Application Note

AN_475

FTDI Windows 10 IoT Solutions

Version 1.0

Issue Date: 2016-05-23

This document describes the FTDI Windows 10 IoT demonstration, which uses FTDI USB bridging and EVE display controller devices to provide the entire I/O and graphical user interface solutions for an IoT sensor application.

The Application is targeted at the Raspberry Pi 2 ARM platform. However, due to the flexibility afforded by using the FTDI USB bridging device as the hardware interface, the application can also be used on a wide variety of other platforms.

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

This document describes FTDI's Internet Of Things (IoT) demo, where an FTDI bridging device and FTDI display controller are used in a sensor and display application.

Small embedded computers such as the Raspberry Pi have become increasingly popular in IoT applications, and the introduction of Windows 10 for the Raspberry Pi has further enhanced this.

FTDI's bridging and display solutions allow designers to both provide/extend the I/O capabilities, often a key requirement for IoT devices, and also provide a customized graphical user interface which can be integrated into the product. In the latter case, platforms such as the Raspberry Pi which were originally designed for full size keyboards, mice and HDMI monitors are now being used in compact applications which need an integrated human interface where local control of the device is required.

The FT4232H bridging device provides a versatile means of I/O extension, taking care of the *entire* interfacing requirement in this demonstration including the display, whilst utilizing only a single USB port. The only connections to the Raspberry Pi are the power and the USB connection to the FT4232H bridge. The demo combines the bridging device with an EVE display/touch controller to form a complete and capable I/O and user interface solution from FTDI including the following features:

- I^2C/SPI interfacing to sensors and peripherals
- GPIO lines
- Two additional UART/GPIO ports
- A colour graphics TFT user interface with touch/audio capabilities

The target hardware platform for this project is a Raspberry Pi 2 (32-bit ARM) running the latest version of Windows 10 IoT core. It has the D2XX Universal Windows Driver installed, which is FTDI's new driver for UWP based Windows 10 systems. The Application is developed in Microsoft's Visual Studio IDE and is compatible with the Free Community version of the tool (see Appendix A – References).

The Windows 10 Universal Windows Platform (UWP) was selected for this demo due to its increasing popularity for IoT applications, and because the same source can be built to run across the different Windows 10 platforms. However, the application could also be ported to run on Linux on the Raspberry Pi via the FTDI D2xx driver for Linux, in addition to other OS and platforms which support FTDI's D2xx drivers.

Please refer to the sample code project at:

http://www.ftdichip.com/Support/FTReferenceDesigns.html

1.1 Overview

The basic function of the application is to measure the colour of objects placed in the proximity of the colour sensor. The application detects the presence of an object via the proximity sensor, and reflects this on the screen by increasing the radius of the red indicator circle (shown with no object in proximity in Figure 1). When the proximity reaches a level defined in the software, a GPIO line is used to turn on the white LED to illuminate the object. The colour sensor measures the colour of the object as a set of RGB values and uses these to calculate a hue value. The hue value is displayed in addition to colouring the indicator circle to match the object measured.

An FTDI FT800, on the VM800B module, is used to provide a local display to the Raspberry Pi. The FT800 has a rich feature set and its object oriented design makes it easy to create attractive and intuitive user interfaces. The FT800 also has touch screen and audio features which are readily supported by the VM800B as supplied. These could be used to extend the demo to provide a full display and control solution for embedded processor systems, allowing the application to be run as a self-contained system without any monitor, keyboard, or mouse connected to the Raspberry Pi.

Figure 1 FTDI IoT Demo Hardware

1.2 Scope

This document is intended to help designers of IoT applications to use the FTDI bridging devices to extend the I/O capabilities of their embedded processor board and to use FTDI's EVE display solutions to provide a graphical user interface (which could be extended with touch and audio capabilities). The document uses the Raspberry Pi 2 and Windows 10 as the example platform.

The topics covered by this application note include:

- Using the FTDI Universal Windows Platform D2xx driver on Windows 10
- Using FTDI bridging devices and associated MPSSE functions for SPI and I^2C
- Using the FTDI EVE devices to create a user interface for small embedded computers

2 Hardware

The hardware and source code described in this application note provide a starting point for developing applications to enable USB to SPI and USB to I^2C communications using the FTDI bridging solutions such as the FT4232H. FTDI have several other bridging devices including the FT232H, FT2232H and FTDI C232HM MPSSE cable which could also be used.

The block diagram is shown below and the demo schematic is shown in Figure 2.3.

