status = I2C_DeviceWrite(ftHandle, slaveAddress, bytesToTransfer, buffer, \
                &bytesTransfered, I2C_TRANSFER_OPTIONS_START_BIT);
        APP_CHECK_STATUS(status);
        bytesToTransfer=1;
        bytesTransfered=0;
        status |= I2C_DeviceRead(ftHandle, slaveAddress, bytesToTransfer, buffer, \
                &bytesTransfered, I2C_TRANSFER_OPTIONS_START_BIT);
        APP_CHECK_STATUS(status);
        *data = buffer[0];
        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;
        uint8 data;
        int i,j;

#ifdef _MSC_VER
        Init_libMPSSE();
#endif
        channelConf.ClockRate = I2C_CLOCK_FAST_MODE;/*i.e. 400000 KHz*/
        channelConf.LatencyTimer= 255;
        //channelConf.Options = I2C_DISABLE_3PHASE_CLOCKING;
        //channelConf.Options = I2C_ENABLE_DRIVE_ONLY_ZERO;
```

```
	status = I2C_GetNumChannels(&channels);
	APP_CHECK_STATUS(status);
	printf("Number of available I2C channels = %d\n",channels);

	if(channels>0)
	{
		for(i=0;i<channels;i++)
		{
			status = I2C_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 = I2C_OpenChannel(CHANNEL_TO_OPEN,&ftHandle);
		APP_CHECK_STATUS(status);
		printf("\nhandle=0x%x status=%d\n",ftHandle,status);
		status = I2C_InitChannel(ftHandle,&channelConf);
		APP_CHECK_STATUS(status);

		for(address=START_ADDRESS_EEPROM;address<END_ADDRESS_EEPROM;address++)
		{
			printf("writing address = %d data = %d", address, \
				address+DATA_OFFSET);
			status = write_byte(I2C_DEVICE_ADDRESS_EEPROM, address, \
				address+DATA_OFFSET);
			for(j=0; ((j<RETRY_COUNT_EEPROM) && (FT_OK !=status)); j++)
			{
				printf("---- writing again to address = %d, data =%d\n", \
					address, address+DATA_OFFSET);
				status = write_byte(I2C_DEVICE_ADDRESS_EEPROM, address, \
					address+DATA_OFFSET);
			}
			APP_CHECK_STATUS(status);
		}
		printf("\n");
		for(address=START_ADDRESS_EEPROM; address<END_ADDRESS_EEPROM; address++)
		{
			status = read_byte(I2C_DEVICE_ADDRESS_EEPROM,address, &data);
			for(j=0; ((j<RETRY_COUNT_EEPROM) && (FT_OK !=status)); j++)
			{
				printf("read error... retrying \n");
				status = read_byte(I2C_DEVICE_ADDRESS_EEPROM,address, &data);
			}
			printf("reading address %d data read=%d\n",address,data);
		}
		status = I2C_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+1,* i.e. if the address is 5 then a value 6 is written to that address. When this sample program is compiled and run, we should see an output like the one shown below:

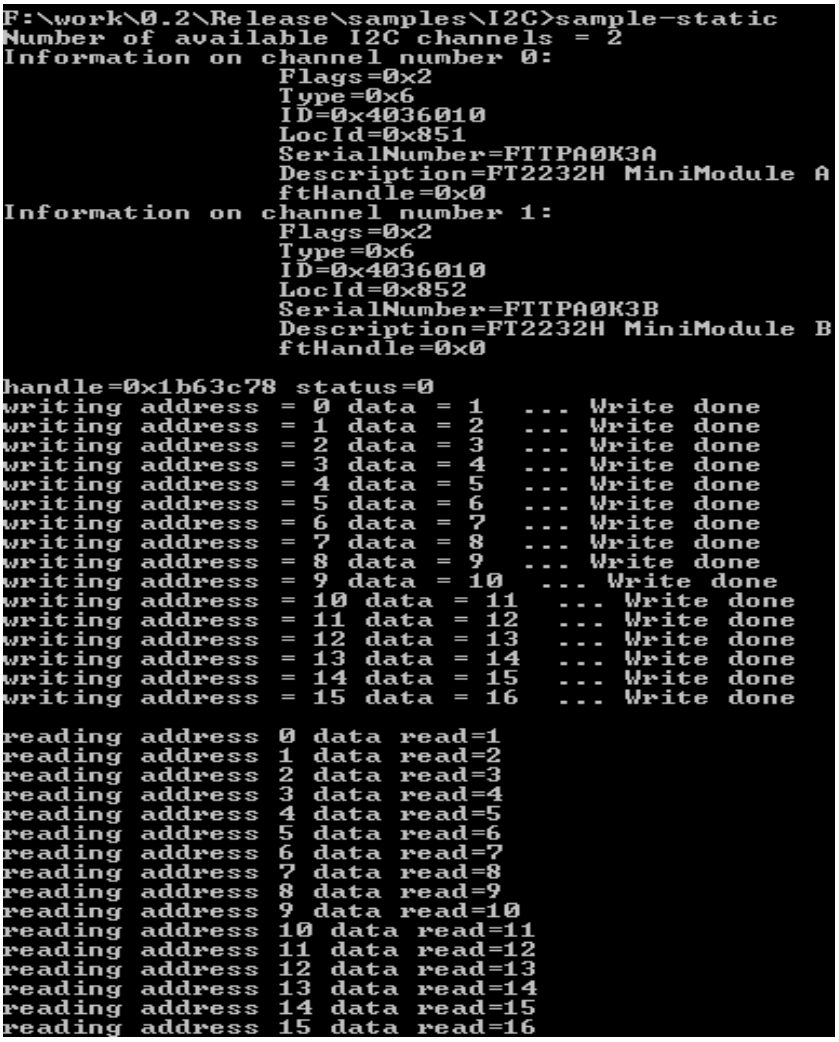

Figure 4 - Sample Output on Windows

```
Number of available I2C 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=0x9a2190 status=0
writing address = 0 data = 1  ... Write done
writing address = 1 data = 2  ... Write done
writing address = 2 data = 3  ... Write done
writing address = 3 data = 4  ... Write done
writing address = 4 data = 5  ... Write done
writing address = 5 data = 6  ... Write done
writing address = 6 data = 7  ... Write done
writing address = 7 data = 8  ... Write done
writing address = 8 data = 9  ... Write done
writing address = 9 data = 10  ... Write done
writing address = 10 data = 11  ... Write done
writing address = 11 data = 12  ... Write done
writing address = 12 data = 13  ... Write done
writing address = 13 data = 14  ... Write done
writing address = 14 data = 15  ... Write done
writing address = 15 data = 16  ... Write done

reading address 0 data read=1
reading address 1 data read=2
reading address 2 data read=3
reading address 3 data read=4
reading address 4 data read=5
reading address 5 data read=6
reading address 6 data read=7
reading address 7 data read=8
reading address 8 data read=9
reading address 9 data read=10
reading address 10 data read=11
reading address 11 data read=12
reading address 12 data read=13
reading address 13 data read=14
reading address 14 data read=15
reading address 15 data read=16
```

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.

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 B – List of Tables & Figures

List of Tables

NA

List of Figures

Appendix C – Revision History

Document Title: AN_177 User Guide For libMPSSE – I2C

Document Reference No.: FT_000466

Clearance No.: FTDI#210

Product Page: http://www.ftdichip.com/FTProducts.htm

Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2011-05-23
1.1	Corrected section 3.1.2 : I2C_GetNumChannels -1 Corrected section 3.2.3 : wrong typedef uintT32 Corrected heading on sections 3.1.3 to 3.1.7 which had wrong text Corrected TOC	2011-05-25
1.2	Added section "Library Infrastructure Functions" Updated sample application Added Linux specific guidelines and download files	2011-06-22
1.3	Added GPIO functions. Added option to disable 3-phase-clocking. Renamed I2C_Device_Read / I2C_Device_Write to I2C_DeviceRead / I2C_DeviceWrite Added note on latency timer value Updated sample application	2011-08-01
1.4	Features added: 1) I2C_TRANSFER_OPTIONS_NACK_LAST_BYTE 2) I2C_TRANSFER_OPTIONS_BREAK_ON_NACK 3) I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BYTES 4) I2C_TRANSFER_OPTIONS_FAST_TRANSFER_BITS 5) I2C_TRANSFER_OPTIONS_NO_ADDRESS 6) I2C_ENABLE_DRIVE_ONLY_ZERO Address provided should be less than 128 Returns FT_DEVICE_NOT_FOUND ff no slave respond when addressed	2011-12-12
1.5	Corrected text explaining Drive-Only-Zero macro in section 3.4.1 Updated US and CN office addresses	2020-05-27