FTD2XX Programmer's Guide

Version 2.01

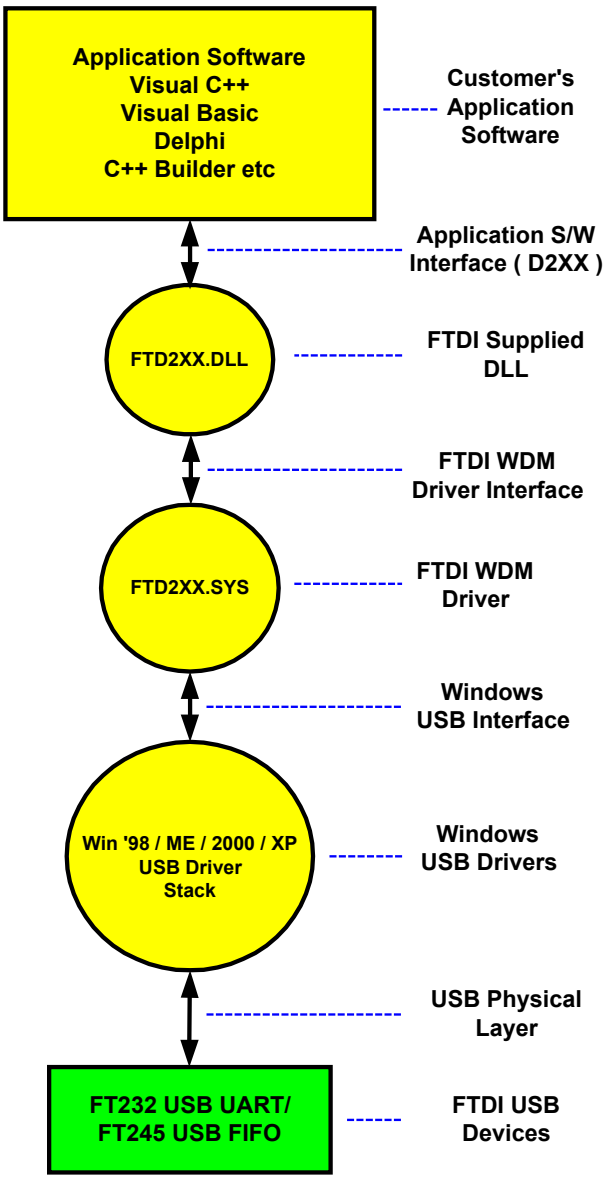
Introduction to FTDI's D2XX 2.0 Driver Technology

FTDI's "D2XX Direct Drivers" for Windows offer an alternative solution to our VCP drivers which allows application software to interface with FT232 USB UART and FT245 USB FIFO devices using a DLL instead of a Virtual Com Port. The architecture of the D2XX drivers consists of a Windows WDM driver that communicates with the device via the Windows USB Stack and a DLL which interfaces the Application Software (written in VC++, C++ Builder, Delphi, VB etc.) to the WDM driver. An INF installation file, Uninstaller program and D2XX Programmers Guide complete the package.

The new version of the D2XX drivers contains many enhanced features and has been divided into four groups for clarity. The Classic Interface Section documents the original D2XX functions that are retained in this new release. The Classic Interface provides a simple, easy to use, set of functions to access these FTDI USB devices. New sections are "The EEPROM Interface" which allows application software to read / program the various fields in the 93C46 EEPROM including a user defined area which can be used for application specific purposes; "The FT232BM / FT245BM Enhancements" which allow control of the additional features in our 2nd generation devices, and the "FT-Win32 API" which is a more sophisticated alternative to the Classic Interface – our equivalent to the native Win 32 API calls that are used to control a legacy serial port. Using the FT-Win32 API, existing Windows legacy Comms applications can easily be converted to use the D2XX interface simply by replacing the standard Win32 API calls with the equivalent FT-Win32 API calls.

Please Note – the Classic Interface and the FT-Win32 API interface are alternatives. Developers should choose one or the other – the two sets of functions should not be mixed.

D2XX Driver Architecture



1. "Classic Interface" Functions

D2XX Classic Programming Interface – Introduction

An FTD2XX device is an FT232 USB UART or FT245 USB FIFO interfacing to Windows application software using FTDI's WDM driver FTD2XX.SYS. The FTD2XX.SYS driver has a programming interface exposed by the dynamic link library FTD2XX.DLL, and this document describes that interface.

D2XX Classic Programming Interface – Overview

FT_ListDevices returns information about the FTDI devices currently connected. In a system with multiple devices this can be used to decide which of the devices the application software wishes to access (using FT_OpenEx below).

Before the device can be accessed, it must first be opened. **FT_Open** and **FT_OpenEx** return a handle that is used by all functions in the Classic Programming Interface to identify the device. When the device has been opened successfully, I/O can be performed using **FT_Read** and **FT_Write**. When operations are complete, the device is closed using **FT_Close**.

Once opened, additional functions are available to reset the device (**FT_ResetDevice**); purge receive and transmit buffers (**FT_Purge**); set receive and transmit timeouts (**FT_SetTimeouts**); get the receive queue status (**FT_GetQueueStatus**); get the device status (**FT_GetStatus**); set and reset the break condition (**FT_SetBreakOn**, **FT_SetBreakOff**); and set conditions for event notification (**FT_SetEventNotification**).

For FT232 devices, functions are available to set the baud rate (**FT_SetBaudRate**), and set a non-standard baud rate (**FT_SetDivisor**); set the data characteristics such as word length, stop bits and parity (**FT_SetDataCharacteristics**); set hardware or software handshaking (**FT_SetFlowControl**); set modem control signals (**FT_SetDTR**, **FT_ClrDTR**, **FT_SetRTS**, **FT_ClrRTS**); get modem status (**FT_GetModemStatus**); set special characters such as event and error characters (**FT_SetChars**). For FT245 devices, these functions are redundant and can effectively be ignored.

D2XX Classic Programming Interface – Reference

The functions that make up the D2XX Classic Programming Interface are defined in this section. Type definitions of the functional parameters and return codes used in the D2XX Classic Programming Interface are contained in the Appendix.

FT_ListDevices

Get information concerning the devices currently connected. This function can return such information as the number of devices connected, and device strings such as serial number and product description.

FT_STATUS **FT_ListDevices** (PVOID *pvArg1*,PVOID *pvArg2*, DWORD *dwFlags*)

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.

Remarks

This function can be used in a number of ways to return different types of information.

In its simplest form, it can be used to return the number of devices currently connected. If *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 string information. If *FT_OPEN_BY_SERIAL_NUMBER* bit is set in *dwFlags*, the serial number string will be returned from this function. If *FT_OPEN_BY_DESCRIPTION* bit is set in *dwFlags*, the product description string will be returned from this function. If neither of these bits is 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* bit is 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 be used to return device string information for all connected devices. If *FT_LIST_ALL* bit is 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.

Examples

Sample code shows how to get the number of devices currently connected.

```
FT_STATUS ftStatus;
DWORD numDevs;

ftStatus = FT_ListDevices(&numDevs,NULL,FT_LIST_NUMBER_ONLY);
if (ftStatus == FT_OK) {
    // FT_ListDevices OK, number of devices connected is in numDevs
}
else {
    // FT_ListDevices failed
}
```

This sample shows how to get the serial number of the first device found. Note that indexes are zero-based. If more than one device is connected, incrementing devIndex will get the serial number of each connected device in turn.

```
FT_STATUS ftStatus;
DWORD devIndex = 0;
char Buffer[16];

ftStatus = FT_ListDevices((PVOID)devIndex,Buffer,FT_LIST_BY_INDEX|FT_OPEN_BY_SERIAL_NUMBER);
if (FT_SUCCESS(ftStatus)) {
    // FT_ListDevices OK, serial number is in Buffer
}
else {
    // FT_ListDevices failed
}
```

This sample shows how to get the product descriptions of all the devices currently connected.

```
FT_STATUS ftStatus;
char *BufPtrs[3];           // pointer to array of 3 pointers
char Buffer1[64];           // buffer for product description of first device found
char Buffer2[64];           // buffer for product description of second device
DWORD numDevs;

// initialize the array of pointers
BufPtrs[0] = Buffer1;
BufPtrs[1] = Buffer2;
BufPtrs[2] = NULL;          // last entry should be NULL

ftStatus = FT_ListDevices(BufPtrs,&numDevs,FT_LIST_ALL|FT_OPEN_BY_DESCRIPTION);
if (FT_SUCCESS(ftStatus)) {
    // FT_ListDevices OK, product descriptions are in Buffer1 and Buffer2, and
    // numDevs contains the number of devices connected
}
else {
    // FT_ListDevices failed
}
```

FT_Open

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

FT_STATUS **FT_Open** (int *iDevice*, FT_HANDLE **ftHandle*)

Parameters

iDevice

Must be 0 if only one device is attached. For multiple devices 1, 2 etc.

ftHandle

Pointer to a variable of type FT_HANDLE 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.

Remarks

Although this function can be used to open multiple devices by setting *iDevice* to 0, 1, 2 etc. there is no ability to open a specific device. To open named devices, use the function **FT_OpenEx**.