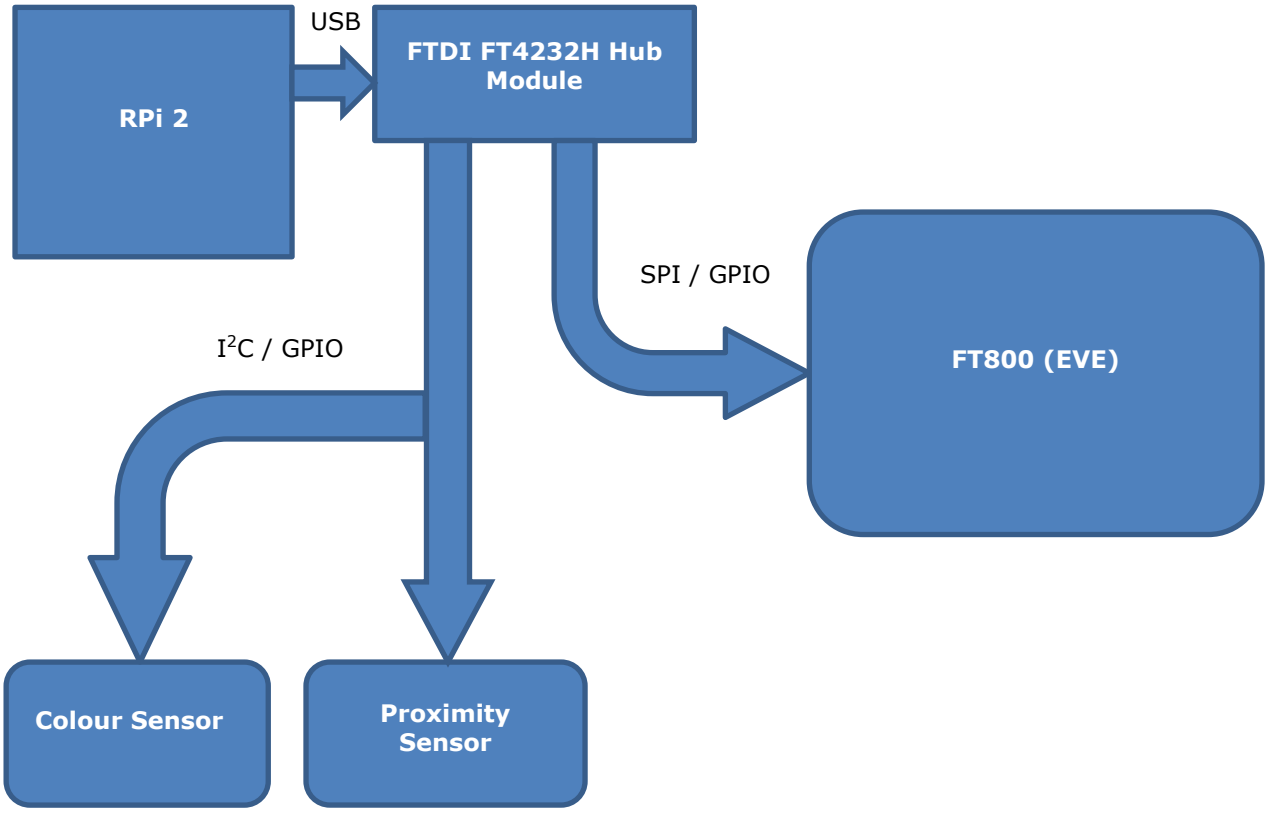

Figure 2.1 Overview of system Elements

FT4232H Hub Module

The FTDI FT4232H Hub Module provides the key I/O interface to the Raspberry Pi. It contains a 4-port USB hub with one port of the hub hosting an FT4232H device and the other ports available to connect to other USB devices. Two channels of the FT4232H are used to provide an I^2C and an SPI interface along with some GPIO lines.

With the configuration used in this demo, the module provides the following interfaces whilst utilizing only one USB port on the Raspberry Pi:

- An SPI Master interface with GPIO on port ADbus via MPSSE (used for the EVE controlled display)
- An I^2C Master interface with GPIO on port BDbus via MPSSE (used for the sensors)
- Two spare UART/Bit-bang ports CDbus and DDbus
- Two spare USB downstream ports on Type-A connectors
- One spare USB downstream port on the pin header

The SPI Master and I^2C Master are provided by the Multi-Protocol Synchronous Serial Engine (MPSSE) interfaces available on ports A and B of the FT4232H. The MPSSE is a versatile data clocking engine which allows a variety of protocols such as I^2C Master, SPI Master and JTAG to be implemented. The sample code provided with this application note includes example functions for implementing the SPI protocol for the FT800 and I^2C protocol for the sensors.

The USB cable between the sensors/display and the computer board also means that they may be located in separate hardware units or that additional I/O can be easily added. For example, in a measurement system, the Raspberry Pi may be located inside the main enclosure of the machine. An FT232H-based module may connect to one USB port of the RPi to provide the interface to an EVE-based display which is on a flexible arm allowing the user to position it conveniently. A second FT232H or FT2232H may be used to connect to the sensors within the main body of the machine with an internal USB connection to a second port on the Raspberry Pi.

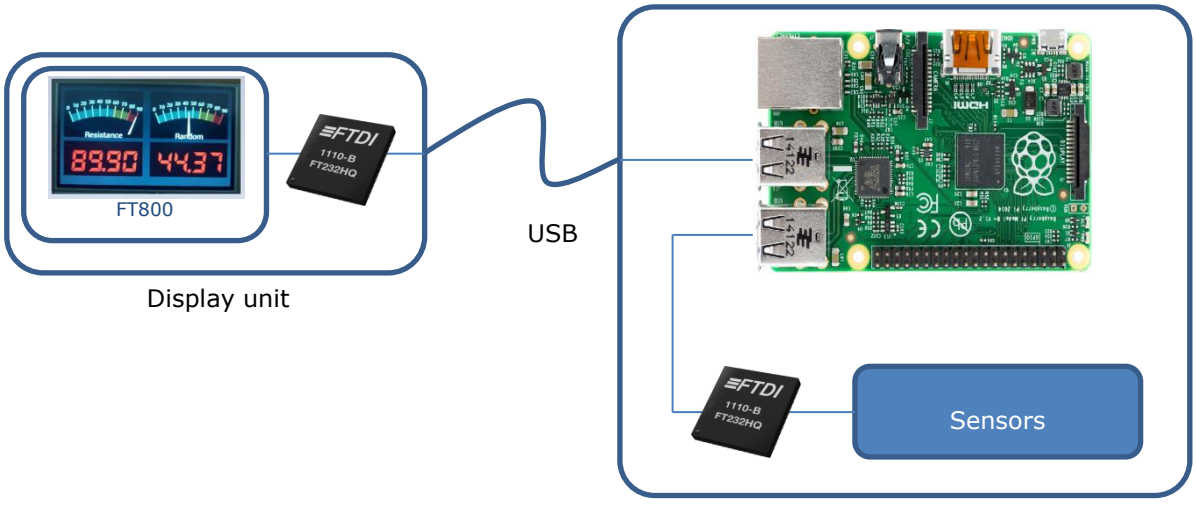

Figure 2.2 Measurement system

VM800B EVE Module

The FTDI VM800B EVE module is controlled via the SPI Master interface. The EVE IC family from FTDI make it easy to create a user interface with a colour LCD panel, touch screen and audio output. The VM800B module allows evaluation of the EVE device features with only a simple connection to a power source and an SPI Master. The module has all of the features needed to control the display, touch and audio over the SPI interface, including backlight driver and amplifier/speaker. The plastic bezel provides an easy and secure way of mounting the display. This demo provides a simple graphical interface but could be extended to include more comprehensive graphics including bitmaps, touch control and audio. More information on the range of EVE devices, evaluation boards and sample code can be found at http://www.ftdichip.com/EVE.htm

Sensors

The I^2C interface has two sensors connected; an Adafruit colour sensor and a MikroE IR proximity sensor. Information on these sensors can be found in Appendix A – References.

