Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

THIS SPEC IS OBSOLETE

Spec No: 002-04832

Spec Title: AN204832 - F2MC-16FX Family MB96330 Series 16-Bit Microcontroller USB Function Library

Replaced by: None

AN204832

F²MC-16FX Family MB96330 Series 16-Bit Microcontroller USB Function Library

Associated Part Family: MB96330 Series

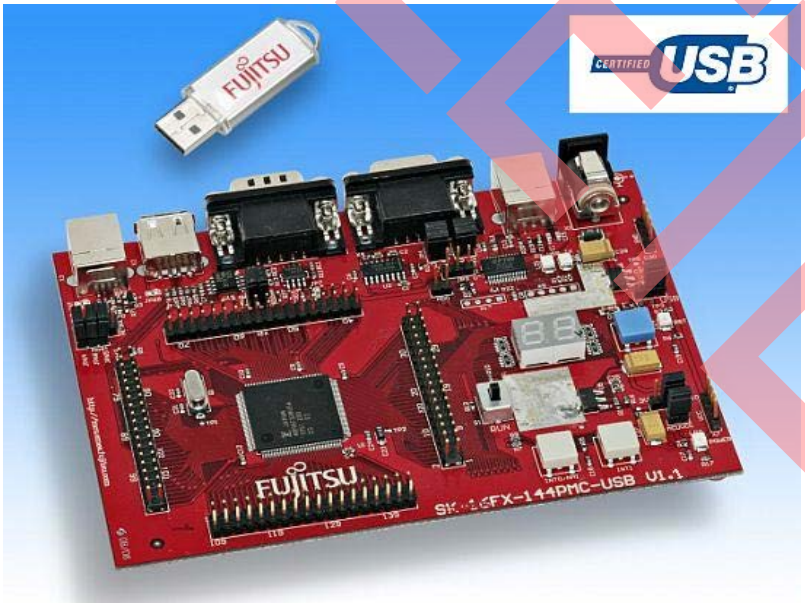
The Windows Cypress Library uses the untouched LibUsbDotNet API as compiled binary and device drivers to provide an own USB API. LibUsbDotNet is provided as GPL and LGPL license.

Contents

1 Introduction

1.1 OpenSource Information

The Windows Cypress Library uses the untouched LibUsbDotNet API as compiled binary and device drivers to provide an own USB API. LibUsbDotNet is provided as GPL and LGPL license and can be downloaded at http://libusbdotnet.sourceforge.net.



1.2 Motivation

Most communications between microcontroller and computer can be realized by simple UART connections like RS232. Today most computers don't have a serial communication interface anymore. USB in embedded systems gets more and more famous since the standard external communication on the computer side is mostly USB. For a simple RS232 connection with USB, a converter chip is needed. This converter having the need of a second chip in the design and more costs only for a simple data connection. For microcontrollers with embedded USB function on board, there is no need of a second chip. In a few cases USB standard classes like mouse, keyboard, mass storage, virtual com port, etc. can be used to communicate between microcontroller and computer. In all other cases, the developer has to provide own drivers and software for the communication between the embedded system application and the computer.

This application note (including software examples) has the challenge to establish a connection between microcontroller and computer without using commercial software. On the microcontroller side own drivers have to be implemented. On the computer side, LibUSB has to be used to build the bridge between USB function and user application.

1.3 Example usage of the .NET library

The component connecting LibUSB with the .NET framework is called LibUsbDotNet and stands under GPL. The library can be downloaded at http://libusbdotnet.sourgeforge.net.

For implementing stable USB functionality with Cypress MCUs, Cypress programmed a USB function library using the LibUsbDotNet library. This .NET framework library can be used in different programs supporting the .NET framework.

1.3.1 Microsoft Visual Studio

The original .NET framework libraries are programmed in Visual C#. The compiled component libraries can be used with every program language of the Microsoft Visual Studio that supports the .NET framework 3.0.

1.3.2 Borland Delphi

At the moment, the component was not tested with Borland Delphi, but should also work with a .NET framework Delphi application supporting .NET framework 3.0.

1.3.3 LabVIEW with Cypress 16FX MCU

LabVIEW offers a graphical lab which can connect different components. As data input or output, measuring cards can be used. With the .NET component it is possible to access the USB data streams directly to implement a simple USB oscilloscope for example. Data can be read from a ADC port and will be displayed in a Waveform Graph.

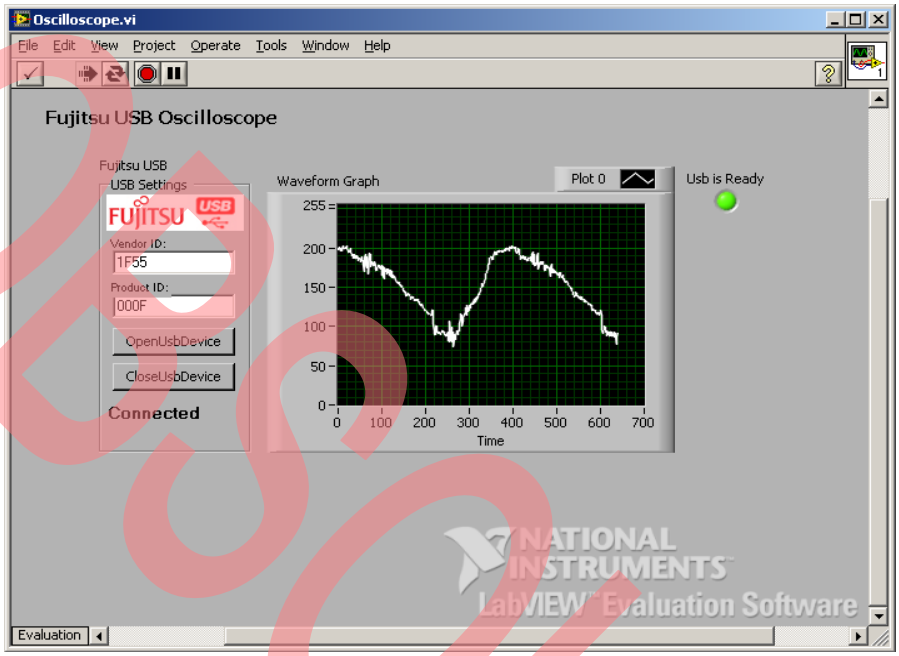
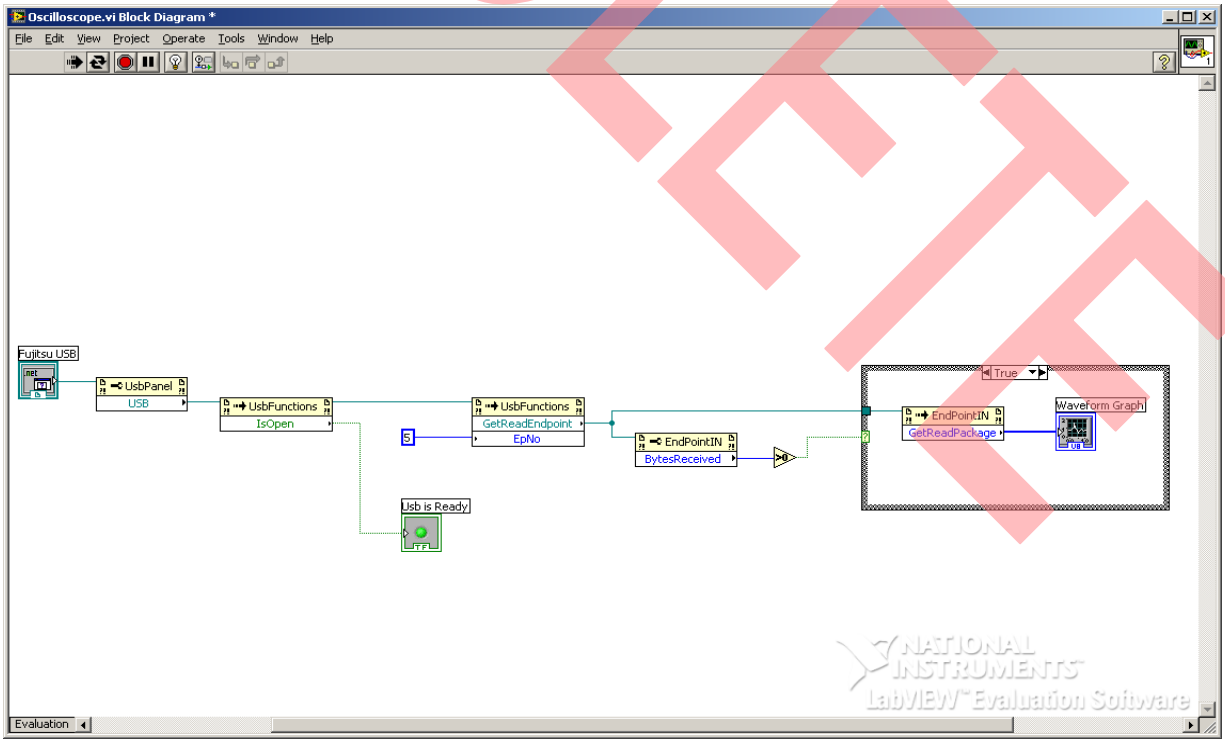


Figure 1-1. Use of LabVIEW to implement a simple USB oscilloscope



2 Quick Start

QUICK START TUTORIAL

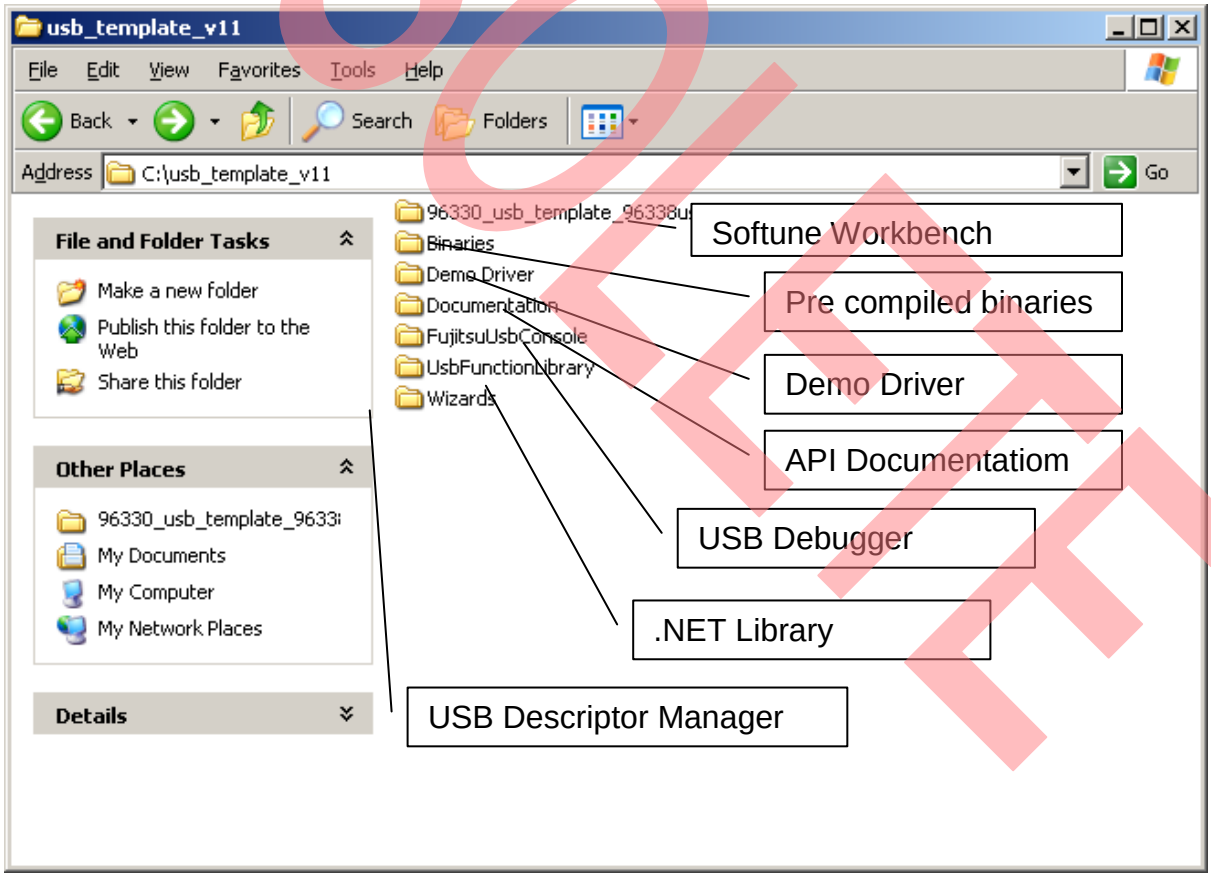
For getting started with USB, following software tools has to been installed on a Microsoft Windows environment:

- Visual Studio 2008 SP1 (C#)
- .NET Framework 3.5 SP1
- Cypress Softune Workbench
- Cypress Flash Programmer
- USB Template with USB Descriptor Manager

The USB Template includes all necessary tools and libraries for developing USB applications. On the microcontroller side, it provides a template project for Softune Workbench with the UsbFunction library included and all necessary settings in *start.asm*, *vectors.c* and *main.c*. For creating the *UsbDescriptor.h* file and template files for *UsbClass.c* and *UsbClass.h* files, the USB Descriptor Manager is also included in this package.

For .NET environment developing tools like Visual Studio or Delphi, two libraries are included to provide the needed functionalities for USB communications (LibUsbDotNet & UsbFunctions).

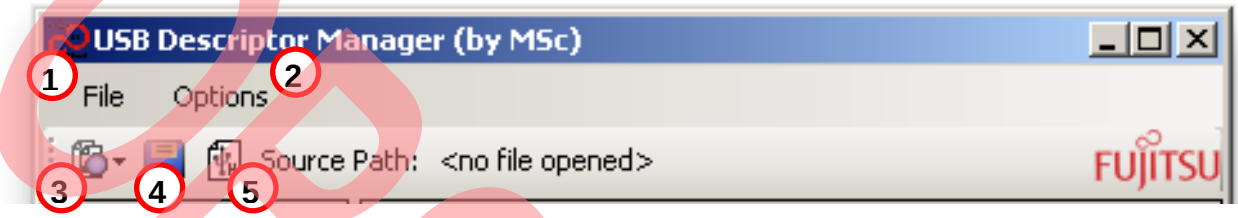
Figure 2-1. Template Content



2.1 Creating the Description Header File

Before the Softune Workbench Template can be used, the Descriptor Manager has to be opened. The Descriptor Manager writes the *UsbDesctiptors.h* file in which all descriptors are set. The Descriptor Manager can also write the files *UsbClass.c* and *UsbClass.h*. These files are templates for the usb class event handlers, which can receive or send data, recognize connect, disconnect and configured events and handle USB class requests.

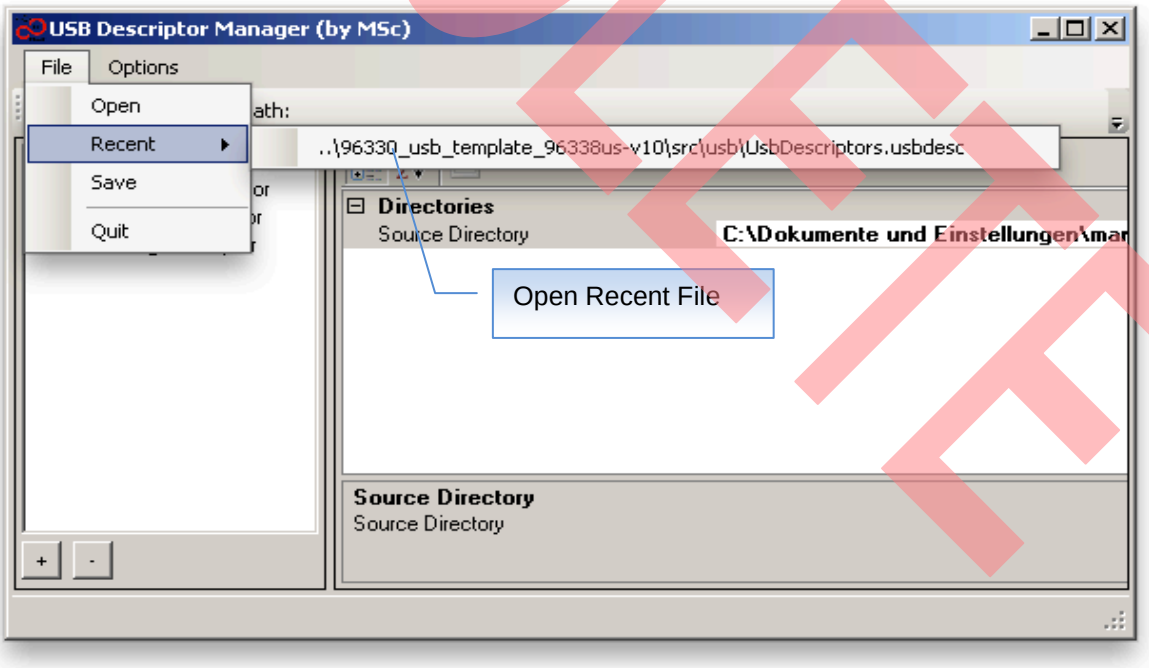
Figure 2-2. USB Descriptors Manager



1: Open or Save files / Open Recent files
2: Export DeviceDescriptors.h file
3: Show or export DeviceDescriptors.h, UsbClass.h or UsbClass.c
4: Save configuration file
5: Export DeviceDescriptors.h file

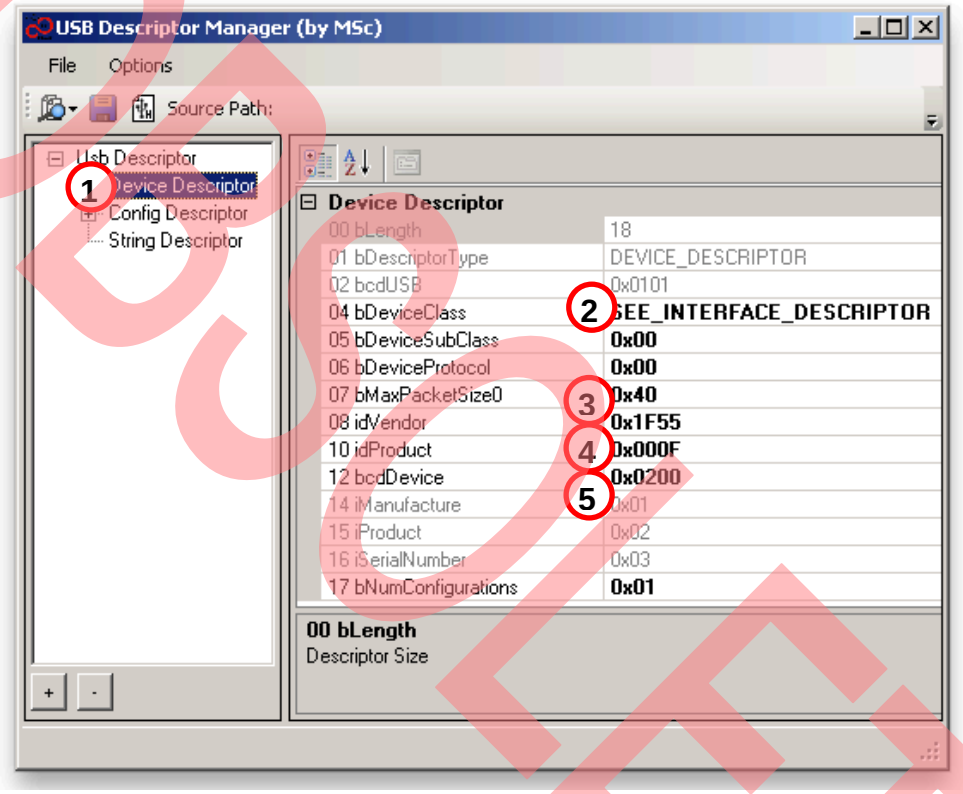
The Descriptor Manager can be found in the *Wizards* folder. Open *Recent File* to load the correct directories to the sources. If no recent file was found or the path can't be used, it can be loaded manually. The configuration file can be found in the project directory: *src\usb\UsbDescriptors.usbdesc*

Figure 2-3. Open the configuration directory



Now the correct paths are set and the user can start to configure his device. First, the device descriptor should be configured. Clicking on the “Device Descriptor” (1) in the left split view can do this. The following view shows an example how to configure a custom class device. The Device Class is set to “SEE_INTERFACE_DESCRIPTOR” (2). Normally the endpoint 0 FIFO buffer is 64 bytes (`h40`) (3). This can be changed to a maximum of 256 bytes in the bMaxPacketSize0. As default the Vendor ID (4) of Cypress is set (`0x1F55`). The Product ID (5) represents a Cypress USB template demo device (`0x000F`).

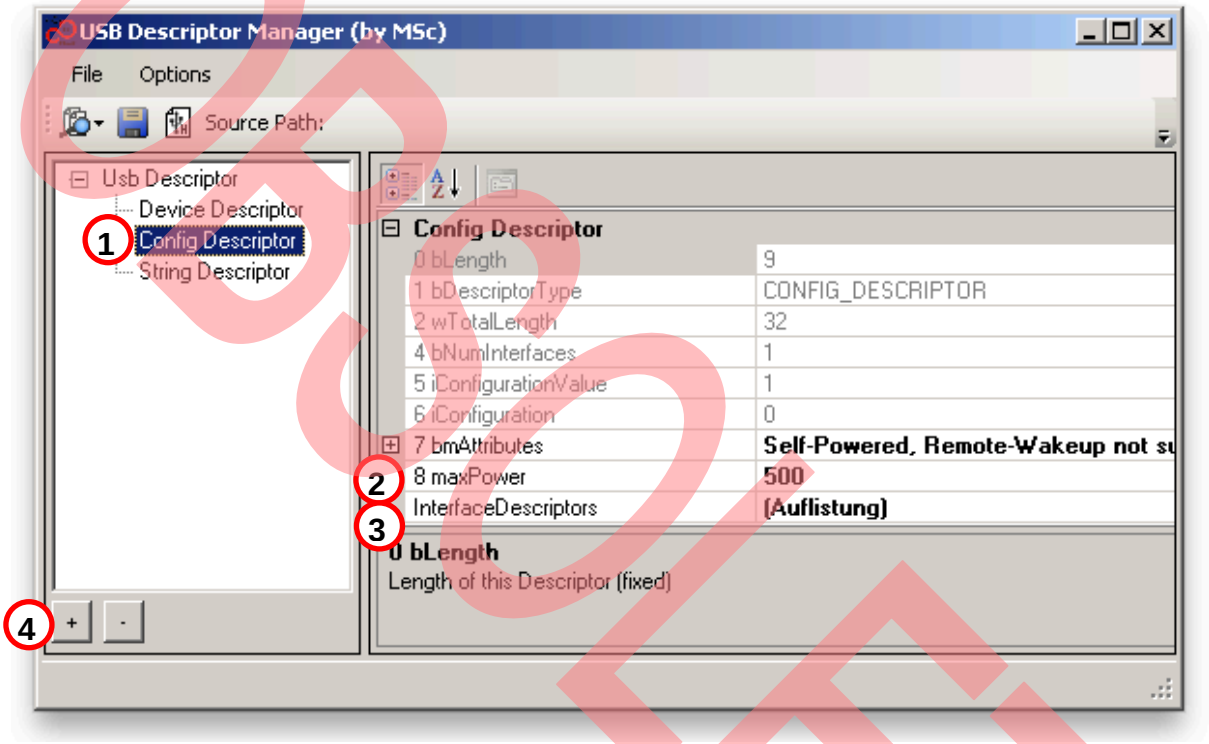
Figure 2-4. Set up the device descriptor



Now the configuration descriptor has to be set. The configuration descriptor is displayed after clicking at the *Config Descriptor* in the left split view (1). With *bmAttributes*, the Self-Powered and Remote-Wakeup features can be defined. To set them, the Attributes can be opened by clicking on the expand symbol [+] before *7 bmAtrributes* (2). The *maxPower* (3) Attribute supports a maximum power of 500 mA and is only used if the device is bus powered.

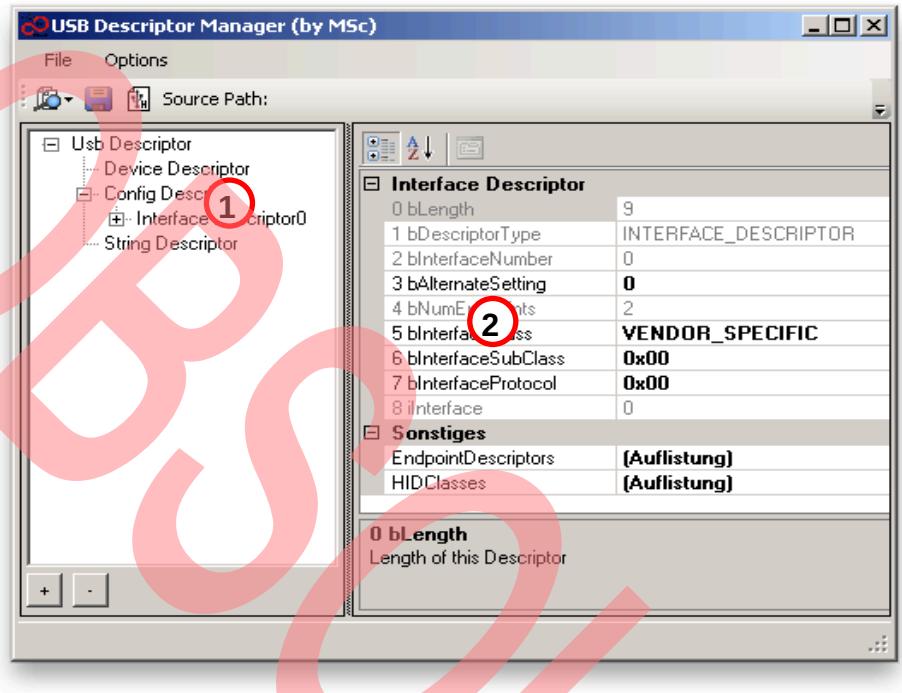
After setting up all features, the interface descriptor has to be added. Be sure, the *Config Descriptor* (1) is selected. In this *Config Descriptor* an *Interface Descriptor* can be added by clicking on the [+] symbol on the left bottom side (4).