Example

This sample shows how to open a device.

```
FT_HANDLE ftHandle;
FT_STATUS ftStatus;
ftStatus = FT_Open(0,&ftHandle);
if (ftStatus == FT_OK) {
    // FT_Open OK, use ftHandle to access device
}
else {
    // FT_Open failed
}
```


FT_OpenEx

Open the named device and return a handle which will be used for subsequent accesses. The device name can be its serial number or device description.

FT_STATUS **FT_OpenEx** (PVOID *pvArg1*, DWORD *dwFlags*, FT_HANDLE **ftHandle*)

Parameters

pvArg1

Meaning depends on dwFlags, but it will normally be interpreted as a pointer to a null terminated string.

dwFlags

FT_OPEN_BY_SERIAL_NUMBER or FT_OPEN_BY_DESCRIPTION.

ftHandle

Pointer to a variable of type FT_HANDLE 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.

Remarks

This function should be used to open multiple devices. Multiple devices can be opened at the same time if they can be distinguished by serial number or device description.

Example

These samples show how to open two devices simultaneously.

Suppose one device has serial number "FT000001", and the other has serial number "FT999999".

```
FT_STATUS ftStatus;
FT_HANDLE ftHandle1;
FT_HANDLE ftHandle2;

ftStatus = FT_OpenEx("FT000001",FT_OPEN_BY_SERIAL_NUMBER,&ftHandle1);
if (ftStatus == FT_OK) {
      // FT_OpenEx OK, device with serial number "FT000001" is open
}
else {
      // FT_OpenEx failed
}

ftStatus = FT_OpenEx("FT999999",FT_OPEN_BY_SERIAL_NUMBER,&ftHandle2);
if (ftStatus == FT_OK) {
      // FT_OpenEx OK, device with serial number "FT999999" is open
}
else {
      // FT_OpenEx failed
}
```


Suppose one device has description "USB HS SERIAL CONVERTER", and the other has description "USB PUMP CONTROLLER".

```
FT_STATUS ftStatus;
FT_HANDLE ftHandle1;
FT_HANDLE ftHandle2;

ftStatus = FT_OpenEx("USB HS SERIAL CONVERTER",FT_OPEN_BY_DESCRIPTION,&ftHandle1);
if (ftStatus == FT_OK) {
      // FT_OpenEx OK, device with description "USB HS SERIAL CONVERTER" is open
}
else {
      // FT_OpenEx failed
}

ftStatus = FT_OpenEx("USB PUMP CONTROLLER",FT_OPEN_BY_DESCRIPTION,&ftHandle2);
if (ftStatus == FT_OK) {
      // FT_OpenEx OK, device with description "USB PUMP CONTROLLER" is open
}
else {
      // FT_OpenEx failed
}
```

FT_Close

Close an open device.

FT_STATUS **FT_Close** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device to close.

Return Value

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

FT_Read

Read data from the device.

FT_STATUS **FT_Read** (FT_HANDLE *ftHandle,* LPVOID *lpBuffer,* DWORD *dwBytesToRead,* LPDWORD *lpdwBytesReturned*)

Parameters

ftHandle

Handle of the device to read.

lpBuffer

Pointer to the buffer that receives the data from the device.

dwBytesToRead

Number of bytes to be read from the device.

lpdwBytesReturned

Pointer to a variable of type DWORD which receives the number of bytes read from the device.

Return Value

FT_OK if successful, FT_IO_ERROR otherwise.

Remarks

FT_Read always returns the number of bytes read in *lpdwBytesReturned*.

This function does not return until *dwBytesToRead* have been read into the buffer. The number of bytes in the receive queue can be determined by calling **FT_GetStatus** or **FT_GetQueueStatus**, and passed to **FT_Read** as *dwBytesToRead* so that the function reads the device and returns immediately.

When a read timeout value has been specified in a previous call to **FT_SetTimeouts**, **FT_Read** returns when the timer expires or *dwBytesToRead* have been read, whichever occurs first. If the timeout occurred, **FT_Read** reads available data into the buffer and returns *FT_OK*.

An application should use the function return value and *lpdwBytesReturned* when processing the buffer. If the return value is *FT_OK*, and *lpdwBytesReturned* is equal to *dwBytesToRead* then **FT_Read** has completed normally. If the return value is *FT_OK*, and *lpdwBytesReturned* is less then *dwBytesToRead* then a timeout has occurred, and the read has been partially completed. Note that if a timeout occurred and no data was read, the return value is still *FT_OK*.

A return value of *FT_IO_ERROR* suggests an error in the parameters of the function, or a fatal error like USB disconnect has occurred.

Example

This sample shows how to read all the data currently available.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
DWORD EventDWord;
DWORD RxBytes;
DWORD TxBytes;
DWORD BytesReceived;
char RxBuffer[256];

FT_GetStatus(ftHandle,&RxBytes,&TxBytes,&EventDWord);

if (RxBytes > 0) {
	ftStatus = FT_Read(ftHandle,RxBuffer,RxBytes,&BytesReceived);
	if (ftStatus == FT_OK) {
		// FT_Read OK
	}
	else {
		// FT_Read Failed
	}
}
```

This sample shows how to read with a timeout of 5 seconds.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
DWORD BytesReceived;
char RxBuffer[256];

FT_SetTimeouts(ftHandle,5000,0);

ftStatus = FT_Read(ftHandle,RxBuffer,RxBytes,&BytesReceived);
if (ftStatus == FT_OK) {
	if (BytesReceived == RxBytes) {
		// FT_Read OK
	}
	else {
		// FT_Read Timeout
	}
}
else {
	// FT_Read Failed
}
```

FT_Write

Write data to the device.

FT_STATUS **FT_Write** (FT_HANDLE *ftHandle,* LPVOID *lpBuffer,* DWORD *dwBytesToWrite,* LPDWORD *lpdwBytesWritten*)

Parameters

ftHandle

Handle of the device to write.

lpBuffer

Pointer to the buffer that contains the data to be written to the device.

dwBytesToWrite

Number of bytes to write to the device.

lpdwBytesWritten

Pointer to a variable of type DWORD which receives the number of bytes written to the device.

Return Value

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

FT_ResetDevice

This function sends a reset command to the device.

FT_STATUS **FT_ResetDevice** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device to reset.

Return Value

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

FT_SetBaudRate

This function sets the baud rate for the device.

FT_STATUS **FT_SetBaudRate** (FT_HANDLE *ftHandle,* DWORD *dwBaudRate*)

Parameters

ftHandle

Handle of the device.

dwBaudRate

Baud rate.

Return Value

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

FT_SetDivisor

This function sets the baud rate for the device. It is used to set non-standard baud rates.

FT_STATUS **FT_SetDivisor** (FT_HANDLE *ftHandle,* USHORT *usDivisor*)

Parameters

ftHandle

Handle of the device.

usDivisor

Divisor.

Return Value

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

Remarks

The application note "Setting Baud rates for the FT8U232AM", which is available on our web site www.ftdichip.com, describes how to calculate the divisor for a non standard baud rate.

Example

Suppose we want to set a baud rate of 5787 baud. A simple calculation suggests that a divisor of 4206 should work.

```
FT_HANDLE ftHandle;  // handle of device obtained from FT_Open or FT_OpenEx
FT_STATUS ftStatus;

ftStatus = FT_SetDivisor(ftHandle,0x4206);
if (ftStatus == FT_OK) {
      // FT_SetDivisor OK, baud rate has been set to 5787 baud
}
else {
      // FT_SetDivisor failed
}
```

FT_SetDataCharacteristics

This function sets the data characteristics for the device.

FT_STATUS **FT_SetDataCharacteristics** (FT_HANDLE *ftHandle,* UCHAR *uWordLength,* UCHAR *uStopBits,*UCHAR *uParity*)

Parameters

ftHandle

Handle of the device.

uWordLength

Number of bits per word - must be FT_BITS_8 or FT_BITS_7.

uStopBits

Number of stop bits - must be FT_STOP_BITS_1 or FT_STOP_BITS_2.

UParity

FT_PARITY_NONE, FT_PARITY_ODD, FT_PARITY_EVEN, FT_PARITY_MARK, FT_PARITY_SPACE.

Return Value

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

FT_SetFlowControl

This function sets the flow control for the device.

FT_STATUS **FT_SetFlowControl** (FT_HANDLE *ftHandle,* USHORT us*FlowControl,* UCHAR u*Xon,*UCHAR u*Xoff*)

Parameters

ftHandle

Handle of the device.

usFlowControl

Must be one of FT_FLOW_NONE, FT_FLOW_RTS_CTS, FT_FLOW_DTR_DSR, FT_FLOW_XON_XOFF

uXon

Character used to signal XON. Only used if flow control is FT_FLOW_XON_XOFF.

uXoff

Character used to signal XOFF. Only used if flow control is FT_FLOW_XON_XOFF.

Return Value

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

FT_SetDTR

This function sets the Data Terminal Ready (DTR) control signal.

