General

Language compatibility

The CEUSB3 API is designed to work with C++ native code to ensure best performance. Due to the wide popularity the library is compatible with Visual C++ 7.1 and higher versions. In addition to this, a wrapper for the .NET Framework 1.1 is also available, so applications written in C++ NET, C# and Visual Basic have access to the API too. The NET wrapper consists of the same classes and methods as the C++ API, but global functions, macros and constants are encapsulated in an additional class (ceUSB3API), based on the fact that NET doesn't support this. Furthermore some data types are not available in all languages, so a few of the parameters have a different value type in the C++ and .NET API. The best place to analyze the differences are the sample sources that ship with the API. The test application shows many parts from the API and is available in C++ native (cntest), C++ NET (cnettest), C# (cstest) and VB (vbtest).

Backward compatibility

The CEUSB3 API is newly designed so there's no compatibility with API's from USB2FPGA or other devices.

ceUSB3 C++ API specs

Basics

The API contains the necessary library and include files. To use the API you have to follow the steps below:

- Include the main header file (ceusb3api.h).
- Link the executable with the main library (ceusb3api.lib).

The whole API is located in namespace ceUSB3, so either tell the compiler to use this namespace (using namespace ceUSB3;) or scope all elements with this namespace separately (e.g. ceUSB3::ceDevice *pDev = ceUSB3::ceDevice::GetDevice(0);).
Pointers retrieved by the API must NOT be deleted, this is done by the API internally. Affected classes are ceDevice and ceInfo.

Error handling

Most functions return a HRESULT code, so you can use the SUCCEEDED() and FAILED() macros defined in the windows API. To retrieve a printable error string from a failed call use GetHRESULTMessage(), which returns the description string of a given error code (Only error codes used by the API).

ceUSB3 NET API specs

Basics

The API can be used by adding a new reference to the project, choose the file browser there and select ceusb3apinet.dll. To be able to use the classes, namespace cesys.ceUSB3NET must be used, the syntax is based on the NET language that is used.

Error handling

Most functions return a System::Int32 code, the C++ macros for error checking are encapsulated in two static methods, ceUSB3API.ceSUCCEEDED() and ceUSB3API.ceFAILED(). In addition, all possible error codes used by the API are defined as constants in that class (ceUSBAPI.ceS_* / ceUSBAPI.ceE_*). To retrieve a printable error string from a failed call use ceUSB3API.GetHRESULTMessage(), which returns the description string of a given error code (Only error codes used by the API).

Additional differences to the C++ API

Because NET doesn't support global functions, Init() and DeInit() are encapsulated in class ceUSB3API too. Furthermore Init() doens't expect a GUID, but a value from the ceUSB3API.ceDeviceType enumeration.

How to use the API

Initialization / Deinitialization

To use the API it must be initialized, this is done by a call to Init() (NET: ceUSB3API.Init()). This function searches for all devices plugged to the computer which matches the given GUID (NET: ceUSB3API.ceDeviceType). You can call this function with different GUID's which builds an internal list of all of them.
After using the API it must be freed, this is done by calling DeInit() (NET: ceUSB3API::DeInit()). To detect any changes in the list of connected devices, you have to call DeInit() and start again with one or more Init() - calls. This forces a reenumeration of all devices. **Attention! This invalidates all pointers you get from the API!**

How to communicate with devices

If the API is initialized correctly, you can retrieve the count of available devices by calling the static member function GetDeviceCount() from class ceDevice. To access one of the devices, call function GetDevice() from the same class and use an index in range of 0 < index < GetDeviceCount() to specify one of the devices. The pointer returned by this function is constant and valid until you call DeInit() (the same call will return the same pointer, so you do not have to store this pointer anywhere).
All communication with the device is done using this class pointer. Before any data can be sent or received, the device must be opened. To do this call method Open() which internally opens the device, set default parameters and retrieves some information about the device. After a successful call to this function you can do those things (descriptions below):

- Configure device (SetGPIFSpeed()).
- Retrieve information (GetInfo()).
- Download FPGA designs (ProgramFPGA()).
- Read and write FPGA registers (ReadRegister()/WriteRegister()).
- Read and write huge blocks of data (ReadBulk()/WriteBulk()).
- Read and write parts of the EEPROM (ReadEeprom()/WriteEeprom()).

To properly finish the use of the device, call Close().

Function description (methods in alphabetic order)

All methods are listed twice, the first one is the C++ native notation, the other one is the NET counterpart in C# notation.