Figure 2-5. Configuration Descriptor



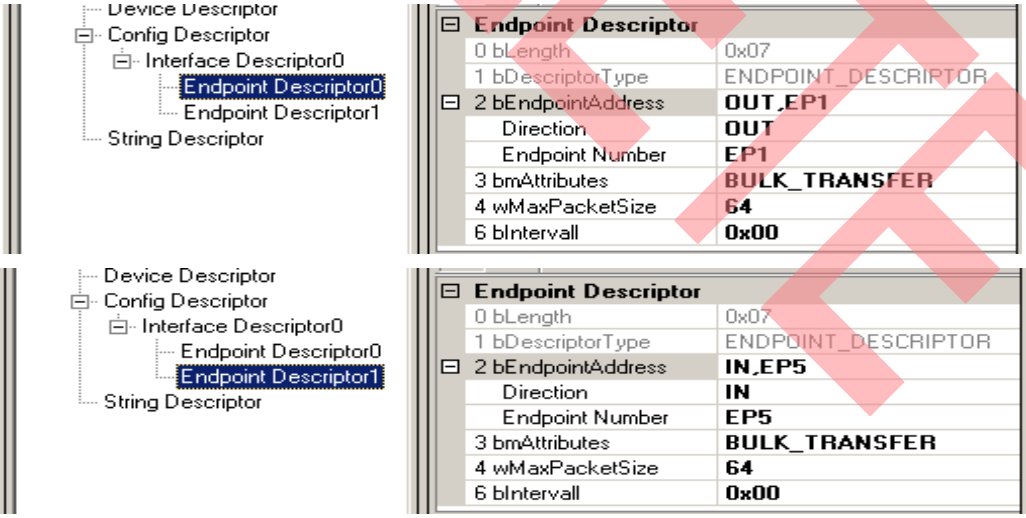
The following figure is an example for an Interface Class configured as USB Custom Class. First the Interface Descriptor is selected (1). In the Interface Descriptor in this example the vendor specific interface class (2) is used.

Figure 2-6. Interface Descriptor



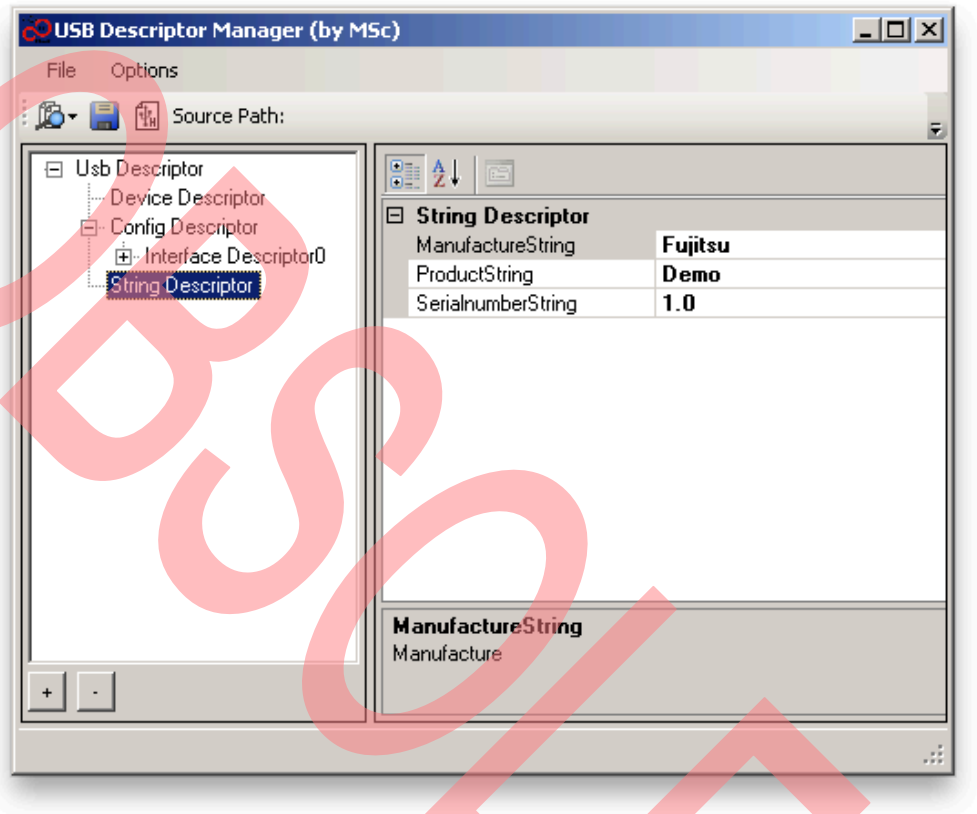
At last the endpoint pipes have to be defined. This can be done by clicking on the left botton [+] while the Interface Descriptor is selected. For each data direction (USB Host), one pipe has to be defined. The direction options IN and OUT are from Computer (HOST) view. In this example EP1 and EP5 with bulk transfer are used.

Figure 2-7. Endpoint Descriptors



For information about the device manufacturer, product string and serial number, the String Descriptor is used. Normally the string descriptor should be as short as possible!

Figure 2-8. String Descriptor



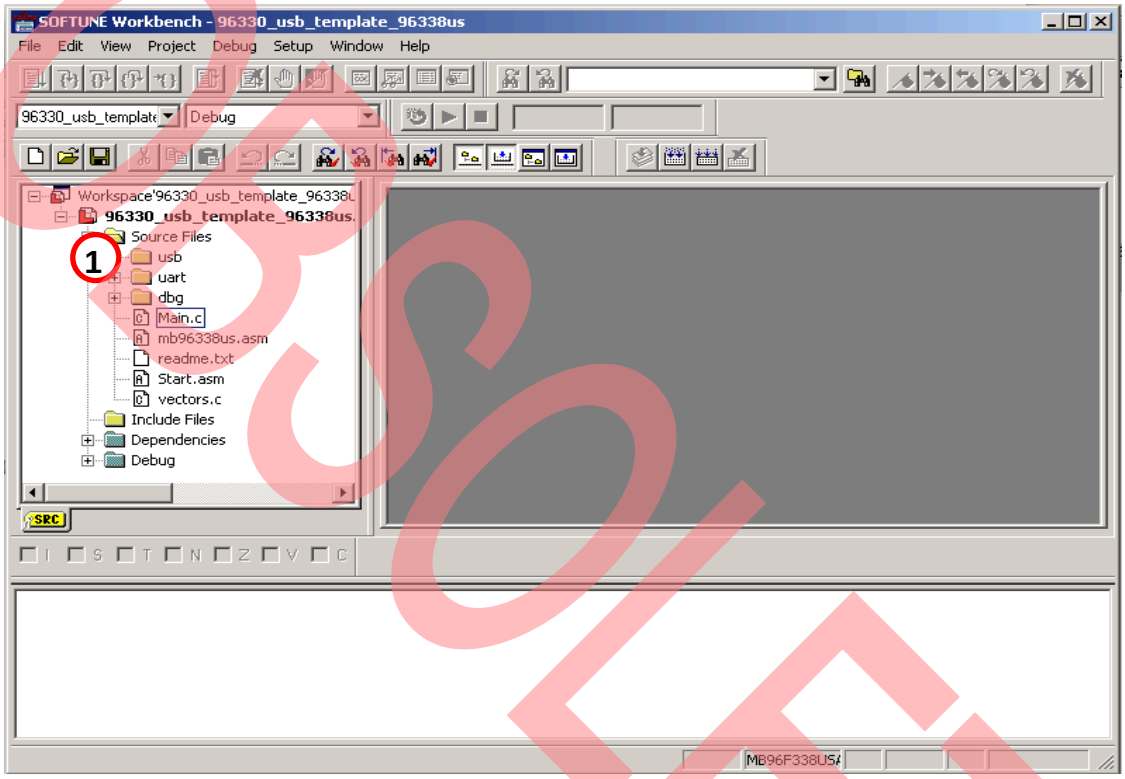
Now all configurations for the USB descriptors are done. As next step the USB template can be created. The *UsbDescriptors.h* file is created by simply clicking the symbol. It will be automatically created or overwritten in *src\usb\UsbDescriptors.h*. Other files can be created by clicking the symbol. First a window will be displayed with the code template. These files can be saved by clicking the symbol in the codewindow. All USB specific files should be saved in the Softune Workbench Project in the directory src\usb.

Note: While saving any template, the original file will be replaced by the new template! This makes sense while using the *UsbDescriptors.h* file. The *UsbClass.h* and *UsbClass.c* are initial templates! Do not overwrite them without knowing what you are doing! Overwrite them only, when starting a new project with this template.

2.2 Using Templates in Softune Workbench

If all files were created with the USB Descriptor Manager, the files *UsbDescriptors.h*, *UsbClass.h* and *UsbClass.c* can be used in the Softune Workbench USB template. Open the Workspace file *96330_usb_template_96338us.wsp*. The files are located in the *USB* directory (1). The simplest way to show a USB communication is a loop interface. Data, which was received by the USB function, will be sent back to the computer (Host).

Figure 2-9. Softune Workbench Project



If all configuration was done described in chapter 2.1; endpoint 1 receives data from host and endpoint 5 can send data to the host. A look in the file *UsbClass.c* helps understanding how the data transfer is working: For receiving data, an event handler handles the reception part. A second procedure helps to handle the received data from the application layer. The sending part is handled by a simply routine, which only echoes the data to the direct sending routine of the UsbFunction library. (See also Chapter 4.3)

Figure 2-10. UsbClass template API

```
void UsbClass_Init(void);  // initiates the UsbClass
uint8_t UsbClass_DataReceiveEventEndpoint1(usb_event_t * stcEvent); // receive handler
uint32_t UsbClass_GetReceivedDataEndpoint1(uint8_t** pu8Buffer); // read data from
application
uint8_t UsbClass_SendDataVia5(uint8_t* u8Buffer, uint32_t u32DataSize, uint8_t
u8TransferMode)
uint8_t UsbClass_ClassEventHandler(usb_event_t * stcEvent); // requests & status event
handler
```

Before the UsbFunction library is initialized, the UsbClass has to be initiated. This is strongly recommended, because the UsbClass initializes all necessary event handlers. This has to be done before the UsbFunction library can call any event!

Figure 2-11. USB initialisations

```
UsbClass_Init();                    // USB Class initialization
UsbFunction_Initialize(TRUE);       // UsbFunction library initialization
```

After all initializations, the loop functionality has to be added in the main loop. Following variables are required: A size variable and a buffer variable. Both have to be added to the main function.

Figure 2-12: Variables for loop device

```
uint32_t u32Size;
uint8_t* pu8Buffer;
```

The loop code checks if new data is arrived and sends this data back.

Figure 2-13. Code example for loop device

```
for(;;)
  {
      u32Size = UsbClass_GetReceivedDataEndpoint1(&pu8Buffer); // Get received data
      if (u32Size > 0) // if it is higher than 0, data was received
      {
          UsbClass_SendDataVia5(pu8Buffer,u32Size,USB_SENDING_MODE_INTERRUPT); // data
back
      }
  }
```

The whole main loop code should look like this:

Figure 2-14. Example main code

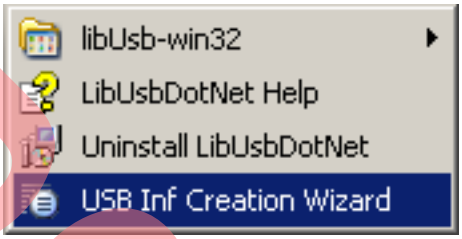
```
void main(void)
{
  uint32_t u32Size;
  uint8_t* pu8Buffer;
  InitIrqLevels();
  __set_il(7);   // allow all levels
  __EI();        // globally enable interrupts
  //InitUART1(); // only for debugging
  UsbClass_Init();
  UsbFunction_Initialize(TRUE);

  for(;;)
  {
      u32Size = UsbClass_GetReceivedDataEndpoint1(&pu8Buffer); // Get received data
      if (u32Size > 0)  // if it is higher than 0, data was received
      {
          UsbClass_SendDataVia5(pu8Buffer,u32Size,USB_SENDING_MODE_INTERRUPT); // data
back
      }
  }
}
```

Now the code can be compiled and flashed into the microcontroller.

2.3 Install Drivers

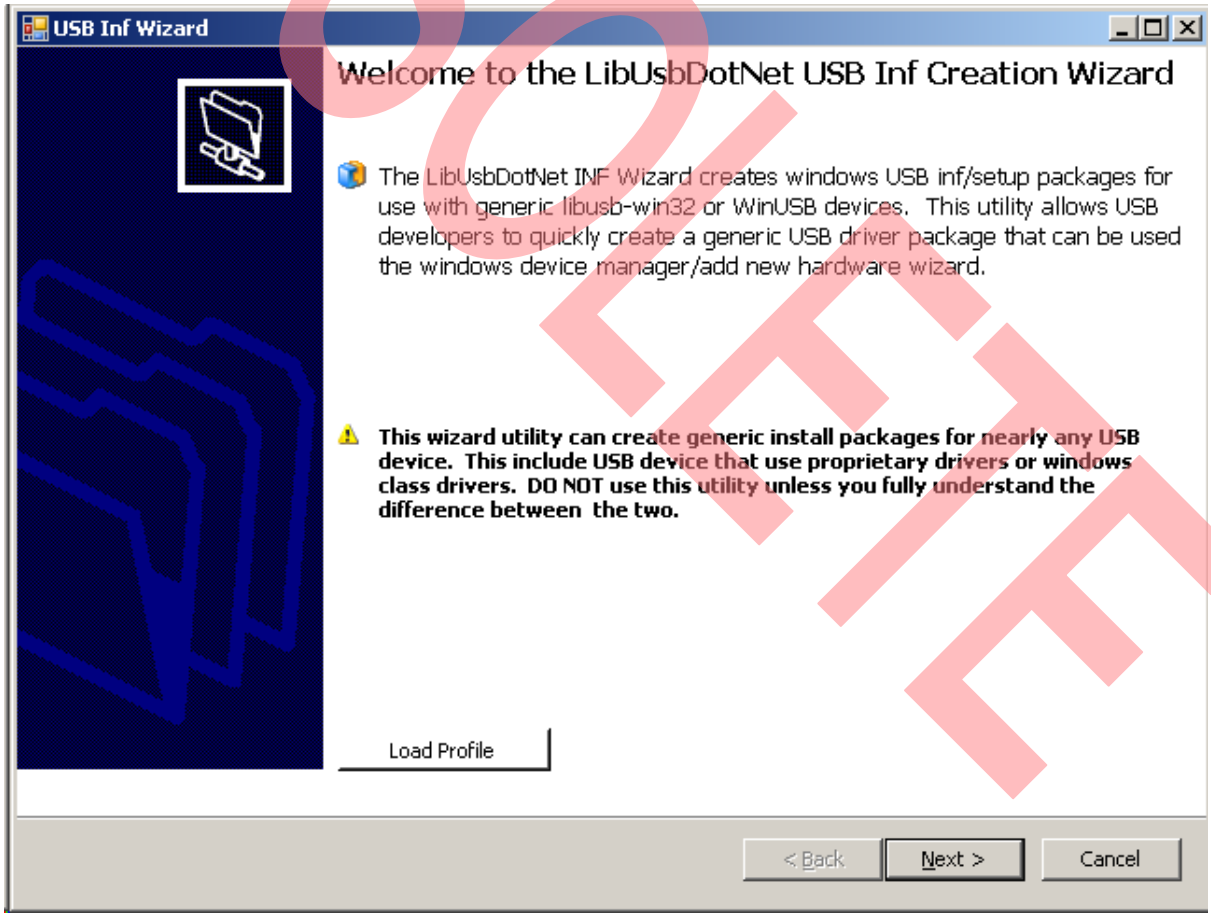
To create custom drivers for own devices, the project LibUsbDotNet has to be installed. The newest version can be downloaded from http://libusbdotnet.sourceforge.net. Version 2.2.0 is added in the Libraries .NET directory. Run *LibUsbDotNet_Setup.2.2.0.exe* to start installing the library. After LibUsbDotNet was installed, the USB Inf Creation Wizard can be opened. The Inf Creation Wizard is part of LibUsbDotNet and can be found in the start menu programs in the rubric LibUsbDotNet.

Figure 2-15. Location of the USB Inf Creation Wizard



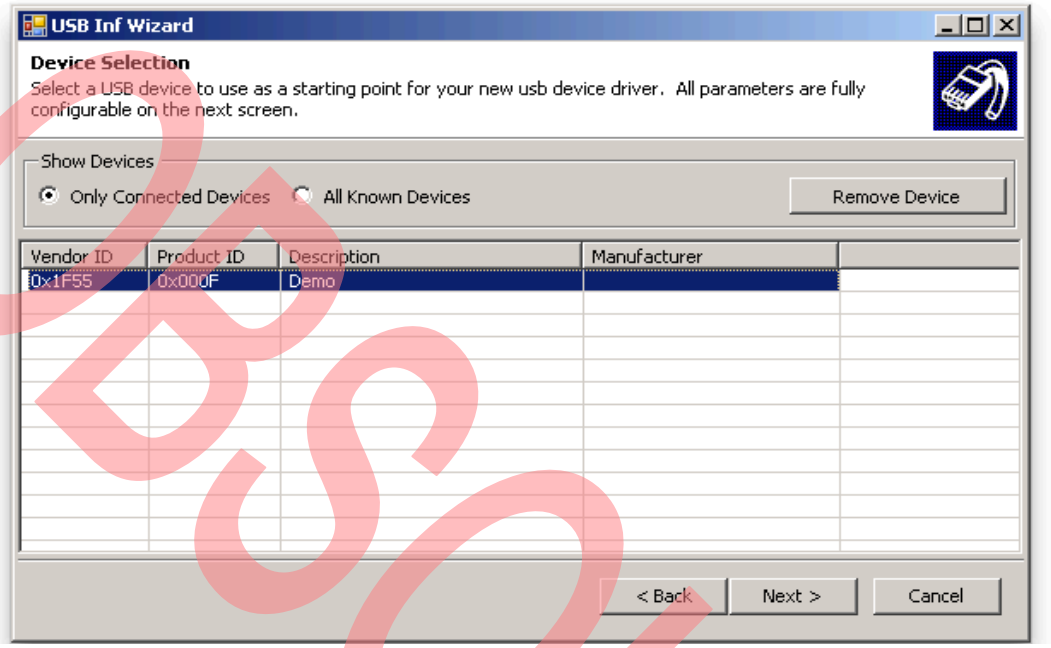
The USB Inf Wizard is build like an installation wizard. The wizard can only be used if the device is connected for which the driver will be created. If the device is connected the next step can be started.

Figure 2-16: LibUsbDotNet: USB Inf Wizard



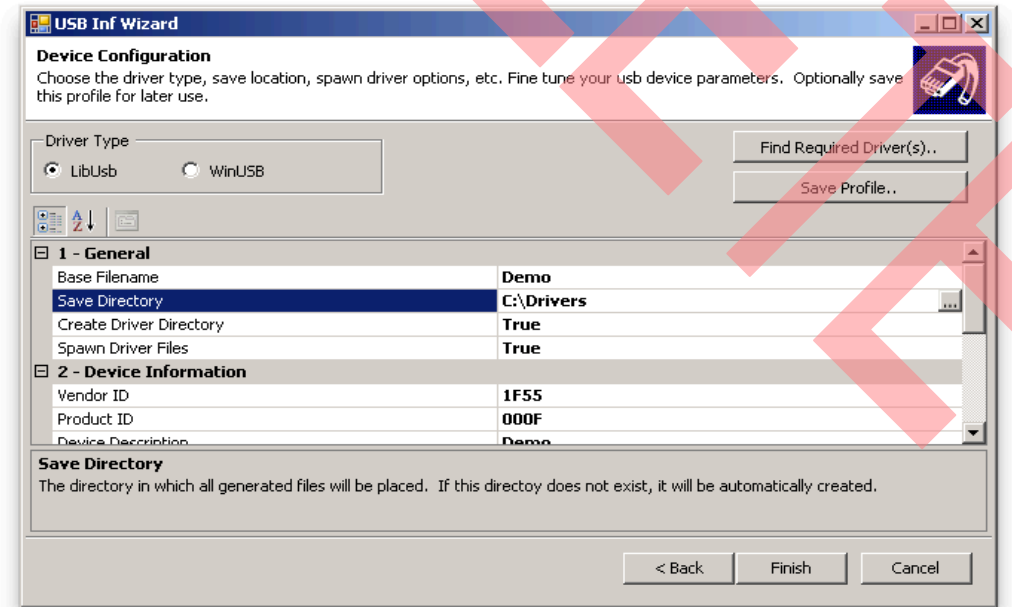
Now the device sould be displayed in the list of connected devices. By selecting it and click on *next*, the driver options can be entered.

Figure 2-17. Choosing Device



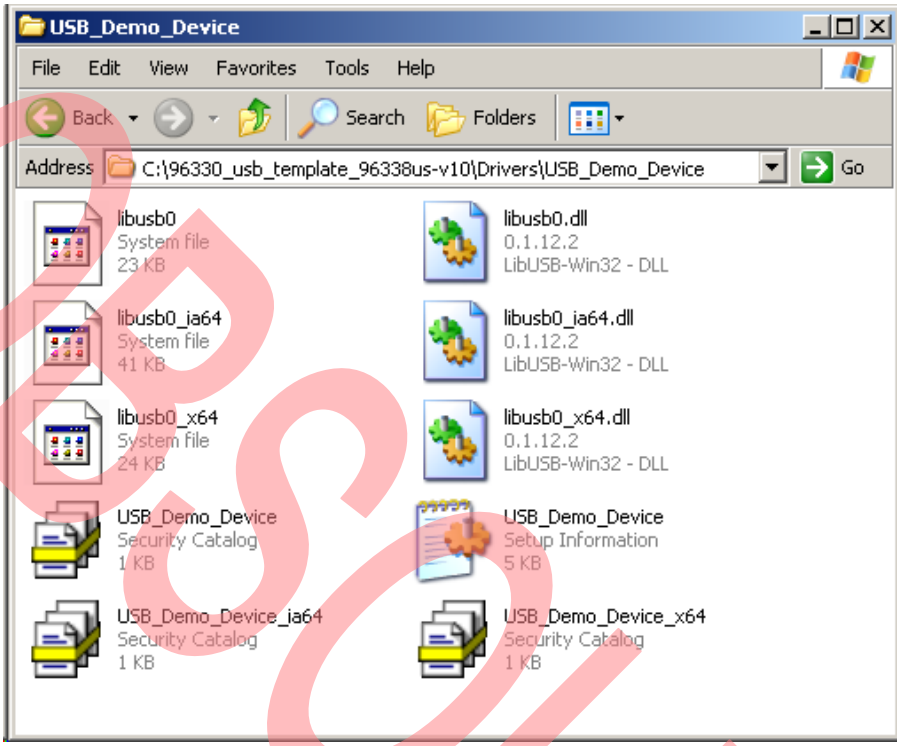
For example create a directory “driver” in the C: root directory. Choose this directory in the Save Directory property with the “...” symbol at the end of the text field. After configuring all values click on *finish* and a new driver directory will be created in *C:\drivers*. In some cases, first *“Find Required Drivers(s)..”* must be clicked and the directory of LibUsbDotNet must be entered (normally C:\Program Files\LibUsbDotNet\).

Figure 2-18. Driver Options



The Drivers can now be found under *C:\Drivers\Demo*. A driver directory can look like in the following screen shot:

Figure 2-19. Created driver directory



Finally just reconnect the USB device and specify the driver directory to install the device in the Windows system.

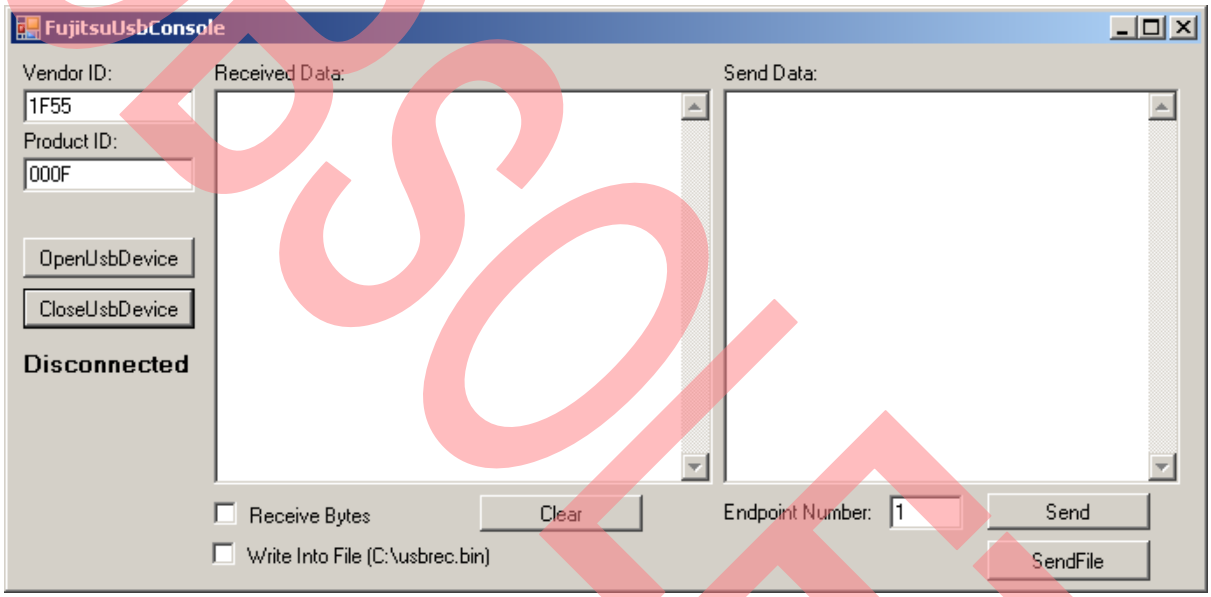
2.4 Using C# as PC frontend

A free version of Visual Studio Express 2008 sould be downloaded from the Microsofts Website to be able to open the existing C# examples. The Cypress Usb Console is a program supporting connect, disconnect, hot plug, sending and receiving data.

Following files have to be added to own projects: *LibUsbDotNet.dll* and *UsbFunctionLibrary.dll*. For simple USB functionality only the *UsbFunctionLibrary.dll* has to been added to the link section, but both files has to be added to the project directory!