FT_STATUS **FT_SetDTR** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to set DTR.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_SetDTR(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetDTR OK
}
else {
    // FT_SetDTR failed
}
```

FT_ClrDTR

This function clears the Data Terminal Ready (DTR) control signal.

FT_STATUS **FT_ClrDTR** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to clear DTR.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_ClrDTR(ftHandle);
if (ftStatus == FT_OK) {
    // FT_ClrDTR OK
}
else {
    // FT_ClrDTR failed
}
```


FT_SetRTS

This function sets the Request To Send (RTS) control signal.

FT_STATUS **FT_SetRTS** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to set RTS.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_SetRTS(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetRTS OK
}
else {
    // FT_SetRTS failed
}
```

FT_ClrRTS

This function clears the Request To Send (RTS) control signal.

FT_STATUS **FT_ClrRTS** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to clear RTS.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_ClrRTS(ftHandle);
if (ftStatus == FT_OK) {
    // FT_ClrRTS OK
}
else {
    // FT_ClrRTS failed
}
```

FT_GetModemStatus

Gets the modem status from the device.

FT_STATUS **FT_GetModemStatus** (FT_HANDLE *ftHandle,* LPDWORD *lpdwModemStatus*)

Parameters

ftHandle

Handle of the device to read.

lpdwModemStatus

Pointer to a variable of type DWORD which receives the modem status from the device.

Return Value

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

FT_SetChars

This function sets the special characters for the device.

FT_STATUS **FT_SetChars** (FT_HANDLE *ftHandle,* UCHAR *uEventCh,* UCHAR *uEventChEn,* UCHAR *uErrorCh,* UCHAR *uErrorChEn*)

Parameters

ftHandle

Handle of the device.

uEventCh

Event character.

uEventChEn

0 if event character is disabled, non-zero otherwise.

uErrorCh

Error character.

uErrorChEn

0 if error character is disabled, non-zero otherwise.

Return Value

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

FT_Purge

This function purges receive and transmit buffers in the device.

FT_STATUS **FT_Purge** (FT_HANDLE *ftHandle,* DWORD *dwMask*)

Parameters

ftHandle

Handle of the device.

dwMask

Any combination of FT_PURGE_RX and FT_PURGE_TX.

Return Value

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

FT_SetTimeouts

This function sets the read and write timeouts for the device.

FT_STATUS **FT_SetTimeouts** (FT_HANDLE *ftHandle,* DWORD *dwReadTimeout,* DWORD *dwWriteTimeout*)

Parameters

ftHandle

Handle of the device.

dwReadTimeout

Read timeout in milliseconds.

dwWriteTimeout

Write timeout in milliseconds.

Return Value

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

Example

This sample shows how to set a read timeout of 5 seconds and a write timeout of 1 second.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_SetTimeouts(ftHandle,5000,1000);
if (ftStatus == FT_OK) {
    // FT_SetTimeouts OK
}
else {
    // FT_SetTimeouts failed
}
```

FT_GetQueueStatus

Gets the number of characters in the receive queue.

FT_STATUS **FT_GetQueueStatus** (FT_HANDLE *ftHandle,* LPDWORD *lpdwAmountInRxQueue*)

Parameters

ftHandle

Handle of the device to read.

lpdwAmountInRxQueue

Pointer to a variable of type DWORD which receives the number of characters in the receive queue.

Return Value

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

FT_SetBreakOn

Sets the BREAK condition for the device.

FT_STATUS **FT_SetBreakOn** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to set the BREAK condition for the device.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_SetBreakOn(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetBreakOn OK
}
else {
    // FT_SetBreakOn failed
}
```

FT_SetBreakOff

Resets the BREAK condition for the device.

FT_STATUS **FT_SetBreakOff** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

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

Example

This sample shows how to reset the BREAK condition for the device.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;
ftStatus = FT_SetBreakOff(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetBreakOff OK
}
else {
    // FT_SetBreakOff failed
}
```


FT_GetStatus

Gets the device status including number of characters in the receive queue, number of characters in the transmit queue, and the current event status.

FT_STATUS **FT_GetStatus** (FT_HANDLE *ftHandle,* LPDWORD *lpdwAmountInRxQueue*, LPDWORD *lpdwAmountInTxQueue*, LPDWORD *lpdwEventStatus*)

Parameters

ftHandle

Handle of the device to read.

lpdwAmountInRxQueue

Pointer to a variable of type DWORD which receives the number of characters in the receive queue.

lpdwAmountInTxQueue

Pointer to a variable of type DWORD which receives the number of characters in the transmit queue.

lpdwEventStatus

Pointer to a variable of type DWORD which receives the current state of the event status.

Return Value

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

Remarks

For an example of how to use this function, see the sample code in *FT_SetEventNotification*.

FT_SetEventNotification

Sets conditions for event notification.

FT_STATUS **FT_SetEventNotification** (FT_HANDLE *ftHandle,* DWORD *dwEventMask,* PVOID *pvArg*)

Parameters

ftHandle

Handle of the device.

dwEventMask

Conditions that cause the event to be set.

pvArg

Interpreted as a handle of an event

Return Value

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

Remarks

An application can use this function to setup conditions which allow a thread to block until one of the conditions is met. Typically, an application will create an event, call this function, then block on the event. When the conditions are met, the event is set, and the application thread unblocked.

dwEventMask is a bit-map that describes the events the application is interested in. *pvArg* is interpreted as the handle of an event which has been created by the application. If one of the event conditions is met, the event is set.

If *FT_EVENT_RXCHAR* is set in *dwEventMask*, the event will be set when a character has been received by the device. If *FT_EVENT_MODEM_STATUS* is set in *dwEventMask*, the event will be set when a change in the modem signals has been detected by the device.

Example

This example shows how to wait for a character to be received or a change in modem status.

First, create the event and call *FT_SetEventNotification*.

```
FT_HANDLE ftHandle;  // handle of an open device
FT_STATUS ftStatus;

HANDLE hEvent;
DWORD EventMask;

hEvent = CreateEvent(
             NULL,
             false,  // auto-reset event
             false,  // non-signalled state
             ""
             );

EventMask = FT_EVENT_RXCHAR | FT_EVENT_MODEM_STATUS;

ftStatus = FT_SetEventNotification(ftHandle,EventMask,hEvent);
```

Sometime later, block the application thread by waiting on the event, then when the event has occurred, determine the condition that caused the event, and process it accordingly.

```
WaitForSingleObject(hEvent,INFINITE);

DWORD EventDWord;
DWORD RxBytes;
DWORD TxBytes;

FT_GetStatus(ftHandle,&RxBytes,&TxBytes,&EventDWord);

if (EventDWord & FT_EVENT_MODEM_STATUS) {

      // modem status event detected, so get current modem status

      FT_GetModemStatus(ftHandle,&Status);

      if (Status & 0x00000010) {
             // CTS is high
      }
       else {
             // CTS is low
      }

      if (Status & 0x00000020) {
             // DSR is high
      }
       else {
             // DSR is low
      }

}


if (RxBytes > 0) {
      // call FT_Read() to get received data from device
}
```

2. EEPROM Programming Interface Functions

EEPROM Programming Interface – Introduction

FTDI has included EEPROM programming support in the D2XX library, and this document describes that interface.

EEPROM Programming Interface – Overview

Functions are provided to program the EEPROM (**FT_EE_Program**), and read the EEPROM (**FT_EE_Read**). Unused space in the EEPROM is called the User Area (EEUA), and functions are provided to access the EEUA. **FT_EE_UASize** gets its size, **FT_EE_UAWrite** writes data into it, and **FT_EE_UARead** is used to read its contents.

EEPROM Programming Interface - Reference

The EEPROM programming interface functions are described in this section. Type definitions of the functional parameters and return codes used in the D2XX EEPROM programming interface are contained in the Appendix.

FT_EE_Program

Program the EEPROM.

FT_STATUS **FT_EE_Program** (FT_HANDLE *ftHandle*, PFT_PROGRAM_DATA *lpData*)

Parameters

ftHandle

Handle of the device.

lpData

Pointer to structure of type FT_PROGRAM_DATA.

Return Value

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

Remarks

This function interprets the parameter *pvArgs* as a pointer to a struct of type FT_PROGRAM_DATA that contains the data to write to the EEPROM. The data is written to EEPROM, then read back and verified.

If the SerialNumber field in FT_PROGRAM_DATA is NULL, or SerialNumber points to a NULL string, a serial number based on the ManufacturerId and the current date and time will be generated.

If *pvArgs* is NULL, the device will be programmed with the default data
{ 0x0403, 0x6001, "FTDI", "FT", "USB HS Serial Converter", "", 44, 1, 0, 1,
FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, 0 }

Example

```
FT_PROGRAM_DATA ftData = {
	0x0403,
	0x6001,
	"FTDI",
	"FT",
	"USB HS Serial Converter",
	"FT000001",
	44,
	1,
	0,
	1,
	FALSE,
	FALSE,
	FALSE,
	FALSE,
	FALSE,
	FALSE,
	0
	};

