Application Note

AN_355

FT232H MPSSE Example - I2C Master with Visual Basic

Version 1.1

Issue Date: 2020-01-10

This application note provides an example of using the MPSSE feature of the FT232H device as an I^2C Master with a Visual Basic .NET project. It can be used with the FTDI FT232H modules (such as UM232H, UM232H-B, C232HM) in addition to custom boards containing the FT232H IC.

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.

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
Copyright © Future Technology Devices International Limited

Table of Contents

Copyright © Future Technology Devices International Limited

Copyright © Future Technology Devices International Limited

1 Introduction

This application note provides an example of using the MPSSE feature of the FT232H device as an I²C Master with a Visual Basic .NET 2013 project.

In this project, the device is used to interface to a Texas Instruments INA219 voltage and current sensing amplifier, in order to create a small adapter for measuring the DC voltage and current consumed by a device.

The application is intended as an example of using the FT232H MPSSE to implement an I²C master but also provides a useful tool when developing applications using USB peripherals and other low-voltage circuits. For example, it could be used for monitoring the consumption of an FTDI VM800P EVE board, or an FT900 MCU board, or a variety of other low voltage DC devices. Monitoring the current consumption can be very helpful both in verifying expected operation and identifying issues during debugging since many aspects of a device's operation are reflected in the current which it consumes.

The project could be easily modified to use a variety of different I²C sensor devices (temperature, light, force, ADC and many others) to create monitoring applications.

A big advantage of using MPSSE is that there is no firmware to develop, program and maintain. The FT232H is a hardware bridge supplied ready to use, with the MPSSE controlled entirely by commands from the PC. This means that any changes in functionality of the end product (such as to add features, read different registers in the I²C peripherals or add support for different I²C peripherals) can all be implemented with a new release of the application program running on the PC.

This application note demonstrates the following principles:

- Using the FTDI D2XX Drivers with Visual Basic NET applications
- Using the FT232H's MPSSE to implement I²C protocol
- Displaying the gathered data in a graphical user interface
- Using the AD3:7 pins as GPIO (useful for C232HM cable applications where only ADBUS is accessible)

Note: This software is intended only for the FT232H device as it uses the open-drain mode available on this device. The FT2232H and FT4232H devices also contain an MPSSE but require additional commands in order to tristate the I²C lines to simulate an open-drain configuration. Application notes AN_411 (C#) and AN_113 (C++) show examples using this technique.

Copyright © Future Technology Devices International Limited

2 Hardware

2.1 Hardware Description

This section describes the hardware used for this example.

UM232H

This module is designed by FTDI to allow easy evaluation of the FT232H and includes all components needed for the device including the USB connector and internal EEPROM. Its dual-in-line footprint allows it to be connected to a breadboard, solder pad board or turned-pin IC socket. It can also be used in the final product to simplify the USB part of the hardware. Note that connections must be made to the VIO and 5V0 power pins before the module will be recognised by a PC.

Power

A small regulator provides a stable 3v3 supply to the INA219 as the USB 5V supply can vary within the tolerance limits defined in the USB spec. The FT232H has a built-in regulator supplying 3v3 which could also be used since the INA219 consumes very little current. Since this internal regulator was supplying the VCCIO rail and the flashing status LED, the separate regulator was selected to ensure a stable supply to the amplifier in this case.

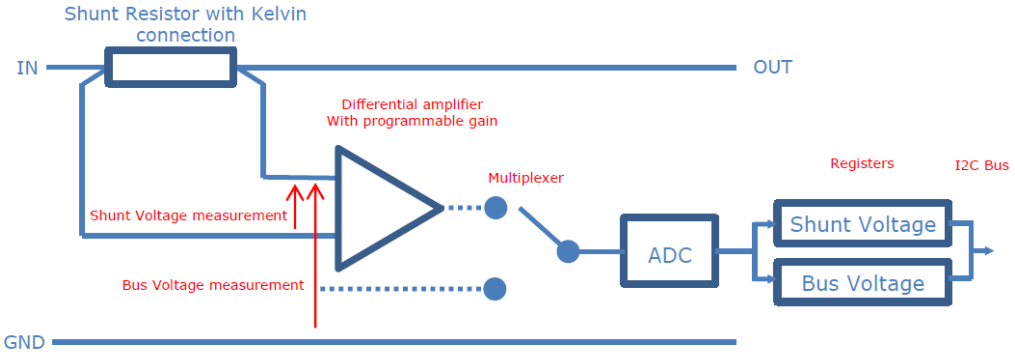
Current Shunt Monitor (INA219)

A current shunt monitor (also known as a current sense amplifier) works by measuring the voltage (the shunt voltage) across a low-value sense resistor which is connected in series with the load to be measured. This allows the current to be determined by a simple Ohms-law calculation.

One key feature of a high-side current shunt monitor amplifier is its ability to measure the difference across a shunt resistor when neither end is at ground potential, and to turn this into a ground-referenced voltage which can be measured by an ADC. This allows it to be connected in the positive side of the power supply to be measured. Since circuits often have several paths to ground, it can be difficult to measure a current accurately if the sense resistor had to be connected in the ground side (low side current sensing).

The INA219 also measures the voltage with respect to ground at the downstream end of the sense resistor, which represents the real voltage that the load sees taking account of any small voltage drop in the sense resistor itself.

Whilst many current shunt monitors output a voltage or current proportional to the shunt voltage, this particular device has an ADC built-in and so can be connected directly via I^2C, which helps reduce the component count and simplifies the hardware design significantly.



Note: This diagram is intended to explain the principles used in this example application and not the full feature set or detailed architecture of the INA219

Figure 1 – Current shunt monitor principles

Copyright © Future Technology Devices International Limited

The INA219's I²C address can be configured via the two address pins. Details are given in the INA219 datasheet. In this example, both lines are grounded for an address of 0x40.

A 0.05 Ohm sense resistor is used to minimise the voltage drop. The sense lines of the amplifier may have small series resistors (e.g. up to 10 Ohms) to provide additional protection against spikes etc. which could occur on the VBUS connection being monitored. An optional capacitor allows filtering to be applied to the measurement if required. The B version (INA219B) was selected as it has higher accuracy.

Due to the surface mount package of this device (and many other sensor devices) a break-out board may be used for prototyping. For example, the Roth RE906 board can be used for the INA219's SOT package. Breakout boards for the device are also available ready-made from various vendors.

Connectors

Since this application is intended to measure USB VBUS currents, two USB connectors were also added in a pass-through configuration to enable the unit to be connected in-line with an existing USB connection. The GND, DP and DM lines are linked directly from the IN connector to the OUT connector. The VBUS line between the two connectors goes through the current sensing resistor. The diagram below shows measurement of a PC mouse current consumption.

The current in, current out and ground connections are also available on a pin header on the rear panel to allow jumper wires to be used instead of the USB connections, which is useful if measuring the voltage and current on a power rail on an evaluation board or similar. Since both the USB connectors and the pin header share the same current sensing circuit, only one of these connection types should be used at any time. The diagram shows measurement of the consumption of an FTDI VM800 being powered from a PSU.

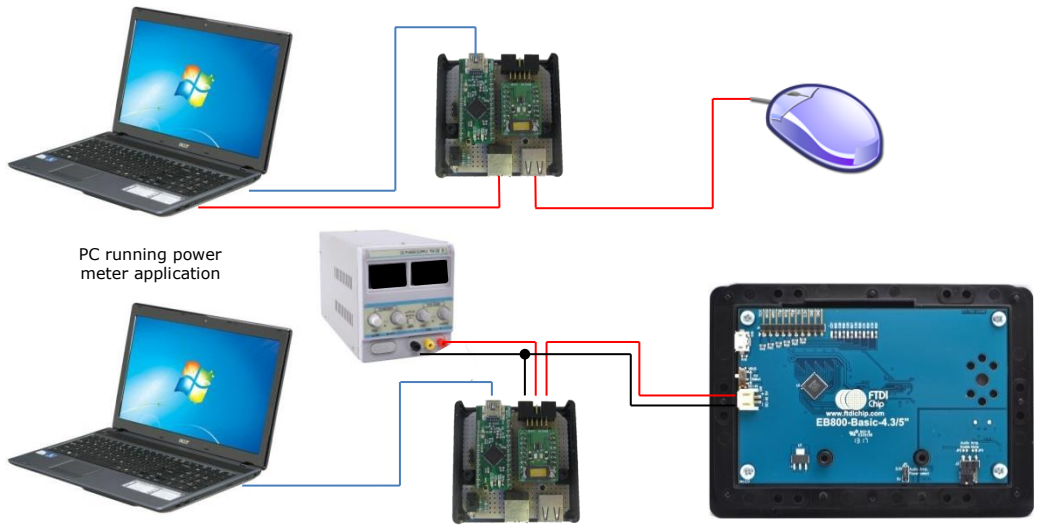


Figure 2 – Examples of USB and pin header measurement connections

GPIO

A status LED was also added to demonstrate the way in which the other pins on port ADBUS which are not used for I²C (AD3-7) can be used as GPIO lines. This is especially useful for the C232HM cables which have no ACBUS access and so any GPIO must be on the ADBUS.

