MCF5272 USB SW Developer Manual. MOTUSB Host Driver for CBI & Isochronous Transfers.

M5272/USB/HD/CBII
Rev. 0.3 05/2002

DigitalDNA from Motorola

m

CONTENTS

Paragraph Title Page

ILLUSTRATIONS

Figure **Title** **Page**

About this document.

This document describes the functionality of the MOTUSB Device Driver and user mode library, and how it is employed in user applications.

Audience.

This document targets USB software developers on the Windows 2000 Host platform.

Suggested reading.

[1] Microsoft Platform SDK, Windows 2000 DDK Documentation
[2] Universal Serial Bus 1.1 Specification

Definitions, Acronyms, and Abbreviations.

The following list defines the acronyms and abbreviations used in this document.

USB	Universal Serial Bus
MOTUSB	Name of this Driver
Win32	Microsoft Windows 32 bit platform
ZLP	Zero Length Packet
WDM	Windows Driver Model
USBDI	USB Driver Interface
HID	Human Interface Device class
API	Application programming interface
HCD	Host Controller Driver
GUID	Global Unique Identifier
PnP	Plug and Play
SDK	Software Development Kit
DDK	Driver Development Kit
PC	Personal Computer
I/O	Input / Output
OS	Operating System

1. Introduction.

1.1. Overview

MOTUSB is a generic Universal Serial Bus (USB) Device Driver for Windows 2000, whose main purpose is to provide access to USB for user mode Win32 applications. This Driver is not Device specific; so that various classes of USB Devices can use it. Support for the USB is built into the Windows 2000 operating system, and developers can either use the Device Driver provided, or create a USB Client Driver manually if the OS does not provide the Driver for that particular Device class.

By using the generic MOTUSB Device Driver it is possible to perform new USB Device development without the necessity to spend time and effort developing a new Device Driver. This may prove to be especially useful during development or testing of a new Device.

1.2. System Requirements.

Hardware platforms:

- Single CPU Intel x386 based PC with Open Host Controller or Universal Host Controller.

Operation systems:

- Windows 2000 Professional

Driver Client developer software:

- Visual C++ 6.0 Professional Edition
- Microsoft Platform SDK for Windows 2000 (Recommended)

Driver developer software:

- Visual C++ 6.0 Professional Edition
- Microsoft Windows 2000 Driver Development Kit

1.3. Driver Capabilities.

- Complies with WDM
- Provides interface to access USB Device from user mode Win32 Client application
- Supports control, bulk, interrupt and isochroous transfer types
- Data transmission on pipes is similar to the data flow on file
- Supports asynchronous (overlapped) I/O
- Can manage connections to several Devices at the same time
- Can be used from multiple threads (processes) at the same time

1.4. Driver Package Content

The Driver package is divided into 3 parts:
1) User Part. Several binary modules are provided: Driver, library and installation file for a sample USB Device on the Motorola ColdFire5272 Evaluation Board.

\ bin

motusb.sys	- Kernel mode Driver
motusb.dll	- User mode library
mcf5272.inf	- Setup (INF) file for sample USB Device

2) Client software. Headers and libraries required for the MOTUSB Client software developer are provided; located at.

\inc

motioctl.h	- defines MOTUSB I/O controls and structures
motstatus.h	- defines MOTUSB Driver and library errors codes.
motusb.h	- defines motusb.dll library programming interface
usb100.h	- defines USB1.0 spec. constants and structures (provided by Microsoft DDK).

\lib

motusb.lib - static library required for linking with Client application, which use *motusb.dll* library API functions.

3) Driver and library source code.

\ src

\sys - MOTUSB Driver source code

\dll - MOTUSB dynamic library source code.

(All paths are specified relative to the MOTUSB package installation directory).

1.5. Quick Start Guide.

This section is intended as a quick MOTUSB Driver INSTALLATION GUIDE for the USB MCF5272 Development Board Firmware

1.5.1. System requirements:

- Single CPU Intel i386 based PC with USB Ports.
- Windows 2000 Professional OS.

NOTE: The Firmware must be downloaded and started prior to Driver installation. The installation will be initiated by the system automatically when connecting the Device to the PC.

1.5.2. Driver installation steps.

1. Logon to Windows 2000 using an administrator account.

2. Ensure that the following 3 files are all contained in the Driver installation directory: **motusb.sys, motusb.dll, mcf5272.inf**

3. Ensure that the VendorID and ProductID members of the Device descriptor on Device have not changed. If you have to change them, it is necessary to make a new installation (INF) file for the VendorID and ProductID member values combination. (See MOTUSB Driver Guide, Chapter 4 for detailed information on the INF file).

4. Connect the Host PC with the UFTP Device running on the MCF5272 development board via a USB cable.

5. “Found New Hardware Wizard” dialog with string “USB Device” will appear. Select “Next” button.

6. Select the radio button labeled "Search for a suitable Driver for your Device (Recommended)" and then hit the "Next" button.

7. “Locate Driver Files” page will appear, click the "Next" button

8. “Insert manufacturer installation disk on the drive…” file prompt dialog will appear. Specify the folder where all Driver files are located and click ok.

9. “Driver Files Search Result” page should appear. If the Driver path is specified correctly “Windows found a Driver for this Device” and the path to mcf5272.inf strings will be shown at the center of the page.

10. Hit the "Next" button, whereupon the "copying Files" message box will be seen briefly; then once again the "Found New Hardware Wizard" box, now displaying the subheading "Hardware Install: The hardware installation is complete". Hit the "Finish" button.

11. A copy of **motusb.sys** should be in the %SystemRoot%\System32\Drivers directory, and the **motusb.dll** in the %SystemRoot%\System32 directory. If the final "Add New Hardware Wizard" box indicates any error, or if the OS indicates that a reboot is required in order to finish the installation of this Device, something has gone wrong. Check the Inf file or Install directory, follow the instructions again for a ‘clean’ install, and start over.

2. Driver Model.

2.1. Driver Model Overview.

The MOTUSB Driver is based on the Windows Driver Model (WDM) architecture. The latest Microsoft Windows operating systems family has begun USB support in WDM. They include USB Device Drivers for hubs, Host controllers and some Device classes (audio, mass-storage, HID, etc.). As well as built-in software components, these systems provide a programming interface for USB Device Drivers, called Universal Serial Bus Driver Interface (USBDI). However USBDI can only be used by kernel-mode components (Drivers), and none of the USB functionality is available in user-mode.

2.2. USB Driver Stack.

All USB Device Drivers in WDM are USBDI Client Drivers. WDM Client Drivers are technically layered and organized as a Driver stack. A USB Client Driver overlays the Drivers USBD.SYS, USBHUB.SYS, and either UHCD.SYS or OPENHCI.SYS. The relationship between these Drivers is illustrated in Figure 2-1. The USB Client Drivers call USBD.SYS to perform the Device configuration and perform the various transfer types. MOTUSB handles the Device configuration calls and the details of communication with the bus Drivers. However, it is of interest to know something more about how communication occurs between the Client Driver and the bus Drivers.

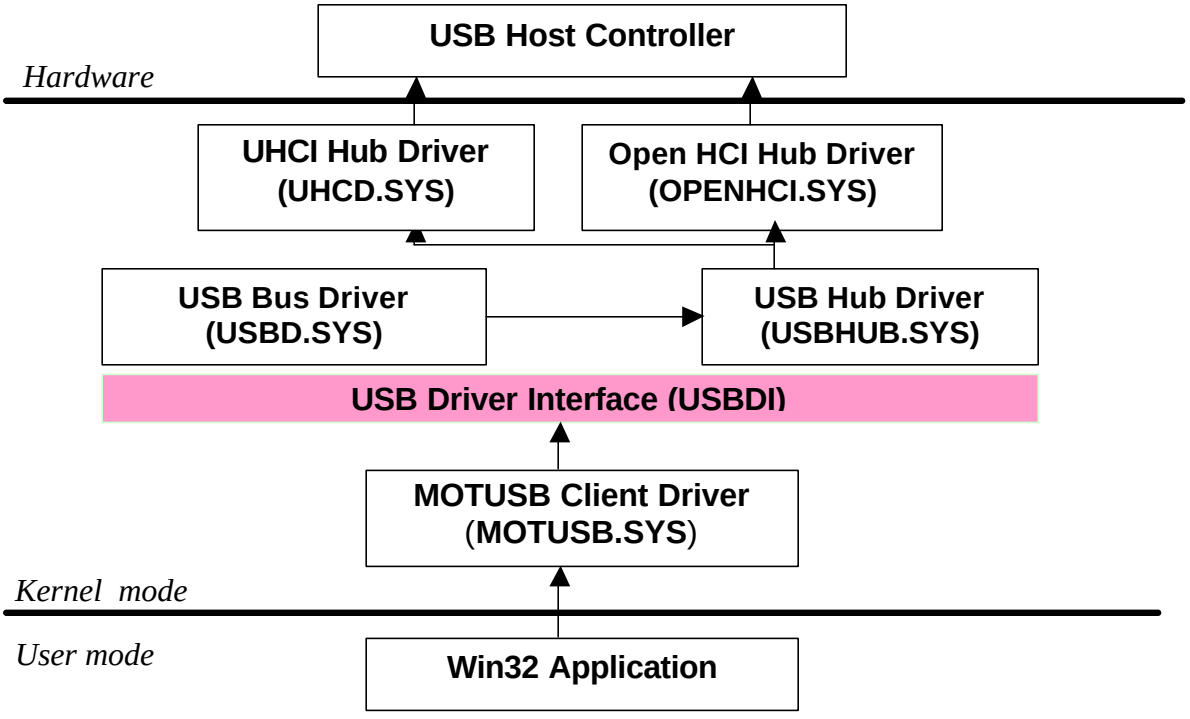

Fig 2.1 USB Stack

2.3. Communication Model.

MOTUSB Driver communication with the user mode Client application consists of connections to Device and pipe objects. Connection to Device or pipe objects is similar to opening file objects. For each physical Device connected for which MOTUSB installed, the Driver creates a Device object. The Client application can perform a Device enumeration procedure, select the required Device and open a handle to this Device or other pipe objects.

The MOTUSB Device Driver is not limited by the Client application handles opened to the Device. Several threads or processes can use the same handle to the same Device; also a single thread (process) can open several handles. The MOTUSB Driver is not responsible for actual Device requests and data flow logic and is represented as an operational block only, providing the gate to take control of the USB Device from within the user mode Client application.

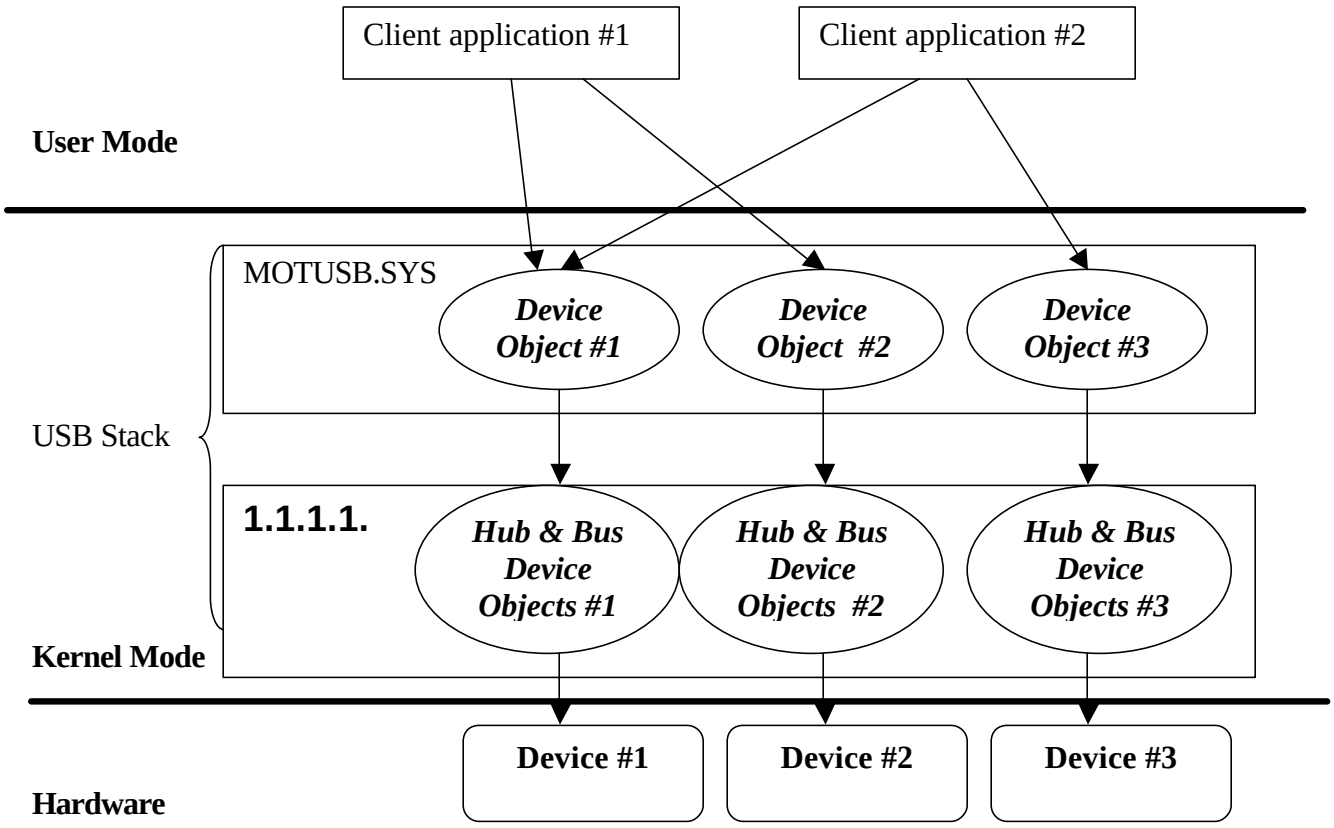

Fig 2.2 Communication model

2.4. Device Object.

The USB Client Driver is loaded by the system components when connecting a USB Device to a USB port. The PCI Enumerator component performs USB Driver selection, according to the Drivers installed on the system, which loads and invokes the Driver ***AddDevice*** dispatch table routine. As a result of this routine the MOTUSB Device Driver itself creates a Device object and attaches this Device object to the USB Driver stack.

Each MOTUSB Device object is associated with a physical USB Device that is connected to the USB. Due to this fact, MOTUSB can handle more than one Device connection.

Table 2.1 Device object states from Client point of view.

State	Description
Disconnected	No physical Device connection. Device object not created or destroyed. All handles opened to Device became invalid and user mode Client is responsible to close them.
Connected	Physical Device connection exists. Device object created. Device became configured (unconfigured) depending on the MOTUSB registry settings. No invalid handles to Device became valid.
Opened	The handle to the Device object opened. Client application can perform Device request on the Device.
Configured	Active configuration for Device selected. Client application can perform Device request on the Device, open pipes and perform and interrupt transfers on those pipes.
Unconfigured	No active configuration for Device selected. Client application can perform only limited set of requests. No pipe connections can exist. Note: Application developers should not use this state. The purpose of this state provided in MOTUSB is for USB test software only.
Locked	Lock access to Device for other owners with handle for the same Device. Client application can lock access to the Device in two ways: for 100% of working time, to monopolize access to the Device lock on demand ensuring that some requests or data flow sequences will not be interrupted by another USB Client application.

2.4.1. Default Device Configuration.

When Device object creation occurs the Driver saves the Device and all configuration descriptors. Following this the Driver performs SET_CONFIGURATION requests for the configuration #0, and configures all the interfaces in that configuration.

2.4.2. Device Interface ID.

MOTUSB registers ”Device Interface ID“ for every Device object it creates. The “Device Interface ID” (henceforward “Device interface”) itself is a global unique identifier (GUID). The MOTUSB Device Interface GUID is defined in the *motioctl.h* header file.

```
#define GUID_CLASS_MOTUSB                                           \
{0x239d60c9, 0xccaf, 0x11d5,                                \
{0xac, 0x21, 0x20, 0x4c, 0x4f, 0x4f, 0x50, 0x20}}
```

The operating system uses this GUID to generate a unique Device name for each Device object in the system. By using such a Device naming scheme, the OS solves all Device naming issues across the entire system.

2.4.3. Device Enumeration By Client.

The OS provides enumeration of Devices by Device Interface ID with the Setup API functions:

```
SetupDiGetClassDevs
SetupDiEnumDeviceInterfaces
and others
```

These functions require the Device Interface GUID, which can be found in *motioctl.h* header file as a **GUID_CLASS_MOTUSB** definition constant. This GUID is shared across all components based on MOTUSB, since each Device object created by MOTUSB has the same Device Interface ID.

As a result of Device enumeration functions *SetupDiGetClassDevs* and *SetupDiEnumDeviceInterfaces*, the Client application retrieves a list of all Device objects. In order to differentiate between the Devices an application should query the Device descriptor or string descriptors. In this way, each Device instance can be identified unambiguously.

For a detailed function description see Microsoft Platform SDK documentation.

Sample of Device enumeration:

```
#include <windows.h>
#include <dbt.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <setupapi.h>
#include "motioctl.h"

const GUID _GuidMotUSB = GUID_CLASS_MOTUSB;

HDEVINFO USBGetDeviceList()
{
    HDEVINFO devInfo;
```

```
    devInfo = SetupDiGetClassDevs(
        (LPGUID)&_GuidMotUSB,  // LPGUID ClassGuid,
        NULL,                  // PCTSTR Enumerator,
        NULL,                  // HWND hwndParent,
        DIGCF_DEVICEINTERFACE | DIGCF_PRESENT // DWORD Flags
        );

    return ( devInfo != INVALID_HANDLE_VALUE) ? devInfo : NULL;
}

int main(int argc, char* argv[])
{
    HDEVINFO hDevInfo;
    SP_DEVINFO_DATA DeviceInfoData;
    DWORD i;

    hDevInfo = USBGetDeviceList();

    if (hDevInfo == INVALID_HANDLE_VALUE)  {
        // Insert error handling here.
        return 1;
    }

    // Enumerate through all Devices in Set
    DeviceInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
    for (i=0;SetupDiEnumDeviceInfo(hDevInfo,i,&DeviceInfoData);i++)
    {
        DWORD DataT;
        LPTSTR buffer = NULL;
        DWORD buffersize = 0;

        while (!SetupDiGetDeviceRegistryProperty(hDevInfo,
                                                 &DeviceInfoData,
                                                 SPDRP_DEVICEDESC,
                                                 &DataT,
                                                 (PBYTE)buffer,
                                                 buffersize,
                                                 &buffersize))  {
            if (GetLastError() == ERROR_INSUFFICIENT_BUFFER) {
                // Change the buffer size.
                if (buffer) LocalFree(buffer);
                buffer = (LPTSTR) LocalAlloc(LPTR,buffersize);
            }
            else {
                // Insert error handling here.
                break;
            }
        }

        printf("Result:[%s]\n",buffer);
        if (buffer) LocalFree(buffer);
    }

    if ( GetLastError()!=NO_ERROR &&
         GetLastError()!=ERROR_NO_MORE_ITEMS )  {
```

```
        // Insert error handling here.
        return 1;
    }
    //  Cleanup
    SetupDiDestroyDeviceInfoList(hDevInfo);
    return 0;
}
```

2.4.4. Establishing Connection To Device.

The setup API function *SetupDiGetDeviceInterfaceDetail* application finds the Device name in the **DevicePath** member of the **SP_DEVICE_INTERFACE_DETAIL_DATA** structure parameter. Having this name the Client application can open a handle to the Device object using the *CreateFile* Win32 API function.

After the application has received one or more handles for the Device, operations can be performed on the Device by using a handle. If there is more than one handle to the same Device, it makes no difference which handle is used in order to perform a certain operation. All handles that are associated with the same Device behave in the same manner.

Sample of establishing a Device connection function:

```
HANDLE
USBOpenDevice(
    HDEVINFO devList,
    int      devNum
  )
{
    BOOL                                bOK;
    DWORD                               len;
    DWORD                               Status;
    SP_DEVICE_INTERFACE_DETAIL_DATA *InterfaceData;
    SP_DEVICE_INTERFACE_DATA         DevData = {0};
    HANDLE                                  hDevice = INVALID_HANDLE_VALUE;

    DevData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
    bOK = SetupDiEnumDeviceInterfaces(devList, NULL,
              (LPGUID)&_GuidMotUSB, devNum, &DevData );
    if ( !bOK ) {
        return INVALID_HANDLE_VALUE;
    }

    // get length of the detailed information, allocate buffer
    SetupDiGetDeviceInterfaceDetail(devList, &DevData,
                                    NULL, 0, &len, NULL);
    InterfaceData = (SP_DEVICE_INTERFACE_DETAIL_DATA*) calloc(1, len);
    if ( !InterfaceData )
        return INVALID_HANDLE_VALUE;

    // now get the detailed Device information
```

```
    InterfaceData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
    bOK = SetupDiGetDeviceInterfaceDetail(devList, &DevData,
InterfaceData, len, &len, NULL);
    if ( !bOK ) {
        free( InterfaceData );
        return INVALID_HANDLE_VALUE;
    }

    hDevice = CreateFile(
            InterfaceData->DevicePath,
            GENERIC_READ | GENERIC_WRITE,       // access mode
            FILE_SHARE_WRITE | FILE_SHARE_READ, // share mode
            NULL,                               // security desc.
            OPEN_EXISTING,                      // how to create
            NULL,                                     // file attributes
            NULL                                // template file
            );

    return ( hDevice );
}
```

To close a connection handle on a particular Device object use `CloseHandle` Win32 API function, specifying the opened handle for the Device object in question.

See the Microsoft Platform SDK documentation for further information.

2.4.5. Device Object Functions.

The Device object represents a physical Device. It provides Default Control Pipe transfers, pipe connections, and some system control (Power Management, PnP, etc.). Some Win32 API functions should be used to interact with the MOTUSB Device object.

Table 2.2 Win32 API operations list for a Device object.

Function Name	Description
CreateFile	Opens handle to Device object
CloseHandle	Close handle to Device object
DeviceIoControl	Performs requests on Device object

All operations involving Device object handles can be requested through the DeviceIoControl function, while CreateFile and CloseHandle functions are used for Device object connections only. Note that no data flow operations via ReadFile and WriteFile can be performed. All data transmission to the Default Control Pipe requires DeviceIoControl usage.

The following tasks can be performed using MOTUSB Device object:

- Descriptor retrieval
- Configuration control
- Setting / Clear Feature for specified recipient

- Getting status of specified recipient
- Device locking / unlocking
- Interface Alternate Setting control
- Sending Class or Vendor Requests
- Device replugging emulation
- Device resetting

For a full description of Device requests, refer to the Programming Interface section.

2.5. Pipe Object.

Pipe objects provide the ability to perform data flow transactions through the pipes on the Device. Each pipe object points to a particular Device endpoint. The USB 1.1 specification defines control, bulk, interrupt and isochronous endpoint types. The bulk and isochronous endpoints specify the data flow direction from Device to PC, or from PC to Device. A MOTUSB pipe object can be used for any endpoint type except for control. The interface with the MOTUSB endpoint object is the same for any endpoint type.

Once the Device becomes configured, the Client application can open handles to pipe objects. Each interface configuration on the Device defines a particular set of endpoints through which data transmission can be performed. So only handles for pipes supported by an active configuration, and interfaces configured within it can be obtained. No pipe handles can be valid or opened on an unconfigured Device. Note that 'Set Configuration' and 'Set Interface' requests will fail if pipe connections to a Device exist.

2.5.1. Opening Connection To Pipe.

MOTUSB represents pipe object connections as for the Device objects, but with a somewhat different naming scheme. A pipe object connection can be created independently of a Device object connection. So in order to establish a connection to a pipe object, the programmer may use a procedure similar to the one described above, but specifying a different file name.

NOTE: As mentioned above USDI monopolizes endpoint #0, so no pipe connection to this endpoint can be established. MOTUSB Device objects expose functionality that can be applied to this endpoint.

The file name format for pipe objects is as follows:
<***Device instance name***> \ <***decimal endpoint address***>, where

Device instance name is string obtained from the `SetupDiGetDeviceInterfaceDetail Setup API` call in **DevicePath** member;

decimal endpoint address is an endpoint address for which a pipe should be open.

