Application Note

AN_407

D3XX .NET Programmers Guide

Version 1.0

Issue Date: 2016-11-01

FTDI provides a .NET DLL class library to its SuperSpeed USB drivers. This document provides information about the class library for FTD3XX .NET DLL.

Future Technology Devices International Limited (FTDI)
Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
Web Site: http://ftdichip.com

Table of Contents

1 Introduction

The D3XX .NET library is a .NET class library for the D3XX interface which is a proprietary interface specifically for FTDI SuperSpeed USB devices (FT60x series). It allows application developers to write C# applications for the FT60X chip series.

C# is a simple, powerful, type-safe, and object-oriented programming language designed for building all kinds of applications, ranging from desktop to mobile to web. C# applications run on top of .NET framework which provides a rich set of high-level programming environment, interfaces and services (such as automatic memory management and hardware/OS virtualization) while providing low-level access to native memory and APIs.

This document provides an explanation of the class library available to application developers to build .NET applications for FT60X chips using any modern language such as C#, F#, Managed C++ or Visual Basic programming language.

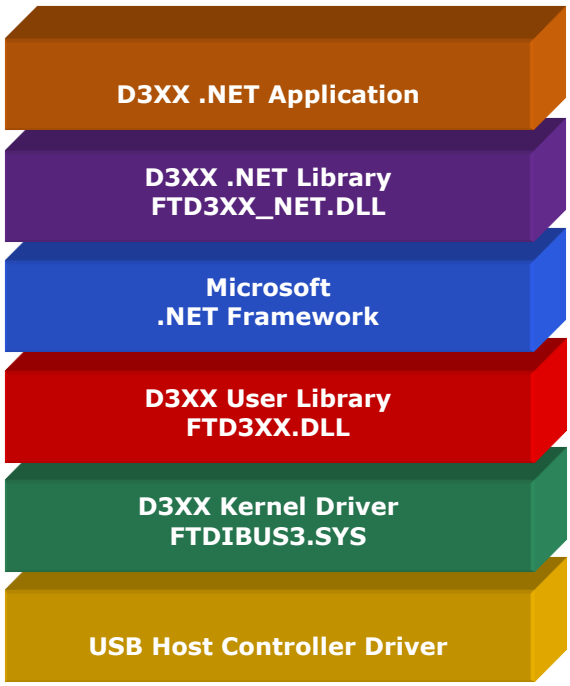

Figure 1.1 D3XX .NET Driver Architecture

D3XX .NET Library (FTD3XX_NET.dll) is a wrapper class library that translates managed code to the native/unmanaged D3XX User Library (FTD3XX.dll). The D3XX .NET Library source code is written in C# and is provided as open-source (similar to the D2XX .NET library) so customers can customize it for their specific requirements (i.ei.e., recompile to use .NET framework versions lower than 4.5).

Any software code examples given in this document are for information only. The examples are not guaranteed (i.ei.e., error handling has been removed for conciseness). For demo applications, please refer to the website. FTDI provides a number of demo applications to help customerscustomers' jumpstart their projects. Console-based and UI-based modern WPF demo applications are provided.

2 Overview

FT600 and FT601 are the first devices in a brand new USB SuperSpeed series from FTDI Chip. The devices provide a USB 3.0 SuperSpeed to FIFO Bridge, with up to 5Gbps of bandwidth. With the option of 16 bit (FT600) and 32 bit (FT601) wide parallel FIFO interfaces, the FT60x enables connectivity for numerous applications including high resolution cameras, displays, multifunction printers and much more.

The FT60x series implements a proprietary Function Protocol to maximize USB 3.0 bandwidth. The Function Protocol is implemented using 2 interfaces – communication interface and data interface. The data interface (Interface 1) can contain at most 4 channels with each channel having a read and write BULK endpoint, for a maximum total of 8 data endpoints. The communication interface (Interface 0) includes 2 dedicated endpoints, EP OUT BULK 0x01 and EP IN INTERRUPT 0x81. The OUT BULK endpoint is for sending session list commands from the host, targeted mainly for high volume, data traffic between the host and the FT60x device. The EP IN INTERRUPT endpoint is for host notification about the IN pipes that have pending data which is not scheduled by the session list, targeted mainly for low volume traffic. Combining the use of the two endpoints above provides performance and flexibility.

Interfaces	Endpoints	Description
0	0x01	OUT BULK endpoint for Session List commands
	0x81	IN INTERRUPT endpoint for Notification List commands
1	0x02-0x05	OUT BULK endpoint for application write access
	0x82-0x85	IN BULK endpoint for application read access

Table 1 FT600/FT601 Series Function Protocol Interfaces and Endpoints

shows the D3XX API functions together with the mapping to the D3XX .NET Class Library.

Category	D3XX Functions	D3XX .NET Class Library
Device Management	FT_CreateDeviceInfoList(), FT_GetDeviceInfoList(), FT_ListDevices(), FT_GetDeviceInfoDetail()	CreateDeviceInfoList(), GetDeviceInfoList(), GetNumberOfDevicesConnected(), <No mapping>
Device Opening/Closing	FT_Create(), FT_Close(), FT_IsDevicePath()	OpenByIndex(),OpenByDescription(), OpenBySerialNumber(), Close() IsOpen property, IsUSB3 property IsDevicePath
Data Transfer	FT_WritePipe(), FT_ReadPipe(), FT_InitializeOverlapped, FT_ReleaseOverlapped FT_GetOverlappedResult()	WritePipe(),WritePipeAsync(), ReadPipe(),ReadPipeAsync(), <No mapping> <No mapping> WaitAsync()

Category	D3XX Functions	D3XX .NET Class Library
	FT_AbortPipe(), FT_FlushPipe() FT_SetStreamPipe(), FT_ClearStreamPipe() FT_SetPipeTimeout(), FT_GetPipeTimeout()	AbortPipe(), FlushPipe() SetStreamPipe(), ClearStreamPipe() SetPipeTimeout(), GetPipeTimeout()
Vendor-Specific Control Transfer	FT_ControlTransfer(), FT_GetFirmwareVersion() FT_GetChipConfiguration(), FT_SetChipConfiguration()	ControlTransfer(), FirmwareVersion property GetChipConfiguration(), SetChipConfiguration(), ResetChipConfiguration()
GPIO	FT_ReadGPIO(), FT_WriteGPIO(), FT_EnableGPIO(),FT_ SetGPIOPull()	ReadGPIO(), WriteGPIO(), EnableGPIO(),SetGPIOPull()
Notification	FT_SetNotificationCallback(), FT_ClearNotificationCallback()	SetNotificationCallback(), ClearNotificationCallback()
Port Re-enumeration	FT_CycleDevicePort()	CycleDevicePort()
Version	FT_GetDriverVersion(), FT_GetLibraryVersion()	DriverVersion property, LibraryVersion property
Descriptor	FT_GetVIDPID FT_GetDeviceDescriptor(), FT_GetConfigurationDescriptor(), FT_GetInterfaceDescriptor(), FT_GetPipeInformation()	VendorID/ProductID property DeviceDescriptor property, ConfigurationDescriptor property, InterfaceDescriptor property, ReservedPipeInformation property, DataPipeInformation property
Suspend	FT_SetSuspendTimeout(), FT_GetSuspendTimeout()	SetSuspendTimeout(), GetSuspendTimeout()

Table 2 D3XX to D3XX .NET Mapping

3 D3XX .NET Class Library

The D3XX .NET Class Library, *FTD3XX_NET.DLL,* has a class name *FTDI* and a namespace *FTD3XX_NET*. The class provides the following functions and properties.

3.1 GetNumberOfDevicesConnected

```
FT_STATUS
GetNumberOfDevicesConnected (
    out UInt32 pulNumDevices
)
```

Summary

Returns the number of D3XX devices connected to the system. The count includes both unopened and open D3XX devices.

Parameters

pulNumDevices	An out reference to an unsigned integer to store the number of devices connected.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;

ftStatus = d3xxDevice.GetNumberOfDevicesConnected(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
{
    return;

}
```

3.2 CreateDeviceInfoList

```
FT_STATUS
CreateDeviceInfoList(
    out UInt32 pulNumDevices
)
```

Summary

Builds a device information list and returns the number of D3XX devices connected to the system. The list contains information about both unopen and open D3XX devices.

Parameters

pulNumDevices	An out reference to an unsigned integer to store the number of devices connected.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

An application should call this function before calling *GetDeviceInfoList* which can be used to get information about the devices attached to the system.

If the devices connected to the system change, the device info list will not be updated until *CreateDeviceInfoList* is called again.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;
UInt32 i = 0;

ftStatus = d3xxDevice.CreateDeviceInfoList(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
{
    return;
}

List<FTDI.FT_DEVICE_INFO> ListDeviceInfo;
ftStatus = d3xxDevice.GetDeviceInfoList(out ListDeviceInfo);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var DeviceInfo in ListDeviceInfo)
{
    string SerialNumber = d3xxDevice.GetSerialNumber(DeviceInfo);
    string Description = d3xxDevice.GetDescription(DeviceInfo);

    Console.WriteLine("\tDEVICE[{0:d}]", i);
    Console.WriteLine("\tSerialNumber            : " + SerialNumber);
    Console.WriteLine("\tDescription             : " + Description);
    Console.WriteLine("\tFlags                   : {0:d}", DeviceInfo.Flags);
    Console.WriteLine("\tLocId                   : {0:d}", DeviceInfo.LocId);
    Console.WriteLine("\tID                      : 0x{0:X8}", DeviceInfo.ID);
    Console.WriteLine("\tType                    : {0:d}\r\n", DeviceInfo.Type);
}
```

3.3 GetDeviceInfoList

```
FT_STATUS
GetDeviceInfoList(
   out List<FTDI.FT_DEVICE_INFO> listDevices
)
```

Summary

Returns a list of device information for all D3XX devices connected to the system.

Parameters

listDevices	An out reference to a list of device information to contain information for all devices connected to the system.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

This function should only be called after calling *CreateDeviceInfoList*. If the devices connected to the system change, the device info list will not be updated until *CreateDeviceInfoList* is called again.

Information is not available for devices which are open in other processes. In this case, the Flags parameter of the *FT_DEVICE_INFO* will indicate that the device is open, but other fields will be unpopulated.

The *Type* field of *FT_DEVICE_INFO* structure can be used to determine the device type. Currently, D3XX only supports FT60X devices, FT600 and FT601. The values returned in the *Type* field are located in the *FT_DEVICES* enumeration. FT600 and FT601 devices have values of *FT_DEVICE_600* and *FT_DEVICE_601*, respectively.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;
UInt32 i = 0;

ftStatus = d3xxDevice.CreateDeviceInfoList(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
    return;

List<FTDI.FT_DEVICE_INFO> ListDeviceInfo;
ftStatus = d3xxDevice.GetDeviceInfoList(out ListDeviceInfo);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
    return;

foreach (var DeviceInfo in ListDeviceInfo)
{
    string SerialNumber = d3xxDevice.GetSerialNumber(DeviceInfo);
    string Description = d3xxDevice.GetDescription(DeviceInfo);

    Console.WriteLine("\tDEVICE[{0:d}]", i);
    Console.WriteLine("\tSerialNumber             : " + SerialNumber);
    Console.WriteLine("\tDescription              : " + Description);
    Console.WriteLine("\tFlags                    : {0:d}", DeviceInfo.Flags);
    Console.WriteLine("\tLocId                    : {0:d}", DeviceInfo.LocId);
    Console.WriteLine("\tID                       : 0x{0:X8}", DeviceInfo.ID);
    Console.WriteLine("\tType                     : {0:d}\r\n", DeviceInfo.Type);
}
```

3.4 OpenByIndex

```
FT_STATUS
OpenByIndex(
   UInt32 ulIndex
)
```

Summary

Opens a D3XX device located at the specified index in relation to the list of device information.

Parameters

ulIndex	Index of the device to open in relation to the list of device information

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The parameter *ulIndex* refers to the position the specific device appears in the list of device information for all D3XX devices connected that can be retrieved using *GetDeviceInfoList*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;


ftStatus = d3xxDevice.CreateDeviceInfoList(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
{
    return;
}

List<FTDI.FT_DEVICE_INFO> ListDeviceInfo;

ftStatus = d3xxDevice.GetDeviceInfoList(out ListDeviceInfo);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

for (UInt32 i = 0; i < ListDeviceInfo.Count; i++)
{
    ftStatus = d3xxDevice.OpenByIndex(i);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }

    ftStatus = d3xxDevice.Close();
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }
}
```

3.5 OpenBySerialNumber

```
FT_STATUS
OpenBySerialNumber(
   string szSerialNumber
)
```

Summary

Opens a D3XX device with the specified serial number.