Two of these ADBUS lines are available on the rear connector. The software configures them as inputs and reflects their state via radio buttons. This could be developed to show for example a marker on the current graph when triggered or to provide a start/stop signal from hardware. Or one or both could be configured as an output to provide a signal to another piece of equipment when a certain current level is observed.

Copyright © Future Technology Devices International Limited

The ACBUS lines are also available for GPIO purposes, where AC7:0 provide up to 8 additional I/Os. It is therefore possible to have up to 13 GPIO lines in addition to the I^2C lines.

I^2C

For the I^2C itself, the lines AD2:0 are used. AD0 is the MPSSE's clock out pin and is therefore connected to the SCL line of the I^2C bus. The MPSSE always generates the clock and as such is always the I^2C Master. It is not designed for multi-master operation.

For data, the MPSSE has two separate pins; ADBUS 1 is the Data Out pin and ADbus2 is the Data In pin. Since the I^2C bus uses a single bi-directional data line, the data in and data out lines are connected together to allow data to be both written out and read in. The I^2C libraries in the software application set the pin states to be output for AD0 (clock) and AD1 (data out) and to be an input for AD2 (data in). The FT232H has a drive-only-zero feature which allows any of AD7:0 and AC7:0 to be selected as open-drain.

FT232H EEPROM Settings

It is recommended that the FT232H has a configuration EEPROM fitted and that the device is set to FIFO mode. Provided that the RD# and SI/WU# lines on ACBUS are de-asserted (pulled high) by the hardware then, the 8 ADBUS lines will begin as tri-state when the FT232H starts up. If set for UART mode, the ADBUS lines would drive out their idle UART states until the I2C_ConfigureMPSSE function is called. The device can enter MPSSE mode from either UART or FIFO but the FIFO mode avoids any contention in the time between the user connecting the hardware and starting the application. Note that even in FIFO mode, some ACBUS lines are driven (e.g. TXE#) and so this should be considered when assigning ACBUS lines for GPIO purposes. Asynchronous FIFO pin assignments can be found in the FT232H datasheet.

It is suggested that the VCP option be turned off in the EEPROM so that the UM232H is not accidentally opened by the user in other COM port applications such as terminals. This ensures the port is available to be found by the meter application.

With an EEPROM, the device description string and serial number can also be changed to allow easy identification of the device. For example, the code provided looks for description "UM232H".

Note that FTDI FT232H modules/cables such as UM232H and C232HM have configuration EEPROMs fitted on-board. FTDI's free FT_PROG software can be used to change EEPROM settings over USB if required.

2.2 Prototype Hardware

Figure 3 – Prototype Hardware

Copyright © Future Technology Devices International Limited

2.3 Schematic

The following diagram shows the schematic of the hardware unit:

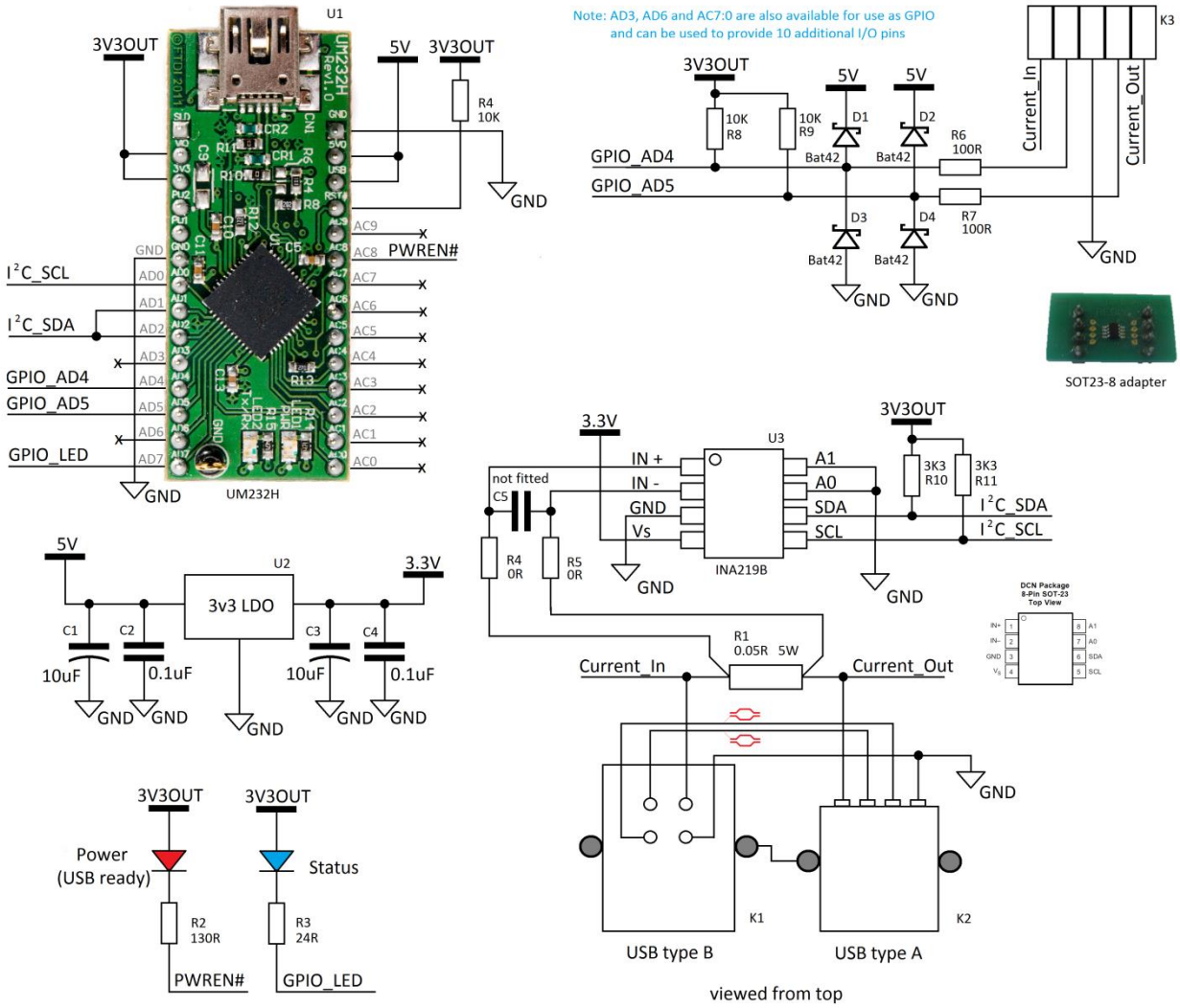


Figure 4 – Schematic of the prototype meter

Copyright © Future Technology Devices International Limited

3 Software

The software is written in Visual Studio 2013 using Visual Basic NET. Later versions of Visual Studio should also upgrade the project when opening.

The code for this application note is arranged in three particular files. These are discussed further in the following sections.

- **Application code (see iChart.vb)**

  The main application itself, which handles the user interface and calls the generic I^2C functions below.

- **I^2C Functions (see FT232H_I2C.vb)**

  These functions form a generic I^2C library and can be called from the Application code. This allows them to be used for a variety of different applications. Their main purpose is to convert the I^2C calls from the main application into a series of MPSSE commands which are in turn sent using the calls to the D2xx driver.

- **D2xx VB Wrapper (see FTDI_VB_Wrapper.vb)**

  When programming in VB, a wrapper is needed to allow the application to call the D2xx functions in the FTDI driver. FTDI also have a wrapper for C# applications. C++ applications can call the functions directly.

The source code for this Application Note can be downloaded here.

3.1 Sample code package

Two different example code projects have been included in the provided zip file; the full current meter application and a simplified I^2C application listing. They both share the same library and wrapper layers.

The simplified example allows the I^2C parts of the code to be easily identified without the additional graphics code for the chart etc. and may be a good starting point for developing other applications which don't require a chart display.

Note: This application is intended to demonstrate the MPSSE programming required to implement an I^2C Master interface. Some error checking and handling have been implemented where possible without affecting readability of the MPSSE code. However, the code is not intended to reflect all best practices for Windows application programming such as error handling and GUI implementation.

Both the main application and the library functions are intended to be used as the basis for further development and may require customisation to suit the particular application and I^2C peripheral. The functions are not intended as a ready-made library to suit all applications without modification. This flexibility to customise the code at lower levels allows fine-tuning of the I^2C routines to suit the intended application.

Copyright © Future Technology Devices International Limited

4 Current Meter Application Code

This section briefly outlines the main meter application. The application consists of the main window and has handlers for each of the buttons.