Assuming that the Device instance name returned from the *SetupDiGetDeviceInterfaceDetail* calls are the following:

\\?\USB#Vid_abcd&Pid_1234#5&e752ac&0&1#{239D60C9-CCAF-11d5-AC21-204C4F4F5020}

Then for the endpoint address 0x81, the user application would call *CreateFile* with the following file name input:

\\?\USB#Vid_abcd&Pid_1234#5&e752ac&0&1#{239D60C9-CCAF-11d5-AC21-204C4F4F5020}\129

In addition the Client application must link pipe objects to a Device object. This guarantees that *ReadFile / WriteFile* to pipes that belong to the same Device handle, will not be blocked when this Device handle is locked by the Device. So pipe linking is a mandatory condition to ensure pipe objects connection. The following sample illustrates pipe linking (assumes that Device *hDevice* is already opened, and pipe handle *hPipe* is also open):

```
USB_HANDLE_INFO handleInfo; // Kernel-mode handle to pipe object

// Get the opened pipe kernel-mode handle
DeviceIoControl (hPipe, IOCTL_USB_GET_HANDLE,  NULL, NULL,
                 &handleInfo,  sizeof(USB_HANDLE_INFO),  NULL);


// Link pipe handle hPipe to Device hDevice
DeviceIoControl     (hDevice,     IOCTL_USB_LINK_PIPE,     &handleInfo,
sizeof(USB_HANDLE_INFO),  NULL,  0,  NULL);
```

2.5.2. Pipe Object Functions.

The pipe object represents a physical channel on the Device through which data flow transactions can be performed. The *ReadFile* and *WriteFile* functions are responsible for requests for data transactions on an opened pipe handle. The *CancelIO* routine should be used to abort all outstanding transactions on a pipe object. *CreateFile* and *CloseHandle* are used for Device object connection only.

Table 2.3 Win32 API operations list for a pipe object.

Function Name	Description
CreateFile	Creates pipe object
CloseHandle	Closes pipe object handle
WriteFile, WriteFileEx	Performs data transmission to Device. Used on bulk or isochronous OUT pipes.
ReadFile, ReadFileEx	Performs data transmission from Device. Used on bulk IN, interrupt and isochronous IN pipes.
CancelIO	Cancels all pending input and output operations that were issued by the calling thread for the specified pipe handle.
DeviceIoControl	Performs requests on pipe object.

2.6. Attaching and Removing Notifications.

The Microsoft Windows operating system provides service routines for attaching a PnP Device or removing handling. Several API functions can be found in the *dbt.h* header

file in the Microsoft SDK. In order to use notifications, the Client application should register the automatic *RegisterDeviceNotification* API function. The caller can be notified by a window handle. The notification transforms to a WM_DEVICECHANGE window message, where the **lParam** parameter points to the buffer with DEV_BROADCAST_DEVICEINTERFACE structure, from which the Client application can extract the required fields concerning notification.

To be notified about Device attachment or removing events, the caller must specify the MOTUSB Interface ID to the *RegisterDeviceNotification* function.

Example:

```
#include <dbt.h>
……….

HDEVINFO RegisterDevNotify(HWND hWnd)
{
    HDEVINFO                        hDevNotify;
    DEV_BROADCAST_DEVICEINTERFACE filter;

    if (!hWnd)
       return NULL;

    ZeroMemory(&filter, sizeof(filter) );
    filter.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    filter.dbcc_Devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    filter.dbcc_classguid = g_GuidMotUSB;
    hDevNotify = RegisterDeviceNotification(
                    hWnd,
                    &filter,
                    DEVICE_NOTIFY_WINDOW_HANDLE
                 );

    return hDevNotify;
}
```

NOTE: This sample requires the Microsoft SDK for Windows 2000 to be installed. However the developer can use libraries and headers provided with Microsoft Visual C++ 6.0. In this case the “/DWINVER=0x0500” C compiler directive should be specified.

The Client application should use the *UnregisterDeviceNotification* Win32 API function when it no longer needs notification.

For further information about notifications, refer to the Microsoft Windows 2000 SDK.

3. Programming Interface.

3.1. Transfers.

The USB specification defines 4 transfer types: control, bulk, interrupt, and isochronous. The MOTUSB Device object handle is required for control transfers, while the others require a MOTUSB pipe object handle to perform data I/O operations.

3.2. Control Transfers.

All USB Devices must support the control transfer type for configuration, command, and status information. Control transfer applies to the default endpoint (zero) and monopolized by USBDI. From the programmer point of view, the control transfers are not represented as data flow, but rather “Device control commands”. All the control transactions on the bus are under the responsibility of USBDI. The USB specification defines a set of standard requests on the Default Control Endpoint. Thus, although USBDI provides a mechanism for direct access to the default endpoint, the MOTUSB Driver does not make this functionality available in the user mode, and only provides a set of standard requests.

The Client application can perform control transactions using the `DeviceIoControl` Win32 API function on the Device handle, specifying some I/O control code and parameters block. The following list of MOTUSB requests perform control transfer (for a detailed requests description see **Device requests** section):

IOCTL_USB_CLASS_OR_VENDOR_REQUEST
IOCTL_USB_CYCLE_PORT
IOCTL_USB_FEATURE_CONTROL
IOCTL_USB_GET_CONFIGURATION
IOCTL_USB_GET_DESCRIPTOR
IOCTL_USB_GET_INTERFACE
IOCTL_USB_GET_STATUS
IOCTL_USB_RESET_DEVICE
IOCTL_USB_RESET_PIPE
IOCTL_USB_SET_CONFIGURATION
IOCTL_USB_SET_INTERFACE
IOCTL_USB_UNCONFIGURE_DEVICE

3.3. Bulk and Interrupt Transfers

Bulk and interrupt transfers may be applied through the pipes opened up on the Device. For interrupt and bulk transfers the buffer size can be larger than the maximum packet size of the endpoint, as reported in the endpoint descriptor.

The MOTUSB Driver does not limit the transfer size. Each endpoint object should be configured to the preferred transfer size. This value is specified in the *MaxTransferSize* member of the *USBIO_INTERFACE_SETTING* on the *IOCTL_USB_SET_CONFIGURATION* or *IOCTL_USB_SET_INTERFACE* requests. If an application request to transfer a data buffer that is larger that the endpoint transfer size is received, the MOTUSB Driver performs a staging I/O which breaks the data buffer into parts which fit into the maximum transfer size and requests a data I/O operation for each such part.

3.3.1. Bulk Write Transfers.

A write operation on a bulk-out endpoint performs bulk data transfer from the Host (PC) to the Device. To perform bulk write transfers the Client application should first establish a connection to the pipe and to call *WriteFile* (*WriteFileEx*) Win32 API functions, specifying the pipe object handle into the *hFile* argument. The data buffer and buffer size should be specified in the corresponding *lpBuffer* and *nNumberOfBytesToWrite* arguments.

The transfer consists of packets. These packets are sent to the USB Device. If the last packet of the buffer is smaller than the maximum packet size of the endpoint, a smaller data packet is transferred. If the size of the last packet of the buffer is equal to the maximum packet size this packet is sent. No additional zero length packet is sent by the Driver. In order to send a zero length data packet, it is necessary to set the buffer length to zero and use a NULL buffer pointer.

3.3.2. Bulk and Interrupt Read Transfers.

A read operation on bulk-in or interrupt endpoints performs a bulk or interrupt data transfer from the Device to the Host (PC). To issue bulk or interrupt read transfers the Client application should first establish a connection to the pipe and to call *ReadFile* (*ReadFileEx*) Win32 API functions, to perform transfers specifying the pipe object handle in the *hFile* function argument. The data buffer and buffer size should be specified in the corresponding *lpBuffer* and *nNumberOfBytesToRead* arguments.

A read operation will be completed if the whole buffer is filled or a short packet is transmitted. A short packet is a packet that is shorter than the maximum transfer size of the endpoint. To read a data packet with a length of zero, the buffer size has to be at least one byte. A read operation with a NULL buffer will be completed with success by the system without performing a read operation on the USB. The behavior of short packets depends on the registry parameter *ShortTransferOk*. If this parameter value is set, a read operation that returns a data packet that is shorter than the maximum packet size of the endpoint is completed with success. Otherwise, every data packet from the endpoint that is smaller than the maximum packet size causes an error.

3.4. Isochronous Transfers.

Isochronous transfers can be applied through the pipes opened on the Device. The Client application should specify the special structure buffer to perform an isochronous transfer operation using `ReadFile` / `WriteFile` Win32 API functions. The buffer should consist of a fixed header and a variable length packets header and data parts. The isochronous transfer buffer format is shown in the figure below:

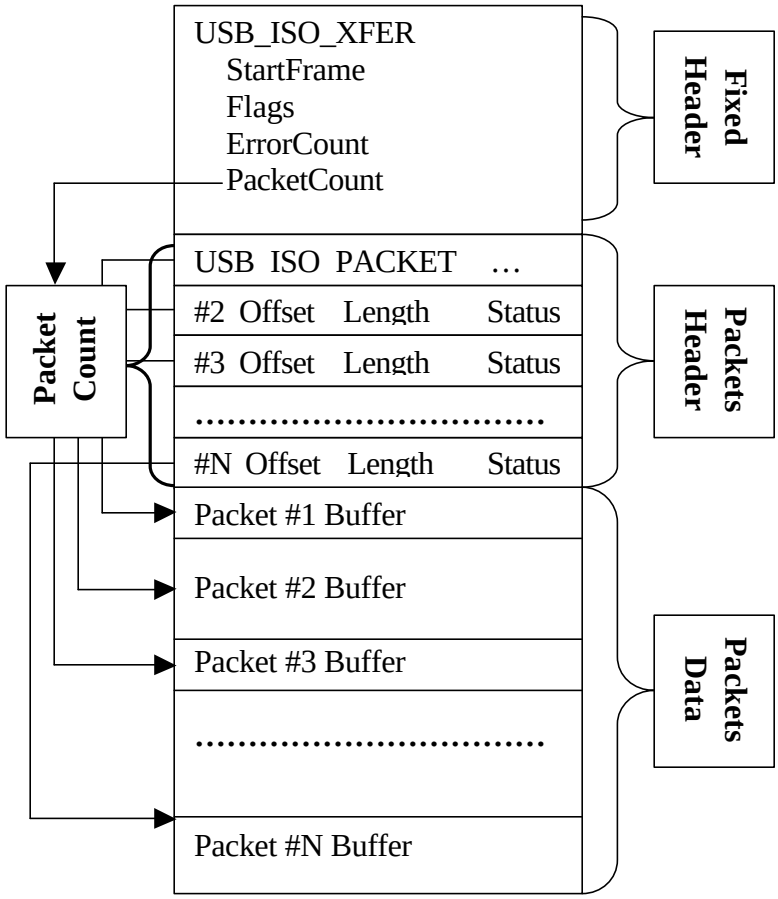

Fig 3.1 Isochronous Transfer Buffer Format.

Hence, the buffer that follows the header is divided into packets. Each packet is transmitted within one USB frame (1 ms). The size of the packet can be different in each frame. This allows support for any data rate of the isochronous data stream.

The isochronous transfer buffer is described by the `USB_ISO_XFER` structure. This structure contains an array of `USB_ISO_PACKET` structures, which provide information about packet data buffers. The `PacketCount` member of `USB_ISO_XFER` determines the packet count in a given transfer.

Each *USB_ISO_PACKET* hold *Offset* member indicates the offset of the packet data buffer (in Packets Data section), from the beginning of the buffer. The *Length* member determines the size of a packet buffer. The *Status* member indicates the I/O operation result returned when the Driver completes the whole transfer.

An isochronous transfer may not be started immediately. The Client application can specify the 11-bit *StartFrame* number in the Fixed Header part of the buffer. In this case the transfer begins in this frame. Otherwise the *USB_ISO_TRANSFER_ASAP* bit mask should be set in the *Flags* field of *USB_ISO_XFER***.** In such a case the Driver puts the request in the queue and begins transmitting as soon as possible**.** This makes the Client application capable of implementing a double buffering scheme. In this scheme the Client should request a new transfer without waiting for the previous one to complete, by specifying the *USB_ISO_TRANSFER_ASAP* flag**.**

When the MOTUSB Driver completes I/O requests the *StartFrame* member of *USB_ISO_XFER* will specify the actual frame number when a transmission was started, the *ErrorCount* member of *USB_ISO_XFER* will specify the total error count in this transfer. The *Status* field of each *USB_ISO_PACKET* will be zero, for each successfully transmitted packet, or 0x9 if an error occurred (or short packet processed).

NOTE:
No more than 255 packets can be processed within a single isochronous transfer request.

3.4.1. Isochronous Write Transfers.

A write operation on isochronous–out endpoints performs isochronous data transfer from the Host (PC) to the Device. To perform isochronous write transfers the Client application should first establish connection to the pipe, build an isochronous transfer buffer and specify it to *WriteFile* (*WriteFileEx*) Win32 API routines. The sizes of the packets have to be less than or equal to the maximum packet size of the endpoint. There must be no gaps between the data packets in the transfer buffer. The *Offset* and *Length* member of the *USB_ISO_PACKET* structures have to be initialized correctly before the transfer is started.

When the MOTUSB Driver completes write I/O requests, it changes the *Length* member of each packet according to the actual bytes which were processed in that packet. Normally this field should be zero, indicating that all packet data was sent, otherwise this field will contain the number of bytes remaining in the packet buffer as not sent.

3.4.2. Isochronous Read Transfers.

A read operation on isochronous–in endpoints performs an isochronous data transfer from the Device to the Host (PC). In order to perform isochronous read transfers the Client application should first establish a connection to the pipe, build an isochronous transfer buffer and specify it to *ReadFile* (*ReadFileEx*) Win32 API routines. The sizes of the

packets have to be less than or equal to the maximum packet size of the endpoint. There must be no gaps between the data packets in the transfer buffer. The *Offset* and *Length* member of the *USB_ISO_PACKET* structures have to be initialized correctly before the transfer is started. Note that because the size of the received packets may be less than the maximum packet size, data packets are not arranged continuously within the transfer buffer.

When the MOTUSB Driver completes read I/O requests, it changes the *Length* member of each packet according to the actual bytes which were processed in that packet. Normally this field should specify the total number of bytes read for a particular pipe.

3.4.3. Using Asynchronous I/O.

Using asynchronous (Overlapped) I/O means a thread does not need to wait for a request completion, to be able to perform some task while the Driver processes the I/O request. Overlapped I/O can be applied to any transfer type. If a Client wants to perform overlapped operations, it should open a pipe by specifying the *FILE_FLAG_OVERLAPPED* file attribute parameter to the *CreateFile* function. Then for each Win32 API call related to the Device or pipe object, the caller should specify the *OVERLAPPED* structure buffer pointer.

Overlapped I/O is very important for isochronous transfers. The major issue with these transfers is that for the most part the Client application should deliver or receive data in real time. When the application performs Read or Write request to the MOTUSB Driver, the I/O System does not guarantee that this request will be available in the frame time limit (1 millisecond normally). The only possible solution is to put several requests to the Driver, wait until some of them complete and then put further requests with the *USB_ISO_TRANSFER_ASAP* flag set.

3.5. Device Requests.

The I/O Control requests are submitted to the Driver using the Win32 function *DeviceIoControl*.

The *DeviceIoControl* function is defined as follows:

```
BOOL DeviceIoControl(
      HANDLE hDevice,                  // handle to Device object
      DWORD dwIoControlCode,           // control code of operation to perform
      LPVOID lpInBuffer,               // pointer to buffer to supply input data
      DWORD nInBufferSize,             // size of input buffer
      LPVOID lpOutBuffer,              // pointer to buffer to receive output data
      DWORD nOutBufferSize,            // size of output buffer
      LPDWORD lpBytesReturned, // pointer to variable to receive
                                       // output byte count
```

```
    LPOVERLAPPED lpOverlapped       // pointer to overlapped structure
                                    // for asynchronous operation
);
```

Refer to the Microsoft Platform SDK documentation for more information. The following sections describe the I/O Control codes that may be passed to the *DeviceIoControl* function as *dwIoControlCode* and the parameters required for *lpInBuffer*, *nInBufferSize*, *lpOutBuffer*, and *nOutBufferSize*.

Table 3.1 Device requests summary.

Request code	Description
IOCTL_USB_CLASS_OR_VENDOR_REQUEST	Performs class or vendor request
IOCTL_USB_CYCLE_PORT	Emulates port connecting, disconnecting
IOCTL_USB_FEATURE_CONTROL	Clear or sets feature on the Device
IOCTL_USB_GET_CONFIGURATION	Request the configuration from the Device
IOCTL_USB_GET_DESCRIPTOR	Request the descriptor from the Device
IOCTL_USB_GET_HANDLE	Returns the kernel mode handle
IOCTL_USB_GET_INTERFACE	Requests interface alternate setting
IOCTL_USB_GET_STATUS	Returns status for spec. recipient
IOCTL_USB_LINK_PIPE	Links pipe handle to Device
IOCTL_USB_LOCK_DEVICE	Locks the Device
IOCTL_USB_RESET_DEVICE	Resets Device
IOCTL_USB_RESET_PIPE	Resets specified pipe
IOCTL_USB_SET_INTERFACE	Selects interface setting on the Device
IOCTL_USB_SET_CONFIGURATION	Selects configuration on the Device
IOCTL_USB_UNCONFIGURE_DEVICE	Puts Device into unconfigured state

3.5.1. IOCTL_USB_CLASS_OR_VENDOR_REQUEST

Performs class or vendor request to the USB.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_CLASS_OR_VENDOR_REQUEST* structure.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_CLASS_OR_VENDOR_REQUEST* structure.

lpOutBuffer
Points to the data buffer if request has IN or OUT data stage (*nOutBufferSize != 0*). In case of IN data stage the data from the Device will be placed in this buffer, in case of OUT data stage the data in this buffer will be transmitted to the Device. Must be NULL if class or vendor request does not require a data stage.

nOutBufferSize
Specify data buffer size in bytes in case of class or vendor request with IN or OUT data stage. Must be 0 if class or vendor request does not require a data stage.

Comments:
A SETUP request appears on the default pipe (endpoint zero) of the USB Device with the given parameters. If a data phase is required an IN or OUT token appears on the bus and the successful transfer is acknowledged by an IN or OUT token with a zero length data packet from the Device. If no data phase is required an IN token appears on the bus and the Device acknowledges with a zero length data packet.

3.5.2. IOCTL_USB_CYCLE_PORT

The operation requests Device re-enumeration.

DeviceIoControl parameters:

lpInBuffer
Not used with the operation. Must be NULL.

nInBufferSize
Not used with the operation. Must be 0.

lpOutBuffer
Not used with the operation. Must be NULL.

nOutBufferSize
Not used with the operation. Must be 0.

Comments:
This request has the same effect as disconnecting and connecting a Device to/from the port. During this operation the MOTUSB Driver should be unloaded and loaded again by USBDI.

When the USBI unloads the Driver all Device and pipe handles became invalid. The Client application receives a PnP notification about the Device being removed and should close all handles to that Device.

During Device re-enumeration the following requests appear on the bus:

- Device Reset
- GET_DEVICE_DESCRIPTOR
- Device Reset
- SET_ADDRESS
- GET_DEVICE_DESCRIPTOR
- GET_CONFIGURATION_DESCRIPTOR

NOTE: Additional requests can appear depending on the descriptors for the Device.

After the re-enumeration process, the operating system loads the MOTUSB Driver again. The Client software receives a PnP notification about the Device being attached and can reopen the required handles. This request does not work if the system-provided multi-interface Driver is used. This Driver expects that all functional Device Drivers to send a `CYCLE_PORT` request within 5 seconds.

3.5.3. IOCTL_USB_FEATURE_CONTROL

Requests set or clear specified feature.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_FEATURE_REQUEST* structure. The buffer must be completely filled by the caller to specify request parameters.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of the *USB_FEATURE_REQUEST* structure.

lpOutBuffer
No output information will be returned. Must be NULL.

nOutBufferSize
No output information will be returned Must be 0.

Comments:
This request clears or sets a specified feature to the recipient. *CLEAR_FEATURE* or *SET_FEATURE* request appears on the bus depending upon the *bClear* flag of the *USB_FEATURE_REQUEST* input request.

3.5.4. IOCTL_USB_GET_CONFIGURATION

Requests current Device configuration value.

DeviceIoControl parameters:

lpInBuffer
No input information specified. Must be NULL.

nInBufferSize
No input information specified. Must be 0.

lpOutBuffer
Driver returns a current configuration value into *`USB_GET_CONFIGURATION_REQUEST`* structure.

nOutBufferSize
Specifies size of output buffer. Must be equal to the size of the *`USB_GET_CONFIGURATION_REQUEST`* structure.

Comments:
The *`bConfigurationValue`* member of the descriptor of the current configuration is returned in *`bConfigValue`* of the *`USB_GET_CONFIGURATION_REQUEST`* structure. A value of zero returned, should be considered as an unconfigured Device state. Within this request no action on the bus occurred. The MOTUSB Driver maintains an internal variable to track the active configuration index, and change it along with changing configuration requests.

3.5.5. IOCTL_USB_GET_DESCRIPTOR

Requests specified descriptor from Device.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_DESC_REQUEST* structure. The buffer must be completely filled by the caller to specify requested descriptor parameters.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_DESC_REQUEST* structure.

lpOutBuffer
Pointer to the descriptor buffer. The type of this buffer varies depending of requested descriptor type specified in *lpInBuffer* (*DescriptorType* member of *USB_DESC_REQUEST* structure**).**

Description by Desciptor type:

- for Device descriptor, Driver returns a *USB_DEVICE_DESCRIPTOR* structure.
- for String descriptor, Driver returns the string descriptor in a *USB_STRING_DESCRIPTOR* structure. The string itself is found in the variable-length *bString* member of the string descriptor.
- for Configuration Descriptor, the Driver returns the configuration descriptor in a *USB_CONFIGURATION_DESCRIPTOR* structure, followed by the interface and endpoint descriptors for that configuration. The Driver can access the interface and endpoint descriptors as *USB_INTERFACE_DESCRIPTOR*, and *USB_ENDPOINT_DESCRIPTOR* structures. The Driver also returns any class-specific or Device-specific descriptors.
- for Endpoint Descriptor, Driver returns a *USB_ENDPOINT_DESCRIPTOR* structure for requested endpoint.
- for Interface Descriptor, Driver returns a *USB_INTERFACE_DESCRIPTOR* structure for requested interface.

This buffer is completely filled by the Driver and specifies the requested descriptor information if the request was successful.

nOutBufferSize
Specifies output buffer size in bytes. For configuration descriptor this member must be equal to the size of the *USB_CONFIGURATION_DESCRIPTOR* or greater (if the caller also

inquires of other descriptors for this configuration). For any other descriptor type this member must be equal to the size of the corresponding *lpOutBuffer* structure (see *lpOutBuffer* description).

Comments:

For all descriptors except the string descriptor no action on the bus occurs. They are cached after the Device object is created. The request for Device, configuration and string descriptors can be performed on an unconfigured Device in order to retrieve information for further configurations and alternate interface setting selection.

To be able to request interface or endpoint descriptors, the Device must be configured and the current configuration index must be specified in the *ConfigIndex* member of *USB_DESC_REQUEST* structure, otherwise the request returns an error. When the configuration descriptor and other descriptors for that configuration are acquired in a single request, the size of the output buffer should be a multiple of the packet size of the default pipe.

For *USB_DESC_REQUEST* structure members refer to the *USB_DESC_REQUEST* structure description.

3.5.6. IOCTL_USB_GET_HANDLE

Requests kernel mode handle by user mode handle

DeviceIoControl parameters:

lpInBuffer
None.

nInBufferSize
None.

lpOutBuffer
Points to the *USB_HANDLE_INFO* structure buffer.

nOutBufferSize
Must be equal to size of *USB_HANDLE_INFO* structure.

Comments:
The Client application should use this request for pipe linking. The request should appear on a pipe object handle and should return a kernel mode pipe object for linking.