The sample code provided has functions for initializing and reading the sensors which can easily be modified to communicate with other I^2C sensors. As MPSSE has separate Data In and Data Out pins, these are linked to form a single SDA data line. The MPSSE command sequences sent by the library function also handle the direction for the Data Out pin so that the SDA line becomes tristate when the slave will be driving the line. This allows the code to be used with the FT2232H and FT4232H which do not have the open-drain mode but have multiple channels which make them well-suited to this application.

The I^2C functions also provide GPIO functionality on the unused pins of the port (bits 3-7). One of these is used to control the white LED used on the RGB colour sensor which illuminates the object being measured. The other lines are available for future extension of the demo.

Power Supply

The demo is powered from a single 5V input. A 3v3 LDO regulator provides a 3v3 rail for the sensors. The Proximity sensor runs from a 3v3 supply whilst the colour sensor runs from a 5V supply to provide additional voltage for its white LED but includes an I^2C level shifter allowing 3v3 operation on the I^2C bus. Alternative sensors with an I^2C or SPI interface could be connected to the FT4232H to suit a variety of measurement applications.

Note that the Raspberry Pi was powered via its GPIO header in the demo but this method of powering bypasses the reverse polarity and overvoltage protection. The Raspberry Pi 2 could instead be powered via its USB micro B connector.

A photo of the complete demo unit is shown in Figure 1 on page 3.

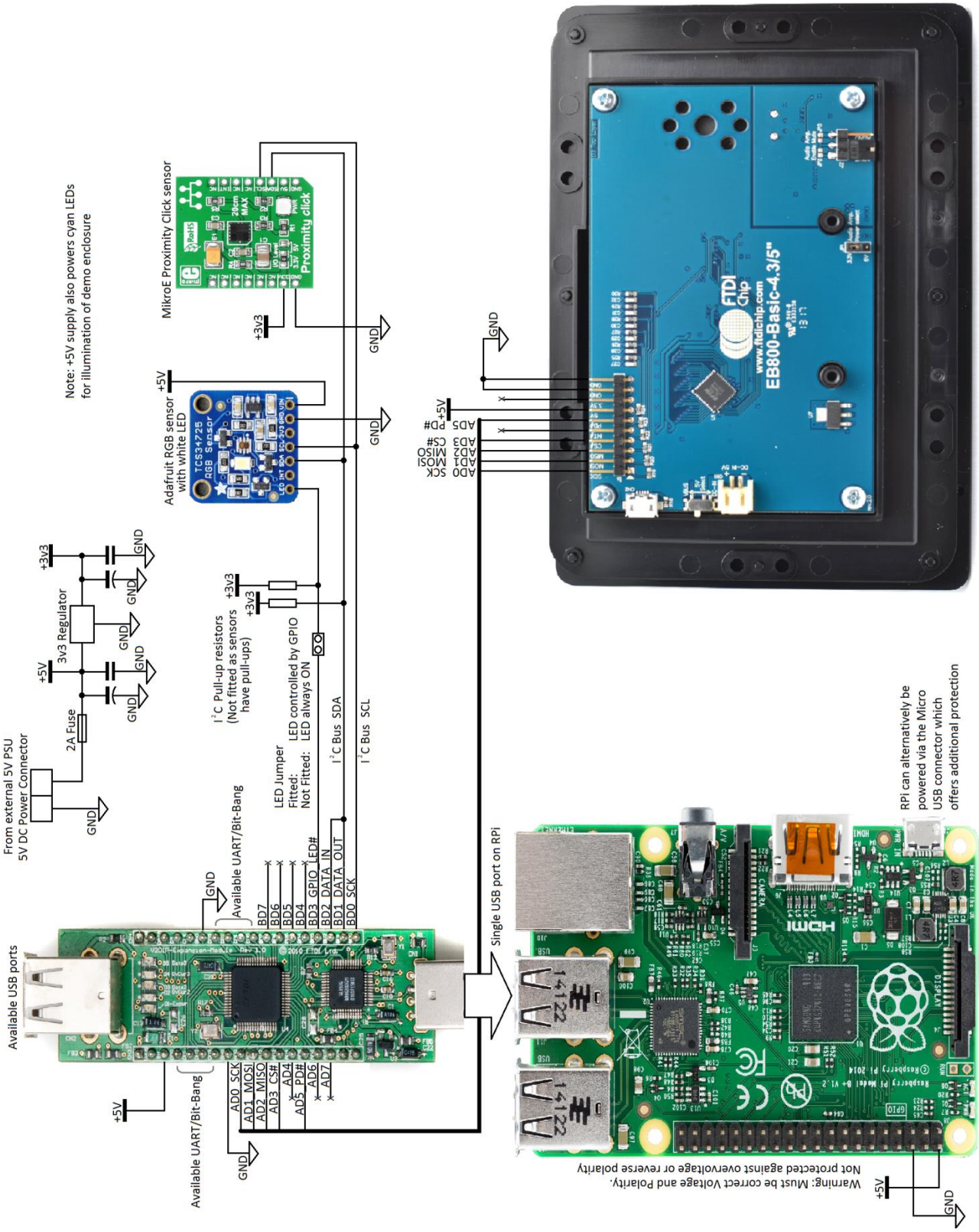

Figure 2.3 Schematic

3 Example Program

The Application is written as a Universal Windows Platform (UWP) architecture using C# and the .Net framework. It is developed in Microsoft's Visual Studio IDE and is compatible with the Free Community version of the tool (see Appendix A – References).