Parameters

szSerialNumber	Serial number of the device to open

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The parameter *szSerialNumber* refers to a serial number that appears in the list of device information for all D3XX devices connected that can be retrieved using *GetDeviceInfoList*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;


ftStatus = d3xxDevice.CreateDeviceInfoList(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
{
    return;
}

List<FTDI.FT_DEVICE_INFO> ListDeviceInfo;

ftStatus = d3xxDevice.GetDeviceInfoList(out ListDeviceInfo);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var DeviceInfo in ListDeviceInfo)
{
    string szSerialNumber = d3xxDevice.GetSerialNumber(DeviceInfo);

    ftStatus = d3xxDevice.OpenBySerialNumber(szSerialNumber);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }

    ftStatus = d3xxDevice.Close();
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }
}
```

3.6 OpenByDescription

```
FT_STATUS
OpenByDescription(
   string szDescription
)
```

Summary

Opens a D3XX device with the specified description.

Parameters

szDescription	Description of the device to open

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The parameter *szDescription* refers to a description that appears in the list of device information for all D3XX devices connected that can be retrieved using *GetDeviceInfoList*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulNumDevices;


ftStatus = d3xxDevice.CreateDeviceInfoList(out ulNumDevices);
if (ftStatus != FTDI.FT_STATUS.FT_OK || ulNumDevices == 0)
{
    return;
}

List<FTDI.FT_DEVICE_INFO> ListDeviceInfo;

ftStatus = d3xxDevice.GetDeviceInfoList(out ListDeviceInfo);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var DeviceInfo in ListDeviceInfo)
{
    string szDescription = d3xxDevice.GetDescription(DeviceInfo);

    ftStatus = d3xxDevice.OpenByDescription(szDescription);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }

    ftStatus = d3xxDevice.Close();
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return;
    }
}
```

3.7 Close

FT_STATUS
FT_Close()

Summary
Close the handle to an open D3XX device.

Parameters
None.

Return Value
FT_OK if successful, otherwise the return value is an FT error code.

3.8 WritePipe

```
FT_STATUS
WritePipe(
   byte bPipe,
   byte[] pBuffer,
   UInt32 ulBytesToTransfer,
   ref UInt32 pulBytesTransferred
)
```

Summary

Write data to a pipe synchronously.

Parameters

bPipe	Pipe to send data to. Corresponds to the *PipeId* field in the *FT_PIPE_INFORMATION* structure. In the *PipeId* field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
pBuffer	Buffer that contains the data to write.
ulBytesToTransfer	The number of bytes to write. This number must be less than or equal to the size, in bytes, of the Buffer.
pulBytesTransferred	A reference to an unsigned integer to contain the number of bytes written to the pipe.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Starting from version 1.2.0.5, the kernel driver has a default transfer timeout value of 5000 milliseconds or 5 seconds and this can be changed by calling *SetPipeTimeout*. An application can call *SetPipeTimeout* with a timeout value of 0 to disable timeouts.

If WritePipe fails with an error code (status other than FT_OK), an application should call AbortPipe. In addition, it should observe the Abort Recovery procedure stated in AN_412_FT600_FT601 USB Bridge chips Integration document.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

for (UInt32 j = 0; j < bytes.Length; j++)
    bytes[j] = 0xAA;

foreach (var desc in d3xxDevice.DataPipeInformation )
{
    if (desc.PipeId < 0x80) // Is a write-pipe?
    {
        ftStatus = d3xxDevice.WritePipe(desc.PipeId, bytes, (UInt32)bytes.Length,
            ref bytesTransferred);
    }
}

ftStatus = d3xxDevice.Close();
```

3.9 ReadPipe

```
FT_STATUS
ReadPipe(
   byte bPipe,
   byte[] pBuffer,
   UInt32 ulBytesToTransfer,
   ref UInt32 pulBytesTransferred
)
```

Summary

Read data from a pipe synchronously.

Parameters

bPipe	Pipe to receive data from. Corresponds to the *PipeId* field in the *FT_PIPE_INFORMATION* structure. In the *PipeId* field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
pBuffer	Buffer to contain the data to read.
ulBytesToTransfer	The number of bytes to read. This number must be less than or equal to the size, in bytes, of the Buffer.
pulBytesTransferred	A reference to an unsigned integer to contain the number of bytes read from the pipe.

Return Value

FT_OK if successful or timeout, otherwise the return value is an FT error code.

Starting from version 1.2.0.5, the kernel driver has a default transfer timeout value of 5000 milliseconds or 5 seconds and this can be changed by calling *SetPipeTimeout*. An application can call *SetPipeTimeout* with a timeout value of 0 to disable timeouts.

If ReadPipe fails with an error code (status other than FT_OK), an application should call AbortPipe. In addition, it should observe the Abort Recovery procedure stated in AN_412_FT600_FT601 USB Bridge chips Integration document.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

foreach (var desc in d3xxDevice.DataPipeInformation)
{
    if (desc.PipeId > 0x80) // Is a read-pipe?
        ftStatus = d3xxDevice.ReadPipe(desc.PipeId, bytes, (UInt32)bytes.Length,
            ref bytesTransferred);
}

ftStatus = d3xxDevice.Close();
```

3.10 WritePipeAsync

```
FT_STATUS
WritePipeAsync(
    byte bPipe,
    byte[] pBuffer,
    UInt32 ulBytesToTransfer,
    ref UInt32 pulBytesTransferred,
    ref System.Threading.NativeOverlapped pOverlapped
)
```

Summary

Write data to a pipe asynchronously.

Parameters

bPipe	Pipe to send data to. Corresponds to the *PipeId* field in the *FT_PIPE_INFORMATION* structure. In the *PipeId* field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
pBuffer	Buffer that contains the data to write.
ulBytesToTransfer	The number of bytes to write. This number must be less than or equal to the size, in bytes, of the Buffer.
pulBytesTransferred	A reference to an unsigned integer to contain the number of bytes written to the pipe.
pOverlapped	A reference to a *NativeOverlapped* structure from the *System.Threading* namespace.

Return Value

FT_IO_PENDING if the transfer was sent asynchronously.
FT_OK if transfer completed immediately, otherwise the return value is an FT error code.

The function operates asynchronously and immediately returns FT_IO_PENDING. *WaitAsync* should be called to wait for the completion of this operation.

If WritePipeAsync fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call AbortPipe. In addition, it should observe the Abort Recovery procedure stated in AN_412_FT600_FT601 USB Bridge chips Integration document.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];
var pOverlapped = new System.Threading.NativeOverlapped();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.WritePipeAsync(0x02, bytes, (UInt32)bytes.Length,
    ref bytesTransferred, ref pOverlapped);
if (ftStatus != FTDI.FT_STATUS.FT_IO_PENDING) return;

ftStatus = d3xxDevice.WaitAsync(ref pOverlapped, ref bytesTransferred, true);
if (ftStatus != FTDI.FT_STATUS.FT_OK || bytesTransferred != (UInt32)bytes.Length)
    return;

ftStatus = d3xxDevice.Close();
```

3.11 ReadPipeAsync

```
FT_STATUS
ReadPipeAsync(
    byte bPipe,
    byte[] pBuffer,
    UInt32 ulBytesToTransfer,
    ref UInt32 pulBytesTransferred,
    ref System.Threading.NativeOverlapped pOverlapped
)
```

Summary

Read data from a pipe asynchronously.

Parameters

bPipe	Pipe to receive data from. Corresponds to the *PipeId* field in the *FT_PIPE_INFORMATION* structure. In the *PipeId* field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
pBuffer	Buffer to contain the data to read.
ulBytesToTransfer	The number of bytes to read. This number must be less than or equal to the size, in bytes, of the Buffer.
pulBytesTransferred	A reference to an unsigned integer to contain the number of bytes read from the pipe.
pOverlapped	A reference to a *NativeOverlapped* structure from the *System.Threading* namespace.

Return Value

FT_IO_PENDING if the transfer was sent asynchronously.
FT_OK if transfer completed immediately, otherwise the return value is an FT error code.

The function operates asynchronously and immediately returns FT_IO_PENDING. *WaitAsync* should be called to wait for the completion of this operation.

If ReadPipeAsync fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call AbortPipe. In addition, it should observe the Abort Recovery procedure stated in AN_412_FT600_FT601 USB Bridge chips Integration document.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];
var pOverlapped = new System.Threading.NativeOverlapped();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.ReadPipeAsync(0x82, bytes, (UInt32)bytes.Length,
    ref bytesTransferred, ref pOverlapped);
if (ftStatus != FTDI.FT_STATUS.FT_IO_PENDING) return;

ftStatus = d3xxDevice.WaitAsync(ref pOverlapped, ref bytesTransferred, true);
if (ftStatus != FTDI.FT_STATUS.FT_OK || bytesTransferred != (UInt32)bytes.Length)
    return;

ftStatus = d3xxDevice.Close();
```

3.12 WaitAsync

```
FT_STATUS
WaitAsync(
   ref System.Threading.NativeOverlapped pOverlapped,
   ref UInt32 pulBytesTransferred,
   bool bWait
)
```

Summary

Retrieves the result of an asynchronous write or read operation to or from a pipe.

Parameters

pOverlapped	A reference to ana *NativeOverlapped* structure from the *System.Threading* namespace that was specified when the overlapped operation was started using *WritePipeAsync* or *ReadPipeAsync*.
pulBytesTransferred	A reference to an unsigned integer that receives the number of bytes that were actually transferred by a read or write operation. The variable will only be updated when FT_OK is returned.
bWait	If this parameter is true, the function does not return until the asynchronous operation has been completed or the timeout expires in case of a read operation. Otherwise, if this parameter is false, the function returns FT_IO_INCOMPLETE until the asynchronous operation has completed.

Return Value

FT_IO_INCOMPLETE if bWait is false and IO transfer is not completed.
FT_OK if successful or timeout, otherwise the return value is an FT error code.

Starting from version 1.2.0.5, the kernel driver has a default transfer timeout value of 5000 milliseconds or 5 seconds and this can be changed by calling *SetPipeTimeout*. An application can call *SetPipeTimeout* with a timeout value 0 to disable timeouts.

If WaitAsync fails with an error code (status other than FT_OK), an application should call AbortPipe. In addition, it should observe the Abort Recovery procedure stated in AN_412_FT600_FT601 USB Bridge chips Integration document.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OTHER_ERROR;
UInt32 ulBytesTransferred = 0;
UInt32 ulPacketSize = 14745600; // QuadHD: 2560 x 1440 x 4
UInt32 ulQueueSize = 3; // Triple-buffering
NativeOverlapped[] listOverlapped = new NativeOverlapped[ulQueueSize];
List<byte[]> listBuffer = new List<byte[]>();
Int32 i = 0;
bool bResult = true;

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

for (i = 0; i < ulQueueSize; i++)
{
    byte[] bBuffer = new byte[oParams.ulPacketSize];
    listBuffer.Add(bBuffer);
}

// Queue up the initial batch of requests
for (i = 0; i < ulQueueSize; i++)
{
    ftStatus = d3xxDevice.ReadPipeAsync(0x82, listBuffer[i], ulPacketSize,
            ref ulBytesTransferred, ref listOverlapped[i]);
    if (ftStatus != FTDI.FT_STATUS.FT_IO_PENDING)
    {
        d3xxDevice.AbortPipe(0x82);
        SetStopComplete();
        d3xxDevice.Close();
        return;
    }
}

i = 0;

// Infinite transfer loop
while (!IsStopped())
{
    ulBytesTransferred = 0;

    // Wait for transfer to finish
    ftStatus = d3xxDevice.WaitAsync(ref listOverlapped[i],
            ref ulBytesTransferred, true);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        d3xxDevice.AbortPipe(0x82);
        break;
    }

    // Resubmit to keep requests full
    ftStatus = d3xxDevice.ReadPipeAsync(0x82, listBuffer[i], ulPacketSize,
            ref ulBytesTransferred, ref listOverlapped[i]);
    if (ftStatus != FTDI.FT_STATUS.FT_IO_PENDING)
    {
        d3xxDevice.AbortPipe(0x82);
        break;
    }

    // Roll-over
    if (++i == ulQueueSize)
        i = 0;
}

SetStopComplete();

ftStatus = d3xxDevice.Close();
```

3.13 AbortPipe

```
FT_STATUS
AbortPipe(
   byte bPipe
)
```

Summary

Aborts all pending and ongoing transfers in a pipe.

Parameters

bPipe	Pipe to be aborted.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
NativeOverlapped pOverlapped = new NativeOverlapped();
byte[] bBuffer = new byte[65536];
UInt32 bytesTransferred = 0;

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var desc in d3xxDevice.DataPipeInformation)
{
    if (desc.PipeId > 0x80) // Is a read-pipe?
    {
        ftStatus = d3xxDevice.ReadPipeAsync(desc.PipeId, bBuffer, (UInt32)bBuffer.Length,
            ref bytesTransferred, ref pOverlapped);

        // Abort the asynchronous read immediately
        ftStatus = d3xxDevice.AbortPipe(desc.PipeId);
    }
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.14 FlushPipe

```
FT_STATUS
FlushPipe(
    byte bPipe
)
```

Summary

Discards any data that is cached in an IN pipe.

Parameters

bPipe	Pipe to be flushed.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

3.15 SetStreamPipe

```
FT_STATUS
SetStreamPipe(
    byte bPipe,
    UInt32 ulStreamSize
)
```

Summary

Enable streaming protocol transfer for a specified pipe. This is for applications that transfer a fixed size of data to or from the device.

Parameters

bPipe	Pipe to enable streaming data.
ulStreamSize	Size of the data to be streamed constantly to or from the device.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

This function is used to improve performance. This informs the chip that the host will be reading or writing a fixed number of bytes continuously on the specified pipe. When this is used, *WritePipe* and *ReadPipe* (including *WritePipeAsync* and *ReadPipeAsync*) no longer send a session command to the chip because *SetStreamPipe* already informs the chip how much data to request. For example, when stream mode is set on an IN pipe, the chip will continuously read *ulStreamSize* bytes from the FIFO corresponding to the specified IN pipe. This function can be used in synchronous and asynchronous transfers.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

for (UInt32 i = 0; i < bytes.Length; i++)
    bytes[i] = 0xAA;

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.SetStreamPipe(0x02, (UInt32)bytes.Length);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
    return;

for (UInt32 i = 0; i < 1000; i++)
{
    // Use DataPipeInformation to get the actual PipeID
    ftStatus = d3xxDevice.WritePipe(0x02, bytes, (UInt32)bytes.Length,
            ref bytesTransferred);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
        return;
}

ftStatus = d3xxDevice.ClearStreamPipe(0x02);
ftStatus = d3xxDevice.Close();
```

3.16 ClearStreamPipe

```
FT_STATUS
ClearStreamPipe(
   byte bPipe
)
```

Summary

Disable streaming protocol transfer for a specified pipe.

