FTDI provides a DLL application interface to its SuperSpeed USB drivers. This document provides the application programming interface (API) for the FTD3XX DLL function library.

Use of FTDI devices in life support and/or safety applications is entirely at the user’s risk, and the user agrees to defend, indemnify and hold FTDI harmless from any and all damages, claims, suits or expense resulting from such use.
Table of Contents

1 Introduction ................................................................. 5

2 D3XX FT60X Functions ..................................................... 6
   2.1 FT_CreateDeviceInfoList ............................................. 6
   2.2 FT_GetDeviceInfoList ............................................... 7
   2.3 FT_GetDeviceInfoDetail ............................................. 8
   2.4 FT_ListDevices ................................................... 10
   2.5 FT_Create ......................................................... 12
   2.6 FT_Close .......................................................... 14
   2.7 FT_WritePipe ...................................................... 15
   2.8 FT_ReadPipe ....................................................... 17
   2.9 FT_WritePipeEx .................................................... 19
   2.10 FT_ReadPipeEx .................................................... 21
   2.11 FT_GetOverlappedResult ......................................... 23
   2.12 FT_InitializeOverlapped ......................................... 24
   2.13 FT_ReleaseOverlapped ............................................ 25
   2.14 FT_SetStreamPipe ................................................ 26
   2.15 FT_ClearStreamPipe .............................................. 27
   2.16 FT_SetPipeTimeout ............................................... 28
   2.17 FT_GetPipeTimeout ............................................... 29
   2.18 FT_AbortPipe ..................................................... 30
   2.19 FT_GetDeviceDescriptor ......................................... 31
   2.20 FT_GetConfigurationDescriptor ................................. 32
   2.21 FT_GetInterfaceDescriptor ...................................... 33
   2.22 FT_GetPipeInformation .......................................... 34
   2.23 FT_GetDescriptor ................................................ 35
   2.24 FT_ControlTransfer ............................................... 36
2.25 FT_GetVIDPID .........................................................37
2.26 FT_EnableGPIO ....................................................38
2.27 FT_WriteGPIO .....................................................39
2.28 FT_ReadGPIO .......................................................40
2.29 FT_SetGPIOPull ....................................................41
2.30 FT_SetNotificationCallback ......................................42
2.31 FT_ClearNotificationCallback .....................................43
2.32 FT_GetChipConfiguration .........................................44
2.33 FT_SetChipConfiguration .........................................45
2.34 FT_IsDevicePath ...................................................47
2.35 FT_GetDriverVersion ..............................................48
2.36 FT_GetLibraryVersion .............................................49
2.37 FT_CycleDevicePort ..............................................50
2.38 FT_SetSuspendTimeout ............................................51
2.39 FT_GetSuspendTimeout ............................................52

3 Contact Information .................................................53

Appendix A – References ............................................54
  Major differences with D2XX ......................................54
  Type Definitions .....................................................55
  Support for multiple devices .....................................60
  Achieving maximum performance .................................60
  Code Samples .......................................................61
  Document References ..............................................74
  Acronyms and Abbreviations ....................................74

Appendix B – List of Tables & Figures ............................75
  List of Tables .......................................................75
  List of Figures .......................................................75
Appendix C – Revision History ........................................... 76
1 Introduction

The D3XX interface is a proprietary interface specifically for FTDI SuperSpeed USB devices (FT60x series). D3XX implements a proprietary protocol different from D2XX in order to maximize USB 3.0 bandwidth. This document provides an explanation of the functions available to application developers via the FTD3XX library. Any software code examples given in this document are for information only. The examples are not guaranteed and are not supported by FTDI.

FT600 and FT601 are the first devices in a brand new USB SuperSpeed series from FTDI Chip. The devices provide a USB 3 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, 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 bandwidth. The Function Protocol is implemented using 2 interfaces – communication interface and data interface. The data interface contains 4 channels with each channel having a read and write BULK endpoint, for a total of 8 data endpoints. The communication interface includes 2 dedicated endpoints, EP OUT BULK 0x01 and EP IN INTERRUPT 0x81. The OUT BULK endpoint is for receiving session list commands from the host, targeted mainly for high 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 traffic. Combining the use of the two endpoints above provides performance and flexibility.

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Endpoints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x01</td>
<td>OUT BULK endpoint for Session List commands</td>
</tr>
<tr>
<td></td>
<td>0x81</td>
<td>IN INTERRUPT endpoint for Notification List commands</td>
</tr>
<tr>
<td>1</td>
<td>0x02-0x05</td>
<td>OUT BULK endpoint for application write access</td>
</tr>
<tr>
<td></td>
<td>0x82-0x85</td>
<td>IN BULK endpoint for application read access</td>
</tr>
</tbody>
</table>

Table 1 - FT600 Series Function Protocol Interfaces and Endpoints
2 D3XX FT60X Functions

2.1 FT_CreateDeviceInfoList

```
FT_STATUS
FT_CreateDeviceInfoList(
    LPDWORD lpdwNumDevs,
)
```

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

- `lpdwNumDevs`: Pointer to unsigned long to store the number of devices connected.

Return Value

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

An application can use this function to get the number of devices attached to the system. It can then allocate space for the device information list and retrieve the list using FT_GetDeviceInfoList or FT_GetDeviceInfoDetail.

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

```
FT_STATUS
FT_GetDeviceInfoList(     
    FT_DEVICE_LIST_INFO_NODE *ptDest,     
    LPDWORD lpdwNumDevs,     
)
```

**Summary**

Returns a device information list and the number of D3XX devices in the list.

**Parameters**

- `ptDest` Pointer to an array of FT_DEVICE_LIST_INFO_NODE structures.
- `lpdwNumDevs` Pointer to unsigned long to store the number of devices connected.

**Return Value**

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

This function should only be called after calling FT_CreateDeviceInfoList. If the devices connected to the system change, the device info list will not be updated until FT_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_LIST_INFO_NODE will indicate that the device is open, but other fields will be unpopulated.

The array of FT_DEVICE_LIST_INFO_NODE contains all available data on each device. The storage for the list must be allocated by the application. The number of devices returned by FT_CreateDeviceInfoList can be used to do this.

The Type field of FT_DEVICE_LIST_INFO_NODE 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.
2.3 FT_GetDeviceInfoDetail

```
FT_STATUS
FT_GetDeviceInfoDetail(
    DWORD dwIndex,
    LPDWORD lpdwFlags,
    LPDWORD lpdwType,
    LPDWORD lpdwID,
    LPDWORD lpdwLocId,
    LPVOID lpSerialNumber,
    LPVOID lpDescription,
    FT_HANDLE *pftHandle
);
```

**Summary**

Returns an entry from the device information list detail located at a specified index.

**Parameters**

- **dwIndex**
  - Index of the entry in the device info list.
  - The index value is zero-based.
- **lpdwFlags**
  - Pointer to unsigned long to store the flag value.
- **lpdwType**
  - Pointer to unsigned long to store device type.
- **lpdwID**
  - Pointer to unsigned long to store device ID.
- **lpdwLocId**
  - Pointer to unsigned long to store the device location ID.
- **lpSerialNumber**
  - Pointer to buffer to store device serial number as a null-terminated string.
- **lpDescription**
  - Pointer to buffer to store device description as a null-terminated string.
- **pftHandle**
  - Pointer to a variable of type FT_HANDLE where the handle will be stored.

**Return Value**

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

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

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

To return the whole device info list as an array of FT_DEVICE_LIST_INFO_NODE structures, use FT_GetDeviceInfoList.
Get and display the list of devices connected using FT_GetDeviceInfoList

```c
FT_STATUS ftStatus;
DWORD numDevs = 0;

ftStatus = FT_CreateDeviceInfoList(&numDevs);
if (!FT_FAILED(ftStatus) && numDevs > 0)
{
    FT_DEVICE_LIST_INFO_NODE *devInfo = (FT_DEVICE_LIST_INFO_NODE*)malloc(
        sizeof(FT_DEVICE_LIST_INFO_NODE) * numDevs);
    ftStatus = FT_GetDeviceInfoList(devInfo, &numDevs);
    if (!FT_FAILED(ftStatus))
    {
        printf("List of Connected Devices\n\n");
        for (DWORD i = 0; i < numDevs; i++)
        {
            printf("Device[%d]\n", i);
            printf("| Flags: 0x%x | Type: %d | ID: 0x%08X | ftHandle=0x%lx\n",
                devInfo[i].Flags,
                devInfo[i].Flags & FT_FLAGS_SUPERSPEED ? "[USB 3]":
                devInfo[i].Flags & FT_FLAGS_HISPEED ? "[USB 2]":
                devInfo[i].Flags & FT_FLAGS_OPENED ? "[OPENED]": "",
                devInfo[i].Type,
                devInfo[i].ID,
                devInfo[i].ftHandle);
            printf("| SerialNumber=%s\n", devInfo[i].SerialNumber);
            printf("| Description=%s\n", devInfo[i].Description);
        }
        free(devInfo);
    }
}
```

Get and display the list of devices connected using FT_GetDeviceInfoDetail

```c
FT_STATUS ftStatus;
DWORD numDevs = 0;

ftStatus = FT_CreateDeviceInfoList(&numDevs);
if (!FT_FAILED(ftStatus) && numDevs > 0)
{
    FT_HANDLE ftHandle = NULL;
    DWORD Flags = 0;
    DWORD Type = 0;
    DWORD ID = 0;
    char SerialNumber[16] = { 0 };
    char Description[32] = { 0 };

    printf("List of Connected Devices\n\n");
    for (DWORD i = 0; i < numDevs; i++)
    {
        ftStatus = FT_GetDeviceInfoDetail(i, &Flags, &Type, &ID, NULL,
            SerialNumber, Description, &ftHandle);
        if (!FT_FAILED(ftStatus))
        {
            printf("Device[%d]\n", i);
            printf("| Flags: 0x%x | Type: %d | ID: 0x%08X | ftHandle=0x%lx\n",
                Flags,
                Flags & FT_FLAGS_SUPERSPEED ? "[USB 3]":
                Flags & FT_FLAGS_HISPEED ? "[USB 2]":
                Flags & FT_FLAGS_OPENED ? "[OPENED]": "",
                Type,
                ID,
                ftHandle);
            printf("| SerialNumber=%s\n", SerialNumber);
            printf("| Description=%s\n", Description);
        }
    }
}
```
2.4 FT_ListDevices

```c
FT_STATUS
FT_ListDevices(
    PVOID pArg1,
    PVOID pArg2,
    DWORD Flags
)
```

**Summary**

Gets information for all D3XX devices currently connected. This function can return information such as the number of devices connected, the device serial number and device description strings.

**Parameters**

- `pvArg1` Meaning depends on `dwFlags`.
- `pvArg2` Meaning depends on `dwFlags`.
- `dwFlags` Determines format of returned information.

**Return Value**

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

This function can be used in a number of ways to return different types of information. A more powerful way to get device information is to use the `FT_CreateDeviceInfoList`, `FT_GetDeviceInfoList` and `FT_GetDeviceInfoDetail` functions as they return all the available information on devices.

In its simplest form, it can be used to return the number of devices currently connected. If the `FT_LIST_NUMBER_ONLY` bit is set in `dwFlags`, the parameter `pvArg1` is interpreted as a pointer to a DWORD location to store the number of devices currently connected.

It can be used to return device information: if the `FT_OPEN_BY_SERIAL_NUMBER` bit is set in `dwFlags`, the serial number string will be returned; if the `FT_OPEN_BY_DESCRIPTION` bit is set in `dwFlags`, the product description string will be returned; if none of these bits are set, the serial number string will be returned by default.

