# Table of Contents

## Part I  Welcome to the FTD2XX Programmer's Guide

## Part II  Classic Interface Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT_SetVIDPID</td>
<td>6</td>
</tr>
<tr>
<td>FT_GetVIDPID</td>
<td>7</td>
</tr>
<tr>
<td>FT_ListDevices</td>
<td>8</td>
</tr>
<tr>
<td>FT_Open</td>
<td>11</td>
</tr>
<tr>
<td>FT_OpenEx</td>
<td>12</td>
</tr>
<tr>
<td>FT_Close</td>
<td>14</td>
</tr>
<tr>
<td>FT_Read</td>
<td>15</td>
</tr>
<tr>
<td>FT_Write</td>
<td>17</td>
</tr>
<tr>
<td>FT_ResetDevice</td>
<td>18</td>
</tr>
<tr>
<td>FT_SetBaudRate</td>
<td>19</td>
</tr>
<tr>
<td>FT_SetDivisor</td>
<td>20</td>
</tr>
<tr>
<td>FT_SetDataCharacteristics</td>
<td>21</td>
</tr>
<tr>
<td>FT_SetFlowControl</td>
<td>22</td>
</tr>
<tr>
<td>FT_SetDtr</td>
<td>23</td>
</tr>
<tr>
<td>FTClrDtr</td>
<td>24</td>
</tr>
<tr>
<td>FT_SetRts</td>
<td>25</td>
</tr>
<tr>
<td>FTClrRts</td>
<td>26</td>
</tr>
<tr>
<td>FT_GetModemStatus</td>
<td>27</td>
</tr>
<tr>
<td>FT_SetChars</td>
<td>28</td>
</tr>
<tr>
<td>FT_Purge</td>
<td>29</td>
</tr>
<tr>
<td>FT_SetTimeouts</td>
<td>30</td>
</tr>
<tr>
<td>FT_GetQueueStatus</td>
<td>31</td>
</tr>
<tr>
<td>FT_SetBreakOn</td>
<td>32</td>
</tr>
<tr>
<td>FT_SetBreakOff</td>
<td>33</td>
</tr>
<tr>
<td>FT_GetStatus</td>
<td>34</td>
</tr>
<tr>
<td>FT_SetEventNotification</td>
<td>35</td>
</tr>
<tr>
<td>FT_IoCtl</td>
<td>38</td>
</tr>
<tr>
<td>FT_SetWaitMask</td>
<td>39</td>
</tr>
<tr>
<td>FT_WaitOnMask</td>
<td>40</td>
</tr>
<tr>
<td>FT_GetDeviceInfo</td>
<td>41</td>
</tr>
<tr>
<td>FT_SetResetPipeRetryCount</td>
<td>43</td>
</tr>
</tbody>
</table>
Part III  EEPROM Programming Interface Functions  55
  1  FT_ReadEE ................................................................. 56
  2  FT_WriteEE ................................................................. 57
  3  FT_EraseEE ................................................................. 58
  4  FT_EE_Read ................................................................. 59
  5  FT_EE_ReadEx ............................................................... 61
  6  FT_EE_Program ............................................................. 62
  7  FT_EE_ProgramEx .......................................................... 64
  8  FT_EE_UARed ................................................................. 65
  9  FT_EE_UAWrite ............................................................... 66
 10  FT_EE_UASize ............................................................... 67

Part IV  Extended API Functions  68
  1  FT_GetLatencyTimer ......................................................... 69
  2  FT_SetLatencyTimer ......................................................... 70
  3  FT_GetBitMode ............................................................... 71
  4  FT_SetBitMode ............................................................... 72
  5  FT_SetUSBParameters .................................................... 74

Part V  FT-Win32 API Functions  75
  1  FT_W32_CreateFile .......................................................... 76
  2  FT_W32_CloseHandle ....................................................... 78
  3  FT_W32_ReadFile ............................................................. 79
  4  FT_W32_WriteFile ............................................................ 82
  5  FT_W32_GetLastError ....................................................... 84
  6  FT_W32_GetOverlappedResult ............................................ 85
  7  FT_W32_ClearCommBreak ................................................ 86
  8  FT_W32_ClearCommError ................................................ 87
  9  FT_W32_EscapeCommFunction .......................................... 89
10 FT_W32_GetCommModemStatus ................................................................. 90
11 FT_W32_GetCommState ................................................................. 91
12 FT_W32_GetCommTimeouts ............................................................... 92
13 FT_W32_PurgeComm ................................................................. 93
14 FT_W32_SetCommBreak ............................................................... 94
15 FT_W32_SetCommMask ................................................................. 95
16 FT_W32_SetCommState ................................................................. 96
17 FT_W32_SetCommTimeouts ............................................................ 97
18 FT_W32_SetupComm ................................................................. 98
19 FT_W32_WaitCommEvent .............................................................. 99

Part VI Appendix 101
1 Type Definitions ................................................................. 102
2 FTD2XX.H ................................................................. 108

Index 124
Welcome to the FTD2XX Programmer's Guide

FTDIs "D2XX Direct Drivers" for Windows offer an alternative solution to our VCP drivers which allows application software to interface with FT232R USB UART, FT245R USB FIFO, FT2232C Dual USB UART/FIFO, FT232BM USB UART, FT245BM USB FIFO, FT8U232AM USB UART and FT8U245AM 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 Visual C++, C++ Builder, Delphi, VB etc.) to the WDM driver. An INF installation file, uninstaller program and D2XX programmers guide complete the package.

The document is divided into four parts:

- **Classic Interface Functions** which explains the original functions with some more recent additions
- **EEPROM Interface** which allows application software to read/program the various fields in the FT232R/FT245R internal EEPROM or external 93C46/93C56/93C66 EEPROM for other devices, including a user defined area which can be used for application specific purposes.
- **Extended API Functions** which allow control of the additional features available from our 2nd generation devices onwards.
- **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 that 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.

Main Differences Between Windows and Windows CE D2XX Drivers

- Location IDs are not supported by Windows CE
- FT_ResetPort and FT_CyclePort are not available
- Windows CE does not support overlapped IO for
# Classic Interface Functions

## Introduction

An FTD2XX device is an FT232R USB UART, FT245R USB FIFO, FT2232C Dual USB UART/FIFO, FT232BM USB UART, FT245BM USB FIFO, FT8U232AM USB UART or FT8U245AM USB FIFO interfacing to Windows application software using FTDIs 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.

## 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 FT232R devices, FT2232C devices used in UART mode, FT232BM and FT8U232AM 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 FT245R devices, FT2232C devices used in FIFO mode, FT245BM and FT8U245AM devices, these functions are redundant and can effectively be ignored.

## Reference

Type definitions of the functional parameters and return codes used in the D2XX classic programming interface are contained in the appendix.
2.1 FT_SetVIDPID

A Linux specific command to include your own VID and PID within the internal device list table.

FT_STATUS FT_SetVIDPID (DWORD dwVID, DWORD dwPID)

Parameters

- dwVID: Device VID.
- dwPID: Device PID.

Return Value

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

Remarks

The driver will support a limited set of VID and PID matched devices (VID 0x0403 with PIDs 0x6001, 0x6010, 0x6006 only). In order to use the driver with alternative VID and PIDs the FT_SetVIDPID function must be used prior to calling FT_ListDevices, FT_Open, FT_OpenEx or FT_CreateDeviceInfoList.
2.2 FT_GetVIDPID

A Linux specific command to retrieve the current VID and PID within the internal device list table.

FT_STATUS FT_GetVIDPID (DWORD * pdwVID, DWORD * pdwPID)

Parameters
pdwVID Pointer to DWORD that will contain the internal VID.
pdwPID Pointer to DWORD that will contain the internal PID.

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

Remarks
See FT_SetVIDPID.

© Future Technology Devices International Ltd. 2006
2.3 **FT_ListDevices**

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

**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. 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 **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 **FT_OPEN_BY_SERIAL_NUMBER** bit is set in **dwFlags**, the serial number string will be returned; if **FT_OPEN_BY_DESCRIPTION** bit is set in **dwFlags**, the product description string will be returned; if **FT_OPEN_BY_LOCATION** bit is set in **dwFlags**, the Location ID will be returned; if none 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** 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 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.

The location ID of a device is returned if **FT_LIST_BY_INDEX** and **FT_OPEN_BY_LOCATION** bits are set in **dwFlags**. In this case the parameter **pvArg1** is interpreted as the index of the device, and the parameter **pvArg2** is interpreted as a pointer to a variable of type long to contain the location
ID. Indexes are zero-based, and the error code \texttt{FT\_DEVICE\_NOT\_FOUND} is returned for an invalid index. \textbf{Please note that Windows CE and Linux do not support location IDs.}

The location IDs of all connected devices are returned if \texttt{FT\_LIST\_ALL} and \texttt{FT\_OPEN\_BY\_LOCATION} bits are set in \texttt{dwFlags}. In this case, the parameter \texttt{pvArg1} is interpreted as a pointer to an array of variables of type long to contain the location IDs, and the parameter \texttt{pvArg2} is interpreted as a pointer to a DWORD location to store the number of devices currently connected.

\section*{Examples}
The examples that follow use these variables.

\begin{verbatim}
FT\_STATUS ftStatus;
DWORD numDevs;

Get the number of devices currently connected
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
}

Get serial number of first device
DWORD devIndex = 0; // first device
char Buffer[64]; // more than enough room!
ftStatus = FT\_ListDevices((PVOID)devIndex,Buffer,FT\_LIST\_BY\_INDEX|FT\_OPEN\_BY\_SERIAL\_NUMBER);
if (ftStatus == FT\_OK) {
    // FT\_ListDevices OK, serial number is in Buffer
} else {
    // FT\_ListDevices failed
}

Note that indexes are zero-based. If more than one device is connected, incrementing \texttt{devIndex} will get the serial number of each connected device in turn.

Get device descriptions of all devices currently connected
char *BufPtrs[3]; // pointer to array of 3 pointers
char Buffer1[64]; // buffer for description of first device
char Buffer2[64]; // buffer for description of second device

// 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 (ftStatus == FT\_OK) {
    // FT\_ListDevices OK, product descriptions are in Buffer1 and Buffer2, and numDevs contains the number of devices connected
} else {
    // FT\_ListDevices failed
}
\end{verbatim}
Note that this example assumes that two devices are connected. If more devices are connected, then the size of the array of pointers must be increased.

Get locations of all devices currently connected

```c
long locIdBuf[16];
ftStatus = FT_ListDevices(locIdBuf,&numDevs,FT_LIST_ALL|FT_OPEN_BY_LOCATION);
if (ftStatus == FT_OK) {
    // FT_ListDevices OK, location IDs are in locIdBuf, and
    // numDevs contains the number of devices connected
}
else {
    // FT_ListDevices failed
}
```

Note that this example assumes that no more than 16 devices are connected. If more devices are connected, then the size of the array of pointers must be increased.
2.4 FT_Open

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

\[ \text{FT\_STATUS FT\_Open (int } iDevice, \text{ 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.

```c
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
}
```
2.5 **FT_OpenEx**

Open the specified device and return a handle that will be used for subsequent accesses. The device can be specified by its serial number, device description or location.

This function can also be used to open multiple devices simultaneously. Multiple devices can be opened at the same time if they can be distinguished by serial number or device description. Alternatively, multiple devices can be opened at the same time using location IDs - location information derived from their physical locations on USB. Location IDs can be obtained using the utility USBView and are given in hexadecimal format.