Parameters

bPipe	Pipe to disable streaming data.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 bytesTransferred = 0;
byte[] bytes = new byte[16777216];

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

for (UInt32 i = 0; i < bytes.Length; i++)
{
    bytes[i] = 0xAA;
}

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.SetStreamPipe(0x82, (UInt32)bytes.Length);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

for (UInt32 i = 0; i < 1000; i++)
{
    // Use DataPipeInformation to get the actual PipeID
    ftStatus = d3xxDevice.ReadPipe(0x82, bytes, (UInt32)bytes.Length,
            ref bytesTransferred);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        d3xxDevice.Close();
        return;
    }
}

ftStatus = d3xxDevice.ClearStreamPipe(0x82);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
```

3.17 SetPipeTimeout

```
FT_STATUS
SetPipeTimeout(
   byte bPipe,
   UInt32 ulTimeoutInMs
)
```

Summary

Configures the timeout value of a given endpoint. *ReadPipe* and *WaitAsync* will timeout and return if there is no response within *TimeoutInMs* amount of time. This will override the default timeout of 5000 milliseconds or 5 seconds.

Parameters

bPipe	Pipe ID of the pipe to be used When 0xFF is used as ucPipeID, then the input specified in *ulTimeoutInMs* will be applied on all the endpoints.
ulTimeoutInMs	Time out in milliseconds the request on the specified pipe will timeout. If set to 0 (zero), transfers will not timeout. In this case, the transfer waits indefinitely until the transfer completes normally or the transfer is manually cancelled (call to *AbortPipe*). If set to a nonzero value (time-out interval), the request will be terminated once the timeout occurs. Default timeout value is 5000 milliseconds or 5 seconds.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Note that this function is only available starting from version 1.2.0.5 of the D3XX driver and D3XX library. Refer to *DriverVersion* and *LibraryVersion* properties to get the versions of the D3XX driver and D3XX library. Older versions don't support timeout on read transfers.

The value set is valid only until the life cycle of the handle. Reopening of the handle will reset the timeout to the default value.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulTimeout = 200;

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.SetPipeTimeout(0x82, ulTimeout);

ftStatus = d3xxDevice.Close();
```

3.18 GetPipeTimeout

```
FT_STATUS
GetPipeTimeout(
   byte bPipe,
   ref UInt32 ulTimeoutInMs
)
```

Summary

Gets the timeout value of a given endpoint.

Parameters

bPipe	Pipe ID of the pipe to be used.
ulTimeoutInMs	Time out in milliseconds the request on the specified pipe will timeout. If the return status is FT_OK, then this field will contain the timeout value configured for the corresponding pipe id. Default timeout value is 5000 milliseconds or 5 seconds.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Note that this function is only available starting from version 1.2.0.5 of the D3XX driver and D3XX library. Refer to *DriverVersion* and *LibraryVersion* properties to get the versions of the D3XX driver and D3XX library. Older versions don't support timeout on read transfers.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulTimeout = 0;


// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Use DataPipeInformation to get the actual PipeID
ftStatus = d3xxDevice.GetPipeTimeout(0x82, ref ulTimeout);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.19 GetChipConfiguration

```
FT_STATUS
GetChipConfiguration(
   FTDI.FT_CONFIGURATION oConfiguration
)
```

Summary

Retrieves the configuration of the device.

Parameters

oConfiguration — A reference to a FT_CONFIGURATION class object.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Remarks

A utility application called FT60x Chip Configuration Programmer is available for querying and modifying the chip's configuration.

Below is a detailed description of the parameters of *FT60x Chip Configuration*. For the data type of each parameter, please refer to Appendix A.

Parameter	Description
VendorID	Vendor identification as specified in the idVendor field of the USB Device Descriptor. This should match the VID in the Windows installation file (INF). If the customer changes this, then corresponding changes in the INF installation file are required also, meaning a separate WHQL process should be completed if driver certification is required.
ProductID	Product identification as specified in the idProduct field of the USB Device Descriptor. This should match the PID in the Windows installation file (INF). If the customer changes this, then corresponding changes in the INF installation file are required also, meaning a separate WHQL process should be completed if driver certification is required.
Manufacturer	Manufacturer string as contained in the String Descriptor corresponding to the iManufacturer field in the USB Device Descriptor. This is a 15 character printable string which uniquely identifies the manufacturer from other manufacturers using FT60X chips.
Description	Product Description string as contained in the String Descriptor corresponding to the iProduct field in the USB Device Descriptor. This is a 31 character printable string, which uniquely identifies the product from other products of a manufacturer using FT60X chips.
SerialNumber	Serial Number string as contained in the String Descriptor corresponding to the iSerialNumber field in the USB Device Descriptor. This is a 15 character string which uniquely identifies the item from other items of the same product of a manufacturer using FT60X chips.
FIFOClock	Clock speed of the FIFO in MHz (100 or 66)
FIFOMode	Mode of the FIFO (245 or 600)
ChannelConfig	Number of channels or pipes. A channel is equivalent to 2 pipes – 1

	for OUT and 1 for IN. (4 channels, 2 channels, 1 channel, 1 OUT pipe, 1 IN pipe) Refer to CHANNEL_CONFIG enumeration
PowerAttributes	Power attributes as specified in the bmAttributes field of the USB Configuration Descriptor. Bit 5 indicates if the device supports Remote Wakeup capability while Bit 6 indicates if the device is self-powered or bus-powered. Note that Bit 7 should always be 1, based on the USB specification
PowerConsumption	Maximum power consumption as specified in the bMaxPower field of the USB Configuration Descriptor. Note that a value of 0xC means a maximum power consumption of 0xC * 8 if USB 3.0 and 0xC * 2 if USB 2.0.
MSIO_Control	Configuration to control the drive strengths of FIFO pins
GPIO_Control	Configuration to control the drive strengths of GPIO pins
OptionalFeatureSupport	Bitmap indicating the optional feature support. Refer to **Error! Reference source not found.**.
BatteryChargingGPIOConfig	Bitmap indicating the type of power source detected that the device is connected to by the Battery Charging module of the chip. Refer to **Error! Reference source not found.**. Default setting : 11100100b (0xE4) 7 - 6 : DCP = 11b (GPIO1 = 1 GPIO0 = 1) 5 - 4 : CDP = 10b (GPIO1 = 1 GPIO0 = 0) 3 - 2 : SDP = 01b (GPIO1 = 0 GPIO0 = 1) 1 - 0 : Unknown/Off = 00b (GPIO1 = 0 GPIO0 = 0)
FlashEEPROMDetection	Bitmap indicating the status of the chip configuration initialization. Refer to **Error! Reference source not found.**.

Table 3 Chip Configuration Description

Bits	Description
15-6	Reserved
5	Enable Notification Message for Ch4 IN (Default value = 0) When this bit is set, notification messages will be enabled for the IN pipe of channel 4. Refer to Bit 2 for more information.
4	Enable Notification Message for Ch3 IN (Default value = 0) When this bit is set, notification messages will be enabled for the IN pipe of channel 3. Refer to Bit 2 for more information.
3	Enable Notification Message for Ch2 IN (Default value = 0) When this bit is set, notification messages will be enabled for the IN pipe of channel 2. Refer to Bit 2 for more information.
2	Enable Notification Message for Ch1 IN (Default value = 0) When this bit is set, notification messages will be enabled for the IN pipe of channel 1. Host applications will not actively read on this pipe, instead they will register a callback function using *SetNotificationCallback*. The callback function will be called when there is data available for the pipe. Using the provided information, the application can either read the data (using *ReadPipe*) or ignore/flush the data (using *FlushPipe*). This feature is intended for unexpected small packets (maximum of 4kb), such as error status information from the FIFO master to the host application. For example, for a camera device, the user can select 2 channel configurations because the application requires 2 IN pipes – 1 for camera data, 1 for control / error status information. Notification messages should be used for the control / error status information pipe but not for the camera data pipe.

1	Disable Cancel Session On Underrun (Default value = 0) When this bit is set, the chip will not cancel or invalidate IN requests from the host application when an underrun condition is received from the FIFO master and if the packet size received from the FIFO master is a multiple of the USB max packet size(USB3 : 1024, USB2 : 512). By default, the chip always cancels or invalidates IN requests from the host application when an underrun condition is received from the FIFO master. Underrun conditions happen when the FIFO master provides less data than the FIFOSegmentSize. FIFOSegmentSize is as follows: CHANNEL_CONFIG_4 : 1KB CHANNEL_CONFIG_2 : 2KB CHANNEL_CONFIG_1 : 4KB CHANNEL_CONFIG_1_OUTPIPE : 8KB CHANNEL_CONFIG_1_INPIPE : 8KB
0	Enable Battery Charging Detection (Default value = 0) When this bit is set, the 2 GPIOs will be configured to indicate the type of power source the device is connected to. The GPIO setting is indicated by BatteryChargingGPIOConfig.

Table 4 Bitmap for OptionalFeatureSupport

Bits	Description
7 – 6	DCP Dedicated Charging Port
5 – 4	CDP Charging Downstream Port
3 – 2	SDP Standard Downstream Port
1 - 0	Unknown/Off

Table 5 Bitmap for BatteryCharginGPIOConfig

Bit	Description
7	GPIO 1 status if Bit 5 is set(High = 1, Low = 0)
6	GPIO 0 status if Bit 5 is set(High = 1, Low = 0)
5	Is GPIO used as configuration input ? (Yes = 1, No = 0)
4	Is custom memory used ? (Custom = 1, Default = 0)
3	Is custom configuration data checksum invalid ? (Invalid = 1, Valid = 0)
2	Is custom configuration data invalid ? (Invalid = 1, Valid = 0)
1	Is Memory Not Exist ? (Not Exists = 1, Exist 0)
0	Is ROM ? (ROM = 1, Flash = 0)

Table 6 Bitmap for FlashEEPROMDetection

When the chip is set to default chip configuration, the 2 GPIO pins can be set to high or low to change the FIFO mode (FIFOMode) and Channel configuration (ChannelConfig). This feature allows customers to experiment with various channel configurations by simply adding or removing jumpers on the pins.

GPIO 1	GPIO 0	FIFOMode	ChannelConfig
0	0	FIFO_MODE_245	CHANNEL_CONFIG_1
0	1	FIFO_MODE_600	CHANNEL_CONFIG_1

1	0	FIFO_MODE_600	CHANNEL_CONFIG_2
1	1	FIFO_MODE_600	CHANNEL_CONFIG_4

Table 7 GPIO pins in Default Chip Configuration

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
var conf = new FTDI.FT_60XCONFIGURATION();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

ftStatus = d3xxDevice.GetChipConfiguration(conf);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

Console.WriteLine("\tChip Configuration");
Console.WriteLine("\tVendorID                 : 0x{0:X4}", conf.VendorID);
Console.WriteLine("\tProductID                : 0x{0:X4}", conf.ProductID);
Console.WriteLine("\tManufacturer             : " + conf.Manufacturer);
Console.WriteLine("\tDescription              : " + conf.Description);
Console.WriteLine("\tSerialNumber             : " + conf.SerialNumber);
Console.WriteLine("\tPowerAttributes          : 0x{0:X2}", conf.PowerAttributes);
Console.WriteLine("\tPowerConsumption         : 0x{0:X4}", conf.PowerConsumption);
Console.WriteLine("\tFIFOMode                 : 0x{0:X2}", oConfigurationData.FIFOMode);
Console.WriteLine("\tChannelConfig            : 0x{0:X2}", conf.ChannelConfig);
Console.WriteLine("\tOptionalFeatureSupport   : 0x{0:X4}", conf.OptionalFeatureSupport);
Console.WriteLine("\tBatteryChargingGPIOConfig: 0x{0:X2}", conf.BattChargingGPIOConfig);
Console.WriteLine("\tMSIO_Control             : 0x{0:X8}", conf.MSIO_Control);
Console.WriteLine("\tGPIO_Control             : 0x{0:X8}", conf.GPIO_Control);
Console.WriteLine("\tFlashEEPROMDetection     : 0x{0:X2}", conf.FlashEEPROMDetection);

if (conf.FlashEEPROMDetection > 0)
{
    bool bROM = (conf.FlashEEPROMDetection &
        (1 << (byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.ROM)) > 0;

    Debug.Log("\t\tMEMORY           : {0}", bROM ? "Rom" : "Flash");

    if (bROM)
    {
        bool bMemoryExist = (conf.FlashEEPROMDetection &
            (1 << (byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.MEMORY_NOTEXIST)) > 0;

        Debug.Log("\t\tMEMORY_NOTEXIST  : {0}", bMemoryExist ? "Invalid" : "Valid");
    }

    bool bCustom = (conf.FlashEEPROMDetection &
        (1 << (byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.CUSTOM)) > 0;

    Debug.Log("\t\tVALUES           : {0}", bCustom ? "Custom" : "Default");
```

```
    if (!bCustom)
    {
        // Default configuration

        bool bGPIO0 = (conf.FlashEEPROMDetection &
            (1 << (byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.GPIO_0)) > 0;

        bool bGPIO1 = (conf.FlashEEPROMDetection &
            (1 << (byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.GPIO_1)) > 0;

        Debug.Log("\t\tGPIO_0           : {0}", bGPIO0 ? "High" : "Low");
        Debug.Log("\t\tGPIO_1           : {0}", bGPIO1 ? "High" : "Low");

        if (bGPIO0 && bGPIO1)
        {
            Debug.Log("\t\tChannel          : 4 channels, 600 mode");
        }
        else if (!bGPIO0 && bGPIO1)
        {
            Debug.Log("\t\tChannel          : 2 channels, 600 mode");
        }
        else if (bGPIO0 && !bGPIO1)
        {
            Debug.Log("\t\tChannel          : 1 channel, 600 mode");
        }
        else if (!bGPIO0 && !bGPIO1)
        {
            Debug.Log("\t\tChannel          : 1 channel, 245 mode");
        }
    }
    else
    {
        // Custom configuration

        bool bInvalidData = (conf.FlashEEPROMDetection &
            (1<<(byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.CUSTOMDATA_INVALID))>0;

        bool bInvalidDataChecksum = (conf.FlashEEPROMDetection &
            (1<<(byte)FTDI.FT_60XCONFIGURATION_FLASH_ROM_BIT.CUSTOMDATACHKSUM_INVALID))>0;

        Debug.Log("\t\tCUSTOMDATA      : {0}", bInvalidData? "Invalid" : "Valid");
        Debug.Log("\t\tCUSTOMDATACHKSUM: {0}", bInvalidDataChecksum? "Invalid" : "Valid");
    }
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.20 SetChipConfiguration

```
FT_STATUS
SetChipConfiguration(
   FTDI.FT_CONFIGURATION oConfiguration
)
```

Summary

Sets the configuration of the device using the given values.