Global functions (NET: class ceUSB3API)

C++:	`void DeInit()`
NET:	`void ceUSB3API.DeInit()`
Info	Frees all resources allocated by Init(), this must be called after using the API.
Returns	-
Errors	-

C++:	`const char *GetHRESULTMessage(HRESULT hr)`
NET:	`String ceUSB3API.GetHRESULTMessage(System.Int32 hr)`
Info	Returns the error string bound to the given error code hr.
Returns	Error string
Errors	-

C++:	`HRESULT Init(const GUID &Guid)`
NET:	`Int32 ceUSBAPI.Init(ceUSB3API.DeviceType T)`
Info	Initializes the API and searches for devices with the given type (via GUID or device type). The function can be called multiple times with different types to enumerate and use different devices. Resources allocated by that call must be freed by calling DeInit() after use. Possible GUID's (C++): **GUID_INTERFACE_CEUSB3** **GUID_INTERFACE_PSAA4096V2** **GUID_INTERFACE_ADCMI3** Possible Types (NET): **ceDT_CEUSB3** **ceDT_PSAA4096V2** **ceDT_ADCMI3**
Returns	Error code
Errors	`S_OK:` no error `E_FAIL:` error searching devices

class ceDevice

C++:	`HRESULT ceDevice::AbortPipe(uint uiPipeNumber)`
NET:	`Int32 ceDevice.AbortPipe(UInt32 uiPipeNumber)`
Info	Forces the USB bus driver to abort the transfer on a given pipe (uiPipeNumber).
Returns	Error code
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails `E_INVALIDARG:` uiPipeNumber is out of range

C++:	`void ceDevice::Close()`
NET:	`void ceDevice.Close()`
Info	Closes the device.
Returns	-
Errors	-

C++:	`HRESULT ceDevice::GetAsyncResult(ceAsyncHandle *pH, uint *uiTransfered)`
NET:	`Int32 ceDevice::GetAsyncResult(ref ceAsyncHandle pH, ref UInt32 uiTransfered)`
	This method is needed when using any of the following functions using the async call convention: ReadBulk(), WriteBulk(). Usage: After starting an async operation, use the async handle (pH) to check if the transfer is complete. Afterwards you have to call GetAsyncResult() to cleanup the call and retrieve the count of bytes transferred via this operation (uiTransfered). A good example on how to use this can be found in the test application that ships with the API, which is available in all supported languages.
Returns	Error code
Errors	`S_OK:` no error `E_FAIL:` the function fails `E_INVALIDARG:` pH is NULL

C++:	`ceDevice *ceDevice::GetDevice(uint uiIdx)`
NET:	`ceDevice ceDevice.GetDevice(UInt32 uiIdx)`
Info	Returns a pointer to a device which is selected by a zero based index (uiIdx). This pointer is valid until DeInit() is called. Never try to delete this object, this is done automatically.
Returns	Pointer to device with the given index, NULL otherwise.
Errors	-

C++:	`ceDevice *ceDevice::GetDevice(uint uiIdx)`
NET:	`ceDevice ceDevice.GetDevice(UInt32 uiIdx)`
Info	Returns a pointer to a device which is selected by a zero based index (uiIdx). This pointer is valid until DeInit() is called. Never try to delete the returned object, this is done automatically.
Returns	Pointer to device with the given index, NULL otherwise.
Errors	-

C++:	`uint ceDevice::GetDeviceCount()`
NET:	`UInt32 ceDevice.GetDeviceCount()`
Info	Returns the count of devices find during the call of Init(), if Init() is called multiple times, the total number is returned.
Returns	Count of devices found in the system.
Errors	-

C++:	`uint ceDevice::GetDeviceCount()`
NET:	`UInt32 ceDevice.GetDeviceCount()`
Info	Returns the count of devices find during the call of Init(), if Init() is called multiple times, the total number is returned.
Returns	Count of devices found in the system.
Errors	-

C++:	`ceInfo *ceDevice::GetInfo()`
NET:	`ceInfo ceDevice.GetInfo()`
Info	Returns a static pointer to a ceInfo class instance bound to the device. This holds additional information about the device. Never try to delete the returned object, this is done by DeInit() automatically.
Returns	Pointer to info class.
Errors	-

C++:	`uint ceDevice::GetLastError()`
NET:	`UInt32 ceDevice.GetLastError()`
Info	Returns the last error occurred in the driver. This may help to find out unexpected errors.
Returns	Driver error code.
Errors	-

C++:	`uint ceDevice::GetLastFirmwareError()`
NET:	`UInt32 ceDevice.GetLastFirmwareError()`
Info	Returns the last error occurred in the firmware. This may help to find out unexpected errors.
Returns	Firmware error code.
Errors	-