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

**Parameters**

<table>
<thead>
<tr>
<th>pvArg1</th>
<th>Meaning depends on dwFlags, but it will normally be interpreted as a pointer to a null terminated string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwFlags</td>
<td>FT_OPEN_BY_SERIAL_NUMBER, FT_OPEN_BY_DESCRIPTION or FT_OPEN_BY_LOCATION.</td>
</tr>
<tr>
<td>ftHandle</td>
<td>Pointer to a variable of type FT_HANDLE where the handle will be stored. This handle must be used to access the device.</td>
</tr>
</tbody>
</table>

**Return Value**

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

**Remarks**

The meaning of pvArg1 depends on dwFlags: if dwFlags is FT_OPEN_BY_SERIAL_NUMBER, pvArg1 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, pvArg1 is interpreted as a pointer to a null-terminated string that represents the device description; if dwFlags is FT_OPEN_BY_LOCATION, pvArg1 is interpreted as a long value that contains the location ID of the device. **Please note that Windows CE and Linux do not support location IDs.**

ftHandle is a pointer to a variable of type FT_HANDLE where the handle is to be stored. This handle must be used to access the device.

**Examples**

The examples that follow use these variables.

```c
FT_STATUS ftStatus;
FT_STATUS ftStatus2;
FT_HANDLE ftHandle1;
FT_HANDLE ftHandle2;
long dwLoc;
```

**Open a device with serial number "FT000001"**

```c
ftStatus = FT_OpenEx("FT000001", FT_OPEN_BY_SERIAL_NUMBER, &ftHandle1);
```
if (ftStatus == FT_OK) {
    // success - device with serial number "FT000001" is open
} else {
    // failure
}

Open a device with device description "USB Serial Converter"

ftStatus = FT_OpenEx("USB Serial Converter", FT_OPEN_BY_DESCRIPTION, &ftHandle1);
if (ftStatus == FT_OK) {
    // success - device with device description "USB Serial Converter" is open
} else {
    // failure
}

Open 2 devices with serial numbers "FT000001" and "FT999999"

ftStatus = FT_OpenEx("FT000001", FT_OPEN_BY_SERIAL_NUMBER, &ftHandle1);
ftStatus2 = FT_OpenEx("FT999999", FT_OPEN_BY_SERIAL_NUMBER, &ftHandle2);
if (ftStatus == FT_OK && ftStatus2 == FT_OK) {
    // success - both devices are open
} else {
    // failure - one or both of the devices has not been opened
}

Open 2 devices with descriptions "USB Serial Converter" and "USB Pump Controller"

ftStatus = FT_OpenEx("USB Serial Converter", FT_OPEN_BY_DESCRIPTION, &ftHandle1);
ftStatus2 = FT_OpenEx("USB Pump Controller", FT_OPEN_BY_DESCRIPTION, &ftHandle2);
if (ftStatus == FT_OK && ftStatus2 == FT_OK) {
    // success - both devices are open
} else {
    // failure - one or both of the devices has not been opened
}

Open a device at location 23

dwLoc = 0x23;
ftStatus = FT_OpenEx(dwLoc, FT_OPEN_BY_LOCATION, &ftHandle1);
if (ftStatus == FT_OK) {
    // success - device at location 23 is open
} else {
    // failure
}

Open 2 devices at locations 23 and 31

dwLoc = 0x23;
ftStatus = FT_OpenEx(dwLoc, FT_OPEN_BY_LOCATION, &ftHandle1);
dwLoc = 0x31;
ftStatus2 = FT_OpenEx(dwLoc, FT_OPEN_BY_LOCATION, &ftHandle2);
if (ftStatus == FT_OK && ftStatus2 == FT_OK) {
    // success - both devices are open
} else {
    // failure - one or both of the devices has not been opened
}
2.6  FT_Close

Close an open device.

FT_STATUS FT_Close (FT_HANDLE ftHandle)

Parameters
ftHandle
Handle of the device.

Return Value
FT_OK if successful, otherwise the return value is an FT error code.
2.7 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.
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 than 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;
FT_STATUS ftStatus;
DWORD EventDWord;
DWORD TxBytes;
DWORD BytesReceived;
char RxBuffer[256];
ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

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
    }
}

FT_Close(ftHandle);

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

FT_HANDLE ftHandle;
FT_STATUS ftStatus;
DWORD RxBytes = 10;
DWORD BytesReceived;
char RxBuffer[256];

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

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_Close(ftHandle);
2.8 **FT_Write**

Write data to the device.

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

**Parameters**

- *ftHandle*  
  Handle of the device.

- *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.
2.9  **FT_ResetDevice**

This function sends a reset command to the device.

\[
\text{FT_STATUS FT_ResetDevice (FT_HANDLE ftHandle)}
\]

**Parameters**

- \(ftHandle\) Handle of the device.

**Return Value**

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

This function sets the Baud rate for the device.

```c
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.
2.11 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" is available from the Application Notes section of the FTDI website describes how to calculate the divisor for a non-standard Baud rate.
2.12 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.
2.13  **FT_SetFlowControl**

This function sets the flow control for the device.

\[
\text{FT\_STATUS } \text{FT\_SetFlowControl} \ (\text{FT\_HANDLE ftHandle, USHORT usFlowControl, UCHAR uXon, UCHAR uXoff})
\]

**Parameters**
- `ftHandle`  Handle of the device.
- `usFlowControl`  Must be one of `FT\_FLOW\_NONE`, `FT\_FLOW\_RTS\_CTS`, `FT\_FLOW\_DTR\_DSR` or `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.
2.14 **FT_SetDtr**

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

```c
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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}
ftStatus = FT_SetDtr(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetDtr OK
} else {
    // FT_SetDtr failed
}

FT_Close(ftHandle);
```
2.15  FT_ClrDtr

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

\[
\text{FT\_STATUS FT\_ClrDtr (FT\_HANDLE ftHandle)}
\]

**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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_ClrDtr(ftHandle);
if (ftStatus == FT_OK) {
    // FT_ClrDtr OK
} else {
    // FT_ClrDtr failed
}

FT_Close(ftHandle);
```
2.16 **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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) { // FT_Open failed
    return;
}

ftStatus = FT_SetRts(ftHandle);
if (ftStatus == FT_OK) { // FT_SetRts OK
} else { // FT_SetRts failed
}

FT_Close(ftHandle);
```
2.17 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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_ClrRts(ftHandle);
if (ftStatus == FT_OK) {
    // FT_ClrRts OK
} else {
    // FT_ClrRts failed
}

FT_Close(ftHandle);
```
2.18 **FT_GetModemStatus**

Gets the modem status from the device.

```c
FT_STATUS FT_GetModemStatus (FT_HANDLE ftHandle, LPDWORD lpdwModemStatus)
```

**Parameters**

- **ftHandle**
  - Handle of the device.

- **lpdwModemStatus**
  - Pointer to a variable of type DWORD which receives the modem status from the device. Status lines are bit-mapped as follows:
    - CTS = 0x10
    - DSR = 0x20
    - RI = 0x40
    - DCD = 0x80

**Return Value**

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

This function sets the special characters for the device.

\[
\text{FT\_STATUS FT\_SetChars (FT\_HANDLE ftHandle, UCHAR uEventCh, UCHAR uEventChEn, UCHAR uErrorCh, UCHAR uErrorChEn)}
\]

Parameters
- **ftHandle**: Handle of the device.
- **uEventCh**: Event character.
- **uEventChEn**: 0 if event character disabled, non-zero otherwise.
- **uErrorCh**: Error character.
- **uErrorChEn**: 0 if error character disabled, non-zero otherwise.

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

This function purges receive and transmit buffers in the device.

```c
FT_STATUS FT_Purge (FT_HANDLE ftHandle, DWORD dwMask)
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ftHandle</code></td>
<td>Handle of the device.</td>
</tr>
<tr>
<td><code>dwMask</code></td>
<td>Any combination of FT_PURGE_RX and FT_PURGE_TX.</td>
</tr>
</tbody>
</table>

**Return Value**

FT_OK if successful, otherwise the return value is an FT error code.
2.21 **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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_SetTimeouts(ftHandle, 5000, 1000);
if (ftStatus == FT_OK) {
    // FT_SetTimeouts OK
}
else {
    // FT_SetTimeouts failed
}

FT_Close(ftHandle);
```
2.22 **FT_GetQueueStatus**

Gets the number of characters in the receive queue.

```c
FT_STATUS FT_GetQueueStatus (FT_HANDLE ftHandle, LPDWORD lpdwAmountInRxQueue)
```

**Parameters**

- `ftHandle` Handle of the device.
- `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.
2.23 **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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_SetBreakOn(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetBreakOn OK
} else {
    // FT_SetBreakOn failed
}

FT_Close(ftHandle);
```
2.24 **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.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_SetBreakOff(ftHandle);
if (ftStatus == FT_OK) {
    // FT_SetBreakOff OK
}
else {
    // FT_SetBreakOff failed
}

FT_Close(ftHandle);
```
2.25 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.
- 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.
2.26 **FT_SetEventNotification**

Sets conditions for event notification.

```c
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 the 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.

**Windows and Windows CE 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**.

```c
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.

```c
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
    }
}
```

**Linux 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`.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;
EVENT_HANDLE eh;
DWORD EventMask;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

pthread_mutex_init(&eh.eMutex, NULL);
pthread_cond_init(&eh.eCondVar, NULL);

EventMask = FT_EVENT_RXCHAR | FT_EVENT_MODEM_STATUS;
ftStatus = FT_SetEventNotification(ftHandle, EventMask, (PVOID)&eh);

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.

```c
pthread_mutex_lock(&eh.eMutex);
pthread_cond_wait(&eh.eCondVar, &eh.eMutex);
pthread_mutex_unlock(&eh.eMutex);

DWORD EventDWord;
DWORD RxBytes;
DWORD TxBytes;
DWORD Status;

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

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

FT_Close(ftHandle);
2.27 **FT_IoCtl**

Undocumented function.

```
FT_STATUS FT_IoCtl (FT_HANDLE ftHandle, DWORD dwIoControlCode, LPVOID lpInBuf,
                    DWORD nInBufSize, LPVOID lpOutBuf, DWORD nOutBufSize,
                    LPDWORD lpBytesReturned, LPOVERLAPPED lpOverlapped)
```
2.28 FT_SetWaitMask

Undocumented function.

FT_STATUS FT_SetWaitMask (FT_HANDLE ftHandle, DWORD dwMask)
2.29 FT_WaitOnMask

Undocumented function.

FT_STATUS FT_WaitOnMask (FT_HANDLE ftHandle, DWORD dwMask)
2.30 **FT_GetDeviceInfo**
Get device information.

```c
FT_STATUS FT_GetDeviceInfo(FT_HANDLE ftHandle, FT_DEVICE *pftType, LPDWORD lpdwID, PCHAR pcSerialNumber, PCHAR pcDescription, PVOID pvDummy)
```

**Parameters**
- `ftHandle` Handle of the device.
- `pftType` Pointer to unsigned long to store device type.
- `lpdwID` Pointer to unsigned long to store device ID.
- `pcSerialNumber` Pointer to buffer to store device serial number as a null-terminated string.
- `pcDescription` Pointer to buffer to store device description as a null-terminated string.
- `pvDummy` Reserved for future use - should be set to NULL.

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

**Remarks**
This function is used to return the device type, device ID, device description and serial number.

The device ID is encoded in a DWORD - the most significant word contains the vendor ID, and the least significant word contains the product ID. So the returned ID 0x04036001 corresponds to the device ID VID_0403&PID_6001.

**Example**
This example shows how to get information about a device.

