



Ver2.0 - Sep 16, 2003

CEUSB2 API SPECIFICATIONS

CeUsb2 Application Programming Interface
Specifications for Software Programmers

CONTENTS

| | |
|--|-----------|
| CONTENTS | 1 |
| 1 INTRODUCTION | 4 |
| 1.1 About This Documentation..... | 4 |
| 1.2 Prerequisites | 4 |
| 1.3 Distribution | 5 |
| 1.4 Overview | 5 |
| 2 USING CEUSB2 API | 5 |
| 2.1 Enumerating CeUsb2 devices..... | 6 |
| 2.2 Opening and Closing CeUsb2 devices | 7 |
| 2.3 Retrieving versioning information..... | 8 |
| 2.4 USB Descriptor Handling | 10 |
| 2.5 USB Configuration and Interface Handling | 13 |
| 2.6 Using UsbInterface class | 17 |
| 2.7 USB Vendor and Class Requests | 18 |
| 2.8 Controlling the FX2 CPU..... | 20 |
| 2.9 Controlling the GPIF | 21 |
| 2.10 Data transfer with Bulk Pipes..... | 22 |
| 2.11 Data transfer with Isochronous Pipes | 24 |
| 2.12 Data transfer with Isochronous streaming..... | 26 |
| 2.13 FPGA configuration..... | 28 |
| 2.14 RS232 – Serial interface handling..... | 30 |
| 2.15 Other USB operations..... | 31 |
| 2.16 Error handling | 32 |
| 3 CEUSB2 API REFERENCE | 34 |
| 3.1 CeUsb2Ld class..... | 34 |
| 3.1.1 CeUsb2Ld::CeUsb2Ld | 35 |
| 3.1.2 CeUsb2Ld::~~CeUsb2Ld..... | 35 |
| 3.1.3 CeUsb2Ld::Open | 35 |
| 3.1.4 CeUsb2Ld::Close..... | 35 |
| 3.1.5 CeUsb2Ld::GetDriverInfo..... | 36 |
| 3.1.6 CeUsb2Ld::ConfigureDevice | 36 |
| 3.1.7 CeUsb2Ld::VendorRequest | 36 |
| 3.2 CeUsb2 class..... | 37 |
| 3.2.1 CeUsb2::CeUsb2..... | 37 |
| 3.2.2 CeUsb2::CeUsb2 (2)..... | 37 |
| 3.2.3 CeUsb2::CeUsb2 (3)..... | 37 |

| | | |
|--------|---------------------------------------|----|
| 3.2.4 | CeUsb2::~~CeUsb2..... | 38 |
| 3.2.5 | CeUsb2::operator= | 38 |
| 3.2.6 | CeUsb2::Open | 38 |
| 3.2.7 | CeUsb2::Close..... | 38 |
| 3.2.8 | CeUsb2::GetDeviceHandle..... | 39 |
| 3.2.9 | CeUsb2::GetLinkName | 39 |
| 3.2.10 | CeUsb2::GetFriendlyName..... | 39 |
| 3.2.11 | CeUsb2::GetDriverInfo..... | 39 |
| 3.2.12 | CeUsb2::GetUSBDError | 40 |
| 3.2.13 | CeUsb2::CancelRequests..... | 40 |
| 3.2.14 | CeUsb2::GetConfiguration..... | 40 |
| 3.2.15 | CeUsb2::SelectConfiguration..... | 40 |
| 3.2.16 | CeUsb2::GetInterface | 41 |
| 3.2.17 | CeUsb2::SelectInterface | 41 |
| 3.2.18 | CeUsb2::GetCurrentFrameNumber | 41 |
| 3.2.19 | CeUsb2::GetStatus..... | 42 |
| 3.2.20 | CeUsb2::GetPowerState..... | 42 |
| 3.2.21 | CeUsb2::SetPowerState | 42 |
| 3.2.22 | CeUsb2::FeatureRequest | 43 |
| 3.2.23 | CeUsb2::DescriptorRequest | 43 |
| 3.2.24 | CeUsb2::GetInterfaceDescriptor | 43 |
| 3.2.25 | CeUsb2::GetInterfaceInformation | 44 |
| 3.2.26 | CeUsb2::ResetDevice..... | 45 |
| 3.2.27 | CeUsb2::ResetPipe | 45 |
| 3.2.28 | CeUsb2::AbortPipe | 45 |
| 3.2.29 | CeUsb2::VendorRequest..... | 45 |
| 3.2.30 | CeUsb2::VendorOrClassRequest | 46 |
| 3.2.31 | CeUsb2::BulkRead | 47 |
| 3.2.32 | CeUsb2::BulkWrite..... | 47 |
| 3.2.33 | CeUsb2::IsoRead | 48 |
| 3.2.34 | CeUsb2::IsoWrite..... | 48 |
| 3.2.35 | CeUsb2::IsoStreamStart | 49 |
| 3.2.36 | CeUsb2::IsoStreamStop | 49 |
| 3.2.37 | CeUsb2::IsoStreamRead | 49 |
| 3.2.38 | CeUsb2::ControlRead..... | 50 |
| 3.2.39 | CeUsb2::ControlWrite | 50 |
| 3.3 | DeviceList class | 51 |
| 3.3.1 | DeviceList::DeviceList..... | 51 |
| 3.3.2 | DeviceList::DeviceList (2) | 51 |
| 3.3.3 | DeviceList::~~DeviceList..... | 51 |
| 3.3.4 | DeviceList::Init | 51 |
| 3.3.5 | DeviceList::operator[]..... | 52 |
| 3.3.6 | DeviceList::GetNumDevices | 52 |
| 3.4 | Descriptor class | 52 |
| 3.4.1 | Descriptor::Descriptor | 52 |
| 3.4.2 | Descriptor::Descriptor (2)..... | 52 |

| | | |
|--------|--|----|
| 3.4.3 | Descriptor::~Descriptor | 53 |
| 3.4.4 | Descriptor::Init..... | 53 |
| 3.4.5 | Descriptor::GetNumConfigurations | 53 |
| 3.4.6 | Descriptor::GetNumInterfaces | 53 |
| 3.4.7 | Descriptor::GetDeviceDescriptor | 54 |
| 3.4.8 | Descriptor::GetConfigurationDescriptor | 54 |
| 3.4.9 | Descriptor::GetInterfaceDescriptor..... | 54 |
| 3.4.10 | Descriptor::GetEndpointDescriptor | 54 |
| 3.4.11 | Descriptor::GetStringDescriptor | 55 |
| 3.5 | UsbInterface class | 55 |
| 3.5.1 | UsbInterface::UsbInterface | 55 |
| 3.5.2 | UsbInterface::UsbInterface (2)..... | 56 |
| 3.5.3 | UsbInterface::~UsbInterface | 56 |
| 3.5.4 | UsbInterface::Init..... | 56 |
| 3.5.5 | UsbInterface::GetInterfaceNumber | 56 |
| 3.5.6 | UsbInterface::GetAlternateSetting | 57 |
| 3.5.7 | UsbInterface::GetNumPipes | 57 |
| 3.5.8 | UsbInterface::GetPipePtr | 57 |
| 3.6 | GPIF class | 57 |
| 3.6.1 | Public Member Variables..... | 57 |
| 3.6.2 | GPIF::GPIF | 58 |
| 3.6.3 | GPIF::GPIF (2)..... | 58 |
| 3.6.4 | GPIF::~GPIF | 58 |
| 3.6.5 | GPIF::Init..... | 59 |
| 3.6.6 | GPIF::Refresh..... | 59 |
| 3.6.7 | GPIF::Restore | 59 |
| 3.6.8 | GPIF::LoadGpifInitData..... | 59 |
| 3.6.9 | GPIF::LoadGpifWaveformData | 60 |
| 3.6.10 | GPIF::GpifSingleRead | 60 |
| 3.6.11 | GPIF::GpifSingleWrite..... | 60 |
| 3.6.12 | GPIF:: GpifSingleWrite_Wordwide..... | 61 |
| 3.6.13 | GPIF:: GpifSingleRead_Wordwide | 61 |
| 3.6.14 | GPIF::GpifAbort | 61 |
| 3.6.15 | GPIF::GetGpifStatus | 62 |
| 3.7 | FpgaConfig class | 62 |
| 3.7.1 | FpgaConfig::FpgaConfig..... | 62 |
| 3.7.2 | FpgaConfig::FpgaConfig (2) | 62 |
| 3.7.3 | FpgaConfig::~FpgaConfig..... | 63 |
| 3.7.4 | FpgaConfig::Init | 63 |
| 3.7.5 | FpgaConfig::IsConfigured..... | 63 |
| 3.7.6 | FpgaConfig::Reset..... | 63 |
| 3.8 | Fpga class..... | 64 |
| 3.8.1 | Fpga::Fpga | 64 |
| 3.8.2 | Fpga::Fpga (2) | 64 |
| 3.8.3 | Fpga::~Fpga | 65 |
| 3.8.4 | Fpga::Configure | 65 |

| | | |
|--------|-----------------------------------|----|
| 3.9 | RS232 class..... | 65 |
| 3.9.1 | RS232::RS232..... | 65 |
| 3.9.2 | RS232::RS232 (2)..... | 65 |
| 3.9.3 | RS232::~~RS232..... | 66 |
| 3.9.4 | RS232::TxSend | 66 |
| 3.9.5 | RS232::RxStart..... | 66 |
| 3.9.6 | RS232::RxStop | 67 |
| 3.9.7 | RxNotify | 67 |
| 3.10 | CeUsb2 API Global Functions | 67 |
| 3.10.1 | GetApiInfo..... | 67 |
| 3.10.2 | GetFirmwareInfo | 68 |
| 3.10.3 | IsHighSpeed | 68 |
| 3.10.4 | FormatErrorString | 68 |
| 3.10.5 | FormatUSBDErrorStr..... | 69 |
| 3.10.6 | SetCpuSettings..... | 69 |
| 3.10.7 | GetCpuSettings | 69 |
| 3.10.8 | FwReinit..... | 70 |
| 3.10.9 | FwSoftReset | 70 |
| 3.11 | CeUsb2 API structures..... | 70 |
| 3.11.1 | CEUSB2_API_INFO | 70 |
| 3.12 | CeUsb2 API Macros | 71 |
| 3.12.1 | CE_SUCCESS | 71 |
| 3.12.2 | GET_NUM_CONFIGURATIONS..... | 71 |
| 3.12.3 | GET_NUM_INTERFACES..... | 71 |
| 3.12.4 | GET_NUM_ENDPOINTS | 72 |
| 3.12.5 | GET_INTERFACE_DESC_SIZE | 72 |
| 3.13 | CeUsb2 API Error Codes..... | 72 |

1 INTRODUCTION

1.1 About This Documentation

CeUsb2 API is the application programming interface for CeUsb2 compatible devices from CESYS GmbH. It has several classes and some global functions for device input output, general USB functionality, USB data transfers, firmware configuration, FPGA configuration, error handling and other features of CeUsb2 devices. Before reading this documentation it would be helpful to check general CeUsb2 design guide documentation, CeUsb2dg.pdf, to understand the components which comprise the complete CeUsb2 software.

1.2 Prerequisites

The whole CeUsb2 Application programming interface is written in C++ language using Microsoft Visual C++ 6.0. Therefore an intermediate or upper knowledge of C++ and Windows programming (Win 32 SDK) is necessary to understand the class documentations and programming examples.

CeUsb2 boards are USB2.0 devices, so it is necessary to understand basics of the USB protocol in order to understand the USB communication and data transfer operations.

1.3 Distribution

After you install CeUsb2 software, you can find the header file for the CeUsb2 API, CeUsb2Api.h, in CeUsb2\inc directory. It includes class, structure, enumeration, macro and constant definitions for the API. You will also need guids.h header file for the constant GUID definitions for the current CeUsb2 device interfaces.

API executable, CeUsb2Api.dll, is a Win 32 dynamic link library (DLL). No other technologies except basic Win32 SDK are used when developing the API, such as MFC or COM. So you should be able to link this DLL with any 32-bit Windows based project written in C++. CeUsb2Api.lib library file is also required for the linker. Both the DLL and the library file can be found in lib\fre\i386 directory (debug versions are in lib\chk\i386).

1.4 Overview

CeUsb2 application programming interface stands on the input output control (IOCTL) interface of the CeUsb2 driver. For more information about this IOCTL interface check CeUsb2 driver specification document.

It is possible for the programmers to use the IOCTL interface and write a complete project either in C or C++ languages. However the API simplifies the usage of the IOCTL interface by implementing upper level C++ classes and some global exported functions for more sophisticated error handling.

Chapter 2 explains the usage of the API with some example codes. All of these codes are tested in separate projects and most of them are ready to be used as templates.

Chapter 3 is a reference for the API. It explains the classes with their member functions, global functions, structures, enumerations, macros and error codes.

2 USING CEUSB2 API

This chapter explains the usage of the CeUsb2 API. Every subsection has some code snippets explaining the concept more detailed. Code examples uses PrintBuffer, WsPrintBuffer functions for formatted and unformatted text printing and PrintError function for error searching and printing purposes. Implementations of these functions can be found in the last section of this chapter, Error Handling. Internally all three functions use printf method of standard input output library. You can change the sources and direct the output to another type of console dialog or user specified target.

2.1 Enumerating CeUsb2 devices

Before using any device classes or functions from CeUsb2 API, available CeUsb2 devices connected to the system must be enumerated. DeviceList class enables device enumeration with a given 128 bit device interface GUID number which identifies a separate device interface for CeUsb2. For more information about device interface check CeUsb2 driver specifications document. If no GUID number is specified (2nd constructor is not used) then the default GUID for CeUsb2 interface is used for enumeration (GUID_INTERFACE_CEUSB2). GUID definitions for the supported CeUsb2 device interfaces can be found in inc\guids.h header file.

After DeviceList is initialized with the Init function, it stores the information for a device in its internal device list if it can find at least one. This list can be accessed with an index operator ([]) which returns a copy of the element at the given index. Before using the index operator be sure that you check the size of the list (i.e. number of devices found) so that you don't pass an invalid index value which references an invalid device object.

Device information is stored in the list as CeUsb2 class object, with device link name and device friendly link name strings. These two variables are enough to identify a CeUsb2 device. You can retrieve device link and friendly name with CeUsb2 class's functions GetLinkName and GetFriendlyName.