The user interface has the following features:

- Numerical readout of current and voltage values
- Device details panel showing how many FTDI devices in total are connected and the status of the meter
- Two radio buttons used as indicators only, showing the state of the two GPIO lines which are available on the pin header of the meter
- Scrolling chart showing the profile of the measured current. Three ranges available for the chart are selectable via radio buttons, allowing the user to select full scale of 200mA, 800mA or 1600mA.
- Save chart button allows saving of the chart area as a bitmap image file for future reference or inclusion in documents etc.
-

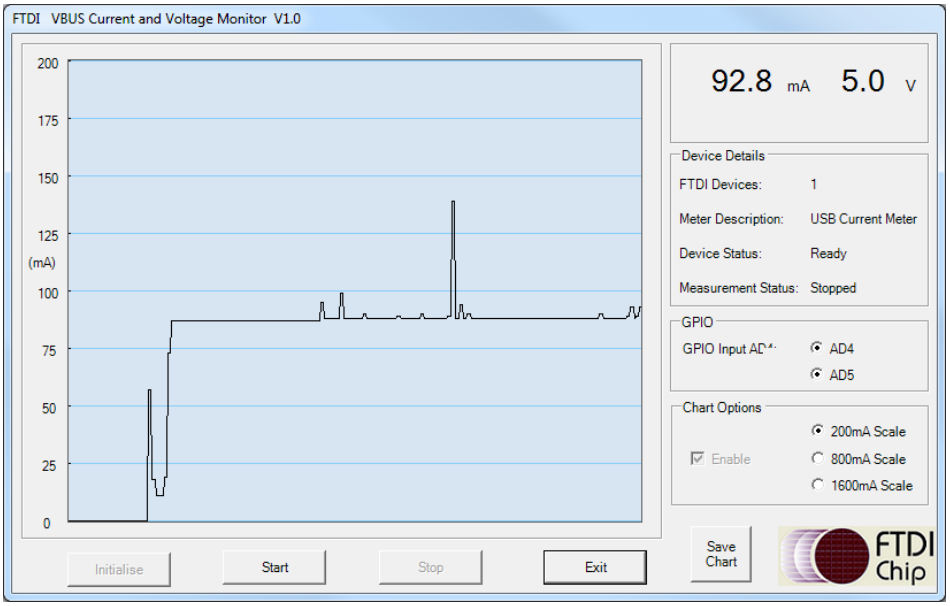


Figure 5 – Application window showing connection of a Flash drive

4.1 Form1_Load

On loading of the form, this handler sets the properties of the buttons and radio buttons to their initial states.

4.2 Button_Init_Click

When the user clicks the Initialise button, D2xx calls are used to determine how many FTDI devices are connected. If the user runs the program without having any FTDI driver installed (or if they have run the setup executable but not connected a device yet) then the program will fail to find the D2xx DLL. A try-catch around the first call will catch this exception in a user-friendly manner instead of a system exception.

The application then does an open by description of "UM232H". This string should be changed if the module or cable being used has a different string. Or a drop-down could be added to the GUI to allow the user to select the device. Then, the I2C_ConfigureMPSSE function is called to set the device up for MPSSE

Copyright © Future Technology Devices International Limited

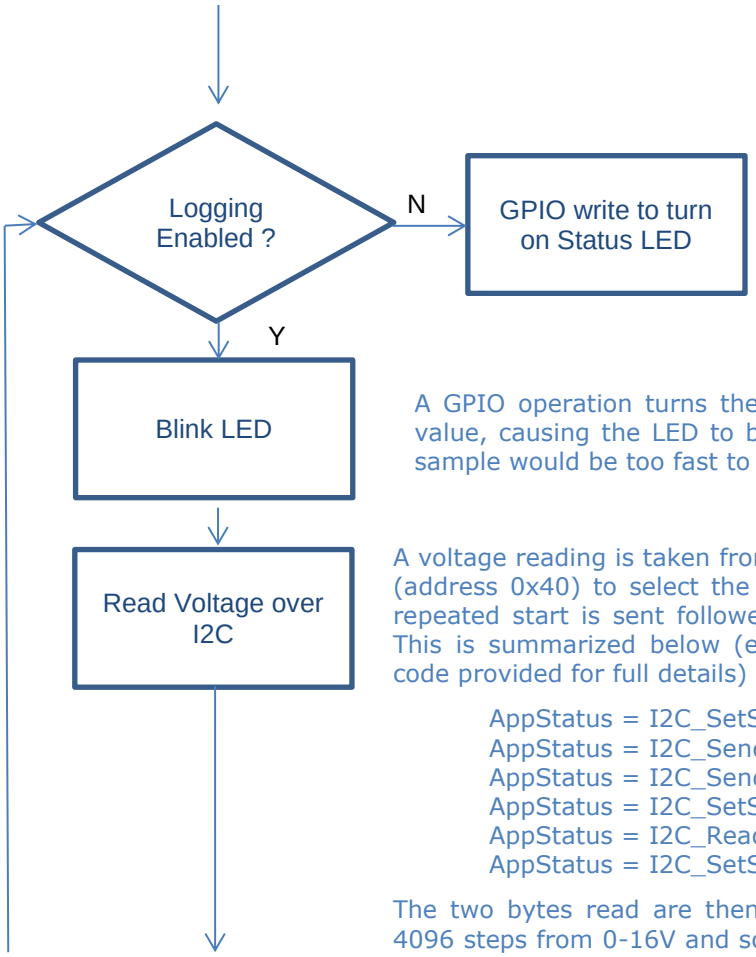
operation. On completion, the status LED on ADBUS 7 is illuminated by setting the global variables for GPIO data and direction and calling I2C_SetLinesIdle.

4.3 Button_Start_Click

When the user then clicks the start button, the application uses the I²C functions to write to the configuration register of the INA219. It sets the full scale for voltage to 16V and the full scale for current to 80mV sense voltage.

Note: Full details of the INA219's registers, result format and I²C protocol can be found in the INA219 datasheet. The FTDI application note AN_356 also has additional information on using the INA219.

It then enters a loop which runs as long as the LoggingEnabled variable is True. This loop carries out the actions detailed below. Note that a more advanced application could start a thread which carries out the data collection and passes it back to the main application for display.



If the logging is being stopped, a GPIO operation ensures the status LED is turned on which indicates device configured but stopped.

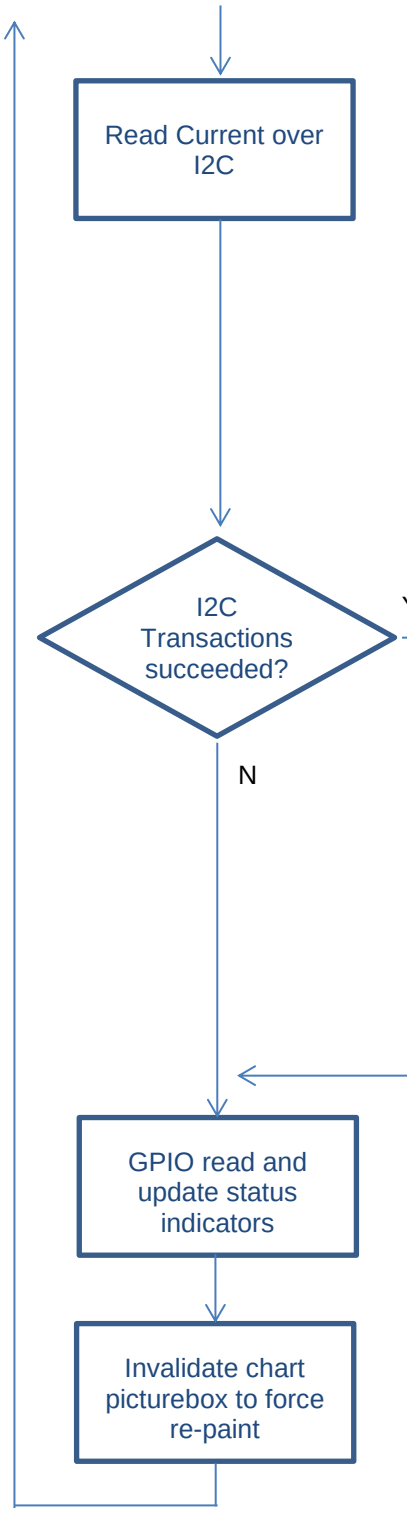
A GPIO operation turns the status LED on or off depending on a counter value, causing the LED to blink. A counter is used as blinking off at every sample would be too fast to be visible.

A voltage reading is taken from the INA219. A write is carried out to the INA219 (address 0x40) to select the voltage result register (register index 2). Then a repeated start is sent followed by a 2-byte read transaction to address 0x40. This is summarized below (error checking omitted for clarity, see the source code provided for full details)

```
AppStatus = I2C_SetStart()
AppStatus = I2C_SendDeviceAddrAndCheckACK(&H40, False)
AppStatus = I2C_SendByteAndCheckACK(&H2)
AppStatus = I2C_SetStart()
AppStatus = I2C_Read2BytesWithAddr(&H40)
AppStatus = I2C_SetStop()
```

The two bytes read are then masked to leave only the 12 result bits, giving 4096 steps from 0-16V and so 4mV per step. The variable *voltage* now contains the value in units of Volts.