FT_HANDLE ftHandle;
FT_STATUS ftStatus = FT_Open(0,&ftHandle);
if (ftStatus == FT_OK) {
	ftStatus = FT_EE_Program(ftHandle,&ftData);
	if (ftStatus == FT_OK) {
		// FT_EE_Program OK!
	}
	else {
		// FT_EE_Program FAILED!
	}
}
```

FT_EE_Read

Read the contents of the EEPROM.

FT_STATUS **FT_EE_Read** (FT_HANDLE *ftHandle*, PFT_PROGRAM_DATA *lpData*)

Parameters

ftHandle

Handle of the device.

lpData

Pointer to struct of type FT_PROGRAM_DATA.

Return Value

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

Remarks

This function interprets the parameter *pvArgs* as a pointer to a struct of type *FT_PROGRAM_DATA* that contains storage for the data to be read from the EEPROM.

The function does not perform any checks on buffer sizes, so the buffers passed in the *FT_PROGRAM_DATA* struct must be big enough to accommodate their respective strings (including null terminators). The sizes shown in the following example are more than adequate and can be rounded down if necessary. The restriction is that the Manufacturer string length plus the Description string length is less than or equal to 40 characters.

Example

```
FT_HANDLE ftHandle;
FT_STATUS ftStatus = FT_Open(0,&ftHandle);
if (ftStatus != FT_OK) {
      // FT_Open FAILED!
}

FT_PROGRAM_DATA ftData;
char ManufacturerBuf[32];
char ManufacturerIdBuf[16];
char DescriptionBuf[64];
char SerialNumberBuf[16];

ftData.Manufacturer = ManufacturerBuf;
ftData.ManufacturerId = ManufacturerIdBuf;
ftData.Description = DescriptionBuf;
ftData.SerialNumber = SerialNumberBuf;

ftStatus = FT_EE_Read(ftHandle,&ftData);
if (ftStatus == FT_OK) {
      // FT_EE_Read OK, data is available in ftData
}
else {
      // FT_EE_Read FAILED!
}
```

FT_EE_UARead

Read the contents of the EEUA.

FT_STATUS **FT_EE_UARead** (FT_HANDLE *ftHandle*, PUCHAR *pucData*, DWORD *dwDataLen*, LPDWORD *lpdwBytesRead*)

Parameters

ftHandle

Handle of the device.

pucData

Pointer to a buffer that contains storage for data to be read.

dwDataLen

Size, in bytes, of buffer that contains storage for data to be read.

lpdwBytesRead

Pointer to a DWORD that receives the number of bytes read.

Return Value

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

Remarks

This function interprets the parameter *pucData* as a pointer to an array of bytes of size *dwDataLen* that contains storage for the data to be read from the EEUA. The actual number of bytes read is stored in the DWORD referenced by *lpdwBytesRead*.

If *dwDataLen* is less than the size of the EEUA, then *dwDataLen* bytes are read into the buffer. Otherwise, the whole of the EEUA is read into the buffer..

An application should check the function return value and *lpdwBytesRead* when **FT_EE_UARead** returns.

Example

```
FT_HANDLE ftHandle;
FT_STATUS ftStatus = FT_Open(0,&ftHandle);
if (ftStatus != FT_OK) {
      // FT_Open FAILED!
}

char Buffer[64];
DWORD BytesRead;

ftStatus = FT_EE_UARead(ftHandle,Buffer,64,&BytesRead);
if (ftStatus == FT_OK) {
      // FT_EE_UARead OK
      // User Area data stored in Buffer
      // Number of bytes read from EEUA stored in BytesRead
}
else {
      // FT_EE_UARead FAILED!
}
```


FT_EE_UAWrite

Write data into the EEUA.

FT_STATUS **FT_EE_UAWrite** (FT_HANDLE *ftHandle*, PUCHAR *pucData*, DWORD *dwDataLen*)

Parameters

ftHandle

Handle of the device.

pucData

Pointer to a buffer that contains the data to be written.

dwDataLen

Size, in bytes, of buffer that contains the data to be written.

Return Value

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

Remarks

This function interprets the parameter *lpData* as a pointer to an array of bytes of size *dwDataLen* that contains the data to be written to the EEUA. It is a programming error for *dwDataLen* to be greater than the size of the EEUA.

Example

```
FT_HANDLE ftHandle;
FT_STATUS ftStatus = FT_Open(0,&ftHandle);
if (ftStatus != FT_OK) {
      // FT_Open FAILED!
}

char *Buffer = "hello, world";
ftStatus = FT_EE_UAWrite(ftHandle,Buffer,12);
if (ftStatus == FT_OK) {
      // FT_EE_UAWrite OK
      // User Area contains "hello, world"
}
else {
      // FT_EE_UAWrite FAILED!
}
```

FT_EE_UASize

Get size of EEUA.

FT_STATUS **FT_EE_UASize** (FT_HANDLE *ftHandle*, LPDWORD *lpdwSize*)

Parameters

ftHandle

Handle of the device.

lpdwSize

Pointer to a DWORD that receives the size, in bytes, of the EEUA.

Return Value

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

Example

```
FT_HANDLE ftHandle;
FT_STATUS ftStatus = FT_Open(0,&ftHandle);
if (ftStatus != FT_OK) {
      // FT_Open FAILED!
}

DWORD EEUA_Size;

ftStatus = FT_EE_UASize(ftHandle,&EEUA_Size);
if (ftStatus == FT_OK) {
      // FT_EE_UASize OK
      // EEUA_Size contains the size, in bytes, of the EEUA
}
else {
      // FT_EE_UASize FAILED!
}
```


3. FT232BM & FT245BM Extended API Functions

Extended Programming Interface - Introduction

FT232BM is FTDI's second generation USB UART IC. FT245BM is FTDI's second generation USB FIFO IC. They offer extra functionality, including programmable features, to their predecessors. The programmable features are supported by extensions to the D2XX driver, and the programming interface is exposed by FTD2XX.DLL.

Extended Programming Interface - Overview

New features include a programmable receive buffer timeout and bit bang mode. The receive buffer timeout is controlled via the latency timer functions **FT_GetLatencyTimer** and **FT_SetLatencyTimer**. Bit bang mode is controlled via the functions **FT_GetBitMode** and **FT_SetBitMode**.

Before these functions can be accessed, the COM port must first be opened. The Win32API function, **CreateFile**, returns a handle that is used by all functions in the programming interface to identify the port. After opening the port successfully, the function **FT_GetDeviceInfo** can be used to get information about the device underlying the port, and to confirm that the port is a virtual COM port.

Extended Programming Interface - Reference

The functions that comprise the FTD2XX programming interface are described in this section. See the Appendix for definitions of data types used in the descriptions of the functions below.

FT_GetLatencyTimer

Get the current value of the latency timer.

FT_STATUS **FT_GetLatencyTimer** (FT_HANDLE *ftHandle*, PUCHAR *pucTimer*)

Parameters

ftHandle

Handle of the device.

pucTimer

Pointer to unsigned char to store latency timer value.

Return Value

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

Remarks

In the FT8U232AM and FT8U245AM devices, the receive buffer timeout that is used to flush remaining data from the receive buffer was fixed at 16 ms. In the FT232BM, this timeout is programmable and can be set at 1 ms intervals between 1ms and 255 ms. This allows the device to be better optimized for protocols requiring faster response times from short data packets.

Example

```
HANDLE ftHandle;     // valid handle returned from FT_W32_CreateFile
FT_STATUS ftStatus;
UCHAR LatencyTimer;
ftStatus = FT_GetLatencyTimer(ftHandle,&LatencyTimer);
if (ftStatus == FT_OK) {
      // LatencyTimer contains current value
}
else {
      // FT_GetLatencyTimer FAILED!
}
```

FT_SetLatencyTimer

Set the latency timer.

FT_STATUS **FT_SetLatencyTimer** (FT_HANDLE *ftHandle*, UCHAR *ucTimer*)

Parameters

ftHandle

Handle of the device.

ucTimer

Required value, in milliseconds, of latency timer. Valid range is 1 - 255.

Return Value

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

Remarks

In the FT8U232AM and FT8U245AM devices, the receive buffer timeout that is used to flush remaining data from the receive buffer was fixed at 16 ms. In the FT232BM, this timeout is programmable and can be set at 1 ms intervals between 1ms and 255 ms. This allows the device to be better optimized for protocols requiring faster response times from short data packets.

Example

```
HANDLE ftHandle;     // valid handle returned from FT_W32_CreateFile
FT_STATUS ftStatus;
UCHAR LatencyTimer = 10;
ftStatus = FT_SetLatencyTimer(ftHandle,LatencyTimer);
if (ftStatus == FT_OK) {
      // LatencyTimer set to 10 milliseconds
}
else {
      // FT_SetLatencyTimer FAILED!
}
```

FT_GetBitMode

Get the current value of the bit mode.

FT_STATUS **FT_GetBitMode** (FT_HANDLE *ftHandle*, PUCHAR *pucMode*)

Parameters

ftHandle

Handle of the device.

pucTimer

Pointer to unsigned char to store bit mode value.

Return Value

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

Remarks

For a description of Bit Bang Mode, see the specification "FT232BM USB UART (USB - Serial) I.C.", DS232B Version 1.0, FTDI Ltd. 2002.

Example