The Application is partitioned into a number of classes. The application is multithreaded and uses the Microsoft Task Parallel Library (TPL). While for development purposes there is a user interface present in the example code, the application is ultimately targeted to be run as a startup application on a headless device.

The Application includes a reference to the FTDI FTD2xx.dll which is included in the example project provided or which may be obtained from the FTDI website. Within the sample project the dll is located in a sub directory of the project called .\Library and referred to in the References section as shown in the excerpt from the solution explorer shown below.

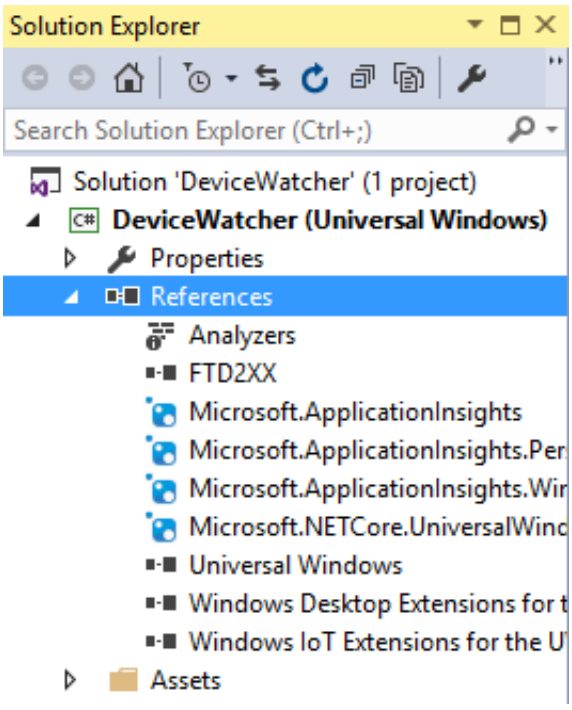

Figure 3.1 Solution Explorer

The FTD2XX DLL is the entry point for accessing the device drivers for the USB to I^2C/SPI devices. The API and programming guide for this can be found on the FTDI website:

FTDI D2xx Programmers Guide

An overview of the structure of the application is shown as a UML class diagram in the following figure. From this diagram it can be seen that the application is broadly partitioned into 5 classes. The FTApp class contains the main User application code. The classes CoPro, EveDL and MPSSEHAL together form a Stack to allow communications with the EVE display via the FTDI FT4232H Hub Module. The class Sensors forms the interface to the I^2C proximity and colour sensors.

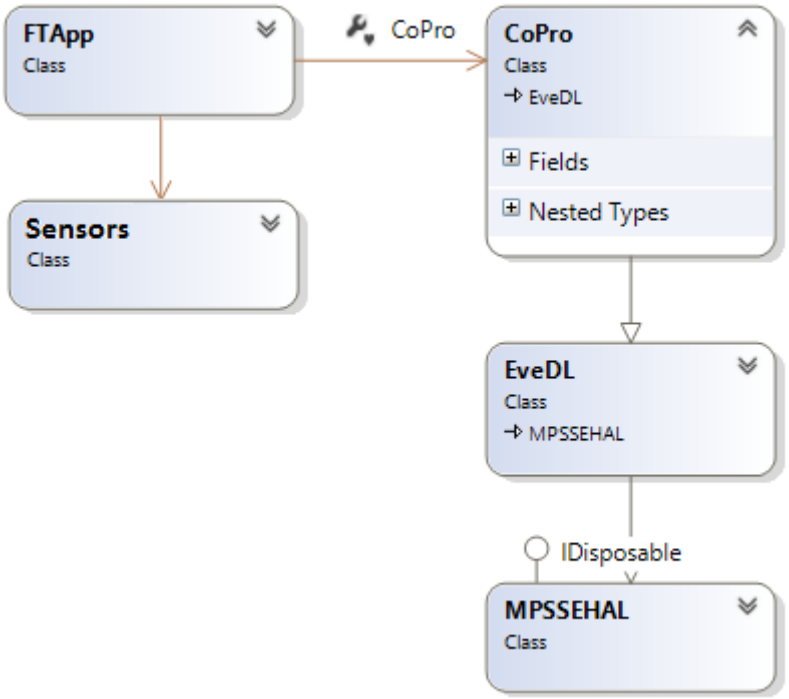

Figure 3.2 Class Diagram

3.1 FTApp

An expanded model of the FTApp Class is shown below:

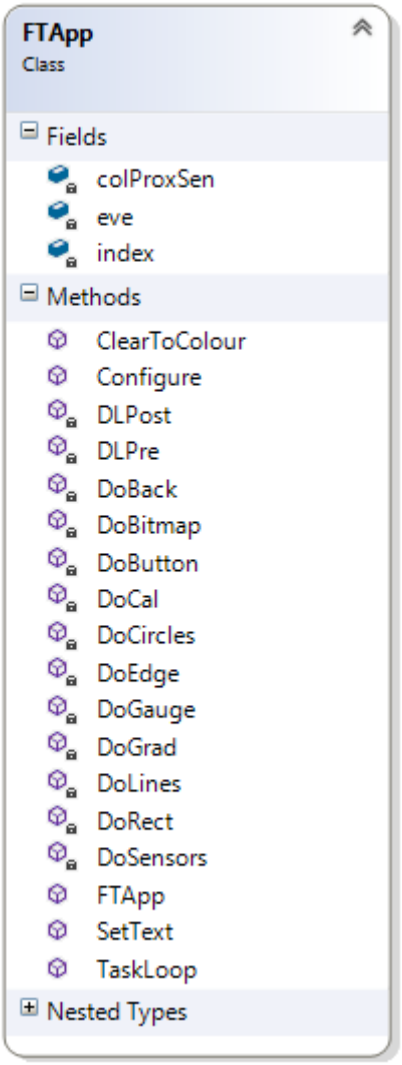

Figure 3.3 FT_App Class

It can be seen that the class contains the definition of several methods including a constructor, Configure and TaskLoop methods. In this case a single instance of the Class FTapp is created in the application and retained in the main class variable app. A timer is then set to call EveRun() after a 5 second delay. When EveRun is called it creates a Task which configures the app by passing in a device object for the respective Interface device to which the EVE display is connected. It then starts the main TaskLoop() method with the calls:

```
await app.Configure(myDevice);
await app.TaskLoop();
```