Code example 2.1 shows how to enumerate CeUsb2 devices. If any devices are found their link name and friendly names are displayed.

```
// Code example 2.1
// Device enumeration with DeviceList

// includes
#include <stdio.h> //necessary for text input output
#include <windows.h> //necessary for Win 32 type definitions and functions
#include "..\..\inc\CeUsb2Api.h" // CeUsb2 API main header file
#include <InitGUID.h> // this file is necessary for GUID macros
#include "..\..\inc\guids.h" // CeUsb2 device interface GUIDs are defined in this file

//
```

```

//function prototypes, see section Error Handling for more information about these functions
//
void PrintBuffer(char* pBuffer); // prints a formatted string
void WsPrintBuffer(char* pFormat, ...); // prints an unformatted string
// searches the reason for an error, and prints an explanation about its reason
void PrintError(CeUsb2* Dev, DWORD dwError, BOOL bSearchReason,
               BOOL bCheckUSBDError, const char* szUserDescription, ...);

UINT uiNumDevs = 0; // variable for number of devices
CeUsb2 gDevice; // device object
UINT i = 0; // used for loops
DWORD dwStatus = 0; // status code, initially 0 to indicate success

DeviceList DevList(GUID_INTERFACE_CEUSB2); // device enumeration object

dwStatus = DevList.Init(); // enumerate devices

if(!CE_SUCCESS(dwStatus))
{
    // error enumerating devices, report error, return error code
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
              "ERROR - Device enumeration error");
    return dwStatus;
}

PrintBuffer("Device enumeration is successful\n");
uiNumDevs = DevList.GetNumDevices(); // get number of devices

if(uiNumDevs == 0)
{
    // Unable to find any CeUsb2 devices in the system
    PrintBuffer("No devices found\n");
    return CEUSB2_API_STATUS_NO_DEVICES_FOUND;
}
else
{
    // found some devices
    WsPrintBuffer("%d %s found\n", uiNumDevs,( uiNumDevs == 1) ? "device": "devices");

    // display the devices' link name and friendly name
    for (i = 0; i < uiNumDevs; i++)
    {
        WsPrintBuffer("Device %d \n Link name : %s\n Friendly name : %s\n",
                      i,DevList[i].GetLinkName(),
                      DevList[i].GetFriendlyName());
    }
}

PrintBuffer("\n");
return dwStatus;

```

2.2 Opening and Closing CeUsb2 devices

After a successful enumeration if there are CeUsb2 devices connected to the system, they can be used for input / output by user mode programs. However, before any operation a CeUsb2 device must be opened with Open function of the CeUsb2 class. Open function uses CreateFile function of Win32 SDK, and stores a file handle (device handle). You can use GetDeviceHandle function to retrieve this 32 bit handle. After all processing is done the device must be closed with Close function. Close function nullifies the device handle and releases all resources used by the device. It is called also by the CeUsb2 destructor.

Code example 2.2 shows how to open and close CeUsb2 devices. It also uses the index operator of DeviceList class.

```
// Code example 2.2, cont. 2.1
// CeUsb2 Open & Close routines

UINT DevIndex = 0; // device index to open

gDevice = DevList[DevIndex]; // get the device, at the given index
dwStatus = gDevice.Open(); // open the device

if(CE_SUCCESS(dwStatus))
{
    //device opened successfully
    WsPrintBuffer("Device %u opened successfully\n", DevIndex);

    // retrieve and display device handle
    WsPrintBuffer("Handle          0x%08X\n", gDevice.GetDeviceHandle());
}
else
{
    // error opening device
    PrintError(&gDevice,dwStatus,TRUE,TRUE, "ERROR - Unable to open device");
    return dwStatus;
}

//
// do all processing here
//
gDevice.Close(); // close the device
```

2.3 Retrieving versioning information

Some versioning information for the CeUsb2 API, device driver and the firmware can be retrieved from CeUsb2Api.

GetApiInfo global function retrieves the executable name, version number and build-number of the API itself.

GetDriverInfo global function retrieves the executable name, interface version number, driver version number, build number and build type string of the CeUsb2 device driver.

GetFirmwareInfo global function retrieves the version number, device type string and the executable name of the 8051 firmware executable running on the CeUsb2 board.

IsHighSpeed function checks whether a CeUsb2 device high-speed USB2 device or not. Notice that CeUsb2 driver supports both high-speed and full-speed USB devices if the current firmware is able to handle both speed specifications. For more information check general CeUsb2 design guide, CeUsb2dg.pdf.

Code example 2.3 shows how to use GetApiInfo, GetDriverInfo, GetFirmwareInfo and IsHighSpeed functions.

```
// Code example 2.3, cont. 2.2
// Versioning information

CEUSB2_API_INFO ApiInfo; // API information object
CEUSB2_DRIVER_INFO DeviceInfo; // device (driver) information structure
CEUSB2_FIRMWARE_INFO Fw; // firmware information object
CHAR DevStr[256]; // device type string
CHAR FwStr[256]; // firmware name string
BOOL bIsHighSpeed; // is the device high-speed USB??

GetApiInfo(ApiInfo); // get API information

// display API information
WsPrintBuffer("Api Executable   %s\n", ApiInfo.ApiName);
WsPrintBuffer("Api Version      %u.%u\n", ApiInfo.MajorVersion, ApiInfo.MinorVersion);
WsPrintBuffer("Build Number    0x%08X\n", ApiInfo.BuildNumber);
PrintBuffer("\n");

// get driver information
dwStatus = gDevice.GetDriverInfo(&DeviceInfo);

if(!CE_SUCCESS(dwStatus))
{
    // error retrieving driver information
    PrintError(&gDevice, dwStatus, TRUE, TRUE,
              "ERROR - Unable to get driver information");
    return dwStatus;
}
else
{
    // display device information
    WsPrintBuffer("Driver Executable   %s\n",      DeviceInfo.Name);
    WsPrintBuffer("Interface Version  0x%04X\n", DeviceInfo.InterfaceVersion);

    WsPrintBuffer("Driver Version      %u.%u\n", DeviceInfo.MajorVersion,
                  DeviceInfo.MinorVersion);
    WsPrintBuffer("Driver Build Number  %d\n", DeviceInfo.BuildNumber);
    WsPrintBuffer("Build Type          %s\n", DeviceInfo.BuildTypeStr);
    PrintBuffer("\n");
}
}
```

```

dwStatus = GetFirmwareInfo(&gDevice,&Fw,DevStr,FwStr); // get firmware information

if(!CE_SUCCESS(dwStatus))
{
    // error retrieving firmware information
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to get firmware information");
    return dwStatus;
}
else
{
    // display firmware information
    WsPrintBuffer("Firmware Name   %s\n",FwStr);
    WsPrintBuffer("Firmware Version %u.%u\n",Fw.MajorVersion,Fw.MinorVersion);
    WsPrintBuffer("Device Type     %s\n",DevStr);
    PrintBuffer("\n");
}

// retrieve USB speed information
dwStatus = IsHighSpeed(&gDevice,&bIsHighSpeed);
if(CE_SUCCESS(dwStatus))
{
    WsPrintBuffer("Device Speed   %s\n",(bIsHighSpeed)?"High speed":"Full speed");
}
else
{
    // error retrieving USB speed information
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to get USB speed information");
}

PrintBuffer("\n");

```

2.4 USB Descriptor Handling

CeUsb2 device, like all other USB devices, report its configuration information through descriptors.

Device descriptors contain information on the device as a whole. Configuration descriptors contain a description of each individual device configuration. Interface descriptors contain a description of each individual interface defined in a configuration. Endpoint descriptors contain a description of each individual endpoint defined in an interface-alternate setting. String descriptors contain Unicode text strings. For more information about descriptors check current Windows DDK documentation from Microsoft, or general CeUsb2 software package documentation, CeUsb2.pdf.

DescriptorRequest function is used to get or to set a descriptor on a CeUsb2 device. It is a general descriptor handling function. On CeUsb2 this function is used only to retrieve device, configuration and string descriptors.

All Cesys devices have at least 5 string descriptors. The first one is the 2-byte language identifier as 0x0409. This identifier is used when retrieving the other string descriptors.

Second string descriptor is the logo of the Cesys and currently implemented as “Cesys” in the firmware.

Third string is the protocol name. This string is “USB2” for all CeUsb2 devices.

Fourth string is the device type name which is supported by this firmware. It can be something like “Generic”.

The fifth and the last one is the name of the firmware itself. For example “CeU2g001”, refers to a Cesys USB2 generic firmware with an index number with 001. This name implies that some other firmwares can support this generic type of devices.

GetInterfaceDescriptor function is used to retrieve an interface descriptor with all endpoints and alternate settings it contains. Parsing is required to get each individual endpoint descriptor.

A more convenient way for descriptor handling is using Descriptor class of the API. It simplifies the usage of CeUsb2 descriptor functions and gives a more detailed interface with separate functions to retrieve device, configuration, interface, endpoint and string descriptors.

Code example 2.4 shows how to use Descriptor class’s functions to retrieve and display all USB descriptors of a CeUsb2 device.

```
// Code example 2.4, cont. 2.3
// Descriptor handling with Descriptor class

UCHAR NumConfigurations = 0; // number of configurations
UCHAR NumInterfaces = 0; // number of interfaces a configuration has
USB_DEVICE_DESCRIPTOR DevDesc; // device descriptor
USB_CONFIGURATION_DESCRIPTOR ConfDesc; // configuration descriptor
USB_INTERFACE_DESCRIPTOR IntDesc; // interface descriptor
USB_ENDPOINT_DESCRIPTOR EndpDesc; // endpoint descriptor
UCHAR StringIndex = 1; // string descriptor index
TCHAR StrDesc[256]; // string descriptor text

Descriptor gDesc = Descriptor(&gDevice); // descriptor class object

// initializes descriptor class, retrieves and stores device descriptor, configuration descriptor,
// and first string descriptor (index 0) which defines the language id
dwStatus = gDesc.Init();

if(!CE_SUCCESS(dwStatus))
{
    // error initializing Descriptor class
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to initialize Descriptor class");
    return dwStatus;
}
```

```

// retrieve and display device descriptor
gDesc. GetDeviceDescriptor(&DevDesc);

PrintBuffer("Device descriptor\n");
WsPrintBuffer("bLength %u\nbDescriptorType %u\nbcdUSB 0x%04X\nbDeviceClass %u\n"
    "bDeviceSubClass %u\nbDeviceProtocol %u\nbMaxPacketSize0 %u\n"
    "idVendor 0x%04X\nidProduct 0x%04X\nbcdDevice 0x%04X\n"
    "iManufacturer %u\niProduct %u\niSerialNumber %u\nbNumConfigurations %u\n\n",
    DevDesc.bLength, DevDesc.bDescriptorType, DevDesc.bcdUSB,
    DevDesc.bDeviceClass, DevDesc.bDeviceSubClass, DevDesc.bDeviceProtocol,
    DevDesc.bMaxPacketSize0, DevDesc.idVendor, DevDesc.idProduct,
    DevDesc.bcdDevice, DevDesc.iManufacturer, DevDesc.iProduct,
    DevDesc.iSerialNumber, DevDesc.bNumConfigurations);

// get number of configurations
NumConfigurations = gDesc.GetNumConfigurations();

for(UCHAR c= 1;c<=NumConfigurations;c++)
{
    // retrieve and display configuration descriptor with index c
    gDesc.GetConfigurationDescriptor(c,&ConfDesc);
    WsPrintBuffer("Configuration descriptor %u\n",c);
    WsPrintBuffer("bLength %u\nbDescriptorType %u\nwTotalLength %u\n"
        "bNumInterfaces %u\nbConfigurationValue %u\niConfiguration %u\n"
        "bmAttributes 0x%02X\nMaxPower %u\n\n",
        ConfDesc.bLength, ConfDesc.bDescriptorType, ConfDesc.wTotalLength,
        ConfDesc.bNumInterfaces, ConfDesc.bConfigurationValue, ConfDesc.iConfiguration,
        ConfDesc.bmAttributes, ConfDesc.MaxPower);

    NumInterfaces = gDesc.GetNumInterfaces(c); // get number of interfaces

    for(UCHAR inf= 0; inf<NumInterfaces; inf++)
    {
        UCHAR AlternateSetting = 0;

        while(TRUE)
        {
            // get interface descriptor with configuration number c interface number inf, and
            // alternate setting AlternateSetting
            dwStatus = gDesc.GetInterfaceDescriptor(c,inf,AlternateSetting,&IntDesc);

            if(!CE_SUCCESS(dwStatus))
                break;

            // display interface descriptor
            WsPrintBuffer("Interface descriptor %u - Alternate setting %u\n",
                inf,AlternateSetting);
            WsPrintBuffer("bLength %u\nbDescriptorType %u\n"
                "bInterfaceNumber %u\nbAlternateSetting %u\n"
                "bNumEndpoints %u\nbInterfaceClass %u\n"
                "bInterfaceSubClass %u\nbInterfaceProtocol %u\n"
                "iInterface %u\n\n",
                IntDesc.bLength, IntDesc.bDescriptorType,
                IntDesc.bInterfaceNumber, IntDesc.bAlternateSetting,
                IntDesc.bNumEndpoints, IntDesc.bInterfaceClass,

```

```

        IntDesc.bInterfaceSubClass, IntDesc.bInterfaceProtocol,
        IntDesc.iInterface
    );

    // retrieve and display all endpoint descriptors this interface has
    for(CHAR ep=0;ep<IntDesc.bNumEndpoints;ep++)
    {
        dwStatus = gDesc.GetEndpointDescriptor(c,inf,AlternateSetting,ep,
        &EndpDesc);
        if(!CE_SUCCESS(dwStatus))
            continue;

        // display interface descriptor
        WsPrintBuffer("Endpoint descriptor %u, Interface %u"
            " Alternate setting %u\n",ep,inf,AlternateSetting);
        WsPrintBuffer("bLength %u\nbDescriptorType %u\n"
            "bEndpointAddress 0x%02X\nbmAttributes %u\n"
            "wMaxPacketSize %u\nInterval %u\n",
            EndpDesc.bLength, EndpDesc.bDescriptorType,
            EndpDesc.bEndpointAddress, EndpDesc.bmAttributes,
            EndpDesc.wMaxPacketSize, EndpDesc.bInterval
        );
    }
    AlternateSetting++;
}

// retrieve and display string descriptor starting with index 1. Notice that the string descriptor
// with index 0 is the language id which is retrieved by Init function of the Descriptor class
while(TRUE)
{
    lstrcpy(StrDesc,_T(""));
    dwStatus = gDesc.GetStringDescriptor(StrDesc,StringIndex,0);
    if(!CE_SUCCESS(dwStatus))
        break;

    // display string descriptor
    WsPrintBuffer("String descriptor %u : %s\n", StringIndex, StrDesc);

    StringIndex++;
}
}

```

2.5 USB Configuration and Interface Handling

CeUsb2 class has the ability to retrieve detailed information about USB configuration and interface of a CeUsb2 device. GetConfiguration returns the current configuration number. GetInterface returns the current alternate setting for a given interface in the current configuration. GetInterfaceInformation retrieves detailed information about the current interface including all transfer

pipes (control, bulk, isochronous). Interface buffer returned by `GetInterfaceInformation` functions requires parsing since it combines interface information and pipe structures.