```
HANDLE ftHandle;     // valid handle returned from FT_W32_CreateFile
UCHAR BitMode;
FT_STATUS ftStatus;
ftStatus = FT_GetBitMode(ftHandle,&BitMode);
if (ftStatus == FT_OK) {
      // BitMode contains current value
}
else {
      // FT_GetBitMode FAILED!
}
```

FT_SetBitMode

Set the value of the bit mode.

FT_STATUS **FT_SetBitMode** (FT_HANDLE *ftHandle*, UCHAR *ucMask*, UCHAR *ucEnable*)

Parameters

ftHandle

Handle of the device.

ucMask

Required value for bit mode mask.

ucEnable

Enable value, 0 = FALSE, 1 = TRUE.

Return Value

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

Remarks

For a description of Bit Bang Mode, see the specification "FT232BM USB UART (USB - Serial) I.C.", DS232B Version 1.0, FTDI Ltd. 2002.

Example

```
HANDLE ftHandle;     // valid handle returned from FT_W32_CreateFile
FT_STATUS ftStatus;
UCHAR Mask = 0xff;
UCHAR Enable = 1;
ftStatus = FT_SetBitMode(ftHandle,Mask,Enable);
if (ftStatus == FT_OK) {
      // 0xff written to device
}
else {
      // FT_SetBitMode FAILED!
}
```

FT_SetUSBParameters

Set the USB request transfer size.

FT_STATUS **FT_SetUSBParameters** (FT_HANDLE *ftHandle*, DWORD *dwInTransferSize*, DWORD *dwOutTransferSize*)

Parameters

ftHandle

Handle of the device.

dwInTransferSize

Transfer size for USB IN request.

dwOutTransferSize

Transfer size for USB OUT request.

Return Value

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

Remarks

Previously, USB request transfer sizes have been set at 4096 bytes and have not been configurable. This function can be used to change the transfer sizes to better suit the application's requirements.

Note that, at present, only *dwInTransferSize* is supported.

Example

```
HANDLE ftHandle;     // valid handle returned from FT_W32_CreateFile
FT_STATUS ftStatus;
DWORD InTransferSize = 16384;
ftStatus = FT_SetUSBParameters(ftHandle,InTransferSize,0);
if (ftStatus == FT_OK)
; // In transfer size set to 16 Kbytes
else
; // FT_SetUSBParameters FAILED!
```

4. FT-Win32 API Function Calls

FT-Win32 Programming Interface - Introduction

The D2XX interface now incorporates functions based on Win32 API and Win32 COMM API calls. This facilitates the porting of communications applications from VCP to D2XX.

FT-Win32 Programming Interface - Overview

Before the device can be accessed, it must first be opened. **FT_W32_CreateFile** returns a handle that is used by all functions in the programming interface to identify the device. When the device has been opened successfully, I/O can be performed using **FT_W32_ReadFile** and **FT_W32_WriteFile**. When operations are complete, the device is closed using **FT_W32_CloseHandle**.

FT-Win32 Programming Interface - Reference

The functions that comprise the FTD2XX programming interface are described in this section. See the Appendix for definitions of data types used in the descriptions of the functions below.

FT_W32_CreateFile

Open the named device and return a handle that will be used for subsequent accesses. The device name can be its serial number or device description.

FT_HANDLE **FT_W32_CreateFile**(LPCSTR *lpszName*, DWORD *dwAccess*, DWORD *dwShareMode*, LPSECURITY_ATTRIBUTES *lpSecurityAttributes*, DWORD *dwCreate*, DWORD *dwAttrsAndFlags*, HANDLE *hTemplate*)

Parameters

lpszName

Pointer to a null terminated string that contains the name of the device. The name of the device can be its serial number or description as obtained from the FT_ListDevices function.

dwAccess

Type of access to the device. Access can be GENERIC_READ, GENERIC_WRITE, or both.

dwShareMode

How the device is shared. This value must be set to 0.

lpSecurityAttributes

This parameter has no effect and should be set to NULL.

dwCreate

This parameter must be set to OPEN_EXISTING.

dwAttrsAndFlags

File attributes and flags. This parameter is a combination of FILE_ATTRIBUTE_NORMAL, FILE_FLAG_OVERLAPPED if overlapped I/O is used, FT_OPEN_BY_SERIAL_NUMBER if *lpszName* is the device's serial number, and FT_OPEN_BY_DESCRIPTION if *lpszName* is the device's description.

hTemplate

This parameter must be NULL.

Return Value

If the function is successful, the return value is a handle.
If the function is unsuccessful, the return value is the Win32 error code INVALID_HANDLE_VALUE.

Remarks

This function must be used if overlapped I/O is required.

Example

This example shows how a device can be opened for overlapped I/O using its serial number.

```
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
char Buf[64];

ftStatus = FT_ListDevices(0,Buf,FT_LIST_BY_INDEX|FT_OPEN_BY_SERIAL_NUMBER);

ftHandle = FT_W32_CreateFile(Buf,GENERIC_READ|GENERIC_WRITE,0,0,
                                    OPEN_EXISTING,
                                    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED |
             FT_OPEN_BY_SERIAL_NUMBER,
                                    0);

if (ftHandle == INVALID_HANDLE_VALUE)
      ; // FT_W32_CreateDevice failed
```

This example shows how a device can be opened for non-overlapped I/O using its description.

```
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
char Buf[64];

ftStatus = FT_ListDevices(0,Buf,FT_LIST_BY_INDEX|FT_OPEN_BY_DESCRIPTION);

ftHandle = FT_W32_CreateFile(Buf,GENERIC_READ|GENERIC_WRITE,0,0,
                                    OPEN_EXISTING,
                                    FILE_ATTRIBUTE_NORMAL | FT_OPEN_BY_DESCRIPTION,
                                    0);

if (ftHandle == INVALID_HANDLE_VALUE)
        ; // FT_W32_CreateDevice failed
```

FT_W32_CloseHandle

Close the specified device.

BOOL **FT_W32_CloseHandle** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how to close a device after opening it for non-overlapped I/O using its description.

```
FT_STATUS ftStatus;
FT_HANDLE ftHandle;
char Buf[64];

ftStatus = FT_ListDevices(0,Buf,FT_LIST_BY_INDEX|FT_OPEN_BY_DESCRIPTION);

ftHandle = FT_W32_CreateFile(Buf,GENERIC_READ|GENERIC_WRITE,0,0,
                                 OPEN_EXISTING,
                                 FILE_ATTRIBUTE_NORMAL | FT_OPEN_BY_DESCRIPTION,
                                 0);

if (ftHandle == INVALID_HANDLE_VALUE)
; // FT_W32_CreateDevice failed
else {
      // FT_W32_CreateFile OK, so do some work, and eventually ...
      FT_W32_CloseHandle(ftHandle);
}
```

FT_W32_ReadFile

Read data from the device.

BOOL **FT_W32_ReadFile** (FT_HANDLE *ftHandle*, LPVOID *lpBuffer*, DWORD *dwBytesToRead*, LPDWORD *lpdwBytesReturned*, LPOVERLAPPED *lpOverlapped*)

Parameters

ftHandle

Handle of the device.

lpBuffer

Pointer to a buffer that receives the data from the device.

dwBytesToRead

Number of bytes to be read from the device.

lpdwBytesReturned

Pointer to a variable that receives the number of bytes read from the device.

LpOverlapped

Pointer to an overlapped structure.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Remarks

This function supports both non-overlapped and overlapped I/O.

Non-overlapped I/O

The parameter, *lpOverlapped,* must be NULL for non-overlapped I/O.

This function always returns the number of bytes read in *lpdwBytesReturned*.

This function does not return until *dwBytesToRead* have been read into the buffer. The number of bytes in the receive queue can be determined by calling **FT_GetStatus** or **FT_GetQueueStatus**, and passed as *dwBytesToRead* so that the function reads the device and returns immediately.

When a read timeout has been setup in a previous call to **FT_W32_SetCommTimeouts**, this function returns when the timer expires or *dwBytesToRead* have been read, whichever occurs first. If a timeout occurred, any available data is read into *lpBuffer* and the function returns a non-zero value.

An application should use the function return value and *lpdwBytesReturned* when processing the buffer. If the return value is non-zero and *lpdwBytesReturned* is equal to *dwBytesToRead* then the function has completed normally. If the return value is non-zero and *lpdwBytesReturned* is less then *dwBytesToRead* then a timeout has occurred, and the read request has been partially completed. Note that if a timeout occurred and no data was read, the return value is still non-zero.

A return value of *FT_IO_ERROR* suggests an error in the parameters of the function, or a fatal error like USB disconnect has occurred.

Overlapped I/O

When the device has been opened for overlapped I/O, an application can issue a request and perform some additional work while the request is pending. This contrasts with the case of non-overlapped I/O in which the application issues a request and receives control again only after the request has been completed.

The parameter, *lpOverlapped,* must point to an initialized OVERLAPPED structure.

If there is enough data in the receive queue to satisfy the request, the request completes immediately and the return code is non-zero. The number of bytes read is returned in *lpdwBytesReturned*.

If there is not enough data in the receive queue to satisfy the request, the request completes immediately, and the return code is zero, signifying an error. An application should call **FT_W32_GetLastError** to get the cause of the error. If the error code is ERROR_IO_PENDING, the overlapped operation is still in progress, and the application can perform other processing. Eventually, the application checks the result of the overlapped request by calling **FT_W32_GetOverlappedResult**.

If successful, the number of bytes read is returned in *lpdwBytesReturned*.

Example

This example shows how to read 256 bytes from the device using non-overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for non-overlapped i/o
char Buf[256];
DWORD dwToRead = 256;
DWORD dwRead;

if (FT_W32_ReadFile(ftHandle, Buf, dwToRead, &dwRead, &osWrite)) {
      if (dwToRead == dwRead)
              ; // FT_W32_ReadFile OK
      else
              ; // FT_W32_ReadFile timeout
}
else
      ; // FT_W32_ReadFile failed
```