3.2 Task Loop

The Task loop is the main point of execution which runs as a continuous infinite loop.

```
for (int loop = 0; true; loop++)
{
   …
```

The FTApp class aggregates a Sensor class as a colProxObject. This object is used to obtain a set of readings from the connected colour and proximity sensors with:

```
proximity = await colProxSen.GetProximity();
color      = await colProxSen.GetColor();
```

The function then calls its class's private method DoSensors with the values for proximity and colour as parameters. The DoSensors method is then responsible for building up the graphics command to construct the screen displayed on the LCD panel of EVE. The application communicates with EVE by the Graphics communication stack, the top of which is formed by the Class CoPro. The app refers to CoPro by means of the class member variable eve.

Within the Do Sensors method the display list is built up from graphic primitives.

3.2.1 The graphics primitives:

The EVE graphics controller accepts a number of graphics primitives and widgets which can be used to display a range of graphical attributes from simple objects such as points/lines through to more complex items such as gauges, dials and images. These graphics items are represented in the code as a set of classes from which the Application developer may instantiate an object. For example, text can be displayed with the command below:

```
CoPro.Text hw = new CoPro.Text(280, 90,  28, 0, "Hello, World");
```

The constructor for the Text class is prototyped as

```
public Text (uint inx, uint  iny, uint infont, byte inoptions, string intext)
```

where

inx is the x coordinate of the location the text is to be placed,

iny is the y coordinate of the location the text is to be placed,
infont is the font to be used
inoptions is an options variable
inText is a standard C# string representing the text to be displayed.

It can be seen that the action of this statement is to construct a Text object containing the string "Hello, world" and to be displayed at coordinate (280,90) and to be rendered in the system font 28.

The object is assigned to the variable hw.

In basic terms the hw object is sent to the display by pushing it onto a FIFO stack. However, before this can be done a number of Display commands must be constructed to both precede and terminate the build-up of the screen. This will be discussed in following sections.

The following class diagram shows a subset of the co-processor Graphics Library commands which are available for the Application developer. It can be seen that the classes exist in a shallow inheritance hierarchy where common functionality is implemented in the FTEveCmd base class.

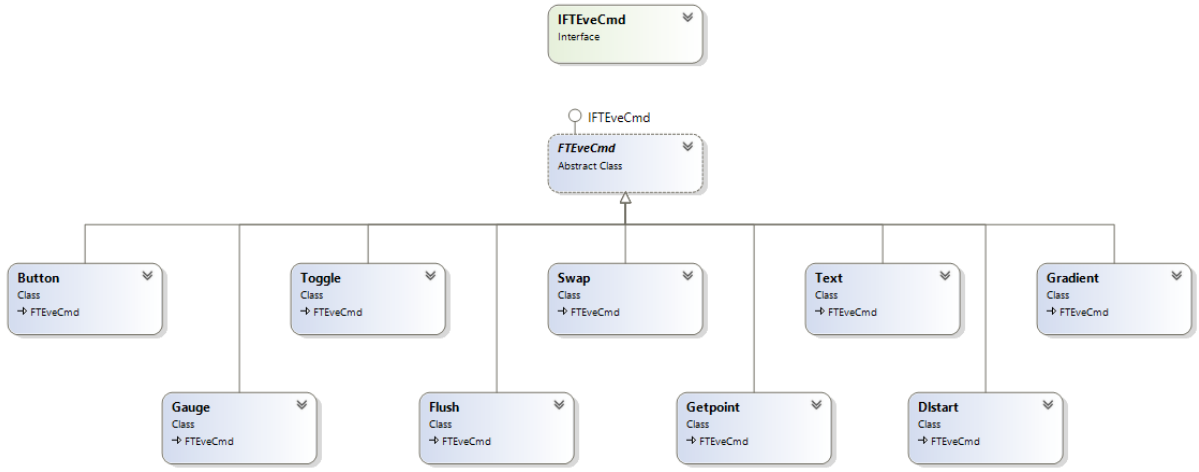

Figure 3.4 Class diagram of the Co-Processor Graphics Library

In Addition to the Co processor graphics object, the Stack also exposes the Display List commands. The display list commands are a more primitive set of commands for drawing primitives etc. and are defined in the EVDL class. However they share the IFTEveCmd in common with the Co-Processor commands.

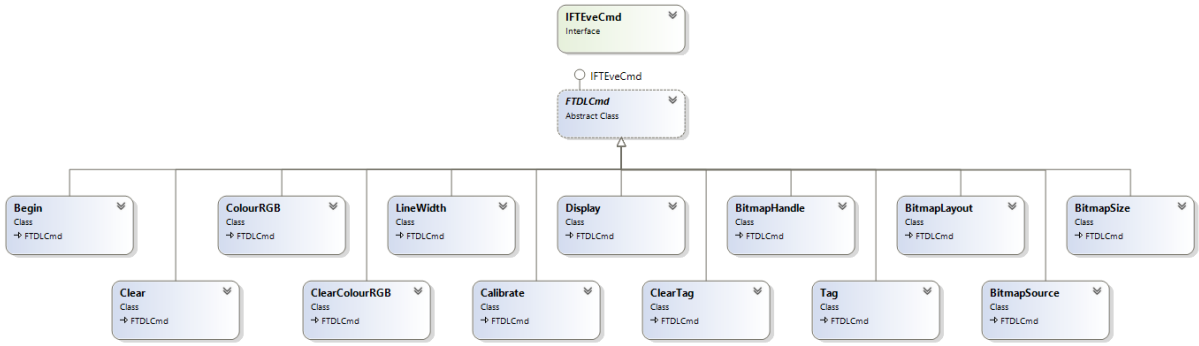

Figure 3.5 Display List commands

The interface between the class at the top of the graphics stack and the user application is in essence a FIFO stack. The stack holds IFTEveCmd types and the CoPro Class exposes a public method PushCmd() which takes a IFTEveCmd argument and its prototypes:

```
public void PushCmd(IFTEveCmd cmd)
```