It can be used to return device string information for a single device. If `FT_LIST_BY_INDEX` and `FT_OPEN_BY_SERIAL_NUMBER` or `FT_OPEN_BY_DESCRIPTION` bits are set in `dwFlags`, the parameter `pvArg1` is interpreted as the index of the device, and the parameter `pvArg2` is interpreted as a pointer to a buffer to contain the appropriate string. Indexes are zero-based, and the error code `FT_DEVICE_NOT_FOUND` is returned for an invalid index.

It can also be used to return device string information for all connected devices. If `FT_LIST_ALL` and `FT_OPEN_BY_SERIAL_NUMBER` or `FT_OPEN_BY_DESCRIPTION` bits are set in `dwFlags`, the parameter `pvArg1` is interpreted as a pointer to an array of pointers to buffers to contain the appropriate strings and the parameter `pvArg2` is interpreted as a pointer to a DWORD location to store the number of devices currently connected. Note that, for `pvArg1`, the last entry in the array of pointers to buffers should be a NULL pointer so the array will contain one more location than the number of devices connected.
Get the number of devices currently connected

```c
FT_STATUS ftStatus;
DWORD numDevs = 0;
ftStatus = FT_ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
```

Get the serial number of the first device

```c
FT_STATUS ftStatus;
DWORD devIndex = 0;
char SerialNumber[16] = { 0 };
ftStatus = FT_ListDevices((PVOID)devIndex, SerialNumber, FT_LIST_BY_INDEX | FT_OPEN_BY_SERIAL_NUMBER);
```

Get the product description of the first device

```c
FT_STATUS ftStatus;
DWORD devIndex = 0;
char Description[32] = { 0 };
ftStatus = FT_ListDevices((PVOID)devIndex, Description, FT_LIST_BY_INDEX | FT_OPEN_BY_DESCRIPTION);
```

Get device serial numbers of all devices currently connected

```c
char *BufPtrs[3] = { NULL }; // pointer to array of 3 pointers
char SerialNumber1[16] = { 0 }; // buffer for serial number of first device
char SerialNumber2[16] = { 0 }; // buffer for serial number of second device
// initialize the array of pointers
BufPtrs[0] = SerialNumber1;
BufPtrs[1] = SerialNumber2;
BufPtrs[2] = NULL; // last entry should be NULL
ftStatus = FT_ListDevices(BufPtrs, &numDevs, FT_LIST_ALL | FT_OPEN_BY_SERIAL_NUMBER);
```

Get device descriptions of all devices currently connected

```c
char *BufPtrs[3] = { NULL }; // pointer to array of 3 pointers
char Description1[32] = { 0 }; // buffer for description of first device
char Description2[32] = { 0 }; // buffer for description of second device
// initialize the array of pointers
BufPtrs[0] = Description1;
BufPtrs[1] = Description2;
BufPtrs[2] = NULL; // last entry should be NULL
ftStatus = FT_ListDevices(BufPtrs, &numDevs, FT_LIST_ALL | FT_OPEN_BY_DESCRIPTION);
```
2.5 FT_Create

```
FT_STATUS
FT_Create(
    PVOID pvArg,
    DWORD dwFlags,
    FT_HANDLE* pftHandle
)
```

### Summary

Open the device and return a handle which will be used for subsequent accesses.

### Parameters

- **pvArg**
  
  Pointer to an argument whose type depends on the value of `dwFlags`
  
  - If `FT_OPEN_BY_SERIAL_NUMBER`, `pvArg` is of type `CHAR*`
  
  - If `FT_OPEN_BY_DESCRIPTION`, `pvArg` is of type `CHAR*`
  
  - If `FT_OPEN_BY_INDEX`, `pvArg` is of type `ULONG`

- **dwFlags**
  
  - `FT_OPEN_BY_SERIAL_NUMBER`
  
  - `FT_OPEN_BY_DESCRIPTION`
  
  - `FT_OPEN_BY_INDEX`

- **pftHandle**
  
  Pointer to a variable where the handle will be stored.
  
  This handle must be used to access the device.

### Return Value

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

The parameter specified in `pvArg` depends on `dwFlags`: if `dwFlags` is `FT_OPEN_BY_SERIAL_NUMBER`, `pvArg` is interpreted as a pointer to a null-terminated string that represents the serial number of the device; if `dwFlags` is `FT_OPEN_BY_DESCRIPTION`, `pvArg` is interpreted as a pointer to a null-terminated string that represents the device description; if `dwFlags` is `FT_OPEN_BY_INDEX`, `pvArg` is interpreted as an integer value indicating the index of the device.

To allow multiple FT60x devices to be connected to a machine, it is assumed that String Descriptors (Manufacturer, Product Description, and Serial Number) in the USB Device Descriptor are updated to suitable values using FT_SetChipConfiguration or using the FT60x Chip Configuration Programmer tool provided by FTDI, which is available here. 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 same manufacturer. The Serial Number must uniquely identify the device from other devices with the same Product name and Manufacturer name.

Using `FT_OPEN_BY_SERIAL_NUMBER` allows an application to open a device that has the specified Serial Number. Using `FT_OPEN_BY_DESCRIPTION` allows an application to open a device that has the specified Product Description. Using `FT_OPEN_BY_INDEX` is a fall-back option for instances where the devices connected to a machine do not have a unique Serial Number or Product Description.
Open a device with serial number "000000000001"

```c
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
ftStatus = FT_Create("000000000001", FT_OPEN_BY_SERIAL_NUMBER, &ftHandle);
```

Open a device with product description "FTDI SuperSpeed-FIFO Bridge"

```c
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
ftStatus = FT_Create("FTDI SuperSpeed-FIFO Bridge", FT_OPEN_BY_DESCRIPTION, &ftHandle);
```

Open a device with serial number "1234567890ABCde"

```c
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
ftStatus = FT_Create("1234567890ABCde", FT_OPEN_BY_SERIAL_NUMBER, &ftHandle);
```

Open a device with product description "This is My Product Description"

```c
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
ftStatus = FT_Create("This is My Product Description", FT_OPEN_BY_DESCRIPTION, &ftHandle);
```
2.6 FT_Close

```
FT_STATUS
FT_Close(
    FT_HANDLE ftHandle
)
```

**Summary**

Close an open device.

**Parameters**

- `ftHandle`: A handle to the device

**Return Value**

- `FT_OK` if successful, otherwise the return value is an FT error code.
2.7 FT_WritePipe

```c
FT_STATUS
FT_WritePipe(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID,
    PUCHAR pucBuffer,
    ULONG ulBufferLength,
    PULONG pulBytesTransferred,
    LPOVERLAPPED pOverlapped
)
```

**Summary**
Write data to pipe.

**Parameters**
- `ftHandle`: A handle to the device
- `ucPipeID`: Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
- `pucBuffer`: Buffer that contains the data to write.
- `ulBufferLength`: The number of bytes to write. This number must be less than or equal to the size, in bytes, of the Buffer.
- `pulBytesTransferred`: A pointer to a ULONG variable that receives the actual number of bytes written to the pipe.
- `pOverlapped`: An optional pointer to an OVERLAPPED structure, used for asynchronous operations.

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

If `lpOverlapped` is NULL, `FT_WritePipe` operates synchronously, that is, it returns only when the transfer has been completed.

If `lpOverlapped` is not NULL, `FT_WritePipe` operates asynchronously and immediately returns `FT_IO_PENDING`. `FT_GetOverlappedResult` should be called to wait for the completion of this asynchronous operation. When supplying the `lpOverlapped` to `FT_WritePipe`, the event parameter of `lpOverlapped` should be initialized using `FT_InitializeOverlapped`.

If an `FT_WritePipe` call fails with an error code (status other than `FT_OK` or `FT_IO_PENDING`), an application should call `FT_AbortPipe`. To ensure that the pipe is in a clean state it is recommended to follow the abort procedure mentioned in the section 3.2 of "AN_412_FT600_FT601 USB Bridge chips Integration".
Synchronous write to pipe 0x02

```c
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ftStatus = FT_WritePipe(ftHandle, 0x02, acBuf, BUFFER_SIZE &ulBytesTransferred, NULL);
```

Asynchronous write to pipe 0x02

```c
OVERLAPPED vOverlapped = {0};
ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped);
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_WritePipe(ftHandle, 0x02, acBuf, BUFFER_SIZE, &ulBytesTransferred, &vOverlapped);
if (ftStatus == FT_IO_PENDING)
{
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped, &ulBytesTransferred, TRUE);
}
FT_ReleaseOverlapped(ftHandle, &vOverlapped);
```

Triple buffering / 3 asynchronous write to pipe 0x02

```c
#define NUM_BUFFERS 3
#define BUFFER_SIZE 8294400 // Full-HD: 1920 x 1080 x 4

UCHAR acBuf[NUM_BUFFERS][BUFFER_SIZE] = {0xFF};
OVERLAPPED vOverlapped[NUM_BUFFERS] = {0};

for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped[i]);
}

// Queue up the initial batch of requests
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_WritePipe(ftHandle, 0x02, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);
}

int i=0;

// Infinite transfer loop
while (bKeepGoing)
{
    // Wait for transfer to finish
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped[i], &ulBytesTransferred[i], TRUE);

    // Re-submit to keep request full
    ftStatus = FT_WritePipe(ftHandle, 0x02, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);

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

for (int i=0; i<NUM_BUFFERS; i++)
{
    FT_ReleaseOverlapped(ftHandle, &vOverlapped);
}
2.8 FT_ReadPipe

FT_STATUS
FT_ReadPipe(  
    FT_HANDLE ftHandle,  
    UCHAR ucPipeID,  
    P UCHAR pucBuffer,  
    ULONG ulBufferLength,  
    P ULONG pulBytesTransferred,  
    LPOVERLAPPED pOverlapped  
)

Summary
Read data from pipe.

Parameters
- **ftHandle**: A handle to the device
- **ucPipeID**: Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
- **pucBuffer**: Buffer that will contain the data read.
- **ulBufferLength**: The number of bytes to read. This number must be less than or equal to the size, in bytes, of Buffer.
- **pulBytesTransferred**: A pointer to a ULONG variable that receives the actual number of bytes read from the pipe.
- **pOverlapped**: An optional pointer to an OVERLAPPED structure, this is used for asynchronous operations.

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

If lpOverlapped is NULL, FT_ReadPipe operates synchronously, that is, it returns only when the transfer has been completed.

If lpOverlapped is not NULL, FT_ReadPipe operates asynchronously and immediately returns FT_IO_PENDING. FT_GetOverlappedResult should be called to wait for the completion of this asynchronous operation. When supplying the lpOverlapped to FT_ReadPipe, the event parameter of lpOverlapped should be initialized using FT_InitializeOverlapped.

Default read timeout value is 5 seconds and this can be changed by calling FT_SetPipeTimeout API.

If the timeout occurred, FT_ReadPipe (FT_GetOverlappedResult in case of asynchronous call), returns with an error code FT_TIMEOUT.

An application can call FT_SetPipeTimeout with a timeout value 0 to disable timeouts.

If FT_ReadPipe call fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call FT_AbortPipe. To ensure that the pipe is in a clean state it is recommended to follow the abort procedure mentioned in section 3.2 of "AN_412_FT600_FT601 USB Bridge chips Integration".
Synchronous read from pipe 0x82

```c
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_ReadPipe(ftHandle, 0x82, acBuf, BUFFER_SIZE &ulBytesTransferred, NULL);
```

Asynchronous read from pipe 0x82

```c
OVERLAPPED vOverlapped = {0};
ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped);

UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_ReadPipe(ftHandle, 0x82, acBuf, BUFFER_SIZE, &ulBytesTransferred, &vOverlapped);
// (ftStatus == FT_IO_PENDING)
{
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped, &ulBytesTransferred, TRUE);
}
FT_ReleaseOverlapped(ftHandle, &vOverlapped);
```

Triple buffering / 3 asynchronous read from pipe 0x82

```c
#define NUM_BUFFERS 3
#define BUFFER_SIZE 8294400  // Full-HD: 1920 x 1080 x 4

UCHAR acBuf[NUM_BUFFERS][BUFFER_SIZE] = {0xFF};
OVERLAPPED vOverlapped[NUM_BUFFERS] = {0};

for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped[i]);
}

// Queue up the initial batch of requests
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_ReadPipe(ftHandle, 0x82, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);
}

int i=0;

// Infinite transfer loop
while (bKeepGoing)
{
    // Wait for transfer to finish
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped[i], &ulBytesTransferred[i], TRUE);

    // Re-submit to keep request full
    ftStatus = FT_ReadPipe(ftHandle, 0x82, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);

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

for (int i=0; i<NUM_BUFFERS; i++)
{
    FT_ReleaseOverlapped(ftHandle, &vOverlapped);
}
```
## 2.9 FT_WritePipeEx

```c
FT_STATUS FT_WritePipeEx(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID,
    PUCCHAR pucBuffer,
    ULONG ulBufferLength,
    PULONG pulBytesTransferred,
    LPOVERLAPPED pOverlapped
)
```

### Summary
Writes data to the pipe. FT_WritePipeEx has much lower latency compared to FT_WritePipe when used for asynchronous transfers with FT_SetStreamPipe. However, the maximum input buffer size supported for this API is 1 Mega Byte to guarantee the lower latencies.

### Parameters
- `ftHandle`: A handle to the device
- `ucPipeID`: Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
- `pucBuffer`: Buffer that contains the data to write.
- `ulBufferLength`: The number of bytes to write. This number must be less than or equal to the size, in bytes, of the Buffer.
- `pulBytesTransferred`: A pointer to a ULONG variable that receives the actual number of bytes written to the pipe.
- `pOverlapped`: An optional pointer to an OVERLAPPED structure, used for asynchronous operations.

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

If `lpOverlapped` is NULL, FT_WritePipeEx operates synchronously, that is, it returns only when the transfer has been completed.

If `lpOverlapped` is not NULL, FT_WritePipeEx operates asynchronously and immediately returns FT_IO_PENDING. FT_GetOverlappedResult should be called to wait for the completion of this asynchronous operation. When supplying the `lpOverlapped` to FT_WritePipeEx, the event parameter of `lpOverlapped` should be initialized using FT_InitializeOverlapped.

If an FT_WritePipeEx call fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call FT_AbortPipe. To ensure that the pipe is in a clean state it is recommended to follow the abort procedure mentioned in the section 3.2 of “AN_412_FT600_FT601 USB Bridge chips Integration”.

Synchronous write to pipe 0x02

```c
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_WritePipeEx(ftHandle, 0x02, acBuf, BUFFER_SIZE &ulBytesTransferred, NULL);
```

Asynchronous write to pipe 0x02

```c
OVERLAPPED vOverlapped = {0};
ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped);

UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_WritePipeEx(ftHandle, 0x02, acBuf, BUFFER_SIZE, &ulBytesTransferred, &vOverlapped);
if (ftStatus == FT_IO_PENDING)
{
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped, &ulBytesTransferred, TRUE);
}
FT_ReleaseOverlapped(ftHandle, &vOverlapped);
```

Multiple asynchronous write to pipe 0x02

```c
#define NUM_BUFFERS 16
#define BUFFER_SIZE (256*1024)

UCHAR acBuf[NUM_BUFFERS][BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred[NUM_BUFFERS] = 0;
OVERLAPPED vOverlapped[NUM_BUFFERS] = {0};

ftStatus = FT_SetStreamPipe(ftHandle, FALSE, FALSE, 0x02, BUFFER_SIZE);
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped[i]);
}

// Queue up the initial batch of requests
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_WritePipeEx(ftHandle, 0x02, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);
}

int i=0;

// Infinite transfer loop
while (bKeepGoing)
{
    // Wait for transfer to finish
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped[i], &ulBytesTransferred[i], TRUE);

    // Re-submit to keep request full
    ftStatus = FT_WritePipeEx(ftHandle, 0x02, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);

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

for (int i=0; i<NUM_BUFFERS; i++)
{
    FT_ReleaseOverlapped(ftHandle, &vOverlapped);
}
2.10 FT_ReadPipeEx

**FT_STATUS**

```c
FT_STATUS FT_ReadPipeEx(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID,
    PUCHAR pucBuffer,
    ULONG ulBufferLength,
    PULONG pulBytesTransferred,
    LPOVERLAPPED pOverlapped
)
```

### Summary

Reads data from the pipe. An enhanced version of FT_ReadPipe for improved latencies between reads. However to get the maximum benefit, this API should be used asynchronously with FT_SetStreamPipe.

### Parameters

- **ftHandle**: A handle to the device
- **ucPipeID**: Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
- **pucBuffer**: Buffer that will contain the data read.
- **ulBufferLength**: The number of bytes to read. This number must be less than or equal to the size, in bytes, of Buffer.
- **pulBytesTransferred**: A pointer to a ULONG variable that receives the actual number of bytes read from the pipe.
- **pOverlapped**: An optional pointer to an OVERLAPPED structure, this is used for asynchronous operations.

### Return Value

- **FT_OK** if successful, otherwise the return value is an FT error code.
  
  If lpOverlapped is NULL, FT_ReadPipeEx operates synchronously, that is, it returns only when the transfer has been completed.

  If lpOverlapped is not NULL, FT_ReadPipeEx operates asynchronously and immediately returns FT_IO_PENDING. FT_GetOverlappedResult should be called to wait for the completion of this asynchronous operation. When supplying the lpOverlapped to FT_ReadPipeEx, the event parameter of lpOverlapped should be initialized using FT_InitializeOverlapped.

  Default read timeout value is 5 seconds and this can be changed by calling **FT_SetPipeTimeout** API.

  If the timeout occurred, FT_ReadPipeEx (FT_GetOverlappedResult in case of asynchronous call), returns with an error code FT_TIMEOUT.

  An application can call **FT_SetPipeTimeout** with a timeout value 0 to disable timeouts.

  If the FT_ReadPipeEx call fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call FT_AbortPipe. To ensure that the pipe is in a clean state it is recommended to follow the abort procedure mentioned in section 3.2 of "AN_412_FT600_FT601 USB Bridge chips Integration".
Synchronous read from pipe 0x82

```
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_ReadPipeEx(ftHandle, 0x82, acBuf, BUFFER_SIZE &ulBytesTransferred, NULL);
```

Asynchronous read from pipe 0x82

```
OVERLAPPED vOverlapped = {0};
ftStatus = FT_InitializeOverlappedEx(ftHandle, &vOverlapped);
UCHAR acBuf[BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred = 0;
ftStatus = FT_ReadPipeEx(ftHandle, 0x82, acBuf, BUFFER_SIZE, &ulBytesTransferred, &vOverlapped);
if (ftStatus == FT_IO_PENDING)
{
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped, &ulBytesTransferred, TRUE);
}
FT_ReleaseOverlapped(ftHandle, &vOverlapped);
```

Multiple asynchronous read from pipe 0x82

```
#define NUM_BUFFERS 16
#define BUFFER_SIZE (256*1024)
UCHAR acBuf[NUM_BUFFERS][BUFFER_SIZE] = {0xFF};
ULONG ulBytesTransferred[NUM_BUFFERS] = 0;
OVERLAPPED vOverlapped[NUM_BUFFERS] = {0};
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_InitializeOverlapped(ftHandle, &vOverlapped[i]);
}
ftStatus = FT_SetStreamPipe(ftHandle, FALSE, FALSE, 0x82, BUFFER_SIZE);

// Queue up the initial batch of requests
for (int i=0; i<NUM_BUFFERS; i++)
{
    ftStatus = FT_ReadPipeEx(ftHandle, 0x82, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);
}
int i=0;

// Infinite transfer loop
while (bKeepGoing)
{
    // Wait for transfer to finish
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlapped[i], &ulBytesTransferred[i], TRUE);
    // Re-submit to keep request full
    ftStatus = FT_ReadPipeEx(ftHandle, 0x82, &acBuf[i][0], BUFFER_SIZE, &ulBytesTransferred[i], &vOverlapped[i]);
    // Roll-over
    if (++i == NUM_BUFFERS)
    {
        i = 0;
    }
}
for (int i=0; i<NUM_BUFFERS; i++)
{
    FT_ReleaseOverlapped(ftHandle, &vOverlapped);
}
2.11 FT_GetOverlappedResult

FT_STATUS FT_GetOverlappedResult( 
    FT_HANDLE ftHandle, 
    LPOVERLAPPED pOverlapped, 
    PULONG pulLengthTransferred, 
    BOOL bWait 
)

Summary
Retrieves the result of an overlapped operation to a pipe

Parameters
- ftHandle: A handle to the device
- pOverlapped: A pointer to an OVERLAPPED structure that was specified when the overlapped operation was started using FT_WritePipe or FT_ReadPipe. This parameter should be initialized using FT_InitializeOverlapped and released using FT_ReleaseOverlapped.
- pulLengthTransferred: A pointer to a variable that receives the number of bytes that were actually transferred by a read or write operation.
- bWait: If this parameter is TRUE, and the Internal member of the pOverlapped structure is FT_IO_PENDING, the function does not return until the operation has been completed. If this parameter is FALSE and the operation is still pending, the function returns FALSE and the GetLastError function returns FT_IO_INCOMPLETE.

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

In case the call fails with an error code (status other than FT_OK or FT_IO_PENDING), an application should call FT_AbortPipe. To ensure that the pipe is in a clean state it is recommended to follow the abort procedure mentioned in section 3.2 of "AN_412_FT600_FT601 USB Bridge chips Integration".
2.12 FT_InitializeOverlapped

```
FT_STATUS
FT_InitializeOverlapped(
    FT_HANDLE ftHandle,
    LPOVERLAPPED pOverlapped
)
```

**Summary**
Initialize resource for overlapped parameter

**Parameters**
- `ftHandle`: A handle to the device
- `pOverlapped`: A pointer to an OVERLAPPED structure that will be used when using `FT_WritePipe` and `FT_ReadPipe` asynchronously. This parameter should be released using `FT_ReleaseOverlapped` after usage.

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.13 FT_ReleaseOverlapped

```c
FT_STATUS FT_ReleaseOverlapped(
    FT_HANDLE ftHandle,
    LPOVERLAPPED pOverlapped
)
```

**Summary**
Releases resource for the overlapped parameter

**Parameters**
- `ftHandle`: A handle to the device
- `pOverlapped`: A pointer to an OVERLAPPED structure that was used when using FT_WritePipe and FT_ReadPipe asynchronously

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.14 FT_SetStreamPipe

```c
FT_STATUS
FT_SetStreamPipe(
    FT_HANDLE ftHandle,
    BOOL bAllWritePipes,
    BOOL bAllReadPipes,
    UCHAR ucPipeID,
    ULONG ulStreamSize
)
```

**Summary**
Sets streaming protocol transfer for specified pipes. This is for applications that transfer (write or read) a fixed size of data to or from the device.

**Parameters**
- `ftHandle`: A handle to the device
- `bAllWritePipes`: Sets all write pipes (OUT endpoints) to start using streaming transfer
- `bAllReadPipes`: Sets all read pipes (IN endpoints) to start using streaming transfer
- `ucPipeID`: Set only a specific pipe to start using streaming transfer; Only effective if `bAllWritePipes` and `bAllReadPipes` are FALSE
- `ulStreamSize`: Sets the fixed size of data to be transferred to or from the device

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.15 FT_ClearStreamPipe