Parameters

oConfiguration	A reference to a FT_CONFIGURATION class object.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The device will restart after the chip configuration is written to the device.

If an application intends to change the chip configuration dynamically, it has to close the handle and open a new handle using *Close* and *OpenByXXX*, respectively.

A utility application called FT60x Chip Configuration Programmer is available for querying and modifying the chip's configuration.

To allow multiple FT60x devices to be connected to a machine, customers are required to update the String Descriptors (Manufacturer, Product Description, Serial Number) in the USB Device Descriptor by calling *SetChipConfiguration* or using the FT60x Chip Configuration Programmer tool provided by FTDI.

Manufacturer name, a 30 byte Unicode string (or 15 byte printable ASCII string), will uniquely identify the customer from other FT60x customers. Product Description, a 62 byte Unicode string (or 31 byte printable ASCII string), will uniquely identify the product from other products of the customer. Serial Number, a 30 byte Unicode string (or 15 byte alpha-numeric ASCII string), will uniquely identify the item from other items of the same product of a manufacturer.

Sample maxed-out values:

Manufacturer: My Company Name (15 chars maximum)
Description: This Is My Product Description0 (31 chars maximum)
SerialNumber: 1234567890ABCde (15 chars maximum)

The function will cause the device to re-enumerate so the application needs to reopen the driver handle to the device. The re-enumeration process, including driver re-initialization, can take from 200 milliseconds to 2 seconds so applications can either sleep or do some polling on *GetNumberOfDevicesConnected* before calling *OpenByXXX*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
var conf = new FTDI.FT_60XCONFIGURATION();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Set default values
conf.VendorID = FTDI.FT_60XCONFIGURATION_DEFAULT_VENDORID;
conf.ProductID = FTDI.FT_60XCONFIGURATION_DEFAULT_PRODUCTID_601;
conf.PowerAttributes = FTDI.FT_60XCONFIGURATION_DEFAULT_POWERATTRIBUTES;
conf.PowerConsumption = FTDI.FT_60XCONFIGURATION_DEFAULT_POWERCONSUMPTION;
conf.BatteryChargingGPIOConfig = FTDI.FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING;
conf.OptionalFeatureSupport = FTDI.FT_60XCONFIGURATION_DEFAULT_OPTIONALFEATURE;
conf.MSIO_Control = FTDI.FT_60XCONFIGURATION_DEFAULT_MSIOCONTROL;
conf.GPIO_Control = FTDI.FT_60XCONFIGURATION_DEFAULT_GPIOCONTROL;
conf.FlashEEPROMDetection = 0;

// Set custom values
conf.FIFOMode = (byte)FTDI.FT_60XCONFIGURATION_FIFO_MODE.MODE_600;
conf.ChannelConfig = (byte)FTDI.FT_60XCONFIGURATION_CHANNEL_CONFIG.ONE;
conf.Manufacturer = "FTDI";
conf.Description = "FTDI SuperSpeed-FIFO Bridge";
conf.SerialNumber = "123456789012345";

ftStatus = d3xxDevice.SetChipConfiguration(conf);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Allow the device to re-enumerate before re-opening the device by sleeping
// Or better yet poll for GetNumberOfDevicesConnected before reopening the handle
Thread.Sleep(1000);
```

3.21 ResetChipConfiguration

FT_STATUS
ResetChipConfiguration()

Summary

Sets the configuration of the device to default values.

Parameters

None.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The function will cause the device to re-enumerate so the application needs to reopen the driver handle to the device. Re-enumeration process, including driver re-initialization, can take from 200 milliseconds to 2 seconds so application can either sleep or do some polling on *GetNumberOfDevicesConnected* before calling *OpenByXXX*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

ftStatus = d3xxDevice.ResetChipConfiguration();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Allow the device to re-enumerate before re-opening the device by sleeping
// Or better yet poll for GetNumberOfDevicesConnected before reopening the handle
Thread.Sleep(1000);
```

3.22 EnableGPIO

```
FT_STATUS
EnableGPIO(
   UInt32 ulMask,
   UInt32 ulDirection
)
```

Summary

Configures pin into GPIO mode and the GPIO direction.

Parameters

ulMask	Reserved for future. Currently it is ignored.
ulDirection	Bit0 and Bit1 are used and bits [31:2] are unused. Bit0 controls the direction of GPIO0 and Bit1 controls the direction of GPIO1. A value of 0 indicates Input direction while a value of 1 indicates Output direction.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

3.23 WriteGPIO

```
FT_STATUS
WriteGPIO(
   UInt32 ulMask,
   UInt32 ulData
)
```

Summary

Sets the GPIO lines status

Parameters

ulMask	Mask to select the bits that are to be written. A value of 0 means ignore while a value of 1 means check.
ulData	Data to write the GPIO status. Bit0 and bit1 hold the value to be written to GPIO pins. A value of 1 means high while a value of 0 means low. Bits in input mode are ignored.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulDirection = ((byte)FTDI.FT_GPIO_DIRECTION.OUT << (byte)FTDI.FT_GPIO.GPIO_0) |
                     ((byte)FTDI.FT_GPIO_DIRECTION.OUT << (byte)FTDI.FT_GPIO.GPIO_1);
UInt32 ulMask = (UInt32)FTDI.FT_GPIO_MASK.GPIO_ALL;

ftStatus = d3xxDevice.OpenByIndex(0);

ftStatus = d3xxDevice.EnableGPIO(ulMask, ulDirection);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    Debug.Log("EnableGPIO failed! ftStatus={0}", ftStatus);
    return;
}

for (UInt32 i = 0; i < 4; i++)
{
    ftStatus = d3xxDevice.WriteGPIO((UInt32)FTDI.FT_GPIO_MASK.GPIO_ALL, i);

    Debug.Log("\tWrite: {0:d} (GPIO0={1:d} GPIO1={2:d})", i,
        i & (UInt32)FTDI.FT_GPIO_MASK.GPIO_0,
        (i & (UInt32)FTDI.FT_GPIO_MASK.GPIO_1) >> 1);

    UInt32 ulValue = 0;
    ftStatus = d3xxDevice.ReadGPIO(ref ulValue);

    Debug.Log("\tRead:  {0:d} (GPIO0={1:d} GPIO1={2:d})", ulValue,
        ulValue & (UInt32)FTDI.FT_GPIO_MASK.GPIO_0,
        (ulValue & (UInt32)FTDI.FT_GPIO_MASK.GPIO_1) >> 1);
}

ftStatus = d3xxDevice.Close();
```

3.24 ReadGPIO

```
FT_STATUS
ReadGPIO(
   ref UInt32 ulData
)
```

Summary

Returns the status of GPIO0 and GPIO1

Parameters

ulData	A pointer to data to receive the GPIO status. Bit0 and bit1 reflect the GPIO pin status. A value of 1 means high while a value of 0 means low. Bits in output mode are ignored.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulDirection = ((byte)FTDI.FT_GPIO_DIRECTION.OUT << (byte)FTDI.FT_GPIO.GPIO_0) |
                     ((byte)FTDI.FT_GPIO_DIRECTION.OUT << (byte)FTDI.FT_GPIO.GPIO_1);
UInt32 ulMask = (UInt32)FTDI.FT_GPIO_MASK.GPIO_ALL;

ftStatus = d3xxDevice.OpenByIndex(0);

ftStatus = d3xxDevice.EnableGPIO(ulMask, ulDirection);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    Debug.Log("EnableGPIO failed! ftStatus={0}", ftStatus);
    return TestResult;
}

UInt32 ulValue = 0;
ftStatus = d3xxDevice.ReadGPIO(ref ulValue);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    Debug.Log("ReadGPIO failed! ftStatus={0}", ftStatus);
    return TestResult;
}

Debug.Log("\tRead:  {0:d} (GPIO0={0:d} GPIO1={1:d})", ulValue,
    ulValue & (UInt32)FTDI.FT_GPIO_MASK.GPIO_0,
    (ulValue & (UInt32)FTDI.FT_GPIO_MASK.GPIO_1) >> 1);

ftStatus = d3xxDevice.Close();
```

3.25 SetGPIOPull

```
FT_STATUS
SetGPIOPull(
   UInt32 u32Mask,
   UInt32 u32Pull
)
```

Summary

Set GPIO internal pull resistors. This API is available only for RevB parts.

Parameters

u32Mask	Each bit represents one GPIO pull setting corresponding to GPIO2-GPIO0; Bit 0 corresponds to GPIO0 and bit 2 corresponds to GPIO2. Set the bit to 1 to apply the pull setting in u32Pull and 0 to skip.
u32Pull	Each pair of bits represents one GPIO pull setting. Bit 0 and 1 are used to configure pull settings for GPIO0, bits 2 and 3 for GPIO1 and bits 4 and 5 for GPIO2. 2'b00: 50k ohm pull-down (default) 2'b01: Hi-Z 2'b10: 50k ohm pull-up 2'b11: Hi-Z

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

3.26 SetNotificationCallback

```
FT_STATUS
SetNotificationCallback(
   FTDI.FT_NOTIFICATION_CALLBACK_DATA pCallback,
   IntPtr pvCallbackContext
)
```

Summary

Sets a callback function which will be called when data is available for notification-enabled IN endpoints

Parameters

pCallback	A delegate to a callback function to be called by the library to indicate data status availability in one of the IN endpoints.
pvCallbackContext	A pointer to the user context that will be used when the callback function is called. Can be set to IntPtr.Zero if no context is necessary.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

This function should be called only if the notification message feature is enabled for any IN pipe in the chip configuration. Refer to the bits 2-5 of the *OptionalFeatureSupport* member of the chip configuration structure.

When the chip configuration has the notification message feature turned on for specific IN pipes, the application must not actively call *ReadPipe* or *ReadPipeAsync*. It should register a callback function using *SetNotificationCallback*. It should only call *ReadPipe* or *ReadPipeAsync* when the callback function is called. The registered callback function will be called by the driver once the chip sends a notification (about data availabilitiyavailability on a notification-enabled pipe) via the reserved notification pipe 0x81. The callback function will be called with parameters describing the pipe ID and the data size. Using this information, the application can either read this data (using *ReadPipe*) or flush/ignore this data (using *FlushPipe*).