3.5.7. IOCTL_USB_GET_INTERFACE

Requests specified interface alternate setting.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_INTERFACE_SETTING* structure.

nInBufferSize
Specify input buffer size in bytes. Must be equal to the size of *USB_INTERFACE_SETTING* structure.

lpOutBuffer
Pointer to the buffer containing *USB_INTERFACE_SETTING* structure.

nOutBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_INTERFACE_SETTING* structure.

Comments:
The *GET_INTERFACE* request appears on the bus. The *InterfaceIndex* member of the input structure should specify the interface descriptor index within the selected configuration for which the request is issued. On successful completion, the Driver fills the *AltSettings* member of this structure with the current alternate setting for the interface The pointers to *lpInBuffer* and *lpOutBuffer* may refer to the same buffer.

3.5.8. IOCTL_USB_GET_STATUS

Requests status from the specified recipient.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_STATUS_REQUEST* structure. The buffer must be completely filled by the caller to specify request parameters.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_STATUS_REQUEST* structure.

lpOutBuffer
Pointer to the buffer containing *USB_STATUS_REQUEST* structure.

nOutBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_STATUS_REQUEST* structure.

Comments:
This request appears as a *GET_STATUS* request on the bus. The Client application must specify the recipient in a *Target* member of the input buffer structure. If the request succeeds, the Driver returns to the recipient, the status in the *Status* member of the output structure buffer. The pointers to *lpInBuffer* and *lpOutBuffer* may refer to the same buffer.

3.5.9. IOCTL_USB_LINK_PIPE

Requests to link a pipe object handle to a Device by the given pipe object handle

DeviceIoControl parameters:

lpInBuffer
Pointer to *USB_HANDLE_INFO* structure buffer. The buffer data contents can be obtained from *IOCTL_USB_GET_HANDLE* request.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_HANDLE_INFO* structure.

lpOutBuffer
Not applicable. Should be zero.

nOutBufferSize
Not applicable. Should be zero.

Comments:
The Client application should perform the requests upon the establishment of a connection to a pipe object. The request informs the Device object that the opened handle belongs to that Device object. The Client uses this request after *IOCTL_USB_GET_HANDLE* on the opened pipe handle, resulting in the kernel mode pipe handle in *USB_HANDLE_INFO* structure buffer.

3.5.10. IOCTL_USB_LOCK_DEVICE

Locks access to the Device by a specified Device handle.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer with *USB_LOCK_REQUEST* structure.

nInBufferSize
Must be equal to size of *USB_LOCK_REQUEST* structure.

lpOutBuffer
Not used with the operation. Must be NULL.

nOutBufferSize
Not used with the operation. Must be 0.

Comments:
By using this operation, the Client application can lock access to a particular Device, preventing access by other Clients. This function locks the Device by means of the Device handle. The Device handle specified in the request then becomes a master handle, so that a request from any other Device handle will be blocked or returned with error. Only access to those operations that change Device state or perform data transfers will be blocked.

The request blocks the following operations on the Device:

IOCTL_USB_RESET_DEVICE
IOCTL_USB_UNCONFIGURE_DEVICE
IOCTL_USB_FEATURE_CONTROL
IOCTL_USB_CLASS_OR_VENDOR_REQUEST
IOCTL_USB_CYCLE_PORT
IOCTL_USB_RESET_PIPE
IOCTL_USB_SET_CONFIGURATION
IOCTL_USB_SET_INTERFACE

ReadFile or *WriteFile* requests to pipe objects linked to different Device objects, will be blocked or returned with error. The operation should be used when the Host software allows different threads (processes) to share a single Device. In this case the request is very useful to synchronize Device request transactions, for different Device and pipe handle holders.

The Driver tracks the Device lock count, so that the caller must provide the same count of unlock operations as for lock, until *USB_TOTAL_UNLOCK* flag is specified.

3.5.11. IOCTL_USB_RESET_DEVICE

Resets Device and parent port objects.

DeviceIoControl parameters:

lpInBuffer
Not used with the operation. Must be NULL.

nInBufferSize
Not used with the operation. Must be 0.

lpOutBuffer
Not used with the operation. Must be NULL.

nOutBufferSize
Not used with the operation. Must be 0.

Comments:
This request sends a USB Reset over the bus. As a result of this all pending transactions on the bus should be aborted. This request causes all of the status and configuration values associated with endpoints in the affected interfaces, to be set to their default values. After resetting the Device, the OS selects the active configuration and the interfaces within it, so that the Device remains configured. All handles to Device and pipe objects remain valid. This request should not appear on any unconfigured Device.

3.5.12. IOCTL_USB_RESET_PIPE

Request clears error condition on a pipe.

DeviceIoControl parameters:

lpInBuffer
Not used with the operation. Must be NULL.

nInBufferSize
Not used with the operation. Must be 0.

lpOutBuffer
Not used with the operation. Must be NULL.

nOutBufferSize
Not used with the operation. Must be 0.

Comments:
The Client should use this request if an error occurs while transferring data to or from a pipe. The Driver halts the pipe and returns an error code. No further transfers can be performed while the pipe is halted. This request causes a stall condition on an endpoint to be cleared (except for isochronous pipes). In addition the USB Host controller will be reinitialized.

3.5.13. IOCTL_USB_SET_CONFIGURATION

Select specified configuration for a Device.

DeviceIoControl parameters:

lpInBuffer
Pointer to the buffer containing *USB_SET_CONFIGURATION_REQUEST* structure. The buffer must be completely filled by caller.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_SET_CONFIGURATION_REQUEST* structure .

lpOutBuffer
No output information will be returned. Must be NULL.

nOutBufferSize
No output information will be returned Must be 0.

Comments:
Within this request, the *SET_CONFIGURATION* request appears on the bus. Only the configuration contained in descriptors can be used. This request can be used to configure multiple interface Devices in a single call. Additionally, the caller can specify only the set of interfaces that will be configured for a selected configuration. To invoke this request, the Device must be in the unconfigured state. This request causes all of the status and configuration values associated with endpoints in the affected interfaces, to be set to their default values. Note, that to invoke this requests no pipe connections should be open on a Device.

3.5.14. IOCTL_USB_SET_INTERFACE

Selects interface alternate setting and transfer size

DeviceIoControl parameters:

lpInBuffer
Pointer to the *USB_INTERFACE_SETTING* structure buffer. The buffer must be completely filled by caller.

nInBufferSize
Specify input buffer size in bytes. Must be equal to size of *USB_INTERFACE_SETTING* structure.

lpOutBuffer
No output information will be returned. Must be NULL.

nOutBufferSize
No output information will be returned Must be 0.

Comments:
The *SET_INTERFACE* request appears on the USB. This request ensures that all pipes pending requests on the bus will be aborted. The pipe objects for a specified alternate setting will be created and will be got ready to open. If an invalid alternate is setting specified, the Driver generates an error. The previous configuration becomes invalid and the Client should use the Set Configuration or Set Interface calls again. Note, that to invoke this request, no pipe connections should be open on a Device.

3.5.15. IOCTL_USB_UNCONFIGURE_DEVICE

This operation requests to put Device into unconfigured state.

DeviceIoControl parameters:

lpInBuffer
Not used with the operation. Must be NULL.

nInBufferSize
Not used with the operation. Must be 0.

lpOutBuffer
Not used with the operation. Must be NULL.

nOutBufferSize
Not used with the operation. Must be 0.

Comments:
The Device will be treated as unconfigured, and only the following set of requests can then be applied:

IOCTL_USB_GET_DESCRIPTOR
IOCTL_USB_GET_CONFIGURATION
IOCTL_USB_SET_CONFIGURATION
IOCTL_USB_CYCLE_PORT.

Establishing a connection to a pipe object is not permitted while the Device is unconfigured.

After this operation the *IOCTL_USB_GET_CONFIGURATION* request should return a zero configuration value. The Client software developer should rarely use this operation, because some issues exist in operating system while working in this state. However this operation can be useful for new Device testing. In addition the Client software should use this request before setting a different configuration on the already configured Device.

3.6. Structures.

3.6.1. USB_CLASS_OR_VENDOR_REQUEST

Definition:

```
typedef struct {
    REQUEST_TARGET          Target;
    UCHAR                   Type;
    UCHAR                   ResBits;
    UCHAR                   Request;
    USHORT                  Value;
} USB_CLASS_OR_VENDOR_REQUEST, *PUSB_CLASS_OR_VENDOR_REQUEST;
```

Members:

Target
Request recipient defined by *REQUEST_TARGET* type.

Type
Specifies the type and direction of request.
Direction can be specified by *ORing* with *USB_REQUEST_IN_MASK* constant (defined in *motioctl.h*) for IN – class or vendor requests (Device should return data). If this mask is not applied, the Driver performs OUT – class or vendor request (Device returns no data). The request target must can be one of values defined in 3.8.3.

ResBits
Specifies a value, from 4 to 31 inclusive, that becomes part of the request type code in the USB-defined setup packet. This value is defined by the USB spec. for a class request or the vendor for a vendor request.

Request
Specifies the class or vendor-defined request code for the Device, interface, endpoint, or other Device-defined target.

Value
Is a value, specific to a request, that becomes part of the USB-defined setup packet for the target. This value is defined by the creator of the code used in the request. Check Device class specification for this value.

Comments:

This structure is used by *IOCTL_USB_CLASS_OR_VENDOR_REQUEST* Device request.

3.6.2. USB_DESC_REQUEST

Definition:

```
typedef struct _USB_DESC_REQUEST {
   UCHAR DescriptorType;
   union {
      struct {
         char   ConfigIndex;
         char   InterfaceIndex;
         char   AltSetting;
         char   EndpointIndex;
      };
      struct {
         USHORT  LanguageId;
         char          Index;
      };
   };
} USB_DESC_REQUEST, *PUSB_DESC_REQUEST;
```

Members:

DescriptorType
One of descriptor types. For possible value see table 3.4 in constants section

Index
Used for string descriptors only. Specifies string index. The language table can be obtained with zero index.

LanguageId
Used for string descriptors only. Specifies the language ID of the descriptor to be retrieved.

ConfigIndex
This member is used for configuration, interface or endpoint descriptor request. Specifies index of the configuration descriptor for which the required descriptor is requested. For a configured Device, a value of –1 implies a request descriptor from the current configuration.

InterfaceIndex
This member is used for an interface or endpoint descriptor request. Specifies the index of the interface descriptor in a selected configuration. If the endpoint descriptor requests the value of –1, this means it is necessary to lookup the endpoint of the descriptor by means of the endpoint address, among all the interfaces configured.

AltSetting
This member is used for an interface or endpoint descriptor request. It specifies an interface alternate setting for which a descriptor was requested.

EndpointIndex
This member is used for an endpoint descriptor request. It specifies an endpoint index in the interface , specified by *InterfaceIndex*. In the case of *InterfaceIndex* = –1 the Client application should put an endpoint address in this member.

Comments:
This structure is used by *IOCTL_USB_GET_DESCRIPTOR* and *IOCTL_USB_SET_DESCRIPTOR* requests.

3.6.3. USB_FEATURE_REQUEST

Definition:

```
typedef struct {
    REQUEST_TARGET Target;
    UCHAR          FeatureSelector;
    BOOLEAN        bClear;
} USB_FEATURE_REQUEST, *PUSB_FEATURE_REQUEST;
```

Members:

Target
One of the request recipients, defined by REQUEST_TARGET enumeration.

FeatureSelector
Specifies feature selector.

bClear
Boolean flag indicating, what feature operation the driver must execute. A TRUE value indicates clearing the feature, a FALSE indicates setting the feature.

Comments:
This structure is used by *IOCTL_USB_FEATURE_CONTROL* request.

3.6.4. USB_GET_CONFIGURATION_REQUEST

Definition:

```
typedef struct _USB_GET_CONFIGURATION_REQUEST {
    UCHAR    bConfigValue;
} USB_GET_CONFIGURATION_REQUEST;
```

Members:

bConfigValue

Specifies current configuration value. This value is equal to *`bConfigurationValue`* member of the configuration descriptor for active configuration. If Device in unconfigured state driver returns zero.

Comments:

This structure is used by *`IOCTL_USB_GET_CONFIGURATION`* request.

3.6.5. USB_HANDLE_INFO

Definition:

```
typedef struct _USB_HANDLE_INFO {
    PVOID   ObjectHandle;
} USB_HANDLE_INFO;
```

Members:

ObjectHandle
Kernel mode pipe object handle.

Comments:

The structure used by *IOCTL_USB_LINK_PIPE* and *IOCTL_USB_GET_HANDLE* requests. Specifies a kernel mode pipe object handle for a pipe to Device linking.

3.6.6. USB_INTERFACE_SETTING

Definition:

```
typedef struct _USB_INTERFACE_SETTING {
   USHORT       InterfaceIndex;
   USHORT       AltSetting;
   ULONG        MaxTransferSize;
} USB_INTERFACE_SETTING, *PUSB_INTERFACE_SETTING;
```

Members:

InterfaceIndex
Specifies zero - base interface descriptor index within configuration. If using this structure on configured Device this value specifies index in interfaces configured within the configuration.

AltSetting
Specifies alternate settings value for given interface.

MaxTransferSize
Specifies maximum transfer size for all the pipes of an interface. Maximum transfer size depends on the Device. If Client application performs a transfer with lager size than the maximum transfer size, the driver will break this request into smaller pieces, conforming to this value. The value of –1 is assumed to be: take default maximum transfer size for registry settings.

Comments:

This structure is used by *IOCTL_USB_SET_CONFIGURATION*, *IOCTL_USB_SET_INTERFACE* and *IOCTL_USB_GET_INTERFACE* request.

3.6.7. USB_ISO_PACKET

Definition:

```
typedef struct  _USB_ISO_PACKET {
   ULONG   Offset;
   ULONG   Length;
   ULONG   Status;
} USB_ISO_PACKET, *PUSB_ISO_PACKET;
```

Members:

Offset
Packet buffer offset within isochronous transfer buffer pointed by USB_ISO_XFER structure.

Length
Specifies packet length in bytes. The Client application should set this value for isochronous transfers. When driver completes transfer I/O it fills this member with the number of bytes actually processed for this packet. This value should be less than or equal to endpoint packet size, defined in an endpoint descriptor of the pipe for which the I/O operation should be performed.

Status
The driver returns packet transmitting result to this member. Zero means successful transmission, 0x9 shows that a short packet was processed.

Comments:
This structure is used as part of an isochronous transfer request using the `USB_ISO_XFER` structure and specifies the packet header information. The `Offset` member contains the offset from the beginning of the **USB_ISO_XFER** buffer.

To determine the isochronous transfer buffer size by a given Packet Count and Packet Size use `ISO_XFER_BUF_SIZE(PacketCount, PacketSize)` macro. The result value will include header, packet headers and packet data buffers sizes

To get pointer to the isochronous packet data buffer by given transfer buffer and packet index relative to the transfer buffer use the `PACKET_BUFFER(xfer, index)` macro.

3.6.8. USB_ISO_XFER

Definition:

```
typedef struct _USB_ISO_XFER {
    USHORT          StartFrame;
    ULONG           Flags;
    ULONG           ErrorCount;
    ULONG           PacketCount;
    USB_ISO_PACKET  Packets[1];
} USB_ISO_XFER, *PUSB_ISO_XFER;
```

Members:

StartFrame

Specifies the frame number that the transfer should begin on. This variable must lie within the 2048 frames. If the *USB_ISO_TRANSFER_ASAP* is set in *Flags*, this member contains the frame number that the transfer began on, when the request was returned by the Host controller driver. Otherwise, this member must contain the frame number that this transfer will begin on.

Flags

Specifies zero or a *USB_ISO_TRANSFER_ASAP* flag. If equal to *USB_ISO_TRANSFER_ASAP* the transfer is set to begin on the next frame, if there were no transfers submitted to the pipe since the pipe was opened or last reset. Otherwise, the transfer will begin on the first frame following all currently queued requests for the pipe. The actual frame that the transfer begins on will be adjusted for bus latency by the driver.

ErrorCount;

Contains the number of packets that completed with an error condition on return from the driver.

PacketCount

Specifies the number of packets described by the boundless array member *Packets*. This value can be from 1 to 255.

Packets

Contains a variable-length array of *USB_ISO_PACKET* structures that describe each transfer packet of the isochronous transfer

Comments:

This specifies the buffer form for isochronous transfers. If IsoPacket has *n* entries, the Host controller transfers use *n* frames to transfer data, transferring *Packets [i].Length* bytes beginning, with an offset of *Packets[i].Offset*.

3.6.9. USB_LOCK_REQUEST

Definition:

```
typedef struct _USB_LOCK_REQUEST {
   ULONG   Flags;
} USB_LOCK_REQUEST, *PUSB_LOCK_REQUEST;
```

Members:

Flags

Specifies zero, one, or a combination of the following flags: *USB_LOCK_DEVICE* Acquires Device lock. If there are no other handles to the same Device, the object maintains lock and the Device object handle specified, becomes the owner of the lock. Otherwise the request will be put in the queue by the Device lock queue and processed later. If the application performs this request synchronously, the calling thread will be blocked until the request is processed.

USB_TRY_LOCK_DEVICE
The request is the same as with specifying the USB_LOCK_DEVICE flag. However the driver will not block the thread and put this request in the queue if the Device is already locked, instead it returns with an error immediately.

USB_UNLOCK_DEVICE
Releases single Device lock by a given Device handle.

USB_TOTAL_UNLOCK
Releases all Device locks by a given Device handle.

Comments:

The structure is used by the *IOCTL_USB_LOCK_DEVICE* requests. It is possible to lock a Device several times with the same handle. The Client should also release locks as often as it acquires them. If the Client wants to remove all locks by a particular handle, it should specify the USB_TOTAL_UNLOCK flag.

3.6.10. USB_SET_CONFIGURATION_REQUEST

Definition:

```
typedef struct _USB_SET_CONFIGURATION_REQUEST {
    USHORT                      ConfigIndex;
    LONG                        InterfaceCount;
    USB_INTERFACE_SETTING       Interfaces[USB_MAX_INTERFACE_COUNT];
} USB_SET_CONFIGURATION_REQUEST;
```

Members:

ConfigIndex
Index of configuration descriptor. Used to identify configuration.

InterfaceCount
Count of interfaces that should be configured within this configuration. If –1 is specified, all Interfaces with a configuration become configured with the zero alternate setting and default maximum transfer size.

Interfaces[USB_MAX_INTERFACE_COUNT]
The array of interface settings should be configured within this configuration. This array must contain *`InterfaceCount`* valid entries. Not applicable if *`InterfaceCount`* is equal to –1.

Comments:

This structure is used by *`IOCTL_USB_SET_CONFIGURATION`* request. In addition when *`Interfaces`* member is used (not equal to -1), it is possible to specify the maximum transfer size for each interface that was configured.

3.6.11. USB_STATUS_REQUEST

Definition:

```
typedef union _USB_STATUS_REQUEST  {
   REQUEST_TARGET       Target;
   USHORT                 Status;
} USB_STATUS_REQUEST, *PUSB_STATUS_REQUEST;
```

Members:

Target
One of the request recipients defined by REQUEST_TARGET enumeration.

Status
The status returned by the driver requested by the caller.

Comments:

This structure is used by *IOCTL_USB_GET_STATUS* request.

3.7. Types.

3.7.1. REQUEST_TARGET

Definition:

typedef USHORT REQUEST_TARGET;

Comments:
The type combines request recipient in low byte and recipient index in high byte. For the Device target this value is zero. For interface and endpoint recipients use the following macros:

ENDPOINT_TARGET(index)
INTERFACE_TARGET(index)
This macros combines one of *USBReceipients* enumeration values and index.

To parse this type use *REQUEST_TARGET_TYPE*(target) : returns one of *USBReceipients* enumeration value *REQUEST_TARGET_INDEX*(target) : returns receipient index.

3.7.2. USB_DEVICE_DESCRIPTOR

Definition:

```
typedef struct _USB_DEVICE_DESCRIPTOR {
   UCHAR   bLength;
   UCHAR   bDescriptorType;
   USHORT bcdUSB;
   UCHAR   bDeviceClass;
   UCHAR   bDeviceSubClass;
   UCHAR   bDeviceProtocol;
   UCHAR   bMaxPacketSize0;
   USHORT idVendor;
   USHORT idProduct;
   USHORT bcdDevice;
   UCHAR   iManufacturer;
   UCHAR   iProduct;
   UCHAR   iSerialNumber;
   UCHAR   bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

Comments:

The structure represents the USB1.1 Device descriptor. For member description refer to USB 1.1 specification.

3.7.3. USB_ENDPOINT_DESCRIPTOR

Definition:

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
   UCHAR   bLength;
   UCHAR   bDescriptorType;
   UCHAR   bEndpointAddress;
   UCHAR   bmAttributes;
   USHORT wMaxPacketSize;
   UCHAR   bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

Comments:

The structure represents USB1.1 endpoint descriptor. For member description refer to USB 1.1 specification.

3.7.4. USB_CONFIGURATION_DESCRIPTOR

Definition:

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
   UCHAR   bLength;
   UCHAR   bDescriptorType;
   USHORT wTotalLength;
   UCHAR   bNumInterfaces;
   UCHAR   bConfigurationValue;
   UCHAR   iConfiguration;
   UCHAR   bmAttributes;
   UCHAR   MaxPower;
}                                          USB_CONFIGURATION_DESCRIPTOR,
*PUSB_CONFIGURATION_DESCRIPTOR;
```

Comments:

The structure represents USB1.1 configuration descriptor. For member description refer to USB 1.1 specification.

3.7.5. USB_INTERFACE_DESCRIPTOR

Definition:

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

Comments:

The structure represents USB1.1 interface descriptor. For member description refer to USB 1.1 specification.

3.7.6. USB_STRING_DESCRIPTOR

Definition:

```
typedef struct _USB_STRING_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;
```

Comments:

The structure represents USB1.1 string descriptor. For member description refer to USB 1.1 specification.

3.8. Enumeration Types.

3.8.1. USBReceipients

Definition:

```
enum USBReceipients  {
   DeviceTarget = 0,
   InterfaceTarget,
   EndpointTarget,
   OtherTarget
};
```

Comments:

Request recipients. These values are used by the *REQUEST_TARGET* type.

3.8.2. LockFlags.

Definition:

```
enum LockFlags {
   USB_LOCK_DEVICE       = 1,
   USB_UNLOCK_DEVICE  = 2,
   USB_TOTAL_UNLOCK   = 4,
   USB_TRY_LOCK_DEVICE  = 8,
   USB_LOCK_MASK         = USB_LOCK_DEVICE          |
                           USB_UNLOCK_DEVICE      |
                           USB_TRY_LOCK_DEVICE  |
                           USB_TOTAL_UNLOCK;
};
```

Comments:

Flags used by ***`USB_LOCK_REQUEST`*** structure. The values specify lock type.

3.8.3. RequestsTypes.

Definition:

```
enum RequestsTypes {
   ClassRequest    = 1,
   VendorRequest
};
```

Comments:

Class or Vendor requests types used by **USB_CLASS_OR_VENDOR_REQUEST** structure. Specifies request type.

3.9. Constants.

3.9.1. MOTUSB Defined Constants.

MOTUSB defines several constant values in the *motioctl.h* header file.

The following table shows limitation constants applied to the MOTUSB Driver:

Table 3.2 Driver Limits.

Code	Value	Description
USB_MAX_CONFIG_COUNT	0x7F	Maximum configurations per Device
USB_MAX_INTERFACE_COUNT	0x7F	Maximum interfaces per configuration
USB_MAX_ENDPOINTS_COUNT	0x7F	Maximum endpoints per interface
USB_MAX_TRANSFER_SIZE	0x7FFFFFFF	Maximum transfer size
USB_MAX_ISO_PACKETS	0xFF	Maximum isochronous packet per transfer

The following table shows the flags used in the Driver programming interface:

Table 3.3 Driver flags.

Code	Value	Description
USB_REQUEST_IN_MASK	0x80	Mask for vendor or class request with IN data stage. This mask should be applied to *Type* field of Class or Vendor requests input structure
USB_ISO_TRANSFER_ASAP	0x04	Bit mask for *Flags* member of *USB_ISO_XFER* structure. The flag means isochronous transfer should be started from the first available frame. The isochronous transfer requests can be put in the queue for further I/O processing.