```c
FT_STATUS
FT_ClearStreamPipe(
    FT_HANDLE ftHandle,
    BOOL bAllWritePipes,
    BOOL bAllReadPipes,
    UCHAR ucPipeID
)
```

**Summary**
Clears streaming protocol transfer for specified pipes

**Parameters**
- `ftHandle`: A handle to the device
- `bAllWritePipes`: Sets all write pipes (OUT endpoints) to stop using streaming transfer
- `bAllReadPipes`: Sets all read pipes (IN endpoints) to stop using streaming transfer
- `ucPipeID`: Set only a specific pipe to stop using streaming transfer; Only effective if `bAllWritePipes` and `bAllReadPipes` are FALSE

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

```c
FT_STATUS FT_SetPipeTimeout(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID,
    ULONG ulTimeoutInMs
);
```

**Summary**
Configures the timeout value for a given endpoint. FT_ReadPipe/FT_WritePipe will timeout in case it hangs for TimeoutInMs amount of time. This will override the default timeout of 5sec. This new value is valid only for the life cycle of ftHandle. A new FT_Create call resets the timeout to default.

**Parameters**
- `ftHandle` A handle to the device
- `ucPipeID` Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN. When 0xFF is used as ucPipeID, then the input specified in TimeoutInMs will be applied on all the IN endpoints.
- `ulTimeoutInMs` Timeout in Milliseconds. If set to 0 (zero), transfers will not timeout. In this case, the transfer waits indefinitely until it is manually cancelled (call to FT_AbortPipe) or the transfer completes normally. If set to a nonzero value (time-out interval), the request will be terminated once the timeout occurs. Default timeout value is 5 sec.

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

This new value is valid only for the life cycle of ftHandle. A new FT_Create call resets the timeout to default.
2.17 FT_GetPipeTimeout

```c
FT_STATUS FT_GetPipeTimeout(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID,
    PULONG pTimeoutInMs
)
```

**Summary**
Gets the timeout value configured for a given IN endpoint.

**Parameters**
- `ftHandle` A handle to the device
- `ucPipeID` Corresponds to the bEndpointAddress field in the endpoint descriptor. In the bEndpointAddress field, Bit 7 indicates the direction of the endpoint: 0 for OUT; 1 for IN.
- `pTimeoutInMs` if the return status is FT_SUCCESS, then this field will contain the timeout value configured for the mentioned pipe id.

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

```
FT_STATUS
FT_AbortPipe(
    FT_HANDLE ftHandle,
    UCHAR ucPipeID
)
```

**Summary**
Aborts all of the pending transfers for a pipe.

**Parameters**
- **ftHandle**: A handle to the device
- **ucPipeID**: This is an 8-bit value that consists of a 7-bit address and a direction bit. This parameter corresponds to the bEndpointAddress field in the endpoint descriptor.

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

```c
FT_STATUS
FT_GetDeviceDescriptor(
    FT_HANDLE ftHandle,
    PFT_DEVICE_DESCRIPTOR pDescriptor
)
```

**Summary**
Get the USB device descriptor.

**Parameters**
- `ftHandle`: A handle to the device
- `pDescriptor`: A pointer to a variable of type `FT_DEVICE_DESCRIPTOR` that will contain the device descriptor

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

**Remarks**
Below is the `FT_DEVICE_DESCRIPTOR` structure.

```c
typedef struct _FT_DEVICE_DESCRIPTOR
{
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    USHORT idVendor;
    USHORT idProduct;
    USHORT bcdDevice;
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} FT_DEVICE_DESCRIPTOR, *PFT_DEVICE_DESCRIPTOR;
```
2.20 FT_GetConfigurationDescriptor

```c
FT_STATUS FT_GetConfigurationDescriptor(
    FT_HANDLE ftHandle,
    PFT_CONFIGURATION_DESCRIPTOR pDescriptor
)
```

**Summary**
Get the USB configuration descriptor.

**Parameters**
- `ftHandle`: A handle to the device
- `pDescriptor`: A pointer to a variable of type `FT_CONFIGURATION_DESCRIPTOR` that will contain the configuration descriptor

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

**Remarks**
The FTDI device supports only 1 USB configuration.

Below is the FT_CONFIGURATION_DESCRIPTOR structure.

```c
typedef struct _FT_CONFIGURATION_DESCRIPTOR
{
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} FT_CONFIGURATION_DESCRIPTOR, *PFT_CONFIGURATION_DESCRIPTOR;
```
2.21 FT_GetInterfaceDescriptor

```c
FT_STATUS
FT_GetInterfaceDescriptor(
    FT_HANDLE ftHandle,
    UCHAR ucInterfaceIndex,
    PFT_INTERFACE_DESCRIPTOR pDescriptor
);
```

**Summary**
Get the USB interface descriptor.

**Parameters**
- `ftHandle` A handle to the device
- `ucInterfaceIndex` An index of the interface for the configuration
- `pDescriptor` A pointer to a variable of type `FT_INTERFACE_DESCRIPTOR` that will contain the interface descriptor

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

**Remarks**
FT60x devices have 2 USB interface descriptors. Interface 0 is used for proprietary protocol implementation while Interface 1 is used for the data transfers.

Below is the `FT_INTERFACE_DESCRIPTOR` structure.

```c
typedef struct _FT_INTERFACE_DESCRIPTOR
{
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} FT_INTERFACE_DESCRIPTOR, *PFT_INTERFACE_DESCRIPTOR;
```
2.22 FT_GetPipeInformation

```c
FT_STATUS FT_GetPipeInformation(
    FT_HANDLE ftHandle,
    UCHAR ucInterfaceIndex,
    UCHAR ucPipeIndex,
    PFT_PIPE_INFORMATION pPipeInformation
)
```

### Summary
Get a USB endpoint descriptor of type FT_PIPE_INFORMATION.

### Parameters
- **ftHandle**: A handle to the device
- **ucInterfaceIndex**: An index of the interface for the configuration
- **ucPipeIndex**: An index of the pipe for the interface
- **pPipeInformation**: Pointer to a variable of type PFT_PIPE_INFORMATION that will contain the pipe information

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

### Remarks
FT_PIPE_INFORMATION is derived from the ENDPOINT_DESCRIPTOR from the USB specification.

Below is the FT_PIPE_INFORMATION structure.

```c
typedef struct _FT_PIPE_INFORMATION
{
    FT_PIPE_TYPE PipeType;
    UCHAR PipeId;
    USHORT MaximumPacketSize;
    UCHAR Interval;
} FT_PIPE_INFORMATION, *PFT_PIPE_INFORMATION;
```
2.23 FT_GetDescriptor

```
FT_STATUS
FT_GetDescriptor(
    FT_HANDLE fHandle,
    UCHAR ucDescriptorType,
    UCHAR ucIndex,
    UCHAR pucBuffer,
    ULONG ulBufferLength,
    PULONG pulLengthTransferred
)
```

Summary

Parameters
- `fHandle` : A handle to the device
- `ucDescriptorType` : Type of descriptor corresponding to the bDescriptorType field of a standard device descriptor
- `ucIndex` : Index of the descriptor
- `pucBuffer` : Pointer to a buffer that will contain the descriptor
- `ulBufferLength` : Length of the buffer provided
- `pulLengthTransferred` : Length of the data copied to the buffer

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

Notes
Below are the different types of descriptors.

```
FT_DEVICE_DESCRIPTOR_TYPE  0x01
FT_CONFIGURATION_DESCRIPTOR_TYPE  0x02
FT_STRING_DESCRIPTOR_TYPE  0x03
FT_INTERFACE_DESCRIPTOR_TYPE  0x04
```
2.24 FT_ControlTransfer

```c
FT_STATUS
FT_ControlTransfer(
    FT_HANDLE ftHandle,
    FT_SETUP_PACKET tSetupPacket,
    PUCHAR pucBuffer,
    ULONG ulBufferLength,
    PULONG pulLengthTransferred
)
```

**Summary**
Transmits control data over the default control endpoint

**Parameters**
- `ftHandle` A handle to the device
- `tSetupPacket` The 8-byte setup packet
- `pucBuffer` Pointer to a buffer that contains the data to transfer
- `ulBufferLength` Length of data to transfer
- `pulLengthTransferred` Length of data transferred

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.25 FT_GetVIDPID

```c
FT_STATUS FT_GetVIDPID(  
    FT_HANDLE ftHandle,  
    PUSHORT puwVID,  
    PUSHORT puwPID
)
```

**Summary**
Get the vendor ID and product ID.

**Parameters**
- **ftHandle**: A handle to the device
- **puwVID**: Pointer to a variable of type USHORT that will contain the VID
- **puwPID**: Pointer to a variable of type USHORT that will contain the PID

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

```c
FT_STATUS
FT_EnableGPIO(
    FT_HANDLE ftHandle,
    UINT32 u32Mask,
    UINT32 u32Dir
)
```

**Summary**
Enables the pins to GPIO mode and sets the input/output direction.

**Parameters**
- `ftHandle`: A handle to the device
- `u32Mask`: Mask to select the bits that are to be enabled (Configure as GPIO). 1=enable, 0=ignore.
- `u32Dir`: Bit0 and bit1 are used and bit [31:2] are unused (ignored). Bit0 controls the direction of GPIO0 and bit1 controls the direction of GPIO1. 0=input, 1=output

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

```c
FT_STATUS FT_WriteGPIO(
    FT_HANDLE ftHandle,
    UINT32 u32Mask,
    UINT32 u32Data
)
```

**Summary**
Sets the status of GPIO0 and GPIO1

**Parameters**
- `ftHandle`: A handle to the device
- `u32Mask`: mask to select the bits that are to be written. 1=write, 0=ignore
- `u32Data`: data to write the GPIO status. Bit0 and bit1 hold the value to be written to the GPIO pins; 1=high, 0=low. Bits in input mode are ignored

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

```c
FT_STATUS FT_ReadGPIO(  
    FT_HANDLE ftHandle,  
    UINT32 *pu32Data  
)
```

**Summary**
Returns the status of GPIO0 and GPIO1

**Parameters**
- `ftHandle`: A handle to the device.
- `pu32Data`: Pointer to received GPIO status data. Bit0 and bit1 reflect the GPIO pin status; 1=high, 0=low. Bits in output mode are ignored.

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.29 FT_SetGPIOPull

**Summary**
Set GPIO internal pull resistors. This API is available only for RevB parts or later.

**Parameters**
- `ftHandle`: A handle to the device.
- `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.
2.30 FT_SetNotificationCallback

<table>
<thead>
<tr>
<th>FT_STATUS</th>
<th>FT_SetNotificationCallback(</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT_HANDLE ftHandle,</td>
<td></td>
</tr>
<tr>
<td>FT_NOTIFICATION_CALLBACK pCallback,</td>
<td></td>
</tr>
<tr>
<td>PVOID pvCallbackContext)</td>
<td></td>
</tr>
</tbody>
</table>

**Summary**
Sets a receive notification callback function which will be called when data is available for IN endpoints where no read requests are currently ongoing.

**Parameters**
- **ftHandle**: A handle to the device.
- **pCallback**: A pointer to the 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.

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

**Remarks**
The callback 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.