```c
FT_HANDLE ftHandle;
FT_DEVICE ftDevice;
FT_STATUS ftStatus;
DWORD deviceID;
char SerialNumber[16];
char Description[64];

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_GetDeviceInfo(
    ftHandle,
    &ftDevice,
    &deviceID,
    SerialNumber,
    Description,
```
if (ftStatus == FT_OK) {
    if (ftDevice == FT_DEVICE_2232C)
        ; // device is FT2232C
    else if (ftDevice == FT_DEVICE_BM)
        ; // device is FTU232BM
    else if (ftDevice == FT_DEVICE_AM)
        ; // device is FT8U232AM
    else
        ; // unknown device (this should not happen!)
    // deviceID contains encoded device ID
    // SerialNumber, Description contain 0-terminated strings
}
else {
    ; // FT_GetDeviceType FAILED!
}
FT_Close(ftHandle);
2.31 FT_SetResetPipeRetryCount

Set the ResetPipeRetryCount.

FT_STATUS FT_SetResetPipeRetryCount (FT_HANDLE ftHandle, DWORD dwCount)

Parameters
ftHandle Handle of the device.
dwCount Unsigned long containing required ResetPipeRetryCount.

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

Remarks
This function is used to set the ResetPipeRetryCount. ResetPipeRetryCount controls the maximum number of times that the driver tries to reset a pipe on which an error has occurred. ResetPipeRequestRetryCount defaults to 50. It may be necessary to increase this value in noisy environments where a lot of USB errors occur.

Not available in Linux.

Example
This example shows how to set the ResetPipeRetryCount to 100.

FT_HANDLE ftHandle;  // valid handle returned from FT_OpenEx
FT_STATUS ftStatus;
DWORD dwRetryCount;

dwRetryCount = 100;
ftStatus = FT_SetResetPipeRetryCount(ftHandle,dwRetryCount);
if (ftStatus == FT_OK) {
   // ResetPipeRetryCount set to 100
} else {
   // FT_SetResetPipeRetryCount FAILED!
}
2.32 **FT_StopInTask**

Stops the driver's IN task.

\[
\text{FT\_STATUS FT\_StopInTask (FT\_HANDLE ftHandle)}
\]

**Parameters**

- **ftHandle**
  Handle of the device.

**Return Value**

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

**Remarks**

This function is used to put the driver's IN task (read) into a wait state. It can be used in situations where data is being received continuously, so that the device can be purged without more data being received. It is used together with **FT\_RestartInTask**, which sets the IN task running again.

**Example**

This example shows how to use FT\_StopInTask.

```c
FT\_HANDLE ftHandle;
FT\_STATUS ftStatus;

ftStatus = FT\_Open(0, &ftHandle);
if(ftStatus != FT\_OK) {
    // FT\_Open failed
    return;
}

do {
    ftStatus = FT\_StopInTask(ftHandle);
} while (ftStatus != FT\_OK);

// Do something - for example purge device
//
do {
    ftStatus = FT\_RestartInTask(ftHandle);
} while (ftStatus != FT\_OK);

FT\_Close(ftHandle);
```
2.33 FT_RestartInTask

Restart the driver's IN task.

**FT_STATUS FT_RestartInTask (FT_HANDLE ftHandle)**

**Parameters**

*ftHandle* Handle of the device.

**Return Value**

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

**Remarks**

This function is used to restart the driver's IN task (read) after it has been stopped by a call to *FT_StopInTask*.

**Example**

This example shows how to use FT_RestartInTask.

```c
FT_HANDLE ftHandle;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if(ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

do {
    ftStatus = FT_StopInTask(ftHandle);
} while (ftStatus != FT_OK);

// // Do something - for example purge device
//
do {
    ftStatus = FT_RestartInTask(ftHandle);
} while (ftStatus != FT_OK);

FT_Close(ftHandle);
```
2.34 FT_ResetPort

Send a reset command to the port.

FT_STATUS FT_ResetPort (FT_HANDLE ftHandle)

Parameters

ftHandle Handle of the device.

Return Value

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

Remarks

This function is used to attempt to recover the port after a failure. It is not equivalent to an unplug-replug event.

Not available in Windows CE and Linux.

Example

This example shows how to reset the port.

FT_HANDLE ftHandle; // valid handle returned from FT_OpenEx
FT_STATUS ftStatus;

ftStatus = FT_ResetPort(ftHandle);
if (ftStatus == FT_OK) {
    // Port has been reset
} else {
    // FT_ResetPort FAILED!
}
2.35 **FT_CyclePort**

Send a cycle command to the USB port.

\[
\text{FT\_STATUS}\ \text{FT\_CyclePort}\ (\text{FT\_HANDLE}\ ftHandle)
\]

**Parameters**

- *ftHandle*  
  Handle of the device.

**Return Value**

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

**Remarks**

The effect of this function is the same as disconnecting then reconnecting the device from USB. Possible use of this function is in situations where a fatal error has occurred and it is difficult, or not possible, to recover without unplugging and replugging the USB cable. This function can also be used after re-programming the EEPROM to force the FTDI device to read the new EEPROM contents which previously required a physical disconnect-reconnect.

As the current session is not restored when the driver is reloaded, the application must be able to recover after calling this function.

*Not available in Windows 98, Windows CE and Linux.*

*For FT2232C devices, FT_CyclePort will only work under Windows XP.*

**Example**

This example shows how to cycle the port.

```c
FT_HANDLE ftHandle; // valid handle returned from FT_OpenEx
FT_STATUS ftStatus;

ftStatus = FT_CyclePort(ftHandle);
if (ftStatus == FT_OK) {
    // Port has been cycled.
} else {
    // FT_CyclePort FAILED!
}
```
2.36  **FT_CreateDeviceInfoList**

This function builds a device information list and returns the number of D2XX devices connected to the system. The list contains information about both unopen and open devices.

\[
\text{FT_STATUS FT_CreateDeviceInfoList (LPDWORD lpdwNumDevs)}
\]

**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.

**Remarks**

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`.

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

**Example**

This example shows how to call `FT_CreateDeviceInfoList`.

```c
FT_STATUS ftStatus;
DWORD numDevs;

// create the device information list
//
ftStatus = FT_CreateDeviceInfoList(&numDevs);
if (ftStatus == FT_OK) {
    printf("Number of devices is %d\n",numDevs);
} else {
    // FT_CreateDeviceInfoList failed
}
```
2.37 **FT_GetDeviceInfoList**

This function returns a device information list and the number of D2XX devices in the list.

**FT_STATUS** `FT_GetDeviceInfo` *(FT_DEVICE_LIST_INFO_NODE *pDest, LPDWORD lpdwNumDevs)*

**Parameters**
- *pDest* Pointer to an array of FT_DEVICE_LIST_INFO_NODE structures.
- lpdwNumDevs Pointer to the number of elements in the array.

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

**Remarks**
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.

Location ID information is not returned for devices that are open when `FT_CreateDeviceInfoList` is called.

The array of FT_DEVICE_LIST_INFO_NODES contains all available data on each device. The structure of FT_DEVICE_LIST_INFO_NODES is given in the Appendix. 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.

When programming in Visual Basic, LabVIEW or similar languages, `FT_GetDeviceInfoDetail` may be required instead of this function.

Please note that Windows CE and Linux do not support location IDs. As such, the Location ID parameter in the structure will be empty under Windows CE and Linux.

**Example**
This example shows how to call `FT_GetDeviceInfoList`.

```c
FT_STATUS ftStatus;
FT_DEVICE_LIST_INFO_NODE *devInfo;
DWORD numDevs;

// create the device information list
//
ftStatus = FT_CreateDeviceInfoList(&numDevs);
if (ftStatus == FT_OK) {
    printf("Number of devices is %d\n",numDevs);
}

//
```
// allocate storage for list based on numDevs
//
devInfo = (FT DEVICE LIST INFO NODE*)malloc(sizeof(FT DEVICE LIST INFO NODE)*numDevs);

// get the device information list
//
ftStatus = FT GetDeviceInfoList(devInfo,&numDevs);
if (ftStatus == FT_OK) {
    for (int i = 0; i < numDevs; i++) {
        printf("Dev %d:\n",i);
        printf("  Flags=0x%x\n",devInfo[i].Flags);
        printf("  Type=0x%x\n",devInfo[i].Type);
        printf("  ID=0x%x\n",devInfo[i].ID);
        printf("  LocId=0x%x\n",devInfo[i].LocId);
        printf("  SerialNumber=%s\n",devInfo[i].SerialNumber);
        printf("  Description=%s\n",devInfo[i].Description);
        printf("  ftHandle=0x%x\n",devInfo[i].ftHandle);
    }
}
2.38  **FT_GetDeviceInfoDetail**

This function returns an entry from the device information list.

```c
FT_STATUS FT_GetDeviceInfoDetail (DWORD dwIndex, LPDWORD lpdwFlags, LPDWORD lpdwType, LPDWORD lpdwID, LPDWORD lpdwLocId, PCHAR pcSerialNumber, PCHAR pcDescription, FT_HANDLE *ftHandle)
```

**Parameters**

- **dwIndex**: Index of the entry in the device info list.
- **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.
- **pcSerialNumber**: Pointer to buffer to store device serial number as a null-terminated string.
- **pcDescription**: Pointer to buffer to store device description as a null-terminated string.
- ***ftHandle**: 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.

**Remarks**

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.

The index value is zero-based.

The flag value is a 4-byte bit map containing miscellaneous data. Bit 0 (least significant bit) of this number indicates if the port is open (1) or closed (0). The remaining bits (1 - 31) are reserved at this time.

Location ID information is not returned for devices that are open when **FT_CreateDeviceInfoList** is called.

To return the whole device info list as an array of FT_DEVICE_LIST_INFO_NODE structures, use **FT_GetDeviceInfoList**.

**Please note that Windows CE and Linux do not support location IDs.** As such, the Location ID parameter in the structure will be empty under Windows CE and Linux.

**Example**
This example shows how to call FT_GetDeviceInfoDetail.

```
FT_STATUS ftStatus;
FT_HANDLE ftHandleTemp;
DWORD numDevs;
DWORD Flags;
DWORD ID;
DWORD Type;
DWORD LocId;
char SerialNumber[16];
char Description[64];

// create the device information list
//
ftStatus = FT_CreateDeviceInfoList(&numDevs);
if (ftStatus == FT_OK) {
    printf("Number of devices is %d\n",numDevs);
}

// get information for device 0
//
ftStatus = FT_GetDeviceInfoDetail(0, &Flags, &Type, &ID, &LocId, SerialNumber,
    Description, &ftHandleTemp);
if (ftStatus == FT_OK) {
    printf("Dev 0:\n");
    printf(" Flags=0x%x\n",Flags);
    printf(" Type=0x%x\n",Type);
    printf(" ID=0x%x\n",ID);
    printf(" LocId=0x%x\n",LocId);
    printf(" SerialNumber=%s\n",SerialNumber);
    printf(" Description=%s\n",Description);
    printf(" ftHandle=0x%x\n",ftHandleTemp);
}
```
2.39 **FT_GetDriverVersion**

This function returns the D2XX driver version number.

**FT_STATUS FT_GetDriverVersion** (FT_HANDLE ftHandle, LPDWORD lpdwDriverVersion)

### Parameters
- *ftHandle*: Handle of the device.
- *lpdwDriverVersion*: Pointer to the driver version number.

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

### Remarks
A version number consists of major, minor and build version numbers contained in a 4-byte field (unsigned long). Byte0 (least significant) holds the build version, Byte1 holds the minor version, and Byte2 holds the major version. Byte3 is currently set to zero.

For example, driver version "3.01.02" is represented as 0x00030102. Note that a device has to be opened before this function can be called.

**Not available in Windows CE or Linux.**

### Example

This example shows how to call FT_GetDriverVersion.