C++:	`HRESULT ceDevice::Open()`
NET:	`Int32 ceDevice.Open()`
Info	Opens the device.
Returns	Error code.
Errors	`S_OK:` no error `S_FALSE:` device already open `E_FAIL:` error retrieving information from driver `E_OPEN:` failed to open device

C++:	`HRESULT ceDevice::ProgramFPGA(ceFPGA *pFPGA)`
NET:	`Int32 ceDevice.ProgramFPGA(ceFPGA pFPGA)`
Info	Downloads a FPGA design to the device. This should be the first step after opening the device. Without a running design the hardware won't do anything.
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails `E_INVALIDARG:` invalid design `E_NOPIPE:` no matching pipe found `E_FPGA_INIT:` fpga init pin doesn't switch `E_FPGA_NC:` fpga not configured

C++:	`HRESULT ceDevice::ProgramFPGA(ceFPGA *pFPGA)`
NET:	`Int32 ceDevice.ProgramFPGA(ceFPGA pFPGA)`
Info	Downloads a FPGA design (pFPGA) to the device. This should be the first step after opening the device. Without a running design the hardware won't do anything.
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails `E_INVALIDARG:` invalid design `E_NOPIPE:` no matching pipe found `E_FPGA_INIT:` fpga init pin doesn't switch `E_FPGA_NC:` fpga not configured

C++:
```
HRESULT ceDevice::ReadBulk(uchar *pucData, uint uiSize, uint
&uiTransfered, ceAsyncHandle *pH, uint uiPipe, uint
uiTimeOut)
```

NET:
```
Int32 ceDevice.ReadBulk(Byte[] pucData, UInt32 uiSize, ref
UInt32 uiTransfered, ref ceAsyncHandle pH, UInt32 uiPipe,
UInt32 uiTimeOut)
```

Info

This function should be used to transfer huge blocks of data from device to host. It is able to work in sync or async mode, depending on the given parameters. Parameter pucData should point to a buffer that is able to hold the requested data, while uiSize must be data count of bytes that should be received. The maximum allowed count of bytes in one call can be retrieved by method GetPipeBufferSize() from attached class ceInfo (use GetInfo() to get it). Furthermore this count must be dividable by 512.
If pH is NULL, than synced I/O is active, if pH is a valid async handle, async I/O will be used. Using synced I/O, uiTransfered will return the count of bytes transferred, which can be unequal to the requested transfer count, otherwise this return value is undefined.
To specify a special pipe for the transfer, uiPipe can be used, but in most cases a value of 0xffffffff let the API decide the best pipe. The last parameter, uiTimeOut is only valid using synced I/O, a timeout for transfer completion in milliseconds can be specified here.

Returns

Error code.

Errors

```
S_OK:               no error
E_OPEN:             device not open
E_FAIL:             call to driver fails
E_INVALIDARG:       invalid data ptr, uiSize = 0 or uiSize not dividable by 512
E_NOPIPE:           no matching pipe found/uiPipe invalid
E_TIMEOUT:          call is timed out (sync)
E_PENDING:          device is in pending mode (async)
```

C++:
```
HRESULT ceDevice::ReadEeprom(uint uiAddress, uchar *pucData,
uint uiSize)
```

NET:
```
Int32 ceDevice.ReadEeprom(uint uiAddress, Byte[] pucData,
UInt32 uiSize)
```

Info

Reads data from on board EEPROM. 7 KB are free for use, starting at address 0. Maximum transfer size is 4 KB. uiAddress sets the base offset, pucData should be huge enough to hold the requested data, while uiSize sets the count of bytes that should be transferred.

Returns

Error code.

Errors

```
S_OK:               no error
E_OPEN:             device not open
E_FAIL:             call to driver fails
E_INVALIDARG:       uiAddress+uiSize > 7k, uiSize>4096 or 0==pucData
```

C++:
```
HRESULT ceDevice::ReadRegister(ushort usAddress, ushort
&usValue)
```

NET:
```
Int32 ceDevice.ReadRegister(UInt16 usAddress, ref UInt16
usValue)
```

Info

Read the value of FPGA register usAddress, the result will be stored in usValue..

Returns

Error code.