```c
VOID (*FT_NOTIFICATION_CALLBACK)(PVOID pvCallbackContext, UCHAR ucPipeID, ULONG ulRecvNotificationLength);
```

- **pvCallbackContext**: A pointer to the user context used when FT_SetNotificationCallback was called.
- **ucPipeID**: The IN pipe where data is available for reading.
- **ulRecvNotificationLength**: Number of bytes available for reading.

When the chip configuration has notifications turned on for specific pipe/s, the application must not actively call FT_ReadPipe. It should register a callback function using FT_SetNotificationCallback. It should only call FT_ReadPipe when the callback function is called. The registered callback function will be called by the driver once firmware sends a notification (about data availability on a notification-enabled pipe) on the notification pipe 0x81. The callback function will be called with parameters describing the pipe ID and the data size. Using this information, applications can either read this data or flush/ignore this data.

The notification feature caters for short unexpected data, 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 FIFO Master. For example, a customer can use 2 channel configuration (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 FIFO master, etc.). A notification feature can be set on Pipe 0x83. In this configuration applications will actively read on the data pipe 0x82 and passively read on pipe 0x83.
2.31 FT_ClearNotificationCallback

```c
FT_STATUS
FT_ClearNotificationCallback(
    FT_HANDLE ftHandle
)
```

**Summary**
Clears the notification callback set by FT_SetNotificationCallback

**Parameters**
- `ftHandle` A handle to the device.

**Return Value**
- `FT_OK` if successful, otherwise the return value is an FT error code.
2.32 FT_GetChipConfiguration

```
FT_STATUS FT_GetChipConfiguration(
    FT_HANDLE ftHandle,
    PVOID pvConfiguration
)
```

**Summary**

Returns the chip configuration.

**Parameters**

- `ftHandle` A handle to the device.
- `pvConfiguration` Pointer to a configuration structure that will contain the chip configuration. For the FT60x, use `FT_60XCONFIGURATION`.

**Return Value**

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

**Remarks**

A utility application called FT60X Chip Configuration Programmer, which is available [here](#), can be used to query and modify the chip's configuration.

For detailed information about the configuration please refer to [AN_370 Configuration Programmer Guide](#).
2.33 FT_SetChipConfiguration

```c
FT_STATUS
FT_SetChipConfiguration(
    FT_HANDLE ftHandle,
    PVOID pvConfiguration
)
```

**Summary**
This API can be used to modify the configurable fields in the chip configuration.

**Parameters**
- `ftHandle` A handle to the device
- `pvConfiguration` Pointer to a configuration structure that contains the chip configuration. For FT60X, use `FT_60XCONFIGURATION`. If NULL, the configuration will be reset to default configuration. Refer to `FT_GetChipConfiguration` for the details of the default configuration.

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

**Remarks**
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 `FT_Close` and `FT_Create`, respectively.

For detailed information about the configuration parameters please refer to [AN_370 Configuration Programmer Guide](#).

A utility application called FT60X Chip Configuration Programmer, which is available [here](#), can be used to query and modify 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 `FT_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 bytes should be converted to a String Descriptor when added to the StringDescriptors field of the `FT_60XCONFIGURATION` structure. Refer to the code in the next page for the sample code.
BOOL SetChipConfiguration()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    FT_60XCONFIGURATION oConfigurationData = { 0 };

    ftStatus = FT_Create(0, FT_OPEN_BY_INDEX, &ftHandle);
    oConfigurationData.VendorID = CONFIGURATION_DEFAULT_VENDORID;
    oConfigurationData.ProductID = CONFIGURATION_DEFAULT_PRODUCTID_601;
    oConfigurationData.PowerAttributes = CONFIGURATION_DEFAULT_POWERATTRIBUTES;
    oConfigurationData.PowerConsumption = CONFIGURATION_DEFAULT_POWERCONSUMPTION;
    oConfigurationData.FIFOClock = CONFIGURATION_DEFAULT_FIFOCLOCK;
    oConfigurationData.BatteryChargingGPIOConfig = CONFIGURATION_DEFAULT_BATTERYCHARGING;
    oConfigurationData.MSIO_Control = CONFIGURATION_DEFAULT_MSIOCONTROL;
    oConfigurationData.GPIO_Control = CONFIGURATION_DEFAULT_GPIOCONTROL;
    oConfigurationData.Reserved = 0;
    oConfigurationData.Reserved2 = 0;
    oConfigurationData.FlashEEPROMDetection = 0;
    oConfigurationData.ChannelConfig = CONFIGURATION_CHANNEL_CONFIG_1;
    oConfigurationData.OptionalFeatureSupport = CONFIGURATION_OPTIONAL_FEATURE_DISABLECANCELSESSIONUNDERRUN;
    SetStringDescriptors(oConfigurationData.StringDescriptors, sizeof(oConfigurationData.StringDescriptors),"MyCompany", "This Is My Product Description", "1234567890ABCde");
    FT_SetChipConfiguration(ftHandle, &oConfigurationData);
    FT_Close(ftHandle);
    return TRUE;
}

BOOL SetStringDescriptors(UCHAR* pStringDescriptors, ULONG ulSize, CONST CHAR* pManufacturer, CONST CHAR* pProductDescription, CONST CHAR* pSerialNumber)
{
    LONG lLen = 0; UCHAR bLen = 0; UCHAR* pPtr = pStringDescriptors;
    // Manufacturer: Should be 15 bytes maximum printable characters
    lLen = strlen(pManufacturer);
    if ((lLen < 1 || lLen > 16) return FALSE;
    for (LONG i = 0; i < lLen; i++) if (!isprint(pManufacturer[i])) return FALSE;

    // Product Description: Should be 31 bytes maximum printable characters
    lLen = strlen(pProductDescription);
    if ((lLen < 1 || lLen > 32) return FALSE;
    for (LONG i = 0; i < lLen; i++) if (!isprint(pProductDescription[i])) return FALSE;

    // Serial Number: Should be 15 bytes maximum alphanumeric characters
    lLen = strlen(pSerialNumber);
    if ((lLen < 1 || lLen > 16) return FALSE;
    for (LONG i = 0; i < lLen; i++) if (!isalnum(pSerialNumber[i])) return FALSE;

    // Manufacturer
    bLen = strlen(pManufacturer);
    pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
    for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++) {
        pPtr[i] = pManufacturer[j]; pPtr[i + 1] = \0;
    }
    pPtr += pPtr[0];

    // Product Description
    bLen = strlen(pProductDescription);
    pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
    for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++) {
        pPtr[i] = pProductDescription[j]; pPtr[i + 1] = \0;
    }
    pPtr += pPtr[0];

    // Serial Number
    bLen = strlen(pSerialNumber);
    pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
    for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++) {
        pPtr[i] = pSerialNumber[j]; pPtr[i + 1] = \0;
    }
    return TRUE;
}
2.34 FT_IsDevicePath

```c
FT_STATUS FT_IsDevicePath(
    FT_HANDLE ftHandle,
    CONST CHAR* pucDevicePath
)
```

**Summary**
Verifies if device path provided corresponds to the device path of the device handle.

**Parameters**
- `ftHandle`: A handle to the device
- `pucDevicePath`: Pointer to the null-terminated string containing the device path.

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

**Remarks**
When the user calls the Windows API RegisterDeviceNotification to wait for a device-related notification, such as device unplugging and plugging, it has to use a GUID to register a device. The GUID for D3XX devices is

```c
// {D1E8FE6A-AB75-4D9E-97D2-06FA22C7736C}
DEFINE_GUID(GUID_DEVINTERFACE_FOR_D3XX,
    0xd1e8fe6a, 0xab75, 0x4d9e, 0x97, 0xd2, 0x6, 0xfa, 0x22, 0xc7, 0x73, 0x6c);
```

Note that this GUID is different from D2XX devices which is

```c
// {219D0508-57A8-4ff5-97A1-BD86587C6C7E} // D2XX
DEFINE_GUID(GUID_DEVINTERFACE_FOR_D2XX,
    0x219d0508, 0x57a8, 0x4ff5, 0x97, 0xa1, 0xbd, 0x86, 0x58, 0x7c, 0x6c, 0x7e);
```

When WM_DEVICECHANGE event is received, it will be impossible to determine the correct device the event is for, assuming there are multiple D3XX devices connected to the machine. In order to distinguish between 2 or more D3XX devices, this function can be used, as each device will have its own unique device path. As such, the function can check if the device being unplugged is the device currently being processed.
2.35 FT_GetDriverVersion

```
FT_STATUS
FT_GetDriverVersion (  
   FT_HANDLE ftHandle,  
   LPDWORD lpdwVersion 
)
```

**Summary**
Returns the D3XX kernel driver version number.

**Parameters**
- **ftHandle**: A handle to the device
- **lpdwVersion**: Pointer to the version number.

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

**Remarks**
A version number contains a major version number, minor version and build/SVN version. Byte 0 and 1 (least significant) holds the build/SVN version. Byte 2 holds the minor version. Byte 3 holds the major version.
2.36 FT_GetLibraryVersion

```
FT_STATUS
FT_GetLibraryVersion ( 
    LPDWORD lpdwVersion
)
```

**Summary**

Returns the D3XX user driver library version number.

**Parameters**

- `ftHandle`: A handle to the device
- `lpdwVersion`: Pointer to the version number.

**Return Value**

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

**Remarks**

A version number contains a major version number, minor version and build/SVN version. Byte 0 and 1 (least significant) holds the build/SVN version. Byte 2 holds the minor version. Byte 3 holds the major version.
2.37 FT_CycleDevicePort

```
FT_STATUS
FT_CycleDevicePort (
    FT_HANDLE ftHandle
)
```

**Summary**

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

**Parameters**

- `ftHandle`: A handle to the device

**Return Value**

- `FT_OK` if successful, otherwise the return value is an FT error code.
2.38 FT_SetSuspendTimeout

```
FT_STATUS
FT_SetSuspendTimeout ( 
    FT_HANDLE ftHandle, 
    ULONG Timeout
)
```

**Summary**

Configures USB Selective suspend timeout. By default the driver has the suspend feature enabled with an idle timeout of 10sec. This API can be used to override the default values. However the modified values are valid only for the life cycle of the ftHandle. A new FT_Create call will reset the idle timeout to driver default values. When the notification feature is enabled, suspend will be disabled hence this API will fail when the notification feature is enabled.

**Parameters**

- `ftHandle` A handle to the device
- `Timeout` Timeout in Seconds.
  - When set to 0, USB selective suspend will be disabled.
  - When set to non-zero, USB selective suspend is configured to trigger after this idle timeout.

**Return Value**

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

The modified values are valid only for the life cycle of the ftHandle. A new FT_Create call will reset the idle timeout to driver default values.
2.39 FT_GetSuspendTimeout