Some `CeUsb2` devices have more than one configuration, each of which defines another behavior of the device. `SelectConfiguration` selects a configuration for operation. Selecting the configuration with index 0 puts the device in an unconfigured state.

Moreover, some configurations have more than one interface or some interfaces have one or more alternate settings. `SelectInterface` function selects an interface-alternate setting pair for the current configuration.

A more convenient way to deal with configurations and interfaces is using `UsbInterface` class of the API (See 2.6, Using `UsbInterface` class).

Code example 2.5 shows how to use `GetConfiguration`, `GetInterface` and `GetInterfaceInformation` to retrieve and display configuration and interface information of a device. Later assuming that the device has more than one configuration and interface, it selects another configuration and interface – alternate setting pair for the device.

```
// Code example 2.5, cont. 2.4
// Configuration and interface handling with CeUsb2 class

ULONG p = 0; // used for loops
UCHAR CurrentConfig = 0; // current configuration
UCHAR Interface = 0; // current interface
UCHAR AlternateSetting = 0; // current alternate setting
ULONG NumPipes = 0; // number of pipes
ULONG BytesReturned; // number of bytes returned
USB_INTERFACE_INFORMATION IntInfo; // interface information object
// buffer for interface information including all pipes
PUSB_INTERFACE_INFORMATION pPipeBuffer = NULL;

// get number of configurations
NumConfigurations = gDesc.GetNumConfigurations();

// get current configuration
dwStatus = gDevice.GetConfiguration(&CurrentConfig);
if(!CE_SUCCESS(dwStatus))
{
    // error retrieving current configuration
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable get current configuration");
    return dwStatus;
}

WsPrintBuffer("Current configuration is %u\n\n", CurrentConfig);

//get current interface information
```

```

dwStatus = gDevice.GetInterfaceInformation(
    (PVOID)&IntInfo,
    sizeof(USB_INTERFACE_INFORMATION),
    BytesReturned
);
if(!CE_SUCCESS(dwStatus))
{
    // error retrieving interface information
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable get interface information");
    return dwStatus;
}

NumPipes = IntInfo.NumberOfPipes;
Interface = IntInfo.InterfaceNumber;
AlternateSetting = IntInfo.AlternateSetting;

PrintBuffer("Current interface information\n");
WsPrintBuffer("InterfaceNumber %u\nAlternateSetting %u\nClass %02X\nSub class %02X\n"
    "Protocol %02X\nInterfaceHandle %08X\nNumber of pipes %u\n\n",Interface,AlternateSetting,
    IntInfo.Class, IntInfo.SubClass, IntInfo.Protocol, IntInfo.InterfaceHandle, NumPipes);

if(NumPipes)
{
    // determine the size of the pipe buffer
    ULONG Size = sizeof(USB_INTERFACE_INFORMATION) +
        ((NumPipes-1)*sizeof(USB_PIPE_INFORMATION));

    // allocate memory for pipe buffer
    pPipeBuffer = (PUSB_INTERFACE_INFORMATION)malloc(Size);
    if(!pPipeBuffer)
        return CESUB2_API_STATUS_NOT_ENOUGH_MEMORY;
    ZeroMemory(pPipeBuffer,Size);

    // one more get interface information, but this time with all pipes
    dwStatus = gDevice.GetInterfaceInformation(pPipeBuffer, Size, BytesReturned);
    if(!CE_SUCCESS(dwStatus))
    {
        // error retrieving interface information
        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - Unable get interface information");
        free(pPipeBuffer); // free pipe buffer
        return dwStatus;
    }

    // display pipe information
    for( p = 0; p < NumPipes; p++)
    {
        // get pipe information pointer
        PUSB_PIPE_INFORMATION pPipe =
            ((PUSB_PIPE_INFORMATION)&pPipeBuffer->Pipes[0]) + p;

        // get pipe type string
        char Temp[32];
        switch(pPipe->PipeType)
        {
            case _UsbdPipeTypeControl:strcpy(Temp,"Control");break;

```

```

        case _UsbdPipeTypeIsochronous:strcpy(Temp,"Isochronous");break;
        case _UsbdPipeTypeBulk:strcpy(Temp,"Bulk");break;
        case _UsbdPipeTypeInterrupt:strcpy(Temp,"Interrupt");break;
        default:strcpy(Temp,"Unknown");break;
    }

    // diplay pipe information
    WsPrintBuffer("Pipe %u\n",p);
    WsPrintBuffer("MaximumPacketSize %04X\nEndpointAddress %02X\n"
        "Interval %02X\nPipe type %s\nPipeHandle %08X\n"
        "MaximumTransferSize %08X\nPipeFlags %08X\n\n",
        pPipe->MaximumPacketSize, pPipe->EndpointAddress ,
        pPipe->Interval , Temp, pPipe->PipeHandle ,
        pPipe->MaximumTransferSize, pPipe-> PipeFlags
    );

}

    free(pPipeBuffer); // free pipe buffer
}

// get current alternate setting (for testing only, it is already retrieved by GetInterfaceInfo function)
dwStatus = gDevice. GetInterface (Interface,&AlternateSetting);
if(!CE_SUCCESS(dwStatus))
{
    // error retrieving current alternate setting
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable get current alternate setting");
    return dwStatus;
}

// assuming that you have more than one configuration and second configuration's first interface has
// more than one alternate setting and selecting configuration 2 (configurations are indexed starting from 1),
// interface 0 and alternate setting 0
PrintBuffer("Selecting configuration 2, interface 0, alternate setting 1\n\n");
dwStatus = gDevice. SelectConfiguration(2);
if(!CE_SUCCESS(dwStatus))
{
    // error selecting configuration
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable select configuration");
    return dwStatus;
}

#define MAX_SIZE      20*1024 //pipes' maximum transfer size
ULONG pMaxTransferArray[6] = { MAX_SIZE , MAX_SIZE , MAX_SIZE ,
    MAX_SIZE , MAX_SIZE , MAX_SIZE };

dwStatus = gDevice. SelectInterface (0,1, pMaxTransferArray,6);
if(!CE_SUCCESS(dwStatus))
{
    // error selecting interface-alternate setting
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable select interface-alternate setting");
    return dwStatus;
}

```

2.6 Using UsbInterface class

UsbInterface class is an upper level class for interface and pipe handling. It is useful to use it before performing any USB data transfers.

Unlike lower level CeUsb2 interface functions, UsbInterface class doesn't have the ability to change any USB configuration, interface or alternate setting. However, it is more useful for data transfer operations since it retrieves the information about the valid USB pipes defined in the device, with their supported features.

Before any other functions of the UsbInterface class are used, Init member function must be called, and must succeed. It asks the driver for the current interface and retrieves the information about all valid pipes defined in the firmware and puts them into an array as CEUSB2_PIPE_INFO structure objects. GetNumPipes member function retrieves the number of pipes the current interface has. This function should be used before accessing the pipe array. GetPipePtr function returns a pointer to the pipe object with a given index in the array.

Code example 2.6 shows how to use the UsbInterface class for interface and pipe handling. It also displays the features of the pipes it finds.

```
// Code example 2.6, cont. 2.5
// Interface and pipe handling with UsbInterface class

UsbInterface Int(&gDevice); // UsbInterface class object
dwStatus = Int.Init(); // initialize UsbInterface class

if(!CE_SUCCESS (dwStatus))
{
    // error initializing UsbInterface class
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to initialize UsbInterface class");
    return dwStatus;
}

NumPipes = Int. GetNumPipes(); // get number of pipes

// display interface information
WsPrintBuffer("Interface number %u\n", Int.GetInterfaceNumber());
WsPrintBuffer("Alternate setting %u\n", Int.GetAlternateSetting());
WsPrintBuffer("Number of pipes %u\n\n", NumPipes);

for (p = 0; p< NumPipes; p++)
```

```

{
    // get pipe pointer at index p
    PUSB_D_PIPE_INFORMATION pPipe = Int.GetPipePtr(p);

    // get pipe type string
    char Temp[32];
    switch(pPipe->PipeType)
    {
    case _UsbdPipeTypeControl: strcpy(Temp, "Control"); break;
    case _UsbdPipeTypeIsochronous: strcpy(Temp, "Isochronous"); break;
    case _UsbdPipeTypeBulk: strcpy(Temp, "Bulk"); break;
    case _UsbdPipeTypeInterrupt: strcpy(Temp, "Interrupt"); break;
    default: strcpy(Temp, "Unknown"); break;
    }

    // display pipe information
    WsPrintBuffer("Pipe %u\n", p);
    WsPrintBuffer("MaximumPacketSize %04X\nEndpointAddress %02X\n"
        "Interval %02X\nPipe type %s\nPipeHandle %08X\n"
        "MaximumTransferSize %08X\nPipeFlags %08X\n\n",
        pPipe->MaximumPacketSize, pPipe->EndpointAddress ,
        pPipe->Interval , Temp, pPipe->PipeHandle ,
        pPipe->MaximumTransferSize, pPipe->PipeFlags
    );
}

```

2.7 USB Vendor and Class Requests

VendorOrClassRequest function of CeUsb class is used to send a vendor or class specific request to a device, interface, endpoint or another device defined target.

VendorRequest function of CeUsb2 class can send only vendor requests to the device but not to another target. However in most cases this function is enough, since most of the control data transfer over CeUsb2 are vendor requests which are directed to the device itself.

All CeUsb2 firmwares use the generic firmware interface (for more information check CeUsb2gfw.pdf). This interface is defined in inc\CeFirmware.h file, which includes generic vendor request codes and their specifications. If you have a specific firmware from Cesium, then you should check its documentation and see if it defines new vendor commands.

Code example 2.7 shows how to perform USB vendor request through CeUsb2 devices. Error checking is minimized for this example to make the code shorter.

```

// Code example 2.7, cont. 2.6
// Vendor command

```

```

// imaginary vendor commands which are implemented in this specific firmware
// CEUSB2_CMD_GPIF_SINGLE_READ_WRITE is implemented in CeFirmware.h

#define INIT_DATA_TRANSFER          0xC0
#define STOP_DATA_TRANSFER         0xC1
#define CHANGE_CHANNEL              0xC2

ULONG BytesTransferred = 0;
UCHAR Channel = 1;
BOOL bRun = TRUE;
CHAR DataArr[64];

// start data transfer
dwStatus = gDevice.VendorRequest(INIT_DATA_TRANSFER, NULL,0, BytesTransferred, 0, 0, 0 );

// change data transfer channel
dwStatus = gDevice.VendorRequest(CHANGE_CHANNEL,
    &Channel, // new channel value (1 byte)
    1, // buffer length
    BytesTransferred, // actual amount of data sent (must be 1 after return)
    0, // direction OUT
    0, 0 // Index and Value not used for this command
);

while(bRun)
{
    // read data from the device (address 0x40) with single GPIF transactions
    dwStatus = gDevice.VendorRequest(CEUSB2_CMD_GPIF_SINGLE_READ_WRITE,
        DataArr, // data buffer
        64, // data buffer length
        BytesTransferred, // actual amount of data read
        1, // direction IN
        0x40, // start address to read
        0 // not used for this command
    );

    if (!CE_SUCCESS(dwStatus))
    {
        // error reading data
        PrintError(&gDevice,dwStatus, TRUE,TRUE,
            "ERROR - Unable to read data");
        break;
    }
    else
    {
        WsPrintBuffer("%u bytes read\n", BytesTransferred );
    }
}

// stop data transfer
dwStatus = gDevice.VendorRequest(STOP_DATA_TRANSFER, NULL,0, BytesTransferred, 0, 0, 0 );

```

2.8 Controlling the FX2 CPU

GetCpuSettings and SetCpuSettings global API functions are used to control the FX2 CPU (enhanced 8051) on the CeUsb2 board. GetCpuSettings function retrieves the speed, clock inverted and clock output enable parameters from the firmware. SetCpuSettings function updates these parameters. Changes made by this function will not take effect until the FwReinit function is called (or Refresh function of the GPIF class), which directs the execution of the firmware to its initialization function.

FwSoftReset function performs a soft reset (vector to org 00h) in the firmware. If it is called all CPU settings (also GPIF settings) will be lost and default values will be used.

Following code example shows simply how to use these functions in order to play with the CPU settings.

```
// Code example 2.8, cont. 2.7
// FX2 CPU control

UCHAR Temp;
UCHAR CpuSpeed, CpuClkInv, CpuClkOe;

// get CPU settings
dwStatus = GetCpuSettings(&gDevice, &CpuSpeed, &CpuClkInv, &CpuClkOe);
if(!CE_SUCCESS(dwStatus))
{
    // error getting SPU settings
    PrintError(&gDevice,dwStatus, TRUE,TRUE,
        "ERROR - Unable to get CPU settings");
}

if(CpuSpeed == 0) // 12 MHz
    Temp = 12;
else if (CpuSpeed == 1) // 24 Mhz
    Temp = 24;
else // 48 Mhz
    Temp = 48;

WsPrintBuffer("CPU speed %u Mhz\nCpu clock is%sinverted\nCpu clock output is%senabled",
    Temp, (CpuClkInverter == 0) ? "not " : "", (CpuClkOe == 0) ? "not " : "");

CpuSpeed = 2; // 48 Mhz
CpuClkInv = 0;
CpuClkOe = 0;

// set CPU settings
dwStatus = SetCpuSettings(&gDevice, CpuSpeed, CpuClkInv, CpuClkOe);
if(!CE_SUCCESS(dwStatus) || (dwStatus = FwReinit(&gDevice)) != 0)
```

```

{
    // error setting SPU settings
    PrintError(&gDevice,dwStatus, TRUE,TRUE,
              "ERROR - Unable to set CPU settings");
}

//
// Restore previous settings
dwStatus = FwSoftReset(&gDevice);

```

2.9 Controlling the GPIF

Most of the CeUsb2 applications use the GPIF (General purpose interface) for fast data transfers. CeUsb2 Design Guide, CeUsb2dg.pdf, explains the functionalities of the GPIF engine more detailed. This section focuses on the API class GPIF, which is used to control some features of the GPIF engine.

Init member function of the GPIF class should be called and succeed before any other functions are called or public member variables are accessed. This function retrieves the current GPIF settings, 7 byte GPIF initialization data array and 128 byte GPIF waveform data from the firmware and stores them. Initialization data array and waveform data can be updated with LoadGpifInitData and LoadGpifWaveformData functions. All other GPIF settings are stored in public member variables and can be accessed by the user.

Refresh function sends the current GPIF settings and data arrays to the firmware. It uses the FwReinit function internally to reinitialize the firmware, so that the new settings are refreshed. Restore function skips all previous Refresh function calls and restores the initial settings both in the firmware and GPIF class. It also uses the FwReinit function internally to reinitialize the firmware.

GpifSingleRead and GpifSingleWrite functions reads and writes specified amount of data to or from the firmware with address values using the GPIF single read & write transitions. Their word-wide versions are GpifSingleRead_Wordwide and GpifSingleWrite_Wordwide functions which use 16 bit data interface.