Copyright © Future Technology Devices International Limited



This process is repeated for the Current. A write is carried out to the INA219 to select the current shunt voltage result register (register index 1 in the INA219's register map). Then a repeated start is sent followed by a 2-byte read transaction, as summarized below.

```
AppStatus = I2C_SetStart()
AppStatus = I2C_SendDeviceAddrAndCheckACK(&H40, False)
AppStatus = I2C_SendByteAndCheckACK(&H1)
AppStatus = I2C_SetStart()
AppStatus = I2C_Read2BytesWithAddr(&H40)
AppStatus = I2C_SetStop()
```

The current in mA is then calculated from the values read. The code checks the sign bits in the INA219 result to determine the direction of the current. Whilst the INA219 can measure bi-directional current, the sample application will only chart a positive current and so if the current is flowing in reverse, it is displayed as 0mA. The application could be easily extended to measure bi-directional current by means of a larger chart or colour coding of the chart line to indicate direction.

If the I^2C transactions to read the results all succeeded (and therefore the results are valid)

The numerical readouts are updated with the current and voltage values.

The current value is then added to the next location in a circular buffer *CurrentArray*. This array contains a rolling set of the latest 500 values (corresponding to the chart width of 500). The variable *GraphInputPointer* is incremented each time to point to the location where the next result should be added. This array will be used in the chart plotting function.

A GPIO read is carried out to check the values of bits AD4 and AD5 and the radio button indicators on the screen are updated.

The PictureBox which contains the graph is then invalidated so that it will be re-drawn. See the PictureBox2_Paint handler below.

4.4 Button_Stop_Click

When the user clicks the Stop button, the LoggingEnabled variable is set to False so that the while loop taking the measurements will exit after the next complete measurement.

Copyright © Future Technology Devices International Limited

4.5 PictureBox2_Paint

This handler is called whenever Windows re-paints the picture box containing the chart e.g. loading of the form or when the application itself triggers a re-paint to update the chart after the addition of a new data point.

The entire chart is re-drawn including the scale values such that any change in the scale selected by the user is applied immediately.

After plotting the scales and tick marks etc., the actual chart line is drawn. Chart data is taken directly from the same array / circular buffer used to store the 500 data points by the routine which reads from the INA219.

The plotting begins at x = 0 with the oldest data point (i.e. one location above the last point added to the result array, or *GraphInputPointer* + *1*) and continues until x = 499 (which will contain the latest data point measured). The effect of this is that each time the chart is updated, the oldest value will be at the left hand side and the newest value will be at the right-hand side. The chart therefore appears to scroll left as new points are added.

The data is scaled depending on the range selected by the user so that the selected range will fit entirely in the 400 pixels height of the chart.

If the value exceeds the range set, the y value is truncated at 399 and the corresponding pen colour is set to red to indicate the exceedance. The numerical readout will continue to read the correct value up to 1600mA even if the chart range is set for 200mA etc.

Since the coordinates of the picture box are taken from the top down, each Y coordinate is adjusted by Y = (400 – Y) so that they are now targeted to the bottom of the window with positive Y in the up direction.

The chart is then created by drawing a series of lines between pairs of points to create a single chart line.

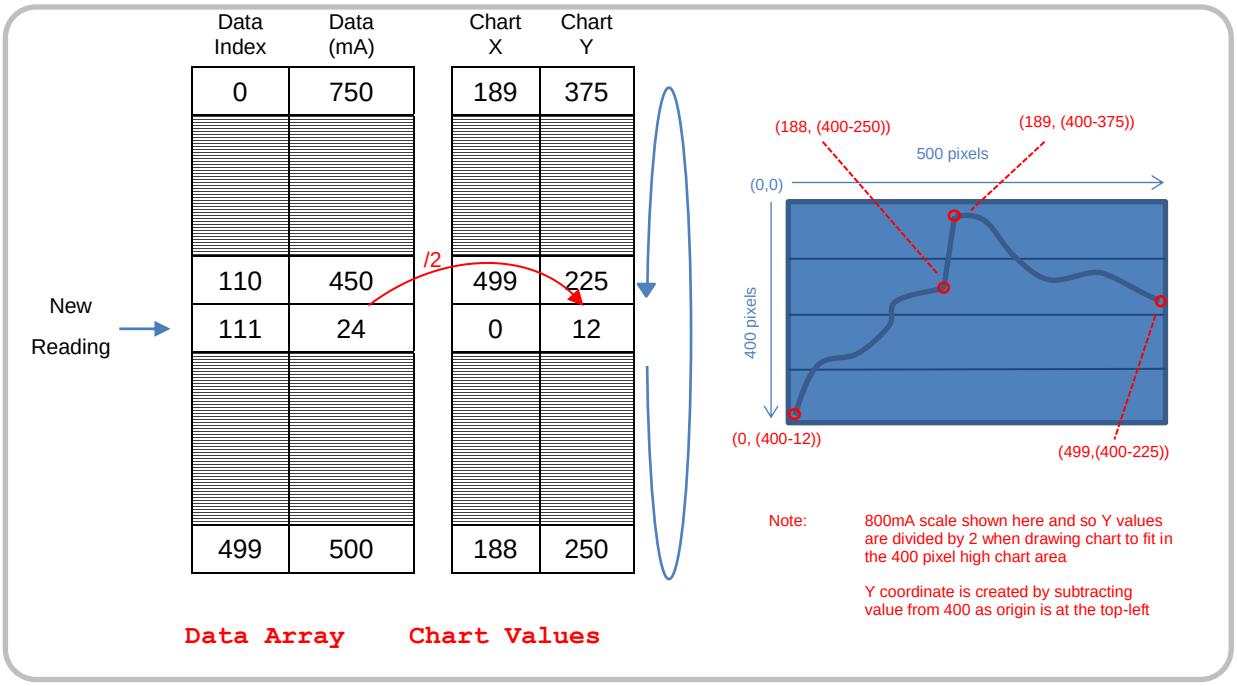


Figure 6 – Chart values

Copyright © Future Technology Devices International Limited

4.6 Button_Save_Click

The Save button can be used when the logging is stopped to save a bitmap image of the chart area. Since the chart is created on a picture box, the DrawToBitmap function in Visual Basic can be used to convert this to a bitmap. A save dialog is displayed allowing the user to select the location to save the file. Some additional error checking and user options etc. may be beneficial in this part of the application.

Figure 7 – Chart saved as bitmap

4.7 Button_Exit_Click

When the Exit button is pressed, a GPIO write is carried out to turn off the status LED and ensure the I^2C lines are in the idle-between-transactions state (both released). An FT_Close then closes the port of the FT232H.

Copyright © Future Technology Devices International Limited

5 I²C Functions

The I²C functions are contained within the file FT232H_I2C.vb. They provide the main application with a set of commands for the familiar I²C operations and create the required buffer of MPSSE commands which are sent to the chip in order to implement them on the I²C lines. By doing so, they avoid the main application from needing to know about the specifics of the MPSSE.

The following functions are provided within the example application:

- I2C_ConfigureMpsse
- I2C_SendByteAndCheckACK
- I2C_SendDeviceAddrAndCheckACK
- I2C_ReadByte
- I2C_Read2BytesWithAddr
- I2C_SetStart
- I2C_SetStop
- I2C_SetLineStatesIdle
- I2C_GetGPIOValuesLow
- I2C_SetGPIOValuesHigh
- I2C_GetGPIOValuesHigh
- These are supported by the functions Receive_Data, Send_Data and FlushBuffer

Note that these functions are intended as a starting point for development of an application rather than a fixed library. They may need to be changed or additional ones created to suit the specific application and intended I²C Slave devices.

These functions use return code 0 to indicate success and a return code of 1 if an operation inside the function call failed. This could be extended to provide additional return codes if it is required to inform the calling function of the reason for failure.

Note: This section uses the terminology I²C transaction to represent the time between I²C Start and I²C Stop i.e. the bus is busy.

Idle-within-transaction	SCL held low and SDA released/floating
Idle-outwith-transaction	SCL and SDA lines are both released/floating

5.1 MPSSE Commands

The application uses a combination of D2xx calls and MPSSE commands to configure the device and then to implement I²C communications.

- The D2xx calls are commands direct to the chip hardware and/or driver. Examples include FT_SetFlowControl, FT_SetBitMode, FT_Read and FT_Write.

  Please refer to the D2XX Programmer's Guide for more information on the D2XX functions.

- The MPSSE commands are built up into a buffer/array by the application and sent to the chip via an FT_Write. The device must be put into MPSSE mode beforehand. The MPSSE engine in the device will then parse and execute these commands in the same sequence. Sending a buffer containing a series of GPIO commands and data clocking commands allows customized data transfers to be achieved and makes the MPSSE very flexible.

  Application note AN_108 details the command set of the MPSSE and AN_135 contains further information on the MPSSE.

Copyright © Future Technology Devices International Limited

5.2 Example Usage of the I2C Functions

The basic I²C demo application provided with this application note provides a simple example of calling the library functions. Please refer to the code provided in the project. The code illustrates the following topics.