```c
FT_STATUS FT_GetSuspendTimeout (FT_HANDLE ftHandle,
PULONG pTimeout)
```

**Summary**
Returns the configured idle timeout value for USB Selective suspend.

**Parameters**
- `ftHandle` : A handle to the device
- `pTimeout` : Return Timeout in Seconds.

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

Head Office – Glasgow, UK
Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758
E-mail (Sales) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com

Branch Office – Tigard, Oregon, USA
Future Technology Devices International Limited (USA)
7130 SW Fir Loop
Tigard, OR 97223-8160
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987
E-mail (Sales) us.sales@ftdichip.com
E-mail (Support) us.support@ftdichip.com
E-mail (General Enquiries) us.admin@ftdichip.com

Branch Office – Taipei, Taiwan
Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan, R.O.C.
Tel: +886 (0) 2 8797 1330
Fax: +886 (0) 2 8751 9737
E-mail (Sales) tw.sales1@ftdichip.com
E-mail (Support) tw.support1@ftdichip.com
E-mail (General Enquiries) tw.admin1@ftdichip.com

Branch Office – Shanghai, China
Future Technology Devices International Limited (China)
Room 1103, No. 666 West Huaihai Road,
Shanghai, 200052
China
Tel: +86 21 62351596
Fax: +86 21 62351595
E-mail (Sales) cn.sales@ftdichip.com
E-mail (Support) cn.support@ftdichip.com
E-mail (General Enquiries) cn.admin@ftdichip.com

Web Site
http://ftdichip.com

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

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Future Technology Devices International Ltd (FTDI) devices incorporated in their systems, meet all applicable safety, regulatory and system-level performance requirements. All application-related information in this document (including application descriptions, suggested FTDI devices and other materials) is provided for reference only. While FTDI has taken care to assure it is accurate, this information is subject to customer confirmation, and FTDI disclaims all liability for system designs and for any applications assistance provided by FTDI. Use of FTDI devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless FTDI from any and all damages, claims, suits or expense resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder. Future Technology Devices International Ltd, Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow G41 1HH, United Kingdom. Scotland Registered Company Number: SC136640
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, 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 is not efficient so D3XX improves the D2XX protocol by 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_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 you are processing one buffer, another request is already ongoing, thereby improving the gap between each request.

<table>
<thead>
<tr>
<th>Asynchronous Transfer</th>
<th>D3XX</th>
<th>D2XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>YES, via API</td>
<td>NO</td>
</tr>
<tr>
<td>Read</td>
<td>YES, via API</td>
<td>YES, via polling</td>
</tr>
</tbody>
</table>

Streaming Transfer Design

In addition, D3XX provides an FT_SetStreamPipe function as a supplement to the FT_WritePipe and FT_ReadPipe. This informs the chip that the host will be reading or writing a specific number of bytes. When this is used, FT_WritePipe and FT_ReadPipe no longer sends a session command to the chip because chip already knows how much data is requested. This is a feature that should be used together with asynchronous transfers.
Type Definitions

UCHAR Unsigned char
USHORT Unsigned short
ULONG Unsigned long

FT_STATUS
FT_OK = 0
FT_INVALID_HANDLE = 1
FT_DEVICE_NOT_FOUND = 2
FT_DEVICE_NOT_OPENED = 3
FT_IO_ERROR = 4
FT_INVALID_ARGUMENTS = 5
FT_INVALID_PARAMETER = 6
FT_INVALID_BAUD_RATE = 7
FT_DEVICE_NOT_OPENED_FOR_ERASE = 8
FT_DEVICE_NOT_OPENED_FOR_WRITE = 9
FT_FAILED_TO_WRITE_DEVICE = 10
FT_EEPROM_READ_FAILED = 11
FT_EEPROM_WRITE_FAILED = 12
FT_EEPROM_ERASE_FAILED = 13
FT_EEPROM_NOT_PRESENT = 14
FT_EEPROM_NOT_PROGRAMMED = 15
FT_INVALID_ARGS = 16
FT_NOT_SUPPORTED = 17
FT_NO_MORE_ITEMS = 18
FT_TIMEOUT = 19
FT_OPERATION_ABORTED = 20
FT_RESERVED_PIPE = 21
FT_INVALID_CONTROL_REQUEST_DIRECTION = 22
FT_INVALID_CONTROL_REQUEST_TYPE = 23
FT_IO_PENDING = 24
FT_IO_INCOMPLETE = 25
FT_HANDLE_EOF = 26
FT_BUSY = 27
FT_NO_SYSTEM_RESOURCES = 28
FT_DEVICE_LIST_NOT_READY = 29
FT_DEVICETE_NOT_CONNECTED = 30
FT_INCOMPLETE_DEVICE_PATH = 31
FT_OTHER_ERROR = 32

FT_DEVICE
FT_DEVICE_UNKNOWN = 3
FT_DEVICE_600 = 600
FT_DEVICE_601 = 601

FT_FLAGS (See FT_GetDeviceInfoDetail)
FT_FLAGS_OPENED = 1
FT_FLAGS_HIGH_SPEED = 2
FT_FLAGS_SUPER_SPEED = 4

FT_PIPE_TYPE (See FT_GetPipeInformation)
FTPipeTypeControl = 0
FTPipeTypeIsochronous = 1
FTPipeTypeBulk = 2
FTPipeTypeInterrupt = 3
Flags (see FT_ListDevices)
  FT_LIST_NUMBER_ONLY = 0x80000000
  FT_LIST_BY_INDEX = 0x40000000
  FT_LIST_ALL = 0x20000000

Flags (see FT_OpenEx)
  FT_OPEN_BY_SERIAL_NUMBER = 0x00000001
  FT_OPEN_BY_DESCRIPTION = 0x00000002
  FT_OPEN_BY_LOCATION = 0x00000004
  FT_OPEN_BY_GUID = 0x00000008
  FT_OPEN_BY_INDEX = 0x00000010

Flags (See FT_EnableGPIO / FT_WriteGPIO / FT_ReadGPIO)
  FT_GPIO_DIRECTION_IN = 0
  FT_GPIO_DIRECTION_OUT = 1
  FT_GPIO_VALUE_LOW = 0
  FT_GPIO_VALUE_HIGH = 1
  FT_GPIO_0 = 0
  FT_GPIO_1 = 1

Flags (See FT_SetNotificationCallback)
  E_FT_NOTIFICATION_CALLBACK_TYPE_DATA = 0
  E_FT_NOTIFICATION_CALLBACK_TYPE_GPIO = 1

Flags (See FT_SetChipConfiguration / FT_GetChipConfiguration)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEALL = 0
  CONFIGURATION_OPTIONAL_FEATURE_ENABLEBATTERYCHARGING = (0x1 << 0)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLECANCELSESSIONUNDERRUN = (0x1 << 1)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLENOTIFICATIONMESSAGE_INCH1 = (0x1 << 2)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLENOTIFICATIONMESSAGE_INCH2 = (0x1 << 3)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLENOTIFICATIONMESSAGE_INCH3 = (0x1 << 4)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLENOTIFICATIONMESSAGE_INCH4 = (0x1 << 5)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLENOTIFICATIONMESSAGE_INCHALL = (0xF << 2)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEUNDERRUN_INCH1 = (0x1 << 6)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEUNDERRUN_INCH2 = (0x1 << 7)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEUNDERRUN_INCH3 = (0x1 << 8)
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEUNDERRUN_INCH4 = (0x1 << 9)
  CONFIGURATION_OPTIONAL_FEATURE_SUPPORT_ENABLE_FIFO_IN_SUSPEND = (1 << 10)
  // available in RevB parts only */
  CONFIGURATION_OPTIONAL_FEATURE_SUPPORT_DISABLE_CHIP_POWERDOWN = (1 << 11)
  // available in RevB parts only */
  CONFIGURATION_OPTIONAL_FEATURE_DISABLEUNDERRUN_INCHALL = (0xF << 6)
  CONFIGURATION_OPTIONAL_FEATURE_ENABLEALL = 0xFFFF

//
// Common descriptor header
//
typedef struct _FT_COMMON_DESCRIPTOR
{
  UCHAR  bLength;
  UCHAR  bDescriptorType;
} FT_COMMON_DESCRIPTOR, *PFT_COMMON_DESCRIPTOR;

//
// Device descriptor
//
typedef struct _FT_DEVICE_DESCRIPTOR
{
    UCHAR   bLength;
    UCHAR   bDescriptorType;
    USHORT  bcdUSB;
    UCHAR   bDeviceClass;
    UCHAR   bDeviceSubClass;
    UCHAR   bDeviceProtocol;
    UCHAR   bMaxPacketSize0;
    USHORT  idVendor;
    USHORT  idProduct;
    USHORT  bcdDevice;
    UCHAR   iManufacturer;
    UCHAR   iProduct;
    UCHAR   iSerialNumber;
    UCHAR   bNumConfigurations;
}
FT_DEVICE_DESCRIPTOR, *PFT_DEVICE_DESCRIPTOR;

//
// Configuration descriptor
//
typedef struct _FT_CONFIGURATION_DESCRIPTOR
{
    UCHAR   bLength;
    UCHAR   bDescriptorType;
    USHORT  wTotalLength;
    UCHAR   bNumInterfaces;
    UCHAR   bConfigurationValue;
    UCHAR   iConfiguration;
    UCHAR   bmAttributes;
    UCHAR   MaxPower;
}
FT_CONFIGURATION_DESCRIPTOR, *PFT_CONFIGURATION_DESCRIPTOR;

//
// Interface descriptor
//
typedef struct _FT_INTERFACE_DESCRIPTOR
{
    UCHAR   bLength;
    UCHAR   bDescriptorType;
    UCHAR   bInterfaceNumber;
    UCHAR   bAlternateSetting;
    UCHAR   bNumEndpoints;
    UCHAR   bInterfaceClass;
    UCHAR   bInterfaceSubClass;
    UCHAR   bInterfaceProtocol;
    UCHAR   iInterface;
}
FT_INTERFACE_DESCRIPTOR, *PFT_INTERFACE_DESCRIPTOR;

//
// String descriptor
//
typedef struct _FT_STRING_DESCRIPTOR
{
    UCHAR   bLength;
    UCHAR   bDescriptorType;
    WCHAR   szString[256];
}
} FT_STRING_DESCRIPTOR, *PFT_STRING_DESCRIPTOR;

//
// Pipe information
//
typedef struct _FT_PIPE_INFORMATION
{
    FT_PIPE_TYPE PipeType;
    UCHAR PipeId;
    USHORT MaximumPacketSize;
    UCHAR Interval;
} FT_PIPE_INFORMATION, *PFT_PIPE_INFORMATION;

//
// Control setup packet
//
typedef struct _FT_SETUP_PACKET
{
    UCHAR RequestType;
    UCHAR Request;
    USHORT Value;
    USHORT Index;
    USHORT Length;
} FT_SETUP_PACKET, *PFT_SETUP_PACKET;

//
// Notification callback information data
//
typedef struct _FT_NOTIFICATION_CALLBACK_INFO_DATA
{
    ULONG ulRecvNotificationLength;
    UCHAR ucEndpointNo;
} FT_NOTIFICATION_CALLBACK_INFO_DATA;

//
// Notification callback information gpio
//
typedef struct _FT_NOTIFICATION_CALLBACK_INFO_GPIO
{
    BOOL bGPIO0;
    BOOL bGPIO1;
} FT_NOTIFICATION_CALLBACK_INFO_GPIO;

//
// Chip configuration structure
//
typedef struct
{
    // Device Descriptor
    USHORT VendorID;
    USHORT ProductID;

    // String Descriptors

UCHAR StringDescriptors[128];

// Configuration Descriptor
UCHAR bInterval; // Interrupt interval (Valid Range 1-16) /* Reserved for RevA */
UCHAR PowerAttributes;
USHORT PowerConsumption;

// Data Transfer Configuration
UCHAR Reserved2;
UCHAR FIFOClock;
UCHAR FIFOMode;
UCHAR ChannelConfig;

// Optional Feature Support
USHORT OptionalFeatureSupport;
UCHAR BatteryChargingGPIOConfig;
UCHAR FlashEEPROMDetection; // Read-only

// MSIO and GPIO Configuration
ULONG MSIO_Control;
ULONG GPIO_Control;

} FT_60XCONFIGURATION, *PFT_60XCONFIGURATION;
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 or using API `FT_SetChipConfiguration()`.

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

Achieving maximum performance

In 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.

<table>
<thead>
<tr>
<th>Channel Configuration</th>
<th>Burst Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 channels</td>
<td>4</td>
</tr>
<tr>
<td>2 channels</td>
<td>8</td>
</tr>
<tr>
<td>1 channel</td>
<td>16</td>
</tr>
<tr>
<td>1 channel with 1 OUT pipe</td>
<td>16</td>
</tr>
<tr>
<td>1 channel with 1 IN pipe</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 2 – FT60X EPC Burst Size

<table>
<thead>
<tr>
<th>Channel Configuration</th>
<th>FIFO Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 channels</td>
<td>1024</td>
</tr>
<tr>
<td>2 channels</td>
<td>2048</td>
</tr>
<tr>
<td>1 channel</td>
<td>4096</td>
</tr>
<tr>
<td>1 channel with 1 OUT pipe</td>
<td>8192</td>
</tr>
<tr>
<td>1 channel with 1 IN pipe</td>
<td>8192</td>
</tr>
</tbody>
</table>

Table 3 - FT60X FIFO Ping/Pong Request Size

In order to maximize performance, FTDI advises customers to consider the following in the design of their FPGA and host-side application for 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 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 performance of D3XX and FT60X.

1. Chip is configured to 1 channel with 1 IN pipe only.
2. Application opens the device using `FT_Create` and then enables streaming mode using `FT_SetStreamMode`.
3. Application initially sends 3 asynchronous requests for 3 frame buffers of size 2560x1440x4 = 14,745,600 bytes each using `FT_ReadPipe`. Application can use any queue size other than 3 but 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 FIFO on the last segment.
5. The driver completes the request for 1 frame and application call to `FT_GetOverlappedResult` 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 user stops the transfer in which case it will call `FT_AbortPipe` to cancel all outstanding requests in the driver before calling `FT_ClearStreamMode` and `FT_Close`.

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

### Code Samples