GpifAbort aborts the current waveform transition of the GPIF and GetGpifStatus retrieves an 8 bit register which specifies the current status of the GPIF engine.

Following code example shows simply how to use the GPIF class.

```

// Code example 2.9, cont. 2.8
// GPIF control

GPIF gpif(&gDevice) ; // GPIF class instance

// initialize GPIF

```

```

dwStatus = gpif.Init() ;
if(!CE_SUCCESS(dwStatus))
{
    PrintError(&gDevice,dwStatus, TRUE,TRUE,
              "ERROR - Unable to get GPIF settings");
}

gpif.m_IfClockSource = 1; //interface clock source is internal
gpif.m_IfClockSpeed = 1; //interface clock speed is 48 MHz
gpif.m_IfClockOe = 1; //interface clock output enable
gpif.m_IfClockInverted = 0; // clock is not inverted
gpif.m_GpifAuto = 0; // GPIF auto mode disabled
gpif.m_GpifWordwide = 0 ; // 8 bit data interface

// refresh the changes
dwStatus = gpif.Refresh();
if(!CE_SUCCESS(dwStatus))
{
    PrintError(&gDevice,dwStatus, TRUE,TRUE,
              "ERROR - Unable to refresh GPIF settings");
}

```

2.10 Data transfer with Bulk Pipes

During USB 1.1 days, bulk transfer was slow and its data buffers were limited to 64 bytes. Therefore isochronous transfer was preferred for high speed data transfers over USB bus. USB 2.0 enhanced the speed of USB 1.1 from 1.2 Mbits/second to 48 Mbits/second and bulk transfer buffers are increased to maximum 1024 bytes. This improvement made bulk transfer more important, since it has the advantage of data transfer error checking, which isochronous type doesn't have.

CeUsb2 class's WriteBulk and ReadBulk functions can be used to read and write data over bulk pipes. Before using these functions UsbInterface class should be used to check available bulk pipes and their transfer direction. OUT pipes can be used for write and IN pipes can be used for read transactions.

It is necessary to reset a pipe with ResetPipe Function of CeUsb2 class before using it. Also, AbortPipe function can be used to abort a pipe.

Code example 2.10 shows how to check for the right bulk pipes with UsbInterface class, how to reset them and how to perform USB bulk transfers through CeUsb2 devices with them.

```

// Code example 2.10, cont. 2.6
// Bulk transfer

// Assume that we have two bulk pipes implemented in the firmware (pipe 0 for writing, pipe 1 for reading).

```

```

// Using the UsbInterface class, we should check if this information is correct (see code example 2.6 also).

#define BUFFER_SIZE 512
UCHAR Buffer[BUFFER_SIZE];

// check if pipe 0 is a write bulk pipe
PUSBD_PIPE_INFORMATION pPipe = Int.GetPipePtr(0);
if(pPipe->PipeType != _UsbdPipeTypeBulk || (pPipe->EndpointAddress & 0x80))
{
    // invalid pipe
    dwStatus = CEUSB2_STATUS_INVALID_PIPE;

    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Pipe 0 is invalid");
    return dwStatus;
}

// check if pipe 1 is a read bulk pipe
pPipe = Int.GetPipePtr(1);
if(pPipe->PipeType != _UsbdPipeTypeBulk || !(pPipe->EndpointAddress & 0x80))
{
    // invalid pipe
    dwStatus = CEUSB2_STATUS_INVALID_PIPE;

    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Pipe 1 is invalid");
    return dwStatus;
}

// reset both pipes
if((dwStatus = gDevice.ResetPipe(0)) != 0 ||
(dwStatus = gDevice.ResetPipe(1)) != 0)
{
    // failed to reset pipe
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to reset pipe");
    return dwStatus;
}

ZeroMemory(Buffer,BUFFER_SIZE); // zero buffer

while (bRun)
{
    // write data to the bulk pipe 0
    dwStatus = gDevice. IsoWrite(0,Buffer, BUFFER_SIZE, BytesTransferred, NULL);
    if(!CE_SUCCESS(dwStatus))
    {
        // error writing to bulk pipe 0
        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - Unable to write to bulk pipe 0");
        return dwStatus;
    }

    WsPrintBuffer("%u bytes written\n",BytesTransferred );

    // read data from the bulk pipe 1

```

```

dwStatus = gDevice. BulkRead(1,Buffer, BUFFER_SIZE, BytesTransferred, NULL);
if(!CE_SUCCESS(dwStatus))
{
    // error reading from bulk pipe 1
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to read from bulk pipe 1");
    return dwStatus;
}

WsPrintBuffer("%u bytes read\n",BytesTransferred );

Sleep(0); // wait some amount of time
}

```

2.11 Data transfer with Isochronous Pipes

Isochronous endpoints typically handle time-critical, streamed data that is delivered or consumed in byte-sequential order. From USB 1.1 to USB 2.0 isochronous buffer size limit is increased from 1023 to 1024. The most important feature of isochronous pipes is that they can be used for data streaming as it is explained in the next section.

CeUsb2 has two functions, IsoRead and IsoWrite, for reading and writing over isochronous pipes. Reading and writing process is similar to bulk transfer except that isochronous functions need some more parameters other than pipe number. This information is passed to these functions with a CEUSB2_ISO_TRANSFER_INFO structure object which is shown below.

```

// Isochronous transfer information structure
typedef struct _CEUSB2_ISO_TRANSFER_INFO
{
    ULONG PipeNumber;           // pipe number
    ULONG PacketSize;          // packet size
    ULONG FramesPerBuffer;     // number of frames in a buffer
    ULONG BufferCount;          // total numbers of buffers
}CEUSB2_ISO_TRANSFER_INFO, *PCEUSB2_ISO_TRANSFER_INFO;

```

PipeNumber is the zero based pipe number associated with the endpoint to read from.

PacketSize is the amount of ISO data to read during each frame. This value usually corresponds to the max packet size of the ISO endpoint, but can be less. FramesPerBuffer is the number of USB frames of data to transfer in a single URB (USB Request Block).

BufferCount is the number of transfer URBs to use for this transfer.

In order to maintain an ISO stream, the driver must maintain at least two sets of ISO transfer buffers, so that when one set completes the next set of transfers can be ready to go. The driver ping pongs between these two sets of frame buffers

until the transfer completes. BufferCount determines how many buffer-sets the driver ping pongs between, and FramesPerBuffer determines how many frames of USB data are represented by each of those buffers.

Code example 2.11 shows how to perform USB isochronous transfer using IsoRead and IsoWrite functions through CeUsb2 devices.

```
// Code example 2.11, cont. 2.10
// Isochronous transfer

// Assume that we have two isochronous pipes implemented in the firmware (pipe 0 for writing, pipe 1 for
// reading).
// Using the UsbInterface class, we should check if this information is correct (see also code example 2.6).

CEUSB2_ISO_TRANSFER_INFO IsoInfoWrite;
CEUSB2_ISO_TRANSFER_INFO IsoInfoRead;

#define FRAMES_PER_BUFFER 10
#define BUFFER_COUNT      2

PUCHAR pIsoBuffer = NULL;

// check if pipe 0 is a write isochronous pipe
pPipe = Int.GetPipePtr(0);
if(pPipe->PipeType != _UsbdPipeTypeIsochronous || (pPipe->EndpointAddress & 0x80))
{
    // invalid pipe
    dwStatus = CEUSB2_STATUS_INVALID_PIPE;

    PrintError(&gDevice,dwStatus,TRUE,TRUE,
               "ERROR - Pipe 0 is invalid");
    return dwStatus;
}

// check if pipe 1 is a read isochronous pipe
pPipe = Int.GetPipePtr(1);
if(pPipe->PipeType != _UsbdPipeTypeIsochronous || !(pPipe->EndpointAddress & 0x80))
{
    // invalid pipe
    dwStatus = CEUSB2_STATUS_INVALID_PIPE;

    PrintError(&gDevice,dwStatus,TRUE,TRUE,
               "ERROR - Pipe 1 is invalid");
    return dwStatus;
}

// reset both pipes
if((dwStatus = gDevice.ResetPipe(0)) != 0 ||
(dwStatus = gDevice.ResetPipe(1)) != 0)
{
    // failed to reset pipe
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
               "ERROR - Unable to reset pipe");
    return dwStatus;
}
```

```

}

IsoInfoWrite.PipeNumber = 0;
IsoInfoWrite.PacketSize = BUFFER_SIZE; //512
IsoInfoWrite.FramesPerBuffer = FRAMES_PER_BUFFER;
IsoInfoWrite.BufferCount = BUFFER_COUNT;

IsoInfoRead.PipeNumber = 1;
IsoInfoRead.PacketSize = BUFFER_SIZE; //512
IsoInfoRead.FramesPerBuffer = FRAMES_PER_BUFFER;
IsoInfoRead.BufferCount = BUFFER_COUNT;

ULONG Size = BUFFER_SIZE * FRAMES_PER_BUFFER * BUFFER_COUNT;
pIsoBuffer = (PCHAR) malloc(Size);
ZeroMemory(pIsoBuffer,Size); // zero buffer

while (bRun)
{
    // write data to the isochronous pipe 0
    dwStatus = gDevice.IsoWrite(&IsoInfoWrite, pIsoBuffer, Size, BytesTransferred, NULL);
    if(!CE_SUCCESS(dwStatus))
    {
        // error writing to isochronous pipe 0
        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - Unable to write to isochronous pipe 0");
        break;
    }

    WsPrintBuffer("%u bytes written\n",BytesTransferred );

    // read data from the isochronous pipe 1
    dwStatus = gDevice.IsoRead(&IsoInfoRead, pIsoBuffer, Size, BytesTransferred, NULL);
    if(!CE_SUCCESS(dwStatus))
    {
        // error reading from isochronous pipe 1
        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - Unable to read from isochronous pipe 1");
        break;
    }

    WsPrintBuffer("%u bytes read\n",BytesTransferred );

    Sleep(0); // wait some amount of time
}

free(pIsoBuffer);

```

2.12 Data transfer with Isochronous streaming

CeUsb2 driver has the ability to transfer data from target device to the host with isochronous streaming techniques. After the streaming is started it starts reading

data from isochronous pipes and puts it into an internal FIFO buffer whose size is determined by the user. Stream read requests reads actually from this buffer.

IsoStreamStart function of CeUsb2 class starts isochronous streaming. It reserves memory for internal FIFO buffer and starts reading with a signaling mechanism connected to a kernel mode thread. Streaming parameters are passed to this function as CEUSB2_ISO_TRANSFER_INFO structure object which is shown below.

```
// Isochronous streaming information structure
typedef struct _CEUSB2_ISO_TRANSFER_INFO_EX
{
    ULONG PipeNumber;           // pipe number
    ULONG PacketSize;          // packet size
    ULONG FramesPerBuffer;     // number of frames in a buffer
    ULONG BufferCount;          // total numbers of buffers
    ULONG StreamBufferSize;    // stream FIFO buffer size
}CEUSB2_ISO_TRANSFER_INFO_EX, *PCEUSB2_ISO_TRANSFER_INFO_EX;
```

PipeNumber is the zero based pipe number associated with the endpoint to read from.

PacketSize is the amount of ISO data to read during each frame. This value usually corresponds to the max packet size of the ISO endpoint, but can be less. FramesPerBuffer is the number of USB frames of data to transfer in a single URB (USB Request Block).

BufferCount is the number of transfer URBs to use for this transfer.

StreamBufferSize is the size of internal isochronous FIFO buffer.

In order to maintain an ISO stream, the driver must maintain at least two sets of ISO transfer buffers, so that when one set completes the next set of transfers can be ready to go. The driver ping pongs between these two sets of frame buffers until the transfer completes. BufferCount determines how many buffer-sets the driver ping pongs between, and FramesPerBuffer determines how many frames of USB data are represented by each of those buffers.

IsoStreamStop function of CeUsb2 class traverses the functionality of the IsoStreamStart and stops isochronous streaming. After stopping the kernel mode isochronous read thread, it frees the FIFO buffer.

IsoStreamStart function of the CeUsb2 class can be used to read specified amount of data from the isochronous FIFO buffer. If the buffer has less data than the requested, the user gets only this amount of data. Notice that, if the user doesn't read from the FIFO for some amount of time and if it gets full, internal thread will not stop reading from the device. However, incoming bytes will be thrown away.

Code example 2.12 shows how to perform data transfer with isochronous streaming through CeUsb2 devices.

```

// Code example 2.12, cont. 2.11
// Data transfer with isochronous streaming

// TODO:
// Check and reset pipes (see example code 2.9)

CEUSB2_ISO_TRANSFER_INFO_EX IsoInfoStream;

IsoInfoStream.PipeNumber = 1;
IsoInfoStream.PacketSize = BUFFER_SIZE; //512
IsoInfoStream.FramesPerBuffer = FRAMES_PER_BUFFER;
IsoInfoStream.BufferCount = BUFFER_COUNT;
IsoInfoStream.StreamBufferSize = 1024 * 1024; // 1 MByte internal isochronous data buffer in the driver

// start isochronous streaming
dwStatus = gDevice.IsoStreamStart(&IsoInfoStream);
if(!CE_SUCCESS(dwStatus))
{
    // error starting isochronous data streaming
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        "ERROR - Unable to start isochronous streaming");
    return dwStatus;
}

while(bRun)
{
    // read from the stream
    dwStatus = gDevice.IsoStreamRead(
        Buffer, BUFFER_SIZE, BytesTransferred,NULL);
    if(!CE_SUCCESS(dwStatus))
    {
        // error reading from isochronous stream

        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - Unable to read from isochronous stream");
        break;
    }

    WsPrintBuffer("%u bytes read\n",BytesTransferred );
    Sleep(1);
}

// stop isochronous streaming
dwStatus = gDevice.IsoStreamStop();

```

2.13 FPGA configuration

Detailed explanation about FPGA configuration can be found in CeUsb2 general design guide (CeUsb2dg.pdf).

Two classes of the CeUsb2 API are responsible from FPGA configuration: FpgaConfig and Fpga.

FpgaConfig class has raw level functions to control the FPGA chip. Init function initializes the FPGA for configuration. IsConfigured function checks whether a configuration process is completed successfully. Reset function sends a reset signal to the design running in the FPGA.

Fpga class is derived from FpgaConfig class and implements an upper level and easier to use interface for FPGA configuration process. Internally it uses the raw functions of the FpgaConfig class.

Its constructor takes an FPGA configuration file path as an argument (IpcConfigFile). Its Configure function can read files with extensions exo and rbt (raw bit file), extract the configuration data, and download the configuration data into the FPGA.