In this way the application can exploit polymorphic behaviour and instantiate both CoPro and Display list command objects and pass them to the Graphics Stack., e.g.

```
eve.PushCmd(new CoPro.Dlstart());
eve.PushCmd(new EveDL.ClearColourRGB(0x00, 0x00, 0x00));
eve.PushCmd(new EveDL.Clear());
```

The operation is illustrated in the diagram below, where it can be seen that the user application creates a graphic object, essentially new'ing a Text object for example. This is pushed on to the graphics FIFO with the PushCmd. The application is then free to continue while a thread within CoProcessor removes the Commands from the other side of the FIFO, where this can be processed prior to sending the appropriate data to Eve.

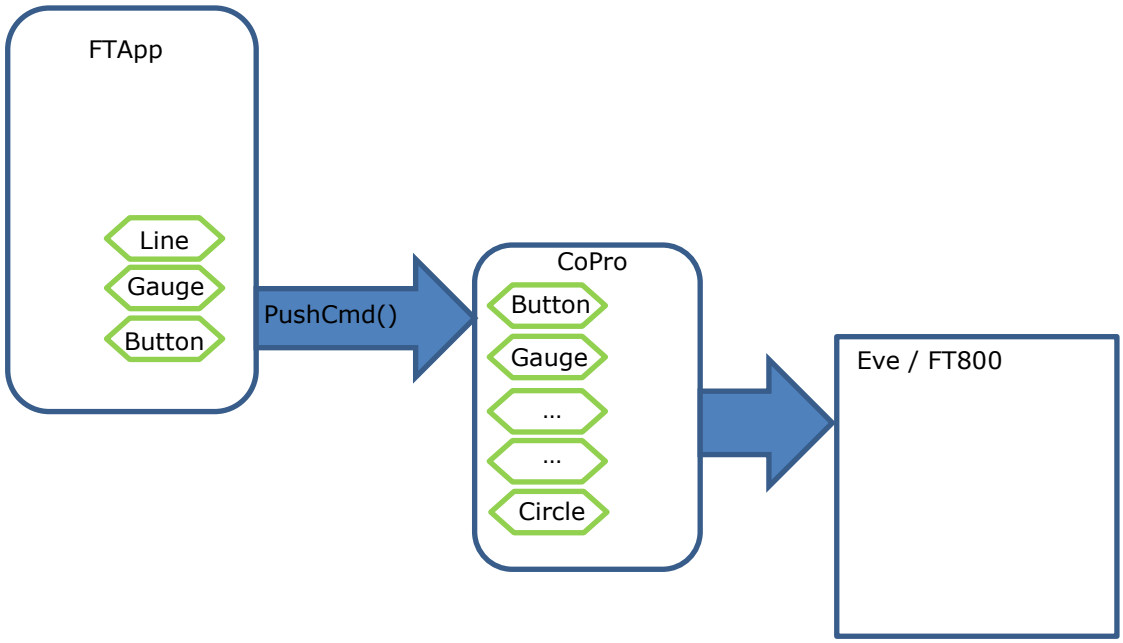

Figure 3.6 Graphics stack illustration

3.3 Display Screen

The EVE devices require a number of initial commands to initiate the display of a particular screen. Then, once the actual content has been sent, a set of three commands are used to commit the screen to the display. These commands are provided within `DLPre()` and `DLPost()` respectively.

The DLPre command is written as:

```
private void DLPre()
        {
            eve.PushCmd(new CoPro.Dlstart());
            eve.PushCmd(new EveDL.ClearColourRGB(0x00, 0x00, 0x00));
            eve.PushCmd(new EveDL.Clear());
            eve.PushCmd(new EveDL.ColourRGB((byte)0x00, (byte)(0x00), (byte)(0x00)));
        }
```

As shown, the preamble consists of a DLStart command followed by clearing the screen to a defined colour.

The DLPost command is written as:

```
        private void DLPost()
        {
            eve.PushCmd(new EveDL.Display());
            eve.PushCmd(new CoPro.Swap());
            eve.PushCmd(new CoPro.Flush());
        }
```

As shown, to complete the actions to display a screen, the Display command is appended to the list followed by a swap command. The flush is used to indicate the end of the overall data set for the new screen to be created.