- Opening the port and handling an exception in a user-friendly manner if the driver was not installed
- Configuring the MPSSE and setting the GPIO lines to all input as an initial state
- Configure the INA219 via I²C writes to its control register
- Reading the values from the INA219.
    - Use an I²C write to send the register address and then
    - Use an I²C read of two bytes to retrieve the data.
- Read the ADBUS GPIO and display AD6 to AD3 on radio button indicators (AD7 is for LED)
- Read the ACBUS GPIO and display AC7 to AC0 on radio button indicators
- Closing the device

Refer to the file iChart.vb within the code project for the main application code. Each section is highlighted by a heading as shown below:

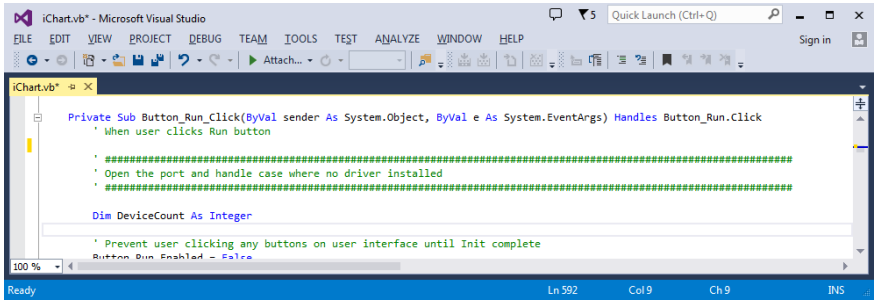


Figure 8 – Basic example code

The code runs through the steps mentioned above after the user clicks the Run button. Each step has a 2 second delay to allow the steps to be visualised as the program runs. The Measurement status entry informs the user of the current step in progress.

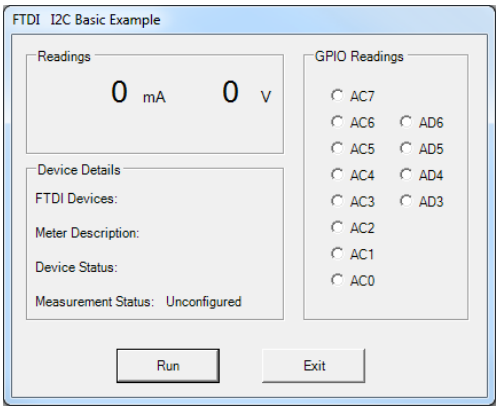


Figure 9 – Basic example application

Copyright © Future Technology Devices International Limited

5.3 Function Descriptions

5.3.1 I²C_ConfigureMpsse

Configures the MPSSE in the device for operation as an I2C Master

Public Function I2C_ConfigureMpsse() As Byte		
IN	Global	FT_Handle of open device channel
IN	Global	ClockDivisor value
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Return	Byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) before proceeding to make any further calls to this I2C library.

This function uses a combination of device API calls (e.g. FT_SetLatencyTimer) and MPSSE commands sent via an FT_Write (e.g. setting MPSSE clock divider).

It requires that the device channel is already open with a valid handle. It then uses the SetBitMode command to enter MPSSE mode. The flow control is set to RTS_CTS mode to ensure that the driver uses flow control.

The latency timer is left at 16ms as the I²C functions use the MPSSE's send immediate command to get bytes back to the PC quickly when required. This is preferable as setting low latency timer values increases the USB traffic (since it sends a packet back at the interval specified even if no data is being transferred) whereas Send Immediate will only do so when required.

After purging the read buffer to ensure it is empty, the function checks that the device is correctly in MPPSE mode by sending a bad command. This is an invalid command 0xAA which is not part of the MPSSE command set. The MPSSE should respond with two bytes, 0xFA followed by the invalid command 0xAA which is received. Reading these back confirms that the device is correctly in MPSSE mode. This is then repeated with 0xAB.

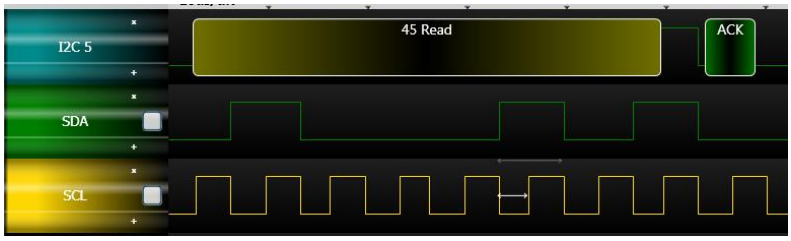
The FT232H has a drive-only-zero feature which can be enabled individually on any of the 16 ACBUS and ADBUS pins. This is effectively an open-drain mode for the selected pins and is again ideal for I²C where the lines are pulled down for logic 0 but released (pulled up by external resistors rather than driven high) for logic 1, thereby allowing many devices to share the same clock and data lines.

Note: The open-drain feature is only available for the FT232H but not for the FT2232H or FT4232H.

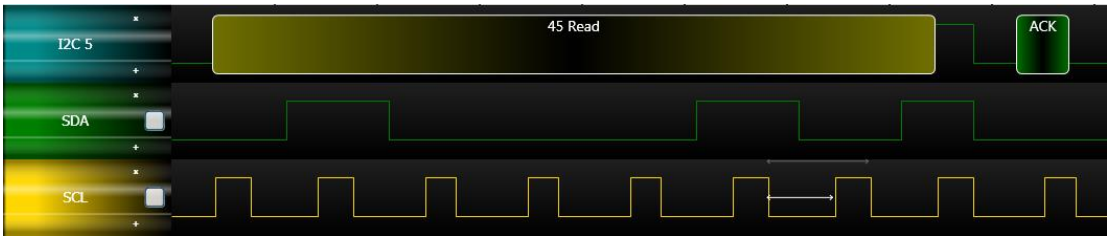
The 3-phase clock mode is enabled which gives three clock phases. This is necessary as the I²C protocol requires data to be valid on both clock edges. The comparison of two vs three phase clocking is shown below.

- There is now both a rising and a falling edge whilst the data is stable, as required by the I²C protocol.
- The three-phase clocking extends each clock cycle by a half-cycle and so each cycle will now be 50% longer. Therefore the frequency (for a given clock divider value) in three phase mode is 2/3rdof the value than the same waveform in two-phase mode.
- The clock duty cycle is now 33.3% rather than 50%.

Copyright © Future Technology Devices International Limited



Two-phase clocking: Clock Period: 6.5us, Clock Frequency: 153.846 KHz, Duty Cycle 50%



Three-phase clocking: Clock Period: 10us, Clock Frequency: 100 KHz, Duty Cycle 33.3%

Figure 10 – Three-Phase clocking

The clock divider is set to give the required I^2C clock rate, which is created by dividing down the 60MHz clock which is supplied internally to the MPSSE. The divider calculation below is for a 400 KHz I^2C rate. 600 KHz is used as the basis due to the actual rate being 1/3rd lower in three-phase clock mode.

MPSSE Clock Source = 60MHz

$$Clock\ Rate = \frac{60MHz}{(1 + CLOCKDIVIDER) * 2}$$

$$600{,}000 = \frac{60{,}000{,}000}{(1 + CLOCKDIVIDER) * 2}$$

$$600{,}000 * (1 + CLOCKDIVIDER) * 2 = 60{,}000{,}000$$

$$1 + CLOCKDIVIDER = 50$$

$$CLOCKDIVIDER = 49 \qquad (0x31)$$

An optimised equation for three phase clock mode is:

$$Clock\ Divisor = \frac{30}{(Clock\ Rate * 1.5)} - 1$$

Assuming Clock Rate in MHz, three-phase clocking enabled, and MPSSE divide-by-5 option disabled

Note: The Clock Divisor is always an integer and so some rates may need to be rounded to the nearest available value.

The I2C_SetLineStatesIdle function is then called to set the I^2C lines to their idle states and will also apply the GPIO values to ADBUS required by the application (which can be set in global variables ADbusVal and ADBusDir before calling the I2C_ConfigureMPSSE function. A call could also be added here to set the high byte GPIO if desired.

Copyright © Future Technology Devices International Limited

The function returns a status value where 0 indicates success. It would be possible to extend this to return additional error codes in the event that the function failed to allow the calling code to identify the cause.

5.3.2 I²C_SendByteAndCheckACK

Writes a byte to the I2C bus

Public Function I2C_SendByteAndCheckACK(ByRef DataSend As Byte) As Byte		
IN	Passed in	Data byte to be sent
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Global	I2C_Ack is a boolean value (True if the I²C device ACKed the byte)
OUT	Return	Byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) and then check if I2C_Ack is True to determine if the I²C peripheral ACKed the byte written.

This function will clock out one byte to the I²C bus, to write a value to the attached I²C peripheral. Before calling this function, the I²C device should be addressed via I2C_SendDeviceAddrAndCheckACK.

The first command clocks out one byte MSB first, which is the data byte to be sent.

A GPIO command is then added to the buffer. This is to ensure that the SDA line is always released after the write regardless of the last data value. The value written combines the normal state of the I²C lines (AD2:0) when within a transaction (SCL low, SDA released) with the GPIO values of AD7:3. The GPIO lines are therefore also updated to the current values of their global variables. This command adds negligible delay but can be removed if the application does not require SDA to be at a particular state when idle within a transaction or can be changed so that SDA stays pulled low between transfers during a transaction.

The third command clocks in one bit which is the ACK bit from the peripheral.