Errors

```
S_OK:               no error
E_OPEN:             device not open
E_FAIL:             call to driver fails
```

C++:	`HRESULT ceDevice::ResetFPGA()`
NET:	`Int32 ceDevice.ResetFPGA()`
Info	Pulses the FPGA reset pin.
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails

C++:	`HRESULT ceDevice::ResetPipe(uint uiPipeNumber)`
NET:	`Int32 ceDevice.ResetPipe(UInt32 uiPipeNumber)`
Info	Forces the USB bus driver to reset pipe number uiPipeNumber.
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails `E_INVALIDARG:` uiPipeNumber out of range

C++:	`HRESULT ceDevice:SetGPIFSpeed(ceGPIFSpeed Speed)`
NET:	`Int32 ceDevice.SetGPIFSpeed(ceDevice.ceGPIFSpeed Speed)`
Info	Allows the adjustment of the GPIF speed between 30 and 48 MHz. Default value is 48 MHz. It is not necessary to change this value except for some special cases. Possible enumerators are: **`ceGPIFS_30MHz`** **`ceGPIFS_48MHz`**
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails `E_INVALIDARG:` uiPipeNumber out of range

C++:
```
HRESULT ceDevice::WriteBulk(uchar *pucData, uint uiSize, uint
&uiTransfered, ceAsyncHandle *pH, uint uiPipe, uint
uiTimeOut)
```

NET:
```
Int32 ceDevice.WriteBulk(Byte[] pucData, UInt32 uiSize, ref
UInt32 uiTransfered, ref ceAsyncHandle pH, UInt32 uiPipe,
UInt32 uiTimeOut)
```

Info

This function should be used to transfer huge blocks of data from host to device. It is able to work in sync or async mode, depending on the given parameters. Parameter pucData should point to the buffer which contains the data to send, while uiSize must be data count of bytes that should be transfered. The maximum allowed count of bytes in one call can be retrieved by method GetPipeBufferSize() from attached class ceInfo (use GetInfo() to get it). Furthermore this count must be even.
If pH is NULL, than synced I/O is active, if pH is a valid async handle, async I/O will be used. Using synced I/O, uiTransfered will return the count of bytes transferred, which can be unequal to the requested transfer count, otherwise this return value is undefined.
To specify a special pipe for the transfer, uiPipe can be used, but in most cases a value of 0xffffffff let the API decide the best pipe. The last parameter, uiTimeOut is only valid using synced I/O, a timeout for transfer completion in milliseconds can be specified here.

Returns

Error code.

Errors

S_OK:	no error
E_OPEN:	device not open
E_FAIL:	call to driver fails
E_INVALIDARG:	invalid data ptr, uiSize = 0 or uiSize not dividable by 512
E_NOPIPE:	no matching pipe found/uiPipe invalid
E_TIMEOUT:	call is timed out (sync)
E_PENDING:	device is in pending mode (async)

C++:
```
HRESULT ceDevice::WriteEeprom(uint uiAddress, uchar *pucData,
uint uiSize)
```

NET:
```
Int32 ceDevice.WriteEeprom(uint uiAddress, Byte[] pucData,
UInt32 uiSize)
```

Info

Writes data to on board EEPROM. 7 KB are free for use, starting at address 0. Maximum transfer size is 4 KB. uiAddress sets the base offset, pucData must hold the data, while uiSize sets the count of bytes that should be transferred.

Returns

Error code.

Errors

S_OK:	no error
E_OPEN:	device not open
E_FAIL:	call to driver fails
E_INVALIDARG:	uiAddress+uiSize > 7k, uiSize>4096 or 0==pucData

C++:	`HRESULT ceDevice::WriteRegister(ushort usAddress, ushort &usValue)`
NET:	`Int32 ceDevice.WriteRegister(UInt16 usAddress, ref UInt16 usValue)`
Info	Write value usValue to FPGA register usAddress..
Returns	Error code.
Errors	`S_OK:` no error `E_OPEN:` device not open `E_FAIL:` call to driver fails

class ceInfo

C++:	`const char *ceInfo::GetDeviceName()`
NET:	`String ceInfo.GetDeviceName()`
Info	Returns the name of the device (Same name as listed in the device manager).
Returns	Requested information.
Errors	-

C++:	`const char *ceInfo::GetDevicePath()`
NET:	`String ceInfo.GetDevicePath()`
Info	Returns the internal name of windows path to the device. For informational purposes only.
Returns	Requested information.
Errors	-

C++:	`const char *ceInfo::GetDriverInfo()`
NET:	`String ceInfo.GetDriverInfo()`
Info	Returns the description and version of the used driver. For informational purposes only.
Returns	Requested information.
Errors	-

C++:	`const char *ceInfo::GetFirmwareInfo()`
NET:	`String ceInfo.GetFirmwareInfo()`
Info	Returns the description and version of the used firmware. For informational purposes only.
Returns	Requested information.
Errors	-

C++:	`const char *ceInfo::GetHostController()`
NET:	`String ceInfo.GetHostController()`
Info	Returns the description of the host controller this device is connected to. For informational purposes only.
Returns	Requested information.
Errors	-