3.9.2. USB Specification Defined Constants.

MOTUSB **programming** interface uses the following USB 1.1 specification constants provided by DDK in usb100.h header file:

Table 3.4 Descriptor types.

Code	Value	Comments
USB_DEVICE_DESCRIPTOR_TYPE	0x01	Device Descriptor
USB_CONFIGURATION_DESCRIPTOR_TYPE	0x02	Configuration Descriptor
USB_STRING_DESCRIPTOR_TYPE	0x03	String Descriptor
USB_INTERFACE_DESCRIPTOR_TYPE	0x04	Interface Descriptor
USB_ENDPOINT_DESCRIPTOR_TYPE	0x05	Endpoint Descriptor

Table 3.5 Endpoint Types.

Code	Value	Comments
USB_ENDPOINT_TYPE_CONTROL	0x00	Control Endpoint
USB_ENDPOINT_TYPE_ISOCHRONOUS	0x01	Isochronous Endpoint
USB_ENDPOINT_TYPE_BULK	0x02	Bulk Endpoint
USB_ENDPOINT_TYPE_INTERRUPT	0x03	Interrupt Endpoint

Table 3.6 Feature Selectors.

Code	Value	USB Spec. Value
USB_FEATURE_ENDPOINT_STALL	0x00	ENDPOINT_HALT
USB_FEATURE_REMOTE_WAKEUP	0x01	DEVICE_REMOTE_WAKEUP

Table 3.7 Status Values.

Code	Value	Comments
USB_GETSTATUS_SELF_POWERED	0x01	Device is self powered
USB_GETSTATUS_REMOTE_WAKEUP_ENABLED	0x02	Device supports remote wakeup
USB_GETSTATUS_ENDPOINT_HALT	0x01	Endpoint Stall Feature Set

Table 3.8 bmAttributes of Configuration Descriptor.

Code	Value	Comments
USB_CONFIG_BUS_POWERED	0x80	Is set if this configuration is powered by the bus
USB_CONFIG_SELF_POWERED	0x40	This configuration is self-powered and does not use power from the bus
USB_CONFIG_REMOTE_WAKEUP	0x20	Is set if this configuration supports remote wakeup.

3.10. Error codes.

The Driver maps error codes returned by USBDI for MOTUSB Client applications. These errors are returned by USBDI if an error on the bus occurs. The MOTUSB diver provides only the gate between USBDI and the Client application and makes no assumptions about the values. For a detailed description refer to the Microsoft DDK Documentation.

Table 3.9 Mapped error codes.

Code	Value
USB_STATUS_CRC	0xE0100001L
USB_STATUS_BTSTUFF	0xE0100002L
USB_STATUS_DATA_TOGGLE_MISMATCH	0xE0100003L
USB_STATUS_STALL_PID	0xE0100004L
USB_STATUS_DEV_NOT_RESPONDING	0xE0100005L
USB_STATUS_PID_CHECK_FAILURE	0xE0100006L
USB_STATUS_UNEXPECTED_PID	0xE0100007L
USB_STATUS_DATA_OVERRUN	0xE0100008L
USB_STATUS_DATA_UNDERRUN	0xE0100009L
USB_STATUS_BUFFER_OVERRUN	0xE010000CL
USB_STATUS_BUFFER_UNDERRUN	0xE010000DL
USB_STATUS_NOT_ACCESSED	0xE010000FL
USB_STATUS_FIFO	0xE0100010L
USB_STATUS_ENDPOINT_HALTED	0xE0100030L
USB_STATUS_NO_MEMORY	0xE0100100L
USB_STATUS_INVALID_URB_FUNCTION	0xE0100200L
USB_STATUS_INVALID_PARAMETER	0xE0100300L
USB_STATUS_ERROR_BUSY	0xE0100400L
USB_STATUS_REQUEST_FAILED	0xE0100500L
USB_STATUS_INVALID_PIPE_HANDLE	0xE0100600L
USB_STATUS_NO_BANDWIDTH	0xE0100700L
USB_STATUS_INTERNAL_HC_ERROR	0xE0100800L
USB_STATUS_ERROR_SHORT_TRANSFER	0xE0100900L
USB_STATUS_BAD_START_FRAME	0xE0100A00L
USB_STATUS_ISOCH_REQUEST_FAILED	0xE0100B00L
USB_STATUS_FRAME_CONTROL_OWNED	0xE0100C00L
USB_STATUS_FRAME_CONTROL_NOT_OWNED	0xE0100D00L
USB_STATUS_CANCELED	0xE0110000L
USB_STATUS_CANCELING	0xE0120000L

Several error codes returned by MOTUSB are specific to the MOTUSB Driver and library.

Table 3.10 MOTUSB error codes.

Code	Description
USB_STATUS_ALREADY_CONFIGURED	Device is already configured
USB_STATUS_UNCONFIGURED	Device is unconfigured
USB_STATUS_NO_SUCH_DEVICE	The specified Device doesn't exists
USB_STATUS_DEVICE_NOT_FOUND	The specified Device not found in system
USB_STATUS_IO_PENDING	I/O operation is still in progress
USB_STATUS_NOT_SUPPORTED	Operation isn't supported by Driver
USB_STATUS_IO_TIMEOUT	Request timeout
USB_STATUS_DEVICE_REMOVED	Device was removed
USB_STATUS_PIPE_NOT_LINKED	Pipe not linked
USB_STATUS_PIPE_CONNECTED	Device cannot be reconfigured because pipe connections already exist.
USB_STATUS_DEVICE_LOCKED	Device is locked by another handle

4. MOTUSB Library.

4.1. Library Overview.

The MOTUSB library is based on the functionality of the MOTUSB Device Driver. The purpose of this library is to simplify USB development processes for user mode Client applications, that use the MOTUSB Device Driver. The library maps all functionality provided by the MOTUSB Device Driver. Developers should find it preferable to use the library programming interface than to communicate directly with the Device Driver through Win32 API.

4.2. Compiling And Linking.

Required headers:

\inc

motusb.h	- MOTUSB library programming interface
motstatus.h	- MOTUSB errors codes

Required libraries:

\lib

motusb.lib	- MOTUSB library

4.3. Handles.

The Library uses other handles than OS (HANDLE). The major problems with OS handles is that in some cases they can become invalid. A MOTUSB Client should track these cases and reopen the handles where possible. The MOTUSB library automatically supports such tracking and, moreover, MOTUSB handles never become invalid. When the Client application wants to perform an operation on a handle that turns out to be in an invalid state, the library returns a corresponding error code. The Client application does not track such cases as Device disconnection from the bus. The library closes all handles upon disconnection and reopens them if a Device with the same VendorID / ProductID connects to the bus.

The type of handle *usb_t* is common for both Device and pipe objects. However the library differs between them, and requests which apply to a Device handle, should not be used for pipe objects, and similarly pipe object handles should not be used for Device requests.

The Library does not provide a way to open a pipe object without the assistance of the Device. The Library maintains an open pipe list for each Device object. The Client application first needs to open a Device object handle using the *USBOpenDevice* routine,

then, provided the Device is configured, the application can request a pipe object to open on the Device using the *USBOpenPipe* routine.

In the MOTUSB library each Client should provide the handle as a parameter for most functions.

4.4. Error codes.

All functions in the Library except *USBGetDeviceList, USBReleaseDeviceList*, and *USBGetErrorText* return MOTUSB Driver error codes (See MOTUSB Error codes section). MOTUSB shares the same error code for a Driver and a library. For further information on "MOTUSB error codes" in the "*Returns*" statement specified, refer to the errors codes for the Driver.

4.5. Notes about overlapped I/O.

The MOTUSB library provides a way to make an overlapped I/O for the Client application. Every handle in the library is opened for overlapped I/O operation, since each handle library maintains a variable of the structure. Most functions require a variable of the OVERLAPPED structure as a parameter structure type. The caller can specify NULL to this parameter. In this case, the library will use an internal variable and blocks the calling thread until the request completes. If the caller specifies a non-zero value for this parameter, it should use the *USBWaitIO* function to determine where the actual I/O request completes. This can be done in another thread context for example, so that the main thread remains unblocked, and the Client can perform other operations while waiting for the actual I/O to complete.

4.6. Functions Descriptions.

Table 4.1 Library functions summary.

Function	Handle	Description
• **Devices enumeration**		
USBGetDeviceList	N/A	Retrieves all connected Devices for which installed
USBReleaseDeviceList	N/A	Frees Device list requested by prior function
• **Device, pipe connections**		
USBOpenDevice	Device	Establishes connection to Device object
USBCloseDevice	Device	Closes connection to Device object
USBOpenPipe	Device	Establishes connection to bulk or interrupt pipe object
USBClosePipe	Pipe	Closes connection to bulk or interrupt pipe object
• **Descriptors**		
USBGetDeviceDesc	Device	Requests Device descriptor
USBGetConfigDesc	Device	Requests configuration descriptor

USBGetInterfaceDesc	Device	Requests specified interface descriptor
USBGetEndpointDesc	Device	Requests specified endpoint descriptor
USBGetStringDesc	Device	Requests specified string descriptor
USBPipeGetDescriptor	Pipe	Requests endpoint descriptor of a pipe
• **Configuration**		
USBSetConfiguration	Device	Selects the Device configuration
USBGetConfiguration	Device	Requests the selects selected Device configuration
USBUnconfigureDevice	Device	Puts Device into unconfigured state
USBGetInterface	Device	Returns interface alternate setting
USBSetInterface	Device	Selects interface alternate setting

Function	Handle	Description
• **Device control**		
USBResetDevice	Device	Resets Device
USBSetFeature	Device	Sets feature for specified recipient
USBClearFeature	Device	Clears feature for specified recipient
USBGetStatus	Device	Retrieves status for specified recipient
USBClassOrVendorRequest	Device	Performs class or vendor (IN or OUT) requests
USBCyclePort	Device	Emulates Device replugging
• **Device locking**		
USBLockDevice	Device	Locks/Unlocks access to Device by Device handle
• **Device Notifications**		
USBRegisterDevNotify	N/A	Registers Device attaching/removing notification to the window
USBUnregisterDevNotify	N/A	Unregisters window form Device attaching/removing notification
• **Pipes I/O**		
USBResetPipe	Pipe	Stops all pending I/O for pipe and reintializes the Host controller
USBReadPipe	Pipe	Performs data transfer from Device to Host
USBWritePipe	Pipe	Performs data transfer from Host to Device
USBBuildIsoXfer	Pipe	Creates isochronous transfer buffer
• **Common**		
USBWaitIO	Pipe / Device	Waits until last pipe I/O operation completes
USBIoCtrl	Pipe / Device	Performs Device request directly
USBCancelIO	Pipe / Device	Aborts all pending I/O requests, applied by calling thread, for particular Device or pipe handle.
• **Errors**		
USBGetErrorText	N/A	Returns error text for specified error code

4.6.1. USBBuildIsoXfer

Definition:

```
DWORD USBAPI
USBBuildIsoXfer(
   IN  usb_t        Pipe,
   IN  BYTE         PacketCount,
   IN  USHORT       StartFrame,
   IN  DWORD        Flags,
   OUT PVOID        *Buffer,
   OUT ULONG        *BufferSize
   );
```

Parameters:

Pipe
Points to the opened isochronous pipe handle.

PacketCount
Specifies packet count in transfer buffer.

StartFrame
Points to variable used for overlapped I/O. Can be NULL.

Flags
Isochronous transfer flags. Can be zero or *USB_ISO_TRANSER_ASAP*.

Buffer
Points to the buffer created by this routine. Formally this points to *USB_ISO_XFER* structure header.

BufferSize
The parameter will hold the created buffer size.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some error code

Comments:
The function creates an isochonous transfer buffer. This buffer can be used by *USBReadPipe* and *USBWritePipe* functions for isochronous endpoints. It fills the packets header according to the maximum packet size for a specified endpoint. If the Client wants another packet length for some of the transfer packets it should modify the *Offset* and *Length* member of *USB_ISO_PACKET* manually. When the Client no longer needs the buffer it should release the memory the *Buffer* parameter points to, using the *free* standard library routine.

4.6.2. USBCancelIO.

The function aborts all pending I/O requests on a handle.

Definition:

```
DWORD USBAPI
USBCancelIO (
   IN usb_t                    Pipe,
   IN OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

Pipe
Points to the opened pipe handle.

byThread
Flag that is used to abort a pending I/O by calling a thread or by means of a pipe handle.

pOverlapped
Points to a variable used for overlapped. Can be NULL.

Returns:

On success, routine returns *USB_STATUS_SUCCESS*, or else some error code.

Comments:

Aborts all pending I/O requests, applied by calling a thread, for a particular Device or pipe handle.

4.6.3. USBClassOrVendorRequest

Definition:

```
DWORD USBAPI
USBClassOrVendorRequest(
   IN  usb_t                                   Device,
   IN  PUSB_CLASS_OR_VENDOR_REQUEST Request,
   IN  OUT LPVOID                              Buffer,
   IN  DWORD                                   Bufsize,
   IN  OUT OVERLAPPED                  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

Request
Points to the request parameter block

Buffer
Points to the output buffer (in case of an IN request)

Bufsize
Specifies the size of output buffer in bytes.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code.

Comments:
This function performs class or vendor request. The caller must completely fill the request parameter block of `PUSB_CLASS_OR_VENDOR_REQUEST` type. The function sends `IOCTL_USB_CLASS_OR_VENDOR_REQUEST` request to the Driver.

4.6.4. USBClearFeature

Definition:

```
DWORD USBAPI
USBClearFeature(
   IN usb_t                  Device,
   IN REQUEST_TARGET  Target,
   IN UCHAR                  Feature,
   IN OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

Target
One of request recipients, defined by *REQUEST_TARGET* type.

Feature
Specifies feature selector.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code.

Comments:
The function clears a specified feature for a specified recipient. Feature selectors should be *USB_FEATURE_ENDPOINT_STALL* or *USB_FEATURE_REMOTE_WAKEUP*. The function sends a *IOCTL_USB_FEATURE_CONTROL* request to the Driver.

4.6.5. USBCloseDevice

Definition:

```
DWORD USBAPI
USBCloseDevice(
   IN OUT usb_t* pDevice
 );
```

Parameters:

pDevice

Points to an opened Device handle. On function return set handle to NULL.

Returns:

On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code.

Comments

This function closes a Device handle acquired with the `USBOpenDevice` routine. The Client application should close each opened handle when that handle is no longer needed, or at least at the application cleanup time. Closing the Device handle also causes the closure of all linking pipe handles.

4.6.6. USBClosePipe

Definition:

```
DWORD USBAPI
USBClosePipe(
   IN  usb_t* pPipe
 );
```

Parameters:

pPipe

Points to pipe handle. On function return set handle to NULL.

Returns:

On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments:

The function closes the connection to a pipe object specified by the `pipe` handle. The pipe handle is unlinked from the Device object. All pending I/Os on this pipe will be aborted.

4.6.7. USBCyclePort

Definition:

```
DWORD USBAPI
USBCyclePort(
   IN  usb_t                    Device,
   IN  OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments
This request requests Device replugging emulation.
The function sends a `IOCTL_USB_CYCLE_PORT` request to the Driver.

4.6.8. USBGetConfigDesc

Definition:

```
DWORD USBAPI
USBGetConfigDesc(
   IN  usb_t                Device,
   IN  int                  ConfigIndex,
   OUT LPVOID               Desc,
   IN  OUT LPDWORD          Size,
   IN  OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

Desc
Points to the buffer to return configuration descriptor in.

Size
Specifies the bytes count to be returned for the configuration. This parameter must be equal to the size of the *USB_CONFIGURATION_DESCRIPTOR* or greater (if the caller acquires other descriptors for this configuration also). Also it should be a multiple of the packet size of the default pipe.

ConfigIndex
Specifies requested configuration descriptor index (zero - biased)

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
The function returns the configuration descriptor in a *USB_CONFIGURATION_DESCRIPTOR* structure, followed by the interface and endpoint descriptors for that configuration. The Driver can access the interface and endpoint descriptors as *USB_INTERFACE_DESCRIPTOR*, and *USB_ENDPOINT_DESCRIPTOR* structures. The Driver also returns any class-specific or Device-specific descriptors. The function sends a *IOCTL_USB_GET_DESCRIPTOR* request to the Driver.

4.6.9. USBGetConfiguration

Definition:

```
DWORD USBAPI
USBGetConfiguration(
   IN  usb_t                Device,
   OUT UCHAR                *ConfigIndex,
   IN  OUT OVERLAPPED       *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

ConfigIndex
Points to variable to result Device defined configuration value

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments
The function requests an active configuration index. Configuration value returns in the buffer pointed by the `ConfigIndex` parameter. A zero returned value should be considered as unconfigured Device state. The function sends a `IOCTL_USB_GET_CONFIGURATION` request to the Driver.

4.6.10. USBGetDeviceDesc

Definition:

```
DWORD USBAPI
USBGetDeviceDesc(
   IN usb_t                                Device,
   OUT PUSB_DEVICE_DESCRIPTOR  Desc,
   IN     OUT OVERLAPPED               *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

Desc
Points to the buffer for requested Device descriptor

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns ***USB_STATUS_SUCCESS***, or else some MOTUSB error code

Comments:
This function requests the Device descriptor. The caller should allocate a buffer for the ***Desc*** parameter. The function sends a ***IOCTL_USB_GET_DESCRIPTOR*** request to the Driver.

4.6.11. USBGetDeviceList

Definition:

HDEVINFO USBAPI
USBGetDeviceList**(void);**

Parameters:
None

Returns:
The function returns connected MOTUSB Device list in `HDEVINFO` system handle or NULL on any error

Comments:
Using this function, the Client application can retrieve a connected MOTUSB Devices list. This is an essential part of connected Devices enumeration. The caller should provide this handle to the `USBOpenDevice` function. When the Client opens the required Device it should release the system handle using the `USBReleaseDeviceList` routine.

4.6.12. USBGetEndpointDesc

Definition:

```
DWORD USBAPI
USBGetEndpointDesc(
   IN  usb_t                                Device,
   IN  BYTE                                 ConfigIndex,
   IN  BYTE                                 InterfaceIndex,
   IN  BYTE                                 altSetting,
   IN  BYTE                                 EndpointIndex,
   OUT PUSB_ENDPOINT_DESCRIPTOR pDescriptor,
   IN  OUT OVERLAPPED                       *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

ConfigIndex
Specifies configuration descriptor index. The value –1 means selected configuration.

InterfaceIndex
Interface descriptor index within configuration. The value –1 means it is necessary to look up the endpoint descriptor among all the configured interfaces.

AltSetting
Interface alternate setting to lookup endpoint the descriptor within.

EndpointIndex
Endpoint descriptor index within the interface. If *`InterfaceIndex`* parameter is equal to –1 this parameter should specify the endpoint address. The descriptor will be looking through all the configured interfaces.

pDesciptor
Points to the buffer for requested endpoint descriptor

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *`USB_STATUS_SUCCESS`*, or else some MOTUSB error code.

Comments:

This function returns the interface descriptor in a ***PUSB_ENDPOINT_DESCRIPTOR***. The caller should allocate a buffer pointed by the ***pDescriptor,*** large enough to store the ***PUSB_ENDPOINT_DESCRIPTOR*** structure. The endpoint descriptor requested is relative to the interface and configuration descriptors. The caller should properly specify the configuration descriptor index ***ConfigIndex***, interface descriptor index ***InterfaceIndex*** within the configuration, and the endpoint descriptor index ***ep_index*** within that interface. Alternatively by specifying –1 in ***ConfigIndex*** and ***InterfaceIndex,*** a Client can obtain the descriptor by specifying the endpoint address in the ***EndpointIndex*** parameter. The function sends a ***IOCTL_USB_GET_DESCRIPTOR*** request to the Driver.

4.6.13. USBGetErrorText

Definition:

```
LPCTSTR USBAPI
USBGetErrorText(
   IN DWORD Status
 );
```

Parameters:

Status – MOTUSB error code returned by some library routine.

Returns:

Pointer to the string with error message for specified error code. The Client should use `LocalFree` Win32 API function to free memory allocated by this function when it no longer needs this message.

Comments:

The function returns error message string for specified error code.

4.6.14. USBGetInterface

Definition:

```
DWORD USBAPI
USBGetInterface(
   IN  usb_t                  Device,
   IN  UCHAR                  InterfaceIndex,
   OUT PUCHAR                 AltSettings,
   IN  OUT OVERLAPPED   *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

InterfaceIndex
Specifies interface descriptor index within the selected configuration

AltSettings
The function returns current alternate setting to this parameter.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
The routine requests current interface alternate setting. Performs *IOCTL_USB_GET_INTERFACE* request to the Driver.

4.6.15. USBGetInterfaceDesc

Definition:

```
DWORD USBAPI
USBGetInterfaceDesc(
   IN  usb_t                          Device,
   IN  BYTE                           ConfigIndex,
   IN  BYTE                           InterfaceIndex,
   IN  BYTE                           AltSetting,
   IN  PUSB_INTERFACE_DESCRIPTOR   pDescriptor,
   IN  OVERLAPPED                  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

ConfigIndex
Specifies configuration descriptor index to lookup interface descriptor. If this value equals –1 the interfaces descriptor will be sought in the selected configuration.

InterfaceIndex
Requested interface descriptor index within the configuration.

AltSetting
Specifies interface descriptor alternate setting.

pDescriptor
Points to the buffer for requested interface descriptor.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
This function returns the interface descriptor in a *PUSB_INTERFACE_DESCRIPTOR*. The caller should allocate a buffer pointed by the *pDescriptor*, large enough to store the *PUSB_INTERFACE_DESCRIPTOR* structure. The interface descriptor requested is relative to

the configuration descriptor. The caller should properly specify the configuration descriptor index *ConfigIndex* and interface descriptor index *InterfaceIndex* within the configuration.

The function sends a *IOCTL_USB_GET_DESCRIPTOR* request to the Driver.

4.6.16. USBGetStatus

Definition:

```
DWORD USBAPI
USBGetStatus(
   IN  usb_t                    Device,
   IN  REQUEST_TARGET   Target,
   OUT USHORT*                  wStatus,
   IN  OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

Target
One of request recipient defined by REQUEST_TARGET enumeration.

wStatus
Points to the buffer to return status.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code.

Comments:
The function clears a specified feature for the specified recipient. Feature selectors should be *USB_FEATURE_ENDPOINT_STALL* or *USB_FEATURE_REMOTE_WAKEUP*.
The function sends a *IOCTL_USB_STATUS_CONTROL* request to the Driver.

4.6.17. USBGetStringDesc

Definition:

```
DWORD USBAPI
USBGetStringDesc(
   IN  usb_t                       Device,
   IN  BYTE                        Index,
   IN   USHORT                     LangId,
   OUT PUSB_STRING_DESCRIPTOR      pDescriptor,
   IN  OUT DWORD                   *cbSize,
   IN  OUT OVERLAPPED              *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

Index
Requested string descriptor index

LangId
Requested language ID for string descriptor

pDescriptor
Points to the buffer for requested string descriptor

cbSize
Specifies bytes count of string descriptor to be returned

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments:
This function returns the string descriptor in a `USB_STRING_DESCRIPTOR` structure. The string itself is found in the variable-length `bString` member of the string descriptor. The caller should allocate enough memory to store the string in the `bString` member. The function sends a `IOCTL_USB_GET_DESCRIPTOR` request to the Driver.