This example shows how to read 256 bytes from the device using overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for overlapped i/o
char Buf[256];
DWORD dwToRead = 256;
DWORD dwRead;
OVERLAPPED osRead = { 0 };

if (!FT_W32_ReadFile(ftHandle, Buf, dwToRead, &dwRead, &osWrite)) {
      if (FT_W32_GetLastError(ftHandle) == ERROR_IO_PENDING) {
              // write is delayed so do some other stuff until ...
          if (!FT_W32_GetOverlappedResult(ftHandle, &osRead, &dwRead, FALSE))
                  ; // error
              else {
                      if (dwToRead == dwRead)
                              ; // FT_W32_ReadFile OK
                      else
                              ; // FT_W32_ReadFile timeout
              }
      }
}
else {
      // FT_W32_ReadFile OK
}
```

FT_W32_WriteFile

Write data to the device.

BOOL **FT_W32_WriteFile** (FT_HANDLE *ftHandle*, LPVOID *lpBuffer*, DWORD *dwBytesToWrite*, LPDWORD *lpdwBytesWritten*, LPOVERLAPPED *lpOverlapped*)

Parameters

ftHandle

Handle of the device.

lpBuffer

Pointer to the buffer that contains the data to write to the device.

dwBytesToWrite

Number of bytes to be written to the device.

lpdwBytesWritten

Pointer to a variable that receives the number of bytes written to the device.

lpOverlapped

Pointer to an overlapped structure.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Remarks

This function supports both non-overlapped and overlapped I/O.

Non-overlapped I/O

The parameter, *lpOverlapped,* must be NULL for non-overlapped I/O.

This function always returns the number of bytes written in *lpdwBytesWritten*.

This function does not return until *dwBytesToWrite* have been written to the device.

When a write timeout has been setup in a previous call to **FT_W32_SetCommTimeouts**, this function returns when the timer expires or *dwBytesToWrite* have been written, whichever occurs first. If a timeout occurred, *lpdwBytesWritten* contains the number of bytes actually written, and the function returns a non-zero value.

An application should always use the function return value and *lpdwBytesWritten*. If the return value is non-zero and *lpdwBytesWritten* is equal to *dwBytesToWrite* then the function has completed normally. If the return value is non-zero and *lpdwBytesWritten* is less then *dwBytesToWrite* then a timeout has occurred, and the write request has been partially completed. Note that if a timeout occurred and no data was written, the return value is still non-zero.

Overlapped I/O

When the device has been opened for overlapped I/O, an application can issue a request and perform some additional work while the request is pending. This contrasts with the case of non-overlapped I/O in which the application issues a request and receives control again only after the request has been completed.

The parameter, *lpOverlapped,* must point to an initialized OVERLAPPED structure.

This function completes immediately, and the return code is zero, signifying an error. An application should call **FT_W32_GetLastError** to get the cause of the error. If the error code is ERROR_IO_PENDING, the overlapped operation is still in progress, and the application can perform other processing. Eventually, the application checks the result of the overlapped request by calling **FT_W32_GetOverlappedResult**.

If successful, the number of bytes written is returned in *lpdwBytesWritten*.

Example

This example shows how to write 128 bytes to the device using non-overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for overlapped i/o
char Buf[128];              // contains data to write to the device
DWORD dwToWrite = 128;
DWORD dwWritten;

if (FT_W32_WriteFile(ftHandle, Buf, dwToWrite, &dwWritten, &osWrite)) {
      if (dwToWrite == dwWritten)
              ; // FT_W32_WriteFile OK
      else
              ; // FT_W32_WriteFile timeout
}
else
      ; // FT_W32_WriteFile failed
```

This example shows how to write 128 bytes to the device using overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for overlapped i/o
char Buf[128];               // contains data to write to the device
DWORD dwToWrite = 128;
DWORD dwWritten;
OVERLAPPED osWrite = { 0 };

if (!FT_W32_WriteFile(ftHandle, Buf, dwToWrite, &dwWritten, &osWrite)) {
      if (FT_W32_GetLastError(ftHandle) == ERROR_IO_PENDING) {
              // write is delayed so do some other stuff until ...
          if (!FT_W32_GetOverlappedResult(ftHandle, &osWrite, &dwWritten, FALSE))
                      ; // error
              else {
                      if (dwToWrite == dwWritten)
                              ; // FT_W32_WriteFile OK
                      else
                              ; // FT_W32_WriteFile timeout
              }
      }
}
else {
      // FT_W32_WriteFIle OK
}
```

FT_W32_GetLastError

Gets the last error that occurred on the device.

BOOL **FT_W32_GetLastError** (FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Remarks

This function is normally used with overlapped I/O. For a description of its use, see **FT_W32_ReadFile** and **FT_W32_WriteFile**..

FT_W32_GetOverlappedResult

Gets the result of an overlapped operation.

BOOL **FT_W32_GetOverlappedResult** (FT_HANDLE *ftHandle*, LPOVERLAPPED *lpOverlapped*, LPDWORD *lpdwBytesTransferred*, BOOL *bWait*)

Parameters

ftHandle

Handle of the device.

lpOverlapped

Pointer to an overlapped structure.

lpdwBytesTransferred

Pointer to a variable that receives the number of bytes transferred during the overlapped operation.

bWait

Set to TRUE if the function does not return until the operation has been completed.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Remarks

This function is used with overlapped I/O. For a description of its use, see **FT_W32_ReadFile** and **FT_W32_WriteFile**.

FT_W32_ClearCommBreak

Puts the communications line in the non-BREAK state.

BOOL **FT_W32_ClearCommBreak**(FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how put the line in the non-BREAK state.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile

if (!FT_W32_ClearCommBreak(ftHandle))
      ; // FT_W32_ClearCommBreak failed
else
      ; // FT_W32_ClearCommBreak OK
```

FT_W32_ClearCommError

Gets information about a communications error and get current status of the device.

BOOL **FT_W32_ClearCommError**(FT_HANDLE *ftHandle*,LPDWORD *lpdwErrors,* LPFTCOMSTAT *lpftComstat*)

Parameters

ftHandle

Handle of the device.

lpdwErrors

Variable that contains the error mask.

lpftComstat

Pointer to COMSTAT structure.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how to use this function.

```
static COMSTAT oldCS = {0};
static DWORD dwOldErrors = 0;

FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
COMSTAT newCS;
DWORD dwErrors;
BOOL bChanged = FALSE;

if (!FT_W32_ClearCommError(ftHandle, &dwErrors, (FTCOMSTAT *)&newCS))
      ; // FT_W32_ClearCommError failed


if (dwErrors != dwOldErrors) {
      bChanged = TRUE;
   dwErrorsOld = dwErrors;
}

if (memcmp(&oldCS, &newCS, sizeof(FTCOMSTAT))) {
      bChanged = TRUE;
      oldCS = newCS;
}

if (bChanged) {

      if (dwErrors & CE_BREAK)
            ; // BREAK condition detected
      if (dwErrors & CE_FRAME)
            ; // Framing error detected
      if (dwErrors & CE_RXOVER)
            ; // Receive buffer has overflowed
      if (dwErrors & CE_TXFULL)
            ; // Transmit buffer full
      if (dwErrors & CE_OVERRUN)
            ; // Character buffer overrun
      if (dwErrors & CE_RXPARITY)
            ; // Parity error detected

      if (newCS.fCtsHold)
            ; // Transmitter waiting for CTS
      if (newCS.fDsrHold)
            ; // Transmitter is waiting for DSR
      if (newCS.fRlsdHold)
            ; // Transmitter is waiting for RLSD
      if (newCS.fXoffHold)
            ; // Transmitter is waiting because XOFF was received
      if (newCS.fXoffSent)
            ; //
      if (newCS.fEof)
            ; // End of file character has been received
      if (newCS.fTxim)
            ; // Tx immediate character queued for transmission

      // newCS.cbInQue contains number of bytes in receive queue
      // newCS.cbOutQue contains number of bytes in transmit queue

}
```

FT_W32_EscapeCommFunction

Perform an extended function.

BOOL **FT_W32_EscapeCommFunction**(FT_HANDLE *ftHandle*,DWORD *dwFunc*)

Parameters

ftHandle

Handle of the device.

dwFunc

The extended function to perform can be one of the following values.