```c
#include "stdafx.h"
#include <initguid.h> // For DEFINE_GUID
// // Define when linking with static library
// Undefine when linking with dynamic library
// #define FTD3XX_STATIC
// // Include D3XX library
// #include "FT60X\include\FTD3XX.h"
#pragma comment(lib, "FTD3XX.lib")

// Device Interface GUID.
DEFINE_GUID(GUID_DEVINTERFACE_FOR_D3XX,
0xd1e8fe6a, 0xab75, 0x4d9e, 0x97, 0xd2, 0x6a, 0xfa, 0x22, 0xc7, 0x73, 0x6c);

// Demonstration of querying of USB descriptors
BOOL DescriptorTest()
{
    FT_DEVICE_DESCRIPTOR DeviceDescriptor = {0};
    FT_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor = {0};
    FT_INTERFACE_DESCRIPTOR InterfaceDescriptor = {0};
    FT_PIPE_INFORMATION Pipe;
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE fhHandle;
    GUID DeviceGUID[2] = {0};

    // Open a device handle by GUID
    memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &fhHandle);
    if (FT_FAILED(ftStatus))
    {
        // Handle error
    }
    else
    {
        // Query device descriptors
        // ...
    }
}
```
FT_Close(ftHandle);
return FALSE;
}

// Get configuration descriptor
// to determine the number of interfaces (bNumConfigurations) in the configuration
//
ftStatus = FT_GetDeviceDescriptor(ftHandle, &DeviceDescriptor);
if (FT_FAILED(ftStatus))
{
    FT_Close(ftHandle);
    return FALSE;
}

// Get configuration descriptor
// to determine the number of interfaces (bNumInterfaces) in the configuration
//
ftStatus = FT_GetConfigurationDescriptor(ftHandle, &ConfigurationDescriptor);
if (FT_FAILED(ftStatus))
{
    FT_Close(ftHandle);
    return FALSE;
}

for (int j=0; j<ConfigurationDescriptor.bNumInterfaces; j++)
{
    // Get interface descriptor
    // of 2nd interface (interface[1]) to get number of pipes
    // The 1st interface is reserved for FT60X protocol design to maximize USB3.0 performance
    //
    ftStatus = FT_GetInterfaceDescriptor(ftHandle, j, 0, &InterfaceDescriptor);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    for (int i=0; i<InterfaceDescriptor.bNumEndpoints; i++)
    {
        // Get pipe information
        // to get endpoint number and endpoint type
        //
        ftStatus = FT_GetPipeInformation(ftHandle, j, 0, i, &Pipe);
        if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }
    }
}

// Close device handle
//
FT_Close(ftHandle);
return TRUE;
BOOL LoopbackTest()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    GUID DeviceGUID[2] = {0};

    // Open a device handle by GUID
    memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);
    if (FT_FAILED(ftStatus))
    {
        return FALSE;
    }

    // Write and read loopback transfer
    DWORD dwNumIterations = 10;
    for (DWORD i=0; i<dwNumIterations; i++)
    {
        // Write to channel 1 ep 0x02
        UCHAR acWriteBuf[BUFFER_SIZE] = {0xFF};
        ULONG ulBytesWritten = 0;
        ftStatus = FT_WritePipe(ftHandle, 0x02, acWriteBuf, sizeof(acWriteBuf), &ulBytesWritten, NULL);
        if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }

        // Read from channel 1 ep 0x82
        UCHAR acReadBuf[BUFFER_SIZE] = {0xAA};
        ULONG ulBytesRead = 0;
        ftStatus = FT_ReadPipe(ftHandle, 0x82, acReadBuf, sizeof(acReadBuf), &ulBytesRead, NULL);
        if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }

        // Compare bytes read with bytes written
        if (memcmp(acWriteBuf, acReadBuf, sizeof(acReadBuf)))
        {
            FT_Close(ftHandle);
            return FALSE;
        }
    }

    // Close device handle
    FT_Close(ftHandle);
    return TRUE;
}
BOOL AsyncLoopbackTest()
{
   FT_STATUS ftStatus = FT_OK;
   FT_HANDLE ftHandle;
   GUID DeviceGUID[2] = {0};

   // Open device by GUID
   memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
   ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);

   // Write and read loopback transfer
   DWORD dwNumIterations = 10;
   for (DWORD i=0; i<dwNumIterations; i++)
   {
      // Write to channel 1 ep 0x02
      UCHAR acWriteBuf[BUFFER_SIZE] = {0xFF};
      ULONG ulBytesWritten = 0;
      ULONG ulBytesToWrite = sizeof(acWriteBuf);
      {
         // Create the overlapped io event for asynchronous transfer
         OVERLAPPED vOverlappedWrite = {0};
         vOverlappedWrite.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

         // Write asynchronously
         // FT_WritePipe is a blocking/synchronous function.
         // To make it unblocking/asynchronous operation, vOverlapped parameter is supplied.
         // When FT_WritePipe is called with overlapped io,
         // the function will immediately return with FT_IO_PENDING
         ftStatus = FT_WritePipe(ftHandle, 0x02, acWriteBuf, ulBytesToWrite, &ulBytesWritten,
                        &vOverlappedWrite);
         if (ftStatus == FT_IO_PENDING)
         {
            // Poll until all data requested ulBytesToWrite is sent
            do
            {
               // FT_GetOverlappedResult will return FT_IO_INCOMPLETE if not yet finish
               ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlappedWrite, &ulBytesWritten,
                           FALSE);
               if (ftStatus == FT_IO_INCOMPLETE)
               {
                  continue;
               }
               else if (FT_FAILED(ftStatus))
               {
                  CloseHandle(vOverlappedWrite.hEvent);
                  FT_Close(ftHandle);
                  return FALSE;
               }
               else //if (ftStatus == FT_OK)
               {
                  // exit now
                  break;
               }
            }
            while (1);
         }
      }
      // Delete the overlapped io event
      CloseHandle(vOverlappedWrite.hEvent);
   }
}
// Read from channel 1 ep 0x82
//
UCHAR acReadBuf[BUFFER_SIZE] = {0xAA};
ULONG ulBytesRead = 0;
ULONG ulBytesToRead = sizeof(acReadBuf);
{
  // Create the overlapped io event for asynchronous transfer
  OVERLAPPED vOverlappedRead = {0};
vOverlappedRead.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

  // Read asynchronously
  // FT_ReadPipe is a blocking/synchronous function.
  // To make it unblocking/asynchronous operation, vOverlapped parameter is supplied.
  // When FT_ReadPipe is called with overlapped io, the function will immediately return
  if (FT_FAILED(ftStatus = FT_ReadPipe(ftHandle, 0x82, acReadBuf, ulBytesToRead, &ulBytesRead, &vOverlappedRead)))
    continue;
else
  {
    ftStatus = FT_GetOverlappedResult(ftHandle, &vOverlappedRead, &ulBytesRead, FALSE);
    if (FT_FAILED(ftStatus))
      continue;
  } else if (FT_FAILED(ftStatus))
  {
    CloseHandle(vOverlappedRead.hEvent);
    FT_Close(ftHandle);
    return FALSE;
  } else //if (ftStatus == FT_OK)
  {
    // exit now
    break;
  }
}
// Delete the overlapped io event
CloseHandle(vOverlappedRead.hEvent);

// Compare bytes read with bytes written
//
if (memcmp(acWriteBuf, acReadBuf, sizeof(acReadBuf)))
{
  FT_Close(ftHandle);
  return FALSE;
}

// Close device
//
FT_Close(ftHandle);

return TRUE;
BOOL ChipConfigurationTest()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    GUID DeviceGUID[2] = {0};

    // Open a device handle by GUID
    memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    // Get chip configuration
    FT_D3XXCONFIGURATION oConfigurationData = {0};
    ftStatus = FT_GetChipConfiguration(ftHandle, &oConfigurationData);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    // Set chip configuration
    oConfigurationData.FIFOMode = FIFO_MODE_600;
    oConfigurationData.ChannelConfig = CHANNEL_CONFIG_4;
    oConfigurationData.OptionalFeatureSupport = OPTIONAL_FEATURE_SUPPORT_DISABLECANCELSESSIONUNDERRUN;
    ftStatus = FT_SetChipConfiguration(ftHandle, &oConfigurationData);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    // Close device handle
    FT_Close(ftHandle);
    return TRUE;
}
BOOL NotificationDataTest()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    GUID DeviceGUID[2] = {0};
    USERCONTEXT UserContext = {0};
    UCHAR ucSendBuffer[LOOPBACK_DATA] = {0};
    BOOL bResult = TRUE;

    // // Enable notification message feature
    //
    if (!EnableNotificationMessage())
    {
        return FALSE;
    }

    // // Open a device handle by GUID
    //
    memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    // // Set/register the callback function
    //
    UserContext.m_ftHandle = ftHandle;
    ftStatus = FT_SetNotificationCallback(ftHandle, NotificationCallback, &UserContext);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }

    // // Readback data using notification message
    //
    ULONG ulBytesTransferred = 0;
    DEBUG(_T("\n\nWriting %d bytes
"), sizeof(ucSendBuffer));
    ftStatus = FT_WritePipe(ftHandle, 0x02, ucSendBuffer, sizeof(ucSendBuffer),
&ulBytesTransferred, NULL);
    if (FT_FAILED(ftStatus))
    {
        bResult = FALSE;
        goto exit;
    }
    DEBUG(_T("\n\nWriting %d bytes DONE
"), ulBytesTransferred);
    while (UserContext.m_ulCurrentRecvData != LOOPBACK_DATA && UserContext.m_ftStatus == FT_OK)
    {
        Sleep(1);
    }
    if (memcmp(ucSendBuffer, UserContext.m_ucRecvBuffer, LOOPBACK_DATA))
    {
        bResult = FALSE;
        goto exit;
    }

    // // Return success
    //
    return TRUE;
}

// // Enable notification message feature
//
// EnableNotificationMessage()
//
BOOL EnableNotificationMessage()
{
    // // Call FTDI function
    //
    FT_STATUS ftStatus = FT_OK;
    ftStatus = FT_EnableNotificationMessage();
    if (FT_FAILED(ftStatus))
    {
        return FALSE;
    }

    // // Return success
    //
    return TRUE;
}
exit:

    // Clear/unregister the callback function