The notification feature caters for short rare unexpected data (less than 1024 bytes), such as error handling communication. It is not meant for actual data transfers. Actual data transfers are scheduled. Notification messages are only for unscheduled data such as a termination signal from the FPGA. For example, customer can use 2 channel configurationconfigurations (2IN, 2OUT). One IN pipe, 0x82, can be used for camera data transfer. The other IN pipe, 0x83, can be used for a communication channel such as stop signal, start signal, status/error reporting (inform about overflow issue in the FPGA, etcetc.). Notification features can be set on Pipe 0x83. Applications will then actively read on the data pipe 0x82 while passively read on pipe 0x83.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
bool bLoopbackFails = false;
UInt32 loopbytes = 4096;
byte[] writebytes = new byte[loopbytes];
byte[] readbytes = new byte[loopbytes];
var callback = new FTDI.FT_NOTIFICATION_CALLBACK_DATA(NotificationDataCallback);

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

g_Event = new AutoResetEvent(false);

ftStatus = d3xxDevice.SetNotificationCallback(callback, IntPtr.Zero);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

UInt32 byteswritten = 0;
ftStatus = d3xxDevice.WritePipe(0x02, writebytes, loopbytes, ref byteswritten);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

UInt32 totalbytesread = 0;
do
{
    g_Event.WaitOne();

    byte[] tempbuf = new byte[g_NotificationSize];
    UInt32 bytesread = 0;

    ftStatus = d3xxDevice.ReadPipe(g_NotificationPipe, tempbuf, g_NotificationSize,
            ref bytesread);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        break;
    }
    else if (g_NotificationSize == bytesread)
    {
        Array.Copy(tempbuf, 0, readbytes, totalbytesread, bytesread);
        totalbytesread += bytesread;
    }
}
while (totalbytesread < byteswritten);

bool same = writebytes.SequenceEqual(readbytes);
d3xxDevice.ClearNotificationCallback();
ftStatus = d3xxDevice.Close();

public static void NotificationDataCallback(IntPtr pvContext,
    FTDI.FT_NOTIFICATION_TYPE eType, FTDI.FT_NOTIFICATION_CALLBACK_INFO pvInfo)
{
    if (eType == FTDI.FT_NOTIFICATION_TYPE.DATA)
    {
        var Info = (FTDI.FT_NOTIFICATION_INFO_DATA)pvInfo;
        g_NotificationPipe = Info.ucEndpointNo;
        g_NotificationSize = Info.ulDataLength;
        g_Event.Set();
    }
}
```

3.27 ClearNotificationCallback

FT_STATUS
ClearNotificationCallback()

Summary
Clears the notification callback set by SetNotificationCallback.

Parameters
None.

Return Value
FT_OK if successful, otherwise the return value is an FT error code.

3.28 CycleDevicePort

FT_STATUS
CycleDevicePort()

Summary

Power cycles the device port. This causes the device to be re-enumerated by the host system.

Parameters

None.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

The function will cause the device to re-enumerate so the application needs to reopen the driver handle to the device. Re-enumeration process, including driver re-initialization, can take from 200 milliseconds to 2 seconds so applications can either sleep or do some polling on *GetNumberOfDevicesConnected* before calling *OpenByXXX*.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

ftStatus = d3xxDevice.CycleDevicePort();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Allow the device to re-enumerate before re-opening the device by sleeping
// Or better yet poll for GetNumberOfDevicesConnected before reopening the handle
Thread.Sleep(500);
```

3.29 IsDevicePath

```
FT_STATUS
IsDevicePath(
   string szDevicePath
)
```

Summary

Checks if the device path provided is the same as the path of the currently opened device. This is used for hot plugging with *Win32 RegisterDeviceNotification()* function.

Parameters

szDevicePath	A string containing the device path of the device plugged/unplugged as provided in the *WndProc* handler

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

There are two ways to do hot plugging in .NET – using *WMI ManagementEventWatcher* class and using *Win32 RegisterDeviceNotification*. This function can be used for *Win32 RegisterDeviceNotification. RegisterDeviceNotification* allows the application to register for plugging/unplugging of USB devices. In the application's *WndProc* handler, this function can be called to determine if the device unplugged is the same as the device currently opened. This is necessary to differentiate an FT60X device from other D3XX and FT60X devices connected to the machine.

Sample Code

```
public bool IsDevicePathEx(string szDevicePath)
{
    if (!d3xxDevice.IsOpen)
    {
        return false;
    }

    var ftStatus = d3xxDevice.IsDevicePath(szDevicePath);
    if (ftStatus != FTDI.FT_STATUS.FT_OK)
    {
        return false;
    }

    return true;
}

private IntPtr WndProc(IntPtr hwnd, int msg, IntPtr wparam, IntPtr lparam, ref bool handled)
{
    // Refer to Appendix for the implementation of HotPlug2 class
    if (msg == HotPlug2.WM_DEVICECHANGE)
    {
        var obj = (HotPlug2.DevBroadcastDeviceinterface)
            Marshal.PtrToStructure(lparam, typeof(HotPlug2.DevBroadcastDeviceinterface));

        Int32 lOffset = (Marshal.SizeOf(typeof(UInt32)) * 3 + Marshal.SizeOf(typeof(Guid)));
        Int32 lSize = Marshal.ReadInt32(lparam, 0) - lOffset - 4;
        byte[] Name = new byte[lSize];
        for (Int32 i = 0; i < lSize; i++)
        {
            Name[i] = Marshal.ReadByte(lparam, lOffset + i);
        }
```

```
            string devicePath = System.Text.Encoding.Unicode.GetString(Name);

            switch ((Int32)wparam)
            {
                case HotPlug2.DBT_DEVICEARRIVAL:
                    {
                        if (devicePath.Contains(FTDI.FT_GUID.ToString()))
                        {
                            Console.WriteLine("Device attached! [{0}] [{1}]",
                                     devicePath, FTDI.FT_GUID.ToString());

                            ProcessHotPlugDeviceInserted();
                        }

                        break;
                    }
                case HotPlug2.DBT_DEVICEREMOVECOMPLETE:
                    {
                        if (oTaskManager.IsDevicePathEx(devicePath))
                        {
                            Console.WriteLine("Device detached! [{0}]", devicePath);

                            ProcessHotPlugDeviceRemoved();
                        }

                        break;
                    }
            }
        }

        handled = false;
        return IntPtr.Zero;
    }
```

3.30 SetSuspendTimeout

```
FT_STATUS
SetSuspendTimeout(
   UInt32 ulTimeout
)
```

Summary

Configures the idle timeout value for USB selective suspend.

Parameters

ulTimeout	Idle timeout value in seconds. When set to 0, the device will not be suspended even when the device is idle for a long period. When set to non-zero, the device will be suspended after being idle for this timeout value.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

By default, the driver has a suspend feature enabled and set to 10 seconds. If the device is idle for 10 seconds, the driver will suspend the device to save power. This idle timeout value can be modified using this function.

The value set is valid only until the life cycle of the handle. Reopening of the handle will reset the timeout to the default value.

Note that this function is only available starting from version 1.2.0.5 of the D3XX driver and D3XX library. Refer to *DriverVersion* and *LibraryVersion* properties to get the versions of the D3XX driver and D3XX library. Older versions use a non-configurable 10-second value.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulTimeout = 0;


// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");

// Set suspend idle timeout to 1 minute
ulTimeout = 60;
ftStatus = d3xxDevice.SetSuspendTimeout(ulTimeout);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    ftStatus = d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
```

3.31 GetSuspendTimeout

```
FT_STATUS
GetSuspendTimeout(
   ref UInt32 ulTimeout
)
```

Summary

Gets the idle timeout value for USB selective suspend.

Parameters

ulTimeout	Idle timeout value in seconds. Default timeout value is 10 seconds.

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

Note that this function is only available starting from version 1.2.0.5 of the D3XX driver and D3XX library. Refer to *DriverVersion* and *LibraryVersion* properties to get the versions of the D3XX driver and D3XX library. Older versions use a non-configurable 10-second value.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();
UInt32 ulTimeout = 0;


// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

// Get suspend idle timeout
ftStatus = d3xxDevice.GetSuspendTimeout(ref ulTimeout);
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    ftStatus = d3xxDevice.Close();
    return;
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.32 ControlTransfer

```
FT_STATUS
ControlTransfer(
   FT_SETUP_PACKET tSetupPacket,
   byte[] pBuffer,
   UInt32 ulBytesToTransfer,
   ref UInt32  pulBytesTransferrred
)
```

Summary

Transmits control data over the default control endpoint

Parameters

tSetupPacket	The 8-byte setup packet
pBuffer	Pointer to a buffer that contains the data to transfer
ulBytesToTransfer	Length of data to transfer
pulBytesTransferred	Length of data transferred

Return Value

FT_OK if successful, otherwise the return value is an FT error code.

3.33 IsOpen

```
bool IsOpen;
```

Summary

Property that contains information if the class currently has a handle to an opened device. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();


Console.WriteLine("\tIsOpen before Open is {0}", d3xxDevice.IsOpen);

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tIsOpen after Open is {0}", d3xxDevice.IsOpen);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tIsOpen after Close is {0}", d3xxDevice.IsOpen);
```

3.34 IsUSB3

```
bool IsUSB3;
```

Summary

Property that contains information if the device is running in USB 3.0 mode. Device will run in USB 2.0 mode if it is connected using a USB 2.0 cable or connected via a USB 2.0 host controller. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();


// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tIsUSB3 is {0}", d3xxDevice.IsUSB3);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.35 VendorID

UInt16 VendorID;

Summary

Property that contains the VendorID (VID). This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tVendorID  : 0x{0:X4}", d3xxDevice.VendorID);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.36 ProductID

UInt16 ProductID;

Summary

Property that contains the ProductID (PID). This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tProductID : 0x{0:X4}", d3xxDevice.ProductID);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.37 Manufacturer

string Manufacturer;

Summary

Property that contains the Manufacturer string pointed to by the *iManufacturer* field in the USB device descriptor. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tManufacturer        : {0}", d3xxDevice.Manufacturer);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.38 ProductDescription

```
string ProductDescription;
```

Summary

Property that contains the Product Description string pointed to by the *iProduct* field in the USB device descriptor. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tProductDescription  : {0}", d3xxDevice.ProductDescription);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.39 SerialNumber

string SerialNumber;

Summary

Property that contains the Serial Number string pointed to by the *iSerialNumber* field in the USB device descriptor. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tSerialNumber        : {0}", d3xxDevice.SerialNumber);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.40 DriverVersion

UInt32 DriverVersion;

Summary

Property that contains the version number of the D3XX kernel driver. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tDriver Version   : 0x{0:X8}", d3xxDevice.DriverVersion);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.41 LibraryVersion

UInt32 LibraryVersion;

Summary

Property that contains the version number of the D3XX user library. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tLibrary Version  : 0x{0:X8}", d3xxDevice.LibraryVersion);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.42 FirmwareVersion

UInt32 FirmwareVersion;

Summary

Property that contains the version number of the chip. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

Console.WriteLine("\tFirmware Version : 0x{0:X8}", d3xxDevice.FirmwareVersion);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.43 DeviceDescriptor

FTDI.FT_DEVICE_DESCRIPTOR DeviceDescriptor

Summary

Property that contains the USB device descriptor. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

var desc = d3xxDevice.DeviceDescriptor;

Console.WriteLine("\tDEVICE DESCRIPTOR");
Console.WriteLine("\tbLength             : 0x{0:X2}", desc.bLength);
Console.WriteLine("\tbDescriptorType     : 0x{0:X2}", desc.bDescriptorType);
Console.WriteLine("\tbcdUSB              : 0x{0:X4}", desc.bcdUSB
Console.WriteLine("\tbDeviceClass        : 0x{0:X2}", desc.bDeviceClass);
Console.WriteLine("\tbDeviceSubClass     : 0x{0:X2}", desc.bDeviceSubClass);
Console.WriteLine("\tbDeviceProtocol     : 0x{0:X2}", desc.bDeviceProtocol);
Console.WriteLine("\tbMaxPacketSize0     : 0x{0:X2}", desc.bMaxPacketSize0);
Console.WriteLine("\tidVendor            : 0x{0:X4}", desc.idVendor);
Console.WriteLine("\tidProduct           : 0x{0:X4}", desc.idProduct);
Console.WriteLine("\tbcdDevice           : 0x{0:X4}", desc.bcdDevice);
Console.WriteLine("\tiManufacturer       : 0x{0:X2}", desc.iManufacturer);
Console.WriteLine("\tiProduct            : 0x{0:X2}", desc.iProduct);
Console.WriteLine("\tiSerialNumber       : 0x{0:X2}", desc.iSerialNumber);
Console.WriteLine("\tbNumConfigurations  : 0x{0:X2}", desc.bNumConfigurations);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.44 ConfigurationDescriptor

FTDI.FT_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor

Summary

Property that contains the USB configuration descriptor. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

var desc = d3xxDevice.ConfigurationDescriptor;

Console.WriteLine("\tCONFIGURATION DESCRIPTOR");
Console.WriteLine("\tbLength             : 0x{0:X2}", desc.bLength,);
Console.WriteLine("\tbDescriptorType     : 0x{0:X2}", desc.bDescriptorType);
Console.WriteLine("\twTotalLength        : 0x{0:X4}", desc.wTotalLength);
Console.WriteLine("\tbNumInterfaces      : 0x{0:X2}", desc.bNumInterfaces);
Console.WriteLine("\tbConfigurationValue : 0x{0:X2}", desc.bConfigurationValue);
Console.WriteLine("\tiConfiguration      : 0x{0:X2}", desc.iConfiguration);
Console.WriteLine("\tbmAttributes        : 0x{0:X2}", desc.bmAttributes);
Console.WriteLine("\tMaxPower            : 0x{0:X2}", desc.MaxPower);

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.45 InterfaceDescriptor

List<FTDI.FT_INTERFACE_DESCRIPTOR> InterfaceDescriptor

Summary

Property that contains the USB interface descriptors. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var desc in d3xxDevice.InterfaceDescriptors)
{
    Console.WriteLine("\tINTERFACE DESCRIPTOR");
    Console.WriteLine("\tbLength            : 0x{0:X2}", desc.bLength);
    Console.WriteLine("\tbDescriptorType    : 0x{0:X2}", desc.bDescriptorType);
    Console.WriteLine("\tbInterfaceNumber   : 0x{0:X2}", desc.bInterfaceNumber);
    Console.WriteLine("\tbAlternateSetting  : 0x{0:X2}", desc.bAlternateSetting);
    Console.WriteLine("\tbNumEndpoints      : 0x{0:X2}", desc.bNumEndpoints);
    Console.WriteLine("\tbInterfaceClass    : 0x{0:X2}", desc.bInterfaceClass);
    Console.WriteLine("\tbInterfaceSubClass : 0x{0:X2}", desc.bInterfaceSubClass);
    Console.WriteLine("\tbInterfaceProtocol : 0x{0:X2}", desc.bInterfaceProtocol);
    Console.WriteLine("\tiInterface         : 0x{0:X2}", desc.iInterface);
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.46 ReservedPipeInformation

List<FTDI.FT_PIPE_INFORMATION> ReservedPipeInformation

Summary

Property that contains the pipe information similar to the USB endpoint descriptor of the reserved pipes. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var desc in d3xxDevice.ReservedPipeInformation)
{
    Console.WriteLine("\tPIPE INFORMATION");
    Console.WriteLine("\tPipeType           : {0:d} ({1})", desc.PipeType,
                                                  desc.PipeType.ToString());
    Console.WriteLine("\tPipeId             : 0x{0:X2}", desc.PipeId);
    Console.WriteLine("\tMaximumPacketSize  : 0x{0:X4}", oDescriptor.MaximumPacketSize);
    Console.WriteLine("\tInterval           : 0x{0:X2}", oDescriptor.Interval);
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

3.47 DataPipeInformation

List<FTDI.FT_PIPE_INFORMATION> DataPipeInformation

Summary

Property that contains the pipe information similar to the USB endpoint descriptor of the data pipes. This property is implicitly queried during the opening of the device.