```c
FT_HANDLE ftHandle;
FT_STATUS status;
DWORD dwDriverVer;

// Get driver version

status = FT_Open(0,&ftHandle);
if (status == FT_OK) {
    status = FT_GetDriverVersion(ftHandle,&dwDriverVer);
    if (status == FT_OK)
        printf("Driver version = 0x%x\n",dwDriverVer);
    else
        printf("error reading driver version\n");
    FT_Close(ftHandle);
}
```
2.40 FT_GetLibraryVersion

This function returns D2XX DLL version number.

FT_STATUS FT_GetLibraryVersion (LPDWORD lpdwDLLVersion)

**Parameters**

*lpdwDLLVersion* Pointer to the DLL version number.

**Return Value**

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

**Remarks**

A version number consists of major, minor and build version numbers contained in a 4-byte field (unsigned long). Byte0 (least significant) holds the build version, Byte1 holds the minor version, and Byte2 holds the major version. Byte3 is currently set to zero.

For example, driver version "3.01.02" is represented as 0x00030102. Note that this function does not take a handle, and so it can be called without opening a device.

**Not available in Windows CE or Linux.**

**Example**

This example shows how to call FT_GetLibraryVersion.

```c
FT_STATUS status;
DWORD dwLibraryVer;

//
// Get DLL version
//

status = FT_GetLibraryVersion(&dwLibraryVer);
if (status == FT_OK)
    printf("Library version = 0x%x\n", dwLibraryVer);
else
    printf("error reading library version\n");
```
3 EEPROM Programming Interface Functions

Introduction
FTDI has included EEPROM programming support in the D2XX library. This section describes that interface.

Overview
Functions are provided to program the EEPROM (FT_EE_Program, FT_EE_ProgramEx, FT_WriteEE), read the EEPROM (FT_EE_Read, FT_EE_ReadEx, FT_ReadEE) and erase the EEPROM (FT_EraseEE).

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

Reference
Type definitions of the functional parameters and return codes used in the D2XX EEPROM programming interface are contained in the appendix.
3.1 FT_ReadEE

Read a value from an EEPROM location.

FT_STATUS FT_ReadEE (FT_HANDLE ftHandle, DWORD dwWordOffset, LPWORD lpwValue)

Parameters
- ftHandle: Handle of the device.
- dwWordOffset: EEPROM location to read from.
- lpwValue: Pointer to the value read from the EEPROM.

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

Write a value to an EEPROM location.

\[
\text{FT_STATUS FT_WriteEE (FT_HANDLE ftHandle, DWORD dwWordOffset, WORD wValue)}
\]

**Parameters**
- **ftHandle**: Handle of the device.
- **dwWordOffset**: EEPROM location to write to.
- **wValue**: Value to write to EEPROM.

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

Erase the EEPROM.

```c
FT_STATUS FT_EraseEE (FT_HANDLE ftHandle)
```

**Parameters**

- `ftHandle` Handle of the device.

**Return Value**

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

**Remarks**

This function will erase the entire contents of an EEPROM, including the user area.
3.4 **FT_EE_Read**

Read the contents of the EEPROM.

**FT_STATUS** `FT_EE_Read` (FT_HANDLE `ftHandle`, PFT_PROGRAM_DATA `pData`)

**Parameters**
- `ftHandle` Handle of the device.
- `pData` 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 `pData` 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.

Note that the DLL must be informed which version of the FT_PROGRAM_DATA structure is being used. This is done through the Signature1, Signature2 and Version elements of the structure. Signature1 should always be 0x00000000, Signature2 should always be 0xFFFFFFFF and Version can be set to use whichever version is required. For compatibility with all current devices Version should be set to the latest version of the FT_PROGRAM_DATA structure which is defined in FTD2XX.h.

**Example**

```c
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.Signature1 = 0x00000000;
ftData.Signature2 = 0xffffffff;
ftData.Version = 0x00000002;  // EEPROM structure with FT232R extensions
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!
}
3.5 **FT_EE_ReadEx**

Read the contents of the EEPROM and pass strings separately.

```c
FT_STATUS FT_EE_ReadEx (FT_HANDLE ftHandle, PFT_PROGRAM_DATA pData, char *Manufacturer, char *ManufacturerId, char *Description, char *SerialNumber)
```

**Parameters**

- **ftHandle** Handle of the device.
- **pData** Pointer to a structure of type FT_PROGRAM_DATA.
- ***Manufacturer** Pointer to a null-terminated string containing the manufacturer name.
- ***ManufacturerID** Pointer to a null-terminated string containing the manufacturer ID.
- ***Description** Pointer to a null-terminated string containing the device description.
- ***SerialNumber** Pointer to a null-terminated string containing the device serial number.

**Return Value**

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

**Remarks**

This variation of the standard **FT_EE_Read** function was included to provide support for languages such as LabVIEW where problems can occur when string pointers are contained in a structure.

This function interprets the parameter **pData** 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** structure 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.

Note that the DLL must be informed which version of the **FT_PROGRAM_DATA** structure is being used. This is done through the **Signature1**, **Signature2** and **Version** elements of the structure. **Signature1** should always be 0x00000000, **Signature2** should always be 0xFFFFFFFF and **Version** can be set to use whichever version is required. For compatibility with all current devices **Version** should be set to the latest version of the **FT_PROGRAM_DATA** structure which is defined in **FTD2XX.h**.

The string parameters in the **FT_PROGRAM_DATA** structure should be passed as DWORDs to avoid overlapping of parameters. All string pointers are passed out separately from the **FT_PROGRAM_DATA** structure.
3.6 FT_EE_Program
Program the EEPROM.

FT_STATUS FT_EE_Program (FT_HANDLE ftHandle, PFT_PROGRAM_DATA pData)

Parameters
ftHandle Handle of the device.
pData 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 pData 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.

Note that the DLL must be informed which version of the FT_PROGRAM_DATA structure is being used. This is done through the Signature1, Signature2 and Version elements of the structure. Signature1 should always be 0x00000000, Signature2 should always be 0xFFFFFFFF and Version can be set to use whichever version is required. For compatibility with all current devices Version should be set to the latest version of the FT_PROGRAM_DATA structure which is defined in FTD2XX.h.

If pData is NULL, the structure version will default to 0 (original BM series) and 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, FALSE, 0}

Example
// Version 2 structure for programming a BM device.
// Other elements would need non-zero values for FT2232C, FT232R or FT245R devices.
FT_PROGRAM_DATA ftData = {
    0x00000000, // Header - must be 0x00000000
    0xFFFFFFFF, // Header - must be 0xffffffff
    0x00000002, // Header - FT_PROGRAM_DATA version
    0x0403, // VID
    0x6001, // PID
    "FTDI", // Manufacturer
    "FT", // Manufacturer ID
    "USB HS Serial Converter", // Description
    "FT000001", // Serial Number
    44, // MaxPower
    1, // PnP
0, // SelfPowered
1, // RemoteWakeup
1, // non-zero if Rev4 chip, zero otherwise
0, // non-zero if in endpoint is isochronous
0, // non-zero if out endpoint is isochronous
0, // non-zero if pull down enabled
1, // non-zero if serial number to be used
0, // non-zero if chip uses USBVersion
0x0110 // BCD (0x0200 => USB2)
//
// FT2232C extensions (Enabled if Version = 1 or Version = 2)
//
0, // non-zero if Rev5 chip, zero otherwise
0, // non-zero if in endpoint is isochronous
0, // non-zero if in endpoint is isochronous
0, // non-zero if out endpoint is isochronous
0, // non-zero if pull down enabled
0, // non-zero if serial number to be used
0, // non-zero if chip uses USBVersion
0x0, // BCD (0x0200 => USB2)
0, // non-zero if interface is high current
0, // non-zero if interface is high current
0, // non-zero if interface is 245 FIFO
0, // non-zero if interface is 245 FIFO CPU target
0, // non-zero if interface is Fast serial
0, // non-zero if interface is to use VCP drivers
0, // non-zero if interface is 245 FIFO CPU target
0, // non-zero if interface is Fast serial
0, // non-zero if interface is to use VCP drivers
//
// FT232R extensions (Enabled if Version = 2)
//
0, // Use External Oscillator
0, // High Drive I/Os
0, // Endpoint size
0, // non-zero if pull down enabled
0, // non-zero if serial number to be used
0, // non-zero if invert TXD
0, // non-zero if invert RTS
0, // non-zero if invert CTS
0, // non-zero if invert DTR
0, // non-zero if invert DSR
0, // non-zero if invert DCD
0, // non-zero if invert RI
0, // Cbus Mux control
0, // Cbus Mux control
0, // Cbus Mux control
0, // Cbus Mux control
0, // non-zero if using D2XX drivers
};

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!
    }
}
3.7 **FT_EE_ProgramEx**

Program the EEPROM and pass strings separately.

**FT_STATUS FT_EE_ProgramEx** (FT_HANDLE ftHandle, PFT_PROGRAM_DATA pData, char *Manufacturer, char *ManufacturerId, char *Description, char *SerialNumber)

**Parameters**
- **ftHandle**
  Handle of the device.
- **pData**
  Pointer to a structure of type FT_PROGRAM_DATA.
- ***Manufacturer**
  Pointer to a null-terminated string containing the manufacturer name.
- ***ManufacturerID**
  Pointer to a null-terminated string containing the manufacturer ID.
- ***Description**
  Pointer to a null-terminated string containing the device description.
- ***SerialNumber**
  Pointer to a null-terminated string containing the device serial number.

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

**Remarks**
This variation of the **FT_EE_Program** function was included to provide support for languages such as LabVIEW where problems can occur when string pointers are contained in a structure.

This function interprets the parameter **pData** 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.

The string pointer parameters in the FT_PROGRAM_DATA structure should be allocated as DWORDs to avoid overlapping of parameters. The string parameters are then passed in separately.

If the SerialNumber field 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.

Note that the DLL must be informed which version of the FT_PROGRAM_DATA structure is being used. This is done through the **Signature1**, **Signature2** and **Version** elements of the structure. **Signature1** should always be 0x00000000, **Signature2** should always be 0xFFFFFFFF and **Version** can be set to use whichever version is required. For compatibility with all current devices **Version** should be set to the latest version of the FT_PROGRAM_DATA structure which is defined in **FTD2XX.h**.

If **pData** is NULL, the structure version will default to 0 (original BM series) and 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, FALSE, 0\}
3.8 FT_EE_UARead

Read the contents of the EEUA.

FT_STATUS FT_EE_UARead (FT_HANDLE ftHandle, UCHAR 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 the 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!
}
unsigned 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!
}
3.9 **FT_EE_UAWrite**

Write data into the EEUA.

FT_STATUS **FT_EE_UAWrite** (FT_HANDLE ftHandle, UCHAR 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**

```c
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, (unsigned char*)buffer, 12);
if(ftStatus != FT_OK) {
    // FT_EE_UAWRITE failed
} else {
    // FT_EE_UAWRITE failed
}
```
3.10 **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**

```c
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!
}
```
4 Extended API Functions

Introduction
FTDI's FT232R USB UART (4th generation), FT245R USB FIFO (4th generation), FT2232C Dual USB UART/FIFO (3rd generation), FT232BM USB UART (2nd generation) and FT245BM USB FIFO (2nd generation) 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.

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 modes and other FT2232C bit modes are controlled via the functions `FT_GetBitMode` and `FT_SetBitMode`. Before these functions can be accessed, the device must first be opened. The Win32API function, CreateFile, returns a handle that is used by all functions in the programming interface to identify the device.