    //
    FT_ClearNotificationCallback(ftHandle);

    // Close device handle
    //
    FT_Close(ftHandle);
    ftHandle = NULL;

    return bResult;
}

static VOID NotificationCallback(PVOID pvCallbackContext, E_FT_NOTIFICATION_CALLBACK_TYPE eCallbackType, PVOID pvCallbackInfo)
{
    switch (eCallbackType)
    {
    case E_FT_NOTIFICATION_CALLBACK_TYPE_DATA:
    {
        FT_NOTIFICATION_CALLBACK_INFO_DATA* pInfo = (FT_NOTIFICATION_CALLBACK_INFO_DATA*)pvCallbackInfo;
        if (pInfo)
        {
            PUSER_CONTEXT pUserContext = (PUSER_CONTEXT)pvCallbackContext;
            ULONG ulBytesTransferred = 0;
            DEBUG(_T("\n\tReading %d bytes!\n"), pInfo->ulRecvNotificationLength);
            FT_STATUS ftStatus = FT_ReadPipe(pUserContext->m_ftHandle,
                pInfo->ucEndpointNo,
                &pUserContext->m_ucRecvBuffer[pUserContext->m_ftHandle - 1],
                &ulBytesTransferred,
                NULL);

            if (FT_FAILED(ftStatus))
            {
                DEBUG(_T("NotificationCallback FT_ReadPipe failed 0x%x\n"), ftStatus);
            }
            else
            {
                pUserContext->m_ulCurrentRecvData += ulBytesTransferred;
                DEBUG(_T("\n\tReading %d bytes DONE!\n"), ulBytesTransferred);
            }
            pUserContext->m_status = ftStatus;
        }
        break;
    }
    default:
    {
        break;
    }
    }
}
static BOOL EnableNotificationMessage()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    GUID DeviceGUID[2] = {0};
    FT_60XCONFIGURATION oConfigurationData = {0};

    // Open a device handle by GUID
    // memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);
    if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }

    // Get configuration
    // ftStatus = FT_GetChipConfiguration(ftHandle, &oConfigurationData);
    if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }

    // Enable notification message for IN pipe for all channels
    oConfigurationData.OptionalFeatureSupport |= OPTIONAL_FEATURE_SUPPORT_ENABLENOTIFICATIONMESSAGE_INCHALL;

    // Set configuration
    // ftStatus = FT_SetChipConfiguration(ftHandle, &oConfigurationData);
    if (FT_FAILED(ftStatus))
        {
            FT_Close(ftHandle);
            return FALSE;
        }

    // Close device handle
    // FT_Close(ftHandle);

    // After setting configuration, device will reboot
    // Wait for about 5 seconds for device and driver be ready
    // Sleep(5000);
    return TRUE;
}
BOOL GPIOTest()
{
    FT_STATUS ftStatus = FT_OK;
    FT_HANDLE ftHandle;
    BOOL bResult = TRUE;
    UINT32 u32Data = 0;

    // Open a device handle by GUID
    // memcpy(&DeviceGUID[0], &GUID_DEVINTERFACE_FOR_D3XX, sizeof(GUID));
    ftStatus = FT_Create(&DeviceGUID[0], FT_OPEN_BY_GUID, &ftHandle);
    if (FT_FAILED(ftStatus))
    {
        FT_Close(ftHandle);
        return FALSE;
    }
    if (FT_FAILED(ftStatus))
    {
        bResult = FALSE;
        return FALSE;
    }
    // Get GPIO status
    ftStatus = FT_ReadGPIO(ftHandle, &u32Data);
    if (FT_FAILED(ftStatus))
    {
        CMD_LOG(_T("FT_ReadGPIO failed\n"));
        bResult = FALSE;
        goto exit;
    }
    CMD_LOG(_T("Initial GPIO bitmap : %d\n"), u32Data);
    CMD_LOG(_T("moving both the GPIOs to Output mode\n"));
    ftStatus = FT_EnableGPIO(ftHandle, 0x3, 0x3); // bit 0 and 1 both set.
    if (FT_FAILED(ftStatus))
    {
        CMD_LOG(_T("FT_EnableGPIO failed\n"));
        bResult = FALSE;
        goto exit;
    }
    // Get GPIO status
    ftStatus = FT_ReadGPIO(ftHandle, &u32Data);
    if (FT_FAILED(ftStatus))
    {
        CMD_LOG(_T("FT_ReadGPIO failed\n"));
        bResult = FALSE;
        goto exit;
    }
    CMD_LOG(_T("GPIO bitmap after EnableGPIO : %d\n"), u32Data);
    CMD_LOG(_T("Making both the GPIO high\n"));
    // set both the GPIOs to high.
    ftStatus = FT_WriteGPIO(ftHandle, 0x3, 0x3);
    if (FT_FAILED(ftStatus))
    {
        CMD_LOG(_T("FT_WriteGPIO failed\n"));
        bResult = FALSE;
        goto exit;
    }
    // Get GPIO status
    ftStatus = FT_ReadGPIO(ftHandle, &u32Data);
}
if (FT_FAILED(ftStatus))
{
    CMD_LOG(_T("\t FT_ReadGPIO failed\n"));
    bResult = FALSE;
    goto exit;
}
CMD_LOG(_T("\t GPIO bitmap after FT_WriteGPIO : %d\n"), u32Data);

exit:
// // Close device handle
// FT_Close(ftHandle);
ftHandle = NULL;

    return bResult;
}
// Set chip configuration using the structure created
ftStatus = FT_SetChipConfiguration(ftHandle, &oConfigurationData);
if (ftStatus == FT_INVALID_PARAMETER)
{
    FT_Close(ftHandle);
    return FALSE;
}

FT_Close(ftHandle);
return TRUE;

static BOOL SetStringDescriptors(
    UCHAR* pStringDescriptors, ULONG ulSize,
    CONST CHAR* pManufacturer, CONST CHAR* pProductDescription, CONST CHAR* pSerialNumber)
{
    LONG lLen = 0; UCHAR bLen = 0;
    UCHAR* pPtr = pStringDescriptors;
    if (ulSize != 128 || pStringDescriptors == NULL)
        return FALSE;
    if (pManufacturer == NULL || pProductDescription == NULL || pSerialNumber == NULL)
        return FALSE;

    // Verify input parameters
    {
        // Manufacturer: Should be 15 bytes maximum printable characters
        lLen = strlen(pManufacturer);
        if (lLen < 1 || lLen >= 16)
            return FALSE;
        for (LONG i = 0; i < lLen; i++)
            if (!isprint(pManufacturer[i]))
                return FALSE;

        // Product Description: Should be 31 bytes maximum printable characters
        lLen = strlen(pProductDescription);
        if (lLen < 1 || lLen >= 32)
            return FALSE;
        for (LONG i = 0; i < lLen; i++)
            if (!isprint(pProductDescription[i]))
                return FALSE;

        // Serial Number: Should be 15 bytes maximum alphanumeric characters
        lLen = strlen(pSerialNumber);
        if (lLen < 1 || lLen >= 16)
            return FALSE;
        for (LONG i = 0; i < lLen; i++)
            if (!isalnum(pSerialNumber[i]))
                return FALSE;
    }

    // Construct the string descriptors
    {
        // Manufacturer
        bLen = strlen(pManufacturer);
        pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
        for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++)
        {
            pPtr[i] = pManufacturer[j];
            pPtr[i + 1] = '\0';
        }
        pPtr += pPtr[0];

        // Product Description
        bLen = strlen(pProductDescription);
        pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
    }
for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++) {
    pPtr[i] = pProductDescription[j];
    pPtr[i + 1] = '\0';
}

pPtr += pPtr[0];

// Serial Number
bLen = strlen(pSerialNumber);
pPtr[0] = bLen * 2 + 2; pPtr[1] = 0x03;
for (LONG i = 2, j = 0; i < pPtr[0]; i += 2, j++) {
    pPtr[i] = pSerialNumber[j];
    pPtr[i + 1] = '\0';
}

return TRUE;
## Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Terms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamically Linked Library</td>
</tr>
<tr>
<td>D3XX</td>
<td>FTDI's proprietary &quot;direct&quot; driver interface via FTD3XX.DLL</td>
</tr>
<tr>
<td>EP</td>
<td>Endpoint</td>
</tr>
<tr>
<td>EPC</td>
<td>Endpoint Controller</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>LIB</td>
<td>Static Library</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>
Appendix B – List of Tables & Figures

List of Tables

Table 1 - FT600 Series Function Protocol Interfaces and Endpoints ................................................................. 5
Table 2 – FT60X EPC Burst Size ………………………………………………………………………………………………………… 60
Table 3 - FT60X FIFO Ping/Pong Request Size …………………………………………………………………………………………………… 60

List of Figures

Figure 1 - D3XX Driver Architecture ………………………………………………………………………………………………………… 5
Figure 2 - Device with Default String Descriptors …………………………………………………………………………………………………… 13
Figure 3 - Device with Customized String Descriptors …………………………………………………………………………………………………… 13
# Appendix C – Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Changes</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Initial Release</td>
<td>2015-08-25</td>
</tr>
<tr>
<td>1.1</td>
<td>Added APIs for multiple device feature</td>
<td>2015-12-23</td>
</tr>
<tr>
<td>1.3</td>
<td>Added APIs ((FT_GetDriverVersion, FT_GetLibraryVersion) for multiple device feature</td>
<td>2016-01-28</td>
</tr>
<tr>
<td></td>
<td>Updated FT_ListDevices, FT_GetDeviceInfoList to remove D2XX-related information</td>
<td>2016-01-28</td>
</tr>
<tr>
<td>1.4</td>
<td>Added FT_CycleDevicePort, FT_ResetDevicePort and Achieving Maximum Performance</td>
<td>2016-07-12</td>
</tr>
<tr>
<td></td>
<td>Updated FT_SetNotificationCallback</td>
<td>2016-07-12</td>
</tr>
<tr>
<td></td>
<td>Added FT_SetPipeTimeout, FT_GetPipeTimeout, FT_SetSuspendTimeout, FT_GetSuspendTimeout</td>
<td>2016-07-12</td>
</tr>
<tr>
<td></td>
<td>Replaced FT_SetGPIO, FT_GetGPIO with FT_EnableGPIO, FT_WriteGPIO, FT_ReadGPIO calls.</td>
<td>2016-07-12</td>
</tr>
<tr>
<td></td>
<td>Updated FT_ReadPipe</td>
<td>2016-07-12</td>
</tr>
<tr>
<td></td>
<td>Added a new section Constants Definition as part of the Appendix A</td>
<td>2016-07-12</td>
</tr>
<tr>
<td>1.5</td>
<td>Removed Deprecated APIs.</td>
<td>2016-11-01</td>
</tr>
<tr>
<td></td>
<td>Added FT_SetGPIOPull API.</td>
<td>2016-11-01</td>
</tr>
<tr>
<td></td>
<td>Updated the sample codes</td>
<td>2016-11-01</td>
</tr>
<tr>
<td>1.6</td>
<td>Updated chapter number of AN_412 on pages 14,16 &amp; 18</td>
<td>2017-06-07</td>
</tr>
<tr>
<td>1.7</td>
<td>Update for FT_Create API; Added FT_WritePipeEx and FT_ReadPipeEx</td>
<td>2018-03-28</td>
</tr>
</tbody>
</table>