To test different USB data transfers, the program Cypress*UsbConsole* can be used. All data received from any endpoint will be displayed in the “Received Data” text field. To send data via an endpoint, the endpoint can be entered in the field “Endpoint Number”. The data to be sent can be entered in the text field “Send Data”. Drivers for the Cypress Demo device can be found in the template, too.

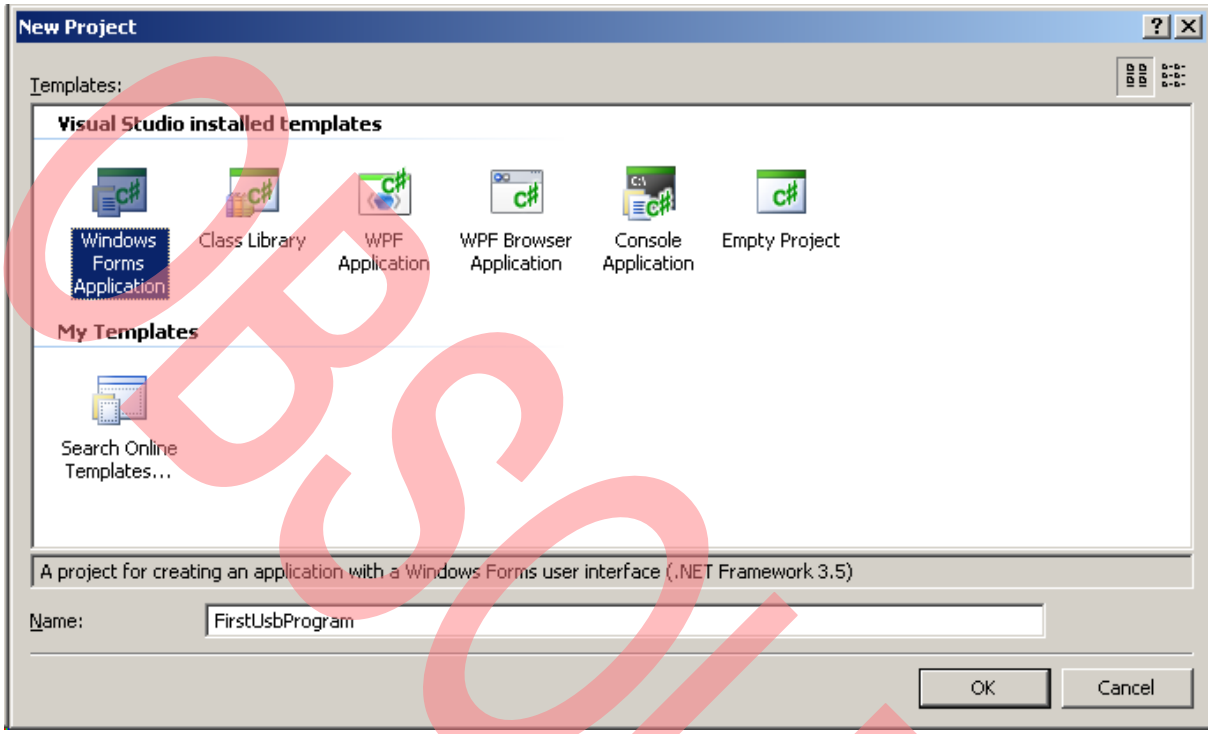
Figure 2-20. USB Test Application C#



2.4.1 First C# USB program

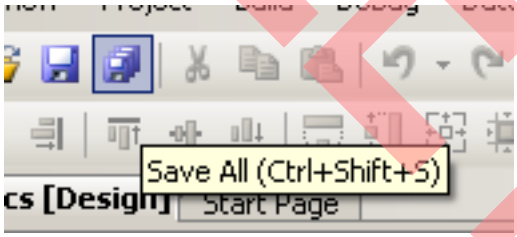
The first C# USB application can be a „Windows Form Application“. Here named „FirstUsbProgram“.

Figure 2-21. Create new Project



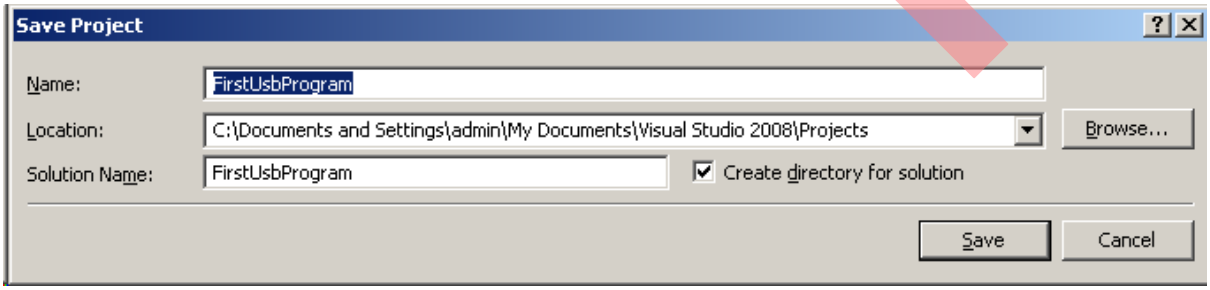
After the project was created, it has to be saved to create all neccesary files and directorys. This can be done by clicking the following button in the toolbar:

Figure 2-22. Save all files



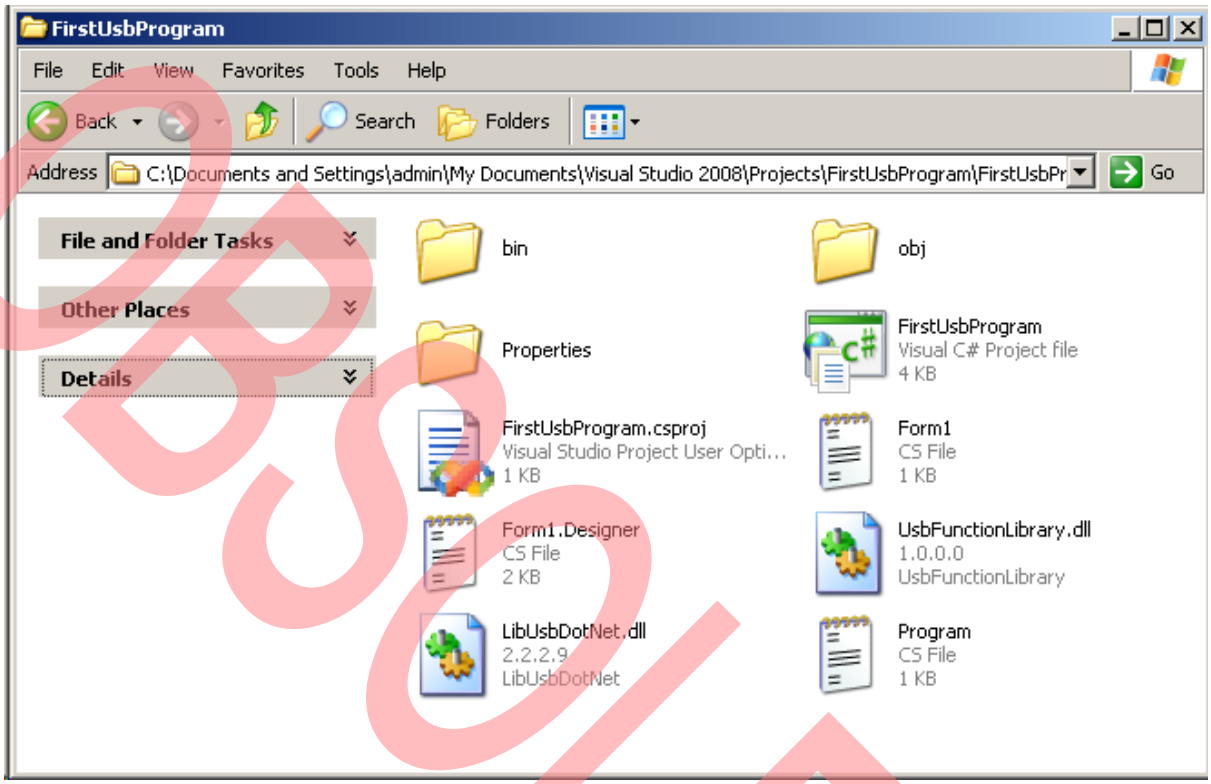
Save the project to a location of your choice.

Figure 2-23. Save project



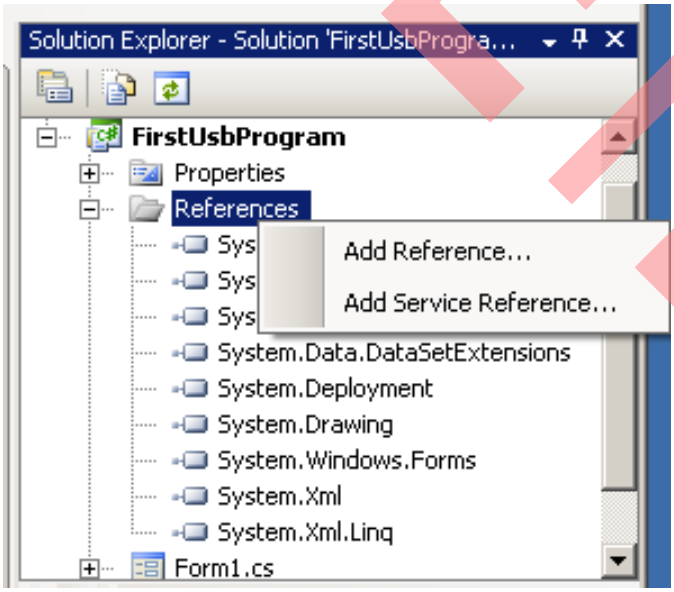
Now all project directories were created. Copy the following files to your project directory: *LibUsbDotNet.dll* and *UsbFunctionLibrary.dll*.

Figure 2-24. Project directory containing the USB libraries



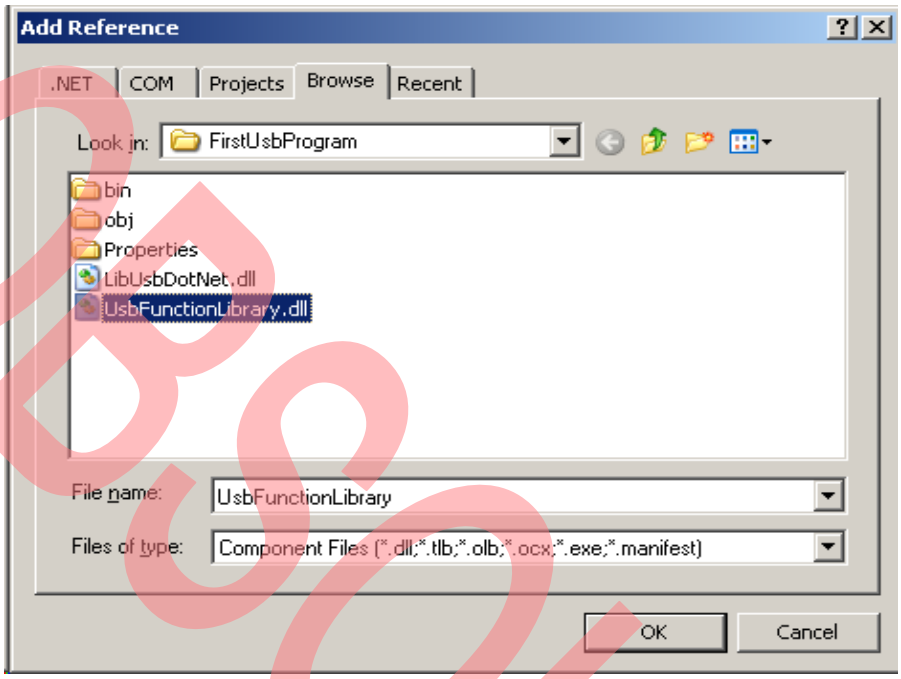
Add the reference for the *UsbFunctionLibrary.dll* to the project. This can be done on the right side with the Solution Explorer. With a rigth-mouse-button click on References, new references can be added.

Figure 2-25: Add a library as reference



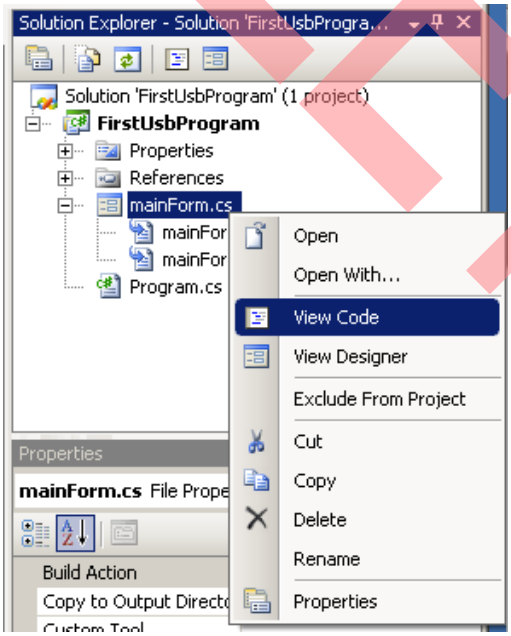
With the tab *Browse* an existing library can be added to the project. Only add the file *UsbFunctionLibrary.dll*. The library *LibUsbDotNet.dll* is needed for the UsbFunctionLibrary, but is not needed as a reference.

Figure 2-26: Add the UsbFunctionLibrary as reference



As next, this reference has to be added to the include section. To add these includes, the code of the project Form must be opened. In the Solution Explorer a right-mouse-button click on the Form opens a context menu. Here the code can be viewed.

Figure 2-27: View code of the project form



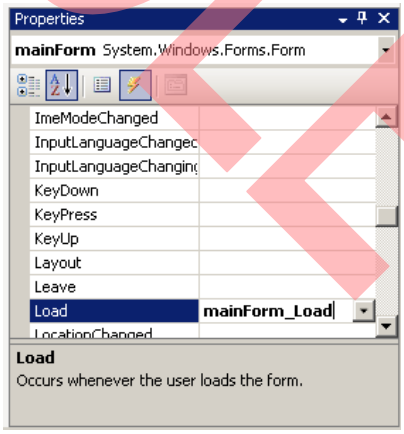
The following code must be added to start using the UsbFunction library. An include has to be done at first. The UsbFunctions object is initialized in the Form class.

Figure 2-28: Code of the project form

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using UsbFunctionLibrary;

namespace FirstUsbProgram
{
    public partial class mainForm : Form
    {
        public UsbFunctions Usb = new UsbFunctions();
        public mainForm()
        {
            InitializeComponent();
        }
    }
}
```

All initializations can be done in the form *Load* event. To switch between properties and event properties, the flash in the toolbar of the Properties section (right bottom) must be clicked. An event method can be easily added by double click the text area behind the event in the *Properties* section.

Figure 2-29: Event properties



The following code can be added in the *mainForm_Load* event handler. For detect a device connection, an additional event *mainForm_DeviceConnected* can be created in the code section.

Figure 2-30: Form load event

```
private void mainForm_Load(object sender, EventArgs e)
{
            short VendorID = 0x1F55;
            short ProductID = 0x000F;
            int ReadBufferSize = 1024;
            USB.OpenUsbDevice(VendorID, ProductID, ReadBufferSize);
            USB.DeviceConnected += mainForm_DeviceConnected;
}
private void mainForm_DeviceConnected(object sender, ConnectEvent e)
{
}
```

Following code could be an example for sending or receiving data:

Figure 2-31: Example Read/Write procedures

```
/// <summary>
/// Send a testmessage via endpoint 1: { 1, 2, 3, 4 }
/// </summary>
private void SendTestMessage()
{
     byte[] Data = { 1, 2, 3, 4 };
     int BytesWritten = 0;
     int Timeout = 1000; //1000ms
     USB.GetWriteEndpoint(1).Write(Data, Timeout, out BytesWritten);
}