Finally, a Send Result Back Immediately command (0x87) is added which will cause the value clocked in from the I²C peripheral (the ACK bit) to be sent back to the host PC as quickly as possible.

The buffers of commands are sent to the FT232H by calling the Send_Data function. The Receive_Data function is used to read a single byte which has come back from the MPSSE and contains the ACK/NAK bit. The function checks this value and sets the I2C_Ack global variable to reflect the state (True means the peripheral ACKed).

The function returns 0 if all calls for the writing of the commands and reading of the ACK bit succeeded. It returns 1 if any of these failed. The code could be extended to return a wider range of error codes if desired.

Copyright © Future Technology Devices International Limited

5.3.3 I²C_SendDeviceAddrAndCheckACK

Sends an I2C address on the bus

Public Function I2C_SendDeviceAddrAndCheckACK(ByRef Address As Byte, ByRef Read As Boolean) As Byte		
IN	Passed In	Address – I²C address of the device to be communicated with
IN	Passed In	Read – Boolean value specifying read (true) or write (false)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
OUT	Global	I2C_Ack is a boolean value (True if the I2C device ACKed the address)
OUT	Return	Byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) and then check if I2C_Ack is True to check if the I²C peripheral ACKed the address written.

This function is very similar to the I2C_SendByteAndCheckACK call discussed above but is specifically intended for addressing the I²C device.

Instead of taking the byte to write as a parameter, this modified function takes two parameters; a 7-bit I²C address (in the lower 7 bits) and a separate Boolean value defining whether to address the device for reading or writing.

Note that documentation for I²C peripheral devices may quote the I²C address in either the 7-bit or 8-bit formats. E.g. some documentation and sample code may quote 7-bit address 0x40 whilst others may quote the write and read values as 0x80 and 0x81 respectively. This code is designed to accept the 7-bit value with the R/W bit specified separately.

It combines these into a single 8-bit value by shifting the 7-bit address one place left and OR'ing with the Read Boolean parameter, and sends this to the I²C bus. As with the I2C_SendByteAndCheckACK call, this function returns the ACK/NAK status via the global variable I2C_Ack.

5.3.4 I²C_ReadByte

Reads a single byte from the I2C bus

Public Function I2C_ReadByte(ByRef ACK As Boolean) As Byte		
IN	Passed In	ACK – Boolean value, sends ACK if True, NAK if False
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Global	InputBuffer(0) has received byte
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) and then read the byte from InputBuffer(0). Read the byte before calling the next I²C function to avoid overwriting.

This function will clock in one byte from the I²C bus, to read the value from a register in the attached I²C peripheral. Before calling this function, the I²C device should be addressed and (if required) have the register selected which is to be read. See section 5.2 and the associated basic example for details.

The first command clocks in one byte MSB first. This is the data byte being read from the I²C peripheral.

The second command clocks out one bit which forms the ACK bit. The SDA value clocked out for the ACK bit can be configured by the application when calling the function. True will result in an ACK (SDA pulled low) whereas False will result in a NAK (SDA left pulled high).

A GPIO command is then added to the buffer. This is to ensure that the SDA line is always released in between bytes regardless of the last data value. The value written combines the normal state of the I²C lines (AD2:0) when within a transaction (SCL low, SDA released) with the GPIO values of AD7:3. The GPIO lines are therefore also updated to the values of their global variables. This command adds negligible delay but can be removed if the application does not require SDA to be at a particular state

Copyright © Future Technology Devices International Limited

when idle within a transaction or can be changed so that SDA stays pulled low between transfers during a transaction.

Finally, a Send Result Back Immediately command 0x87 is added which will cause the byte clocked in from the I²C peripheral to be sent back to the host PC immediately.

The buffer is sent to the FT232H via an FT_Write command by calling the Send_Data function. The Receive_Data function is used to read the single byte which will be clocked in by the MPSSE from the I²C peripheral.

The function returns 0 if all calls for the writing of the commands and reading of the data byte succeeded. It returns 1 if any of these failed. The code could be extended to return a wider range of error codes if desired.

5.3.5 I²C_Read2BytesWithAddr

Addresses the device and reads 2 bytes

Public Function I2C_Read2BytesWithAddr(ByRef Address As Byte) As Byte		
IN	Passed In	Address – I²C address of the device to be communicated with
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Global	I2C_Ack indicates if the address phase was ACKed
OUT	Global	InputBuffer(0) and (1) have received bytes
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) and then check that I2C_Ack is True. Then, read the bytes from InputBuffer(0) and InputBuffer(1). Read the bytes before calling the next I²C function to avoid overwriting.

This function combines the commands from I2C_SendDeviceAddrAndCheckACK and I2C_ReadByte to form a single call which addresses the peripheral for reading, and reads two bytes. It ACKs the first byte read and NAKs the second.

Many I²C peripherals (including the INA219) have 16-bit registers which can be read in a burst. The Master reads the first byte from the peripheral, sending an ACK in response. It then reads the second byte responding with a NACK. The NACK tells the peripheral that the Master does not wish to read further bytes in this transaction.

Whilst this could be accomplished with three separate calls (an address call and two read calls), combining the equivalent operations into a single sequence of MPSSE commands can improve speed as the commands all get sent on a single USB micro frame and the actual operations on the I²C bus will have no gaps between. The actual saving depends on the scheduling of the USB host controller and also whether the USB micro frame rate (125us) is significant compared to the I²C clock rate.

This function is provided as an example of the ways in which a series of operations can be combined. This can have some dependency on the I²C peripheral being used. Since this hybrid function will not return the ACK or the two bytes read until fully complete, it is not possible for the application to check the ACK state of the addressing *before* reading. The calling code can however check the ACK bit after completion of the call and therefore determine whether the two bytes read are valid or not. If the continuation to generating the read cycle is not allowable by the particular I²C peripheral, then this combined function may not be well suited.

In this case, the INA219 also has a timeout if the I²C bus is idle for too long in the middle of a transaction (when the bus isn't fully idle and released) and so grouping the calls is beneficial as it ensures that there won't be gaps between them which may cause a timeout if the USB bus gets very busy for a short time. Gaps between writes is not an issue as the INA219 would NAK the following write attempt but gaps between reads can't be detected as the Master provides the ACK in a read operation. In this case, the I²C start and stop could even be added to the same buffer of commands.

Copyright © Future Technology Devices International Limited

The reader may find other opportunities to group commands in a way that optimises the communication for their I²C peripherals.

5.3.6 I²C_SetStart

Sends the I2C Start condition

Public Function I2C_SetStart() As Byte		
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success)

This function will put the I²C Start condition onto the bus to begin an I²C transaction. The function can also be used as a repeat start which is used in transactions which read data from an I²C peripheral, in between writing to the device to select the register to be read and the actual reading operation.

The function builds a buffer of GPIO commands for the MPSSE, which it will work through in sequence. Each GPIO write is repeated six times to hold the associated pin state for a longer time.

Both SCK and SDA are initially high (open drain pulled up). The SDA line is brought low by putting ADbus1 low. Then, the SCL line is also brought low (ADbus0) to complete the sequence. The line is then left in the idle-during-transaction state which is SDA released and SCL held low. This could be easily changed if required by the applications.

5.3.7 I²C_SetStop

Sends the I2C Stop condition

Public Function I2C_SetStop() As Byte		
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success)

This function will put the I²C Stop condition onto the bus. This ends the I²C transaction on the bus.

The function builds a buffer of GPIO commands for the MPSSE which it will work through in sequence. Each GPIO write is repeated six times to hold the associated pin state for a longer time.

The SCL line is brought high by putting ADbus1 high (tri-state). Then, the SDA line is also brought high (ADbus0) to complete the sequence.

5.3.8 I²C_SetLineStatesIdle

Sets the I2C lines to Idle Outwith Transaction and write ADBUS GPIO

Public Function I2C_SetLineStatesIdle() As Byte		
IN	Global	GPIO_Low_Dir – direction to set for AD7:3 (1 = out, 0 = in)
IN	Global	GPIO_Low_Dat – data values to write to AD7:3
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success). Do not call in the middle of an I²C transaction (i.e. between Start and Stop). This function is only for setting GPIO when no transaction is in progress.

Copyright © Future Technology Devices International Limited

The global variables containing the data and direction for AD7:3 are combined with the idle-outwith-transaction I²C pin states for ADbus2:0 (SCL and SDA both released) and this value is written via the 0x80 GPIO low commands.

Since the use of this function would result in the SCL line being released, it is only intended for use when an I²C transaction isn't in progress.

Note: The other I²C library calls for addressing, reading and writing will update the GPIO lines on ADBUS as part of their operation and so if a GPIO line on ADBUS requires to be changed during a transition, the associated global variables GPIO_Low_Dir and GPIO_Low_Dat can be written before the next I²C operation.

5.3.9 I²C_GetGPIOValuesLow

Gets the GPIO values of ADbus 7:3

Public Function I2C_GetGPIOValuesLow() As Byte		
OUT	Global	InputBuffer(0) has GPIO values in bits 7:3
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that the function returns success. Then read the GPIO value from InputBuffer(0). The SetLineStatesIdle function must have been called at least once before in order to set the directions of the GPIO.