CLRDTR	Clear the DTR signal
CLRRTS	Clear the RTS signal
SETDTR	Set the DTR signal
SETRTS	Set the RTS signal
SETBREAK	Set the BREAK condition
CLRBREAK	Clear the BREAK condition

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how to use this function.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile

FT_W32_EscapeCommFunction(ftHandle,CLRDTS);
FT_W32_EscapeCommFunction(ftHandle,SETRTS);
```


FT_W32_GetCommModemStatus

This function gets the current modem control value.

BOOL **FT_W32_GetCommModemStatus**(FT_HANDLE *ftHandle*,LPDWORD *lpdwStat*)

Parameters

ftHandle

Handle of the device.

lpdwStat

Pointer to a variable to contain modem control value. The modem control value can be a combination of the following.

MS_CTS_ON	Clear to Send (CTS) is on
MS_DSR_ON	Data Set Ready (DSR) is on
MS_RING_ON	Ring Indicator (RI) is on
MS_RLSD_ON	Receive Line Signal Detect (RLSD) is on

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Example

This example shows how to use this function.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
DWORD dwStatus;

if (FT_W32_GetCommModemStatus(ftHandle,&dwStatus)) {
      // FT_W32_GetCommModemStatus ok
      if (dwStatus & MS_CTS_ON)
            ; // CTS is on
      if (dwStatus & MS_DSR_ON)
            ; // DSR is on
      if (dwStatus & MS_RI_ON)
            ; // RI is on
      if (dwStatus & MS_RLSD_ON)
            ; // RLSD is on
}
else
      ; // FT_W32_GetCommModemStatus failed
```

FT_W32_GetCommState

This function gets the current device state.

BOOL **FT_W32_GetCommState**(FT_HANDLE *ftHandle*, LPFTDCB *lpftDcb*)

Parameters

ftHandle

Handle of the device.

lpftDcb

Pointer to a device control block.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Remarks

The current state of the device is returned in a device control block.

Example

This example shows how to use this function.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
FTDCB ftDCB;

if (FT_W32_GetCommState(ftHandle,&ftDCB))
      ; // FT_W32_GetCommState ok, device state is in ftDCB
else
      ; // FT_W32_GetCommState failed
```


FT_W32_GetCommTimeouts

This function gets the current read and write request timeout parameters for the specified device.

BOOL **FT_W32_GetCommTimeouts**(FT_HANDLE *ftHandle*,LPFTTIMEOUTS *lpftTimeouts*)

Parameters

ftHandle

Handle of the device.

lpftTimeouts

Pointer to a COMMTIMEOUTS structure to store timeout information.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Remarks

For an explanation of how timeouts are used, see **FT_W32_SetCommTimeouts**.

Example

This example shows how to retrieve the current timeout values.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
FTTIMEOUTS ftTS;

if (FT_W32_GetCommTimeouts(ftHandle,&ftTS))
      ; // FT_W32_GetCommTimeouts OK
else
      ; // FT_W32_GetCommTimeouts failed
```


FT_W32_PurgeComm

This function purges the device.

BOOL **FT_W32_PurgeComm**(FT_HANDLE *ftHandle*,DWORD *dwFlags*)

Parameters

ftHandle

Handle of the device.

dwFlags

Specifies the action to take. The action can be a combination of the following.

PURGE_TXABORT Terminate outstanding overlapped writes.
PURGE_RXABORT Terminate outstanding overlapped reads.
PURGE_TXCLEAR Clear the transmit buffer.
PURGE_RXCLEAR Clear the receive buffer.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Example

This example shows how to purge the receive and transmit queues.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile

if (FT_W32_PurgeComm(ftHandle,PURGE_TXCLEAR|PURGE_RXCLEAR))
      ; // FT_W32_PurgeComm OK
else
      ; // FT_W32_PurgeComm failed
```

FT_W32_SetCommBreak

Puts the communications line in the BREAK state.

BOOL **FT_W32_SetCommBreak**(FT_HANDLE *ftHandle*)

Parameters

ftHandle

Handle of the device.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how put the line in the BREAK state.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile

if (!FT_W32_SetCommBreak(ftHandle))
      ; // FT_W32_SetCommBreak failed
else
      ; // FT_W32_SetCommBreak OK
```


FT_W32_SetCommMask

This function specifies events that the device has to monitor.

BOOL **FT_W32_SetCommMask**(FT_HANDLE *ftHandle*,DWORD *dwMask*)

Parameters

ftHandle

Handle of the device.

dwMask

Mask containing events that the device has to monitor. This can be a combination of the following.

EV_BREAK	BREAK condition detected
EV_CTS	Change in Clear to Send (CTS)
EV_DSR	Change in Data Set Ready (DSR)
EV_ERR	Error in line status
EV_RING	Ring Indicator (RI) detected
EV_RLSD	Change in Receive Line Signal Detect (RLSD)
EV_RXCHAR	Character received
EV_RXFLAG	Event character received
EV_TXEMPTY	Transmitter empty

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Remarks

This function specifies the events that the device should monitor. An application can call the function **FT_W32_WaitCommEvent** to wait for an event to occur.

Example

This example shows how to monitor changes in the modem status lines DSR and CTS.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
DWORD dwMask = EV_CTS | EV_DSR;

if (!FT_W32_SetCommMAsk(ftHandle,dwMask))
      ; // FT_W32_SetCommMask failed
else
      ; // FT_W32_SetCommMask OK
```

FT_W32_SetCommState

This function sets the state of the device according to the contents of a device control block (DCB).

BOOL **FT_W32_SetCommState**(FT_HANDLE *ftHandle*, LPFTDCB *lpftDcb*)

Parameters

ftHandle

Handle of the device.

lpftDcb

Pointer to a device control block.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Example

This example shows how to use this function to change the baud rate.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
FTDCB ftDCB;

if (FT_W32_GetCommState(ftHandle,&ftDCB)) {
      // FT_W32_GetCommState ok, device state is in ftDCB
      ftDCB.BaudRate = 921600;
      if (FT_W32_SetCommState(ftHandle,&ftDCB))
              ; // FT_W32_SetCommState ok
      else
              ; // FT_W32_SetCommState failed
}
else
      ; // FT_W32_GetCommState failed
```

FT_W32_SetCommTimeouts

This function sets the timeout parameters for I/O requests.

BOOL **FT_W32_SetCommTimeouts**(FT_HANDLE *ftHandle*,LPFTTIMEOUTS *lpftTimeouts*)

Parameters

ftHandle

Handle of the device.

lpftTimeouts

Pointer to a COMMTIMEOUTS structure that contains timeout information.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Remarks

Timeouts are calculated using the information in the FTTIMEOUTS structure.

For read requests, the number of bytes to be read is multiplied by the total timeout multiplier, and added to the total timeout constant. So, if TS is an FTTIMEOUTS structure and the number of bytes to read is dwToRead, the read timeout, rdTO, is calculated as follows.

rdTO = (dwToRead * TS.ReadTotalTimeoutMultiplier) + TS.ReadTotalTimeoutConstant

For write requests, the number of bytes to be written is multiplied by the total timeout multiplier, and added to the total timeout constant. So, if TS is an FTTIMEOUTS structure and the number of bytes to write is dwToWrite, the write timeout, wrTO, is calculated as follows.

wrTO = (dwToWrite * TS.WriteTotalTimeoutMultiplier) + TS.WriteTotalTimeoutConstant

Example

This example shows how to setup a read timeout of 100 milliseconds and a write timeout of 200 milliseconds.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile
FTTIMEOUTS ftTS;

ftTS.ReadIntervalTimeout = 0;
ftTS.ReadTotalTimeoutMultiplier = 0;
ftTS.ReadTotalTimeoutConstant = 100;
ftTS.WriteTotalTimeoutMultiplier = 0;
ftTS.WriteTotalTimeoutConstant = 200;