Sample Code

```
FTDI.FT_STATUS ftStatus = FTDI.FT_STATUS.FT_OK;
FTDI d3xxDevice = new FTDI();

// Use GetDeviceInfoList to get the actual Serial Number of the device
ftStatus = d3xxDevice.OpenBySerialNumber("123456789012345");
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}

foreach (var desc in d3xxDevice.DataPipeInformation)
{
    Console.WriteLine("\tPIPE INFORMATION");
    Console.WriteLine("\tPipeType           : {0:d} ({1})", desc.PipeType,
                                                  desc.PipeType.ToString());
    Console.WriteLine("\tPipeId             : 0x{0:X2}", desc.PipeId);
    Console.WriteLine("\tMaximumPacketSize  : 0x{0:X4}", oDescriptor.MaximumPacketSize);
    Console.WriteLine("\tInterval           : 0x{0:X2}", oDescriptor.Interval);
}

ftStatus = d3xxDevice.Close();
if (ftStatus != FTDI.FT_STATUS.FT_OK)
{
    return;
}
```

4 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.

Appendix A – References

Major differences with D2XX

Interface-Pipe Design
In D2XX, chips can only report 1 channel (1 OUT, 1 IN) for each interface. So *FT_Write* and *FT_Read* do not need to specify which pipe to use. In D3XX, FT60x chips report multiple channels on a single interface. To send data to a specific pipe, it is necessary to specify the pipe ID, ucPipeID, in *FT_WritePipe* and *FT_ReadPipe*.

Protocol Design
D2XX uses polling in the kernel-mode driver to read data from the bus. Users can call some functions (e.g. *FT_GetQueueStatus*) to query if there is data available in the pipe and how much data is available before actually trying to call *FT_Read*. Polling on high bandwidth transfers causes high CPU load and is not efficient so D3XX implements an alternative protocol by scheduling the transfers using session commands instead of polling. When a user calls *FT_ReadPipe*, it first informs the chip it wants a specific number of bytes so the chip will only provide whatever was requested.

Asynchronous Transfer Design
The LPOVERLAPPED parameter for asynchronous transfers is a well-known concept that is present in Win32 API *WriteFile* and *ReadFile*, as well as in WinUSB API *WinUsb_WritePipe* and *WinUsb_ReadPipe*. This parameter allows users to send multiple asynchronous read/write requests to a specific pipe. D2XX does not provide this parameter because it implements polling for *FT_Read*, so in a sense *FT_Read* is asynchronous in nature but *FT_Write* is not. Since D3XX does not do polling, it is necessary to provide this parameter to improve latency between each packet. Users can send multiple asynchronous transfers on a specific pipe – such that while the application is processing one buffer, another request is already ongoing, thereby improving the gap between each request.

Asynchronous Transfer	D3XX	D2XX
Write	YES, via API	NO
Read	YES, via API	YES, via polling

Table 8 Asynchronous Transfer Support in D3XX and D2XX

Streaming Transfer Design
In addition, D3XX provides the *FT_SetStreamPipe* function as a supplement to the *FT_WritePipe* and *FT_ReadPipe*, primarily for performance improvements. This informs the chip that the host will be reading or writing a fixed number of bytes continuously on the specified pipe. When this is used, *FT_WritePipe* and *FT_ReadPipe* no longer sends a session command to the chip because *FT_SetStreamPipe* already informed the chip how much data to request. This is a feature that can be used together with asynchronous transfers to fully maximize data throughput.

Support for multiple devices

To support multiple devices, customers must change the String Descriptors in the USB Device Descriptor (Manufacturer, Product Description and Serial Number) using the FT60x Chip Configuration Programmer.

The Manufacturer name must uniquely identify the manufacturer from other manufacturers. The Product Description must uniquely identify the product name from other product names of the manufacturer. The Serial Number must uniquely identify the device from other devices with the same product name and manufacturer name.

USB String Descriptor	Max ASCII characters	Max Unicode characters	Character restriction
Manufacturer	15	30	Printable
Description	31	62	Printable
Serial Number	15	30	Alphanumeric

Table 9 USB String Descriptor Restrictions

Support for hot plugging

The .NET framework provides a way for .NET applications to register notifications for USB device plugin and plugout via the *WqlEventQuery* and *ManagementEventWatcher* classes in the *System.Management* namespace and assembly.

```
public class HotPlug
{
    public delegate void DelegateHotPlug(object sender, EventArrivedEventArgs e);

    public void Register(
        DelegateHotPlug DeviceInsertedEvent, DelegateHotPlug DeviceRemovedEvent)
    {
        insertQuery = new WqlEventQuery(
            "SELECT * FROM __InstanceCreationEvent " +
            "WITHIN 2 " +
            "WHERE TargetInstance ISA 'Win32_USBHub'"
            );
        insertWatcher = new ManagementEventWatcher(insertQuery);
        insertWatcher.EventArrived += new EventArrivedEventHandler(DeviceInsertedEvent);
        insertWatcher.Start();

        removeQuery = new WqlEventQuery(
            "SELECT * FROM __InstanceDeletionEvent " +
            "WITHIN 2 " +
            "WHERE TargetInstance ISA 'Win32_USBHub'"
            );
        removeWatcher = new ManagementEventWatcher(removeQuery);
        removeWatcher.EventArrived += new EventArrivedEventHandler(DeviceRemovedEvent);
        removeWatcher.Start();
    }

    public void Unregister()
    {
        insertWatcher.Stop();
        insertWatcher.Dispose();
        removeWatcher.Stop();
        removeWatcher.Dispose();
    }

    private WqlEventQuery insertQuery = null;
    private WqlEventQuery removeQuery = null;
```

```
    private ManagementEventWatcher insertWatcher = null;
    private ManagementEventWatcher removeWatcher = null;
}

private void HotPlugDeviceInserted(object sender, EventArrivedEventArgs e)
{
    var instance = (ManagementBaseObject)e.NewEvent["TargetInstance"];
    foreach (var property in instance.Properties)
        if (property.Name.Equals("Caption"))
            if (!property.Value.Equals("USB Composite Device"))
                // Query the D3XX devices connected to the system
                // if no device is currently opened
                break;
}

private void HotPlugDeviceRemoved(object sender, EventArrivedEventArgs e)
{
    var instance = (ManagementBaseObject)e.NewEvent["TargetInstance"];
    foreach (var property in instance.Properties)
        if (property.Name.Equals("DeviceID"))
            string strValue = (string)property.Value;
            break;
}
```

Aside from WMI *ManagementEventWatcher,* the *Win32 RegisterDeviceNotification* function from the operating system's *user32.dll* can also be used for hot plugging.

```
public class HotPlug2
{
    public static void Register(IntPtr hWindow, Guid tGuid)
    {
        if (hNotification == IntPtr.Zero)
        {
            var dbi = new DevBroadcastDeviceinterface();
            dbi.DeviceType = DBT_DEVTYP_DEVICEINTERFACE;
            dbi.ClassGuid = tGuid;
            dbi.Size = Marshal.SizeOf(dbi);
            IntPtr pBuffer = Marshal.AllocHGlobal(dbi.Size);
            Marshal.StructureToPtr(dbi, pBuffer, true);

            hNotification = RegisterDeviceNotification(hWindow, pBuffer, 0);
        }
    }

    public static void Unregister()
    {
        if (hNotification != IntPtr.Zero)
        {
            UnregisterDeviceNotification(hNotification);
            hNotification = IntPtr.Zero;
        }
    }

    [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    private static extern IntPtr RegisterDeviceNotification(
        IntPtr recipient, IntPtr notificationFilter, int flags);
    [DllImport("user32.dll")]
    private static extern bool UnregisterDeviceNotification(IntPtr handle);

    [StructLayout(LayoutKind.Sequential, Size = 284)]
    public struct DevBroadcastDeviceinterface
    {
        internal int Size;
        internal int DeviceType;
        internal int Reserved;
        internal Guid ClassGuid;
        internal short Name;
    }
```

```
        public const int DBT_DEVICEARRIVAL = 0x8000;        // system detected a new device
        public const int DBT_DEVICEREMOVECOMPLETE = 0x8004; // device is gone
        public const int WM_DEVICECHANGE = 0x0219;          // device change event
        private const int DBT_DEVTYP_DEVICEINTERFACE = 5;
        private static IntPtr hNotification = IntPtr.Zero;
    }

    protected override void OnSourceInitialized(EventArgs e)
    {
        base.OnSourceInitialized(e);

        hWindowSource = PresentationSource.FromVisual(this) as HwndSource;
        hWindowSource.AddHook(WndProc);

        HotPlug2.Register(hWindowSource.Handle, FTDI.FT_GUID);
    }
```

Support for .NET framework versions

The D3XX .NET library and demo applications target .NET Framework versions 4.5 and higher. Visual Studio 2013, which was used to develop these projects, uses .NET Framework version 4.5 by default.

.NET Framework Version	Visual Studio Version	Release Date
2.0	2005	Nov 2005
3.5	2008	Nov 2007
4.0	2010	Apr 2010
4.5	2012	Aug 2012
4.5.1	2013	Oct 2013
4.6	2015	Jul 2015
4.6.1	2015	Nov 2015

Table 10 .NET Framework History

Since the library and demo applications are open-source, customers can support compatibility to .NET Framework version 4.0 and lower by updating the Target framework option in the Visual Studio project settings before recompiling.

Achieving maximum performance

In the FT60X, the data throughput varies for each channel configuration because of the allocation of EPC burst size and FIFO ping/pong request size. These values are fixed and cannot be configured by the customer. Below are the tables illustrating the values used

Channel Configuration	Burst Size
4 channels	4
2 channels	8
1 channel	16
1 channel with 1 OUT pipe only	16
1 channel with 1 IN pipe only	16

Table 11 FT60X EPC Burst Size

Channel Configuration	FIFO Size
4 channels	1024
2 channels	2048
1 channel	4096
1 channel with 1 OUT pipe only	8192
1 channel with 1 IN pipe only	8192

Table 12 FT60X FIFO Ping/Pong Request Size

In order to maximize performance, FTDI advices customers to consider the following in the design of their FPGA and host-side application for the FT60X.

FPGA

1. Use any of the three 1 channel variants instead of 2 channels and 4 channels.
2. Use the exact FIFO size when sending data to the FIFO.

Application

1. Use multiple asynchronous transfers and enable streaming mode.
2. Use a large buffer when transmitting data.

Example

Below is a sample design for a QuadHD XRGB8888 Camera Video application that maximizes the performance of the D3XX and FT60X.

1. Chip is configured to 1 channel with 1 IN pipe only.
2. The application opens the device using *OpenByXXX* variants and then enables streaming mode using *SetStreamMode*.
3. The application initially sends 3 asynchronous requests for 3 frame buffers of size 2560x1440x4 = 14,745,600 bytes each using *ReadPipeAsync*. The application can use any queue size other than 3, but the buffer size should be 1 frame bytes.
   The driver will queue the 3 asynchronous requests and process them sequentially.
4. The chip will request a total of 14,745,600 bytes from the FIFO in 4KB segments.
   The chip will request 4KB from Ping and then 4KB from Pong until 14,745,600 bytes has been transmitted. Since 14,745,600 bytes is not divisible by 4KB, then FPGA will give less than 4KB to the FIFO on the last segment.
5. The driver completes the request for 1 frame and the application call to *WaitAsync* unblocks. It renders the frame and immediately resends the request again to ensure the queue is full. Note that queue size is set to 3 in this example.
6. The process is repeated until the user stops the transfer in which case it will call *AbortPipe* to cancel all outstanding requests in the driver before calling *ClearStreamMode* and *Close*.

A data streamer demo application is available in the website for reference purposes.