/// <summary>
/// Receive data via endpoint 5
/// </summary>
/// <param name="WaitForDataReceived"></param>
/// <returns>
/// returns the received data or
/// a zerro length byte array if no data was received
/// </returns>
private byte[] ReceiveData(bool WaitForDataReceived)
{
     if (WaitForDataReceived)
     {
          while (USB.GetReadEndpoint(5).BytesReceived == 0)
          {
              //wait for Data received
          }
     }
     if (USB.GetReadEndpoint(5).BytesReceived > 0)
     {
          return USB.GetReadEndpoint(5).GetReadPackage();
     }
     return new byte[0];
}
```

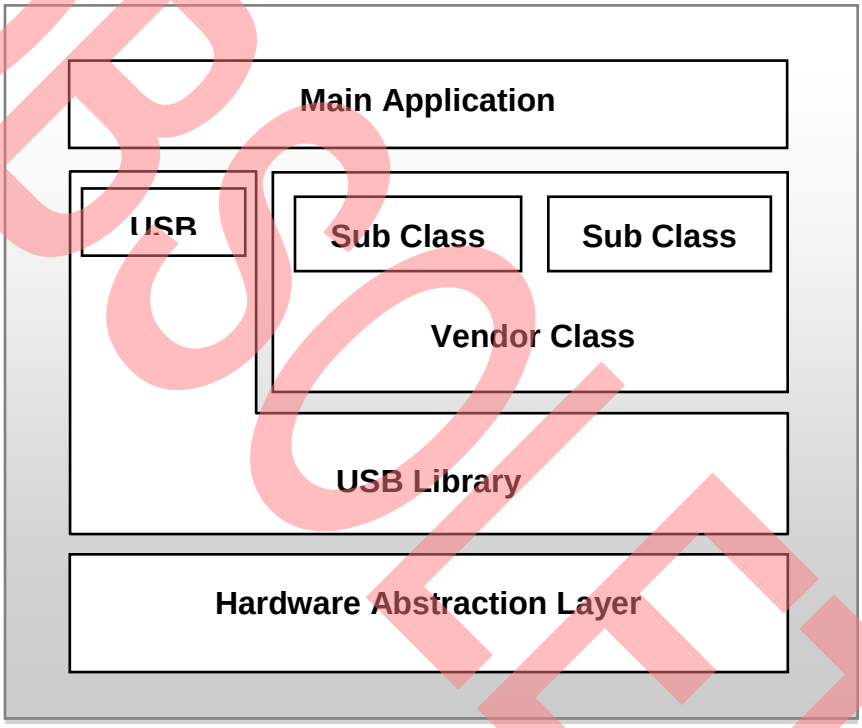
3 USB Device Library Functionality

USB LIBRARY FUNCTIONALITY

3.1 Microcontroller USB Library

For embedded USB programming only three elementary modules are neccessary: The USB Library, the USB Class and the Main Application. The Main Application is executing the user defined program. The USB Classes are the interfaces between the USB Library and the Main Application. Only a simple USB API is the direct interface between Main Application and USB Library, which handles USB initialization and getting the current status of the USB function.

Figure 3-1: Microcontroller USB library – modules overview



Following Modules are provided by the application:

USB Library: The USB function library handles all configuration and data transfers with the hardware abstraction layer. A small USB API is provided to the main application to open or close the USB features. The vendor class handles the main communication.

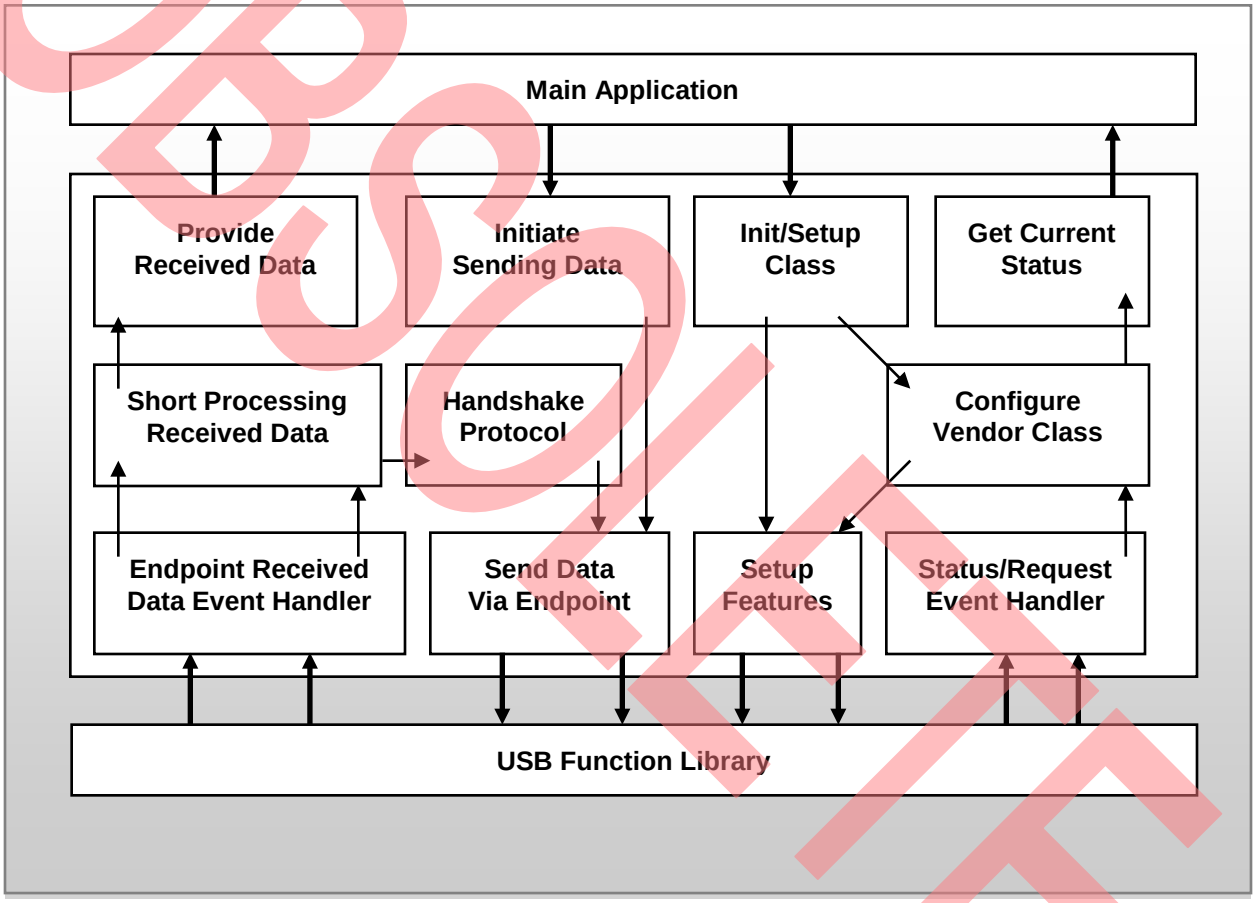
Vendor Class: A vendor class module can be created as a template with the USB Descriptor Manager. All initiated USB transfers or received USB transfers can be accessed from the main application via the vendor class API.

3.1.1 Module Vendor Class / Subclass

The vendor class is a USB application specific module. It has the function to handle all data and control transfers. Control transfers are vendor class specific USB requests, which must be handled normally in this module. For composite USB devices with more than one interface, subclasses can be added. This can be done in a second module or can be handled also in the vendor class. In some reasons subclasses makes sense for combining different standard classes in one device. Most USB devices have only one interface to communicate with. In this case, the Vendor Class and the Sub Class are combined in one module, because the Vendor Class only calls one Subclass.

Figure 3-2 gives an example, which tasks are handled by a vendor class module. The vendor class module builds the interface part between main application and USB function library. Events or Interrupts are received from the USB function library directly.

Figure 3-2: Vendor class module



3.1.2 Module Main Application

The main application only has access to a small USB function library API. This API is only used for initializing the USB function library, connecting and disconnecting a device and getting the USB status. All other features are provided by the vendor class.

The following HID mouse example shall demonstrate why using a USB class to communicate with and not the direct USB library API. First the initialization has to be done. After the initialization the main loop collects all data package as struct and sends it every loop to the data via the USB vendor class. The vendor class has to catch errors, to handle the USB communication and considers if packages can be send. All communication handling is merged in the vendor class module. The sequence (Figure 3-3: Sample interaction sequence) shall demonstrate which part the main application has in the interaction process.

Figure 3-4: Sample mouse application flowchart

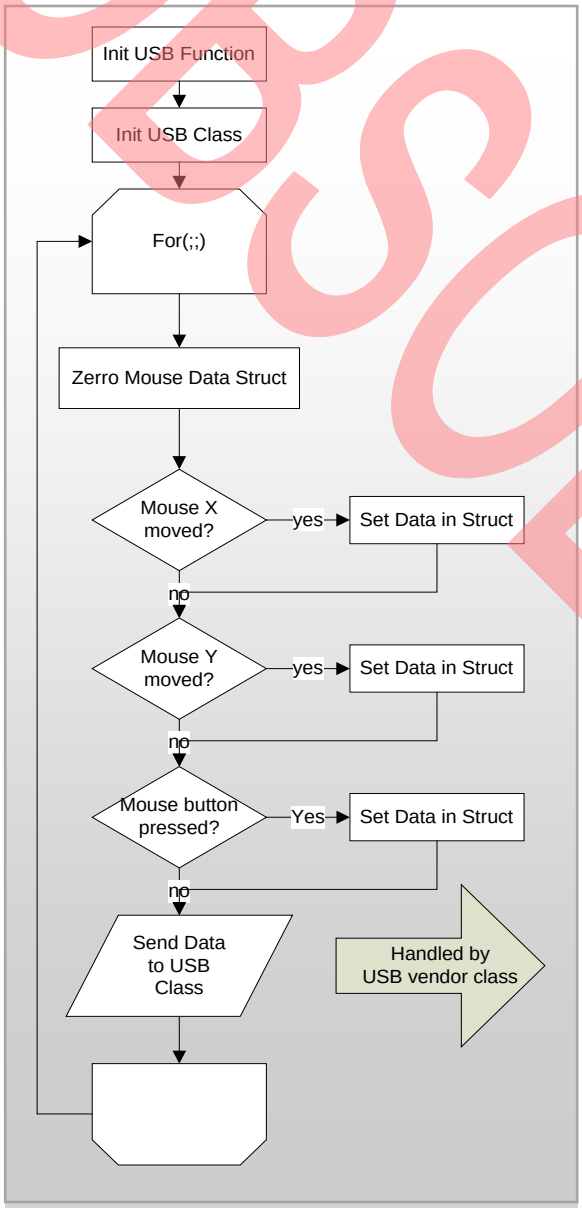
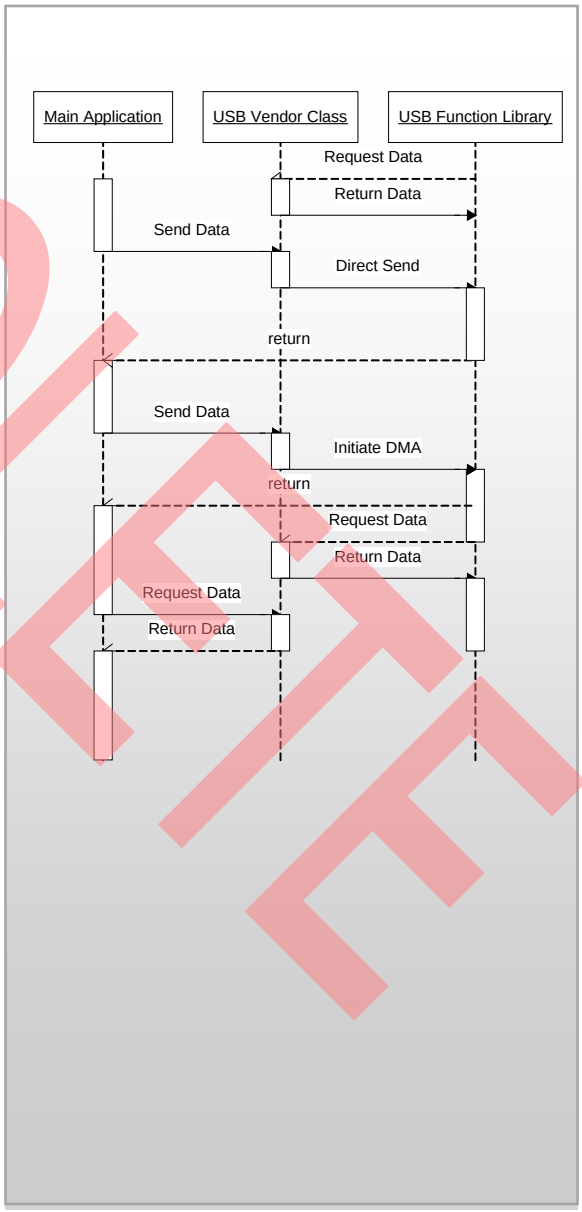


Figure 3-3: Sample interaction sequence



3.1.3 Module USB Function Library

The USB Library handles the initialization and the communication with the hardware abstraction layer. Most of all processes are MCU interrupt controlled. The incoming data via endpoint 0 will be evaluated. The USB Library handles all standard requests that are necessary for the enumeration process. Description headers are defined separate in a description headers include file. For class specific requests the incoming data will be redirected to the vendor class (sub class) module. After a successful enumeration the Endpoints will be initialized. For each Endpoint, which is responsible for a data pipe, the data pipe event handlers will be relocated to the USB vendor class or sub class. To initialize the USB function and getting current connection states, the library provides also a simple API, which can be controlled directly by the Main Application. Figure 3-5 shows a short overview how an enumeration process is handled in the USB function library.

Figure 3-5: Simple view flowchart of a enumeration

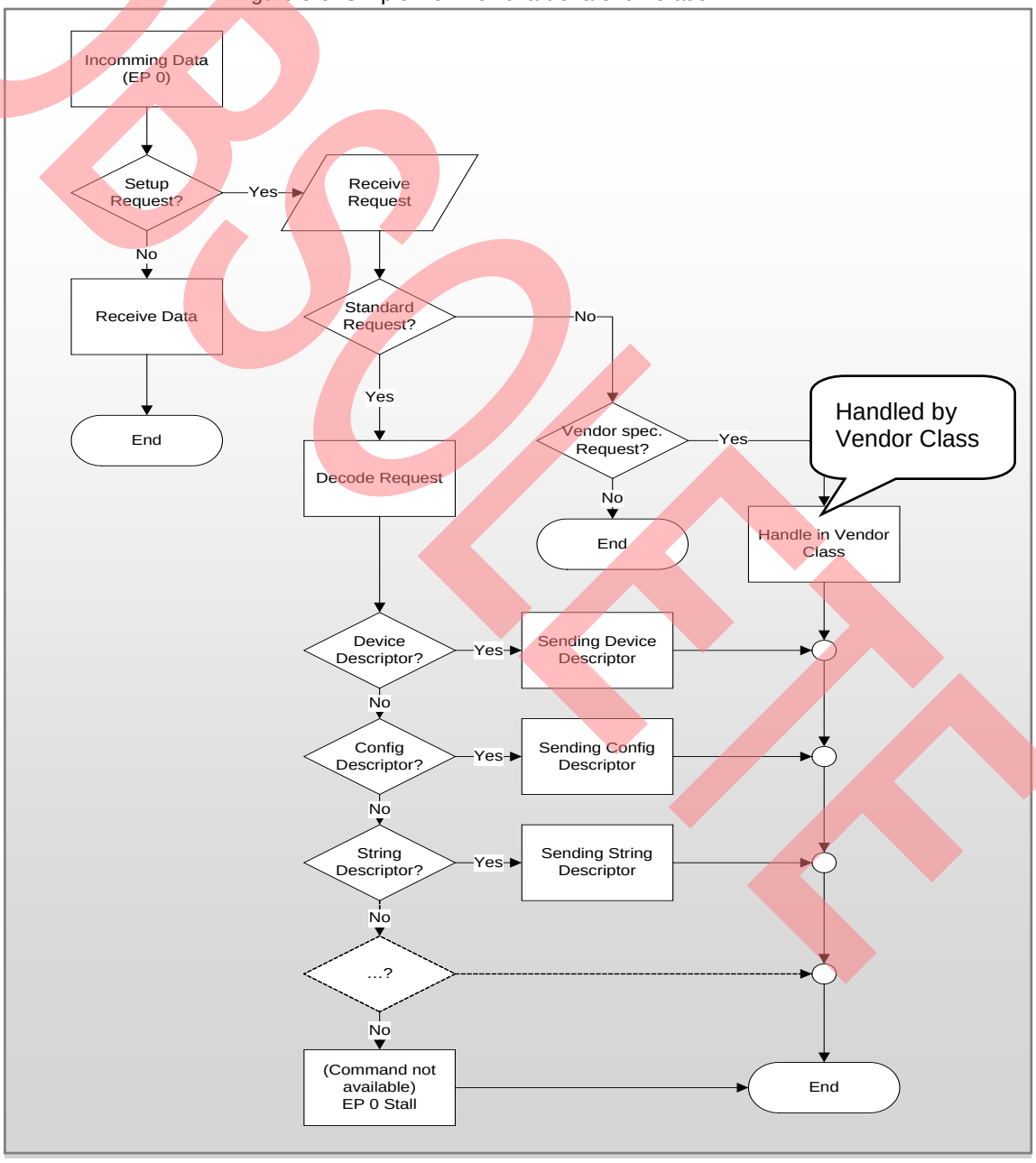
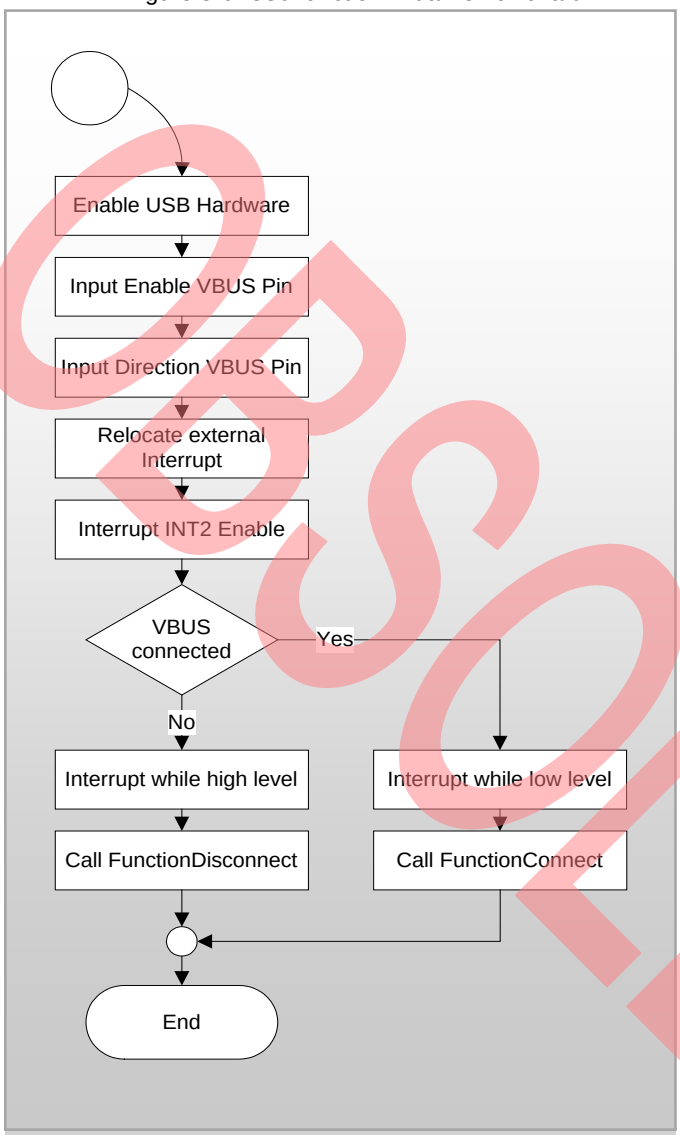


Figure 3-6: UsbFunction Initialize flowchart

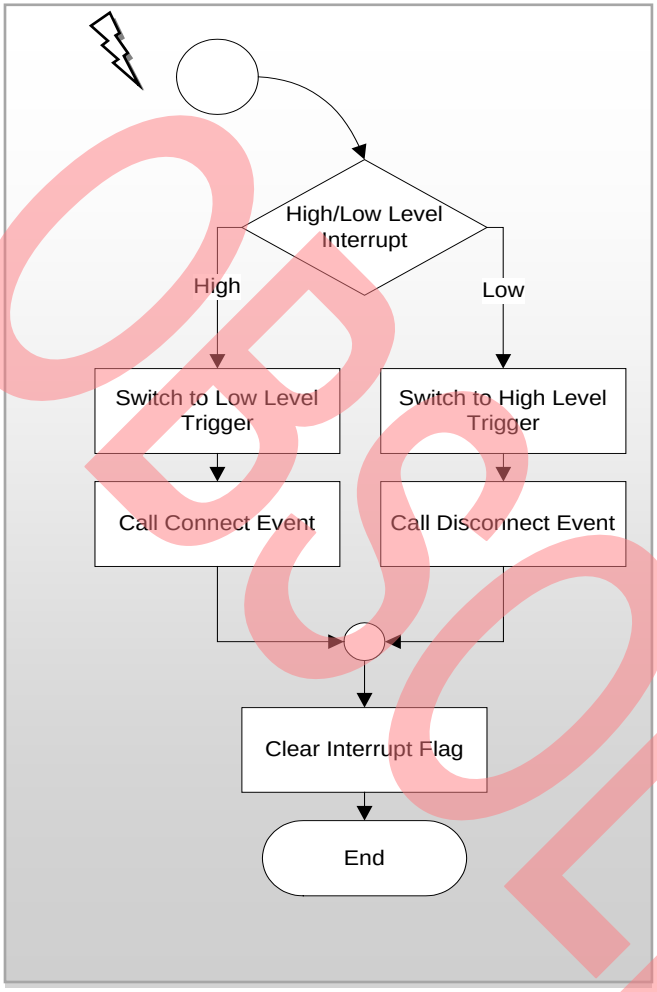


3.1.3.1 UsbFunction_Initialize()

Initiating all necessary settings for the VBUS connection interrupt is the only role of the USB Function initialization. The 16FX series MCUs have the option to execute interrupts while VBUS is high or low levelled. The current USB function initialization is done when a high level interrupt occurs.

As default, the relocated external interrupt 2 is used for VBUS detection. To change these settings, *UsbFunctionHW.h* can be edit to change these settings.

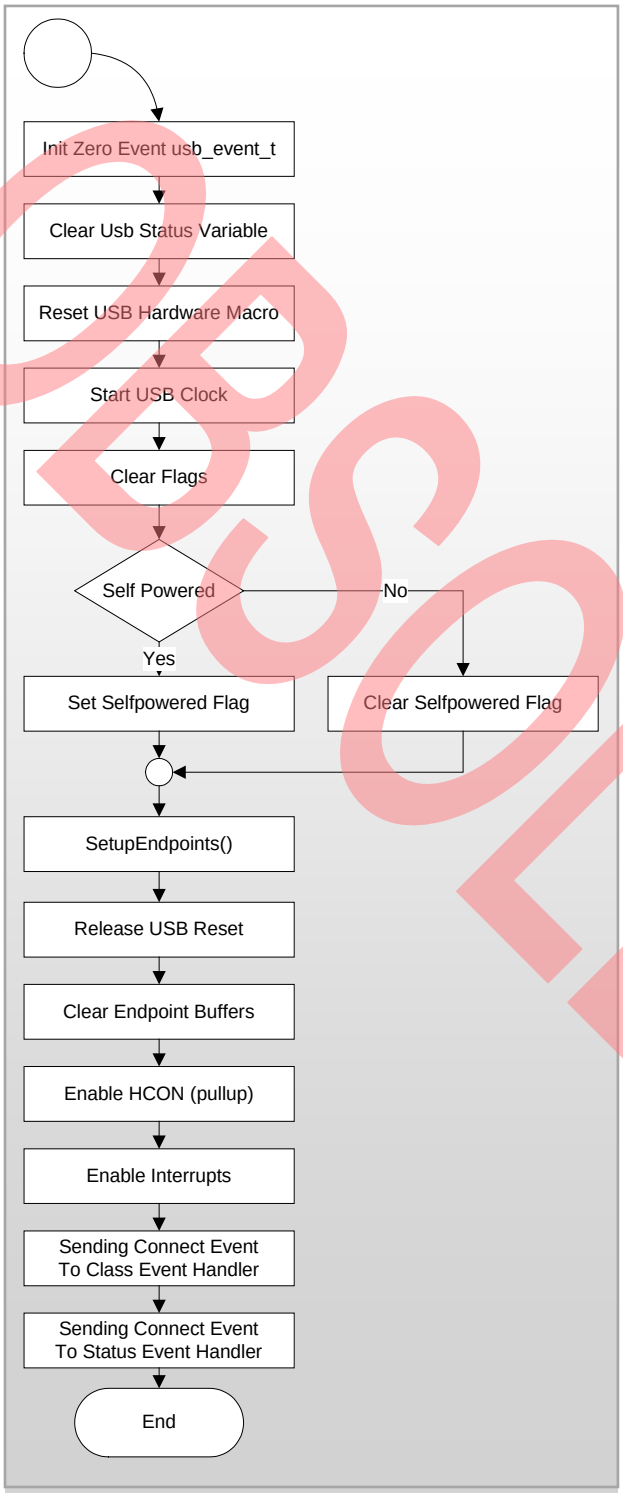
Figure 3-7: VBUS ISR flowchart



3.1.3.2 VBUS Interrupt Routine

The VBUS interrupt service routine is used to call the Connect / Disconnect Event. After a successful interrupt, the detection of high or low level is inverted.

Figure 3-8: VBUS connect event flowchart

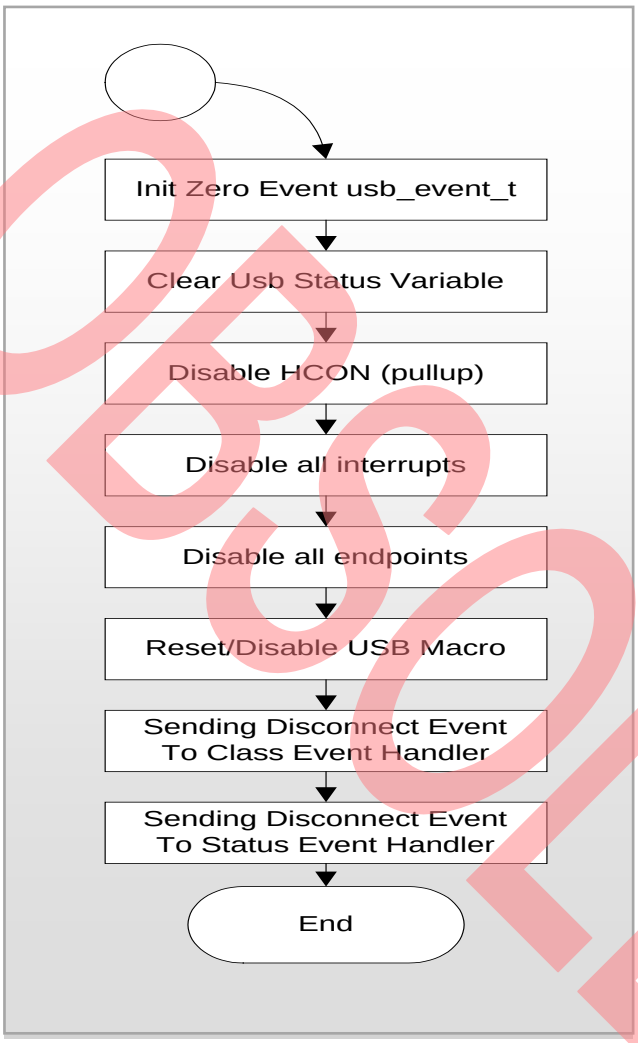


3.1.3.3 VBUS Connect Event

The connection event handles all initializations of the USB function. All information for configuring the necessary endpoints is read from the u8UsbDeviceDescriptor[]
and u8UsbConfigDescriptor[].

After all, the connect event is forwarded to the registered event handlers.

Figure 3-9: VBUS disconnect event flowchart



3.1.3.4 VBUS Disconnect Event

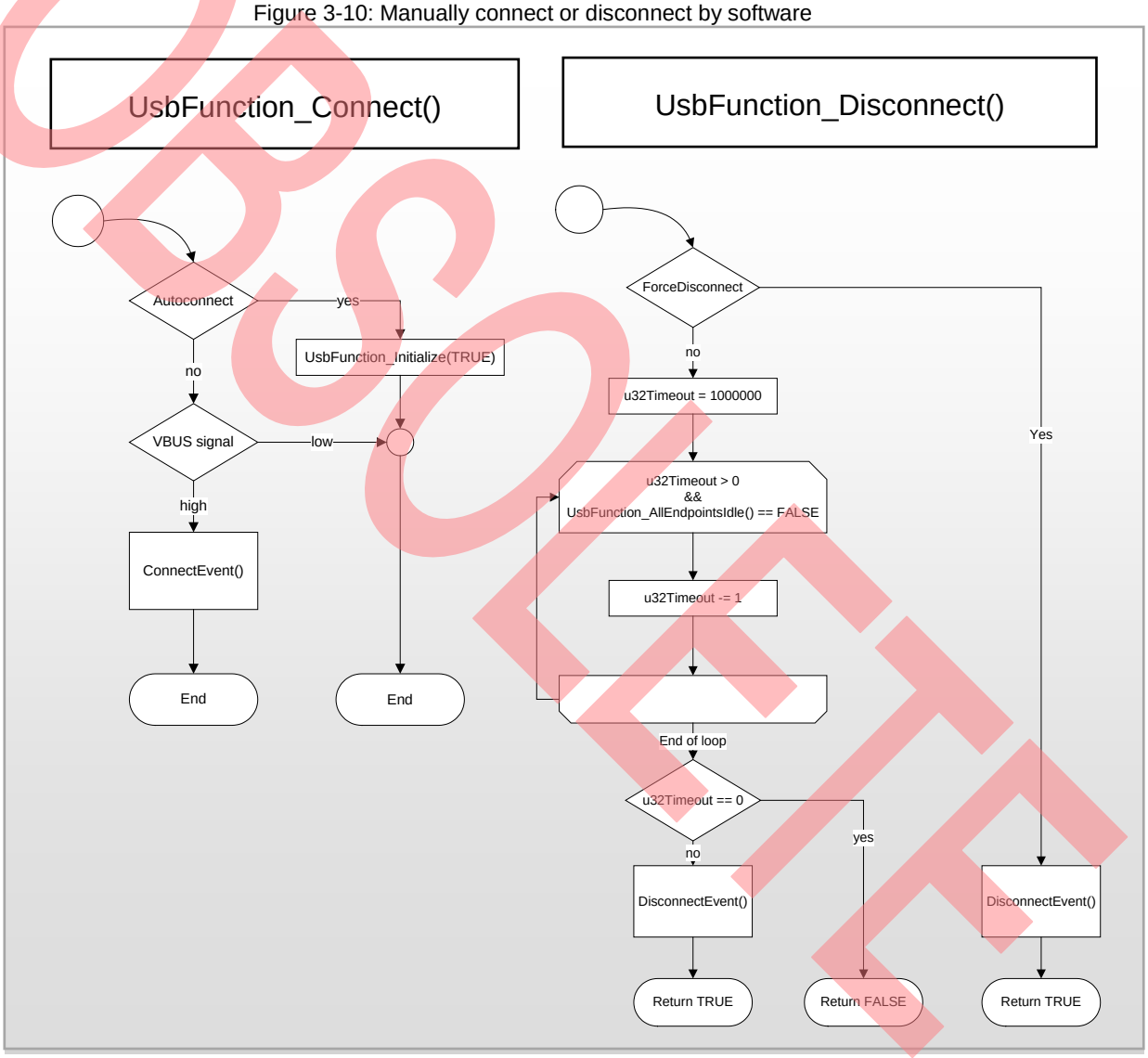
The disconnect event has the job to disable all features. After disabling all USB function macro features, the disconnect event is forwarded to the registered event handlers.

3.1.3.5 Manual Connect / Disconnect:

Normally the USB function functionality is automatically enabled after a connection. For some reasons the USB function hardware abstraction layer can be manually connected or disconnected. This let the programmer the choice between re-enable automatically connection and manual connection. If the connection fails, FALSE will be returned.

To manually disconnect the USB function hardware abstraction layer, the procedure has to check if any transfer is running. The procedure waits until a timeout occurs. If processes are still running, FALSE will be returned and the disconnect procedure will be cancelled. Another possibility is forcing the procedure to cancel all transfers and switch off the USB function hardware abstraction layer.

Figure 3-10: Manually connect or disconnect by software



3.1.3.6 Enumeration – Receiving Standard Requests

The enumeration process is handled as polled transfer, because most packages are small and fast to send. While using small packages after leaving an interrupt, the interrupt would be called again. It would not make much sense to use interrupts in this case, because the normally computers are doing the enumeration very fast. With slow hosts this can be a problem.

After a SETP flag was received, the setup has to be received from the USB FIFO buffer. The request will be decoded in an extra procedure, which considers which respond shall be send. Responds can be different descriptors or vendor class specific requests.

Figure 3-11: Enumeration sequence diagram

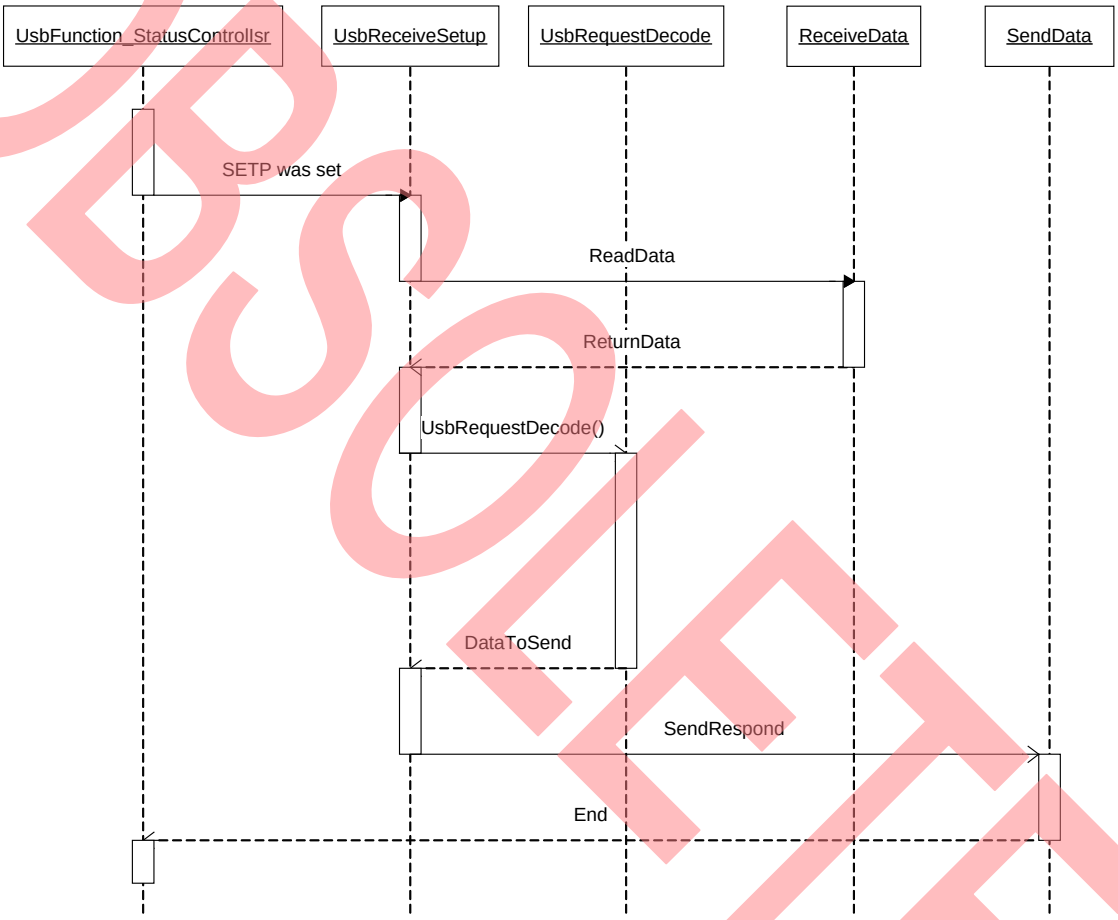
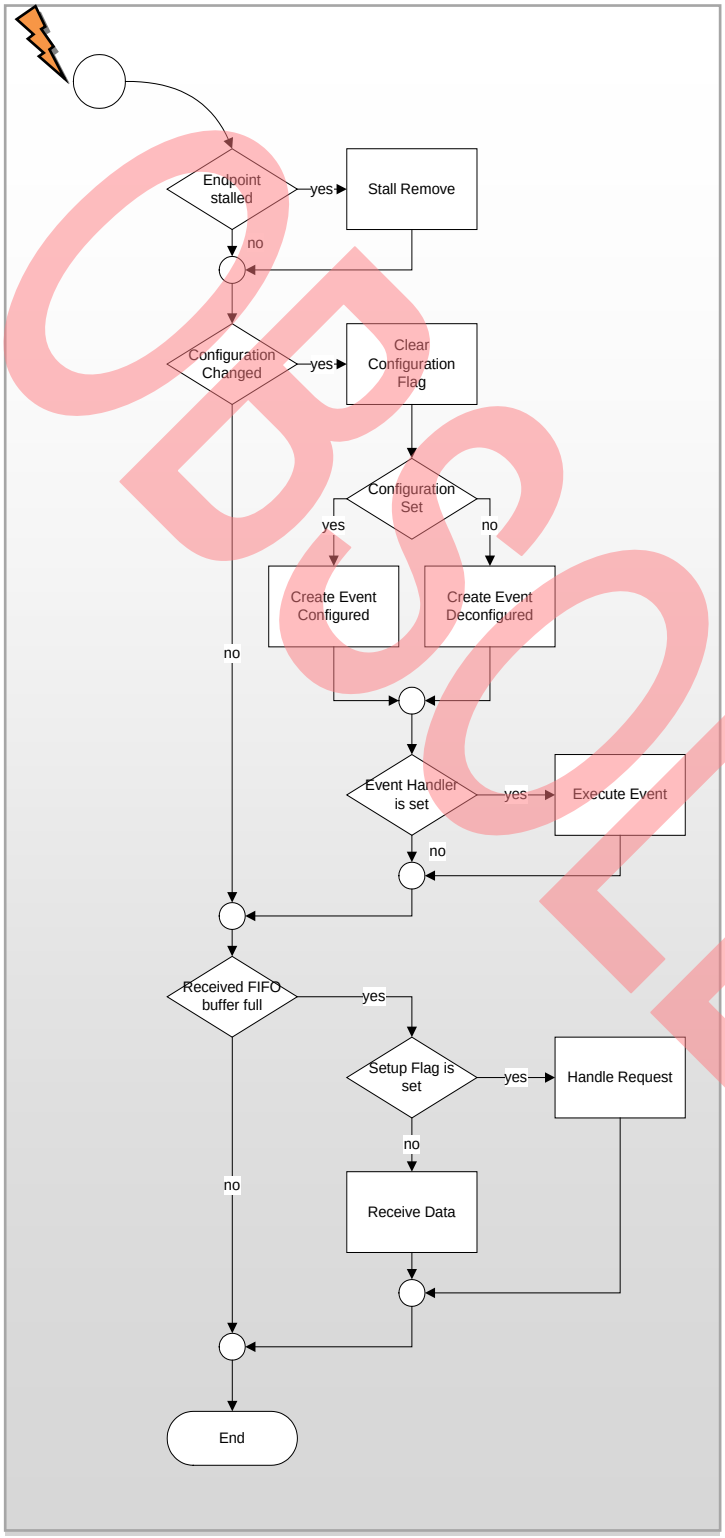


Figure 3-12: UsbFunction status control ISR flowchart



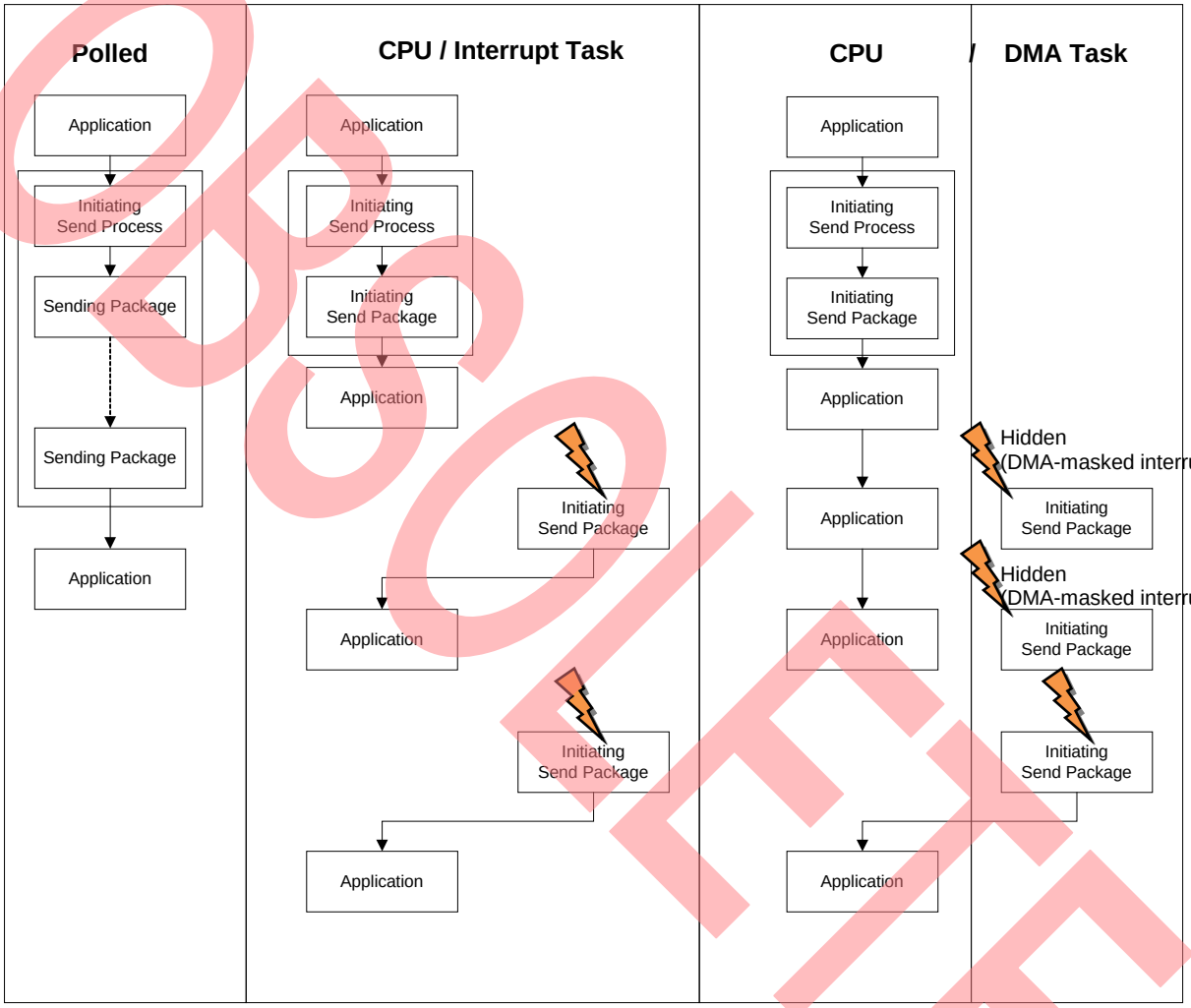
3.1.3.7 Control ISR: UsbFunction_StatusControlIsr()

The status control ISR is used for recognizing different status flag changes like configuration, wakeup, sleep, etc. But also for recognizing a setup flag and receiving data via endpoint 0.

3.1.3.8 Sending Data

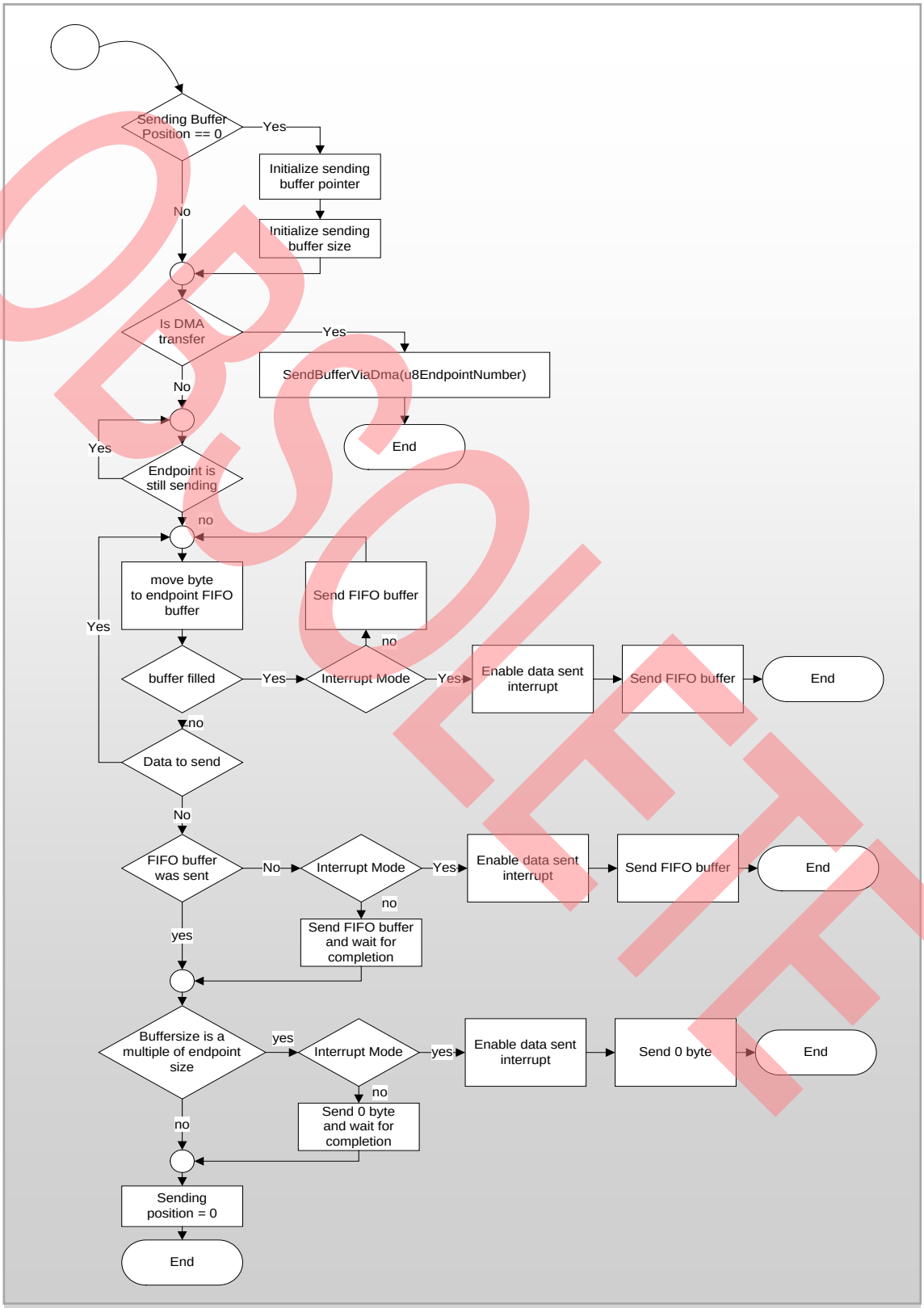
Sending data is available in 3 different types: polled, MCU interrupt and DMA regulated. Polled transfers are meaningful while the application waits for data to be sent. MCU interrupt controlled transfers are useful for small packages with less main application task priority. DMA transfers are used for big packages with a parallel running main application while transfering data at the same time.

Figure 3-13: Different sending modes sequence diagram



The endpoint ISR has to handle reception and transmission interrupts. If DMA is used, the ISR is only called after a group of packages was sent via DMA. This group can be the last or one of a multiple group transfer. The DMA macro can only used with word transfer in the USB macro. If the data package size is odd, the last byte has to be sent manually. Figure 3-14 demonstrates the data transfer function. This function can handle polled, MCU interrupt or DMA controlled transfers.

Figure 3-14: UsbFunction_SendData

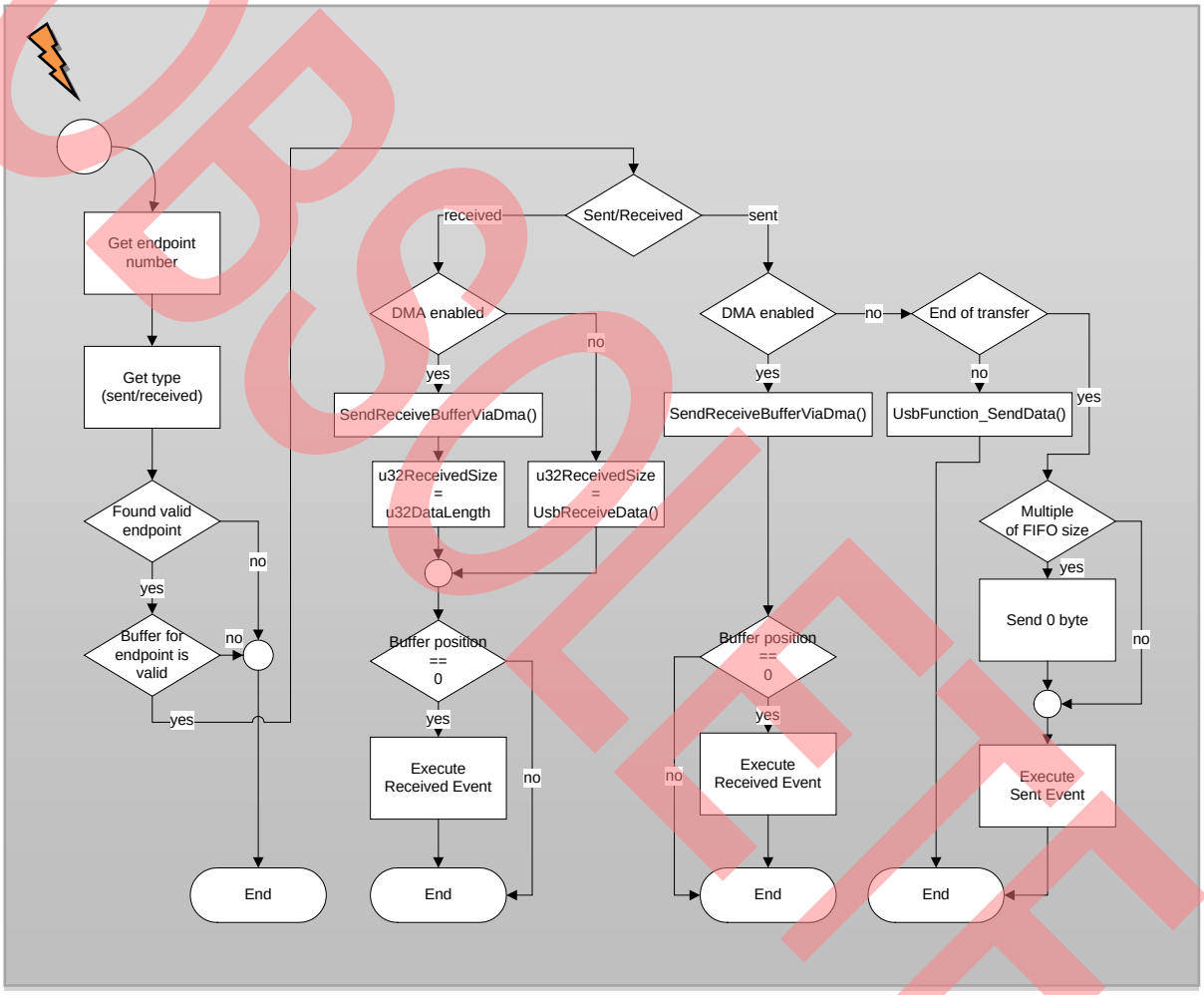


3.1.3.9 Data Sent / Received: UsbFunction_EndpointIsr()

This interrupt service routine handles all data interrupts of the endpoints 1 to 5. For all endpoints of type HOST OUT (microcontroller receives data), which are configured in the configuration descriptor, have enabled data reception interrupts. These kinds of interrupts are handled in the reception routines.

All data transmission interrupts of the configured endpoints of type HOST IN (microcontroller sends data) are disabled, because normally these endpoints are ready to send for almost all the time, which calls an interrupt. For using DMA, these interrupts have to be enabled while starting a new DMA package (HOST IN).

Figure 3-15: Endpoint ISR flowchart



3.1.3.10 DMA Handling

For each endpoint, the USB function library can use one DMA channel. These channels can be dynamically changed with the function `UsbFunction_ChangeDmaSettings`. DMA is optional available for using the USB function library, because for smaller data packages, DMA has a bigger overhead. For smaller data packages, data can be sent in polled transfer mode or in interrupt transfer mode. For receiving data with the microcontroller only interrupt controlled transfers or DMA transfers are available. Receiving data has the limitation that only expected data lengths are supported, because a DMA transfer has to know which length of data has to be transferred. Only bulk transfers ensures, that no packages can be lost and all data is incoming. Bulk transfers will be automatically resent if an error during the transmission occurs. The hardware abstraction layer will automatically do this.

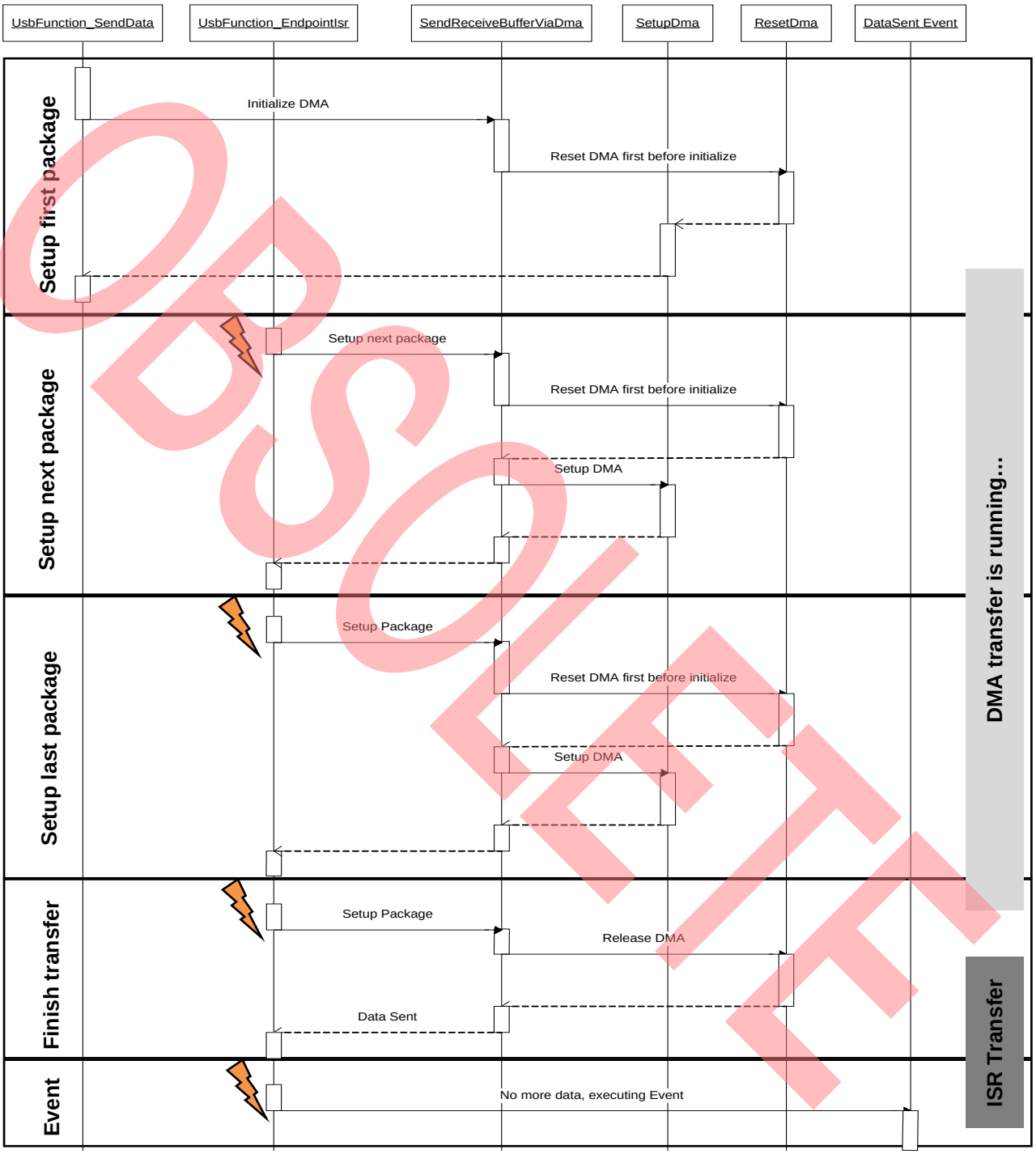
Figure 3-16: Table of available DMA modes

	Polled	MCU Interrupt	DMA
Sending Data (HOST IN)	Yes	Yes	Yes
Receiving Data (HOST OUT)	No	Yes	Only bulk transfers Only fixed data lengths

Figure 3-17 shows a DMA transfer sequence. The sequence demonstrates a sending transfer (HOST IN), which is almost the same diagram for receiving data. Before data can be received, a package has to be requested by software. While using the sending direction, the initialisation to send data also stores the size of data. The sending procedure automatically does all initialisations, which are necessary.

First, the DMA has to be initialized, which is normally done by the `UsbFunction_SendData` procedure or the `UsbFunction_EnableReceiveDma` procedure. Both procedures only enable one session for sending or receiving the defined size of data. After finishing the transfer an event will be called. Only for sending data, the last package is normally sent via MCU interrupt transfer, because all transfers which are a multiple of the endpoint FIFO buffer have to be terminated with a 0 byte transfer or all odd sized data transfers have to be terminated by the last byte missing. Both cases can't be handled with DMA transfers and will be automatically continued and finished as interrupt transfers.

Figure 3-17: DMA transfer sequence diagram

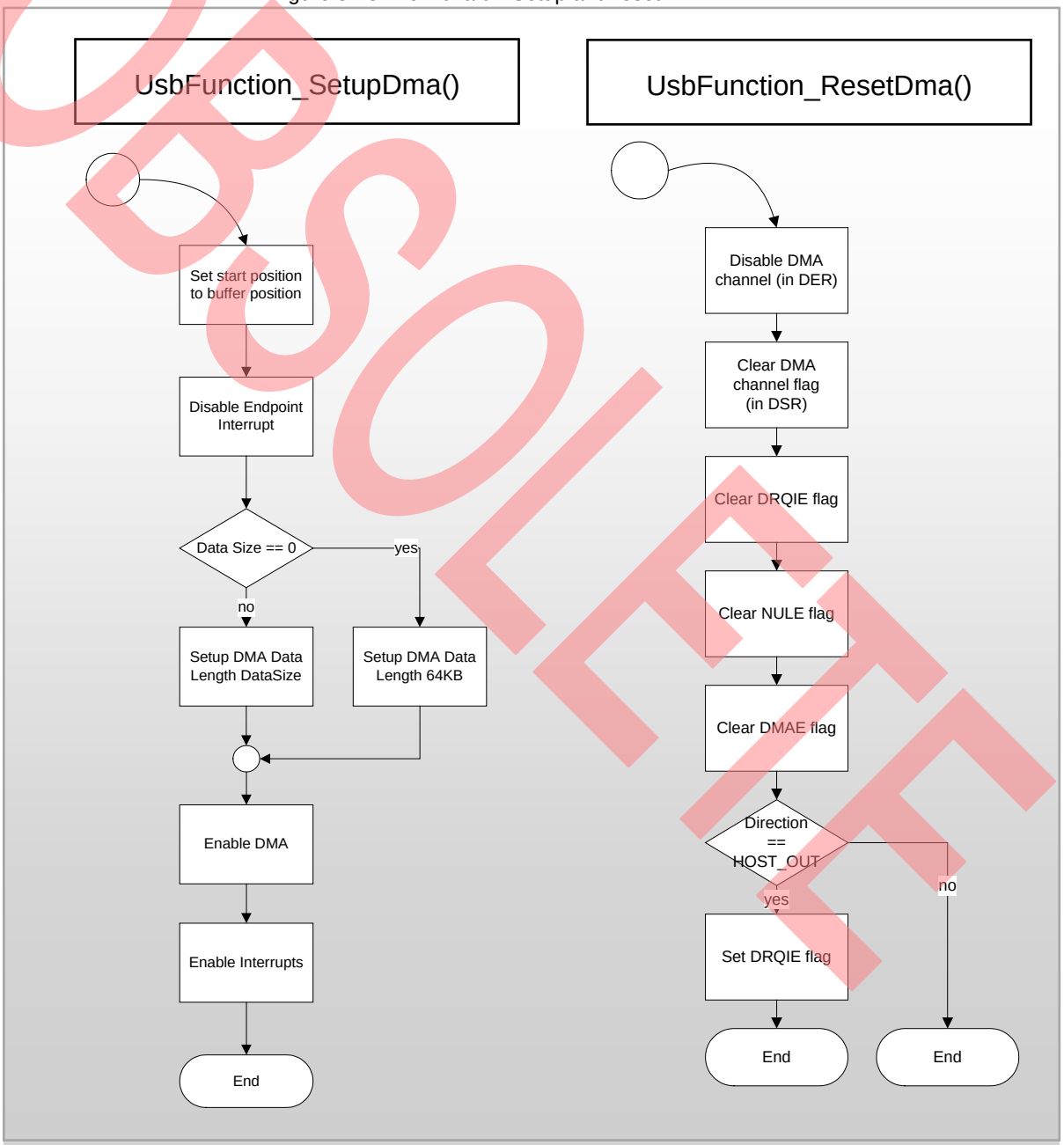


3.1.3.11 Setup and reset DMA

The DMA setup and reset are the smallest part of DMA transfers. The setup only sets the size of data to be transferred, start position, data direction and USB FIFO buffer address.

Figure 3-18 shows how this setup works. To use DMA transfers, the interrupt handler must be enabled for the specified endpoint at the end of setup. The DMA module uses internally the interrupts to synchronise its transfers. For the time DMA transfers are enabled, the resource interrupts are bypassed to the DMA module. After the DMA has finished, the usual interrupt service routine of the specified endpoint will be called. This must be considered while using different transfer modes. Interrupt service routines have to be called from interrupt transfers and DMA transfers.

Figure 3-18: Flow chart – Setup and reset DMA



3.1.3.12 Send/Receive Buffer via DMA: SendReceiveBufferViaDMA()

This procedure is used for validating and setup a new data package from a buffer via DMA. It can be used for sending or receiving data addicted by the endpoint direction. It is not responsible to initiate data transfers. The current position of sending or receiving a data stream is stored in an extra buffer structure, which contains the pointer to the data buffer, the size of data to be sent and the current position of sending or receiving data. The initialisation is normally done by the `UsbFunction_SendData` procedure or the `UsbFunction_EnableReceiveDma` procedure.

Before a new DMA package can be considered, DMA transfers on the used DMA channel have to be resetted.