if (FT_W32_SetCommTimeouts(ftHandle,&ftTS))
      ; // FT_W32_SetCommTimeouts OK
else
      ; // FT_W32_SetCommTimeouts failed
```

FT_W32_SetupComm

This function sets the read and write buffers.

BOOL **FT_W32_SetupComm**(FT_HANDLE *ftHandle*,DWORD *dwReadBufferSize*, DWORD *dwWriteBufferSize*)

Parameters

ftHandle

Handle of the device.

dwReadBufferSize

Length, in bytes, of the read buffer.

dwWriteBufferSize

Length, in bytes, of the write buffer.

Return Value

If the function is successful, the return value is nonzero.

If the function is unsuccessful, the return value is zero.

Remarks

This function has no effect. It is the responsibility of the driver to allocate sufficient storage for I/O requests.

FT_W32_WaitCommEvent

This function waits for an event to occur.

BOOL **FT_W32_WaitCommEvent**(FT_HANDLE *ftHandle*,LPDWORD *lpdwEvent*, LPOVERLAPPED *lpOverlapped*)

Parameters

ftHandle

Handle of the device.

lpdwEvent

Pointer to a location that receives a mask that contains the events that occurred.

lpOverlapped

Pointer to an OVERLAPPED structure.

Return Value

If the function is successful, the return value is nonzero.
If the function is unsuccessful, the return value is zero.

Remarks

This function supports both non-overlapped and overlapped I/O.

Non-overlapped I/O

The parameter, *lpOverlapped,* must be NULL for non-overlapped I/O.

This function does not return until an event that has been specified in a call to **FT_W32_SetCommMask** has occurred. The events that occurred and resulted in this function returning are stored in *lpdwEvent*.

Overlapped I/O

When the device has been opened for overlapped I/O, an application can issue a request and perform some additional work while the request is pending. This contrasts with the case of non-overlapped I/O in which the application issues a request and receives control again only after the request has been completed.

The parameter, *lpOverlapped,* must point to an initialized OVERLAPPED structure.

This function does not return until an event that has been specified in a call to **FT_W32_SetCommMask** has occurred.

If an event has already occurred, the request completes immediately, and the return code is non-zero. The events that occurred are stored in *lpdwEvent*.

If an event has not yet occurred, the request completes immediately, and the return code is zero, signifying an error. An application should call **FT_W32_GetLastError** to get the cause of the error. If the error code is ERROR_IO_PENDING, the overlapped operation is still in progress, and the application can perform other processing. Eventually, the application checks the result of the overlapped request by calling **FT_W32_GetOverlappedResult**. The events that occurred and resulted in this function returning are stored in *lpdwEvent*.

Example

This example shows how to write 128 bytes to the device using non-overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for non-overlapped i/o
DWORD dwEvents;

if (FT_W32_WaitCommEvent(ftHandle, &dwEvents, NULL))
      ; // FT_W32_WaitCommEvents OK
else
      ; // FT_W32_WaitCommEvents failed
```

This example shows how to write 128 bytes to the device using overlapped I/O.

```
FT_HANDLE ftHandle;  // setup by FT_W32_CreateFile for overlapped i/o
DWORD dwEvents;
DWORD dwRes;
OVERLAPPED osWait = { 0 };

if (!FT_W32_WaitCommEvent(ftHandle, &dwEvents, &osWait)) {
      if (FT_W32_GetLastError(ftHandle == ERROR_IO_PENDING) {
            // wait is delayed so do some other stuff until ...
          if (!FT_W32_GetOverlappedResult(ftHandle, &osWait, &dwRes, FALSE))
                  ; // error
              else
                  ; // FT_W32_WaitCommEvent OK
                    // Events that occurred are stored in dwEvents
      }
}
else {
      // FT_W32_WaitCommEvent OK
      // Events that occurred are stored in dwEvents
}
```

Appendix

Type Definitions

Excerpts from the header file FTD2XX.H are included in this Appendix to explain any references in the descriptions of the functions in this document.

For Visual C++ applications, these values are pre-declared in the header file, FTD2XX.H, which is included in the driver release. For other languages, these definitions will have to be converted to use equivalent types, and may have to be defined in an include file or within the body of the code. For non-VC++ applications check the application code examples on the FTDI web site as a translation of these may already exist.

UCHAR unsigned char (1 byte)
PUCHAR Pointer to unsigned char (4 bytes)
PCHAR Pointer to char (4 bytes)
DWORD unsigned long (4 bytes)
FT_HANDLE DWORD

FT_STATUS (DWORD)
FT_OK = 0
FT_INVALID_HANDLE = 1
FT_DEVICE_NOT_FOUND = 2
FT_DEVICE_NOT_OPENED = 3
FT_IO_ERROR = 4
FT_INSUFFICIENT_RESOURCES = 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_OTHER_ERROR = 17

Flags (see FT_OpenEx)
FT_OPEN_BY_SERIAL_NUMBER = 1
FT_OPEN_BY_DESCRIPTION = 2

Flags (see FT_ListDevices)
FT_LIST_NUMBER_ONLY = 0x80000000
FT_LIST_BY_INDEX = 0x40000000
FT_LIST_ALL = 0x20000000

FT_DEVICE DWORD
FT_DEVICE_232BM = 0
FT_DEVICE_232AM = 1
FT_DEVICE_100AX = 2
FT_DEVICE_UNKNOWN = 3

Word Length (see FT_SetDataCharacteristics)
FT_BITS_8 = 8
FT_BITS_7 = 7

Stop Bits (see FT_SetDataCharacteristics)
FT_STOP_BITS_1 = 0
FT_STOP_BITS_2 = 2

Parity (see FT_SetDataCharacteristics)
FT_PARITY_NONE = 0
FT_PARITY_ODD = 1
FT_PARITY_EVEN = 2
FT_PARITY_MARK = 3
FT_PARITY_SPACE = 4

Flow Control (see FT_SetFlowControl)
FT_FLOW_NONE = 0x0000
FT_FLOW_RTS_CTS = 0x0100
FT_FLOW_DTR_DSR = 0x0200
FT_FLOW_XON_XOFF = 0x0400

Purge RX and TX buffers (see FT_Purge)
FT_PURGE_RX = 1
FT_PURGE_TX = 2

Notification Events (see FT_SetEventNotification)

FT_EVENT_RXCHAR = 1

FT_EVENT_MODEM_STATUS = 2

FT_PROGRAM_DATA (EEPROM Programming Interface)

```
typedef struct ft_program_data {
      WORD VendorId;        // 0x0403
      WORD ProductId;       // 0x6001
      char *Manufacturer;   // "FTDI"
      char *ManufacturerId; // "FT"
      char *Description;    // "USB HS Serial Converter"
      char *SerialNumber;   // "FT000001" if fixed, or NULL
      WORD MaxPower;        // 0 < MaxPower <= 500
      WORD PnP;             // 0 = disabled, 1 = enabled
      WORD SelfPowered;     // 0 = bus powered, 1 = self powered
      WORD RemoteWakeup;    // 0 = not capable, 1 = capable
      //
      // Rev4 extensions
      //
      bool Rev4;            // true if Rev4 chip, false otherwise
      bool IsoIn;           // true if in endpoint is isochronous
      bool IsoOut;          // true if out endpoint is isochronous
      bool PullDownEnable;  // true if pull down enabled
      bool SerNumEnable;    // true if serial number to be used
      bool USBVersionEnable;       // true if chip uses USBVersion
      WORD USBVersion;      // BCD (0x0200 => USB2)
} FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;
```

FTCOMSTAT (FT-Win32 Programming Interface)

```
typedef struct _FTCOMSTAT {
    DWORD fCtsHold : 1;
    DWORD fDsrHold : 1;
    DWORD fRlsdHold : 1;
    DWORD fXoffHold : 1;
    DWORD fXoffSent : 1;
    DWORD fEof : 1;
    DWORD fTxim : 1;
    DWORD fReserved : 25;
    DWORD cbInQue;
    DWORD cbOutQue;
} FTCOMSTAT, *LPFTCOMSTAT;
```

FTDCB (FT-Win32 Programming Interface)

```
typedef struct _FTDCB {
    DWORD DCBlength;                // sizeof(FTDCB)
    DWORD BaudRate;                 // Baudrate at which running
    DWORD fBinary: 1;               // Binary Mode (skip EOF check)
    DWORD fParity: 1;               // Enable parity checking
    DWORD fOutxCtsFlow:1;           // CTS handshaking on output
    DWORD fOutxDsrFlow:1;           // DSR handshaking on output
    DWORD fDtrControl:2;            // DTR Flow control
    DWORD fDsrSensitivity:1;        // DSR Sensitivity
    DWORD fTXContinueOnXoff: 1;     // Continue TX when Xoff sent
    DWORD fOutX: 1;                 // Enable output X-ON/X-OFF
    DWORD fInX: 1;                  // Enable input X-ON/X-OFF
    DWORD fErrorChar: 1;            // Enable Err Replacement
    DWORD fNull: 1;                 // Enable Null stripping
    DWORD fRtsControl:2;            // Rts Flow control
    DWORD fAbortOnError:1;          // Abort all reads and writes on Error
    DWORD fDummy2:17;               // Reserved
    WORD wReserved;                 // Not currently used
    WORD XonLim;                    // Transmit X-ON threshold
    WORD XoffLim;                   // Transmit X-OFF threshold
    BYTE ByteSize;                  // Number of bits/byte, 7-8
    BYTE Parity;                    // 0-4=None,Odd,Even,Mark,Space
    BYTE StopBits;                  // 0,2 = 1, 2
    char XonChar;                   // Tx and Rx X-ON character
    char XoffChar;                  // Tx and Rx X-OFF character
    char ErrorChar;                 // Error replacement char
    char EofChar;                   // End of Input character
    char EvtChar;                   // Received Event character
    WORD wReserved1;                // Fill
} FTDCB, *LPFTDCB;
```

FTTIMEOUTS (FT-Win32 Programming Interface)

```
typedef struct _FTTIMEOUTS {
    DWORD ReadIntervalTimeout;          // Maximum time between read chars
    DWORD ReadTotalTimeoutMultiplier;   // Multiplier of characters
    DWORD ReadTotalTimeoutConstant;     // Constant in milliseconds
    DWORD WriteTotalTimeoutMultiplier;  // Multiplier of characters
    DWORD WriteTotalTimeoutConstant;    // Constant in milliseconds
} FTTIMEOUTS, *LPFTTIMEOUTS;
```