C++:	`uint ceInfo::GetPipeBufferSize()`
NET:	`UInt32 ceInfo.GetPipeBufferSize()`
Info	Returns the buffer size of each pipe inside the driver. This is the maximum count of bytes usable by block transfers via ReadBulk() / WriteBulk().
Returns	Requested information.
Errors	-

C++:	`uint ceInfo::GetPipeCount()`
NET:	`UInt32 ceInfo.GetPipeCount()`
Info	Count of pipes supported by the current host-device interface. For informational purposes only.
Returns	Requested information.
Errors	-

C++:	`const char *ceInfo::GetUSBPath()`
NET:	`String ceInfo.GetUSBPath()`
Info	Returns the connection path from device to host controller, including any hub in between. Used ports are enclosed in squared brackets in back of any hub.
Returns	Requested information.
Errors	-

C++:	`bool ceInfo::GetUSBPath()`
NET:	`Boolean ceInfo.GetUSBPath()`
Info	Returns true if the transfer between host and device is in highspeed mode (480MBit/s), false otherwise (15MBit/s).
Returns	Requested information.
Errors	-

class ceFPGA

This class is able to import and export different formats of FPGA designs. This time, rawbit (.RBT) and binary streams (.FPGA, cesys internally used format) are supported. Except ceDevice and ceInfo this class has an public constructor and destructor, so you have to take care about the lifetime of this object.

C++:	`ceFPGA::ceFPGA()`
NET:	`ceFPGA.ceFPGA()`
Info	Class constructor.
Returns	-
Errors	-

C++:	`ceFPGA::~ceFPGA()`
NET:	-
Info	Class destructor.
Returns	-
Errors	-

C++:	`HRESULT ceFPGA::LoadBin(const char *pszFileName)`
NET:	`Int32 ceFPGA.LoadBin(String sFileName)`
Info	Load design from pszFileName / sFileName using bin format importer (created via SaveBin()).
Returns	Error Code.
Errors	`S_OK:` no error `E_OPEN:` can't open file `E_OUTOFMEMORY:` not enough memory available

C++:	`HRESULT ceFPGA::LoadRBT(const char *pszFileName)`
NET:	`Int32 ceFPGA.LoadRBT(String sFileName)`
Info	Load design from pszFileName / sFileName using RBT format importer.
Returns	Error Code.
Errors	`S_OK:` no error `E_OPEN:` can't open file `E_FAIL:` unknown format `E_OUTOFMEMORY:` not enough memory available

C++:	`HRESULT ceFPGA::SaveBin(const char *pszFileName)`
NET:	`Int32 ceFPGA.SaveBin(String sFileName)`
Info	Save design in bin format (smaller and faster loading via LoadBin()).
Returns	Error Code.

Errors	`S_OK: no error`
	`E_OPEN: can't open file`
	`E_FAIL: no design to save (call one of the Load*() methods first)`

C++:	`HRESULT ceFPGA::SetBin(uchar *pucData, uint uiSize)`
NET:	`Int32 ceFPGA.SetBin(Byte[] pucData, uint uiSize)`
Info	Set design based on the binary equivalent given by pucData with size uiSize.
Returns	Error Code.
Errors	`S_OK: no error` `E_OUTOFMEMORY: not enough memory available`

class ceAsyncHandle

This class is a helper class for async operations. It holds all necessary informations about an active transfer in background and is needed for completion. The methods of this class are designed to help to detect transfer finishing.

C++:	`ceAsyncHandle::ceAsyncHandle()`
NET:	`ceAsyncHandle.ceAsyncHandle()`
Info	Class constructor.
Returns	-
Errors	-

C++:	`ceAsyncHandle::~ceAsyncHandle()`
NET:	-
Info	Class destructor.
Returns	-
Errors	-

C++:	`HRESULT ceAsyncHandle::IsComplete(bool *pbComplete)`
NET:	`Int32 ceAsyncHandle.IsComplete(ref Boolean bComplete)`
Info	Check if the attached operation is completed. pbComplete / bComplete will be true if this is done.
Returns	Error Code.
Errors	`S_OK:` no error `E_FAIL:` general error `E_INVALIDARG:` pbComplete is NULL

C++:	`HRESULT ceAsyncHandle::Wait(uint uiTimeOutMs)`
NET:	`Int32 ceAsyncHandle.Wait(uint uiTimeOutMs)`
Info	Wait uiTimeOutMs milliseconds for transfer completion.
Returns	Error Code.
Errors	`S_OK:` no error `E_FAIL:` general error `E_TIMEOUT:` operation has timed out