4.6.18. USBIoCtrl

Definition:

```
DWORD USBAPI
USBIoCtrl(
   IN usb_t                Device,
   IN DWORD                dwIoControlCode,
   IN LPVOID               lpInBuffer,
   IN DWORD                nInBufferSize,
   IN OUT LPVOID           lpOutBuffer,
   IN DWORD                nOutBufferSize,
   OUT LPDWORD             lpBytesReturned,
   IN OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle

dwIoControlCode
Specifies request IOCTL code.

lpInBuffer
Specifies request input buffer.

nInBufferSize
Specifies request input buffer size.

lpOutBuffer
Specifies request output buffer.

nOutBufferSize
Specifies request output buffer size.

lpBytesReturned
Points to variable to hold actual bytes processed by request.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:

On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
The routine performs MOTUSB Device request directly.

4.6.19. USBLockDevice

Definition:

```
DWORD USBAPI
USBLockDevice(
   IN  usb_t          Device,
   IN  DWORD          Flags
 );
```

Parameters:

Device
Points to the opened Device handle

Flags
Can be the following

USB_LOCK_DEVICE	– aquire Device lock
USB_TRY_LOCK_DEVICE	– try to aquire lock Device
USB_UNLOCK_DEVICE	– release Device lock
USB_TOTAL_UNLOCK	– release all locks belongs to this handle

Returns:
On success, routine returns *`USB_STATUS_SUCCESS`*, or else some MOTUSB error code.

Comments:
By using this operation the Client application can lock access to the Device for other Clients. This function locks the Device by means of the Device handle. The Device handle specified in this request then becomes a master handle, so that a request with any other Device handle will be blocked or returned with error. Only access to those operations that change the Device state and data transfers will be blocked. The function sends a *`IOCTL_USB_LOCK_DEVICE`* request to the Driver.

4.6.20. USBOpenDevice

Definition:

```
DWORD USBAPI
USBOpenDevice(
   IN HDEVINFO   devList,
   IN int         index,
   OUT usb_t*     Device
 );
```

Parameters:

devList
The system Device list handle provided by *USBGetDeviceList* function.

Index
The Device index in the list.

Device
Pointer to output Device handle for opened Device object.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
The Client application should use this function to establish a connection to a Device object. Typically this should start from *USBGetDeviceList* to acquire the Device list. Then, for each Device index starting with zero index, it should attempt to open the Device. If the application opens a handle, it can request the descriptor and then decide whether it is a required Device. If it is not a required Device, the Client should close the handle using the *USBCloseDevice* routine and continue attempting to open Devices, by incrementing the *index* parameter. If the function returns *USB_STATUS_NO_SUCH_DEVICE*, this means that the *index parameter* is too big and no Device with such an index is available. In this case the application should stop trying to open the Device and release the Device list using *USBReleaseDeviceList*.

4.6.21. USBOpenPipe

Definition:

```
DWORD USBAPI
USBOpenPipe(
   IN  usb_t        Device,
   IN  UCHAR  endpointAddress,
   OUT usb_t*  pipe
 );
```

Parameters:

Device
Points to the opened Device handle.

enpointAddress
Endpoint address from endpoint descriptor for required pipe.

pipe
Points returned pipe handle.

Returns:
On success, routine returns ***USB_STATUS_SUCCESS***, or else some MOTUSB error code

Comments:
This function establishes a connection to the pipe object for a specified endpoint address. The routine returns a pipe object handle to the ***pipe*** parameter. The pipe object handle links to the Device handle. Closing the Device handle causes the closure of all linking pipe handles.

4.6.22. USBPipeGetDescriptor

Definition:

```
DWORD USBAPI
USBPipeGetDescriptor(
   IN  usb_t                          pipe,
   PUSB_ENDPOINT_DESCRIPTOR desc
 );
```

Parameters:

pipe
Points to the opened pipe handle.

desc
Points to buffer to place the endpoint descriptor.

Returns:
On success, routine returns ***USB_STATUS_SUCCESS***, or else some MOTUSB error code

Comments:
The routine returns an endpoint descriptor by a given pipe handle.

4.6.23. USBReadPipe

Definition:

```
DWORD USBAPI
USBReadPipe(
   IN  usb_t                pipe,
   OUT LPVOID               buf,
   IN  OUT DWORD            *size,
   IN  OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

pipe
Points to the opened pipe handle.

buffer
Points buffer for input data.

size
Specifies requested bytes count. Function returns bytes count actually transmitted in the variable pointed by this parameter.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL to issue a synchronous request.

Returns:

On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments:

This function performs IN transfers from Device to Host. The caller specifies the transfer size in bytes *size* parameter. With bulk and interrupt transfers, if the current maximum transfer length is less than the requested size, the Driver breaks the transfer into blocks. Note that for isochronous transfers, the `buffer` parameter should point to the isochronous transfer buffer (see 3.4).

4.6.24. USBRegisterDevNotify

Definition:

```
HDEVINFO USBAPI
USBRegisterDevNotify(
   IN  HWND      hWnd
 );
```

Parameters:

hWnd specifies window handle for which notification enables.

Returns:

Function returns system notification handle. This handle should be used by `USBUnregisterDevNotify` when the caller deregisters notifications or this window is destroyed.

Comments:

This function registers a specified window for Device notifications. The notification becomes as the `WM_DEVICECHANGE` window message, where the `lParam` parameter points to the buffer with `DEV_BROADCAST_DEVICEINTERFACE` structure, from which the Client application can extract required fields about notification.

4.6.25. USBReleaseDeviceList

Definition:

void USBAPI
USBReleaseDeviceList(
IN HDEVINFO devList
);

Parameters:
devList
the system Device list handle provided by the `USBGetDeviceList` function.

Returns:
None

Comments:
This function releases the system Devices list handle. The Client application should call this routine when it no longer needs the Device list.

4.6.26. USBResetDevice

Definition:

```
DWORD USBAPI
USBResetDevice(
   IN usb_t                    Device,
   IN OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:

On success, routine returns ***USB_STATUS_SUCCESS***, or else some MOTUSB error code.

Comments:

This function is used to reset the Device port. All the pending transactions on the bus should be aborted. The request causes all of the status and configuration values associated with endpoints in the affected interfaces, to be set to their default values.

The function sends a ***IOCTL_USB_RESET_DEVICE*** request to the Driver.

4.6.27. USBResetPipe

Definition:

```
DWORD USBAPI
USBResetPipe(
   IN usb_t                    pipe,
   IN OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

pipe
Points to the opened pipe handle.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:

On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code.

Comments:

This function resets a stalled pipe. It cancels all pending I/O on the pipe and sends *CLEAR_FEATURE* with *USB_FEATURE_ENDPOINT_STALL* selector for the specified endpoint. When receiving *USB_STATUS_STALL_PID* error code on bulk or interrupt transfers, the Client application should try to reset the pipe. If this does not help, the Client should try to reset the Device.

The function sends a *IOCTL_USB_RESET_PIPE* request to the Driver.

4.6.28. USBSetConfiguration

Definition:

```
DWORD USBAPI
USBSetConfiguration(
   IN  usb_t                       Device,
   IN  UCHAR                       ConfigIndex,
   IN  LONG                        InterfaceCount,
   IN  PUSB_INTERFACE_SETTING      Interfaces,
   IN  OUT OVERLAPPED              *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

ConfigIndex
Configuration descriptor index.

InterfaceCount
Interfaces count that should be configured with this call. Should be more than or equal to 1. If –1 is specified, all interfaces are configured.

Interfaces
Points to the buffer that contains the array of interface information items for each interface configured. The count of valid entries should be equal to *`InterfaceCount`*. Must be NULL if *`InterfaceCount`* is equal to –1.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns **USB_STATUS_SUCCESS**, or else some MOTUSB error code.

Comments:
This function configures the Device. The caller can specify only the set of interfaces that will be configured for a selected configuration. To invoke this request, the Device first must be unconfigured. This request causes all of the status and configuration values associated with endpoints in the affected interfaces, to be set to their default values.

The function sends a *`IOCTL_USB_SET_CONFIGURATION`* request to the Driver.

4.6.29. USBSetFeature

Definition:

```
DWORD USBAPI
USBSetFeature (
   IN usb_t                  Device,
   IN REQUEST_TARGET  target,
   IN UCHAR                  feature,
   IN OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

target
One of request recipients defined by *REQUEST_TARGET* enumeration.

feature
Specifies feature selector.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns *USB_STATUS_SUCCESS*, or else some MOTUSB error code

Comments:
This function sets a specified feature for the specified recipient. Feature selectors should be *USB_FEATURE_ENDPOINT_STALL* or *USB_FEATURE_REMOTE_WAKEUP*.

The function sends a *IOCTL_USB_FEATURE_CONTROL* request to the Driver.

4.6.30. USBUnconfigureDevice

Definition:

```
DWORD USBAPI
USBUnconfigureDevice(
   IN usb_t                    Device,
   IN OUT OVERLAPPED *pOverlapped
 );
```

Parameters:

Device
Points to the opened Device handle.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL.

Returns:
On success, routine returns ***USB_STATUS_SUCCESS***, or else some MOTUSB error code.

Comments:
This function puts the Device into the unconfigured state. This operation should be rarely used by Client software developers, due to issues with the Operating System working in this state. However this operation can be very useful for new Device testing. In addition the Client software should use this request before setting different configurations on an already configured Device.

The function sends a ***IOCTL_USB_UNCONFIGURE_DEVICE*** request to the Driver.

4.6.31. USBUnregisterDevNotify

Definition:

```
void USBAPI
USBUnregisterDevNotify(
   IN HDEVINFO hDevInfo
 );
```

Parameters:

hDevInfo

Specifies the system notification handle obtained from **USBRegisterDevNotify.**

Returns:

None

Comments:

This function deregisters a specified window from Device notifications.

4.6.32. USBWaitIO

Definition:

```
DWORD USBAPI
USBWaitIO(
   IN  usb_t                handle,
   OUT DWORD                *BytesTransferred,
   IN  DWORD                Timeout,
   IN OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

handle
Points to the opened Device or pipe handle.

BytesTransferred
Points to buffer for output data size. Can be NULL if not used.

Timeout
Specified interval for a waiting I/O completion in milliseconds. If no timeout value should be used the `INFINITE` constant should be specified.

pOverlapped
Points to variable used for overlapped I/O, that was specified for the last I/O operation on the specified handle.

Returns:

On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code

Comments:

The Client should call this function if it specifies it's own value for the `pOverlapped` parameter. The function waits until the I/O operation completes. The calling thread of this routine will be blocked until the function completes. On return the function returns a bytes count actually transmitted into the variable pointed by `BytesTransferred` parameter (if no NULL is specified).

4.6.33. USBWritePipe

Definition:

```
DWORD USBAPI
USBWritePipe(
   IN  usb_t                  pipe,
   OUT LPVOID                 buf,
   IN  DWORD                  size,
   IN  OUT OVERLAPPED  *pOverlapped
 );
```

Parameters:

pipe
Points to the opened pipe handle.

buffer
Points to the buffer with data to transfer.

size
Specifies bytes count to send.

pOverlapped
Points to variable used for overlapped I/O. Can be NULL of no overlapped I/O required.

Returns:

On success, routine returns `USB_STATUS_SUCCESS`, or else some MOTUSB error code.

Comments:

This function performs OUT transfers from Host to Device. The caller specifies the transfer size in bytes in the `size` parameter. If the current maximum transfer length is less than requested size, the Driver breaks the transfer into blocks (for bulk and interrupt transfer types). Note that for isochronous transfers, the `buffer` parameter should point to the isochronous transfer buffer (see 3.4).

5. Registry Settings.

The default settings for MOTUSB Driver are stored in the registry. These settings are applied to every Device on which the MOTUSB Driver is installed. The registry settings are stored under the following key: *HKLM \ SYSTEM \ CurrentControlSet \ Services \ motusb \ Parameters*.

All the registry settings are shown in the following table:

Table 5.1 Registry settings summary

Value Name	Default Value	Description
CancelIoOnSuspend	0	Handling of outstanding read or write requests when the Device goes into a suspend state (leaves D0): 1 = abort pending requests 0 = do not abort pending requests
MaxTransferSize	65535	Default maximum transfer size in bytes. This value is used on default Device configuration or when the Client application specifies use of default maximum transfer size in Set Configuration or Set Interface calls. Can be from 4096 to 2147483647
RequestTimeout	5000	Timeout interval for synchronous I/O requests, in milliseconds. Zero means infinite (no timeout). USB1.1 specification defines 5 seconds timeout. However this value can be useful during firmware debugging
ShortTransferOK	1	Specifies that short packets in bulk and interrupt transfers are accepted with no errors.
UnsafeRemovalUI	1	Specifies whether Windows “Unsafe Removal” dialog should appear on hot Device disconnection. 1 = dialog should appear 0 = dialog should not appear

6. Driver Installation.

6.1. Installation Procedure.

Various Devices can use the MOTUSB Device Driver. The Device vendor must provide a proper Setup (INF) file for Device.

Assuming the name is < **your_oem** >**.inf** to Device Driver the following steps are required:

12. Logon to Windows 2000 using an administrator account.

13. Ensure that the following 3 files are all contained in the Driver installation directory: **motusb.sys, motusb.dll, mcf5272.inf**

14. Ensure that the VendorID and ProductID members of the Device descriptor on Device have not changed. If you have to change them, it is necessary to make a new installation (INF) file for the VendorID and ProductID member values combination. (See MOTUSB Driver Guide, Chapter 4 for detailed information on the INF file).

15. Connect the Host PC with the UFTP Device running on the MCF5272 development board via a USB cable.

16. “Found New Hardware Wizard” dialog with string “USB Device” will appear. Select “Next” button.

17. Select the radio button labeled "Search for a suitable Driver for your Device (Recommended)" and then hit the "Next" button.

18. “Locate Driver Files” page will appear, click the "Next" button

19. “Insert manufacturer installation disk on the drive…” file prompt dialog will appear. Specify the folder where all Driver files are located and click ok.

20. “Driver Files Search Result” page should appear. If the Driver path is specified correctly “Windows found a Driver for this Device” and the path to mcf5272.inf strings will be shown at the center of the page.

21. Hit the "Next" button, whereupon the "copying Files" message box will be seen briefly; then once again the "Found New Hardware Wizard" box, now displaying the subheading "Hardware Install: The hardware installation is complete". Hit the "Finish" button.

22. A copy of **motusb.sys** should be in the %SystemRoot%\System32\Drivers directory, and the **motusb.dll** in the %SystemRoot%\System32 directory. If the

final "Add New Hardware Wizard" box indicates any error, or if the OS indicates that a reboot is required in order to finish the installation of this Device, something has gone wrong. Check the Inf file or Install directory, follow the instructions again for a 'clean' install, and start over.

1. Make sure that the following 3 files are all contained in the Driver installation directory:
   motusb.sys, motusb.dll, < your_oem>.inf

2. Connect Host PC to a running Device via the USB cable.

3. "Found New Hardware Wizard" dialog with string "USB Device" will appear. Select "Next" button.

4. Select the radio button labeled "Search for a suitable Driver for your Device (Recommended)" and then hit the "Next" button.

5. "Locate Driver Files" page will appear, click the "Next" button

6. "Insert manufacturer installation disk on the drive…" file prompt dialog will appear.

7. Specify the folder where all Driver files are located and click ok.

8. "Driver Files Search Result" page should appear. If Driver path is specified correct "Windows found a Driver for this Device" and path to <your_oem>.inf strings will be shown on the center of the page.

9. Hit the "Next" button. "Copying Files" dialog will be seen briefly, then once again the "Found New Hardware Wizard" box, now displaying the sub-heading, "Hardware Install: The hardware installation is complete". Hit the "Finish" button.

10. A copy of **motusb.sys** should now be in the %SystemRoot%\System32\Drivers directory, and **motusb.dll** in the %SystemRoot%\System32. If the final "Add New Hardware Wizard" box indicates any error, or if the OS indicates that a reboot is required in order to finish the installation of this Device, something has gone wrong. Check the Inf file or Install directory, follow the instructions in the section below for a 'clean' install, and start over again.

Depending on the inf file content, the vendor Device Driver setup may be somewhat different.

NOTE: For Windows 2000, to be able to install a Device Driver, administrator rights are required. The MOTUSB Driver is installed in the same way as any other Plug&Play

Device Driver, where the installation requires administrator rights. Once the MOTUSB Driver is installed, standard user rights are sufficient to load the Driver and to use the Driver by accessing its programming interface.

6.2. Setup (INF) File.

To be installed correctly, Drivers must have an INF file. An INF file is a text file that contains all the necessary information about the Device(s) and file(s) to be installed, such as Driver images, registry information, version information, and so on, to be used by the Setup components. An INF file is basically an ASCII text file. The contents and the syntax of an INF file are documented in the Microsoft Windows 2000 DDK.

The INF file is loaded and interpreted by an operation software component that is closely related to the Plug&Play Manager, called the Device Installer. It handles hot plugging and removal of USB Devices. If the new USB Device has been detected, the system searches its internal INF file database, located in %SystemRoot%\Inf\, for a matching Driver. If no Driver can be found the New Hardware Wizard pops up and the user will be asked for a Driver.

A particular Device can be associated with the MOTUSB Driver through a string that is called Hardware ID. The operation system PnP software component builds this string from the 16-bit vendor ID (VID), the 16-bit product ID (PID), optionally the revision code (REV) and other components. For USB Devices, the Hardware ID is prefixed by the 'USB' identifier. The OS uses the ordered Hardware ID lists provided by the bus Driver, along with INF information, to select Drivers to load for a Device. Starting at the top of the ordered Hardware ID list, the OS tries to match the Hardware ID there with a Hardware ID in a system INF file entry.

Here is the template for Hardware ID string, that the vendor should specify in the INF file.

USB\VID_xxxx&PID_yyyy&REV_zzzz
USB\VID_xxxx&PID_yyyy

The MOTUSB Driver installation should install the Driver *motusb.sys* and dynamic link library *motusb.dll* images. However *motusb.dll* is optional and if the vendor wants to use the Driver directly, no library image is required.

The description of other INF file entries is outside the scope of this guide, please refer to Microsoft Windows 2000 DDK for detailed information. The setup information file template is shown in the next section and should provide the basis for making the INF file to install the MOTUSB Driver on a vendor Device. Note that the template assumes that MOTUSB image files are located in a same directory with an INF file. The minimum set of information should be known before modifying this template:

- ProductID and VendorID values of Device descriptor
- Manufacturer name (replaces '_Your_Device_Manufacturer_Name_Here_' statement)
- Device class (optionaly)

Please see comments in the template for a more detailed description.

6.2.1. Setup (INF) File Template.

```
; ============================================================
; This is a MOTUSB Driver Setup Information (INF) file template.
; ============================================================
[Version]
Signature   = "$WINDOWS NT$"
Provider    = %MfgName%
DriverVer   = 02/23/2002,1.23.0000.00
CatalogFile = motusb.cat

;============================Class Section=====================================
; Select an appropriate class for the Device.
; There are several options:
; + Use the MOTUSB class.
; + Define your own class by generating a GUID and a specify class description.
; + Use a predefined system class. This is required for system defined classes
;   ( HID, Mass Storage, USB Audio for example)
;
;   HID Example:
;   Class=HIDClass
;   ClassGuid={745a17a0-74d3-11d0-b6fe-00a0c90f57da}
;==============================================================================
Class     = MOTUSB
ClassGUID = {31A6857E-E756-413f-93B2-9FC95EDB7608}

;========================= Class Install Section ============================
; The following 3 sections used for own vendor classes only. Remove'em if using
; system defined Device class
;==============================================================================
[ClassInstall]
Addreg=MOTUSBClassReg

[ClassInstall32]
Addreg=MOTUSBClassReg

[MOTUSBClassReg]
HKR,,,,%MyClassName%
HKR,,Icon,,-20          ; Use USB Icon



;========================= Control Flags Section ============================
[ControlFlags]
ExcludeFromSelect=*

; ========================= Driver Source Sections ==========================
[SourceDisksNames]
1=%DiskID%,,,
```

```
[SourceDisksFiles]
motusb.sys = 1
motusb.dll = 1

; NOTE: Replace _Your_Device_Manufacturer_Name_Here_ with Manufacturer name
[Manufacturer]
%MfgName%=_Your_Device_Manufacturer_Name_Here_

; ========================= Device List Section ==============================
; There is the place to add your Device. MOTUSB Driver will be installed
; for Devices your declared here on the Driver installation time.
; ----------------------------------------------------------------------
; To declare your Device:
; + ProductID and VendorID values of Device descriptor should be known.
; + Put YourDeviceDescXXX variable to the string section below
; + Put line like following here (assume VendorID=0x1045, ProductID=0x23):
;   %ColdFire.DeviceDesc0%=MOTUSB, USB\VID_1045&PID_0023
 ; ======================================================================

     ; NOTE: Replace _Your_Device_Manufacturer_Name_Here_ with Manufacturer name
[_Your_Device_Manufacturer_Name_Here_]
%YourDeviceDesc0%=MOTUSB, USB\VID_ABCD&PID_1234
%YourDeviceDesc1%=MOTUSB, USB\VID_ABCD&PID_1235

; ========================= Misc Driver file sections ========================
[DestinationDirs]
DefaultDestDir   = 12
MOTUSB.Files.Sys = 10,System32\Drivers
MOTUSB.Files.Dll = 10,System32

[motusb]
CopyFiles = MOTUSB.Files.Sys, MOTUSB.Files.Dll
AddReg = MOTUSB.AddReg, DeviceParams.NTx86

[motusb.NTx86]
CopyFiles=MOTUSB.Files.Sys, MOTUSB.Files.Dll
AddReg=MOTUSB.AddReg, DeviceParams.NTx86

[motusb.NTx86.Services]
Addservice = motusb, 0x00000002, MOTUSB.NTx86.AddService

[motusb.NTx86.AddService]
DisplayName    = %MOTUSB.SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 3                  ; SERVICE_DEMAND_START
ErrorControl   = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary  = %10%\System32\Drivers\motusb.sys
LoadOrderGroup = Base

[motusb.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,MOTUSB.sys

[motusb.Files.Sys]
motusb.sys

[motusb.Files.Dll]
motusb.dll

; ========================= Registry Settings ================================
; The default registry setting stored here. The last value is a particular
; registry setting value. This values can be modified by vendor to specify
```

```
; different registry setting
; ============================================================================
[DeviceParams.NTx86]
HKLM,"%ConfigPath%"\"%DeviceConfigPath%",RequestTimeout,0x10001,5000
HKLM,"%ConfigPath%"\"%DeviceConfigPath%",UnsafeRemovalUI,0x10001,0
HKLM,"%ConfigPath%"\"%DeviceConfigPath%",ShortTransferOK,0x10001,1
HKLM,"%ConfigPath%"\"%DeviceConfigPath%",MaxTransferSize,0x10001,65535