Before starting configuration process you should know if your firmware implements FPGA configuration data transfer over control pipe or not. If it is implemented on control pipe you should pass TRUE to the bControlTransfer argument of the constructor. If this process is handled over bulk or isochronous pipes, unless you specify a pipe number with PipeNum argument of the constructor (it is initially -1, which means auto search), Configure function will search for the right pipe for FPGA configuration automatically.

Code example 2.13 shows how to configure and reset the FPGA using Fpga class.

```
// Code example 2.13, cont. 2.12
// FPGA configuration

#define FPGA_FILE_PATH      "C:\\test.exo"

Fpga fpga(&gDevice, FPGA_FILE_PATH,
         FALSE, // we won't use control pipe for fpga configuration
         -1 // pipe number will be found by Fpga class automatically
         TRUE // use dynamic GPIF waveform and settings,
         512
         );

// configure fpga
dwStatus = fpga.Configure();
if(!CE_SUCCESS(dwStatus))
{
    // error configuring fpga
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
              " ERROR - Unable to configure Fpga ");
    return dwStatus;
}

// reset fpga
```

```

dwStatus = fpga.Reset();
if(!CE_SUCCESS(dwStatus))
{
    // error resetting fpga
    PrintError(&gDevice,dwStatus,TRUE,TRUE,
        " ERROR - Unable to reset Fpga ");
    return dwStatus;
}

```

2.14 RS232 – Serial interface handling

Some CeUsb2 type devices have one or more serial interfaces. They can be accessed by the vendor commands of the generic firmware interface if the firmware enables the user access to serial input and output.

RS232 class of the CeUsb2 API is an upper level interface for serial input output. TxSend member function sends specified amount of data to the SIO (serial I/O). SIO reading is done by an internal data capturing thread. RxStart starts this thread and takes a pointer to a user defined callback function which will be called by the thread every time data is received over RS232.

Code example 2.14 shows how to send data and start data capturing with RS232 class.

Code example 2.15 is the simple RX notify function sample for code example 2.14.

```

// Code example 2.14
// Serial data communication with RS232 class

// RX procedure prototype
void RxProc(PVOID Context, DWORD dwStatus, PCHAR pBuffer, UCHAR BufferLength);

//
// .....
//

RS232 rs232(&gDevice);
ULONG SampleContext = 0xABCD;

// send data to RS232
dwStatus = rs232.TxSend(Buffer,64,FALSE);

// start RX data capturing
dwStatus = rs232.RxStart((RX_NOTIFY*)RxProc,(PVOID)&SampleContext);

while(bRun)
    ;

// stop RX data capturing

```

```
rs232.RxStop();
```

```
// Code example 2.15
// RX capture procedure

void RxProc(PVOID Context, DWORD dwStatus, PCHAR pBuffer, UCHAR BufferLength)
{
    ULONG ctx = *((ULONG*)Context);

    if(CE_SUCCESS(dwStatus))
    {
        WsPrintBuffer("%u bytes read\n", BufferLength);
    }
    else
    {
        PrintError(&gDevice,dwStatus,TRUE,TRUE,
            "ERROR - RS232 error");
    }
}
}
```

2.15 Other USB operations

CeUsb2 has some other functions which are not mentioned before in this chapter:

GetCurrentFrameNumber retrieves the current frame number on the USB bus. GetStatus retrieves the current status from a CeUsb2 device, interface or endpoint.

GetPowerState and SetPowerState functions get and set the current power state of the CeUsb2 device.

FeatureRequest function sets or clears an USB feature on a CeUsb2 device, interface or endpoint.

Code example 2.16 shows simple usages of some of these functions.

```
// Code example 2.16
// Other USB operations

ULONG FrameNumber = 0; // current USB frame number
UCHAR Status = 0; // USB status
CEUSB2_POWER_INFO PwInfo; // CeUsb2 power information object

// constant strings for power states
static const char* PowerStatesStr[6] = {
    "PowerDeviceUnspecified",
    "PowerDeviceD0",
    "PowerDeviceD1",
    "PowerDeviceD2",
    "PowerDeviceD3",
}
```

```

        "PowerDeviceMaximum"
    };

    // get current frame number
    dwStatus = gDevice.GetCurrentFrameNumber(FrameNumber);
    if(CE_SUCCESS(dwStatus))
        WsPrintBuffer("Current frame number %u, %08X\n\n", FrameNumber, FrameNumber);
    else
        return dwStatus; //error

    // get the status of the first endpoint
    dwStatus = gDevice.GetStatus(RECIPIENT_ENDPOINT, 0, Status);
    if(CE_SUCCESS(dwStatus))
        WsPrintBuffer("Endpoint 0 status %2X\n", Status);
    else
        return dwStatus; //error

    dwStatus = gDevice.GetPowerState(&PwInfo);
    if(CE_SUCCESS(dwStatus))
    {
        WsPrintBuffer("Current power state %2X, %s\n",
            (UCHAR)PwInfo.PowerState, PowerStatesStr[(int)PwInfo.PowerState]);
    }
    else
        return dwStatus; //error

    PwInfo.Wait = TRUE;

    // increase the power state
    PwInfo.PowerState = (CEUSB2_POWER_STATE)((int)PwInfo.PowerState + 1);
    dwStatus = gDevice.SetPowerState(PwInfo);

    // back to previous power state
    PwInfo.PowerState = (CEUSB2_POWER_STATE)((int)PwInfo.PowerState - 1);
    dwStatus = gDevice.SetPowerState(PwInfo);

```

2.16 Error handling

Most of the class member functions and exported global functions of the CeUsb2 API return a 32 bit double word (DWORD) status code which indicates the completion status of the operation. CE_SUCCESS macro can be used to check if this status code indicates error or successful completion.

CeUsb2 API also has two global exported functions, FormatErrorString and FormatUSBDErrorStr, which are used for extensive error handling.

FormatErrorString takes a 32 bit status code and returns a short description of the status as a string. It has an internal error string table and if the error is defined by one of the components of CeUsb2 it returns the corresponding string from the table. If the status code is a system defined code however, it uses

FormatMessage function of Win32 SDK to retrieve a system defined text for the status code.

Ceusb2 driver returns CEUSB2_STATUS_USBD_ERROR status code if an USB specific error occurs, and stores another status code (USB status code) in the driver which can be retrieved with GetUSBDError function of CeUsb2 class.

FormatUSBDErrorStr takes this USB status code as an argument and returns a descriptive text about the error including the current state of the last USB request. The resultant text will be like:

“State = state string Status = status text”

State string can be SUCCESS, PENDING, STALLED, ERROR depending on the state of the request.

Code example 2.17 shows implementations of three functions, PrintBuffer, WsPrintBuffer, and PrintError, which are used in the example source codes of this document.

PrintBuffer directs a formatted text to an output console, dialog or wherever requested.

WsPrintBuffer prints an unformatted text using PrintBuffer function.

PrintError function prints a description of the error with its status code and user supplied description. Upon request, it can search if the error code is USB specific and if so it displays another text using FormatUSBDErrorStr function.

```
// Code example 2.17
// Some helper functions for string formatting and error handling

// prints a formatted buffer to an output console
void PrintBuffer(char* pBuffer)
{
    printf(pBuffer); // for console programs

    //...
}

// prints an unformatted buffer to an output console
void WsPrintBuffer(char* pFormat, ...)
{
    char Buffer[512];

    va_list argptr;
    va_start(argptr, pFormat);
    vsprintf(Buffer, pFormat, argptr);
    va_end(argptr);

    PrintBuffer(Buffer);
}

// searches the reason for an error
void PrintError(
```

```

CeUsb2*      Dev,      // device on which the error occurred, can be NULL
DWORD       dwError, // status code
BOOL        bSearchReason, // if FALSE, reason for the error is not searched, only user
            //supplied text will be displayed
BOOL        bCheckUSBDError, // if TRUE, USBDE error will be searched
const char* szUserDescription // user supplied description, will be displayed first
...
)
{
    char sError[512];

    va_list argptr;
    va_start(argptr,szUserDescription);
    vsprintf(sError,szUserDescription,argptr);
    va_end(argptr);

    PrintBuffer(sError);

    strcpy(sError,"");
    if(bSearchReason)
    {
        FormatErrorString(sError,dwError);

        WsPrintBuffer("\nError code : 0x%08X",dwError);
        WsPrintBuffer("\nDescription : %s\n",sError);

        if(bCheckUSBDError && dwError == CEUSB2_STATUS_USBDE_ERROR)
        {
            ULONG USBDEStatus;

            strcpy(sError,"");
            if(Dev->GetUSBDEError(USBDEStatus))
            {
                FormatUSBDEErrorStr(sError,USBDEStatus);
                WsPrintBuffer("\nUSBDE Error Code : 0x%08X",USBDEStatus);
                WsPrintBuffer("\nDescription : %s\n",sError);
            }
        }
    }

    PrintBuffer("\n");
}

```

3 CEUSB2 API REFERENCE

3.1 CeUsb2Ld class

CeUsb2 has two drivers which are loaded by the system sequentially. The first one, loader driver, can be accessible when the device is in an unconfigured

state, through the CeUsb2Ld class. You will get a configured device from Cesys, so you will hardly need to use this class.

3.1.1 CeUsb2Ld::CeUsb2Ld

```
CeUsb2Ld();
```

Description : Standard constructor for CeUsb2Ld class.

Return Value : None

Arguments : None

3.1.2 CeUsb2Ld::~~CeUsb2Ld

```
virtual ~CeUsb2Ld();
```

Description : Standard destructor for CeUsb2Ld class.

Return Value : None

Arguments : None

3.1.3 CeUsb2Ld::Open

```
DWORD Open(UINT DeviceIndex = 0 );
```

Description : Opens the CeUsb2Ld device with the given device index. After a successful open a 32 bit handle to the device is stored. This handle is released and deleted when Close member function is called.

Return Value : 32 bit status code if an error occurs, else 0

Arguments :

DeviceIndex - 0 based device index.

Notes: Unlike the handler class for CeUsb2 main driver (CeUsb2), CeUsbLd class uses dos device naming (named device object).

3.1.4 CeUsb2Ld::Close

```
void Close(void);
```

Description : Closes a previously opened device instance. It reverses the functionality of the Open function and releases the handle to the device which is stored previously by Open function. This function is called also by the destructor.

Return Value : None

Arguments : None

3.1.5 CeUsb2Ld::GetDriverInfo

```
DWORD GetDriverInfo( PCEUSB2LD_DRIVER_INFO pInfo);
```

Description : Retrieves versioning information from the loader driver.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pInfo - Pointer to a CEUSB2LD_DRIVER_INFO (CeUsb2 loader driver information) structure which will hold the versioning information.

3.1.6 CeUsb2Ld::ConfigureDevice

```
DWORD ConfigureDevice(USHORT DeviceId);
```

Description : All CeUsb2 devices have their own 2 byte device ids which is hard-coded in the serial EEPROM on the board. An unconfigured board however, has 0xFFFF in this field. ConfigureDevice function configures the device with a given device id. If a wrong id is written in the EEPROM, the board will still be in an unconfigured state and will be inaccessible with standard CeUsb2 software.

Return value : 32 bit status code if an error occurs, else 0.

Arguments :

DeviceId - Cesium specific device id.

3.1.7 CeUsb2Ld::VendorRequest

```
DWORD VendorRequest(  
    UCHAR Request,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesTransferred,  
    UCHAR Direction=0,  
    USHORT Value=0,  
    USHORT Index=0  
);
```

Description : Performs a vendor request through CeUsb2Ld driver. This function is added for future compatibility.

Return value : 32 bit status code if an error occurs, else 0

Arguments :

Request – Vendor request code. Loader firmware request codes are defined in CeUsb2Ldif.h header file.

Buffer - Data buffer for vendor request.

BufferLength - Data buffer length.

BytesTransferred - Actual number of bytes transferred.

Direction - Direction of data transfer. Possible values are:

0 (OUT) from host to the device
1 (IN) from device to the host
Value - Vendor command specific value.
Index - Vendor command specific index.

3.2 *CeUsb2* class

CeUsb2 class is the main class that handles all USB functionality through CeUsb2 devices. It gives a raw level USB interface for configuration selection, interface and alternate setting selection, vendor and class requests, feature setting, descriptor handling, power management, control & bulk & isochronous data transfers and more. Before using this class you should search the system for CeUsb devices using DeviceList class of the API (See section 3.3, DeviceList class).

CeUsb2 class can perform USB data transfers both synchronously and asynchronously. You determine this mode when opening the device. However some functions operate always synchronously. Some internal logic handles this situation for you.

3.2.1 CeUsb2::CeUsb2

```
CeUsb2();
```

Description : Standard constructor for CeUsb2 class.

Return Value : None

Arguments : None

3.2.2 CeUsb2::CeUsb2 (2)

```
CeUsb2(const LPCTSTR LinkName, const LPCTSTR FriendlyName);
```

Description : Constructor with link name and friendly name arguments

Return Value : None

Arguments :

LinkName : Device link name string.

FriendlyName: Device friendly link name string.

3.2.3 CeUsb2::CeUsb2 (3)

```
CeUsb2(const CeUsb2 &r);
```

Description : Copy constructor for CeUsb2 class.

Return Value : None

Arguments :

r - Copy constructor right side operator, which is reference to a CeUsb2 class instance.

3.2.4 CeUsb2::~~CeUsb2

```
virtual ~CeUsb2();
```

Description : Standard destructor for CeUsb2 class.

Return Value : None

Arguments : None

3.2.5 CeUsb2::operator=

```
CeUsb2& operator=(const CeUsb2& r);
```

Description : Assignment operator for CeUsb2 class.

Return Value : Reference to this instance of the class.

Arguments :

r - Right operand of assignment operation, which is reference to a CeUsb2 class instance.

3.2.6 CeUsb2::Open

```
DWORD Open(BOOL bAsync = FALSE);
```

Description : Opens the device for operation and stores a 32 bit handle for it. CeUsb2 driver uses device interface instead of the old dos device naming technique. A device interface is uniquely identified by a 128 bit GUID number. This GUID is registered in the system by the driver when it is loaded.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

bAsync - Boolean value that selects between asynchronous and synchronous operation (TRUE - asynchronous).

3.2.7 CeUsb2::Close

```
void Close(void);
```

Description : Closes a previously opened device instance. It reverses the functionality of the Open function and releases the handle to the device which is stored previously by Open function. This function is called also by the destructor.

Return Value : None

Arguments : None

3.2.8 CeUsb2::GetDeviceHandle

```
HANDLE GetDeviceHandle(void);
```

Description : Retrieves the 32 bit handle for the device which is stored by the Open function.

Return Value : 32 bit device handle value or NULL if the device is not open.

Arguments : None

3.2.9 CeUsb2::GetLinkName

```
LPCTSTR GetLinkName(void);
```

Description : Retrieves the link name of the device. It is generated by the driver during the registration of the device interface. This string is very long, ugly, and not informative.

Return Value : Device link name string.