Some cases has to be validated before setup a new DMA transfer: End of transfer, only last byte is missing, data packages lower than 64KB, data packages higher than 64KB, data sizes multiple of 2 or multiple of endpoint FIFO buffer size.

End of transfer:

The end of transfer case only sets the buffer position variable to the end of transfer. This signalizes the following processes, that data was completely sent or received. In case of an outgoing host-IN transfer, a zero package has to be sent, if the last package size was a multiple of the endpoint FIFO size.

Last byte is missing:

While using the DMA mode with the USB hardware abstraction layer, the DMA module can only used with word (2 byte) transfers. If only one byte is missing, the next transfer is a one-byte transfer. This can only be done by sending or receiving the last transfer by additional ending interrupt mode.

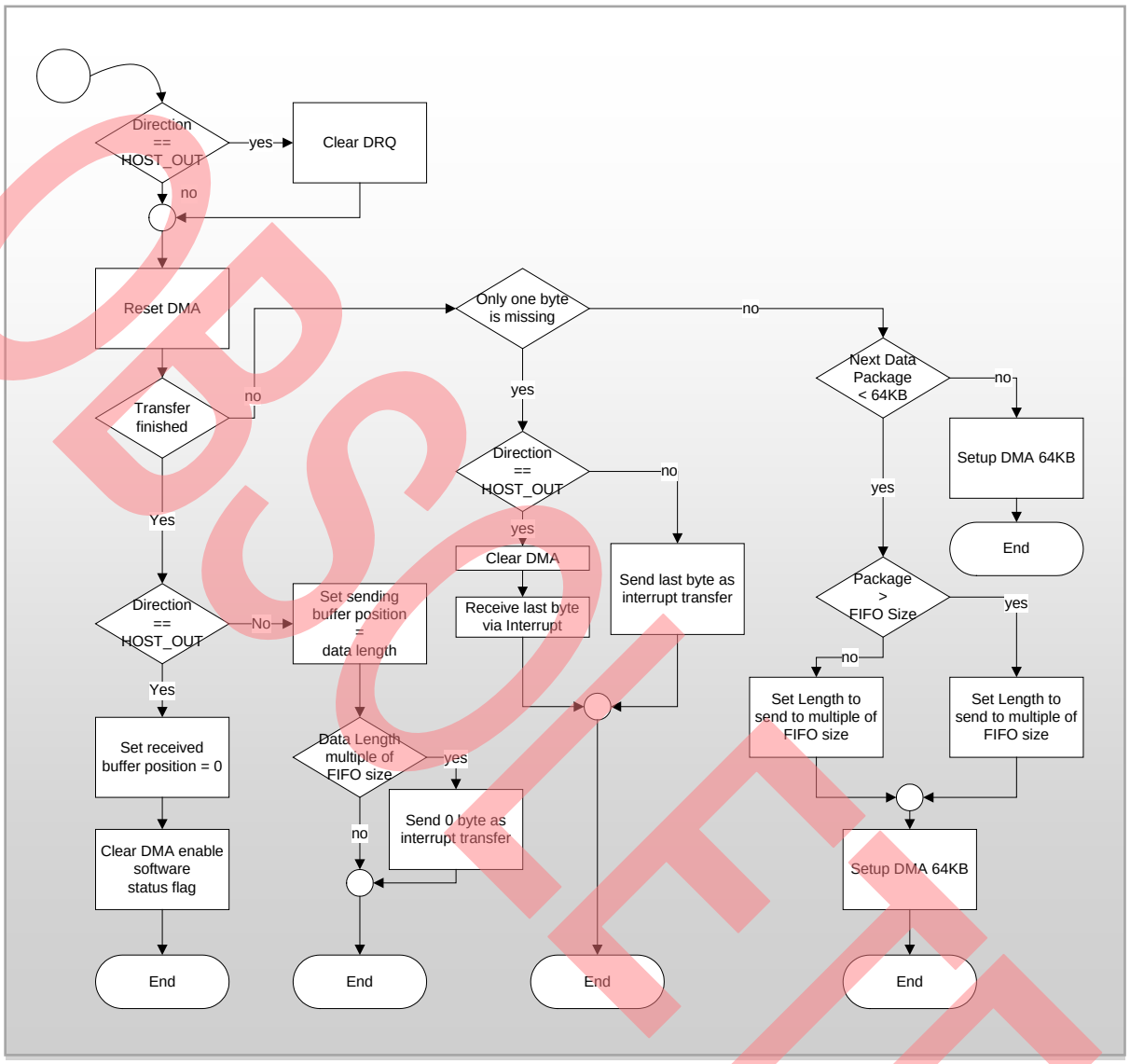
Packages lower or higher than 64KB:

The Cypress DMA abstraction layer does support transfers up to 64KB. The maximum of data which can be sent at once can be 64KB or less than 64KB restricted on the other cases listed above and below.

Data Sizes multiple of 2 or multiple of the endpoint FIFO buffer size:

All data transfers except the last package has to be a multiple of the endpoint FIFO buffer size. The last DMA transfer has to be a multiple of two.

Figure 3-19: Flow chart SendReceiveDataViaDMA()

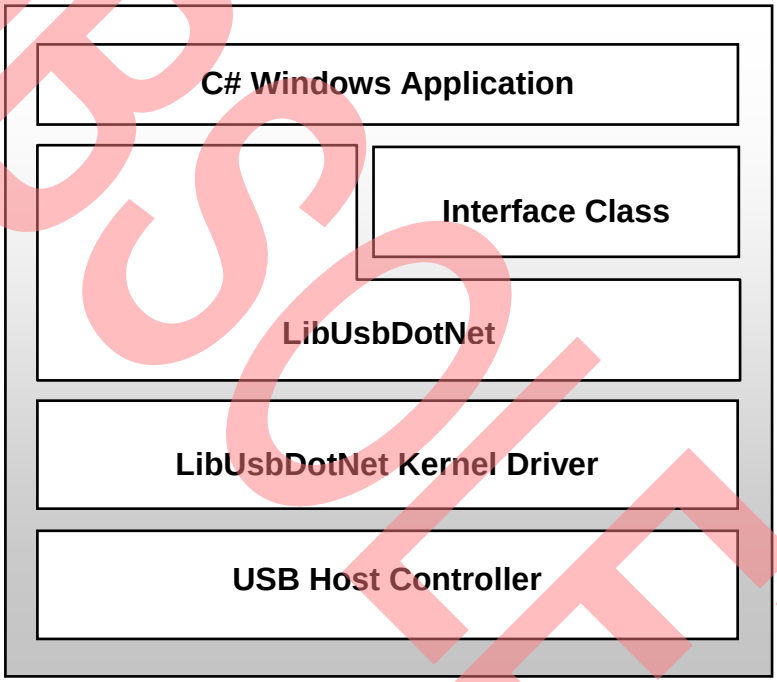


3.2 LibUSB with .NET Framework

LibUSB for Windows has a wrapper library called LibUsbDotNet. This library offers developers an extensive object-oriented API. To provide a simple API, which handles standard procedures like connecting, disconnecting, hot-plug, sending and receiving data, it is useful to write a C# class, which implements this features stable. This class can be a reference project for programming LibUsbDotNet for advanced developers or can be used as an interface class module for beginners to write own USB based applications.

The developer can use the interface class and the original LibUsbDotNet API from his C# project. All necessary parts are handled by the interface class, but can be also handled directly. Disadvantages by using the LibUsbDotNet API directly are possible program crashes and crashes of the microcontroller, which can be prohibited by the interface class.

Figure 3-20: LibUsbDotNet implementation



3.2.1 Interface Class Module: UsbFunctions.cs

The interface class uses the direct API of LibUsbDotNet. LibUsbDotNet mixes the Windows own USB drivers (WinUSB) with the LibUSB drivers (LibUSB Kernel Drivers) and offers an API supporting both APIs in one library. WinUSB can be used to access HID, which has the advantage to have no need of drivers. The use of HID classes for custom protocols is working in the most cases, but is also not really USB compliant. With LibUSB drivers it is possible to handle USB custom class access. The UsbFunctions module builds the backend API between LibUsbDotNet and own applications.

In the UsbFunctions module exists 6 classes, two of them are event classes:

- UsbFunctions Class: The main class, which handles connection, disconnection and some enumeration specific configurations.
- EndpointIN: A class, which handles a reading endpoint. Key benefits are FIFO buffers, received events, and hot-plug
- EndpointOUT A class, which handles a writing endpoint. Key benefits are sending data, receiving data, sending errors, and hot-plug.
- UsbHandshake This class handles an optional protocol handler, which uses two endpoints to setup a handshaking.
- Connect Event Called if the specified USB device was connected / disconnected
- Data Received Event Called if data on a reading endpoint was received

3.2.2 UsbFunctions Class

Main parts of the UsbFunctions Class are the USB device handling. This class handles connecting and disconnecting a specified USB device, handles different events and initializes all necessary endpoints for data communication.

Some key features are hot-plug, FIFO buffers for received data and handshaking FIFO buffers for sending data. Also errors and status events are handled and interpreted.

The UsbFunctions class uses the namespace own EndpointIN and EndpointOUT classes as a backend to communicate with the different endpoint pipes.

Figure 3-21: UML diagram: UsbFunctions Class

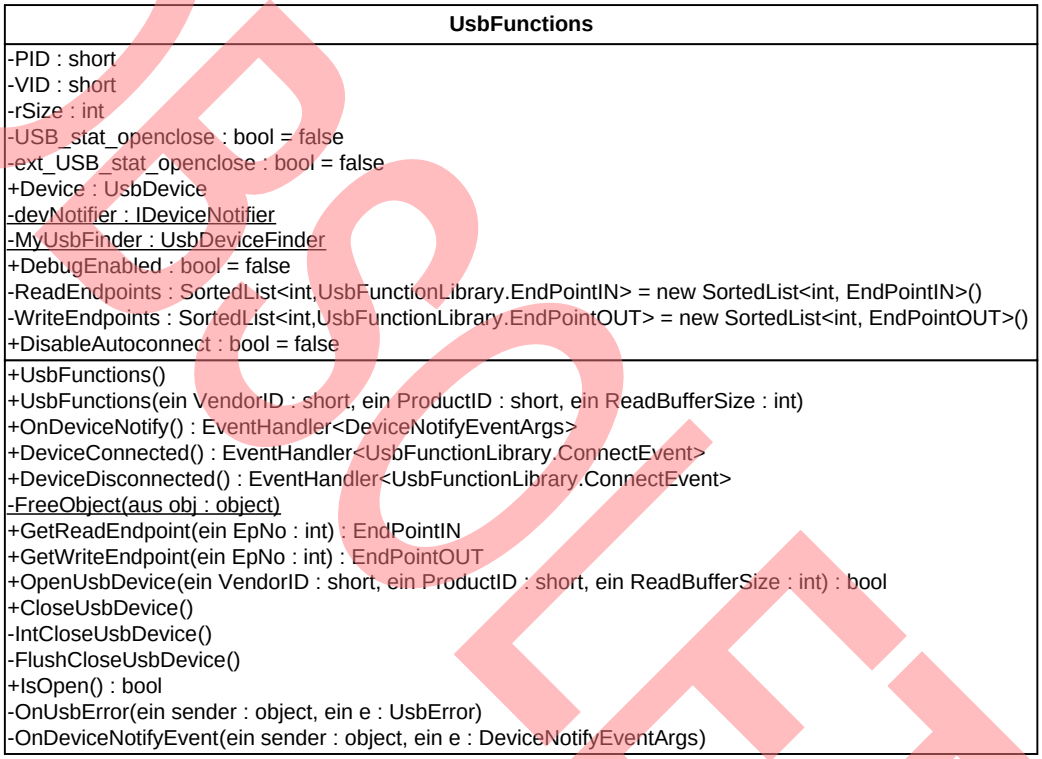
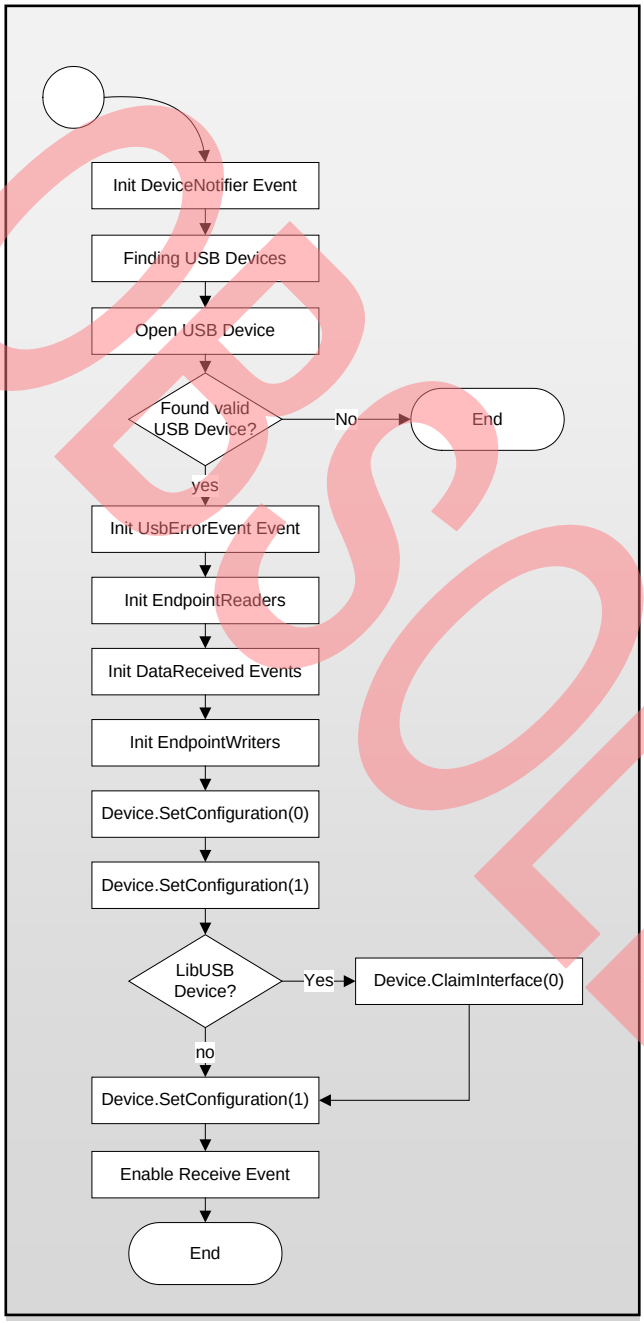


Figure 3-22: Open USB device flowchart



3.2.2.1 Open USB Device Procedure: OpenUsbDevice

Figure 3-22 describes the process of opening a USB device. It can be seen that it is not just a simple open and close procedure, because different steps have to be done:

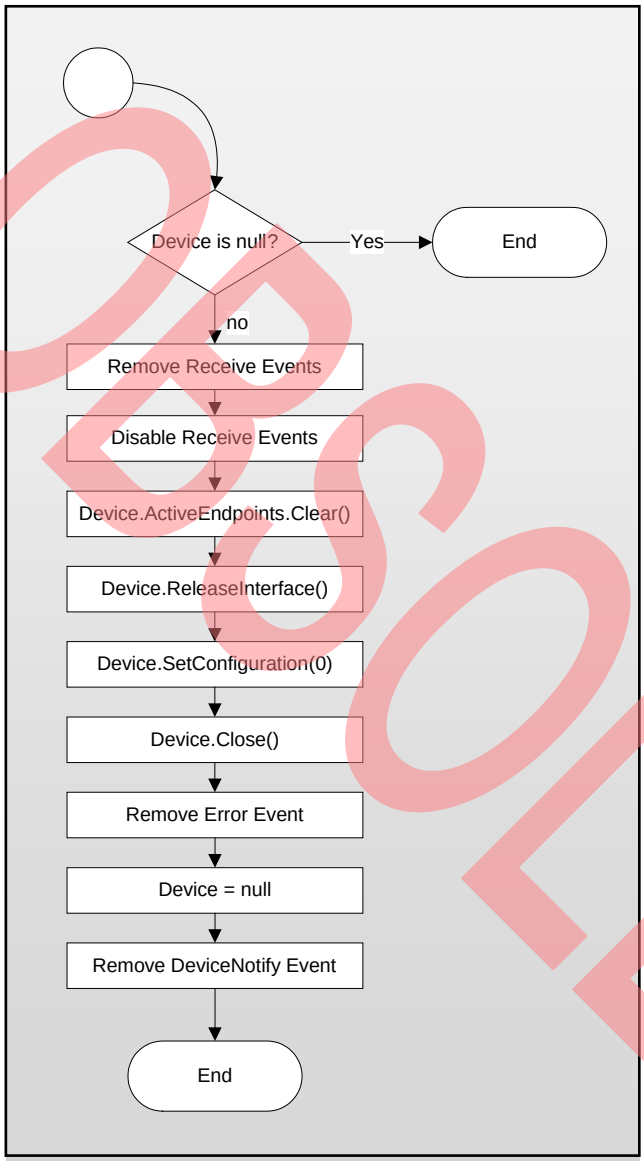
At the beginning also the DeviceNotifier Event will be initialized. This event handler is used to detect hot-plug capabilities (see 3.2.2.3) in the interface class.

To open a USB device, first it has to be found on the USB bus. A USB device can be addressed on different busses and hubs. The searching feature is implemented in the LibUsbDotNet library and scans all busses and hubs, which are connected on the computer. After the scan, the device will be opened. If the device was initialized it is valid (see 3.2.2.2).

The next step initializes the required event handlers and the endpoint writer/reader. The UsbErrorEvent can handle errors as an event. The DataReceived event is called when data is available. The interface class handles automatically all received data events, if the event handler is not externally set.

To initiate the USB device, the device configuration has to be set and the event receiver for every receiving endpoint has to be enabled.

Figure 3-23: Close USB device flowchart

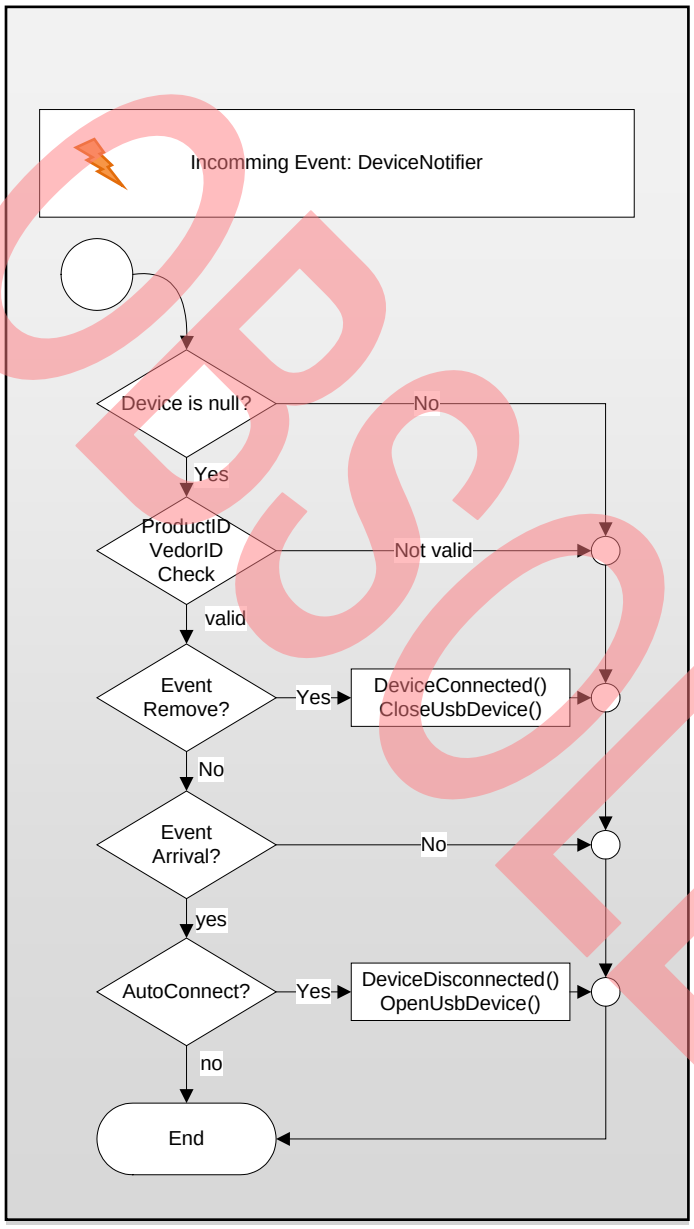


3.2.2.2 Close USB Device Procedure: CloseUsbDevice

To have no risk while closing an uninitialized device, first it has to be checked for validity. After successful check all events can be disabled and the endpoints can be cleared. Before the device can be closed, it has to be configured with no active configuration (set configuration 0).

After all the device will be closed and cleared by setting it to NULL.

Figure 3-24: Device status change flowchart



3.2.2.3 DeviceNotifier Event

This event is not constrained by an opened or closed USB device. This event handler will be executed while plugging in or removing any USB device.

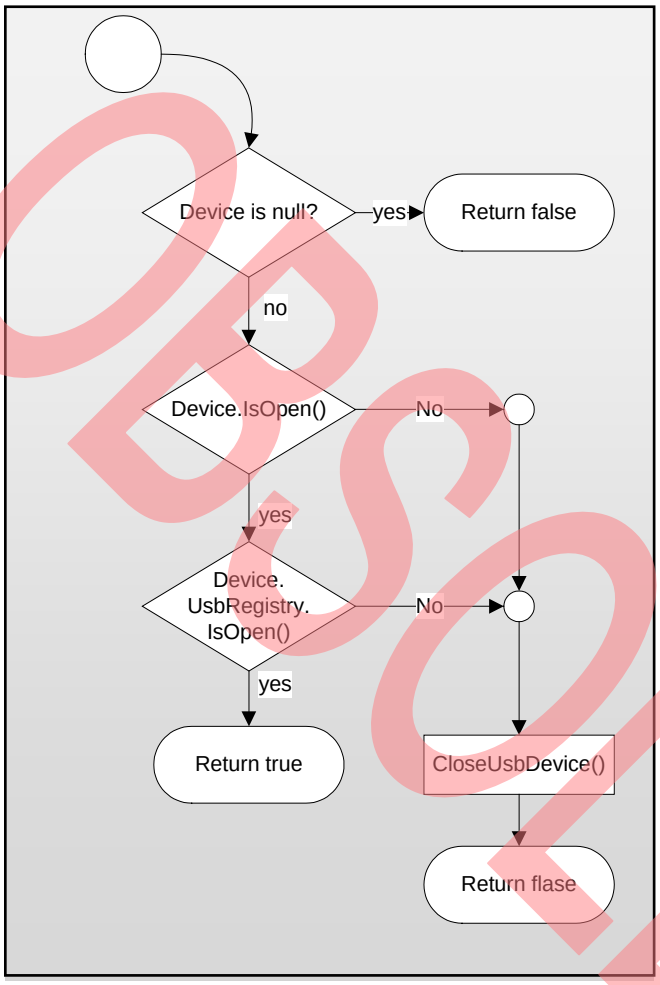
This event is enabled by an external call of the procedure OpenUsbDevice() and will be disabled by an external call of CloseUsbDevice().
Class internal calls leaving the event enabled.

In the USB interface class, this event is used for hot-plug capabilities.

After a successful validation for Vendor- and Product-ID, the device can be automatically opened or closed.

To simplify the detection of the specified device, the event handlers DeviceConnected and DeviceDisconnected can be used.

Figure 3-25: Device status flowchart



3.2.2.4 Check USB Device Status: IsOpen

This procedure is used to get the status of the current USB device and will return true, if the device is initialized.

This procedure performs not only a check; it closes also a device, which is no more available. This makes the USB access more stable, because no Exceptions will happen.

In the process, first the device will be checked if it is valid. After this it will be checked if the device is available and initialized. If it is initialized but not available it will be closed for safety reasons.

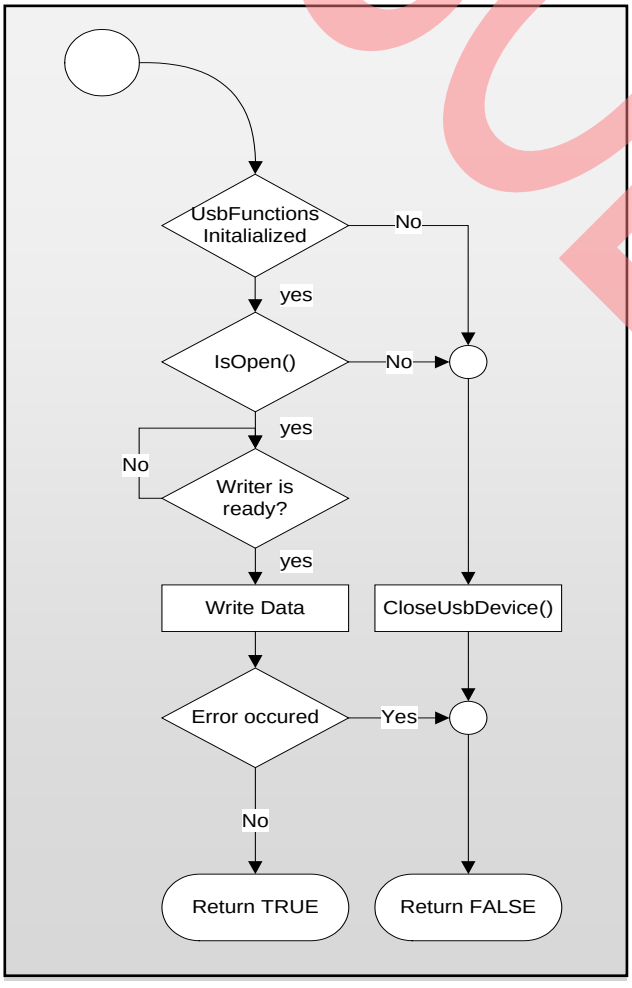
3.2.3 EndpointOUT Class

Main parts are sending data and retry sending data, if sending data fails. The EndpointOUT class uses the UsbFunctions class, to get the UsbDevice class from the LibUsbDotNet namespace to retrieve the EndpointWriter class with which data can be received from a USB device.

Figure 3-26: UML diagram: EndpointOUT class

USBBackend::**EndPointOUT**
+Writer : UsbEndpointWriter +EndpointID : WriteEndpointID +Device : UsbDevice +UsbFunctionsMain : UsbFunctions -SendingError : bool = false
+EndPointOUT() +ReInitalize(ein device : UsbDevice) +EndPointOUT(ein usbFunctionsMain : UsbFunctions, ein device : UsbDevice, ein endpointID : WriteEndpointID) +Write(ein dataout : byte[], ein timeout : int, aus byteswritten : int) : bool +Write(ein dataout : byte[], ein timeout : int, aus byteswritten : int, ein retries : int) : bool +throwDataNotSentEvent(ein sender : object, ein e : EventArgs) +NotSent() : EventHandler<System.EventArgs>

Figure 3-27: Write data flowchart



3.2.4 Write Data to USB Device: Write

The write procedure is available if the device is open and ready. This is done by the UsbFunctions function `IsOpen()`. After a successful check, the writer will be checked if it is ready. The procedure waits until the writer is ready and sends the specified data.

If the device is not available, it will be closed in the `IsOpen()` function.

After initialization of writing data, the function will check if an error occured while writing data. The UsbFunctions Class will automatically forward the error event to the endpoint class, on which the error happened. An error flag will be set, which shows, that an error occured. This flag is used to return TRUE or FALSE at the end.

3.2.5 EndpointIN Class

Main parts of the EndpointIN class are the incoming data handling and event handling. The EndpointIN class uses the UsbFunctions class to get the UsbDevice class from the LibUsbDotNet namespace to retrieve the EndpointReader class with which data can be received from a USB device.

EndPointIN
-BufferList : List = new List<byte[]>() +BytesReceived : int = 0 +EndpointID : ReadEndpointID +Reader : UsbEndpointReader +Device : UsbDevice +ReadBufferSize : int
+EndPointIN() +ReInitalize(ein device : UsbDevice) +EndPointIN(ein device : UsbDevice, ein endpointID : ReadEndpointID, ein readBufferSize : int) -OnRxEndPointData(ein sender : object, ein e : EndpointDataEventArgs) +GetReadPackage() : byte[] +FlushReceiveBuffer() +Enable() +Close() +DataReceived() : EventHandler<UsbFunctionLibrary.DataReceivedEvent> +GetDataReceivedEventHandler() : EventHandler<UsbFunctionLibrary.DataReceivedEvent>

3.2.5.1 Receive Data

Data can be received in polled or event triggered mode. If no event was initialized, the received data is automatically stored (en-queued) in a message buffer queue. This data can be de-queued (received package wise) by using the internal function

The following flowcharts (Figure 3-29 and Figure 3-28) illustrate the two operations, which are required to realize event triggered and polling initiated reception data. The incoming data is received internally via a receive event initiated by the LibUsbDotNet API (Figure 3-28). This event is used to get the received buffer and store it into an interface class internal list (FIFO buffer), if no external event `ReceivedEvent` was set.

To read the incoming data, the function `GetReadPackage()` is used (Figure 3-29). This function can only be used if the user is sure that data is ready to read. Reading the BytesReceived value can check if data is available to read. If the list `BufferList` (FIFO buffer) contains data, only a coherent data package will be returned. The data size of this package will be substracted from the `BytesReceived` value while calling `GetReadPackage()`.

Figure 3-28: Received event flowchart

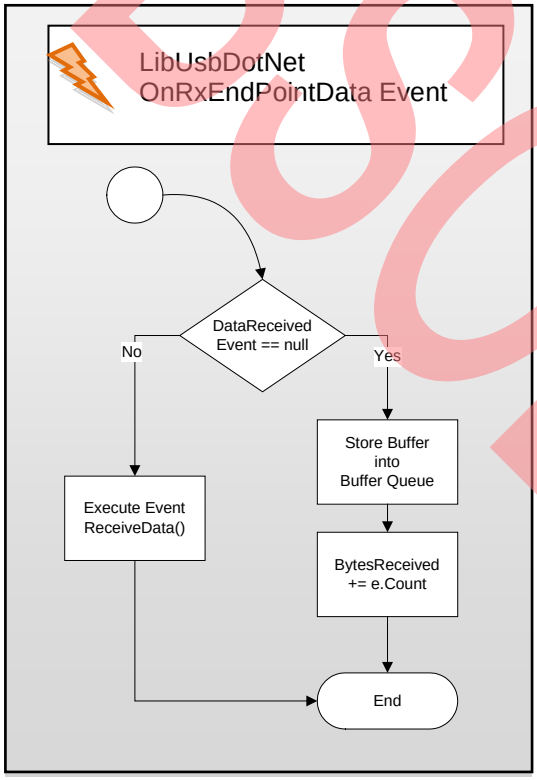
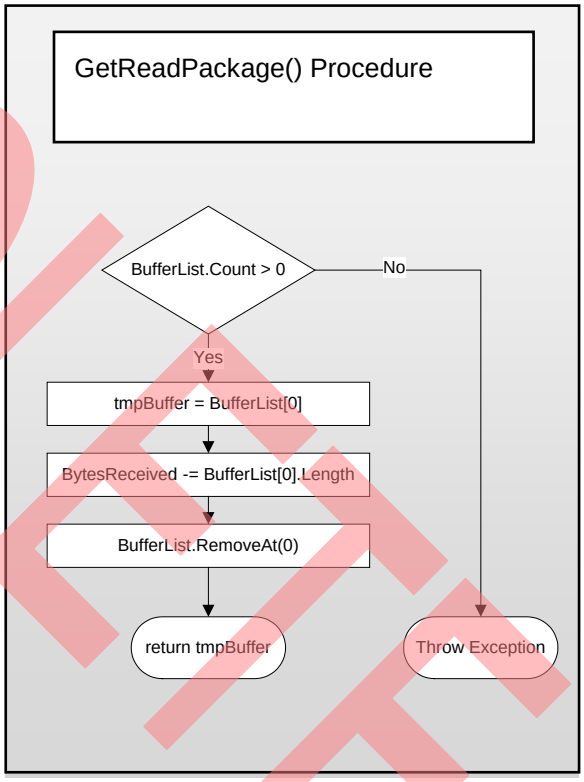


Figure 3-29: Readout data flowchart



3.2.6 UsbHandshake Class

The UsbHandshake Class can be used for sending large package sizes, for which the microcontroller needs some time to process. A handshake from the microcontroller will signalize the UsbHandshake class to send the next data package. Advantages are non-freezing applications, but slower data transfers, if the package sizes are too small (< 1KB).

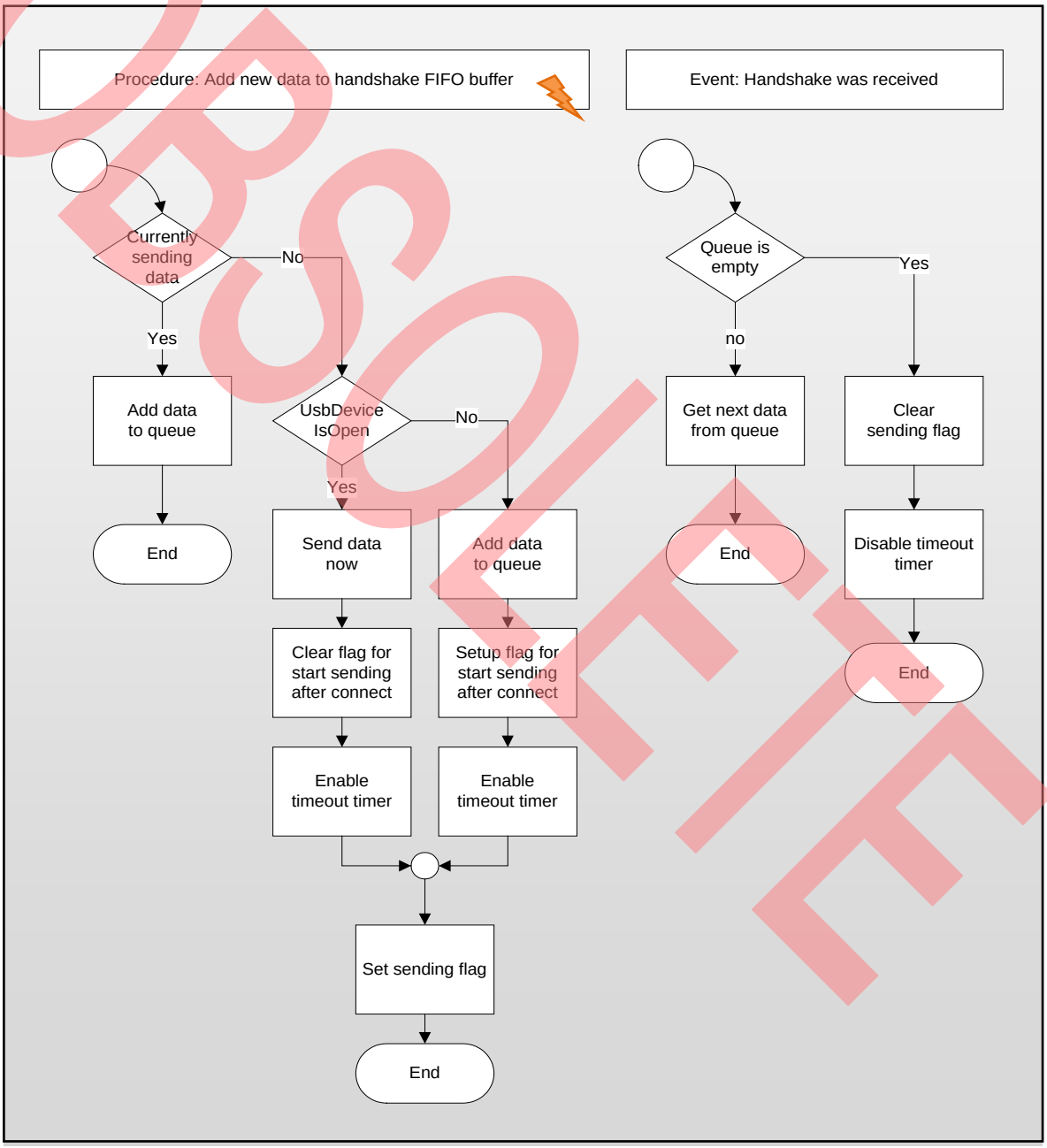
USBBackend::**UsbHandShake**
-TimeoutTimer : Timer = new Timer() -Usb : UsbFunctions -Timeout : uint = 0 - _TimeOut : uint = 100 +SendingData : bool = false +DataReceived : bool = false +PackagesToSend : Queue = new Queue<byte[]>() -EnpointNumberSending : int = 0 -EndPointNumberHandshake : int = 0 -InitSendAfterConnect : bool = false -LastData : byte[]
+UsbHandShake(ein _Usb : UsbFunctions, ein _EnpointNumberSending : int, ein _EndPointNumberHandshake : int) +UsbHandShake(ein _Usb : UsbFunctions, ein _EnpointNumberSending : int) -~UsbHandShake() +Clear() +IsSending() : bool +NumberOfPackages() : int -UsbHandshake_SendingError(ein sender : object, ein e : EventArgs) -UsbHandShake_TimeoutTimer(ein sender : object, ein e : EventArgs) -UsbHandShake_Received(ein sender : object, ein e : DataReceivedEvent) +WriteDataUsb(ein Data : byte[])

3.2.6.1 Write data into handshake FIFO buffer: Add data / Handshake received

Writing data via USB is constrained by receiving a handshake from the specified handshake endpoint. The writing procedure has to add data to the package to the send queue. Every time, a handshake was received, the next package in the queue has been sent.

If the queue is empty, the writing procedure has to initiate the first transfer. If the device is closed, because it was removed for example, the packages will be sent automatically after the device was connected. Figure 3-30 shows the abstract writing procedure and the handshake event.

Figure 3-30: Add data to queue / Handshake was received



4 Application Programming Interface

APPLICATION PROGRAMMING INTERFACE

4.1 Microcontroller USB Function Library

4.1.1 Files

- *UsbFunction.c*: USB Function Library
- *UsbFunction.h*: USB Function Library Header File
  - can be used to setup `__far` buffers instead of `__near` buffers
  - can be used to disable DMA
- *UsbFunctionHW.h* USB Function Library Hardware Settings
  - is used to setup hardware specific defines
- *dbg.c* debug functions
- *dbg.h* debug headers
  - can be fully disabled (no code usage in flash or RAM)

4.1.2 Interrupt Sections

All interrupt specific settings are defined in the *vectors.c* file. In 3 sections, the USB specific interrupts are defined: The IRQ levels, the prototype and the `#pragma intvect` section.

Figure 4-1: IRQ levels