; ============================== String Section ==============================
[Strings]
MfgName              = "<<< Put manufacturer name here >>>"
MyClassName          = "<<< Put vendor defined class here >>>"
YourDeviceDesc0      = "<<< Put your Device #0 description here >>>"
YourDeviceDesc1      = "<<< Put your Device #1 description here >>>"
DiskID               = "<<< Insert your distribution disk description here >>>"
MOTUSB.SvcDesc       = "Motorola USB I/O Driver"
ConfigPath           = "SYSTEM\CurrentControlSet\Services\motusb"
DeviceConfigPath     = "Parameters"
```

6.3. Updating Or Uninstalling.

In order to update or uninstall the MOTUSB Driver, the Device Manager has to be used. In the Device Manager double-click on the entry of the Device and choose the property page that is labeled "Driver". The Driver reinstallation can be issued through the "Update Driver" button. The operating system launches the Upgrade Device Driver Wizard, which searches for Driver files or lets it select a Driver. In order to uninstall the Driver, the "Uninstall" button should be used. The operating system will reinstall a Driver the next time the Device is connected or the system is rebooted. In some cases such automatic reinstallation may be unwelcome, and to avoid this it is necessary to manually remove the INF file that was created by the system at Driver installation time.

During Driver installation Windows stores a copy of the INF file in its internal INF file database that is located in %System32%\INF\. The original INF file is renamed and stored as oemXX.inf for example, where XX is a decimal number. The best way to find the correct INF file is to do a search for some significant string (Device name in Device Manager for example) in all the INF files in the directory %System32%\INF\ and its subdirectories. Once the INF file has been located, remove it. This will prevent Windows from automatically reinstalling the MOTUSB Driver at the time of attaching a USB Device. Instead, the New Hardware Wizard will be launched and the user will be asked for a Driver.

7. Appendix 1: USB Audio Sample for MCF5272.

7.1. Introduction.

7.1.1. Overview.

The following describes a very small application for isochronous transfers demonstration from Host to Device, and from Device to Host. It is designed especially for Motorola ColdFire5272 USB Protocol Stack audio sample (see [3]) firmware. The Client application demonstrates how the ColdFire firmware Driver handles control and isochronous transfers. The MCF5272 USB Driver and audio sample Client firmware components work together on the Device, to perform PCM samples loop-back using simultaneous isochronous transfers through isochronous IN/OUT endpoints. The application is capable of demonstrating the Stand-Alone and uClinux versions of the Device-side firmware component, due to USB protocol transparency. Additionally the application demonstrates a working MOTUSB Driver and dynamic link library software components on the Host side.

7.1.2. System Requirements.

Hardware:

- Single CPU Intel i386 based PC (> 600mHz) with Open Host Controller or Universal Host Controller.
- Sound adapter with
  - 44.1 kHz and 8 kHz sample rates supported
  - Line-In or Mic-In sockets

Software:

- OS: Windows 2000 Professional
- MOTUSB Driver for USB Audio Sample Device installed

7.1.3. Application Capabilities.

- Demonstrates that isochronous ColdFire 5272 USB firmware can meet real time requirements.
- Able to show how Device processes missed frame tokens (by using *syshalt* utility).
- Shows Device s/w can process Isochronous IN and Isochronous OUT transfers transactions simultaneously.
- Shows how Device s/w can process Control and Isochronous transfers transactions Simultaneously.

7.2. Application overview.

7.2.1. Sample Model.

The sample demonstrates an audio loop-back through Isochronous USB firmware on the ColdFire MCF5272. The sound data from a microphone is sent by the Client application to the USB Device. The Client application receives this data from USB and the speech may be heard on the headphones. Thus the audio source samples circulate over the bus and returns to the audio output Device. In addition to the audio loop-back, the Device performs some simple audio processing: changes the volume level according to the commands that the Host Client sends during control transfers.

To perform a loop-back the application has to accomplish the following 4 tasks simultaneously:

1) Takes PCM samples data from Sound-In (microphone for example)
2) Transfers Sound-In Samples to USB through Isochronous OUT endpoint
3) Receives Device loop-back results from Isochronous IN endpoint
4) Writes Device loop-back results to Sound-Out Device (speakers for example)

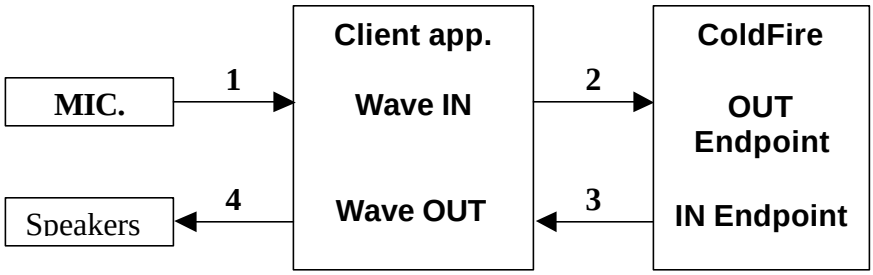

Fig 7.1 Sample model.

When adjusting the volume, the Client application sends a special command to the Device (this task is not shown in the figure) using control transfers. On request of this command, the ColdFire firmware performs volume processing, so that the PCM samples that the Device returns, are not the same as those received from the microphone. Note that due to buffering on the Host Client application and the Device firmware, the returned samples will be delayed approximately ~200ms.

Another option is the sample rate selection. The following sample rates may be selected: 44kHz and 8 kHz. Each sample rate corresponds to a specified alternate setting on the Device. Therefore sample rate selection changes the endpoints' parameters, set according to the interface alternate setting. For sound data transmission with the 44 kHz sample rate, one short packet is transmitted for every 10 packets [3]. For 8 kHz sample rate, no short packets are transmitted. For Device endpoints and interface configurations please refer to [3]. The Stand-Alone firmware is somewhat different from the uClinux firmware

for the isochronous transfer model, so here the short packets processing can be a point of interest.

7.2.2. Audio System Setup.

As mentioned above the sample requires a sound source and a sound output Device. The microphone is a good example of a sound source. Other sound sources can be connected to the Mic-In socket (in case of microphone) or Line-In (TV Tuner as an example). The sound should be heard through the sound output Device (speakers). There are a number of possible causes of missing sound:

- The microphone line (or Line-In) is muted.
- The volume of the microphone line (or Line-In) is turned to the minimum.
- The wave balance is muted or turned to the minimum.

The following figure shows right Line-In and Microphone Balance setting.

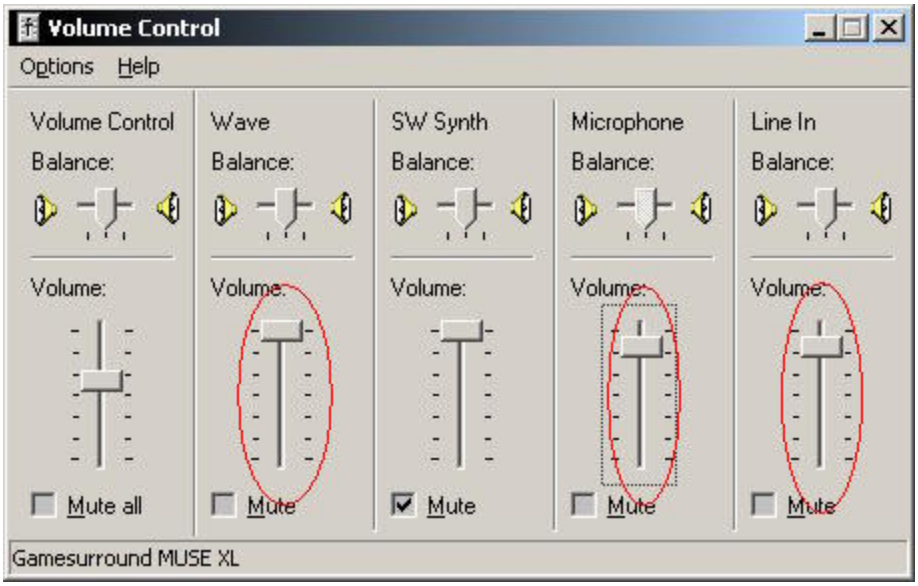

Fig 7.2 Playback properties.

If the sound source Device is working correctly, check the 'Mute' Line-In and Microphone Balance settings. This ensures that no signal should appear on the sound output Device from the sound source input. Otherwise an echo will be heard during sample application execution.

Next go to Options menu and select Properties; check Recording box and press OK, the following dialog should appear:

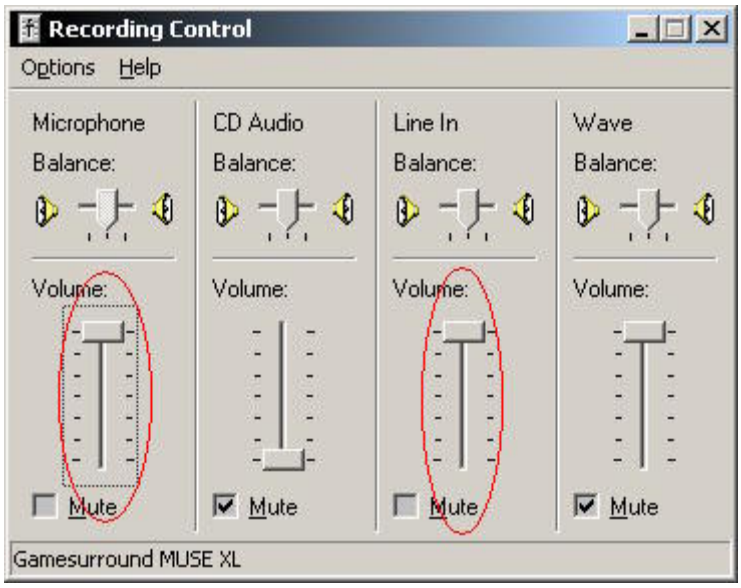

Fig 7.3 Recording properties.

This dialog provides the ability to setup miscellaneous sound source settings. Uncheck the 'Mute' box and setup the volume Line-In and Microphone Balance properties.

7.2.3. Interaction With Sample.

Before starting the application, it is necessary to ensure that the ColdFire USB audio sample firmware was first downloaded to the Device RAM and started up. Connect the Device to the PC via the USB capable. If firmware does not download or start, or some other error occurs, the following message will appear on application startup:

Fig 7.4 "Device is not connected" Message Box.

NOTE:
This message box implies that the MOTUSB Driver did not load. Also it may be the cause of Host or Device software failure, or invalid *VendorID* and *ProductID* members of Device descriptor on the Device.

The application expects the following Device descriptor values:

VendorID = 0xABCD
ProductID = 0x1236

If the application started successfully the user should see the window as in Fig 7.5. This is the main application window through which the user can interact with the Device on ColdFire.

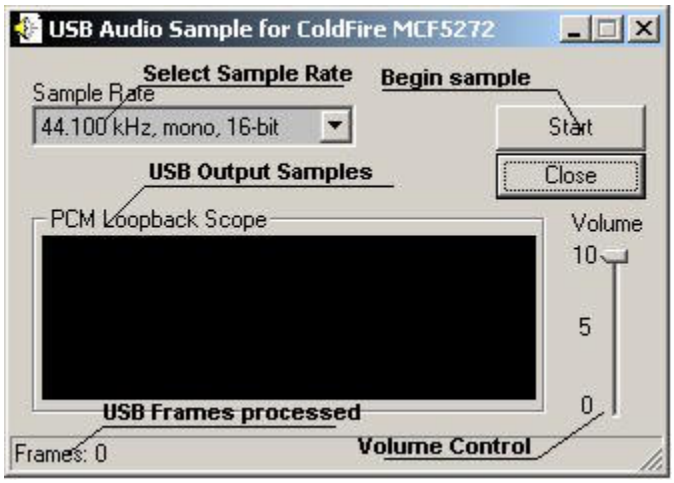

Fig 7.5 Main Application Window.

44.1kHz or 8 kHz sample rate can be selected before starting the sample. In order to start audio loop-back click the “Start” button. The application begins PCM samples delivery from the sound source to the USB Device on ColdFire and receiving processed data from the bus. The sample rate selection becomes unavailable during the loop-back operation. The Device delivered samples will be shown in PCM Loop-back Scope (fig 7.6). Volume control changes will affect the volume level of the data from the Device.

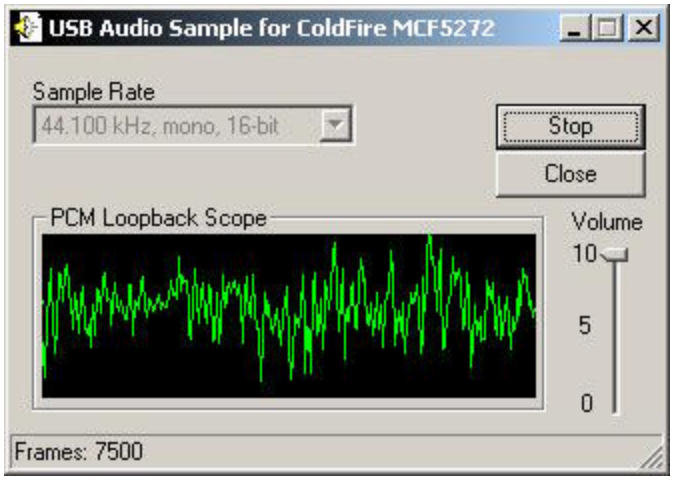

Fig 7.6 Main Application Window (running).

In order to stop the loop-back push the “Stop” button. The stop command will be sent to the Device and data delivery should be stopped within 100 milliseconds.

7.2.4. Missing Frames Emulation.

The additional utility *syshalt.exe* can be used to provide missing frame emulation. The utility halts all the running processes for 5 seconds. Thus, when transmitting data over the bus is in progress, the utility stops audio sampling along with Driver isochronous data delivery. This means that the Device does not receive IN and OUT tokens. The Device firmware and the sample should normally handle such a situation (Host real–time data delivery failure) and continue the loop-back when the system becomes unhalted. Due to the data buffering in this sample some noise may be observed within the first 100 milliseconds after system unhalts. For further information on processing of Device missing frames, refer to [2] and [3].

7.2.5. Known Issues.

The sample application is very time critical and involves the use of small buffers (for 50 milliseconds) for data transmission from sound adapter to USB, in order to achieve low latencies (~200ms). From time to time, depending upon the system loading, this can cause the sound to cut off. Another possible issue is that the USB and the sound adapter can become momentarily out of sync (no sample rate converter is implemented in the sampler). This manifests itself as an audible click during the consequent 50 milliseconds buffer reiteration. Moreover the *syshalt* utility may allow the USB data OUT transfer to overtake the IN transfer, in a single frame. In such a case the critical error message “Error: USB Mistiming” will be shown, and the sound loop-back will be stopped. This may be easily remedied by restarting. All the above mentioned issues should not cause isochronous transfer deadlock, and streaming will continue as soon as possible thereafter. In additional it should be noted that while this sample was tested on a 1 GHz Intel Pentium III processor, it should yield a similar performance on lower speed processors.

8. Appendix 2: USB File Transfer Sample for MCF5272.

8.1. Introduction.

This section describes a small application for file transfers from Host to Device, and from Device to Host. It's based on the UFTP protocol and designed specially for Motorola ColdFire5272 USB Protocol Stack file transfer sample (see [3]) firmware. The Client application demonstrates how the ColdFire firmware Driver handles control, bulk and interrupt transfers. The MCF5272 USB Driver and UFTP Client firmware components are working together on Device to represent a directory and perform transmission over the bus. The application is capable of demonstrating both uCLinux and Stand-Alone versions of the Device-side firmware components on account of the USB protocol transparency. In addition the application can be used to demonstrate a working MOTUSB Driver with dynamic link library and UFTP library software components on the Host side.

8.1.1. System Requirements.

Hardware:

- Single CPU Intel i386 based PC with Open Host Controller or Universal Host Controller.
- At least 800x600, 256 color video adapter

Software:

- OS: Windows 2000 Professional
- MOTUSB Driver for UFTP Device installed

8.1.2. Application Capabilities.

This demo application Client is based on the UFTP Device protocol and possesses the following capabilities:

- View directory content on Device
- Transmit files from Host to Device
- Transmit files from Device to Host
- Setting various transfers length for file transfers
- Deleting files on the Device

8.2. Application overview.

8.2.1. Starting Application.

Before starting the application, it is necessary to ensure that the File Transfer firmware is downloaded to the Device Ram and started up. Connect the Device to the PC via a USB cable. If the firmware was not correctly downloaded or started, or some other errors occurs the following message box will appear at application startup:

Fig 8.1 "Device doesn't connected" Message Box.

NOTE:
This message box implies that the MOTUSB Driver was not loaded properly. Also it may be due to Host or Device software failure, or invalid *VendorID* and *ProductID* members of the Device descriptor on the Device. The application expects the following Device descriptor values:

VendorID = 0xABCD
ProductID = 0x1235

8.2.2. Main Window.

If the application started up successfully the user should see a window similar to the one shown in Fig 8.2. This is the main application window through which the user can interact with the File Transfer Device on ColdFire.

The window contains two file lists:

1) PC Box - shows files at selected folder location
2) ColdFire Box - shows files on File Transfer Device

The ColdFire Box does not perform updates on any external changes made to the Device. The ColdFire file list updates on each write or delete operation.

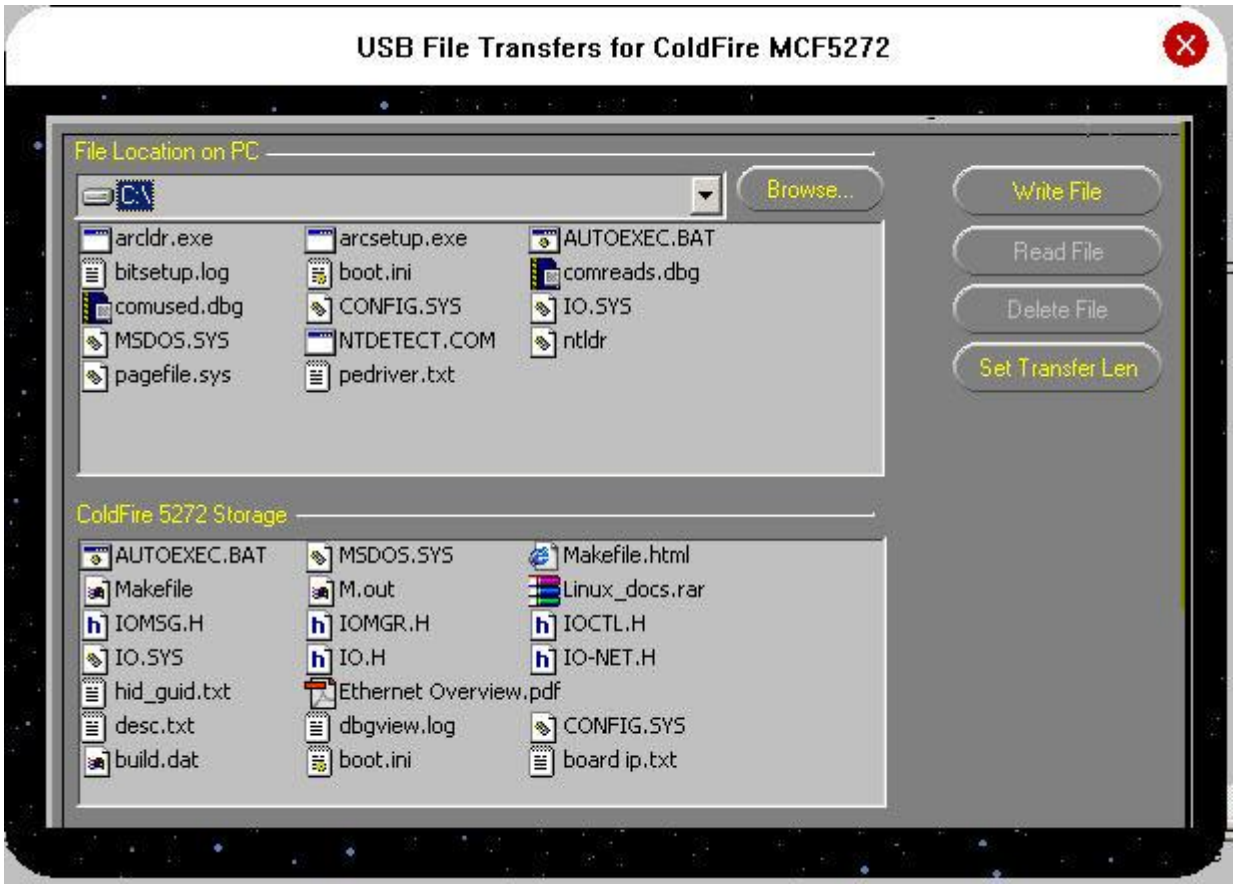

Fig 8.2 Application Main Window.

8.2.3. Application Operations.

1) To write files from Host to Device

Selects the required files in the PC Box and click the "Write File" button. The files should start transmitting to the Device. If any error occurs during file transmission the following error message box should appear:

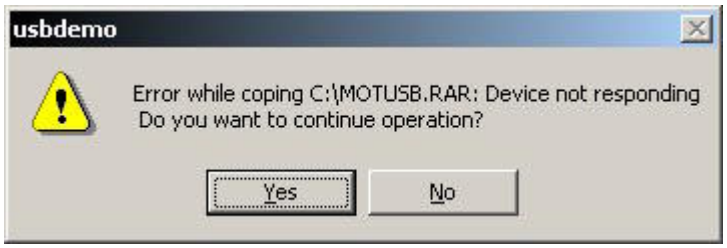

Fig 8.3 "Error while transfer" message box.

From this box the user can select whether he wants to continue transmission of other selected files or not.

2) To read files from Device to Host

Select the required files in the ColdFire Box and click the “Read File” button in the main window. The specified files will be transmitted to the current PC location (which the PC Box shows). This operation does not request a file overwrite operation, should any file with the same name already exist. Therefore any non-system file with the same name will be automatically overwritten without any further warning. If an error occurs during file transmission, the error message box should appear. From this box the user can select whether he wants to continue reception of other selected files or not.

3) To delete files on Device

Select files in the ColdFire Box window and click the “Delete File” button in the main window. If any error occurs during file deletion the error message box should appear. From this box the user can select whether he wants to continue deletion of other selected files or not.

4) Setting transfer unit

Click the “Set transfer length button”. The following dialog should appear:

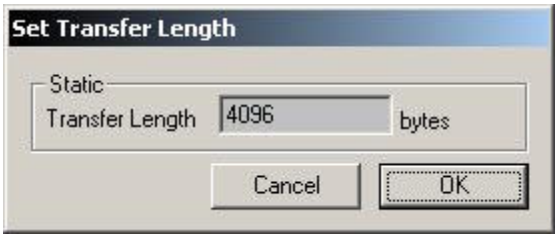

Fig 8.4 Transfer Length Dialog.

In this dialog the user can specify different transfer lengths and submit this selection by clicking OK. Note that due to specific UFTP configuration, the maximum transfer length can be up to 1M.

5) Different folder selection on the PC

If the user wants to change the current folder on the PC, the “Browse button” should be selected. The following dialog will appear:

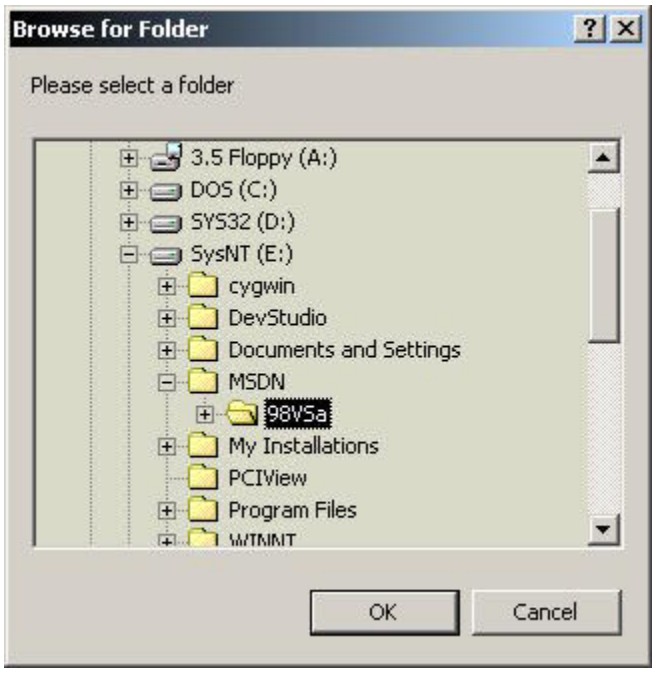

Fig 8.5 Browse for folder dialog.

From this dialog the user can select different folders to be displayed in the PC Box. Once the OK button is selected, the PC Box will be updated with the contents of the newly selected folder. In addition this folder will be inserted into the folder tree window (near the “Browse” button), and the user should be able to select this folder from there.

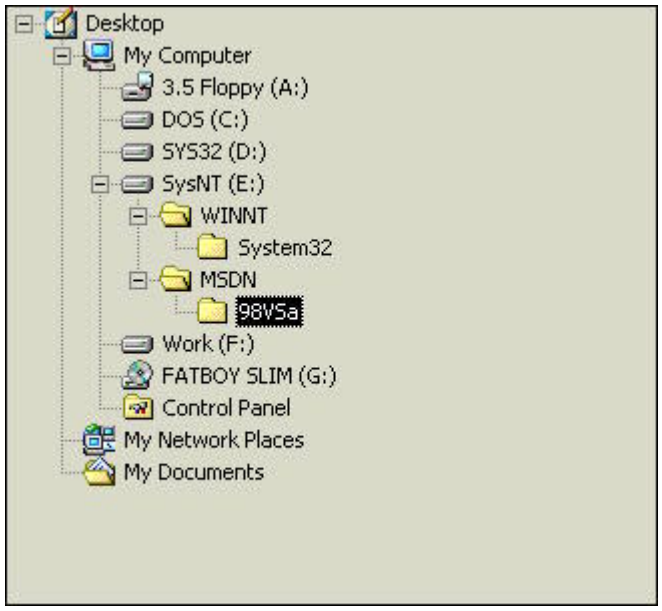

Fig 8.6 Folder tree window.

9. Appendix 3: Test Suite for MCF5272 USB Protocol Stack.

9.1. Introduction.

The USB Test Suite for a UFTP Device is provided to test out most of the software functionality of the Device and Host sides. The following Host software components take a part in some of the tests, and hence are automatically tested themselves:

- MOTUSB Device Driver and library
- UFTP library

The following Device USB Protocol Stack firmware components can be tested (Stand – Alone or uClinux versions):

- CBI USB Driver for MCF5272
- CBI & Isochronous USB Driver for MCF5272
- USB File Transfer Application
- USB Audio Application

The Test Suite does not communicate with the MOTUSB Driver directly. Instead it uses the MOTUSB library to request USB services. Note that this application in no way is intended to provide a USB compliance test. For this the "Microsoft Compliance Test Suite" from USB-IF should be used for USB Device framework testing or other USB Device classes.

9.1.1. System Requirements.

Hardware:

- Single CPU Intel i386 based PC with Open Host Controller or Universal Host Controller.
- At least 640x480, 256 color video adapter

Software:

- OS: Windows 2000 Professional
- MOTUSB Driver for Device installed

Firmware on MCF 5272 Evaluation Board:

- USB File Transfer Client running (VendorID = 0xABCD, ProductID = 0x1235) or
- USB Audio Client running (VendorID = 0xABCD, ProductID = 0x1236)

9.1.2. Test Suite content.

The single executable file provided. Location:

\bin

testsuit.exe – test suite executable

(All paths specified relative to the package installation directory).

9.2. Application Overview

As was mentioned above the Test Suite application is provided especially for the MCF5272 USB Protocol Stack firmware.

The test suite application consist from the following sections:

1) USB Standard Requests Testing

2) File Transfer Testing (applies to USB File Transfer Application and USB Driver firmware)

3) Isochronous Transfers Testing (applies to USB Audio Application and USB Driver firmware)

4) Other tests

Sections (2) and (3) are mutually exclusive, in that one of these tests becomes available, according to the Device firmware. Other tests apply to any Device for which the MOTUSB Driver is installed.

9.2.1. Selecting a Device.

The first screen to appear while executing the Test Suite prompts the user to select a suitable USB Device.

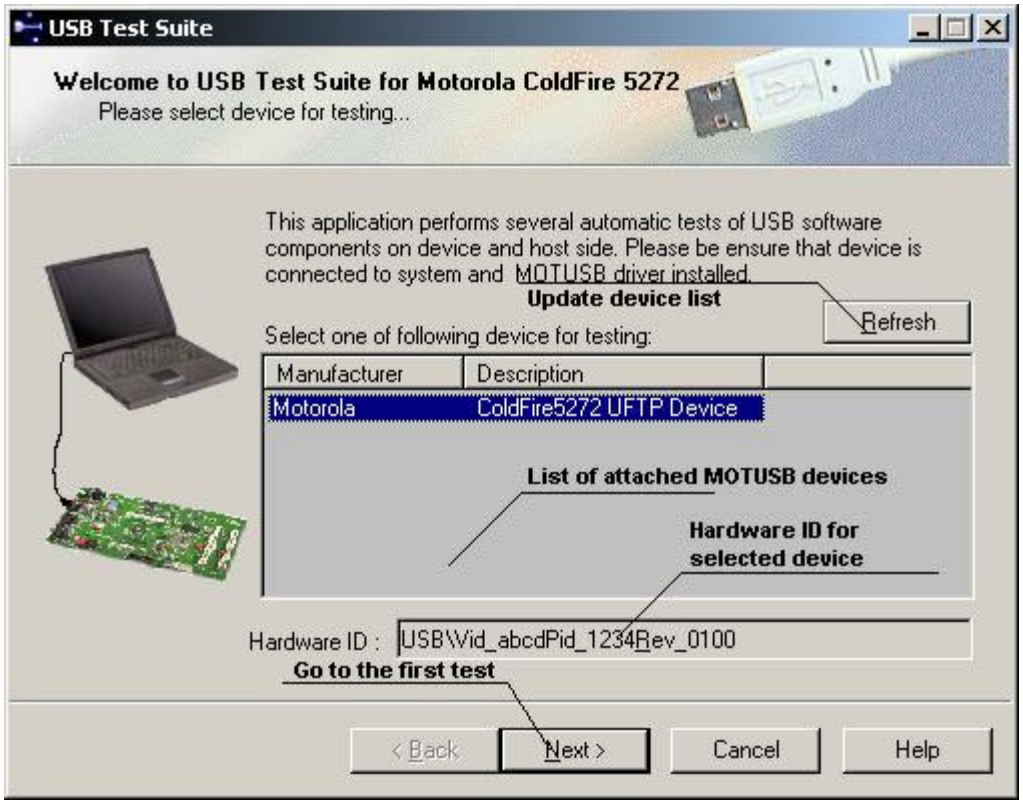

Fig 9.1 Device Selection Page.

USB Devices for which the MOTUSB Driver is installed should appear (disappear) in the Devices list attachment (removal). Once the USB Device is selected, the "Next" is clicked in order to go to the first test page. By using multiple instances of the Test Suite application, it is possible to test more than one Device at the same time. The Test Suite allows multiple instances of MOTUSB Client applications. However using only a single instance of the MOTUSB Client application is strongly recommended for testing. Using multiple instances of MOTUSB Client applications at the simultaneously while testing a Device, can lead to unexpected results.