Arguments : None

3.2.10 CeUsb2::GetFriendlyName

```
LPCTSTR GetFriendlyName(void);
```

Description : : Retrieves the friendly link name of the device. This string is shorter than the link name and is more informative. It is set during the installation of the driver and specified by the driver installation information file's (CeUsb2.inf) HW section.

Return Value : Device friendly link name string.

Arguments : None

3.2.11 CeUsb2::GetDriverInfo

```
DWORD GetDriverInfo(PCEUSB2_DRIVER_INFO pInfo);
```

Description : Retrieves versioning information from the CeUsb2 driver.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

plInfo - Pointer to a CEUSB2_DRIVER_INFO (CeUsb2 driver information) structure.

3.2.12 CeUsb2::GetUSBDError

```
DWORD GetUSBDError(ULONG &USBDError);
```

Description : Retrieves the most recently occurred USB D error code. CeUsb2 driver stores the USB D error codes internally.

Return Value : 32 bit status code if an error, else 0.

Arguments :

USBDError - Reference to a 32 bit long value which will hold the USB D status code.

Notes : This function call also returns a 32 bit status code, since it makes a driver call to retrieve the USB D status.

3.2.13 CeUsb2::CancelRequests

```
DWORD CancelRequests(void);
```

Description : Cancels all pending I/O requests directed to the driver previously. Internally it uses the Canceled function of the Win32 SDK. It returns the GetLastError function's result if Canceled fails; otherwise returns 0.

Return Value : 32 bit status code if an error, else 0.

Arguments : None

3.2.14 CeUsb2::GetConfiguration

```
DWORD GetConfiguration(PUCHAR Configuration);
```

Description : Retrieves the current USB Configuration number.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Configuration - Pointer to an unsigned character value that will hold the current configuration number value. Configuration number is indexed starting from 1. 0 means the device is in an unconfigured state.

3.2.15 CeUsb2::SelectConfiguration

```
DWORD SelectConfiguration(UCHAR Configuration);
```

Description : Selects the specified USB configuration for the CeUsb2 device.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Configuration - USB configuration number to select.

3.2.16 CeUsb2::GetInterface

```
DWORD GetInterface(UCHAR Interface,PUCHAR AlternateSetting);
```

Description : Retrieves the current alternate setting for an interface in the current configuration.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Interface - Interface number.

AlternateSetting - Pointer to an unsigned character value that will hold the alternate setting.

3.2.17 CeUsb2::SelectInterface

```
DWORD SelectInterface(  
    UCHAR InterfaceNumber,  
    UCHAR AlternateSetting,  
    ULONG *pMaxTransferSizeArr,  
    UCHAR ArrSize  
);
```

Description : Selects an interface and alternate setting for the current configuration.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Interface - Interface number.

AlternateSetting - Alternate setting number.

pMaxTransferSizeArr : Maximum USB transfer size array for pipes (starting from pipe 0).

ArrSize - Maximum USB transfer size array's size.

3.2.18 CeUsb2::GetCurrentFrameNumber

```
DWORD GetCurrentFrameNumber(ULONG &FrameNumber);
```

Description : Retrieves the current frame number on the USB bus.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

FrameNumber - Reference to a 32 bit variable which will hold the current frame number.

3.2.19 CeUsb2::GetStatus

```
DWORD GetStatus(  
    CEUSB2_REQUEST_RECIPIENT Recipient,  
    USHORT Index,  
    UCHAR &Status  
);
```

Description : Retrieves status from a device, interface, endpoint, or other device-defined target. These statuses are device and firmware specific. Check the documentation of your firmware about the supported ones.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Recipient - Request recipient.

Index - Index value used with get status request (firmware specific).

Status - Reference to an unsigned character variable which will hold the status value.

3.2.20 CeUsb2::GetPowerState

```
DWORD GetPowerState(PCEUSB2_POWER_INFO pPowerInfo);
```

Description : Retrieves the current power state of the CeUsb2 device.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pPowerInfo - Pointer to an instance of CEUSB2_POWER_INFO structure which will hold information about the current power state of the device.

3.2.21 CeUsb2::SetPowerState

```
DWORD SetPowerState(CEUSB2_POWER_INFO PowerInfo);
```

Description : Sets the current power state of the CeUsb2 device.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

PowerInfo - An instance of CEUSB2_POWER_INFO structure which will hold information about the new power state of the device.

3.2.22 CeUsb2::FeatureRequest

```
DWORD FeatureRequest(PCEUSB2_FEATURE_REQUEST_INFO pFeatureInfo);
```

Description : Sets or clears features on a device, interface, or endpoint. These features are device and firmware specific. Check the documentation of your firmware about the supported ones.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pFeatureInfo - An instance of CEUSB2_FEATURE_REQUEST_INFO structure which will hold information about the USB feature request.

3.2.23 CeUsb2::DescriptorRequest

```
DWORD DescriptorRequest(  
    PCEUSB2_DESCRIPTOR_INFO    pDescInfo,  
    PVOID                      Buffer,  
    ULONG                      BufferLength,  
    ULONG                      &BytesTransferred  
);
```

Description : Retrieves or sets an USB descriptor.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDescInfo - An instance of CEUSB2_DESCRIPTOR_INFO structure which will hold information about the USB descriptor request.

Buffer - Data buffer.

BufferLength - Data buffer length.

BytesTransferred - Actual number of bytes transferred.

Notes : This function is suitable to retrieve USB device, configuration and string descriptors of a CeUsb2 device although it can be used for the others.

GetInterfaceDescriptor function is used instead, to retrieve interface and endpoint descriptors.

Descriptor class is a wrapper class for descriptor handling, which is based on DescriptorRequest and GetInterfaceDescriptor functions of the CeUsb2 class. Use this class instead, since it simplifies descriptor handling process.

3.2.24 CeUsb2::GetInterfaceDescriptor

```
DWORD GetInterfaceDescriptor(  
    UCHAR    ConfigNo,  
    UCHAR    InterfaceNumber,  
    UCHAR    AlternateSetting,  
    PVOID    Buffer,
```

```
ULONG    BufferLength,  
ULONG    &BytesReturned  
);
```

Description : Retrieves an USB interface descriptor.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

ConfigNo - Configuration number

InterfaceNumber - Interface number.

AlternateSetting - Alternate setting number.

Buffer - Data buffer.

BufferLength - Data buffer length.

BytesTransferred - Actual number of bytes transferred.

Notes : Buffer argument of this function can hold interface descriptor itself or with the endpoints it has, depending on the BufferLength argument. Buffer is typically a pointer to a memory location which begins with the form of USB_INTERFACE_DESCRIPTOR structure, and may be preceded by some USB_ENDPOINT_DESCRIPTOR structures depending on the number of endpoints this interface has.

Descriptor class is a wrapper class for descriptor handling, which is based on DescriptorRequest and GetInterfaceDescriptor functions of the CeUsb2 class. Use this class instead, since it simplifies descriptor handling process.

3.2.25 CeUsb2::GetInterfaceInformation

```
DWORD GetInterfaceInformation (  
    PVOID pInterfaceInfo,  
    ULONG Length,  
    ULONG &BytesReturned  
);
```

Description : Retrieves information about the currently selected USB interface.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pInterfaceInfo - Interface information buffer.

Length - Interface information buffer length.

BytesTransferred - Actual number of bytes transferred.

Notes : pInterfaceInfo argument of this function can hold interface information itself or with the pipes it has, depending on the Length argument. pInterfaceInfo is typically a pointer to a memory location which begins with the form of USB_INTERFACE_INFORMATION structure, and may be preceded by some

USB_PIPE_INFORMATION structures depending on the number of pipes this interface has.

UsbInterface class is a wrapper class for USB interface handling, which is based on GetInterfaceInformation function of the CeUsb2 class. Use this class instead, since it simplifies interface and pipe handling process.

3.2.26 CeUsb2::ResetDevice

```
DWORD ResetDevice(void);
```

Description : Resets USB port. Uses internal USBD IOCTL code IOCTL_INTERNAL_USB_RESET_PORT.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.2.27 CeUsb2::ResetPipe

```
DWORD ResetPipe(ULONG PipeNumber);
```

Description : Resets an USB pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

PipeNumber - Pipe number.

3.2.28 CeUsb2::AbortPipe

```
DWORD AbortPipe(ULONG PipeNumber);
```

Description : Aborts an USB pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

PipeNumber - Pipe number.

3.2.29 CeUsb2::VendorRequest

```
DWORD VendorRequest(  
    UCHAR Request,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesTransferred,  
    UCHAR Direction=0,  
    USHORT Value=0,  
    USHORT Index=0
```

```
);
```

Description : Performs an USB vendor request through CeUsb2 device.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Request - Vendor request code. All CeUsb2 devices may use generic firmware interface vendor requests and able to define their specific ones.

Buffer - Data buffer (vendor request specific).

BufferLength - Data buffer length (vendor request specific).

BytesTransferred - Actual number of bytes transferred.

Direction - Direction of data transfer. Possible values are:

0 (OUT) from host to the device

1 (IN) from device to the host

Value - Vendor request specific value.

Index - Vendor request specific index.

3.2.30 CeUsb2::VendorOrClassRequest

```
DWORD VendorOrClassRequest(  
    UCHAR Request,  
    PVOID Buffer ,  
    ULONG BufferLength,  
    ULONG &BytesTransferred,  
    UCHAR Direction = 0,  
    USHORT Value = 0,  
    USHORT Index = 0,  
    CEUSB2_REQUEST_RECIPIENT Recipient = RECIPIENT_DEVICE,  
    UCHAR RequestType = 1  
);
```

Description : Performs an USB vendor or class request through CeUsb2 device.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Request - Vendor or class request code.

Buffer - Data buffer.

BufferLength - Data buffer length (vendor or class request specific).

BytesTransferred - Actual number of bytes transferred (vendor or class request specific).

Direction - Direction of data transfer. Possible values are:

0 (OUT) from host to the device

1 (IN) from device to the host

Value - Vendor or class request specific value.

Index - Vendor or class request specific index.

Recipient - Request recipient.

RequestType - Request type (0 - vendor, 1 - class).

Notes : For vendor requests whose recipient is the device itself, use VendorRequest function instead.

3.2.31 CeUsb2::BulkRead

```
DWORD BulkRead(  
    ULONG PipeNumber,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Reads specified amount of data from a bulk pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

PipeNumber - Pipe number

Buffer - Data buffer

BufferLength - Data buffer length.

BytesRead - Actual number of bytes read.

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.32 CeUsb2::BulkWrite

```
DWORD BulkWrite(  
    ULONG PipeNumber,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesWritten,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Writes specified amount of data to a bulk pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

PipeNumber - Pipe number

Buffer - Data buffer

BufferLength - Data buffer length.

BytesWritten - Actual number of bytes written

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.33 CeUsb2::IsoRead

```
DWORD IsoRead(  
    PCEUSB2_ISO_TRANSFER_INFO pIsoInfo,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Reads specified amount of data from an isochronous pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pIsoInfo - Pointer to an instance of CEUSB2_ISO_TRANSFER_INFO object which will hold information about isochronous data transfer.

Buffer - Data buffer

BufferLength - Data buffer length.

BytesRead - Actual number of bytes read.

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.34 CeUsb2::IsoWrite

```
DWORD IsoWrite(  
    PCEUSB2_ISO_TRANSFER_INFO pIsoInfo,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesWritten,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Writes specified amount of data to an isochronous pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pIsoInfo - Pointer to an instance of CEUSB2_ISO_TRANSFER_INFO object which will hold information about isochronous data transfer.

Buffer - Data buffer

BufferLength - Data buffer length.

BytesWritten - Actual number of bytes written.

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.35 CeUsb2::IsoStreamStart

```
DWORD IsoStreamStart(PCEUSB2_ISO_TRANSFER_INFO_EX pIsoInfo);
```

Description : Starts internal isochronous streaming.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pIsoInfo - Pointer to an instance of CEUSB2_ISO_TRANSFER_INFO_EX object which will hold information about isochronous streaming.

3.2.36 CeUsb2::IsoStreamStop

```
DWORD IsoStreamStop(void);
```

Description : Stops internal isochronous streaming.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.2.37 CeUsb2::IsoStreamRead

```
DWORD IsoStreamRead(  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Reads specified amount of data from an isochronous stream. If the internal isochronous streaming FIFO has less data than the requested, then this data is returned.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

Buffer - Data buffer

BufferLength - Data buffer length.

BytesRead - Actual number of bytes read.

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.38 CeUsb2::ControlRead

```
DWORD ControlRead(  
    PCEUSB2_CONTROL_TRANSFER_INFO pControlInfo,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG &BytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Reads specified amount of data from a Control pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pControlInfo - Pointer to an instance of CEUSB2_CONTROL_TRANSFER_INFO object which will hold information about control data transfer.

Buffer - Data buffer

BufferLength - Data buffer length

BytesRead - Actual number of bytes read

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.2.39 CeUsb2::ControlWrite

```
DWORD ControlWrite(  
    PCEUSB2_CONTROL_TRANSFER_INFO pControlInfo,  
    PVOID Buffer,  
    ULONG BufferLength,  
    ULONG & BytesWritten,  
    LPOVERLAPPED lpOverlapped  
);
```

Description : Writes specified amount of data to a Control pipe.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pControlInfo - Pointer to an instance of CEUSB2_CONTROL_TRANSFER_INFO object which will hold information about control data transfer.

Buffer - Data buffer

BufferLength - Data buffer length

BytesWritten - Actual number of bytes written

lpOverlapped - Pointer to an OVERLAPPED structure instance which will be used for asynchronous operation. If you opened the device as synchronous, pass NULL.

3.3 DeviceList class

DeviceList class is used to enumerate CeUsb2 devices connected to the system. It searches devices and puts the found ones into an internal link list. Device information is saved as CeUsb2 class with device link name and friendly name. The user can use the index operator of DeviceList class and access its link list as an array of CeUsb2 objects. It is important to check the number of devices found, before using the index operator. No error check is done for this operation.

3.3.1 DeviceList::DeviceList

```
DeviceList ();
```

Description : Standard constructor for DeviceList class.

Return Value : None

Arguments : None

3.3.2 DeviceList::DeviceList (2)

```
DeviceList(const GUID &Guid);
```

Description : Constructor for DeviceList class with a 128 bit GUID argument. This GUID is the identifier of CeUs2 device interface.

Return Value : None

Arguments :

Guid - Interface GUID.

Note : GUID definitions can be found in guides.h header file.

3.3.3 DeviceList::~DeviceList

```
virtual ~DeviceList ();
```

Description : Standard destructor for DeviceList class.

Return Value : None

Arguments : None

3.3.4 DeviceList::Init

```
DWORD Init();
```

Description : Searches for the CeUsb2 devices, and puts the found ones in a link list.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.3.5 DeviceList::operator[]

```
CeUsb2 operator[](UINT Index);
```

Description : Index operator for DeviceList class.