```
void InitIrqLevels(void)
{
    volatile int irq;
    for (irq = MIN_ICR; irq <= MAX_ICR; irq++)
    {
     ICR = (irq << 8) | DEFAULT_ILM_MASK;
    }
    ICR = ( 19 << 8) | 2; //VBUS Connect Interrupt on SK-16FX-144PMC-USB
    ICR = (112 << 8) | 3; //USB EP0 IN Interrupt
    ICR = (113 << 8) | 3; //USB EP0 OUT Interrupt
    ICR = (114 << 8) | 3; //USB EP1 Interrupt
    ICR = (115 << 8) | 3; //USB EP2 Interrupt
    ICR = (116 << 8) | 3; //USB EP3 Interrupt
    ICR = (117 << 8) | 3; //USB EP4 Interrupt
    ICR = (118 << 8) | 3; //USB EP5 Interrupt
    ICR = (119 << 8) | 3; //USB Function Flags 1
    ICR = (120 << 8) | 3; //USB Function Flags 2
}
```

Prototype sections:

Figure 4-2: ISR prototypes

```
__interrupt void DefaultIRQHandler (void);
__interrupt void UsbFunction_EndpointIsr(void);
__interrupt void UsbFunction_StatusControlIsr(void);
__interrupt void UsbFunction_VbusIsr(void);
```

Pragma intvect sections:

Figure 4-3: #pragma intvect section