Reference
Type definitions of the functional parameters and return codes used in the D2XX extended programming interface are contained in the appendix.
4.1 **FT_GetLatencyTimer**

Get the current value of the latency timer.

```c
FT_STATUS FT_GetLatencyTimer (FT_HANDLE ftHandle, UCHAR 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 all other FTDI devices, this timeout is programmable and can be set at 1 ms intervals between 2ms and 255 ms. This allows the device to be better optimized for protocols requiring faster response times from short data packets.

**Example**

```c
HANDLE ftHandle;
FT_STATUS ftStatus;
UCHAR LatencyTimer;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_GetLatencyTimer(ftHandle, &LatencyTimer);
if (ftStatus == FT_OK) {
    // LatencyTimer contains current value
} else {
    // FT_GetLatencyTimer FAILED!
}

FT_Close(ftHandle);
```
4.2 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 2 - 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 all other FTDI devices, this timeout is programmable and can be set at 1 ms intervals between 2 ms and 255 ms. This allows the device to be better optimized for protocols requiring faster response times from short data packets.

Example

```c
HANDLE ftHandle;
FT_STATUS ftStatus;
UCHAR LatencyTimer = 10;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_SetLatencyTimer(ftHandle, LatencyTimer);
if (ftStatus == FT_OK) {
    // LatencyTimer set to 10 milliseconds
} else {
    // FT_SetLatencyTimer FAILED!
}

FT_Close(ftHandle);
```
4.3 FT_GetBitMode

Gets the instantaneous value of the data bus.

FT_STATUS FT_GetBitMode (FT_HANDLE ftHandle, PUCHAR pucMode)

Parameters

ftHandle Handle of the device.
pucMode 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 available bit modes for the FT2232C, see the application note "Bit Mode Functions for the FT2232C".
For a description of Bit Bang Mode for the FT232BM and FT245BM, see the application note "FT232BM/FT245BM Bit Bang Mode".
These application notes are available for download from the Application Notes page in the Documents section of the FTDI website.

Example

HANDLE ftHandle;
UCHAR BitMode;
FT_STATUS ftStatus;

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_GetBitMode(ftHandle, &BitMode);
if (ftStatus == FT_OK) {
    // BitMode contains current value
}
else {
    // FT_GetBitMode FAILED!
}

FT_Close(ftHandle);
4.4 FT_SetBitMode

Set the value of the bit mode.

FT_STATUS FT_SetBitMode (FT_HANDLE ftHandle, UCHAR ucMask, UCHAR ucMode)

Parameters

ftHandle Handle of the device.
ucMask Required value for bit mode mask. This sets up which bits are inputs and outputs. A bit value of 0 sets the corresponding pin to an input, a bit value of 1 sets the corresponding pin to an output.
ucMode Mode value. Can be one of the following:
0x0 = Reset
0x1 = Asynchronous Bit Bang
0x2 = MPSSE (FT2232C devices only)
0x4 = Synchronous Bit Bang (FT232R, FT245R and FT2232C devices only)
0x8 = MCU Host Bus Emulation Mode (FT2232C devices only)
0x10 = Fast Opto-Isolated Serial Mode (FT2232C devices only)
0x20 = CBUS Bit Bang Mode (FT232R devices only)

Return Value

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

Remarks

For a description of Bit Bang Mode for the FT232BM and FT245BM, see the application note "FT232BM/FT245BM Bit Bang Mode".
For a description of available bit modes for the FT2232C, see the application note "Bit Mode Functions for the FT2232C".
For a description of the Bit Bang modes available for the FT232R and FT245R devices, see the application note "Bit Bang Modes for the FT232R and FT245R".
Application notes are available for download from the Application Notes page in the Documents section of the FTDI website.

Note that to use CBUS Bit Bang for the FT232R, the CBUS must be configured for CBUS Bit Bang in the EEPROM.

Example

HANDLE ftHandle;
FT_STATUS ftStatus;
UCHAR Mask = 0xff;
UCHAR Mode = 1;  // Set asynchronous bit-bang mode
ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}

ftStatus = FT_SetBitMode(ftHandle, Mask, Mode);
if (ftStatus == FT_OK) {
    // 0xff written to device
} else {
    // FT_SetBitMode FAILED!
}

FT_Close(ftHandle);
4.5 **FT_SetUSBParameters**

Set the USB request transfer size.

```c
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 requirements. Transfer sizes must be set to a multiple of 64 bytes between 64 bytes and 64k bytes.

When `FT_SetUSBParameters` is called, the change comes into effect immediately and any data that was held in the driver at the time of the change is lost.

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

**Example**
```
HANDLE ftHandle;
FT_STATUS ftStatus;
DWORD InTransferSize = 16384;
ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK) {
    // FT_Open failed
    return;
}
ftStatus = FT_SetUSBParameters(ftHandle, InTransferSize, 0);
if (ftStatus == FT_OK) {
    // In transfer size set to 16 Kbytes
} else {
    // FT_SetUSBParameters FAILED!
}
FT_Close(ftHandle);
```
5  FT-Win32 API Functions

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

Linux support for these functions has been added to allow porting windows VCP applications to Linux as well as providing improved event detection. Every effort has been made to keep the continuity between the Linux and Windows implementation of these functions as much as possible. The notable differences have been documented within.

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`.

Reference
Type definitions of the functional parameters and return codes used in the FT-Win32 interface are contained in the appendix.
5.1 **FT_W32_CreateFile**

Open the specified device and return a handle which will be used for subsequent accesses. The device can be specified by its serial number, device description, or location.

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

```c
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. **Ignored in Linux.**
- **dwShareMode**: How the device is shared. This value must be set to 0. **Ignored in Linux.**
- **lpSecurityAttributes**: This parameter has no effect and should be set to NULL.
- **dwCreate**: This parameter must be set to OPEN_EXISTING. **Ignored in Linux.**
- **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 devices serial number, and FT_OPEN_BY_DESCRIPTION if `lpszName` is the devices 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**

The meaning of `pvArg1` depends on `dwAttrsAndFlags`: if FT_OPEN_BY_SERIAL_NUMBER or FT_OPEN_BY_DESCRIPTION is set in `dwAttrsAndFlags`, `pvArg1` contains a pointer to a null terminated string that contains the device's serial number or description; if FT_OPEN_BY_LOCATION is set in `dwAttrsAndFlags`, `pvArg1` is interpreted as a value of type long that contains the location ID of the device.

`dwAccess` can be GENERIC_READ, GENERIC_WRITE or both; `dwShareMode` must be set to 0; `lpSecurityAttributes` must be set to NULL; `dwCreate` must be set to OPEN_EXISTING; `dwAttrsAndFlags` is a combination of FILE_ATTRIBUTE_NORMAL, FILE_FLAG_OVERLAPPED if overlapped I/O is used, FT_OPEN_BY_SERIAL_NUMBER or FT_OPEN_BY_DESCRIPTION or FT_OPEN_BY_LOCATION; `hTemplate` must be NULL.
Windows CE does not support overlapped IO or location IDs.

Examples

The examples that follow use these variables:

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

Open a device for overlapped I/O using its serial number

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

Open a device for non-overlapped I/O using its description

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

Open a device for non-overlapped I/O using its location

long locID;
ftStatus = FT_ListDevices(0,&locID,FT_LIST_BY_INDEX|FT_OPEN_BY_LOCATION);
ftHandle = FT_W32_CreateFile((PVOID) locID,GENERIC_READ|GENERIC_WRITE,0,0,
OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL | FT_OPEN_BY_LOCATION,
0);

if (ftHandle == INVALID_HANDLE_VALUE)
    ; // FT_W32_CreateDevice failed
5.2 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.

```c
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);
}
```
5.3 **FT_W32_ReadFile**

Read data from the device.

```c
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 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. **Ignored in Linux.**

**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, except under Windows CE and Linux where only non-overlapped IO is supported.

**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.

```c
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.

```c
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
    }
}
```
5.4 **FT_W32_WriteFile**

Write data to the device.

```c
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. **Ignored in Linux.**

**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, **except under Windows CE and Linux where only non-overlapped IO is supported.**
  
  **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 than `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.

```c
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.

```c
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 ...
    } else {
        if (dwToWrite == dwWritten){
            // FT_W32_WriteFile OK
        } else{
            // FT_W32_WriteFile timeout
        }
    }
} else {
    // FT_W32_WriteFile OK
}
```
5.5 **FT_W32_GetLastError**

Gets the last error that occurred on the device.

```
DWORD 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 and so is **not supported in Windows CE**. For a description of its use, see `FT_W32_ReadFile` and `FT_W32_WriteFile`.

In Linux this function returns a DWORD that directly maps to the FT Errors (for example the `FT_INVALID_HANDLE` error number).
5.6  **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 and so is not supported in Windows CE or Linux. For a description of its use, see **FT_W32_ReadFile** and **FT_W32_WriteFile**.
5.7 **FT_W32_ClearCommBreak**

Puts the communications line in the non-BREAK state.

```c
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.

```c
FT_HANDLE ftHandle; // setup by FT_W32_CreateFile
if (!FT_W32_ClearCommBreak(ftHandle)){
    // FT_W32_ClearCommBreak failed
} else{
    // FT_W32_ClearCommBreak OK
```
5.8 **FT_W32_ClearCommError**

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

```c
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 FTCOMSTAT 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.

```c
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
}
5.9  **FT_W32_EscapeCommFunction**

Perform an extended function.

```c
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.

```c
FT_HANDLE ftHandle; // setup by FT_W32_CreateFile

FT_W32_EscapeCommFunction(ftHandle, CLRDTS);
FT_W32_EscapeCommFunction(ftHandle, SETRTS);
```
5.10 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.

```c
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
```
5.11 FT_W32_GetCommState

This function gets the current device state.

```c
BOOL FT_W32_GetCommState (FT_HANDLE ftHandle, LPFTDCB lpftDcb)
```

**Parameters**
- `ftHandle` Handle of the device.
- `lpftDcb` Pointer to an FTDCB structure.

**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.

```c
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
```
5.12 **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 an FTTIMEOUTS 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.

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

if (!FT_W32_GetCommTimeouts(ftHandle,&ftTS))  
  // FT_W32_GetCommTimeouts failed
```
5.13  **FT\_W32\_PurgeComm**

This function purges the device.

```c
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.

```c
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
```
5.14 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
5.15 **FT_W32_SetCommMask**

This function specifies events that the device has to monitor.

```c
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.

```c
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
```
5.16 **FT_W32_SetCommState**

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

```c
BOOL FT_W32_SetCommState (FT_HANDLE ftHandle, LPFTDCB lpftDcb)
```

**Parameters**

- `ftHandle` Handle of the device.
- `lpftDcb` Pointer to an FTDCB 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 to change the baud rate.

```c
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
```
5.17 **FT_W32_SetCommTimeouts**

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

```c
BOOL FT_W32_SetCommTimeouts (FT_HANDLE ftHandle, LPFTTIMEOUTS lpftTimeouts)
```

### Parameters
- **ftHandle**
  - Handle of the device.
- **lpftTimeouts**
  - Pointer to an FTTIMEOUTS 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
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.

\[
rdTO = (dwToRead \times TS.ReadTotalTimeoutMultiplier) + TS.ReadTotalTimeoutConstant
\]

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.

\[
wTO = (dwToWrite \times TS.WriteTotalTimeoutMultiplier) + TS.WriteTotalTimeoutConstant
\]

Linux ignores the ReadIntervalTimeout, ReadTotalTimeoutMultiplier and WriteTotalTimeoutMultiplier currently.

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

```c
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
```
5.18 **FT_W32_SetupComm**

This function sets the read and write buffers.