This function sends a GPIO low byte read command to the MPSSE along with a send-immediate command which causes the result to be returned immediately. The resulting value is masked to zero the lower three bits, leaving the upper five bits reflecting the GPIO value. The value can be read in bits 7:3 of the resulting byte via InputBuffer(0) after the function has been called.

5.3.10 I²C_SetGPIOValuesHigh

Sets the data and direction of ACBUS bits 7 to 0

Public Function I2C_SetGPIOValuesHigh(ByRef ACbusDir As Byte, ByRef ACbusVal As Byte) As Byte		
IN	Passed in	ACbusDir contains the directions for the pins (1 = output)
IN	Passed in	ACbusVal contains the values to write to the pins which are outputs
OUT	Return	Byte value containing status code (0 indicates success)
Notes		Check that the function returns success.

This function sends the 0x82 Write GPIO High Byte command followed by the value and direction bytes. This allows control of bits 7 to 0 of the ACBUS port. The values in parameter ACbusVal will be applied to pins which are configured as an output in the ACbusDir parameter (1 = output).

Note that the MPSSE can control and read only bits 0-7 of the ACBUS port. Bits 8 and 9 are configurable in the EEPROM for other functions such as PWREN etc.

This function can be called any time after the MPSSE is initialised and does not affect the I²C lines.

5.3.11 I²C_GetGPIOValuesHigh

Read the values of ACBUS bits 7 to 0.

Public Function I2C_GetGPIOValuesHigh() As Byte		
OUT	Global	InputBuffer(0) has GPIO values for this port
OUT	Return	Byte value containing status code (0 indicates success)
Notes		Check that the function returns success.

Copyright © Future Technology Devices International Limited

This function sends the 0x83 Read GPIO High Byte command followed by a send-immediate opcode which causes the MPSSE to send the resulting byte back on the next available micro frame.

The return value should be checked to be 0 (success) and then the byte containing the pin values can be read via InputBuffer(0).

It is recommended to call I2C_SetGPIOValuesHigh at least once after initialisation before using the GetGPIOValuesHigh. This ensures that the ACBUS pin directions are set as required for the application.

5.3.12 Sending and Receiving Data via D2xx

Since effective use of the FT_Write and FT_Read D2xx calls involves some error checking and additional steps (such as checking the queue status before reading), the I^2C library source provided here wraps these calls into functions Send_Data and Receive_Data. An additional call is used during initialisation to flush any data in the driver's buffer.

5.3.13 Send_Data

Sends the requested number of bytes to the FT232H.

Private Function Send_Data(ByVal BytesToSend As Integer) As Byte		
IN	Global	FT_Handle of open device channel
IN	Passed In	BytesToSend – number of bytes to send
IN	Global	Data to send in SendBuffer(0) to SendBuffer(BytesToSend-1)
OUT	Return	Byte value containing status code (0 indicates success)
OUT	Global	BytesSent – number of bytes actually sent
Notes		Check that function returns 0 (success) which means that all bytes were sent. If return is non-zero, bytes actually sent can be found in BytesSent

This function uses the FT_Write D2xx call to send the data. The USB host in the PC will determine how and when the data is sent but from the application's point of view the data will be sent as quickly as possible. The call blocks until complete and so setting a timeout with FT_SetTimeouts (e.g. 5 seconds as this is only a safety measure) is strongly recommended.

5.3.14 Receive_Data

Reads the requested number of bytes

Private Function Receive_Data(ByVal BytesToRead As Integer) As Byte		
IN	Global	FT_Handle of open device channel
IN	Passed In	BytesToRead – number of bytes to read
OUT	Return	byte value containing status code (0 indicates success)
OUT	Global	BytesRead – number of bytes actually read
OUT	Global	Data read in ReceiveBuffer(0) to ReceiveBuffer(BytesRead-1)
Notes		Check that function returns 0 (success) which means that requested number of bytes were read. If return is non-zero, bytes actually read can be found in BytesRead

This function reads the data from the driver buffer after using FT_GetQueueStatus to check how much data is available. When the MPSSE clocks in data or reads a byte from the GPIO of the chip, it will buffer the data on-chip. This data will be sent back to the PC when the buffer has enough data for a USB frame (510 bytes of data) or if the latency timer ticks over, or if a Send Immediate command is executed by the MPSSE. This library uses the send immediate opcode after any operation involving clock in or GPIO read and so this will be the normal mechanism used in this case. The driver issues IN requests over USB to the device and the chip will put the data into the IN packet when one of the aforementioned conditions occurs. The driver will buffer up this data and make it available for the application to read.

The Queue Status allows the application to check how much is currently buffered. This is especially useful as the FT_Read is a blocking call and so it is best to read only data that is known to be available and so in

Copyright © Future Technology Devices International Limited

theory a timeout should never occur. It is strongly recommended to set a read timeout (e.g. 5 seconds) via FT_SetTimeouts as a safety measure however.

This function works by running a loop which calls GetQueueStatus and if >0 bytes available, it reads these and appends to a buffer/array within the function. This process continues until the expected number of bytes (as passed in) has been received, or until the loop has run for a certain number of cycles (acting as a software timeout).

5.3.15 FlushBuffer

Reads any bytes in the receive buffer of the driver to clear it out

Private Function FlushBuffer() As Byte		
IN	Global	FT_Handle of open device channel
OUT	Return	byte value containing status code (0 indicates success)
Notes		Check that function returns 0 (success) which means that all data was flushed successfully (or no data was there to be flushed)

This function is normally only used when initializing the device for I^2C. It checks the Queue Status of the driver's receive buffer and reads any data in the buffer in order to clear the buffer.

Copyright © Future Technology Devices International Limited

6 Further Development

The sample applications provided with this application note, including the I²C functions, are intended as the basis for further development and customisation to suit the intended application.

In addition to using the code to interface with other types of sensor, some possible areas for further development include:

6.1 Other Languages

This application note uses the general MPSSE command set and so can be ported to any platform which supports D2xx drivers for the FT232H. The MPSSE commands would remain the same but the method of sending and receiving data (i.e. equivalent syntax for FT_Write and FT_Read) would require to be ported to the equivalent calls on the new platform. The Graphical interface will also require porting to the new platform/language. The use of a separate thread for reading of the data may also be used but is outwith the scope of this example.

6.2 Adding support for FT2232H and FT4232H devices

The code supplied is intended for the FT232H. Additional GPIO writes are required in order to use the code on FT2232H and FT4232H as these do not support open-drain functionality on the I²C pins. FTDI application notes AN_411 and AN_113 have examples of this.

6.3 Clock Stretching

The MPSSE does not support clock stretching. The MPSSE has an adaptive clocking feature but this was not specifically designed for I²C and does not provide full clock stretching functionality. For this reason, it is not recommended to use adaptive clocking to implement clock stretching and this is not guaranteed or supported by FTDI.

One solution may be to reduce the SCL rate. Some peripherals do not use their clock stretch capability at lower I²C clock rates. It is recommended to test all functions of the peripheral to ensure that it does not require clock stretching when running at this lower rate. It is also recommended to consult the datasheet or manufacturer to confirm this. The clock divisor is a parameter of the I2C_Configure_MPSSE function call. See section 5.3.1 for details.

When clock stretching is required by the attached peripheral, it is strongly recommended to use the new I²C bridging devices from FTDI including the FT4222H and FT260 which have support for clock stretching functionality. For details of these devices please refer to the links below:

- FT260 Product Page
- FT4222H Product Page

6.4 Hardware

The hardware could be enhanced by the addition of an I²C isolation chip between the UM232H and the INA219, so that the measurement side is completely isolated electrically from the PC used to control and monitor the meter.

Since the interface can address more than one I²C peripheral, a multi-channel monitoring system could be created with several INA219 devices to monitor different rails of a power supply etc.

Copyright © Future Technology Devices International Limited

7 Using the Meter

This section summarises the steps required to use the example provided with a Windows PC.

1. If an FTDI driver is not currently installed, install the latest driver.

    a. Download the executable installer to the PC (onto the desktop etc.). This can be downloaded from the comments column of the Currently Supported Drivers table on the following page: http://www.ftdichip.com/Drivers/D2XX.htm

    b. Right-click and select run-as-administrator

    c. Follow the steps in the installation wizard until finished

    The driver can also be automatically loaded via Windows update when connecting a device if the settings in the OS are configured to allow this.