The complete example from the Sample application method DoSensors is as follows:

```
DLPre();
if (p.Valid)
{
        radius = (ushort)(350 * Math.Log10(num));
        colName = colProxSen.GetColourName(col);

        eve.PushCmd(new CoPro.Gradient(240, 0, (uint)0x000000,
                                       240, 320, (uint)0x808080));
        proxTxt = "FTDI IoT";// + Convert.ToString(num);
        eve.PushCmd(new CoPro.Text(220, 0,   31, 0, proxTxt));

        eve.PushCmd(new CoPro.Text(280, 90,  28, 0, "Red   "    +
                                                    col.R.ToString()));
        eve.PushCmd(new CoPro.Text(280, 110, 28, 0, "Green "    +
                                                    col.G.ToString()));
        eve.PushCmd(new CoPro.Text(280, 130, 28, 0, "Blue  "    +
                                                    col.B.ToString()));
        if (radius > 500)
              eve.PushCmd(new CoPro.Text(280, 180, 28, 0, "Hue:  " +
                                                    colName.ToString()));
        else
              eve.PushCmd(new CoPro.Text(280, 180, 28, 0, "Hue:  " ));
       ushort X0 = 120;
       ushort Y0 = 79;



        eve.PushCmd(new EveDL.Begin(DLprim.POINTS));
        eve.PushCmd(new EveDL.ColourRGB((byte)0xFF, (byte)0xFF, (byte)0xFF));
        eve.PushCmd(new EveDL.PointSize((ushort)(1950)));
        eve.PushCmd(new EveDL.Vertex2II((ushort)X0, (ushort)Y0, 0, 0));
        eve.PushCmd(new EveDL.ColourRGB((byte)0x00, (byte)0x00, (byte)0x00));
        eve.PushCmd(new EveDL.PointSize((ushort)(1900)));
        eve.PushCmd(new EveDL.Vertex2II((ushort)X0, (ushort)Y0, 0, 0));
        if ( radius > 500)
              eve.PushCmd(new EveDL.ColourRGB((byte)col.R, (byte)col.G,
                                              (byte)col.B));
        else
              eve.PushCmd(new EveDL.ColourRGB((byte)0xff, (byte)0x00,
                                                 (byte)0x00));
        eve.PushCmd(new EveDL.PointSize((ushort)(radius+100)));
        eve.PushCmd(new EveDL.Vertex2II((ushort)(X0), (ushort)(Y0), 0, 0));


}

 DLPost();
```

The functions starts with a DLPre() as discussed above. A gradient is drawn to act as a background. Following this some local variables are assigned values for proximity and the RGB data for colours. These variables are then used to provide appropriate strings for the TEXT graphics commands.

A Begin(points) command is added to initiate the drawing of a number of points. A series of Vertex, Color_RGB and Point_Size commands are then used to plot the circles used in the display. A large white point followed by a slightly smaller black point form the white border circle.

A third circle centred on the same coordinate varies in size and colour depending on the values read from the proximity and colour sensors. The colour of this indicator circle is set to red whilst no object is in close proximity, but changes to match the RGB colour of the object when it is in close proximity. The radius and position of the circle are then declared, with the radius being proportionate to the proximity sensor output.

```
eve.PushCmd(new EveDL.PointSize((ushort)(radius+100)));
eve.PushCmd(new EveDL.Vertex2II((ushort)(X0), (ushort)(Y0), 0, 0));
```

Finally, the DLPost causes the graphics commands to be flushed and presented on the Display screen.

Figure 3.7 Illustration of the display

3.4 I²C Sensor Functions

The code for the sensors can be found in the file Sensors.cs which is within the supplied source code zip. The application I²C functions are arranged into several layers which are described in this section:

- Sensor Application Layer
- I²C API Layer
- D2xx Interface Layer

It is recommended that these functions are modified by the developer of the final application based on their chosen sensors in order to provide the best performance and to provide error checking tailored to the sensors, some of which is omitted here for clarity. The application notes AN_108 and AN_135 should be consulted for further details (see Appendix A – References).

3.4.1 Sensor Application Layer

This layer provides the functions which are called by the main application to configure and read the I²C sensors. The code here is specific to the address and register map of the sensors being used, and in turn calls the generic I²C functions lower down. This layer provides an easy way to modify the application to utilize different I²C sensors.

- await ColourSensorConfig();
- await ProximitySensorConfig();
- await ColourSensorReading();
- await ProximitySensorReading();

Each sensor has a Config function which is used to set up the registers in the sensor. This would typically be called once during application start-up to get the sensor configured to the point where the results can be read. Each sensor also has a shorter Reading function which is called each time a reading is to be taken.

Colour

For colour readings, the sensor provides four 16-bit values which represent the clear, red, green and blue quantities of light detected. The sensor measures the light reflected off the object and so requires a white LED to illuminate the object. The application performs some basic operations on the R, G and B results to scale these into 8-bit R, G and B values which can be used by the FT800 device. The resulting RGB value is returned. The demo could be extended by performing more comprehensive calibration and scaling to improve accuracy. The accuracy is also dependent on the ambient conditions in which the sensor is used. A combination of mechanical enclosure design and the processing carried out on the readings can be used to optimise this.

Proximity

For proximity readings, the sensor has an IR source and detector on the same IC. It also has an ambient light sensor which is not used in this application. A status register indicates the readiness of a sample and the result can then be read. The resulting value from the Prox_Value register is a 16-bit value which varies from 0 to 65535 in relation to the distance of the object, with 65535 being the closest. This value is passed to the main application. The main application scales this factor to provide a response that appears linear to the user. As with the colour sensor, the proximity code could be extended to utilize the full feature set of the sensor and carry out calibration or alternative sensors could be used instead.

The function also checks the value of the proximity in comparison to a threshold of 1000. If the proximity is greater than 1000, the GPIO variable is configured such that bit 3 is low. Otherwise this bit is set high. A low value will turn on the white LED to illuminate the object to allow colour readings to be taken, since the object is now close enough to obtain a valid reading from the colour sensor.

Using the I²C functions

An example of writing to a register is shown below, where the devices I²C address is sent followed by the register address and the data to be written. The address is defined in the code as already shifted to the upper 7 bits, allowing the LSB to be OR'ed with the R/W bit.

```
await SendI2CStart();
await SendByteAndCheckACK((byte)(VCNL40x0_ADDRESS | 0x00)); // R/W bit is 0
await SendByteAndCheckACK((byte)(REGISTER_PROX_CURRENT));
await SendByteAndCheckACK((byte)(20));
await SendI2CStop();
```

An example of reading from a sensor using the individual byte functions is shown below, where a write occurs to the devices I²C address specifying the register to be read. A repeated start follows, and then a read cycle to the devices I²C address which will return the value. The read is sent with a NAK to indicate to the sensor that the Master does not wish to read any further bytes in this transaction.

```
await SendI2CStart();
await SendByteAndCheckACK((byte)(VCNL40x0_ADDRESS | 0x00)); // R/W bit is 0
await SendByteAndCheckACK((byte)(REGISTER_COMMAND));
await SendI2CStart();
await SendByteAndCheckACK((byte)(VCNL40x0_ADDRESS | 0x01)); // R/W bit is 1
await ReadByteAndSendNAK();
Command = InputBuffer2[0];
await SendI2CStop();
```

3.4.2 I²C API Layer

This layer constructs sequences of MPSSE commands in order to implement the various operations required by the I²C protocol.