```
#pragma intvect UsbFunction_VbusIsr           19  /* EXT2                          */
#pragma intvect UsbFunction_StatusControlIsr 112  /* USB EP0 IN  (only MB9633xU)   */
#pragma intvect UsbFunction_StatusControlIsr 113  /* USB EP0 OUT (only MB9633xU)   */
#pragma intvect UsbFunction_EndpointIsr      114  /* USB EP1     (only MB9633xU)   */
#pragma intvect UsbFunction_EndpointIsr      115  /* USB EP2     (only MB9633xU)   */
#pragma intvect UsbFunction_EndpointIsr      116  /* USB EP3     (only MB9633xU)   */
#pragma intvect UsbFunction_EndpointIsr      117  /* USB EP4     (only MB9633xU)   */
#pragma intvect UsbFunction_EndpointIsr      118  /* USB EP5     (only MB9633xU)   */
#pragma intvect UsbFunction_StatusControlIsr 119  /* USB Function Flag 1 (only MB9633x
U) */
#pragma intvect UsbFunction_StatusControlIsr 120  /* USB Function Flag 2 (only MB9633x
U) */
```

4.1.3 Typedefinitions

4.1.3.1 usb_request_t

Is used to transfer a USB request to the vendor class. The Standard-Device-Requests are defined in the USB Specification 2.0. A Request contains 8 bytes:

Figure 4-4: USB Device Requests (see USB Specification 2.0, chapter 9.3)

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request:
				Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Definition

```
typedef struct stc_usb_request
{
    uint8_t Request;
    uint8_t Direction;
    uint8_t Type;
    uint8_t Target;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
    uint8_t* u8Data;
} usb_request_t;
```

Values

Request	(see Figure 4-4 *bRequest*)
Direction	data direction (0 = from host, 1 = from device)
Type	request type (see Figure 4-4 *bmRequestType*)
Target	receiver of the request (see Figure 4-4 *bmRequestType* - Recipient)
wValue	see USB Specification 2.0, chapter 9.3
wIndex	see USB Specification 2.0, chapter 9.3
wLength	see USB Specification 2.0, chapter 9.3
u8Data	direct access to the request data

4.1.3.2 usb_buffer_t

Is used to setup the different buffers

Definition

```
typedef struct stc_usb_buffer
{
    puint8_t pu8Buffer;
    uint32_t u32BufferSize;
    uint32_t u32Position;
    uint32_t u32DataLength;
} usb_buffer_t;
```

Values

pu8Buffer	pointer to uint8_t buffer
u32BufferSize	byte size of the buffer
u32Position	buffer is currently processed at this position
u32DataLength	length of data stored in this buffer

4.1.3.3 usb_event_t

Is used to transfer an event

Definition

```
typedef struct stc_usb_event
{
    uint8_t u8UsbStatus;
    uint8_t u8Event;
    usb_request_t * Request;
    usb_buffer_t * pstcEndpointBuffer;
    uint32_t u32DataSize;
    uint8_t u8Endpoint;
} usb_event_t;
```

Values

u8UsbStatus	the current USB status
u8Event	kind of event: USB_EVENT_CONNECT USB_EVENT_DISCONNECT USB_EVENT_RECEIVE USB_EVENT_SENT USB_EVENT_CLASSREQUEST USB_EVENT_CONFIGURED USB_EVENT_DECONFIGURED
Request	if available the vendor class USB request
pstcEndpointBuffer	if available the pointer to the data buffer for this event
u32DataSize	if available the data size for this event
u8Endpoint	if available the endpoint for this event

4.1.3.4 dma_handler_t

Is used to setup the different buffers

Definition

```
typedef struct stc_dma_handler
{
   uint8_t u8DmaNumber;
   volatile uint8_t * DISELn;
   volatile uint8_t * pu8DmaDescriptor;
} dma_handler_t;
```

Values

u8DmaNumber	DMA channel
DISELn	IO Address of the DMA interrupt select register DISEL of this DMA channel
pu8DmaDescriptor	IO Address of the DmaDescriptor

4.1.3.5 usb_EP_t

Is used to setup the different buffers

Definition

```
typedef struct stc_usb_EP {
    uint8_t status;
    volatile uint8_t * EPnDTH;
    volatile uint8_t * EPnDTL;
    volatile uint16_t * EPnS;
    volatile uint16_t * EPnC;
    uint16_t u16Size;
    usb_buffer_t * pstcEndpointBuffer;
    uint8_t (* handler)(usb_event_t * stcEvent);
    dma_handler_t stcDmaHandler;
} usb_EP_t;
```

Values

status	endpoint status (bitmask) EP_STATUS_IDLE EP_STATUS_CONFIGURED EP_STATUS_DISABLED EP_STATUS_ENABLED EP_STATUS_RECEIVING EP_STATUS_SENDING EP_STATUS_DMA_TRANSFER EP_STATUS_DMA_ENABLED
EPnDTH	endpoint data high 8-bit register address
EPnDTL	endpoint data low 8-bit register address
EPnS	endpoint status 8-bit register address
EPnC	endpoint control 8-bit register address
u16Size	endpoint FIFO size
pstcEndpointBuffer	endpoint specific buffer handler
stcEvent	endpoint specific handler (called for sent/received)
stcDmaHandler	endpoint specific DMA handler

4.1.4 API Functions

4.1.4.1 UsbFunction_GetStatus

Returns the status of the USB Function device

Definition

```
uint8_t UsbFunction_GetStatus(void)
```

Return Value

Returns the current status as bitmask:

USB_STATUS_CONNECTED	device is physically connected
USB_STATUS_CONFIGURED	the device is configured
USB_STATUS_DEVDESC	the device descriptor was transfered
USB_STATUS_CONFDESC	the config descriptor was transferred

4.1.4.2 UsbFunction_GetEndpointStatus

Returns the status of the specified endpoint device

Definition

```
uint8_t UsbFunction_GetEndpointStatus(uint8_t u8EndpointNumber)
```

Parameter

u8EndpointNumber

Endpoint 0..5

Return Value

Returns the current status of the specified endpoint as bitmask:

EP_STATUS_IDLE	endpoint is not configured, returned value == 0
EP_STATUS_CONFIGURED	the endpoint was configured but not enabled
EP_STATUS_DISABLED	the endpoint was explicit disabled
EP_STATUS_ENABLED	the endpoint is enabled
EP_STATUS_RECEIVING	the endpoint still receives data
EP_STATUS_SENDING	the endpoint still sends data
EP_STATUS_DMA_TRANSFER	a DMA transfer is ongoing
EP_STATUS_DMA_ENABLED	next receiving data for this session will be DMA

4.1.4.3 UsbFunction_Initialize

Initalizes the external interrupt, which starts or stops the USB function.

Definition

```
void UsbFunction_Initialize(Boolean_t bAutoConnect);
```

Parameter

bAutoConnect

TRUE for automatically connect/disconnect via VBUS ISR
FALSE for manually connect/disconnect

Comments

On attach device, internally the function `ConnectEvent` is called. At removing the device, the internal `DisconnectDevice` function is called.

See Also

`UsbFunction_SetEventHandler`

4.1.4.4 UsbFunction_Connect

Is used to manually connect the USB function if it is physically connected. If the connection fails, FALSE will be returned.

Definition

```
Boolean_t UsbFunction_Connect(Boolean_t bAutoConnect)
```

Parameter

bAutoConnect

Is used, if automatically connect via VBUS ISR shall be activated later.

Return Value

TRUE

If the device is connected to a computer, TRUE will be returned. In case of the argument bAutoConnect is TRUE, automatically TRUE is returned, because the initialisation of the VBUS ISR was successful.

FALSE

If the device was not successful connected, FALSE will be returned. In case of the argument bAutoConnect is TRUE, automatically TRUE is returned, because the initialisation of the VBUS ISR was successful.

Comments

In case of the argument bAutoConnect is TRUE, automatically TRUE is returned, because the initialisation of the VBUS ISR was successful.

4.1.4.5 UsbFunction_Disconnect

Is used to manually disconnect the USB function if it is physically connected. If the disconnection fails, FALSE will be returned. This will also disable the automatically connect via VBUS ISR! Normally this is not a feature of USB, but while using host and function device in one application, the USB function hardware abstraction layer must be able to disable.

Definition

```
Boolean_t UsbFunction_Disconnect(Boolean_t bForceDisconnect)
```

Parameter

bForceDisconnect

Can be used to force disconnecting. If bForceDisconnect is FALSE, the function will wait at 48MHz about 1 second if endpoints are still in use. If the endpoints are still in use over this time, the disconnect will be given up and FALSE will be returned.

Return Value

TRUE

If the device was successful disconnected, TRUE will be returned. In case bForceDisconnect was specified, always TRUE will be returned after a forced disconnect!

FALSE

If bForceDisconnect is FALSE, the function will wait at 48MHz about 1 second if endpoints are still in use. If the endpoints are still in use over this time, the disconnect will be given up and FALSE will be returned.

4.1.4.6 UsbFunction_SendData

Used to send polled, interrupt or DMA controlled data.

Definition

```
void UsbFunction_SendData(
        uint8_t u8EndpointNumber
        uint6_t* buffer
        uint32_t buffersize
        uint8_t u8PollingMode
);
```

Parameter

u8EndpointNumber

Endpoint for sending data (1..5)

buffer

Data buffer

buffersize

Size of buffer

u8PollingMode

Modes of sending data:

`USB_SENDING_MODE_INTERRUPT` transfer data internally with ISR

`USB_SENDING_MODE_POLLED` transfer data internally polled

`USB_SENDING_MODE_DMA` transfer data internally via DMA

4.1.4.7 UsbFunction_SetCustomEndpointBuffer

Is used to setup own buffers for use with an endpoint, in which data can be received.

Definition

```
void UsbFunction_SetCustomEndpointBuffer(
        uint8_t u8Endpointbuffer,
        usb_buffer_t* pstcEndpointBuffer
);
```

Parameter

u8Endpointbuffer

One of 5 endpoints: 1..5

pstcEndpointBuffer

A custom buffer of type `usb_buffer_t`

4.1.4.8 UsbFunction_EnableSOF

Is used to enable the Start Of Frame interrupt called every 1ms from the HOST.

Definition

```
void UsbFunction_EnableSOF(void);
```

4.1.4.9 UsbFunction_DisableSOF

Is used to disable the Start Of Frame interrupt called every 1ms (for Full-Seed) from the HOST.

Definition

```
void UsbFunction_DisableSOF(void);
```

4.1.4.10 UsbFunction_SetStartOfFrameHandler

Is used to for react on the Start Of Frame interrupt.

Definition

```
void UsbFunction_SetStartOfFrameHandler(
      void(* handler)(uint16_t u16FrameNumber),
);
```

Parameter

handler

is used to set the event handler of type:

```
handler(uint16_t u16FrameNumber);
```

4.1.4.11 UsbFunction_ClearStartOfFrameHandler

Is used to clear the event handler

Definition

void UsbFunction_SetStartOfFrameHandler(void);

4.1.4.12 UsbFunction_EnableWakeup

Used to wake-up the computer from suspend mode.

Definition

```
UsbFunction_EnableWakeup(
        Boolean_t bEnableDisable,
);
```

Parameter

bEnableDisable

TRUE: enables the resume flag

FALSE: disables the resume flag

Note

To wakeup the computer from suspend mode, the resume flag has to be toggled.

Example

```
void WakeUpPC(void)
{
    UsbFunction_EnableWakeup(FALSE);
    UsbFunction_EnableWakeup(TRUE);
    UsbFunction_EnableWakeup(FALSE);
}
```

4.1.4.13 UsbFunction_SetDataToSendEndpoint0

Used from USB vendor class to answer USB requests.

Definition

```
UsbFunction_SetDataToSendEndoint0(
        uint8_t* buffer,
        uint32_t size
);
```

Parameter

buffer

buffer to send

size

size of buffer to send

4.1.4.14 UsbFunction_SetEventHandler

Used to get events directly from different interrupts.

Definition