Return Value : Device class (CeUsb2) instance at the given index.

Arguments :

Index - 0 based device index.

3.3.6 DeviceList::GetNumDevices

```
UINT GetNumDevices();
```

Description : Retrieves number of CeUsb2 devices found in the system.

Return Value : Number of devices.

Arguments : None

3.4 *Descriptor class*

Descriptor class is an upper level wrapper interface for CeUsb2 USB descriptor handling. It can retrieve information about device, configuration, interface, endpoint and string descriptors. However, it can't set descriptors, and can't change current USB configuration, interface and alternate setting.

3.4.1 Descriptor::Descriptor

```
Descriptor(void);
```

Description : Standard constructor for Descriptor class.

Return Value : None

Arguments : None

3.4.2 Descriptor::Descriptor (2)

```
Descriptor(CeUsb2 *pDevice);
```

Description : Constructor for Descriptor class with device instance pointer argument.

Return Value : None

Arguments :

pDevice - CeUsb2 device instance pointer.

3.4.3 Descriptor::~Descriptor

```
virtual ~Descriptor();
```

Description : Standard destructor for Descriptor class.

Return Value : None

Arguments : None

3.4.4 Descriptor::Init

```
DWORD Init(void);
```

Description : Initializes descriptor class. Retrieves device descriptor, configuration descriptors, language id, and reserves memory for internal buffers. If this functions fails, don't use other member functions of the Descriptor class.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.4.5 Descriptor::GetNumConfigurations

```
UCHAR GetNumConfigurations();
```

Description : Retrieves number of configurations this CeUsb2 device – firmware pair includes.

Return Value : Number of configurations.

Arguments : None

3.4.6 Descriptor::GetNumInterfaces

```
UCHAR GetNumInterfaces(UCHAR ConfigNo);
```

Description : Returns the number of interfaces a USB configuration has.

Return Value : Number of interfaces.

Arguments :

ConfigNo - Configuration number

3.4.7 Descriptor::GetDeviceDescriptor

```
void GetDeviceDescriptor(PUSB_DEVICE_DESCRIPTOR pDevDesc);
```

Description : Retrieves the USB device descriptor.

Return Value : None

Arguments :

pDevDesc - Buffer for device descriptor.

3.4.8 Descriptor::GetConfigurationDescriptor

```
void GetConfigurationDescriptor(  
    UCHAR ConfigNo,  
    PUSB_CONFIGURATION_DESCRIPTOR pConfDesc  
);
```

Description : Retrieves the USB configuration descriptor excluding the interface and endpoint descriptors.

Return Value : None

Arguments :

ConfigNo – USB Configuration number.

pConfDesc - Buffer for USB configuration descriptor.

3.4.9 Descriptor::GetInterfaceDescriptor

```
DWORD GetInterfaceDescriptor(  
    UCHAR ConfigNo,  
    UCHAR InterfaceNumber,  
    UCHAR AlternateSetting,  
    PUSB_INTERFACE_DESCRIPTOR pInterfaceDesc  
);
```

Description : Retrieves the USB interface descriptor excluding the endpoint descriptors it has.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

ConfigNo - Configuration number.

InterfaceNumber - Interface number.

AlternateSetting - Alternate setting number.

pInterfaceDesc - Buffer for interface descriptor.

3.4.10 Descriptor::GetEndpointDescriptor

```
DWORD GetEndpointDescriptor(  

```

```
    UCHAR ConfigNo,  
    UCHAR InterfaceNumber,  
    UCHAR AlternateSetting,  
    UCHAR EndpointNumber,  
    PUSB_ENDPOINT_DESCRIPTOR pEndpDesc  
);
```

Description : Retrieves an USB endpoint descriptor.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

ConfigNo - Configuration number.

InterfaceNumber - Interface number.

AlternateSetting - Alternate setting number.

EndpointNumber - Endpoint number.

plnterfaceDesc - Buffer for endpoint descriptor.

3.4.11 Descriptor::GetStringDescriptor

```
DWORD GetStringDescriptor(  
    LPTSTR pStr,  
    UCHAR StringIndex,  
    USHORT LanguageId=0  
);
```

Description : Retrieves an USB string descriptor.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pStr - String descriptor string (maximum length is 256 bytes).

StringIndex - 0 based string index.

LanguageId - Language id.

3.5 *UsbInterface class*

UsbInterface class is a wrapper around CeUsb2 class's interface handling functions. It can get information about the current USB interface which is active on the CeUsb2 board, including the communication pipes with their feature such as direction and transfer type (bulk, isochronous, control). However, it can't change the current USB configuration, interface and alternate setting.

3.5.1 UsbInterface::UsbInterface

```
UsbInterface(void);
```

Description : Standard constructor for UsbInterface class.
Return Value : None
Arguments : None

3.5.2 UsbInterface::UsbInterface (2)

```
UsbInterface (CeUsb2 *pDevice);
```

Description : Constructor for UsbInterface class with device instance pointer argument.
Return Value : None
Arguments :
 pDevice - CeUsb2 device instance pointer.

3.5.3 UsbInterface::~UsbInterface

```
virtual ~UsbInterface();
```

Description : Standard destructor for UsbInterface class.
Return Value : None
Arguments : None

3.5.4 UsbInterface::Init

```
DWORD Init();
```

Description : Initializes UsbInterface class, reserves memory for buffers, and retrieves information about current interface from the driver.
Return Value : 32 bit status code if an error occurs, else 0.
Arguments : None

3.5.5 UsbInterface::GetInterfaceNumber

```
UCHAR GetInterfaceNumber();
```

Description : Returns the number (index) of the current interface.
Return Value : Interface number.
Arguments : None

3.5.6 UsbInterface::GetAlternateSetting

```
UCHAR GetAlternateSetting ();
```

Description : Returns the number (index) of the current alternate setting.

Return Value : Alternate setting number.

Arguments : None

3.5.7 UsbInterface::GetNumPipes

```
ULONG GetNumPipes();
```

Description : Returns Returns the number of pipes this interface has.

Return Value : Number of pipes.

Arguments : None

3.5.8 UsbInterface::GetPipePtr

```
PUSBD_PIPE_INFORMATION GetPipePtr(ULONG Index);
```

Description : Returns a pointer to a pipe information structure with the given pipe index.

Return Value : Pointer to pipe information structure, USBD_PIPE_INFORMATION or NULL if there is not a valid pipe with the given index.

Arguments :

Index - Pipe index.

3.6 GPIF class

GPIF class of the CeUsb2 API is used to control some features of the GPIF engine. GPIF settings can be changed by accessing the public member variables of the class. LoadGpifInitData and LoadGpifWaveformData functions update the initialization data array and waveform data. Changes done over the GPIF class will not take affect until the GpifRefresh member function is called.

3.6.1 Public Member Variables

m_IfClockSource 1 byte unsigned character value. It specifies the GPIF interface clock source (0 = external, 1 = internal).

m_IfClockSpeed 1 byte unsigned character value. It specifies the GPIF interface clock speed (0 = 30, 1 = 48 MHz).

m_IfClockOe 1 byte unsigned character value. It specifies the GPIF interface clock output enable (0 = disable, 1 = enable).

m_IfClockInverted 1 byte unsigned character value. It specifies the polarity of the GPIF interface clock (0 = normal, 1 = inverted).

m_GpifAuto 1 byte unsigned character value. It specifies the GPIF transfer mode (0 = normal mode, 1 = auto mode).

m_GpifWordwide 1 byte unsigned character value. It specifies the data mode of the GPIF (0 = 8 bit data, 1 = 16 bit word-wide data).

3.6.2 GPIF::GPIF

```
GPIF();
```

Description : Standard constructor for GPIF class.

Return Value : None

Arguments : None

3.6.3 GPIF::GPIF (2)

```
GPIF (CeUsb2 *pDevice);
```

Description : Constructor for GPIF class with device instance pointer argument.

Return Value : None

Arguments :

pDevice - CeUsb2 device instance pointer.

3.6.4 GPIF::~~GPIF

```
virtual ~GPIF();
```

Description : Standard destructor for GPIF class.

Return Value : None

Arguments : None

3.6.5 GPIF::Init

```
DWORD Init();
```

Description : Initializes the GPIF class. It reads current CPU & GPIF settings and GPIF Init & Waveform data from the firmware and stores all this information. Call this function before calling the other functions of the GPIF class.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.6.6 GPIF::Refresh

```
DWORD Refresh();
```

Description : Sends the current GPIF settings, GPIF Init & Waveform data to the firmware. This function reinitializes the firmware automatically.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.6.7 GPIF::Restore

```
DWORD Restore();
```

Description : Sends the initially stored CPU & GPIF settings and GPIF Init & Waveform data to the firmware. This function reinitializes the firmware automatically.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.6.8 GPIF::LoadGpifInitData

```
void LoadGpifInitData(PUCHAR pData, UCHAR Length);
```

Description : Updates the current GPIF Initialization data. Changes will not take affect before the Refresh function is called.

Return Value : None

Arguments :

pData - GPIF Initialization data.

Length - GPIF Initialization data length (should not be more than 7).

3.6.9 GPIF::LoadGpifWaveformData

```
void LoadGpifWaveformData(PUCHAR pData, UCHAR Offset, UCHAR Length);
```

Description : Updates the current GPIF waveform data or a part of it. Changes will not take affect before the Refresh function is called.

Return Value : None

Arguments :

pData - GPIF waveform data

Offset - GPIF waveform data offset (0 - 127).

Length - GPIF waveform data length (max 128).

3.6.10 GPIF::GpifSingleRead

```
DWORD GpifSingleRead(  
    PUCHAR    pData,  
    USHORT    Address,  
    USHORT    Length,  
    BOOL      IncAddress = TRUE  
);
```

Description : Reads specified amount of data from the firmware using GPIF single read transitions.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pData - Read data buffer.

Address - Offset (Address) to read from.

Length - Read data buffer length.

IncAddress - If TRUE, address is incremented with each GPIF transaction.

3.6.11 GPIF::GpifSingleWrite

```
DWORD GpifSingleWrite(  
    PUCHAR    pData,  
    USHORT    Address,  
    USHORT    Length,  
    BOOL      IncAddress = TRUE  
);
```

Description : Writes specified amount of data to the firmware using GPIF single write transitions.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pData - Write data buffer.

Address - Offset (Address) to write to.

Length - Write data buffer length.

IncAddress - If TRUE, address is incremented with each GPIF transaction.

3.6.12 GPIF:: GpifSingleWrite_Wordwide

```
DWORD GpifSingleWrite_Wordwide (  
    PUSHORT    pData,  
    USHORT     Address,  
    USHORT     Length,  
    BOOL       IncAddress = TRUE  
);
```

Description : Writes specified amount of data to the firmware using GPIF single word write transitions (16 bit data).

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pData - Write data buffer.

Address - Offset (Address) to write to.

Length - Write data buffer length (number of words).

IncAddress - If TRUE, address is incremented with each GPIF transaction.

3.6.13 GPIF:: GpifSingleRead_Wordwide

```
DWORD GpifSingleRead_Wordwide (  
    PUSHORT    pData,  
    USHORT     Address,  
    USHORT     Length,  
    BOOL       IncAddress = TRUE  
);
```

Description : Reads specified amount of data from the firmware using GPIF single word write transitions (16 bit data).

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pData - Read data buffer.

Address - Offset (Address) to write to.

Length - Read data buffer length (number of words).

IncAddress - If TRUE, address is incremented with each GPIF transaction.

3.6.14 GPIF::GpifAbort

```
DWORD GpifAbort();
```

Description : Aborts the current GPIF transition.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.6.15 GPIF::GetGpifStatus

```
DWORD GetGpifStatus(UCHAR *pStatus);
```

Description : Retrieves the current status of the GPIF.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pStatus - GPIF status register(8 bits). GPIF Status registers bits are:

7 = GPIF done bit (0 = GPIF busy, 1 = GPIF idle)

5-0 = Instantaneous states of the RDY pins. The current state of the RDY[5:0] pins, sampled at each rising edge of the GPIF clock.

Bit 6 is not used.

3.7 FpgaConfig class

FpgaConfig class serves raw level FPGA access methods for initializing the FPGA before configuration and checking the configuration status. It can also send a reset signal to the currently active FPGA design. Fpga class which is explained in the next section is derived from FpgaConfig class and has an upper level interface for FPGA configuration process.

3.7.1 FpgaConfig::FpgaConfig

```
FpgaConfig ();
```

Description : Standard constructor for FpgaConfig class.

Return Value : None

Arguments : None

3.7.2 FpgaConfig::FpgaConfig (2)

```
FpgaConfig (CeUsb2 *pDevice);
```

Description : Constructor for FpgaConfig class with device instance pointer argument.

Return Value : None

Arguments :

pDevice - CeUsb2 device instance pointer.

3.7.3 FpgaConfig::~FpgaConfig

```
virtual ~FpgaConfig ();
```

Description : Standard destructor for FpgaConfig class.

Return Value : None

Arguments : None

3.7.4 FpgaConfig::Init

```
DWORD Init(void);
```

Description : Clears the FPGA configuration and initializes it for configuration.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.7.5 FpgaConfig::IsConfigured

```
DWORD IsConfigured (void);
```

Description : Checks whether the FPGA configured successfully or not.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.7.6 FpgaConfig::Reset

```
DWORD Reset(void);
```

Description : Resets FPGA. Actually it sends a reset signal to the FPGA.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.8 Fpga class

Fpga class is derived from FpgaConfig class and implements an upper level and easier to use interface for FPGA configuration process. Internally it uses the raw functions of the FpgaConfig class.

Fpga class can configure an FPGA with a given configuration file path. It can use either USB control pipe or another faster USB pipe (bulk or isochronous) for sending configuration data to the FPGA. Also, if no pipe number is given through its constructor (e.g. pipe number is -1), it can search for a valid pipe for data transfer.

3.8.1 Fpga::Fpga

```
Fpga ();
```

Description : Standard constructor for Fpga class.

Return Value : None

Arguments : None

3.8.2 Fpga::Fpga (2)

```
Fpga( CeUsb2*pDevice,  
      LPCTSTR      lpcConfigFile,  
      BOOL         bControlTransfer = FALSE,  
      int          PipeNum = -1,  
      BOOL         bUseDynamicGpifWvf = TRUE,  
      ULONG        PackageSize = ULONG_MAX  
      );
```

Description : Constructor for Fpga class with device instance pointer argument.

Return Value : None

Arguments :

pDevice - CeUsb2 device instance pointer.

lpcConfigFile - Configuration file path. Currently files with exo and rbt (raw bit file) are supported.

bControlTransfer – If TRUE, configuration is handled over USB control pipe, else it is handled over an USB bulk or isochronous pipe.

PipeNum - USB pipe number. If -1, a valid pipe is searched automatically.

bUseDynamicGpifWvf - Uses internally stored dynamic GPIF waveform data and settings for FPGA configuration. Previous states are restored after the configuration.