```c
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.
5.19 FT_W32_WaitCommEvent
This function waits for an event to occur.

```c
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, except under Windows CE and Linux where only non-overlapped IO is supported.

**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
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
    }
}
6 Appendix

This section contains type definitions of the functional parameters and return codes used in the D2XX programming interface. It also contains a copy of the current FTD2XX.H file.
6.1 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-Visual C++ applications, check the application code examples on the FTDI website 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)
- **LPDWORD**
  - Pointer to 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_NOT_SUPPORTED` = 17
- `FT_OTHER_ERROR` = 18

**Flags (see `FT_OpenEx`)**

- `FT_OPEN_BY_SERIAL_NUMBER` = 1
- `FT_OPEN_BY_DESCRIPTION` = 2
- `FT_OPEN_BY_LOCATION` = 4

**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  
FT_DEVICE_2232C = 4  
FT_DEVICE_232R = 5

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

Modem Status (see FT_GetModemStatus)
CTS = 0x10  
DSR = 0x20  
RI = 0x40  
DCD = 0x80

FT232R CBUS EEPROM OPTIONS - Ignored for FT245R (see FT_EE_Program and FT_EE_Read)
CBUS_TXDEN = 0x00  
CBUS_PWRON = 0x01  
CBUS_TXLED = 0x02  
CBUS_RXLED = 0x03  
CBUS_TXRXLED = 0x04  
CBUS_SLEEP = 0x05  
CBUS_CLK48 = 0x06  
CBUS_CLK24 = 0x07
CBUS_CLK12 = 0x08
CBUS_CLK6 = 0x09
CBUS_IOMODE = 0x0A
CBUS_BITBANG_WR = 0x0B
CBUS_BITBANG_RD = 0x0C

**FT_DEVICE_LIST_INFO_NODE** (see *FT_GetDeviceInfoList*)
typedef struct _ft_device_list_info_node {
    DWORD Flags;
    DWORD Type;
    DWORD ID;
    DWORD LocId;
    char SerialNumber[16];
    char Description[64];
    FT_HANDLE ftHandle;
} FT_DEVICE_LIST_INFO_NODE;

**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
    //
    UCHAR Rev4;    // true if Rev4 chip, false otherwise
    UCHAR IsoIn;   // true if in endpoint is isochronous
    UCHAR IsoOut;  // true if out endpoint is isochronous
    UCHAR PullDownEnable;  // true if pull down enabled
    UCHAR SerNumEnable; // true if serial number to be used
    UCHAR USBVersionEnable; // true if chip uses USBVersion
    WORD USBVersion;  // BCD (0x0200 => USB2)
} FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;

**FT_PROGRAM_DATA** (EEPROM Programming Interface - compatible with DLL version 2.1.4.1 or later)
typedef struct ft_program_data {
    DWORD Signature1;   // Header - must be 0x00000000
    DWORD Signature2;   // Header - must be 0xffffffff
    DWORD Version;      // Header - FT_PROGRAM_DATA version
    // 0 = original
    // 1 = FT2232C extensions
    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
//
UCHAR Rev4; // non-zero if Rev4 chip, zero otherwise
UCHAR IsoIn; // non-zero if in endpoint is isochronous
UCHAR IsoOut; // non-zero if out endpoint is isochronous
UCHAR PullDownEnable; // non-zero if pull down enabled
UCHAR SerNumEnable; // non-zero if serial number to be used
UCHAR USBVersionEnable; // non-zero if chip uses USBVersion
WORD USBVersion; // BCD (0x0200 => USB2)

// FT232C extensions
//
UCHAR Rev5; // non-zero if Rev5 chip, zero otherwise
UCHAR IsoInA; // non-zero if in endpoint is isochronous
UCHAR IsoInB; // non-zero if in endpoint is isochronous
UCHAR IsoOutA; // non-zero if out endpoint is isochronous
UCHAR IsoOutB; // non-zero if out endpoint is isochronous
UCHAR PullDownEnable5; // non-zero if pull down enabled
UCHAR SerNumEnable5; // non-zero if serial number to be used
UCHAR USBVersionEnable5; // non-zero if chip uses USBVersion
WORD USBVersion5; // BCD (0x0200 => USB2)

#define AIsHighCurrent 0x0001
#define BIsHighCurrent 0x0002
#define IFAIsFifo 0x0004
#define IFAIsFifoTar 0x0008
#define IFAIsFastSer 0x0010
#define AIsVCP 0x0020
#define IFBIsFifo 0x0040
#define IFBIsFifoTar 0x0080
#define IFBIsFastSer 0x0100
#define BIsVCP 0x0200

FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;

FT_PROGRAM_DATA (EEPROM Programming Interface - compatible with DLL version 3.1.6.1 or later)
typedef struct ft_program_data { 
  DWORD Signature1; // Header - must be 0x00000000
  DWORD Signature2; // Header - must be 0xffffffff
  DWORD Version; // Header - FT_PROGRAM_DATA version
  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
} FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;
WORD RemoteWakeup; // 0 = not capable, 1 = capable

//
// Rev4 extensions
//
UCHAR Rev4; // non-zero if Rev4 chip, zero otherwise
UCHAR IsoIn; // non-zero if in endpoint is isochronous
UCHAR IsoOut; // non-zero if out endpoint is isochronous
UCHAR PullDownEnable; // non-zero if pull down enabled
UCHAR SerNumEnable; // non-zero if serial number to be used
UCHAR USBVersionEnable; // non-zero if chip uses USBVersion
WORD USBVersion; // BCD (0x0200 => USB2)

//
// FT232C extensions
//
UCHAR Rev5; // non-zero if Rev5 chip, zero otherwise
UCHAR IsoInA; // non-zero if in endpoint is isochronous
UCHAR IsoInB; // non-zero if in endpoint is isochronous
UCHAR IsoOutA; // non-zero if out endpoint is isochronous
UCHAR IsoOutB; // non-zero if out endpoint is isochronous
UCHAR PullDownEnable5; // non-zero if pull down enabled
UCHAR SerNumEnable5; // non-zero if serial number to be used
UCHAR USBVersionEnable5; // non-zero if chip uses USBVersion
WORD USBVersion5; // BCD (0x0200 => USB2)
UCHAR AIsHighCurrent; // non-zero if interface is high current
UCHAR BIsHighCurrent; // non-zero if interface is high current
UCHAR IFAIsFifo; // non-zero if interface is 245 FIFO
UCHAR IFAIsFifoTar; // non-zero if interface is 245 FIFO CPU target
UCHAR IFAIsFastSer; // non-zero if interface is Fast serial
UCHAR AIsVCP; // non-zero if interface is to use VCP drivers
UCHAR IFBIsFifo; // non-zero if interface is 245 FIFO
UCHAR IFBIsFifoTar; // non-zero if interface is 245 FIFO CPU target
UCHAR IFBIsFastSer; // non-zero if interface is Fast serial
UCHAR BIsVCP; // non-zero if interface is to use VCP drivers

//
// FT232R extensions
//
UCHAR UseExtOsc; // Use External Oscillator
UCHAR HighDriveIOs; // High Drive I/Os
UCHAR EndpointSize; // Endpoint size
UCHAR PullDownEnableR; // non-zero if pull down enabled
UCHAR SerNumEnableR; // non-zero if serial number to be used
UCHAR InvertTXD; // non-zero if invert TXD
UCHAR InvertRXD; // non-zero if invert RXD
UCHAR InvertRTS; // non-zero if invert RTS
UCHAR InvertCTS; // non-zero if invert CTS
UCHAR InvertDTR; // non-zero if invert DTR
UCHAR InvertDSR; // non-zero if invert DSR
UCHAR InvertDCD; // non-zero if invert DCD
UCHAR InvertRI; // non-zero if invert RI
UCHAR Cbus0; // Cbus Mux control - Ignored for FT245R
UCHAR Cbus1; // Cbus Mux control - Ignored for FT245R
UCHAR Cbus2; // Cbus Mux control - Ignored for FT245R
UCHAR Cbus3; // Cbus Mux control - Ignored for FT245R
UCHAR Cbus4; // Cbus Mux control - Ignored for FT245R
UCHAR RIsVCP; // non-zero if using VCP drivers

) 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 EovChar; // 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;
6.2 FTD2XX.H

/*++
Copyright (c) 2001-2005  Future Technology Devices International Ltd.
Module Name:
    ftd2xx.h
Abstract:
    Native USB device driver for FTDI FT8U232/245
    FTD2XX library definitions
Environment:
    kernel & user mode
Revision History:

13/03/01    awm    Created.
13/01/03    awm    Added device information support.
19/03/03    awm    Added FT_W32_CancelIo.
12/06/03    awm    Added FT_StopInTask and FT_RestartInTask.
18/09/03    awm    Added FT_SetResetPipeRetryCount.
10/10/03    awm    Added FT_ResetPort.
23/01/04    awm    Added support for open-by-location.
16/03/04    awm    Added support for FT2232C.
23/09/04    awm    Added support for FT232R.
20/10/04    awm    Added FT_CyclePort.
18/01/05    awm    Added FT_DEVICE_LIST_INFO_NODE type.
11/02/05    awm    Added LocId to FT_DEVICE_LIST_INFO_NODE.
25/08/05    awm    Added FT_GetDeadmanTimeout.
02/12/05    awm    Removed obsolete references.
05/12/05    awm    Added FT_GetVersion, FT_GetVersionEx.

--*/

#ifndef FTD2XX_H
#define FTD2XX_H

// The following ifdef block is the standard way of creating macros
// which make exporting from a DLL simpler. All files within this DLL
// are compiled with the FTD2XX_EXPORTS symbol defined on the command line.
// This symbol should not be defined on any project that uses this DLL.
// This way any other project whose source files include this file see
// FTD2XX_API functions as being imported from a DLL, whereas this DLL
// sees symbols defined with this macro as being exported.

#ifdef FTD2XX_EXPORTS
#define FTD2XX_API __declspec(dllexport)
#else
#define FTD2XX_API __declspec(dllimport)
#endif

#endif

#include "ftdevice.h"
typedef PVOID FT_HANDLE;
typedef ULONG FT_STATUS;

// Device status
enum {
    FT_OK,
    FT_INVALID_HANDLE,
    FT_DEVICE_NOT_FOUND,
    FT_DEVICE_NOT_OPENED,
    FT_IO_ERROR,
    FT_INSUFFICIENT_RESOURCES,
    FT_INVALID_PARAMETER,
    FT_INVALID_BAUD_RATE,
    FTDEVICE_NOT_OPENED_FOR_ERASE,
    FT_DEVICE_NOT_OPENED_FOR_WRITE,
    FT_FAILED_TO_WRITE_DEVICE,
    FT_EEPROM_READ_FAILED,
    FT_EEPROM_WRITE_FAILED,
    FT_EEPROM_ERASE_FAILED,
    FT_EEPROM_NOT_PRESENT,
    FT_EEPROM_NOT_PROGRAMMED,
    FT_INVALID_ARGS,
    FT_NOT_SUPPORTED,
    FT_OTHER_ERROR
};

#define FT_SUCCESS(status) ((status) == FT_OK)

// FT_OpenEx Flags
#define FT_OPEN_BY_SERIAL_NUMBER 1
#define FT_OPEN_BY_DESCRIPTION 2
#define FT_OPEN_BY_LOCATION 4

// FT_ListDevices Flags (used in conjunction with FT_OpenEx Flags
#define FT_LIST_NUMBER_ONLY 0x80000000
#define FT_LIST_BY_INDEX 0x40000000
#define FT_LIST_ALL 0x20000000

#define FT_LIST_MASK (FT_LIST_NUMBER_ONLY|FT_LIST_BY_INDEX|FT_LIST_ALL)

// Baud Rates
#define FT_BAUD_300 300
#define FT_BAUD_600 600
#define FT_BAUD_1200 1200
#define FT_BAUD_2400 2400
```c
#define FT_BAUD_4800 4800
#define FT_BAUD_9600 9600
#define FT_BAUD_14400 14400
#define FT_BAUD_19200 19200
#define FT_BAUD_38400 38400
#define FT_BAUD_57600 57600
#define FT_BAUD_115200 115200
#define FT_BAUD_230400 230400
#define FT_BAUD_460800 460800
#define FT_BAUD_921600 921600