2. Using a standard USB A to mini-B cable, connect the mini-B port of the UM232H at the rear of the unit to the USB2 port on the PC. (see the black cable in Figure 11)

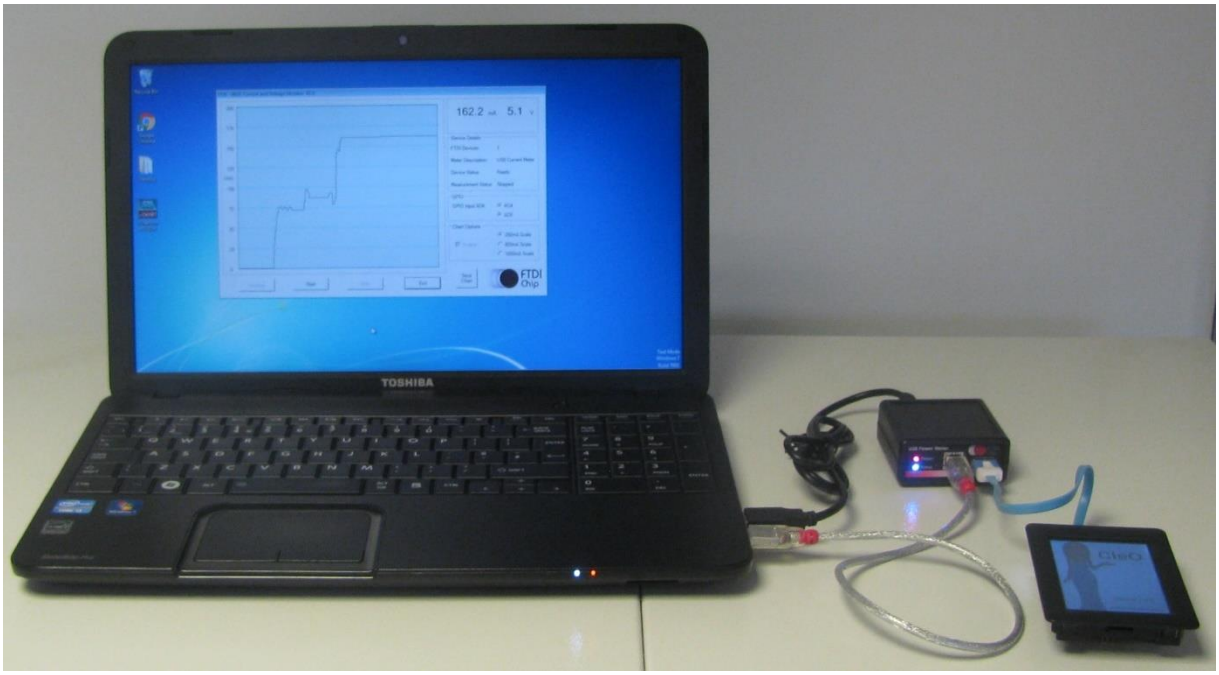


Figure 11 – Connecting the unit to measure the current consumed by a *CleO* board

3. Windows will then complete the driver installation and the device will show up under the Universal Serial Bus Controllers section of the device manager.

4. The red Power LED of the meter will illuminate once the driver installation completes, as the PWREN# signal goes low at this time.

5. Open the sample code by double clicking on the executable file. This can be found in the bin > x86 > Release folder.

    For debugging and modifying the application, the solution (.sln file) can be opened in Visual Studio.

Copyright © Future Technology Devices International Limited

6. Click the Initialise button and the program should find the UM232H and open it and configure the device to MPSSE mode. The blue Status LED will illuminate as the associated GPIO line is set as an output driving low as part of the MPSSE initialisation routine.

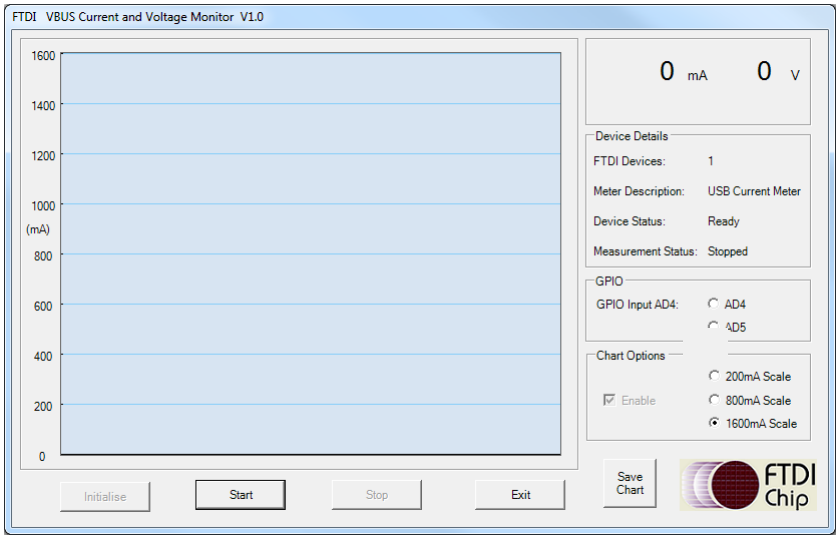


Figure 12 – Application window after opening the program

7. Click the Run button and the blue LED on the front panel of the meter will blink to indicate that the measurements are being taken.

8. Connect a short standard USB A-B cable from the USB port of the host PC of the link being measured. Refer to the silver coloured cable in Figure 11. The Voltage display will now indicate the presence of the USB VBUS voltage which will be approximately 5V.

9. Now connect the peripheral of the link to be measured to the front port of the meter, as shown by the blue cable in Figure 11. In this case, an FTDI *NerO* board with attached *CleO* display is being tested. (see http://www.cleostuff.com/)

   Due to the pass-through configuration of the USB ports on the meter, the peripheral will behave as if it is connected directly to the host PC. The current value and chart will indicate the current flowing in the VBUS connection to the peripheral. See Figure 5 for an example.

10. The range buttons can be used to select a scaling factor for the data, allowing greater detail to be observed at lower currents. Ranges of 200mA, 800mA and 1600mA can be selected.

    If the current measured exceeds the present range, the points are plotted at the top of the chart and coloured red to indicate that they are off-scale.

    Whilst running, the GPIO lines AD4 and AD5 are measured and indicated via the state of their associated radio buttons. These lines can be accessed on the prototype unit via the pin header.

11. To stop the capture, click the 'Stop' button.

12. At this point, the Save Chart button can be used to save a bitmap image of the chart area. Figure 7 shows an example.

The rear panel connector also has pins for GND, Current In and Current Out. These can be used instead of the connections on the front panel. These are ideal for measuring the current of a device on EVB etc. via a jumper inserted in-line with the supply being measured.

Note: Refer to the Readme file in the software package for important notes related to the meter usage.

Copyright © Future Technology Devices International Limited

8 Conclusion

This application note and accompanying example have demonstrated the implementation of a USB-I²C Master interface with the FT232H chip, along with the use of the Visual Basic NET wrapper to control the FT232H from a graphical user interface. It has also provided an example of how to acquire the data and display on a basic scrolling chart user interface. This could be extended to use a wide variety of applications, with the many different I²C sensor types available.

In addition, the demonstration provides a useful tool which can help when developing, debugging and using USB peripherals or other low-voltage DC circuits.

FTDI have a range of FT232H-based modules including UM232H, UM232H-B and C232HM cables. This application will work well with all of these modules as well as with the IC itself if placed on a custom designed PCB. The sample code provided with this application note is designed for the FT232H only.

FTDI have a range of other bridging devices capable of I²C Master Implementation including the FT2232H, FT4232H, FT4222H and FT260. The FT260 is the latest addition to the range of USB to I²C Master Bridge devices and is a HID class which means that it can be used without loading a separate D2xx driver. Further information can be found on these products at the FTDI homepage www.ftdichip.com

Copyright © Future Technology Devices International Limited

9 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 8797 1330
Fax: +886 (0) 2 8751 9737

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

Copyright © Future Technology Devices International Limited

Appendix A - References

Document References

ICs:

FT232H Product Page
FT232H IC Datasheet
INA219 Datasheet (external link)

Hardware:

UM232H Datasheet
C232HM cables
UM232H-B modules

Documents:

AN_108 Command Processor for MPSSE and MCU Host Bus Emulation Modes
AN_255 FT232H I2C example in C++
AN_113 Interfacing FT2232H Hi-Speed Devices To I2C Bus
AN_411 FTx232H MPSSE I2C Example in C#
AN_135 MPSSE Basics
AN_356 AN_356 FT800 Interfacing I²C Sensor to VM800P

Others:

D2XX Drivers
D2XX Programmer's Guide

Source Code:

http://www.ftdichip.com/Support/SoftwareExamples/CodeExamples/VB/AN_355.zip

Acronyms and Abbreviations

Terms	Description
ADC	Analog to Digital Converter
GPIO	General Purpose Input Output
I2C	Inter-IC bus
LED	Light Emitting Diode
MPSSE	Multi-Protocol Synchronous Serial Engine
USB	Universal Serial Bus
VB Net	Visual Basic .NET

Copyright © Future Technology Devices International Limited

Appendix B – List of Tables & Figures

List of Tables

NA

List of Figures

Copyright © Future Technology Devices International Limited

Appendix C – Revision History

Document Title	:	AN_355 FT232H MPSSE Example - I2C Master with Visual Basic
Document Reference No.	:	FT_001138
Clearance No.	:	FTDI#524
Product Page	:	http://www.ftdichip.com/FTProducts.htm
Document Feedback	:	Send Feedback

Revision	Changes	Date
1.0	Initial Release	2017-03-07
1.1	Updated the Clock Stretching section to refer to FT260 and FT4222H for applications needing this feature.	2020-02-12

Copyright © Future Technology Devices International Limited