Code References

```
/// <summary>
/// Status values for FTDI devices.
/// </summary>
public enum FT_STATUS
{
    /// <summary>
    /// Status OK
    /// </summary>
    FT_OK = 0,
    /// <summary>
    /// The device handle is invalid
    /// </summary>
    FT_INVALID_HANDLE,
    /// <summary>
    /// Device not found
    /// </summary>
    FT_DEVICE_NOT_FOUND,
    /// <summary>
    /// Device is not open
    /// </summary>
    FT_DEVICE_NOT_OPENED,
    /// <summary>
    /// IO error
    /// </summary>
    FT_IO_ERROR,
    /// <summary>
    /// Insufficient resources
    /// </summary>
    FT_INSUFFICIENT_RESOURCES,
    /// <summary>
    /// A parameter was invalid
    /// </summary>
    FT_INVALID_PARAMETER,
    /// <summary>
    /// The requested baud rate is invalid
    /// </summary>
    FT_INVALID_BAUD_RATE,
    /// <summary>
    /// Device not opened for erase
    /// </summary>
    FT_DEVICE_NOT_OPENED_FOR_ERASE,
    /// <summary>
    /// Device not poened for write
    /// </summary>
    FT_DEVICE_NOT_OPENED_FOR_WRITE,
    /// <summary>
    /// Failed to write to device
    /// </summary>
    FT_FAILED_TO_WRITE_DEVICE,
    /// <summary>
    /// Failed to read the device EEPROM
    /// </summary>
    FT_EEPROM_READ_FAILED,
    /// <summary>
    /// Failed to write the device EEPROM
    /// </summary>
    FT_EEPROM_WRITE_FAILED,
    /// <summary>
    /// Failed to erase the device EEPROM
    /// </summary>
    FT_EEPROM_ERASE_FAILED,
    /// <summary>
    /// An EEPROM is not fitted to the device
    /// </summary>
    FT_EEPROM_NOT_PRESENT,
    /// <summary>
    /// Device EEPROM is blank
    /// </summary>
```

```
        FT_EEPROM_NOT_PROGRAMMED,
        /// <summary>
        /// Invalid arguments
        /// </summary>
        FT_INVALID_ARGS,
        /// <summary>
        /// Functionality not supported
        /// </summary>
        FT_NOT_SUPPORTED,
        /// <summary>
        /// No more items
        /// </summary>
        FT_NO_MORE_ITEMS,
        /// <summary>
        /// Timed out
        /// </summary>
        FT_TIMEOUT,
        /// <summary>
        /// Operation aborted
        /// </summary>
        FT_OPERATION_ABORTED,
        /// <summary>
        /// Reserved pipe
        /// </summary>
        FT_RESERVED_PIPE,
        /// <summary>
        /// Invalid control request direction
        /// </summary>
        FT_INVALID_CONTROL_REQUEST_DIRECTION,
        /// <summary>
        /// Invalid control request type
        /// </summary>
        FT_INVALID_CONTROL_REQUEST_TYPE,
        /// <summary>
        /// IO pending
        /// </summary>
        FT_IO_PENDING,
        /// <summary>
        /// IO incomplete
        /// </summary>
        FT_IO_INCOMPLETE,
        /// <summary>
        /// End of file
        /// </summary>
        FT_HANDLE_EOF,
        /// <summary>
        /// Busy
        /// </summary>
        FT_BUSY,
        /// <summary>
        /// No system resources
        /// </summary>
        FT_NO_SYSTEM_RESOURCES,
        /// <summary>
        /// Device not ready
        /// </summary>
        FT_DEVICE_LIST_NOT_READY,
        /// <summary>
        /// Device not connected
        /// </summary>
        FT_DEVICE_NOT_CONNECTED,
        /// <summary>
        /// Incorrect device path
        /// </summary>
        FT_INCORRECT_DEVICE_PATH,
        /// <summary>
        /// An other error has occurred
        /// </summary>
        FT_OTHER_ERROR
    };
```

```
/// <summary>
/// Device type identifiers for FT_DEVICE_INFO
/// </summary>
public enum FT_DEVICE
{
    /// <summary>
    /// Unknown device
    /// </summary>
    FT_DEVICE_UNKNOWN = 3,
    /// <summary>
    /// FT600 device
    /// </summary>
    FT_DEVICE_600 = 600,
    /// <summary>
    /// FT601 device
    /// </summary>
    FT_DEVICE_601 = 601,
};

/// <summary>
/// USB Pipe types
/// </summary>
public enum FT_PIPE_TYPE
{
    /// <summary>
    /// USB control pipe
    /// </summary>
    CONTROL,
    /// <summary>
    /// USB isochronous pipe
    /// </summary>
    ISOCHRONOUS,
    /// <summary>
    /// USB bulk pipe
    /// </summary>
    BULK,
    /// <summary>
    /// USB interrupt pipe
    /// </summary>
    INTERRUPT
};

/// <summary>
/// USB Descriptor types
/// </summary>
public enum FT_DESCRIPTOR_TYPE
{
    /// <summary>
    /// USB device descriptor
    /// </summary>
    DEVICE_DESCRIPTOR = 1,
    /// <summary>
    /// USB configuration descriptor
    /// </summary>
    CONFIGURATION_DESCRIPTOR,
    /// <summary>
    /// USB string descriptor
    /// </summary>
    STRING_DESCRIPTOR,
    /// <summary>
    /// USB interface descriptor
    /// </summary>
    INTERFACE_DESCRIPTOR,
    /// <summary>
    /// USB endpoint descriptor
    /// </summary>
    ENDPOINT_DESCRIPTOR,
};
```

```
/// <summary>
/// Notification types
/// </summary>
public enum FT_NOTIFICATION_TYPE
{
    /// <summary>
    /// Data notification
    /// </summary>
    DATA,
    /// <summary>
    /// GPIO notification
    /// </summary>
    GPIO
};

/// <summary>
/// GPIO lines
/// </summary>
public enum FT_GPIO
{
    /// <summary>
    /// GPIO 0
    /// </summary>
    GPIO_0,
    /// <summary>
    /// GPIO 1
    /// </summary>
    GPIO_1,
    /// <summary>
    /// GPIO 2
    /// </summary>
    GPIO_2,
    /// <summary>
    /// GPIO 3
    /// </summary>
    GPIO_3,
    /// <summary>
    /// GPIO 4
    /// </summary>
    GPIO_4,
    /// <summary>
    /// GPIO 5
    /// </summary>
    GPIO_5,
    /// <summary>
    /// GPIO 6
    /// </summary>
    GPIO_6,
    /// <summary>
    /// GPIO 7
    /// </summary>
    GPIO_7,
}

/// <summary>
/// GPIO mask
/// </summary>
public enum FT_GPIO_MASK
{
    /// <summary>
    /// GPIO 0
    /// </summary>
    GPIO_0 = 1,
    /// <summary>
    /// GPIO 1
    /// </summary>
    GPIO_1 = 2,
```

```
        /// <summary>
        /// GPIO ALL
        /// </summary>
        GPIO_ALL = 3,
    }

    /// <summary>
    /// GPIO directions
    /// </summary>
    public enum FT_GPIO_DIRECTION
    {
        /// <summary>
        /// Direction IN
        /// </summary>
        IN,
        /// <summary>
        /// Direction OUT
        /// </summary>
        OUT
    }

    /// <summary>
    /// GPIO values
    /// </summary>
    public enum FT_GPIO_VALUE
    {
        /// <summary>
        /// Value LOW
        /// </summary>
        LOW,
        /// <summary>
        /// Value HIGH
        /// </summary>
        HIGH
    }

    /// <summary>
    /// Battery Charging power source types
    /// </summary>
    public enum FT_BATTERY_CHARGING_TYPE
    {
        /// <summary>
        /// Battery Charging Not Detected/OFF
        /// </summary>
        OFF,
        /// <summary>
        /// Standard Downstream Port
        /// </summary>
        SDP,
        /// <summary>
        /// Charging Downstream Port
        /// </summary>
        CDP,
        /// <summary>
        /// Dedicated Charging Port
        /// </summary>
        DCP,
    };

    /// <summary>
    /// FIFO clock speeds for FT60X chip configuration
    /// </summary>
    public enum FT_60XCONFIGURATION_FIFO_CLK
    {
        /// <summary>
        /// 100 MHz
        /// </summary>
        CLK_100_MHZ,
        /// <summary>
        /// 66 MHz
```

```
        /// </summary>
        CLK_66_MHZ,
        CLK_COUNT
    };

    /// <summary>
    /// FIFO modes for FT60X chip configuration
    /// </summary>
    public enum FT_60XCONFIGURATION_FIFO_MODE
    {
        /// <summary>
        /// 245 mode - has maximum of 1 OUT and 1 IN
        /// </summary>
        MODE_245,
        /// <summary>
        /// 600 mode - can have different channel configuration
        /// </summary>
        MODE_600,
        MODE_COUNT
    };

    /// <summary>
    /// Channel configurations for FT60X chip configuration
    /// </summary>
    public enum FT_60XCONFIGURATION_CHANNEL_CONFIG
    {
        /// <summary>
        /// 4 OUT and 4 IN pipes
        /// </summary>
        FOUR,
        /// <summary>
        /// 2 OUT and 2 IN pipes
        /// </summary>
        TWO,
        /// <summary>
        /// 1 OUT and 1 IN pipes
        /// </summary>
        ONE,
        /// <summary>
        /// 1 OUT pipe only
        /// </summary>
        ONE_OUTPIPE,
        /// <summary>
        /// 1 IN pipe only
        /// </summary>
        ONE_INPIPE,
        /// <summary>
        /// number of channel configurations
        /// </summary>
        COUNT,
    };

    /// <summary>
    /// Optional feature support for FT60X chip configuration
    /// </summary>
    public enum FT_60XCONFIGURATION_OPTIONAL_FEATURE
    {
        /// <summary>
        /// Disable all
        /// </summary>
        DISABLEALL = 0,
        /// <summary>
        /// Enable battery charging
        /// </summary>
        ENABLEBATTERYCHARGING = (0x1 << 0),
        /// <summary>
        /// Disable cancel session underrun
        /// </summary>
        DISABLECANCELSESSIONUNDERRUN = (0x1 << 1),
        /// <summary>
```

```
        /// Enable notification on channel 1 IN pipe
        /// </summary>
        ENABLENOTIFICATIONMESSAGE_INCH1 = (0x1 << 2),
        /// <summary>
        /// Enable notification on channel 2 IN pipe
        /// </summary>
        ENABLENOTIFICATIONMESSAGE_INCH2 = (0x1 << 3),
        /// <summary>
        /// Enable notification on channel 3 IN pipe
        /// </summary>
        ENABLENOTIFICATIONMESSAGE_INCH3 = (0x1 << 4),
        /// <summary>
        /// Enable notification on channel 4 IN pipe
        /// </summary>
        ENABLENOTIFICATIONMESSAGE_INCH4 = (0x1 << 5),
        /// <summary>
        /// Enable notification on all channel IN pipes
        /// </summary>
        ENABLENOTIFICATIONMESSAGE_INCHALL = (0xF << 2),
        /// <summary>
        /// Disable underrun on channel 1 IN pipe
        /// </summary>
        DISABLEUNDERRUN_INCH1 = (0x1 << 6),
        /// <summary>
        /// Disable underrun on channel 2 IN pipe
        /// </summary>
        DISABLEUNDERRUN_INCH2 = (0x1 << 7),
        /// <summary>
        /// Disable underrun on channel 3 IN pipe
        /// </summary>
        DISABLEUNDERRUN_INCH3 = (0x1 << 8),
        /// <summary>
        /// Disable underrun on channel 4 IN pipe
        /// </summary>
        DISABLEUNDERRUN_INCH4 = (0x1 << 9),
        /// <summary>
        /// Disable underrun on all channel IN pipes
        /// </summary>
        DISABLEUNDERRUN_INCHALL = (0xF << 6),
        /// <summary>
        /// Enable all
        /// </summary>
        ENABLEALL = 0xFFFF,
    };

    /// <summary>
    /// Bit flags for FlashEEPROMDetection read-only field of FT60X chip configuration
    /// </summary>
    public enum FT_60XCONFIGURATION_FLASH_ROM_BIT
    {
        /// <summary>
        /// ROM or Flash memory
        /// </summary>
        ROM,
        /// <summary>
        /// ROM or Flash memory exists
        /// </summary>
        MEMORY_NOTEXIST,
        /// <summary>
        /// Custom configuration is invalid
        /// </summary>
        CUSTOMDATA_INVALID,
        /// <summary>
        /// Custom configuration checksum is invalid
        /// </summary>
        CUSTOMDATACHKSUM_INVALID,
        /// <summary>
        /// Custom or default configuration values
        /// </summary>
        CUSTOM,
```

```
        /// <summary>
        /// GPIO is used as input values
        /// </summary>
        GPIO_INPUT,
        /// <summary>
        /// GPIO 0 is low or high
        /// </summary>
        GPIO_0,
        /// <summary>
        /// GPIO 1 is low or high
        /// </summary>
        GPIO_1,
    };

    /// <summary>
    /// Bit offsets for BatteryChargingGPIOConfig field of FT60X chip configuration
    /// </summary>
    public const UInt32 FT_60XCONFIGURATION_BATCHG_BIT_OFFSET_DEF = 0; // Bit 0 and Bit 1
    public const UInt32 FT_60XCONFIGURATION_BATCHG_BIT_OFFSET_SDP = 2; // Bit 2 and Bit 3
    public const UInt32 FT_60XCONFIGURATION_BATCHG_BIT_OFFSET_CDP = 4; // Bit 4 and Bit 5
    public const UInt32 FT_60XCONFIGURATION_BATCHG_BIT_OFFSET_DCP = 6; // Bit 6 and Bit 7
    public const UInt32 FT_60XCONFIGURATION_BATCHG_BIT_MASK = 3;       // 2 bits

    /// <summary>
    /// Default values for all fields of FT60X chip configuration
    /// </summary>
    public const UInt16 FT_60XCONFIGURATION_DEFAULT_VENDORID = 0x0403;
    public const UInt16 FT_60XCONFIGURATION_DEFAULT_PRODUCTID_600 = 0x601E;
    public const UInt16 FT_60XCONFIGURATION_DEFAULT_PRODUCTID_601 = 0x601F;
    public const byte FT_60XCONFIGURATION_DEFAULT_POWERATTRIBUTES = 0xE0;
    public const byte FT_60XCONFIGURATION_DEFAULT_POWERCONSUMPTION = 0x60;
    public const byte FT_60XCONFIGURATION_DEFAULT_FIFOCLOCK
            = (byte)FT_60XCONFIGURATION_FIFO_CLK.CLK_100_MHZ;
    public const byte FT_60XCONFIGURATION_DEFAULT_FIFOMODE
            = (byte)FT_60XCONFIGURATION_FIFO_MODE.MODE_600;
    public const byte FT_60XCONFIGURATION_DEFAULT_CHANNELCONFIG
            = (byte)FT_60XCONFIGURATION_CHANNEL_CONFIG.FOUR;
    public const UInt16 FT_60XCONFIGURATION_DEFAULT_OPTIONALFEATURE
            = (UInt16)FT_60XCONFIGURATION_OPTIONAL_FEATURE.DISABLEALL;
    public const byte FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING = 0xE4;
    public const byte FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING_TYPE_DCP = 0x3;