//
// Word Lengths
//

#define FT_BITS_8 (UCHAR) 8
#define FT_BITS_7 (UCHAR) 7
#define FT_BITS_6 (UCHAR) 6
#define FT_BITS_5 (UCHAR) 5

//
// Stop Bits
//

#define FT_STOP_BITS_1 (UCHAR) 0
#define FT_STOP_BITS_1_5 (UCHAR) 1
#define FT_STOP_BITS_2 (UCHAR) 2

//
// Parity
//

#define FT_PARITY_NONE (UCHAR) 0
#define FT_PARITY_ODD (UCHAR) 1
#define FT_PARITY_EVEN (UCHAR) 2
#define FT_PARITY_MARK (UCHAR) 3
#define FT_PARITY_SPACE (UCHAR) 4

//
// Flow Control
//

#define FT_FLOW_NONE 0x0000
#define FT_FLOW_RTS_CTS 0x0100
#define FT_FLOW_DTR_DSR 0x0200
#define FT_FLOW_XON_XOFF 0x0400

//
// Purge rx and tx buffers
//

#define FT_PURGE_RX 1
#define FT_PURGE_TX 2

//
// Events
//

typedef void (*PFT_EVENT_HANDLER)(DWORD,DWORD);
```
#define FT_EVENT_RXCHAR 1
#define FT_EVENT_MODEM_STATUS 2

// Timeouts
//
#define FT_DEFAULT_RX_TIMEOUT 300
#define FT_DEFAULT_TX_TIMEOUT 300

// Device types
//
typedef ULONG FT_DEVICE;
enum {
    FT_DEVICE_BM,
    FT_DEVICE_AM,
    FT_DEVICE_100AX,
    FT_DEVICE_UNKNOWN,
    FT_DEVICE_2232C,
    FT_DEVICE_232R
};

#ifdef __cplusplus
extern "C" {
#endif

FTD2XX_API FT_STATUS WINAPI FT_Open( int deviceNumber,
                                      FT_HANDLE *pHandle );

FTD2XX_API FT_STATUS WINAPI FT_OpenEx( PVOID pArg1,
                                       DWORD Flags,
                                       FT_HANDLE *pHandle );

FTD2XX_API FT_STATUS WINAPI FT_ListDevices( PVOID pArg1,
                                            PVOID pArg2,
                                            DWORD Flags );

FTD2XX_API FT_STATUS WINAPI FT_Close( FT_HANDLE ftHandle );

#ifdef __cplusplus
}
#endif
FTD2XX_API
FT_STATUS WINAPI FT_Read(
    FT_HANDLE ftHandle,
    LPVOID lpBuffer,
    DWORD nBufferSize,
    LPDWORD lpBytesReturned
);

FTD2XX_API
FT_STATUS WINAPI FT_Write(
    FT_HANDLE ftHandle,
    LPVOID lpBuffer,
    DWORD nBufferSize,
    LPDWORD lpBytesWritten
);

FTD2XX_API
FT_STATUS WINAPI FT_IoCtl(
    FT_HANDLE ftHandle,
    DWORD dwIoControlCode,
    LPVOID lpInBuf,
    DWORD nInBufSize,
    LPVOID lpOutBuf,
    DWORD nOutBufSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);

FTD2XX_API
FT_STATUS WINAPI FT_SetBaudRate(
    FT_HANDLE ftHandle,
    ULONG BaudRate
);

FTD2XX_API
FT_STATUS WINAPI FT_SetDivisor(
    FT_HANDLE ftHandle,
    USHORT Divisor
);

FTD2XX_API
FT_STATUS WINAPI FT_SetDataCharacteristics(
    FT_HANDLE ftHandle,
    UCHAR WordLength,
    UCHAR StopBits,
    UCHAR Parity
);

FTD2XX_API
FT_STATUS WINAPI FT_SetFlowControl(
    FT_HANDLE ftHandle,
    USHORT FlowControl,
    UCHAR XonChar,
    UCHAR XoffChar
);

FTD2XX_API
FT_STATUS WINAPI FT_ResetDevice(
FT_HANDLE ftHandle
    );

FTD2XX_API
FT_STATUS WINAPI FT_SetDtr(
    FT_HANDLE ftHandle
    );

FTD2XX_API
FT_STATUS WINAPI FT_ClrDtr(
    FT_HANDLE ftHandle
    );

FTD2XX_API
FT_STATUS WINAPI FT_SetRts(
    FT_HANDLE ftHandle
    );

FTD2XX_API
FT_STATUS WINAPI FT_ClrRts(
    FT_HANDLE ftHandle
    );

FTD2XX_API
FT_STATUS WINAPI FT_GetModemStatus(
    FT_HANDLE ftHandle,
    ULONG *pModemStatus
    );

FTD2XX_API
FT_STATUS WINAPI FT_SetChars(
    FT_HANDLE ftHandle,
    UCHAR EventChar,
    UCHAR EventCharEnabled,
    UCHAR ErrorChar,
    UCHAR ErrorCharEnabled
    );

FTD2XX_API
FT_STATUS WINAPI FT_Purge(
    FT_HANDLE ftHandle,
    ULONG Mask
    );

FTD2XX_API
FT_STATUS WINAPI FT_SetTimeouts(
    FT_HANDLE ftHandle,
    ULONG ReadTimeout,
    ULONG WriteTimeout
    );

FTD2XX_API
FT_STATUS WINAPI FT_GetQueueStatus(
    FT_HANDLE ftHandle,
    DWORD *dwRxBytes
    );

FTD2XX_API
FT_STATUS WINAPI FT_SetEventNotification(
    FT_HANDLE ftHandle,
    DWORD Mask,
    PVOID Param
);

FTD2XX_API
FT_STATUS WINAPI FT_GetStatus(
    FT_HANDLE ftHandle,
    DWORD *dwRxBytes,
    DWORD *dwTxBytes,
    DWORD *dwEventDWord
);

FTD2XX_API
FT_STATUS WINAPI FT_SetBreakOn(
    FT_HANDLE ftHandle
);

FTD2XX_API
FT_STATUS WINAPI FT_SetBreakOff(
    FT_HANDLE ftHandle
);

FTD2XX_API
FT_STATUS WINAPI FT_SetWaitMask(
    FT_HANDLE ftHandle,
    DWORD Mask
);

FTD2XX_API
FT_STATUS WINAPI FT_WaitOnMask(
    FT_HANDLE ftHandle,
    DWORD *Mask
);

FTD2XX_API
FT_STATUS WINAPI FT_GetEventStatus(
    FT_HANDLE ftHandle,
    DWORD *dwEventDWord
);

FTD2XX_API
FT_STATUS WINAPI FT_ReadEE(
    FT_HANDLE ftHandle,
    DWORD dwWordOffset,
    LPWORD lpwValue
);

FTD2XX_API
FT_STATUS WINAPI FT_WriteEE(
    FT_HANDLE ftHandle,
    DWORD dwWordOffset,
    WORD wValue
);

FTD2XX_API
FT_STATUS WINAPI FT_EraseEE(
typedef struct ft_program_data {
  DWORD Signature1;  // Header - must be 0x00000000
  DWORD Signature2;  // Header - must be 0xffffffff
  DWORD Version;    // Header - FT_PROGRAM_DATA version
    // 0 = original
    // 1 = FT2232C extensions
    // 2 = FT232R extensions
  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
    //
    UCHAR Rev4;     // non-zero if Rev4 chip, zero otherwise
    UCHAR IsoIn;    // non-zero if in endpoint is isochronous
    UCHAR IsoOut;   // non-zero if out endpoint is isochronous
    UCHAR PullDownEnable; // non-zero if pull down enabled
    UCHAR SerNumEnable; // non-zero if serial number to be used
    UCHAR USBVersionEnable; // non-zero if chip uses USBVersion
    WORD USBVersion; // BCD (0x0200 => USB2)
    // FT2232C extensions
    //
    UCHAR Rev5;     // non-zero if Rev5 chip, zero otherwise
    UCHAR IsoInA;   // non-zero if in endpoint is isochronous
    UCHAR IsoInB;   // non-zero if in endpoint is isochronous
    UCHAR IsoOutA;  // non-zero if out endpoint is isochronous
    UCHAR IsoOutB;  // non-zero if out endpoint is isochronous
    UCHAR PullDownEnable5; // non-zero if pull down enabled
    UCHAR SerNumEnable5; // non-zero if serial number to be used
    UCHAR USBVersionEnable5; // non-zero if chip uses USBVersion
    WORD USBVersion5; // BCD (0x0200 => USB2)
    UCHAR AIsHighCurrent; // non-zero if interface is high current
    UCHAR BIsHighCurrent; // non-zero if interface is high current
    UCHAR IFAlsFifo;    // non-zero if interface is 245 FIFO
    UCHAR IFAlsFifoTar; // non-zero if interface is 245 FIFO CPU target
    UCHAR IFAlsFastSer; // non-zero if interface is Fast serial
    UCHAR AlsVCP;       // non-zero if interface is to use VCP drivers
    UCHAR IFBsFifo;     // non-zero if interface is 245 FIFO
    UCHAR IFBsFifoTar;  // non-zero if interface is 245 FIFO CPU target
    UCHAR IFBsFastSer;  // non-zero if interface is Fast serial
    UCHAR BIsVCP;       // non-zero if interface is to use VCP drivers
}
// FT232R extensions

//
UCHAR UseExtOsc;       // Use External Oscillator
UCHAR HighDriveIOs;   // High Drive I/Os
UCHAR EndpointSize;   // Endpoint size

UCHAR PullDownEnableR;   // non-zero if pull down enabled
UCHAR SerNumEnableR;    // non-zero if serial number to be used

UCHAR InvertTXD;    // non-zero if invert TXD
UCHAR InvertRXD;    // non-zero if invert RXD
UCHAR InvertRTS;    // non-zero if invert RTS
UCHAR InvertCTS;    // non-zero if invert CTS
UCHAR InvertDTR;    // non-zero if invert DTR
UCHAR InvertDSR;    // non-zero if invert DSR
UCHAR InvertDCD;    // non-zero if invert DCD
UCHAR InvertRI;     // non-zero if invert RI

UCHAR Cbus0;        // Cbus Mux control
UCHAR Cbus1;        // Cbus Mux control
UCHAR Cbus2;        // Cbus Mux control
UCHAR Cbus3;        // Cbus Mux control
UCHAR Cbus4;        // Cbus Mux control

UCHAR RIisVCP;      // non-zero if using D2XX drivers

) FT_PROGRAM_DATA, *PFT_PROGRAM_DATA;

FTD2XX_API
FT_STATUS WINAPI FT_EE_Program(
    FT_HANDLE ftHandle,
    PFT_PROGRAM_DATA pData
);

FTD2XX_API
FT_STATUS WINAPI FT_EE_ProgramEx(
    FT_HANDLE ftHandle,
    PFT_PROGRAM_DATA pData,
    char *Manufacturer,
    char *ManufacturerId,