One thing to note when comparing the source code for these functions to code for the FT232H (e.g. AN_255) is that the pin directions are written using the MPSSE GPIO low byte command each time the line changes between reading and writing. The FT2232H and FT4232H do not have an open-drain feature but an equivalent operation can be created by tri-stating the data out line when not transmitting onto the I²C bus. The MPSSE makes this possible by allowing GPIO writes to be sent in the same buffer of commands as the clocking operation.

GPIO

The functions in this layer use this combination of GPIO writes and clocking commands to implement the I²C protocol. When performing GPIO writes, the function will define the states of the bits 0-2 as required for I²C implementation and will set the pin state and direction from global variables GPIO_Low_Dat and GPIO_Low_Dir. Therefore, the application may set these global variables at any time and their values will be updated onto the pins the next time an I²C transaction occurs. For example, in this demo a line is used to control the LED on the colour sensor module. It would also be possible to create a function which would perform no I²C transaction but just mask and write the new GPIO states. Likewise, the MPSSE has commands for reading GPIO and so the lines can be used as inputs too.

I²C Functions

A description of the functions provided follows:

InitMPSSE_For_I2C()

After a port on the device has been opened and a handle obtained, this function is called to put the port into MPSSE mode and configure it for I²C usage.

SendI2CStart()

This function uses the MPSSE commands to write the state and direction of the port used for I²C. The sequence of pin states causes an I²C Start condition on the bus. As with all functions in this layer, the non-I²C bits 3-7 of the port are set as per the bits 3-7 of the global GPIO variables.

SendByteAndCheckACK(byte DataByteToSend)

This function will clock out a byte and then clock in a single bit which is the ACK bit from the Slave.

ReadByteAndSendACK()

This function will clock in one byte from the I²C slave and will clock out a single logic 0 bit to send an ACK.

ReadByteAndSendNAK()

This function will clock in one byte from the I^2C slave and will clock out a single logic 1 bit to send a NAK. This will typically be used for the last read in a sequence, where the slave interprets the NAK as an indication that the Master does not want to read further bytes.

Read2BytesAndSendNAK()

This function is provided as an example where multiple bytes can be read within a single buffer of commands to the MPSSE. This follows a similar principle to the single byte reads mentioned above but buffers the commands up and sends with a single FT_Write. This is more efficient when reading multiple bytes because the USB host will normally send the entire command set on a single USB microframe. In comparison, calling the single byte read function twice will require several USB microframe times for the write of the command, return of the byte which was clocked in and then sending the next set of MPSSE commands. Taking advantage of the MPSSE's flexibility, various custom configurations can be created depending on when the host needs to check the result of a transaction before beginning the next.

SendI2CStop()

This function uses the same principle as the SendI2CStart function, but uses the GPIO states to create the Stop condition.

3.4.3 D2xx interface layer

This layer is responsible for communicating with the FTDI D2xx driver. It performs the writing of the MPSSE command buffers generated by the layer above and performs the reading of data which comes back from the FT4232H.

D2xx_FlushBuffer()

This function can be called to read any data remaining in the driver's read buffer in order to ensure the buffer is clear. It is only used during the initialization of the device in this application.

MyD2xx_WriteBytes(byte[] buf, uint bytesToWrite)

This function performs an FT_Write and checks that the correct number of bytes were sent.

D2xx_ReadBytes()

This function reads a specified number of bytes from the device. It waits in a loop until the requested number of bytes are read or until a software timeout expires (500 times round the loop). Each time through the loop will call the myFtdiDevice.GetRxBytesAvailable function followed by myFtdiDevice.Read if the bytes available is greater than 0. Any data read is added to a buffer which will be read by the main application. In this application, the amount of data being read at a time is only small, often just one byte, and so will normally come back quickly in a single USB microframe but the function has been written to demonstrate a technique for larger amounts of data.

4 Summary

The hardware and source code described in this application note demonstrate the way in which FTDI's bridging and display solutions can be used to provide both the I/O capabilities and local user interface when developing IoT applications.

The code provided shows how FTDI's new D2xx Universal Windows Driver for UWP can be used to interface to these bridging/display devices under the Universal Windows Platform and provides a starting point for developing these applications.

FTDI have a wide range of USB bridging devices and EVE display controllers which would allow this demo to be extended significantly to create a fully featured sensor and display application with local touch screen control. This makes them ideal companions to the small embedded processor boards such as the Raspberry Pi 2 which are becoming increasingly popular in IoT and general sensor applications.

5 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com

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: +886-2-8797 1330
Fax: +886-2-8751-9737

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

Appendix A – References

Document References

FT4232H Hi Speed Hub module

EVE Display Controllers

Raspberry Pi

D2xx Programmers Guide

AN_108 MPSSE Command Set

AN_135 MPSSE Basics

Colour Sensor

Proximity Sensor

Visual Studio Community

Acronyms and Abbreviations

Terms	Description
USB	Universal Serial Bus
RPi	Raspberry Pi embedded computer
IoT	Internet Of Things
UWP	Universal Windows Platform
MPSSE	Multi-Protocol Synchronous Serial Engine
EVE	Embedded Video Engine
I^2C	Inter-Integrated Circuit bus
SPI	Serial Peripheral Interface

Appendix B – List of Tables & Figures

List of Figures

Appendix C – Revision History

Document Title: AN_475 FTDI Windows 10 IoT Solutions
Document Reference No.: FT_001334
Clearance No.: FTDI# 503
Product Page: http://www.ftdichip.com/FTProducts.htm
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2016-05-23