PackageSize - USB data transfer packet size. Use 0xFFFFFFFF for automatic size detection. This argument is used for the firmwares which use bulk pipes to configure the FPGA.

3.8.3 Fpga::~Fpga

```
virtual ~Fpga ();
```

Description : Standard destructor for Fpga class.

Return Value : None

Arguments : None

3.8.4 Fpga::Configure

```
DWORD Configure(void);
```

Description : Configures the FPGA with the configuration file name specified by the constructor.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments : None

3.9 RS232 class

RS232 class of the CeUsb2 API is an upper level interface for serial input output. TxSend member function sends specified amount of data to the SIO (serial I/O). SIO reading is done by an internal data capturing thread. RxStart starts this thread and takes a pointer to a user defined RX notification function which will be called by the thread every time data is received over RS232.

3.9.1 RS232::RS232

```
RS232();
```

Description : Standard constructor for RS232 class.

Return Value : None

Arguments : None

3.9.2 RS232::RS232 (2)

```
RS232(CeUsb2 *pDevice);
```

Description : Constructor for RS232 class with device instance pointer argument.

Return Value : None

Arguments :

pDevice - CeUsb2 device instance pointer.

3.9.3 RS232::~~RS232

```
virtual ~RS232();
```

Description : Standard destructor for RS232 class.

Return Value : None

Arguments : None

3.9.4 RS232::TxSend

```
DWORD TxSend(  
    PCHAR      pDataBuf,  
    UCHAR      BufLength,  
    BOOL       bSendAsHexDigits = FALSE  
);
```

Description : Sends specified amount of data to RS232.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDataBuf - Data buffer

BufLength - Data buffer length

bSendAsHexDigits - If TRUE, data is send as hex digits over serial line otherwise no conversion is performed

3.9.5 RS232::RxStart

```
DWORD RxStart(  
    RX_NOTIFY *pNotifyProc,  
    PVOID      pNotifyContext  
);
```

Description : Starts data capturing over RS232.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pNotifyProc - RS232 RX (receive) notify function. This function is called whenever data is received over RS232 (See also 3.8.7, RxNotify).

pNotifyContext - Context that will be passed to the RX notify function.

3.9.6 RS232::RxStop

```
void RxStop();
```

Description : Stops data capturing over RS232.

Return Value : None.

Arguments : None.

3.9.7 RxNotify

```
typedef void *(RX_NOTIFY(PVOID,DWORD,PCHAR,UCHAR));
```

```
// user RX notify function decoration
```

```
void RxNotify(  
    PVOID        pContext,           // Context passed to RxStart  
    DWORD        dwStatus,          // 32 bit status code if an error occurs, else 0  
    PCHAR        pBuffer,           // Data buffer  
    UCHAR        Length,            // Buffer length  
);
```

Description : CeUsb2 RX notify function. User of RxStart function should pass a function pointer with the decoration of RxNotify. This function will be called every time data is received on RS232.

Return Value : None.

Arguments :

pContext - Context passed to RxStart function.

dwStatus - 32 bit status code if an error occurs, else 0.

pBuffer - RX data buffer.

Length - Data buffer length.

3.10 CeUsb2 API Global Functions

3.10.1 GetApiInfo

```
void CEUSB2API_API GetApiInfo(CEUSB2_API_INFO &rInfo);
```

Description : Retrieves versioning information for CeUsb2 API.

Return Value : None.

Arguments :

rInfo - Reference to an instance of CeUsb2 API information structure, CEUSB2_API_INFO.

3.10.2 GetFirmwareInfo

```
DWORD CEUSB2API_API GetFirmwareInfo(  
    CeUsb2 *pDevice,  
    PCEUSB2_FIRMWARE_INFO pFwInfo,  
    LPSTR DeviceTypeStr,  
    LPSTR FirmwareNameStr  
);
```

Description : Retrieves versioning information of the current 8051 firmware running on the board.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

pFwInfo - Pointer to an instance of firmware information structure CEUSB2_FIRMWARE_INFO.

DeviceTypeStr - String identifying the CeUsb2 device type.

FirmwareNameStr - Firmware name string (.hex).

3.10.3 IsHighSpeed

```
DWORD CEUSB2API_API IsHighSpeed(  
    CeUsb2 *pDevice,  
    BOOL *pbIsHighSpeed  
);
```

Description : Checks whether the device is an USB high speed device or not.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

pbIsHighSpeed - Pointer to a boolean variable. TRUE, if the device is an USB high-speed device; else FALSE.

3.10.4 FormatErrorString

```
void CEUSB2API_API FormatErrorString(LPCTSTR lpErrorStr,DWORD dwError);
```

Description : Formats a string with the description of a given error code.

Return Value : None.

Arguments :

lpErrorStr - Error string buffer.
dwError - 32 bit error code.

3.10.5 FormatUSBDErrorStr

```
void CEUSB2API_API FormatUSBDErrorStr(  
    LPTSTR lpErrorStr,  
    ULONG USBDStatus  
);
```

Description : Formats a string with the description of a given USB D error code.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

lpErrorStr - USB D error string buffer.

USB DStatus - 32 bit error code.

3.10.6 SetCpuSettings

```
DWORD CEUSB2API_API SetCpuSettings(  
    CeUsb2*    pDevice,  
    UCHAR      CpuClkSpeed,  
    UCHAR      CpuClkInv,  
    UCHAR      CpuClkOe  
);
```

Description : Set FX2 CPU parameters. Changes will not take effect before the FwReinit global function or Refresh function of the GPIF class is called.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

CpuClkSpeed - CPU clock speed (0 - 12 MHz, 1 - 24 MHz, 2 - 48 MHz, others invalid).

CpuClkInv - CPU clock invert parameter (0 - not inverted, else inverted).

CpuClkOe - CPU clock output enable parameter (0 - not enabled, else enabled).

3.10.7 GetCpuSettings

```
DWORD CEUSB2API_API GetCpuSettings(  
    CeUsb2*    pDevice,  
    UCHAR*     pCpuClkSpeed,  
    UCHAR*     pCpuClkInv,
```

```
    UCHAR*    pCpuClkOe
);
```

Description : Retrieves FX2 CPU parameters.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

pCpuClkSpeed - CPU clock speed (0 - 12 MHz, 1 - 24 MHz, 2 - 48 MHz, others invalid).

pCpuClkInv - CPU clock invert parameter (0 - not inverted, else inverted).

pCpuClkOe - CPU clock output enable parameter (0 - not enabled, else enabled).

3.10.8 FwReinit

```
DWORD CEUSB2API_API FwReinit(CeUsb2 *pDevice);
```

Description : Re-initializes the firmware.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

3.10.9 FwSoftReset

```
DWORD CEUSB2API_API FwSoftReset(CeUsb2 *pDevice);
```

Description : Performs a soft reset (vector to org 00h) in the firmware.

Return Value : 32 bit status code if an error occurs, else 0.

Arguments :

pDevice - Pointer to an instance of device handler class, CeUsb2.

3.11 CeUsb2 API structures

CeUsb2 API uses the structures defined in CeUsb2 IOCTL interface. For more information see CeUsb2 driver specifications document.

3.11.1 CEUSB2_API_INFO

```
typedef struct _CEUSB2_API_INFO
{
    CHAR    ApiName[32];
```

```
    UCHAR    MajorVersion;  
    UCHAR    MinorVersion;  
    ULONG    BuildNumber;  
}CEUSB2_API_INFO,*PCEUSB2_API_INFO;
```

Description : Versioning information structure for this API.

Members :

ApiName – 32 byte API name string.

MajorVersion – Major version number of the API.

MinorVersion – Minor version number of the API.

BuildNumber – Build number of the API.

3.12 CeUsb2 API Macros

3.12.1 CE_SUCCESS

```
CE_SUCCESS (Status)
```

Description : Tests a status code.

Return value : Boolean value. FALSE if it the status code is an error code; else TRUE.

Arguments :

Status - Status code to be tested (32 bit unsigned long).

3.12.2 GET_NUM_CONFIGURATIONS

```
GET_NUM_CONFIGURATIONS(pDevDesc)
```

Description : Retrieves the number of USB configurations that the device includes.

Return value : Number of configurations (8 bit unsigned character).

Arguments :

pDevDesc - Pointer to device descriptor (PUSB_DEVICE_DESCRIPTOR).

3.12.3 GET_NUM_INTERFACES

```
GET_NUM_INTERFACES(pConfigDesc)
```

Description : Retrieves the number of USB interfaces that a USB configuration has.

Return value : Number of interfaces (8 bit unsigned character).

Arguments :

pConfigDesc - Pointer to configuration descriptor (PUSB_CONFIGURATION_DESCRIPTOR).

3.12.4 GET_NUM_ENDPOINTS

GET_NUM_ENDPOINTS(pInterfaceDesc)

Description : Retrieves the number of USB endpoints that an interface has.

Return value : Number of endpoints (8 bit unsigned character).

Arguments :

pInterfaceDesc - Pointer to interface descriptor (PUSB_INTERFACE_DESCRIPTOR).

3.12.5 GET_INTERFACE_DESC_SIZE

GET_INTERFACE_DESC_SIZE(pInterfaceDesc)

Description : Calculates the buffer size for an interface descriptor including its endpoint descriptors.

Return value : Interface descriptor buffer size (32 bit unsigned long).

Arguments :

pInterfaceDesc - Pointer to interface descriptor (PUSB_INTERFACE_DESCRIPTOR).

3.13 CeUsb2 API Error Codes

Error Code : CEUSB2_STATUS_SUCCESS, 0x00000000

Description : Operation successful

Reported by : Driver, API

Error Code : CEUSB2_STATUS_ERROR, 0xE000E001

Description : Undefined CeUsb2 error

Reported by : Driver, API

Error Code : CEUSB2_STATUS_USBD_ERROR, 0xE000E002

Description : USB error

Reported by : Driver

Error Code : CEUSB2_STATUS_NO_KERNEL_MEMORY, 0xE000E003

Description : CeUsb2 driver is unable to allocate kernel memory

Reported by : Driver

Error Code : CEUSB2_STATUS_NO_USER_MEMORY, 0xE000E004
Description : CeUsb2 driver is unable to allocate user-mode memory
Reported by : Driver

Error Code : CEUSB2_STATUS_UNKNOWN_REQUEST, 0xE000E005
Description : Request is not recognized by the CeUsb2 driver
Reported by : Driver

Error Code : CEUSB2_STATUS_INVALID_BUFFER_SIZE, 0xE000E007
Description : Specified buffer size is invalid.
Reported by : Driver,API

Error Code :
CEUSB2_STATUS_UNABLE_TO_GET_INTERFACE_DESCRIPTOR,
0xE000E00A
Description : Unable to get requested USB interface descriptor
Reported by : Driver

Error Code :
CEUSB2_STATUS_UNABLE_TO_PARSE_CONFIG_DESCRIPTOR,
0xE000E00B
Description : Unable to parse the configuration descriptor
Reported by : Driver

Error Code : CEUSB2_STATUS_UNABLE_TO_ALLOCATE_IRP, 0xE000E00C
Description : Failed to reserve memory for request package.
Reported by : Driver

Error Code : CEUSB2_STATUS_ISO_STREAM_ALREADY_STARTED,
0xE000E00D
Description : An attempt is made to start isochronous streaming but it is already started.
Reported by : Driver

Error Code : CEUSB2_STATUS_TIMEOUT, 0xE000E00E
Description : Operation timed out.
Reported by : Driver

Error Code : CEUSB2_STATUS_UNKNOWN_FIRMWARE_STATUS,
0xE000E00E
Description : The status of the firmware is unknown.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_INVALID_PIPE, 0xE000E030
Description : Pipe is invalid.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_INVALID_PIPE_NUMBER, 0xE000E031
Description : Pipe number is invalid.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_INVALID_PIPE_HANDLE, 0xE000E032
Description : Returned (or stored) handle for the USB pipe is invalid.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_PIPE_NOT_CONTROL, 0xE000E035
Description : Specified pipe is not a USB control pipe.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_PIPE_NOT_ISOCHRONOUS, 0xE000E036
Description : Specified pipe is not a USB isochronous pipe.
Reported by : Driver, API

Error Code : CEUSB2_STATUS_PIPE_NOT_BULK, 0xE000E037
Description : Specified pipe is not a USB bulk pipe.
Reported by : Driver, API

Error Code : CESUB2_STATUS_INVALID_INTERFACE, 0xE000E039
Description : Specified USB interface is invalid.
Reported by : Driver, API

Error Code : CEUSB2LD_STATUS_UNABLE_TO_LOCK_DEVICE,
0xE000E080
Description : Device lock or unlock operation failed.
Reported by : Loader Driver

Error Code : CEUSB2LD_STATUS_INVALID_FW, 0xE000E081
Description : Invalid firmware.
Reported by : Loader Driver, API

Error Code : CEUSB2_API_STATUS_SUCCESS, 0x00000000
Description : Operation successful.
Reported by : API

Error Code : CESUB2_API_STATUS_NOT_ENOUGH_MEMORY, 0x000000A0
Description : Unable to reserve user-mode memory.
Reported by : API

Error Code : CESUB2_API_STATUS_DATA_BUFFER_TOO_BIG, 0x000000A1

Description : Data buffer size is too big.
Reported by : API

Error Code : CESUB2_API_STATUS_DATA_BUFFER_TOO_SMALL,
0x000000A2

Description : Data buffer size is too small.
Reported by : API

Error Code : CEUSB2_API_STATUS_INVALID_ENDPOINT_NUMBER,
0x000000A3

Description : Specified endpoint number is invalid.
Reported by : API

Error Code : CEUSB2_API_STATUS_INVALID_FPGA_FILE_EXTENSION,
0x000000A4

Description : Unknown or invalid FPGA file extension.
Reported by : API

Error Code : CEUSB2_API_STATUS_INVALID_HEX_LINE_START, 0x000000A5

Description : Error parsing the hex file: Invalid line start.
Reported by : API

Error Code : CEUSB2_API_STATUS_NO_VALID_PIPE, 0x000000A6

Description : Unable to find a valid pipe for operation
Reported by : API

Error Code : CEUSB2_API_STATUS_UNABLE_TO_OPEN_FILE, 0x000000A7

Description : Failed to open the file.
Reported by : API

Error Code : CEUSB2_API_STATUS_THREAD_ALREADY_RUNNING,
0x000000A8

Description : Attempt made to start a thread, but it is already running.
Reported by : API

Error Code : CEUSB2_API_STATUS_SIZE_EXCEEDED, 0x000000A9

Description : Specified size is exceeded.
Reported by : API

Error Code : CEUSB2_API_STATUS_NO_DEVICES_FOUND, 0x000000AA

Description : Unable to find any CeUsb2 or CeUsb2 compatible devices in the system.
Reported by : API