    public const byte FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING_TYPE_CDP = 0x2;
    public const byte FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING_TYPE_SDP = 0x1;
    public const byte FT_60XCONFIGURATION_DEFAULT_BATTERYCHARGING_TYPE_OFF = 0x0;
    public const byte FT_60XCONFIGURATION_DEFAULT_FLASHDETECTION = 0x0;
    public const UInt32 FT_60XCONFIGURATION_DEFAULT_MSIOCONTROL = 0x10800;
    public const UInt32 FT_60XCONFIGURATION_DEFAULT_GPIOCONTROL = 0x0;
    public const UInt32 FT_60XCONFIGURATION_SIZE = 152;


    //////////////////////////////////////////////////////////////////////////
    // DEVICE INFO
    //////////////////////////////////////////////////////////////////////////

    /// <summary>
    /// Device information to contain information regarding D3XX devices attached to the system
    /// </summary>
    [StructLayout(LayoutKind.Explicit, Size = 68, Pack = 1, CharSet = CharSet.Ansi)]
    public class FT_DEVICE_INFO
    {
        /// <summary>
        /// Bit flags for USB 3 or USB 2 connection, etc
        /// </summary>
        [MarshalAs(UnmanagedType.U4)]
        [FieldOffset(0)]
        public UInt32 Flags;

        /// <summary>
        /// Device type as indicated by FT_DEVICE
        /// </summary>
```

```
        [MarshalAs(UnmanagedType.U4)]
        [FieldOffset(4)]
        public FT_DEVICE Type;

        /// <summary>
        /// Concatenated VID and PID
        /// </summary>
        [MarshalAs(UnmanagedType.U4)]
        [FieldOffset(8)]
        public UInt32 ID;

        /// <summary>
        /// Location identifier
        /// </summary>
        [MarshalAs(UnmanagedType.U4)]
        [FieldOffset(12)]
        public UInt32 LocId;

        /// <summary>
        /// Serial number as indicated in the USB device descriptor
        /// </summary>
        [MarshalAs(UnmanagedType.ByValArray, SizeConst = 16)]
        [FieldOffset(16)]
        public byte[] SerialNumber;

        /// <summary>
        /// Product description as indicated in the USB device descriptor
        /// </summary>
        [MarshalAs(UnmanagedType.ByValArray, SizeConst = 32)]
        [FieldOffset(32)]
        public byte[] Description;

        /// <summary>
        /// Handle of the device if opened
        /// </summary>
        [MarshalAs(UnmanagedType.U4)]
        [FieldOffset(64)]
        public UInt32 ftHandle;
    };


    //****************************************************************************
    // DESCRIPTORS
    //****************************************************************************

    /// <summary>
    /// USB descriptor
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public abstract class FT_DESCRIPTOR
    {
        /// <summary>
        /// Descriptor length
        /// </summary>
        public byte bLength;
        /// <summary>
        /// Descriptor type
        /// </summary>
        public byte bDescriptorType;
    }

    /// <summary>
    /// USB device descriptor
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public class FT_DEVICE_DESCRIPTOR : FT_DESCRIPTOR
    {
        /// <summary>
        /// USB Specification Number which device complies too
        /// </summary>
```

```
        public UInt16 bcdUSB;
        /// <summary>
        /// Class Code (Assigned by USB Org)
        /// </summary>
        public byte bDeviceClass;
        /// <summary>
        /// Subclass Code (Assigned by USB Org)
        /// </summary>
        public byte bDeviceSubClass;
        /// <summary>
        /// Protocol Code (Assigned by USB Org)
        /// </summary>
        public byte bDeviceProtocol;
        /// <summary>
        /// Maximum Packet Size for Zero Endpoint
        /// </summary>
        public byte bMaxPacketSize0;
        /// <summary>
        /// Vendor ID (Assigned by USB Org)
        /// </summary>
        public UInt16 idVendor;
        /// <summary>
        /// Product ID (Assigned by Manufacturer)
        /// </summary>
        public UInt16 idProduct;
        /// <summary>
        /// Device Release Number
        /// </summary>
        public UInt16 bcdDevice;
        /// <summary>
        /// Index of Manufacturer String Descriptor
        /// </summary>
        public byte iManufacturer;
        /// <summary>
        /// Index of Product String Descriptor
        /// </summary>
        public byte iProduct;
        /// <summary>
        /// Index of Serial Number String Descriptor
        /// </summary>
        public byte iSerialNumber;
        /// <summary>
        /// Number of Possible Configurations
        /// </summary>
        public byte bNumConfigurations;
    };

    /// <summary>
    /// USB configuration descriptor
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public class FT_CONFIGURATION_DESCRIPTOR : FT_DESCRIPTOR
    {
        /// <summary>
        /// Total length in bytes of data returned
        /// </summary>
        public UInt16 wTotalLength;
        /// <summary>
        /// Number of Interfaces
        /// </summary>
        public byte bNumInterfaces;
        /// <summary>
        /// Value to use as an argument to select this configuration
        /// </summary>
        public byte bConfigurationValue;
        /// <summary>
        /// Index of String Descriptor describing this configuration
        /// </summary>
        public byte iConfiguration;
        /// <summary>
```

```
        /// Self Powered, Remote Wakeup
        /// </summary>
        public byte bmAttributes;
        /// <summary>
        /// Maximum Power Consumption
        /// </summary>
        public byte MaxPower;
    };

    /// <summary>
    /// USB interface descriptor
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public class FT_INTERFACE_DESCRIPTOR : FT_DESCRIPTOR
    {
        /// <summary>
        /// Number of Interface
        /// </summary>
        public byte bInterfaceNumber;
        /// <summary>
        /// Value used to select alternative setting
        /// </summary>
        public byte bAlternateSetting;
        /// <summary>
        /// Number of Endpoints used for this interface
        /// </summary>
        public byte bNumEndpoints;
        /// <summary>
        /// Class Code (Assigned by USB Org)
        /// </summary>
        public byte bInterfaceClass;
        /// <summary>
        /// Subclass Code (Assigned by USB Org)
        /// </summary>
        public byte bInterfaceSubClass;
        /// <summary>
        /// Protocol Code (Assigned by USB Org)
        /// </summary>
        public byte bInterfaceProtocol;
        /// <summary>
        /// Index of String Descriptor Describing this interface
        /// </summary>
        public byte iInterface;
    };

    /// <summary>
    /// USB string descriptor
    /// </summary>
    [StructLayout(LayoutKind.Sequential, Size = 516, Pack = 1, CharSet = CharSet.Unicode)]
    public class FT_STRING_DESCRIPTOR : FT_DESCRIPTOR
    {
        /// <summary>
        /// Unicode Encoded String
        /// </summary>
        [MarshalAs(UnmanagedType.LPWStr, SizeConst = 512)]
        public string szString;
    };

    /// <summary>
    /// Pipe information
    /// </summary>
    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    public class FT_PIPE_INFORMATION
    {
        /// <summary>
        /// Pipe type
        /// </summary>
        public FT_PIPE_TYPE PipeType;
        /// <summary>
        /// Pipe ID
```

```
        /// </summary>
        public byte PipeId;
        /// <summary>
        /// Max transfer size
        /// </summary>
        public UInt16 MaximumPacketSize;
        /// <summary>
        /// Interval for polling, for interrupt pipe only
        /// </summary>
        public byte Interval;
    };


    //////////////////////////////////////////////////////////////////////////
    // NOTIFICATIONS
    //////////////////////////////////////////////////////////////////////////

    /// <summary>
    /// Notification callback info for polymorphism
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public abstract class FT_NOTIFICATION_INFO
    {
    };

    /// <summary>
    /// Notification callback info for DATA
    /// </summary>
    [StructLayout(LayoutKind.Sequential, Size = 5, Pack = 1)]
    public class FT_NOTIFICATION_INFO_DATA : FT_NOTIFICATION_INFO
    {
        /// <summary>
        /// Data length
        /// </summary>
        public UInt32 ulDataLength;
        /// <summary>
        /// Endpoint number
        /// </summary>
        public byte ucEndpointNo;
    };

    /// <summary>
    /// Notification callback info for GPIO
    /// </summary>
    [StructLayout(LayoutKind.Sequential, Size = 8, Pack = 1)]
    public class FT_NOTIFICATION_INFO_GPIO : FT_NOTIFICATION_INFO
    {
        /// <summary>
        /// GPIO 0 status
        /// </summary>
        public UInt32 bGPIO0;
        /// <summary>
        /// GPIO 1 status
        /// </summary>
        public UInt32 bGPIO1;
    };


    //////////////////////////////////////////////////////////////////////////
    // CONFIGURATION
    //////////////////////////////////////////////////////////////////////////

    /// <summary>
    /// FTXXX chip configuration
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public abstract class FT_CONFIGURATION
    {
        /// <summary>
        /// Vendor ID as specified in USB device descriptor
```

```
        /// </summary>
        public UInt16 VendorID;
        /// <summary>
        /// Product ID as specified in USB device descriptor
        /// </summary>
        public UInt16 ProductID;
        /// <summary>
        /// Manufacturer referenced to in USB device descriptor
        /// </summary>
        public string Manufacturer;
        /// <summary>
        /// Product description as referenced in USB device descriptor
        /// </summary>
        public string Description;
        /// <summary>
        /// Serial number as referenced in USB device descriptor
        /// </summary>
        public string SerialNumber;
    }

    /// <summary>
    /// FT60X chip configuration
    /// </summary>
    [StructLayout(LayoutKind.Sequential)]
    public class FT_60XCONFIGURATION : FT_CONFIGURATION
    {
        /// <summary>
        /// Reserved byte
        /// </summary>
        public byte Reserved;
        /// <summary>
        /// Power attribute as described in USB device descriptor
        /// </summary>
        public byte PowerAttributes;
        /// <summary>
        /// Power consumption as described in USB device descriptor
        /// </summary>
        public UInt16 PowerConsumption;
        /// <summary>
        /// Reserved byte
        /// </summary>
        public byte Reserved2;
        /// <summary>
        /// FIFO clock
        /// </summary>
        public byte FIFOClock;
        /// <summary>
        /// FIFO mode
        /// </summary>
        public byte FIFOMode;
        /// <summary>
        /// Channel configuration
        /// </summary>
        public byte ChannelConfig;
        /// <summary>
        /// Optional features
        /// </summary>
        public UInt16 OptionalFeatureSupport;
        /// <summary>
        /// Battery charing GPIO config
        /// </summary>
        public byte BatteryChargingGPIOConfig;
        /// <summary>
        /// Read-only memory detection
        /// </summary>
        public byte FlashEEPROMDetection;
        /// <summary>
        /// MSIO control
        /// </summary>
        public UInt32 MSIO_Control;
```

```
        /// <summary>
        /// GPIO control
        /// </summary>
        public UInt32 GPIO_Control;
    };

    /// <summary>
    /// Delegates declaration for the FTD3XX.DLL functions
    /// </summary>
    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    public delegate void FT_NOTIFICATION_CALLBACK_DATA(
            IntPtr pvContext, FT_NOTIFICATION_TYPE eType, FT_NOTIFICATION_INFO_DATA pvInfo);

    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    public delegate void FT_NOTIFICATION_CALLBACK_GPIO(
            IntPtr pvContext, FT_NOTIFICATION_TYPE eType, FT_NOTIFICATION_INFO_GPIO pvInfo);

    [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
    public delegate void FT_NOTIFICATION_CALLBACK(
            IntPtr pvContext, FT_NOTIFICATION_TYPE eType, FT_NOTIFICATION_INFO pvInfo);

    /// <summary>
    /// GUID of D3XX devices
    /// </summary>
    public static readonly Guid FT_GUID = new Guid("d1e8fe6a-ab75-4d9e-97d2-06fa22c7736c");
```

Document References

http://www.ftdichip.com/Products/ICs/FT600.html

http://www.ftdichip.com/Products/Modules/SuperSpeedModules.htm

http://www.ftdichip.com/Drivers/D3XX.htm

http://www.ftdichip.com/Support/SoftwareExamples/FT60X.htm

http://www.ftdichip.com/Support/Utilities.htm#FT60X_Configuration

AN_379 D3xx Programmers Guide

Acronyms and Abbreviations

Terms	Description
API	Application Programming Interfaces
DLL	Dynamically Linked Library
D3XX	FTDI's proprietary "direct" driver interface via FTD3XX.DLL
EP	Endpoint
USB	Universal Serial Bus

Appendix B – List of Tables & Figures

List of Tables

List of Figures

Appendix C – Revision History

Document Title: AN_407 D3XX .NET Programmers Guide
Document Reference No.: FT_001315
Clearance No.: FTDI#513
Product Page: http://www.ftdichip.com/FTProducts.htm
Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2016-11-01