NOTE:
In order to successfully select the Device for testing the firmware should be downloaded to the Device RAM and executed. If the MOTUSB Driver has not yet been installed, this would be an appropriate time to connect the Device to a PC and install the Driver. The Device should then appear in the Device list and can be selected for testing.

9.2.2. Automatic Standard Requests Testing.

After the Device selection page the next page is as follows.

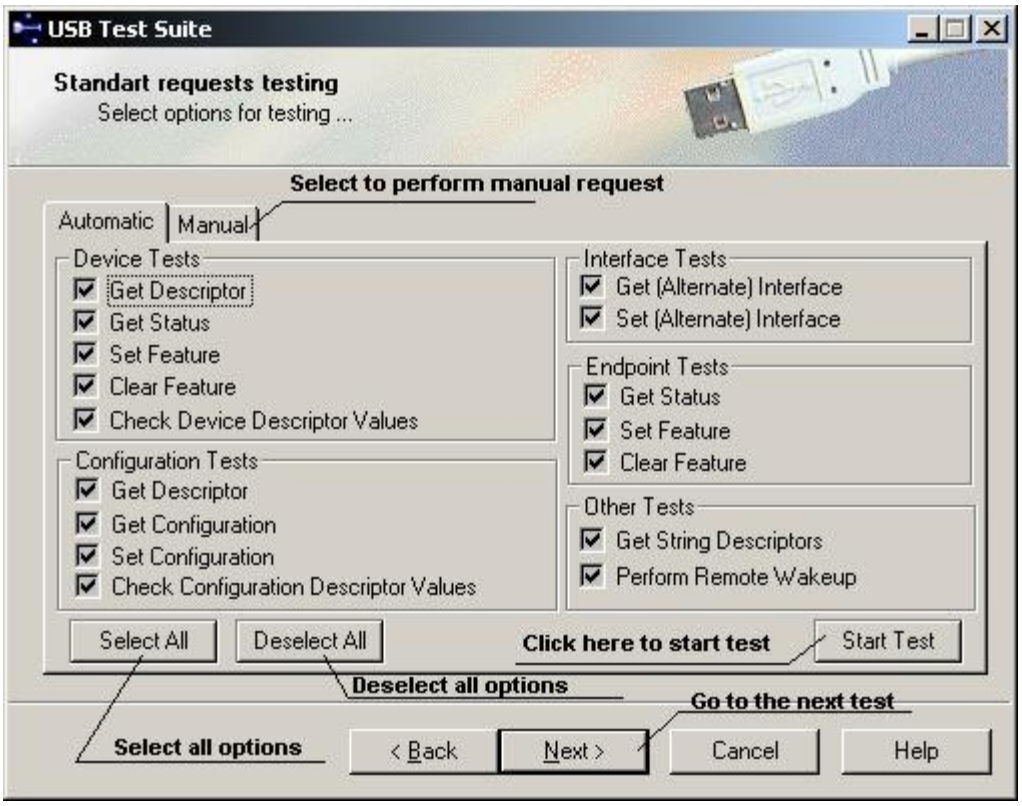

Fig 9.2 Standard requests (Automatic) page.

This is a page for automatic standard requests testing, which provides for some Device and Host side components testing. In this page the user can select various options for automatic Standard request testing. A Device may be tested in either the configured or unconfigured states. It should be noted that this test will lock the Device while it performs the necessary operations. Hence no other Client application can use the Device at that time.

The user should pay particular attention in the following circumstances:

1) If the tester aborts the procedure in the middle of a test, the test can unlock access to the Device in an undetermined state. So other Clients may report an error;

2) It is strongly recommended to never disconnect the Device during this test. USBDI will disrupt the system if the Device becomes disconnected while the Device Unconfigure request is pending. In such a case this will result in the classic “blue screen”.

The user can specify options for testing by selecting or deselecting the various choices. See option descriptions below.

9.2.3. DeviceTests.

Get Descriptor.
Requests Device descriptor. If this option is not specified, configuration and other Device tests cannot be selected.

Get Status
Gets status from the Device. If the Device status receipt was successful, the Test Suite verifies the status value. Device status valid values range from 0x00 to 0x03.

Set Feature / Clear Feature:
This test issues a *`Set Remote Wakeup Feature`* command for each configuration if this feature is supported as indicated in the *`bmAttributes`* field. If the specified option is selected, the Test Suite sets the Remote Wakeup feature on the Device. It then performs *`GetStatus`*, and checks if the Remote Wakeup bit is set. The application then clears this feature and verifies the Device again with *`GET_STATUS`*.

Check Device Descriptor Values:
Checks all Device descriptor fields.

9.2.4. Configuration Tests.

Get Descriptor:
Requests the configuration descriptor. If this option is not selected no other tests for configuration can be performed.

Get Configuration:
This test issues a *`Get Configuration`* command and verifies that the Device responds with success.

Set Configuration:
This test issues a *`Get Configuration`* command. This initially unconfigures the Device. Then all configuration tests are performed for each configuration. This causes the interface and endpoint tests to be performed for every configuration on the Device.

Check Configuration Descriptor Values:
Checks all configuration descriptor members.

9.2.5. Interface Tests.

Get (Alternate) Interface:
This test issues a *Get Interface* command, which receives the alternate setting for the specified interface number. A Device without alternate interfaces should either support this command or respond with a stall, otherwise a warning is generated.

Set (Alternate) Interface:
This test issues a *Set Interface* command, which sets the alternate setting for the specified interface number. A Device without alternate interfaces should either support this command or respond with a stall, otherwise a warning is generated.

9.2.6. Endpoint test.

Get Status:
Gets status from the endpoint. If *Set Feature* or *Clear Feature* options are selected the endpoint is verified with endpoint status *ENDPOINT_HALT* bit.

Set Feature:
This test issues a *Set Feature Stall* command. This test is run on interrupt and bulk endpoints only. If the *Get Status* option is selected, the *ENDPOINT_HALT* bit should be set, and this will be verified with a *Get Status* request.

Clear Feature:
This test issues a *ClearFeature Stall* command. This test is run on interrupt and bulk endpoints only. If the *Get Status* option is selected, the *ENDPOINT_HALT* bit should be reset, and this will be verified with a *Get Status* request.

9.2.7. Other Tests.

Get String Descriptors
If this option is specified, the string corresponding to the string descriptor will be acquired for the Device, along with configuration and interface descriptors.

Perform remote wakeup
This option is not supported by this Driver version. It may be supported in later versions of Driver.

To start automatic testing click “Start” button.
To cancel test click “Stop” button.
To select all options click “Select All” button.
To deselect all options click “Deselect All” button.

9.3. Automatic Standard Requests Results.

The following diagram shows the result of the complete automatic test.

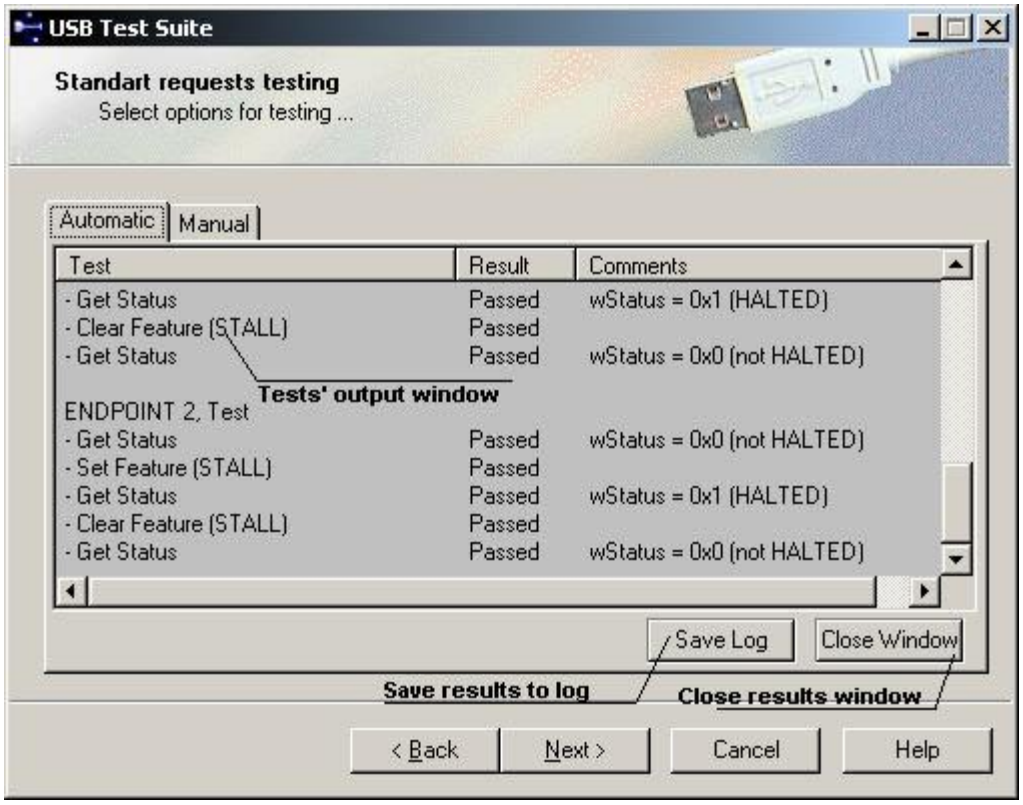

Fig 9.3 Standard requests (Automatic) results.

The user can choose where to save results to a log file, by clicking “Save Log” button or to close the results window. The user may also opt to select the “Next” button, to keep the result window active and go on to next test.

9.4. Manual Testing

In this page the user can perform Standard requests manually and see the result in the output log window.

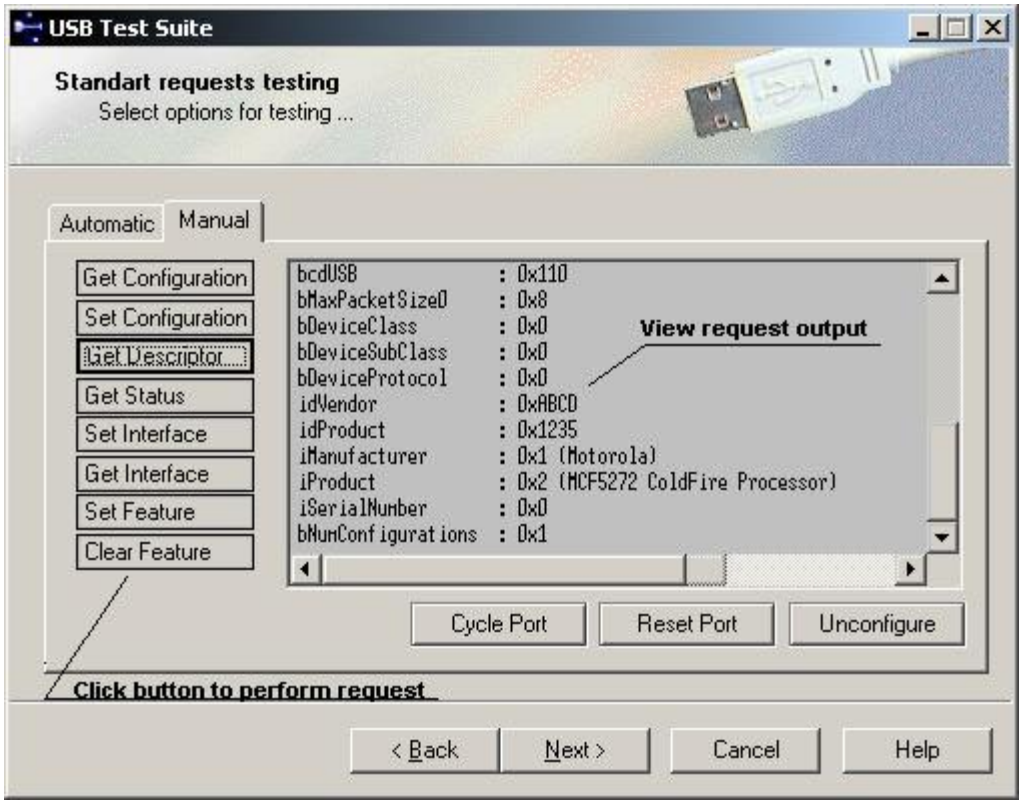

Fig 9.4 Manual requests page.

The names of the buttons on this page correspond to the names of requests. The following sections describe the requests a user can invoke from this page:

9.4.1. Get Configuration.

The current configuration value should appear in the output log window.

9.4.2. Set Configuration

The following dialog should appear.

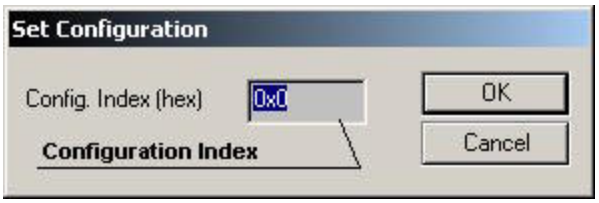

Fig 9.5 Set Configuration Dialog.

The user can specify the index of the configuration descriptor for the configuration desired. When the user submits a request, the Test Suite attempts to set the configuration and outputs results to the log window.

9.4.3. Get Status.

The following dialog should appear.

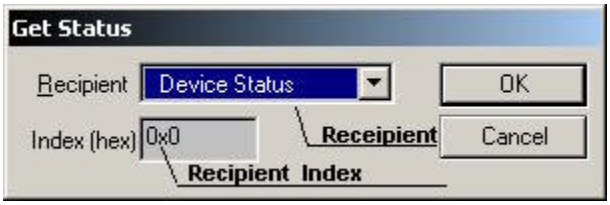

Fig 9.6 Get Status Dialog.

The user can specify the recipient and recipient index. When the user submits a request, the Test Suite attempts to set the configuration and outputs results to the log window.

9.4.4. Set Feature.

The following dialog should appear.

Fig 9.7 Set Feature Dialog.

The user can specify the recipient, recipient index and feature selector. When the user submits a request, the Test Suite attempts to set the configuration and outputs results to the log window.

9.4.5. Clear Feature.

The following dialog should appear.

Fig 9.8 Get Feature Dialog.

The user can specify the recipient, recipient index and feature selector. When the user submits a request, the Test Suite attempts to set the configuration and outputs results to the log window.

9.4.6. Get Interface.

When the user selects the *Get Interface* button, the *Get Interface* window appears as shown in Figure 9.9. The user must specify the interface number to which the program will retrieve the alternate setting. The success or failure of the command is displayed in an output window.

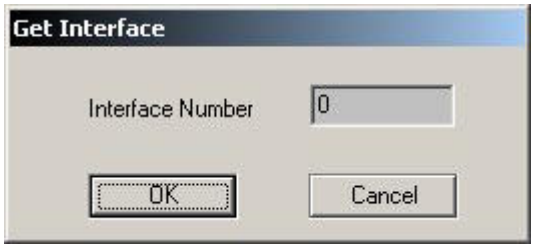

Fig 9.9 Get Interface Dialog.

9.4.7. Set Interface.

When the user selects the *Set Interface* button, the *Set Interface* window appears as shown in Figure 9.10. The user must specify the interface number to which the program will retrieve the alternate setting. The success or failure of the command is displayed in an output window.

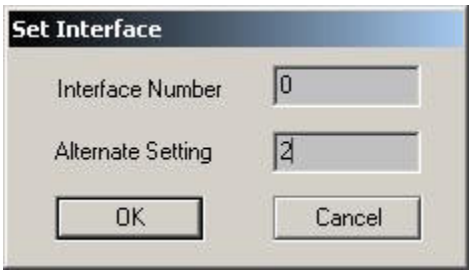

Fig 9.10 Set Interface Dialog.

9.5. File Transfer Firmware Testing.

This test appears only on "File Transfer Device" (i.e. with VendorID = 0xABCD, ProductID = 0x1235). The test procedure performs continuous file transfers with various file length and maximum transfer length parameters. A random file is generated, with a random name, which is written to the Device. It then reads the file from the Device and compares source file with destination file. In addition the test tracks the directory structure and can verify it. When the Device returns memory allocation errors on writing a file, the test removes all files it created (deleting phase) and continues write/read/verify sequences.

9.5.1. Algorithm description.

At the start of this test all files from the Device are removed. The test then prepares a directory on the PC, where all files will be located. This directory is located in the following path:

%TEMP%\usbtest\<Device_instance_number>,

where the *Device_instance_number* is equal to hexadecimal testing Device instance address. Using such a path, the Test Suite can test more that a single File Transfer Device, because of unique files location for each of the Devices under test.

The test consists of the following stages:

1.Write Phase.

The test procedure generates a file on the PC according to given file boundaries parameter, and then sends (generating transfer length by give transfer boundaries parameter) this file to the Device. If the Device returns one of following errors:

- UFTP_NO_POSITION_FOR_NEW_FILE
- UFTP_NOT_ENOUGH_SPACE_FOR_FILE
- UFTP_MEMORY_ALLOCATION_FAIL

the test procedure goes to point (3). Following successful completion of the point (3) stage, the test then tries to write a file and if an error occurs, an error message will be shown and test will be stopped. If all the transactions in this stage result in success, the

procedure goes to stage 2 or the test completes (in case of all specified file and transfer parameter values have already been generated). In the case of test completion, all files will be removed from the Device.

2. Read/Verify Phase.

If the "Check directory on each write" option is set, the Device directory content is verified with the directory content on the PC. If these contents are not the same, an error message appears in output window. If the Device returns *UFTP_MEMORY_ALLOCATION_FAIL* during directory read from the Device, these tests will be skipped and the test procedure will print out warning message. The test then reads a newly created file in stage 1 and verifies that this is the same file on the PC. If file contents are not identical an error message will again be generated and will appear in the output window. On successful completion on this stage, the procedure goes to stage 1.

3. Deletion Phase.

This test procedure removes all files on the Device. If the 'file must exist on delete' option is set, the test procedure assumes that a file for deletion must exist on the Device. If this condition fails, the test procedure shows an error. If the 'Check content on delete' option is set, the test procedure reads the file from the Device and verifies it's content with the contents of same file on the PC, before deletion. If the file content is different, the test procedure stops and shows an error message dialog.

9.5.2. Transfer Testing Page.

As mentioned above, if the File Transfer Device is selected for testing, the following Test Suite page appears.

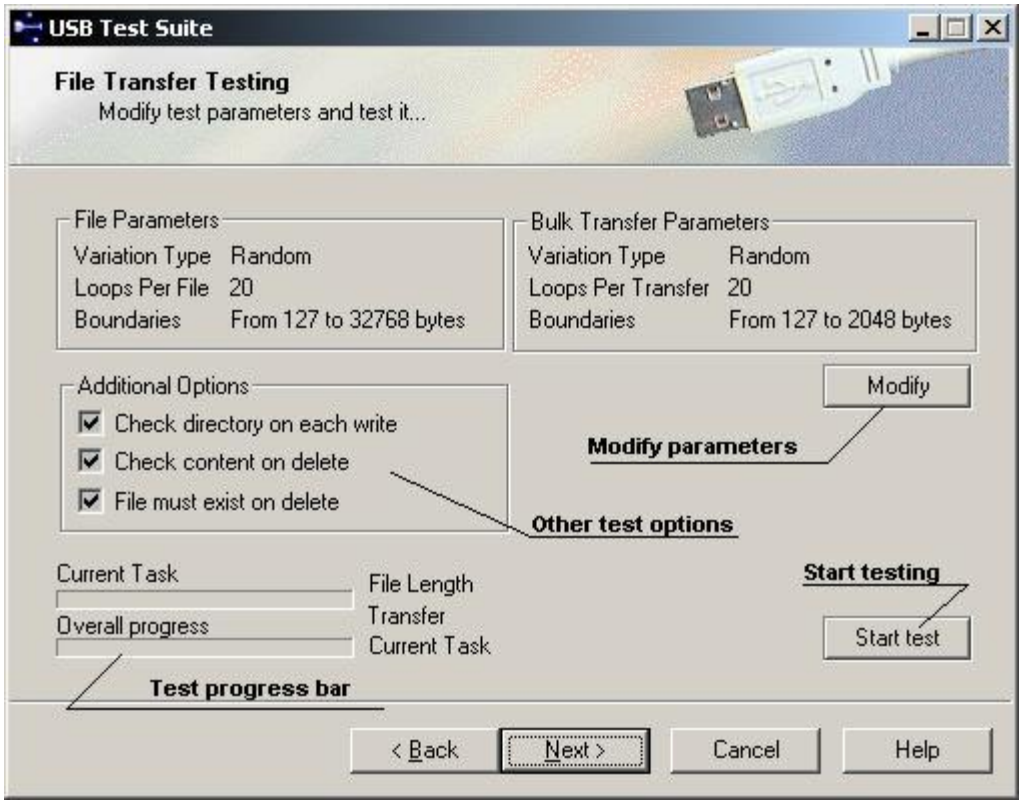

Fig 9.11 File Transfer Page.

Before starting this test the default setting can be modified, in order to specify more detailed test parameters.

Additional test options:

- **Check directory on each write** - verifies directory after each new file written
- **Check content on delete** - verifies file content when entering deleting phase
- **File must exist on delete** - verifies each written file exists on deleting phase

The user can modify file transfer test parameters by clicking the “Modify” button. The following dialog should then appear:

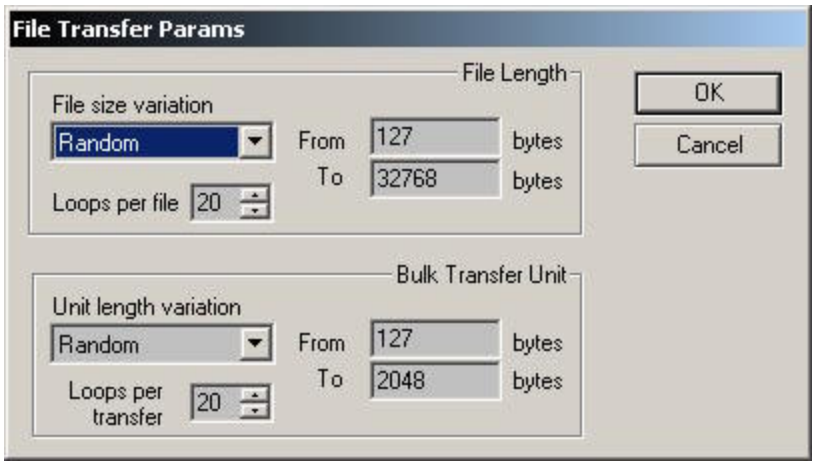

Fig 9.12 File Transfer Test Parameters.

The following variation types can be selected for file and transfer:

Linear: The parameter variation will be incremented on each test stage until "To" value reached.

Fixed: No variation for parameter. The parameter remains fixed for all test stages.

Random: Random variation for the parameter, which can vary in the range ["From","To"] values;

Loops per file – perform several write/read/verify sequences for each file length
Loops per transfer – perform several write/read/verify sequences for transfer size

Clicking the «Start» button will start the test. The output window and log errors are shown in this window. The user also has the ability to cancel the test. After the test terminates (by canceling or ending) the user can save the stest results to a log file.

NOTE: It's recommended to minimize count of running processes, which active use CPU while transfer testing stage. It should get better testing transfer speed, so can minimize transfer testing time. The list of such processes can be acquired from Task Manger "Processes" page.

9.6. Isochronous Transfers Testing.

This test appears only on "USB Audio" firmware (i.e. with VendorID = 0xABCD, ProductID = 0x1236). In case of selected "USB Audio" firmware Device the following page should appear.

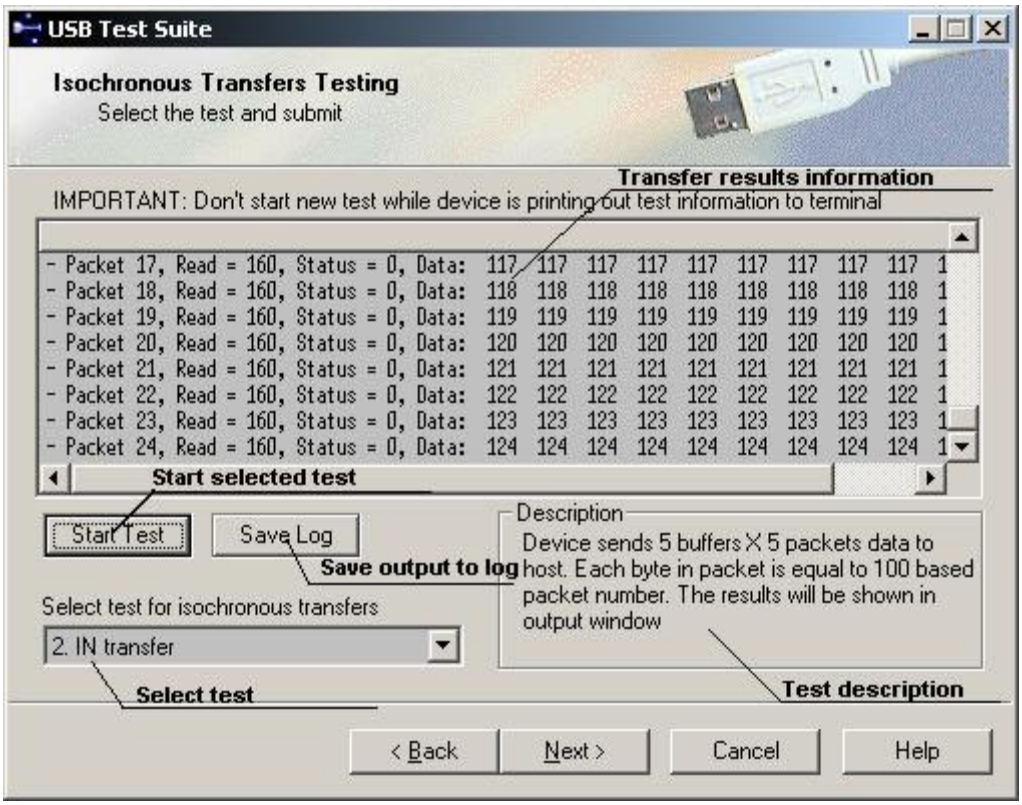

Fig 9.13 Isochronous Transfers Test Page.

The test consists of 6 isochronous transfers test procedures, which covers most of firmware isochronous transfers processing.

9.6.1. Tests Description.

1. OUT Transfer

Host sends 5 buffers X 5 packets of data to the Device. Each byte in a packet is equal to zero based packet number. Device prints output results to the terminal.

2. IN Transfer

Device sends 5 buffers X 5 packets data to the Host. Each byte in a packet is equal to 100 based packet number. The results will be shown in the output window.

Simultaneous IN/OUT transfers
This test sends data from Host to Device and from Device to Host at the same time. Both transfers consist of 5 buffers X 5 packets. The data transmitted to the Device consists of each byte in packet equal to zero based packet number. The Device returns the data from the Host with 1 buffer (for Stand-Alone firmware) or 2 buffers (for uClinux firmware) delay. Delayed buffers will be filled by the Device with 100 based packet number. Other packets should contain information transmitted from the Host.

In all of tests above (1-3) all packets with Status = 0 expected. Short Packets should be assumed as abnormal Device behavior.

The other 3 tests are provided to test how the Device process missing frames (Host does not send IN or/and OUT tokens to isochronous pipes). Normally the Device should be able to process such cases, and continue working in real time when tokens from Host appear on the USB. Note that in some cases when the Host starts to send tokens, short packets buffer (with Status = 0x9) should be expected as normal.

4. OUT transfer (with missing frames)
This test is the similar to the OUT transfer test (1) with missing OUT tokens simulation. OUT tokens for packets #8, #9, #10, #15, #19 are missed (i.e. not sent by the Host).

5. IN transfer (with missing frames)
This test is the similar to the OUT transfer test (2), with missing IN tokens simulation. IN tokens for packets #7, #10, #14, #15 are missed (i.e. not received by the Host).

6. Simultaneous IN/OUT (with missing frames)
This test is similar to the Simultaneous IN/OUT transfers test (3), with missing IN tokens simulation. IN and OUT tokens for packet #6 are missed (i.e. not sent and received by the Host)

In all of above tests (3—6) some packets (located closely to missed frame packets) with Status = 0x9 are expected. Short Packets can be assumed as normal Device behavior.

9.6.2. Performing Tests.

It is important not to overlook the connection of the evaluation board with the terminal cable in order to see the Device output. Select the test from test selecting combo box. Submit the ‘Start Test’ button, check the results in the output window, and on the terminal (Device output). Test results can also be saved to the log file. Please note that the Test Suite outputs only the first 30 bytes for each packet, so the results in the output window and the log file will be somewhat truncated. However the rest of the packet information (in the case of test success) should be the same.

NOTE: No new test should be started while the Device is printing out test information to the terminal. If this should occur by accident, the Device firmware should be restarted in order to restore correct working.

9.6.3. Other tests.

The following page performs testing of cases, with different file length and maximum transfer length parameters.

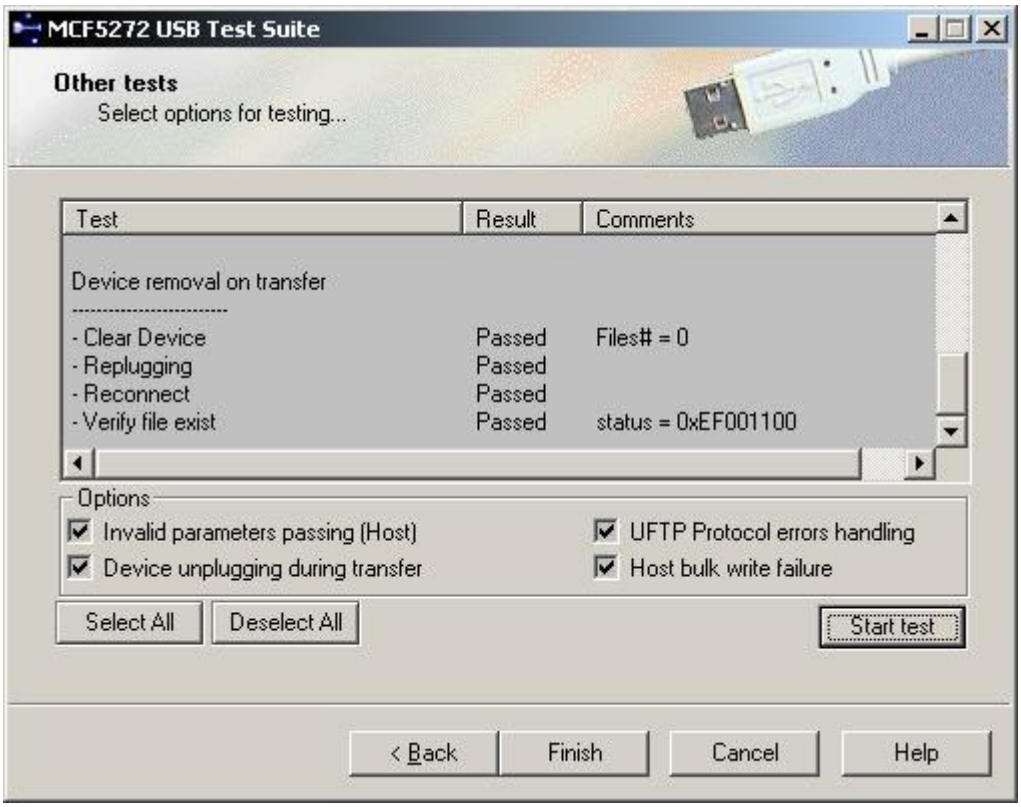

Fig 9.14 Other tests page.

- **Invalid parameters passing**

Tests invalid parameters passing (this test is for Host software only).

- **Device unplugging during transfer (on File Transfer Device)**

Tests Device unplugging during transfer.

- **UFTP Protocol errors handling (on File Transfer Device)**

This test is not valid in this version of the Test Suite, but may be provided in later versions.

- **Host bulk write failure (on File Transfer Device)**

Test for bulk write failure. To start this test click start button.

10. Appendix 4: USB FILE TRANSFER LIBRARY.

10.1. Introduction.

10.1.1. System Requirements .

Hardware platforms:

- Single CPU Intel i386 based PC with Open Host Controller or Universal Host Controller.

Operation systems:

- Windows 2000 Professional

Developer software tools:

- Visual C++ 6.0 Professional Edition
- Microsoft Platform SDK for Windows 2000 (Recommended)

10.1.2. UFTP library content.

Location:

\inc

uftp.h	– library header file
progress.h	– transfer progress routine header file

\lib

uftp.lib	– static library

Additional to link with UFTP library application must be linked with *motusb.lib* file in the **\lib** directory. The UFTP library depends from *motusb.dll*.

10.2. Programming interface

The UFTP library provides the programming interface for communication with a UFTP Device. The USB file transfer protocol defines the following set of requests:

Table 10.1 UFTP requests.

UFTP_READ	Read file from Device to Host
UFTP_WRITE	Write file from Device to Host
UFTP_GET_FILE_INFO	Retrieve file information
UFTP_GET_DIR	Retrieve directory structure
UFTP_SET_TRANSFER_LENGTH	Set maximum transfer length
UFTP_DELETE	Delete file from the Device

The library encapsulates these requests into C language functions. In this way the library provides the simplest way to communicate with the UFTP Device. The library use handles to track the request from different thread contexts. The Client should connect the UFTP interface and receive a handle (**HUFTP**) in order to perform any library operation.

Table 10.2 Functions Summary.

Function Name	**Description**
Uftp_Connect	Connect the UFTP interface
Uftp_Disconnect	Disconnect the UFTP interface
Uftp_SetProgressRoutine	Set the progress routine
Uftp_SendFile	Send file from Host to Device
Uftp_GetFile	Send file from Device to Host
Uftp_GetFileInfo	Get file information about particular file on Device
Uftp_ReadDir	Get directory information
Uftp_SetTransferLength	Set maximum transfer length
Uftp_DelFile	Delete file from Device
Uftp_GetLastError	Get the last UFTP error
Uftp_GetErrorText	Get the string message for specified UFTP error

10.2.1. Function Descriptions.

10.2.1.1. Uftp_Connect

Definition:

```
HUFTP
Uftp_Connect(
   usb_t Device
  );
```

Parameters:

Device – handle to MOTUSB Device object.

Returns:

The function returns handle to UFTP object or NULL if UFTP interface cannot be found on the Device.

Comments:

The function establishes connection with UFTP object. Once the connection is established to the Client, it can perform the required UFTP operations using this handle. When the handle is no longer needed, the Client should use the `Uftp_Disconnect` routine to disconnect the UFTP object handle. The UFTP object maintains information on the CBI endpoint configuration, I/O operation performed, Device locking state along with other data.

10.2.1.2. Uftp_Disconnect

Definition:

```
void
Uftp_Disconnect(
   HUFTP hUftp
  );
```

Parameters:

hUftp – handle to UFTP object obtained from the *`Uftp_Connect`* routine.

Returns:

None.

Comments:

The function breaks the connection with the UFTP object. The UFTP Client should use this routine when the connection is no longer used.

10.2.1.3. Uftp_SetProgressRoutine

Definition:

```
BOOL
Uftp_SetProgressRoutine(
   HUFTP                hUftp,
   PROGRESS_ROUTINE     progressRoutine,
   LPVOID               param
 );
```

Parameters:

hUftp – handle to UFTP object obtained from the `Uftp_Connect` routine.

ProgressRoutine – pointer to transfer progress callback routine. Caller can specify NULL, meaning that no progress routine should be called when the library performs transfer operations.

param – miscellaneous parameter passed to progress routine, when it is called. If caller specifies NULL for progress routine, it must specify NULL for this parameter also.

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the `GetLastError` Win32 API function.

Comments:

The function attaches progress routine to the specified UFTP object. The progress routine is called by `Uftp_SendFile` or `Uftp_GetFile` routines as callback for catching transfer progress notifications. The Client can specify a zero value to `param` and `progressRoutine` parameters to detach the progress routine from the UFTP object.

10.2.1.4. Uftp_SendFile

Definition:

```
BOOL
Uftp_SendFile(
   HUFTP      hUftp,
   LPCTSTR  PathName
  );
```

Parameters:

hUftp – handle to UFTP object obtained from `Uftp_Connect` routine.

PathName – specifies full path to the file on the Host, which should be transmitted to the Device.

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the `Uftp_GetLastError` Win32 API function.

Comments:

This function sends file from the Host specified by path name `PathName` to the Device. If any progress routine is attached to the UFTP object, it will be invoked on each transfer unit transmitted. The following UFTP request will appear on the bus:

- UFTP_SET_TRANSFER_LENGTH
- UFTP_WRITE

The function locks the Device for the following request while transmitting the file, thus several applications can request send file operation.

10.2.1.5. Uftp_GetFile

Definition:

```
BOOL
Uftp_GetFile(
   HUFTP   hUftp,
   LPCTSTR DestFileName,
   LPCTSTR SrcFileName
 );
```

Parameters:

hUftp – handle to UFTP object obtained from `Uftp_Connect` routine.

DestFileName – specifies full path to the file on the Host, which should be transmitted from the Device.

SrcFileName – specifies requested file name from the Device.

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the `Uftp_GetLastError` Win32 API function.

Comments:

This function sends a file from the Device specified by file name `SrcFileName` to the Host file, specified by full file path `DestFileName`. If any progress routine is attached to the UFTP object, it will be invoked on each transfer unit transmitted.
The following UFTP requests will appear on the bus:

- UFTP_GET_FILE_INFO
- UFTP_SET_TRANSFER_LENGTH
- UFTP_READ

The function locks the Device for following request while transmitting the file, thus several application can request get file operation.

10.2.1.6. Uftp_GetFileInfo

Definition:

```
BOOL
Uftp_GetFileInfo(
   HUFTP              hUftp,
   LPCTSTR            FileName,
   UFTP_FILE_INFO* fileInfo
  );
```

Parameters:

hUftp – handle to UFTP object obtained from **Uftp_Connect** routine.

FileName – specifies requested file name from the Device

FileInfo – points to the buffer to return file information

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the *`Uftp_GetLastError`* Win32 API function.

Comments:

This function requests information about the file on the Device specified by file name. The information about the requested file is placed into a buffer pointed to by the *`fileInfo`* parameter. The following UFTP requests will appear on the bus:

- UFTP_GET_FILE_INFO

The function locks the Device for the following request while transmitting the file, thus several applications can request the get file operation.

10.2.1.7. Uftp_ReadDir

Definition:

```
BOOL
Uftp_ReadDir(
   HUFTP   hUftp,
   TCHAR **rgFileNames[],
   DWORD  *dwFilesCount
  );
```

Parameters:

hUftp – handle to UFTP object obtained from *`Uftp_Connect`* routine.

rgFileNames – array with string, that contain file names.

dwFilesCount – points to the buffer to return total file count on Device.

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the *`Uftp_GetLastError`* Win32 API function.

Comments

This function requests an information directory for the Device. The function allocates an array of strings *`rgFileNames`* and puts the file name into this array. The array contains *`dwFilesCount`* valid entries. The caller should **free** each file name, and then **free** *`dwFilesCount`* itself, when the file list array is no longer required. The following UFTP requests will appear on the bus:

- UFTP_GET_DIR

The function locks the Device for the following request while transmitting the file, thus several applications can request the get file operation.

10.2.1.8. Uftp_SetTransferLength

Definition:

```
BOOL
Uftp_SetTransferLength(
   HUFTP    hUftp,
   DWORD  dwTransferLength
  );
```

Parameters:

hUftp – handle to UFTP object obtained from *`Uftp_Connect`* routine.

dwTransferLength – transfer unit to use in I/O operations.

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the *`Uftp_GetLastError`* Win32 API function.

Comments:

This function sets the transfer unit to communicate with the Device. The bigger transfer unit increase transmission speed but requires more memory on the Device. With this function no UFTP requests will appear on the bus.

10.2.1.9. 2.1.9 Uftp_DelFile

Definition:

```
BOOL
Uftp_DelFile(
    HUFTP hUftp,
    LPCTSTR FileName
  );
```

Parameters:

hUftp – handle to UFTP object obtained from *`Uftp_Connect`* routine.

FileName – specifies file name on the Device

Returns:

TRUE if operation completes successfully. FALSE if any error occurred. To get extended error information the Client should call the *`Uftp_GetLastError`* Win32 API function.

Comments:

This function deletes the file specified by *`FileName`* from the Device. If no file with such a name exists, the Device returns UFTP error. The following UFTP requests will appear on the bus:

- UFTP_DELETE

The function locks the Device for the following request.

10.2.1.10. Uftp_GetLastError

Definition:

```
DWORD
Uftp_GetLastError(
    HUFTP hUFTP
  );
```

Parameters:

hUftp – handle to UFTP object obtained from *`Uftp_Connect`* routine.

Returns:

The function returns the last error. This can be either a UFTP error, a MOTUSB error or a system error.

Comments:

The function returns the last error for the specified UFTP object handle. This can be UFTP, MOTUSB, or system error.

10.2.1.11. Uftp_GetErrorText

Definition:

```
LPTSTR
Uftp_GetErrorText(
    DWORD  errCode
  );
```

Parameters:

errCode – error code returns by UFTP operation.

Returns:

The function returns a string error message for the specified error code.

Comments:

The Client application can use this operation to get a uftp error message for the specified error code.

10.2.2. Types used in library.

10.2.2.1. PROGRESS_ROUTINE

Definition:

typedef void (***PROGRESS_ROUTINE**)(**PPROGRESS_STRUCT**);

10.2.2.2. PROGRESS_STRUCT

Definition:

```
typedef struct {
   BYTE      eventCode;
   LPVOID    param;
   DWORD     timeMs;
   DWORD     bytesDone;
   DWORD     bytesTotal;
} PROGRESS_STRUCT, *PPROGRESS_STRUCT;
```

Members:

eventCode

Can be one of the following constant:

EVENT_START - operation starts
EVENT_STOP - operation stops
EVENT_UPDATE - operation has progress

param

Misc. parameter that the Client specifies in *`Uftp_SetProgressRoutine param`* parameter.

timeMs

Time in milliseconds since transfer operation started.

bytesDone

Transferred bytes count.

BytesTotal

Total bytes count to transmit.

Comments:

This structure is used by the progress callback routine. The UFTP library calls the progress routine when it has to transfer results, and puts the pointer to this structure as a parameter. In this way the UFTP Client by using *`Uftp_SetProgressRoutine`* can retrieve notifications of progress results.

10.2.3. Error codes.

To obtain an error code, the Client should use the *Uftp_GetLastError* routine. The library defines the following UFTP error codes:

UFTP_SUCCESS	0
UFTP_FILE_DOES_NOT_EXIST	0xEF001100L
UFTP_MEMORY_ALLOCATION_FAIL	0xEF002100L
UFTP_NO_POSITION_FOR_NEW_FILE	0xEF003100L
UFTP_NOT_ENOUGH_SPACE_FOR_FILE	0xEF004100L

All these error codes are defined by the UFTP protocol. Check USB CBI Transfers Type Client Application Developers Guide for a description of these values. The library can also return any MOTUSB or System error. The Client should use *Uftp_GetErrorText* function to get the error message of any error type.