    char *Description,
    char *SerialNumber
);

FTD2XX_API
FT_STATUS WINAPI FT_EE_Read(
    FT_HANDLE ftHandle,
    PFT_PROGRAM_DATA pData
);

FTD2XX_API
FT_STATUS WINAPI FT_EE_ReadEx(
    FT_HANDLE ftHandle,
    PFT_PROGRAM_DATA pData,
    char *Manufacturer,
    char *ManufacturerId,
    char *Description,
    char *SerialNumber
);
FTD2XX_API
FT_STATUS WINAPI FT_EE_UASize(
    FT_HANDLE ftHandle,
    LPDWORD lpdwSize
);

FTD2XX_API
FT_STATUS WINAPI FT_EE_UAWrite(
    FT_HANDLE ftHandle,
    PUCHAR pucData,
    DWORD dwDataLen
);

FTD2XX_API
FT_STATUS WINAPI FT_EE_UARead(
    FT_HANDLE ftHandle,
    PUCHAR pucData,
    DWORD dwDataLen,
    LPDWORD lpdwBytesRead
);

FTD2XX_API
FT_STATUS WINAPI FT_SetLatencyTimer(
    FT_HANDLE ftHandle,
    UCHAR ucLatency
);

FTD2XX_API
FT_STATUS WINAPI FT_GetLatencyTimer(
    FT_HANDLE ftHandle,
    PUCHAR pucLatency
);

FTD2XX_API
FT_STATUS WINAPI FT_SetBitMode(
    FT_HANDLE ftHandle,
    UCHAR ucMask,
    UCHAR ucEnable
);

FTD2XX_API
FT_STATUS WINAPI FT_GetBitMode(
    FT_HANDLE ftHandle,
    PUCHAR pucMode
);

FTD2XX_API
FT_STATUS WINAPI FT_SetUSBParameters(
    FT_HANDLE ftHandle,
    ULONG ulInTransferSize,
    ULONG ulOutTransferSize
);

FTD2XX_API
FT_STATUS WINAPI FT_SetDeadmanTimeout( FT_HANDLE ftHandle,
ULONG ulDeadmanTimeout
);

FTD2XX_API
FT_STATUS WINAPI FT_GetDeviceInfo(
    FT_HANDLE ftHandle,
    FT_DEVICE *lpftDevice,
    LPDWORD lpdwID,
    PCHAR SerialNumber,
    PCHAR Description,
    LPVOID Dummy
);

FTD2XX_API
FT_STATUS WINAPI FT_StopInTask(
    FT_HANDLE ftHandle
);

FTD2XX_API
FT_STATUS WINAPI FT_RestartInTask(
    FT_HANDLE ftHandle
);

FTD2XX_API
FT_STATUS WINAPI FT_SetResetPipeRetryCount(
    FT_HANDLE ftHandle,
    DWORD dwCount
);

FTD2XX_API
FT_STATUS WINAPI FT_ResetPort(
    FT_HANDLE ftHandle
);

FTD2XX_API
FT_STATUS WINAPI FT_CyclePort(
    FT_HANDLE ftHandle
);

//
// Win32-type functions
//

FTD2XX_API
FT_HANDLE WINAPI FT_W32_CreateFile(
    LPCSTR lpszName,
    DWORD dwAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreate,
    DWORD dwAttrsAndFlags,
    HANDLE hTemplate
);

FTD2XX_API
BOOL WINAPI FT_W32_CloseHandle(
    FT_HANDLE ftHandle
);
FTD2XX_API
BOOL WINAPI FT_W32_ReadFile(
  FT_HANDLE ftHandle,
  LPVOID lpBuffer,
  DWORD nBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);

FTD2XX_API
BOOL WINAPI FT_W32_WriteFile(
  FT_HANDLE ftHandle,
  LPVOID lpBuffer,
  DWORD nBufferSize,
  LPDWORD lpBytesWritten,
  LPOVERLAPPED lpOverlapped
);

FTD2XX_API
DWORD WINAPI FT_W32_GetLastError(
  FT_HANDLE ftHandle
);

FTD2XX_API
BOOL WINAPI FT_W32_GetOverlappedResult(
  FT_HANDLE ftHandle,
  LPOVERLAPPED lpOverlapped,
  LPDWORD lpdwBytesTransferred,
  BOOL bWait
);

FTD2XX_API
BOOL WINAPI FT_W32_CancelIo(
  FT_HANDLE ftHandle
);

//
// Win32 COMM API type functions
//
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;

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 fInX: 1;  /* Enable output X-ON/X-OFF */
DWORD flnX: 1;  /* Enable input X-ON/X-OFF */
DWORD fErrorChar: 1;  /* Enable Err Replacement */
DWORD fNull: 1;  /* Enable Null stripping */
DWORD fOutX: 1;  /* Enable output X-ON/X-OFF */
DWORD fAbortOnError: 1;  /* Abort all reads and writes on Error */
WORD wReserved;  /* Not currently used */
WORD XonLim;  /* Transmit X-ON threshold */
WORD XoffLim;  /* Transmit X-OFF threshold */
BYTE ByteSize;  /* Number of bits/byte, 4-8 */
BYTE Parity;  /* 0-4=None,Odd,Even,Mark,Space */
BYTE StopBits;  /* 0,1,2 = 1, 1.5, 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 for now. */
)
FTDCB, *LPFTDCB;

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;

FTD2XX_API
BOOL WINAPI FT_W32_ClearCommBreak(  
    FT_HANDLE ftHandle
);

FTD2XX_API
BOOL WINAPI FT_W32_ClearCommError(  
    FT_HANDLE ftHandle,
    LPDWORD lpdwErrors,
    LPFTCOMSTAT lpftComstat
);

FTD2XX_API
BOOL WINAPI FT_W32_EscapeCommFunction(  
    FT_HANDLE ftHandle,
    DWORD dwFunc
);

FTD2XX_API
BOOL WINAPI FT_W32_GetCommModemStatus(  
    FT_HANDLE ftHandle,
    *LPFTMSTAT lpftMstat
);
FTD2XX_API
BOOL WINAPI FT_W32_GetCommState(
    FT_HANDLE ftHandle,
    LPFTDCB lpftDcb
);

FTD2XX_API
BOOL WINAPI FT_W32_GetCommTimeouts(
    FT_HANDLE ftHandle,
    FTTIMEOUTS *pTimeouts
);

FTD2XX_API
BOOL WINAPI FT_W32_PurgeComm(
    FT_HANDLE ftHandle,
    DWORD dwMask
);

FTD2XX_API
BOOL WINAPI FT_W32_SetCommBreak(
    FT_HANDLE ftHandle
);

FTD2XX_API
BOOL WINAPI FT_W32_SetCommMask(
    FT_HANDLE ftHandle,
    ULONG ulEventMask
);

FTD2XX_API
BOOL WINAPI FT_W32_SetCommState(
    FT_HANDLE ftHandle,
    LPFTDCB lpftDcb
);

FTD2XX_API
BOOL WINAPI FT_W32_SetCommTimeouts(
    FT_HANDLE ftHandle,
    FTTIMEOUTS *pTimeouts
);

FTD2XX_API
BOOL WINAPI FT_W32_SetupComm(
    FT_HANDLE ftHandle,
    DWORD dwReadBufferSize,
    DWORD dwWriteBufferSize
);

FTD2XX_API
BOOL WINAPI FT_W32_WaitCommEvent(
    FT_HANDLE ftHandle,
    PULONG pulEvent,
    LPOVERLAPPED lpOverlapped
);
typedef struct _ft_device_list_info_node {
    ULONG Flags;
    ULONG Type;
    ULONG ID;
    DWORD LocId;
    char SerialNumber[16];
    char Description[64];
    FT_HANDLE ftHandle;
} FT_DEVICE_LIST_INFO_NODE;

FTD2XX_API
FT_STATUS WINAPI FT_CreateDeviceInfoList(
    LPDWORD lpdwNumDevs
);

FTD2XX_API
FT_STATUS WINAPI FT_GetDeviceInfoList(
    FT_DEVICE_LIST_INFO_NODE *pDest,
    LPDWORD lpdwNumDevs
);

FTD2XX_API
FT_STATUS WINAPI FT_GetDeviceInfoDetail(
    DWORD dwIndex,
    LPDWORD lpdwFlags,
    LPDWORD lpdwType,
    LPDWORD lpdwID,
    LPDWORD lpdwLocId,
    LPVOID lpSerialNumber,
    LPVOID lpDescription,
    FT_HANDLE *pftHandle
);

//
// Version information
//

FTD2XX_API
FT_STATUS WINAPI FT_GetDriverVersion(
    FT_HANDLE ftHandle,
    LPDWORD lpdwVersion
);

FTD2XX_API
FT_STATUS WINAPI FT_GetLibraryVersion(
    LPDWORD lpdwVersion
);

#ifdef __cplusplus

#endif
} #endif

#endif /* FTD2XX_H */
## Index

### - B -
- Baud Rate 19, 20
- Bit Mode 71, 72

### - C -
- Close 14, 78

### - D -
- D2XX Programmer's Guide 4

### - E -
- EEPROM 56, 57, 58, 59, 61, 62, 64, 65, 66, 67
- Events 35

### - F -
- FT_Close 14
- FT_ClrDtr 24
- FT_ClrRts 26
- FT_CreateDeviceInfoList 48
- FT_CyclePort 47
- FT_EE_Program 62
- FT_EE_ProgramEx 64
- FT_EE_Read 59
- FT_EE_ReadEx 61
- FT_EE_UARead 65
- FT_EE_UASize 67
- FT_EE_UAWrite 66
- FT_EraseEE 58
- FT_GetBitMode 71
- FT_GetDeviceInfo 41
- FT_GetDeviceInfoDetail 51
- FT_GetDeviceInfoList 49
- FT_GetLatencyTimer 69
- FT_GetModemStatus 27
- FT_GetQueueStatus 31
- FT_GetStatus 34
- FT_GetVIDPID 7
- FT_IOCTL 38
- FT_ListDevices 8
- FT_Open 11
- FT_OpenEx 12
- FT_Purge 29
- FT_Read 15
- FT_ReadEE 56
- FT_ResetDevice 18
- FT_ResetPort 46
- FT_RestartInTask 45
- FT_SetBaudRate 19
- FT_SetBitMode 72
- FT_SetBreakOff 33
- FT_SetBreakOn 32
- FT_SetChars 28
- FT_SetDataCharacteristics 21
- FT_SetDivisor 20
- FT_SetDtr 23
- FT_SetEventNotification 35
- FT_SetFlowControl 22
- FT_SetLatencyTimer 70
- FT_SetResetPipeRetryCount 43
- FT_SetRts 25
- FT_SetTimeouts 30
- FT_SetUSBParameters 74
- FT_GetVIDPID 6
- FT_SetWaitMask 39
- FT_StopInTask 44
- FT_W32_ClearCommBreak 86
- FT_W32_ClearCommError 87
- FT_W32_CloseHandle 78
- FT_W32_CreateFile 76
- FT_W32_EscapeCommFunction 89
- FT_W32_GetCommModemStatus 90
- FT_W32_GetCommState 91
- FT_W32_GetCommTimeouts 92
- FT_W32_GetLastError 84
- FT_W32_GetOverlappedResult 85
- FT_W32_PurgeComm 93
- FT_W32_ReadFile 79
- FT_W32_SetCommBreak 94
- FT_W32_SetCommMask 95
- FT_W32_SetCommState 96
- FT_W32_SetCommTimeouts 97
- FT_W32_SetupComm 98
Handshaking 22

Introduction 4

Latency 69, 70

Modem Signals 23, 24, 25, 26

Open 11, 12, 76

Read 15, 56, 59, 61, 65, 79

Type Definitions 102

Unplug-Replug 47

Welcome 4
Write 17, 57, 62, 64, 66, 82