```
void UsbFunction_SetEventhandler(
        uint8_t u8Target,
        uint8_t u8Event,
        uint8_t u8Options,
        uint8_t (* handler)(usb_event_t * stcEvent)
);
```

Parameters

u8Target

Target (Application, USB Class, General, Interface)

USB_EVENT_TARGET_GENERAL	before connect / disconnect
USB_EVENT_TARGET_MAINAPP	status change
USB_EVENT_TARGET_VENDOR	Class Requests and status change
USB_EVENT_TARGET_INTERFACE	not used currently

u8Event

USB_EVENT_CONNECT	USB Connect Event
USB_EVENT_DISCONNECT	USB Disconnect Event
USB_EVENT_RECEIVE	Endpoint 1..5 Data Received Event
USB_EVENT_SENT	Endpoint 1..5 Data Sent Event
USB_EVENT_CLASSREQUEST	USB Connect Event
USB_EVENT_CONFIGURED	USB Configured Event
USB_EVENT_DECONFIGURED	USB Configuration Cleared Event

u8Options

In case of use with endpoint events for receiving or sending, the options parameter represents the endpoint (1..5).

handler

The Event handler is called from interrupt.

Comments

Normally all data receive or sent events are handled in the USB Class. USB requests and USB status events are separated from receive and sent events. For each endpoint an event handler exists for received data or sent data.

For events before connection or disconnection, the general event handler is used.

4.2 .NET UsbLibrary

Originally the library was written in C#. It can be compiled as .NET framework library, too. It can be included in every .NET framework ready programming language like Borland Delphi, Visual C#, Visual Basic and Visual C++.

4.2.1 UsbFunctions Class

Offers a simple API in combination with LibUsbDotNet and LibUSB to have easy and stable to USB transfers. The library adds Hot-Plug capabilities and initializes automatically all endpoints, which were configured during the enumeration process. The library helps to add USB features to own .NET projects with LibUsbDotNet.

4.2.1.1 Event: UsbFunctions.DeviceConnected

Is called after a device was connected.

Definition

public event EventHandler<ConnectEvent> **DeviceConnected**

4.2.1.2 Event: UsbFunctions.DeviceDisconnected

Is called after a device was disconnected.

Definition

```
public event EventHandler<DisconnectEvent> DeviceDisconnected
```

4.2.1.3 Event: UsbFunctions.OnDeviceNotify

Is called after a device status was changed.

Definition

```
public event EventHandler<DeviceNotifyEventArgs> OnDeviceNotify
```

4.2.1.4 Field: UsbFunctions.DebugEnabled

Used to display debug information via console. Can be disabled for better performance.

Definition

```
public bool DebugEnabled
```

4.2.1.5 Field: UsbFunctions.UsbDevice

Contains the `LibUsbDotNet.UsbDevice` object used by the UsbFunctions class. This field should be only read and not written.

Definition

```
public UsbDevice Device
```

4.2.1.6 Field: UsbFunctions.DisableAutoconnect

Can be used to disable the autoconnect feature.

Definition

```
public bool DisableAutoconnect
```

4.2.1.7 Field: UsbFunctions.ReadEndpoints

Contains a sorted list with endpoint number as key and the `EndPointIN` class handling this endpoint number as value.

Definition

```
public SortedList<int, EndPointIN> ReadEndpoints
```

4.2.1.8 Field: UsbFunctions.WriteEndpoints

Contains a sorted list with endpoint number as key and the `EndPointOUT` class handling this endpoint number as value.

Definition

```
public SortedList<int, EndPointOUT> WriteEndpoints
```

4.2.1.9 Method: UsbFunctions.OpenUsbDevice

Definition

```
public bool OpenUsbDevice(
    short VendorID,
    short ProductID,
    int ReadBufferSize
);
```

Parameters

VendorID

Type: System.Int16

Vendor ID

ProductID

Type: System.Int16

Product ID

ReadBufferSize

Type: System.Int32

Maximum Read Buffer Size

Return Value

TRUE if it was successful opened

4.2.1.10 Method: UsbFunctions.CloseUsbDevice

Close the UsbDevice.

Definition

Public Sub **CloseUsbDevice**();

4.2.1.11 Method: UsbFunctions.IsOpen

Definition

public bool **IsOpen**()

Parameters

Return Value

TRUE if it device is open

4.2.1.12 Method: UsbFunctions.GetReadEndpoint

Definition

```
public EndPointIN GetReadEndpoint(
      int EpNo
)
```

Parameters

EpNo

Type: `System.Int32`

Number of endpoint: 1..5

Return Value

The endpoint handler

4.2.2 EndPointIN Class

The `EndpointIN` Class adds some features to the LibUsbDotNet own Endpoint Classes, for example Hot-Plug and FIFO buffers as data packages. An `EndPointIN` class object will be created for each found endpoint with direction IN in the UsbFunctions Class.

4.2.2.1 Event: EndpointIN.DataReceived

Is called after data was received. If the event is set, it also disables the received data queue!

Definition

```
public event EventHandler<DataReceivedEvent> DataReceived
```

4.2.2.2 Field: EndpointIN.BytesReceived

Contains the number of bytes received by this endpoint. This is only working if the `DataReceived` event was not set. Otherwise data will be handled by event!

Definition

```
public int BytesReceived
```

4.2.2.3 Field: EndpointIN.Device

Contains the `LibUsbDotNet.UsbDevice` object used by the endpoint class. This field should be only read and not written.

Definition

```
public UsbDevice Device
```

4.2.2.4 Field: EndpointIN.EndpointID

Contains the `LibUsbDotNet.ReadEndpointID` object used by the endpoint class. This field should be only read and not written.

Definition

```
public ReadEndpointID EndpointID
```

4.2.2.5 Field: EndpointIN.ReadBufferSize

Sets the maximum buffer size.

Definition

```
public int ReadBufferSize
```

4.2.2.6 Field: EndpointIN.Reader

Contains the `LibUsbDotNet.UsbEndpointReader` object used by the endpoint class. This field should be only read and not written.

Definition

```
public UsbEndpointReader Reader
```

4.2.2.7 Method: EndPointIN.Close

Close this endpoint.

Definition

```
public void Close()
```

4.2.2.8 Method: EndPointIN.Enable

Enables the receiver.

Definition

```
public void Enable()
```

4.2.2.9 Method: EndPointIN.FlushReceiveBuffer

Clear the whole internal receiving buffer

Definition

```
public void FlushReceiveBuffer()
```

4.2.2.10 Method: EndPointIN.GetReadPackage

Get a reception data package

Definition

```
public byte[] GetReadPackage()
```

Return Value

one package as byte array

4.2.2.11 Method: EndPointIN.GetReceivedEventHandler

Get Event handler for receiving data externally

Definition

```
public EventHandler<EndpointDataEventArgs>
                    GetReceivedEventHandler()
```

Return Value

Event handler for receiving data externally

4.2.2.12 Method: EndPointIN.ReInitialize

Used to re-initialize this endpoint, after the UsbDevice was changed. This should normally be done automatically.

Definition

```
public void ReInitialize(
        UsbDevice device
)
```

Parameters

device

Type: UsbDevice (see also LibUsbDotNet.Main)

Current Opened UsbDevice

4.2.3 EndPointOUT Class

The `EndpointOUT` Class adds some features to the LibUsbDotNet own Endpoint Classes, for example Hot-Plug. An `EndPointOUT` class object will be created for each found endpoint with direction OUT in the UsbFunctions Class.

4.2.3.1 Field: EndpointOUT.Device

Contains the `LibUsbDotNet.UsbDevice` object used by the endpoint class. This field should be only read and not written.

Definition

```
public UsbDevice Device
```

4.2.3.2 Field: EndpointOUT.EndpointID

Contains the `LibUsbDotNet.WriteEndpointID` object used by the endpoint class. This field should be only read and not written.

Definition

```
public WriteEndpointID EndpointID
```

4.2.3.3 Field: EndpointOUT.UsbFunctionsMain

Contains the `UsbFunctions` object used by the endpoint class. This field should be only read and not written.

Definition

```
public UsbFunctions UsbFunctionsMain
```

4.2.3.4 Field: EndpointOUT.Writer

Contains the `LibUsbDotNet.UsbEndpointWriter` object used by the endpoint class. This field should be only read and not written.

Definition

```
public UsbEndpointWriter Writer
```

4.2.3.5 Method: EndPointOut.ReInintialize

Used to reinitialize this endpoint, after the UsbDevice was changed. This should normally be done automatically by the UsbFunctions class.

Definition

```
public void ReInitialize(
        UsbDevice device
)
```

Parameters

device

Type: `UsbDevice` (see also `LibUsbDotNet.Main`)

Current Opened `UsbDevice`

4.2.3.6 Method: EndPointOut.Write

Used to write data to this endpoint.

Definition

```
public bool Write(
        byte[] dataout,
        int timeout,
        out int byteswritten
)
```

Parameters

dataout

Type: `System.Byte[]`

Data array

timeout

Type: `System.Int32`

timeout in ms

byteswritten

Type: `System.Int32`

return number of bytes

Return Value

returns TRUE, if the process was successful initiated

4.2.4 UsbHandShake Class

This class is used to setup a handshake protocol via two endpoints. One endpoint is used to send data, while the second endpoint is used to receive a handshake. This handshake has the best results by sending 2 bytes from the microcontroller containing any data. All data, which shall be sent, will be stored into a queue, which will send each package after a handshake. This handshake data is rejected after its usage and not stored in any data buffer.

4.2.4.1 Field: UsbHandShake.DataReceived

Is TRUE after data was received

Definition

```
public boot DataReceived
```

4.2.4.2 Field: UsbHandShake.PackagesToSend

Contains the package queue.

Definition

```
public Queue<byte[]> PackagesToSend
```

4.2.4.3 Field: UsbHandShake.SendingData

Is TRUE while sending data (data queue is full).

Definition

```
public boot SendingData
```

4.2.4.4 Method: UsbHandShake.Clear

Is used to clear the message queue and reset all flags and states.

Definition

```
Public void Clear();
```

4.2.4.5 Method: UsbHandShake.IsSending

Is used to get the current sending status.

Definition

```
public bool IsSending();
```

Return Value

`True` if the class is still sending data

`False` if the class is idle

4.2.4.6 Method: UsbHandShake.NumberOfPackages

Is used to get the number of packages which are in the queue.

Definition

```
public int NumberOfPackages()
```

Return Value

Returns number of packages in the queue.

4.2.4.7 Method: UsbHandShake.WriteDataUsb

Is used to add data to the package queue.

Definition

```
public void WriteDataUsb(
        byte[] Data
)
```

Parameters

Data

Type: System.Byte[]

Buffer to be sent

4.3 Examples

4.3.1 MCU USB Function Library

4.3.1.1 Initialisations

To initialize the USB functionality, the UsbClass has to be initialized first. The UsbClass sets all necessary event handlers, buffers, and request handlers. After these settings were done, the UsbFunction library can be initialized. The auto connect feature can be set with the first and only parameter. The parameter should be normally TRUE. For advanced usage, it can be FALSE to disable the auto connect feature.

Figure 4-5: Initialisation of the USB device functionality from main application

```
UsbClass_Init();
UsbFunction_Initialize(TRUE);
```

4.3.1.2 Initialisations in UsbClass.c

Normally the *UsbClass.c* module can be generated as a template via the USB Descriptors Manager. A generated sample or hand written code could look like the following example for endpoint 1 as receiving (Host OUT) and endpoint 2 as sending (Host IN). It can be seen, that all buffers are provided by the *UsbClass.c* module. To tell the UsbFunction library to write down received data into the own buffers, the UsbClass has to relocate the buffers while using the `UsbFunction_SetCustomEndpointBuffer`. Events are set with the function `UsbFunction_SetEventHandler`.

Figure 4-6: Initialisation in UsbClass

```
uint8_t DataReceivedFlags = 0;                                  // status for received
data
uint8_t DataSentFlags = 0;                                      // status for sent data
uint32_t u32LastReceivedSize1;                                  // last received size
usb_buffer_t stcCustomClassBuffer1;                             // struct for custom
buffer
uint8_t pu8CustomClassBufferEndpoint1[BUFFER_SIZE_ENDPOINT1]; // custom buffer

void UsbClass_Init(void)
{
    stcCustomClassBuffer1.u32BufferSize = BUFFER_SIZE_ENDPOINT1;     // initializes
buffer
    stcCustomClassBuffer1.pu8Buffer = pu8CustomClassBufferEndpoint1; // initializes
buffer
    UsbFunction_SetCustomEndpointBuffer(1,&stcCustomClassBuffer1);   // initializes
buffer
    UsbFunction_SetEventHandler(USB_EVENT_TARGET_VENDOR,USB_EVENT_RECEIVE,USB_EP_1,
                                UsbClass_DataReceiveEventEndpoint1); // setup received
event

    UsbFunction_SetEventHandler(USB_EVENT_TARGET_VENDOR, USB_EVENT_SENT, USB_EP_5,
                                UsbClass_DataSentEventEndpoint5);    // setup sent
event
    UsbFunction_SetEventHandler(USB_EVENT_TARGET_VENDOR,USB_EVENT_CLASSREQUEST,0,
                                UsbClass_ClassEventHandler);         // setup class
request
}
```

4.3.1.3 Sending data via UsbClass.c template

Sending Data is handled by the procedure `UsbClass_SendDataVia5` in this example. After data was sent, the event `UsbClass_DataSentEventEndpoint5` will be called. In this event a status flag will be set, which can be read by the function `UsbClass_DataSent5()`.

Figure 4-7: Sending data in UsbClass

```
/**
 ******************************************************************************
 ** Is called from main application to send data via endpoint 5
 **
 ** \param pu8Buffer Buffer to send
 ** \param u32DataSize Buffersize
 ** \param u8PollinMode 1: polled sending; 0: interrupt sending;
 **
 ** \return 1: if succesful, 0: if usb was not ready
 ******************************************************************************/
uint8_t UsbClass_SendDataVia5(uint8_t* u8Buffer, uint32_t u32DataSize, uint8_t
u8TransferMode)
{
    if ((UsbFunction_GetStatus() & USB_STATUS_CONFIGURED) > 0)
    {
        DataSentFlags &= ~(1<<USB_EP5);                                  //Clear sent flag
        UsbFunction_SendData(5,u8Buffer,u32DataSize,u8TransferMode); //Send data
        if (u8TransferMode == USB_SENDING_MODE_POLLED)
        {
            DataSentFlags |= (1<<USB_EP5); //Data was sent directly, setting sent flag
        }
        return 1;
    }
    return 0;

/**
 ******************************************************************************
 ** Is called from UsbFunctions.c when a endpoint buffer 5 was sent
 **
 ** \param stcEvent Event containing all neccesary information
 ******************************************************************************/
uint8_t UsbClass_DataSentEventEndpoint5(usb_event_t * stcEvent)
{
    DataSentFlags |= (1<<USB_EP5); //Setting data sent flag
    return 1;
}
/**
 ******************************************************************************
 ** Is used to use get the sent status of endpoint 5
 **
 ******************************************************************************/
uint8_t UsbClass_DataSent5(void)
{
    if ((DataSentFlags & (1<<USB_EP5)) > 0) //Sent flag is set
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

In the main application using `UsbClass_SendDataVia5` can send data. While using DMA or interrupts, the end of transfer can be read by the `UsbClass_DataSent5` procedure. An example application could look like the following code:

Figure 4-8: Sending data

```
char_t message[] = “hello world!”;

/* send message as polled mode */
UsbClass_SendDataVia5((uint8_t*)message,sizeof(message),USB_SENDING_MODE_POLLED);


/* send message as interrupt mode */
UsbClass_SendDataVia5((uint8_t*)message,sizeof(message),USB_SENDING_MODE_INTERRUPT);

/* wait for data sent */
while(Usb_Class_DataSent5() == 0)
{
    __wait_nop();
}


/* send message as DMA mode */
UsbClass_SendDataVia5((uint8_t*)message,sizeof(message),USB_SENDING_MODE_DMA);

/* wait for data sent */
while(Usb_Class_DataSent5() == 0)
{
    __wait_nop();
}
```

Receving data via UsbClass.c template

Data is internally received via event (`UsbClass_DataReceiveEventEndpoint1`). This is called after a package was received and a data received flag will be set. The main application can use the `UsbClass_GetReceivedDataEndpoint1` function to get size of data and pointer of the buffer.

Figure 4-9: Receiving data in UsbClass

```
/**
 ******************************************************************************
 ** Is called from UsbFunctions.c when a endpoint buffer 1 was received
 **
 ** \param stcEvent Event containing all neccesary information for receiving data
 ******************************************************************************/
uint8_t UsbClass_DataReceiveEventEndpoint1(usb_event_t * stcEvent)
{
    uint8_t* pu8Buffer = stcEvent->pstcEndpointBuffer->pu8Buffer;
    uint32_t u32DataSize = stcEvent->u32DataSize;
    u32LastReceivedSize1 = u32DataSize;
    DataReceivedFlags |= (1<<USB_EP1); // setting data received flag

    //Add your code here to process the received buffer

    return 1;
}


/**
 ******************************************************************************
 ** Is used to use received data from endpoint 1
 **
 ** \param pu8Buffer pointer to buffer
 ******************************************************************************/
uint32_t UsbClass_GetReceivedDataEndpoint1(uint8_t** pu8Buffer)
{
    if ((DataReceivedFlags & (1<<USB_EP1)) == 0)
    {
        return 0; // nothing to receive
    }
    DataReceivedFlags -= (1<<USB_EP1);
    *pu8Buffer = pu8CustomClassBufferEndpoint1;
    return u32LastReceivedSize1;
}
```

From main application reception data can be also initiated as DMA transfer. Both transfer types (DMA and interrupt) are shown in the following code example:

Figure 4-10: Receiving data

```
#define   1KB 1024
uint32_t  u32ReceivedSize = 0;
uint8_t* pu8DataBuffer;

/* wait for data was received as interrupt transfer */
while (u32ReceivedSize == 0)
{
    u32ReceivedSize = UsbClass_GetReceivedDataEndpoint1(&pu8DataBuffer);
}
// data was received


/* initiate a DMA receiving transfer */
UsbFunction_EnableReceiveDma(USE_EP_1,1KB);

/* wait for data was received as DMA transfer */
while (u32ReceivedSize == 0)
{
    u32ReceivedSize = UsbClass_GetReceivedDataEndpoint1(&pu8DataBuffer);
}
// data was received
```

4.3.1.4 Handle class events and requests

Class events and requests can be handled in the *UsbClass.c* module. If the USB Descriptors Manager was used to create a template, the handler for class events and requests is set and a template was generated for handling this. Class Events are USB status change events. Class requests are USB requests for the vendor specific application (see also USB specification). Normally USB Descriptor Manager creates the following code:

Figure 4-11: Event handling

```
/**
 ******************************************************************************
 ** Is called from UsbFunctions.c when a connect, disconnct or USB class request
occurs
 **
 ** \param stcEvent Event containing all neccesary information
 ******************************************************************************/
uint8_t UsbClass_ClassEventHandler(usb_event_t * stcEvent)
{
    switch(stcEvent->u8Event)
    {
        case USB_EVENT_CONNECT:
            // paste your connect event here
            break;
        case USB_EVENT_DISCONNECT:
            // paste your disconnect event here
            break;
        case USB_EVENT_RECEIVE:
            dbg("WARNING: The Class Event Handler should not receive any data!\n");
            break;
        case USB_EVENT_SENT:
            dbg("WARNING: The Class Event Handler should not send any data!\n");
            break;
        case USB_EVENT_CONFIGURED:
            // paste your connect event here
            break;
        case USB_EVENT_DECONFIGURED:
            // paste your connect event here
            break;
        case USB_EVENT_CLASSREQUEST:
            switch (stcEvent->Request->Request)
            {
                // decode the request here
                default:
                    break;
            }
            break;
        default:
            // not handled?
            return 0; // return error
    }
    return 1;
}
```

4.3.1.5 Manual connect / disconnect

Normally the USB function device should automatically start the connection after the VBUS connection was established. For some reasons it is required to disconnect or disable the USB device functionality. For example while using host and device mode with the same USB abstraction layer, the different modes must be switchable. The Cypress 16FX USB series are supporting USB host and device mode with one physical USB interface. But they cannot run both in parallel. USB host and function modes must be able to be initialized or de-initialized. The disconnect procedure of the USB function library de-initializes the USB function.

Figure 4-12: Connect USB function with auto connect option

```
/* Connect USB with auto connect feature */
UsbFunction_Connect(TRUE);
```

Figure 4-13: Connect USB function with auto connect option disabled

```
/* Connect USB without auto connect feature */
UsbFunction_Connect(FALSE);
```

Figure 4-14: Disconnect USB function with forcing disconnect

```
/* Disconnect USB with forcing disconnect feature */
UsbFunction_Disconnect(TRUE);
```

Figure 4-15: Disconnect USB function with trial of disconnecting

```
/* Disconnect USB without forcing disconnect feature */
UsbFunction_Disconnect(FLASE);
```

4.3.2 NET UsbLibrary namespace

4.3.2.1 Initialisation

Before any communication can be used, the UsbFunctions library has to been added in the references list. The following C# example gives an example of how to initialize a USB device, which will be auto connected (auto disconnected). The auto connection or disconnection feature is part of the hot-plug (hot-plug out) difficulty and makes it easy to use for the programmer.

Figure 4-16: Initialisation of USB with Visual Studio C#

```
public UsbFunctions USB;
private void Form_Load(object sender,EventArgs e)
{
    short IdVendor = 0x1F55;
    short IdProduct = 0x000F;
    UInt32 MaximumReceivingSize = 1024;
    USB = new UsbFunctions(IdVendor,IdProduct,MaximumReceivingSize);
}
```

4.3.2.2 Sending data

For sending data, the sending endpoint has to be chosen. This is done with the method `UsbFunctions.GetWriteEndpoint(<endpointnumber>)`. The following C# example shows how to send some bytes via a button click.

Figure 4-17: Sending data with Visual Studio C#

```
private void Button_Click(object sender,EventArgs e)
{
    int BytesWritten = 0;
    int Timeout = 1000;    // 1000 ms timeout
    byte[] Data = {1,2,3,4,5};
    bool Result = USB.GetWriteEndpoint(1).Write(Data,Timeout, out BytesWritten);
}
```

4.3.2.3 Receiving data

Data is internally received event driven. This data is added into a reception data queue. Before a package can be received from this queue, it has to be checked for data was actually received. If data was received, this data can be read out by the main application.

Figure 4-18: Receiving data with Visual Studio C#

```
private void Button_Click(object sender,EventArgs e)
{
    int BytesReceived = 0;
    BytesReceived = USB.GetReadEndpoint(5).BytesReceived;
    If (BytesReceived > 0)
    {
        byte[] Data = USB.GetReadEndpoint(5).GetReadPackage();
    }
}
```

4.3.2.4 Receiving data via event

Data can also be received as an event. If the event method way is used, data won't be stored in a data queue. For an event two sections will be needed: An event method and the event initializing. The following code example adds the `DataReceived` event to the initializations. To reinitialize the `DataReceived` event on device connection, a `ConnectEvent` has to be used with the event handlers to be reloaded.

Figure 4-19: Receiving data via event: initialisations

```
public UsbFunctions USB;
private void Form_Load(object sender,EventArgs e)
{
    short IdVendor = 0x1F55;
    short IdProduct = 0x000F;
    UInt32 MaximumReceivingSize = 1024;
    USB = new UsbFunctions(IdVendor,IdProduct,MaximumReceivingSize);

    USB.GetReadEndpoint(5).DataReceived += (Form_UsbEndpointReceived5);
    USB.DeviceConnected += (Form_UsbDeviceConnected);
}

private void Form_UsbDeviceConnected(object sender,ConnectEvent e)
{
    USB.GetReadEndpoint(5).DataReceived -= (Form_UsbEndpointReceived5);
    USB.GetReadEndpoint(5).DataReceived += (Form_UsbEndpointReceived5);
}
```

The method can look like the following C# code:

Figure 4-20: Receiving data via event: event method

```
private void Form_UsbEndpointReceived5(object sender,DataReceivedEvent e)
{
    byte[] Data = e.Buffer;
}
```

4.3.2.5 Connect event

For an event, two methods will be needed: An event method and the event initializing. The following code example adds the `DeviceConnected` event to the initializations:

Figure 4-21: Connect event: initialisations

```
public UsbFunctions USB;
private void Form_Load(object sender,EventArgs e)
{
    short IdVendor = 0x1F55;
    short IdProduct = 0x000F;
    UInt32 MaximumReceivingSize = 1024;
    USB = new UsbFunctions(IdVendor,IdProduct,MaximumReceivingSize);

    USB.DeviceConnected += (Form_UsbDeviceConnected);
}
```

The method can look like the following C# code:

Figure 4-22: Connect event: event method

```
private void Form_UsbDeviceConnected(object sender,DataReceivedEvent e)
{
    short VendorID = e.VendorID;
    short ProductID = e.ProductID;
    UsbFunctions USB = e.Functions;  //static USB
}
```

4.3.2.6 Disconnect event

For an event, two methods will be needed: An event method and the event initializing. The following code example adds the `DeviceDisconnected` event to the initializations:

Figure 4-23: Disconnect event: initialisations

```
public UsbFunctions USB;
private void Form_Load(object sender,EventArgs e)
{
    short IdVendor = 0x1F55;
    short IdProduct = 0x000F;
    UInt32 MaximumReceivingSize = 1024;
    USB = new UsbFunctions(IdVendor,IdProduct,MaximumReceivingSize);

    USB.GetReadEndpoint(5).DeviceDisconnected += (Form_UsbDeviceDisconnected);
}
```

The method can look like the following C# code:

Figure 4-24: Disconnect event: event method

```
private void Form_UsbDeviceDisconnected(object sender,DataReceivedEvent e)
{
    short VendorID = e.VendorID;
    short ProductID = e.ProductID;
    UsbFunctions USB = e.Functions;  //static USB
}
```

4.3.2.7 Handshaking

To use the `UsbHandshake` class, the class has to be initialized with the used endpoints.

Figure 4-25: Initialize UsbHandshake class

```
public UsbFunctions USB;
public UsbHandshake UsbHshk;
private void Form_Load(object sender,EventArgs e)
{
    short IdVendor = 0x1F55;
    short IdProduct = 0x000F;
    UInt32 MaximumReceivingSize = 1024;
    USB = new UsbFunctions(IdVendor,IdProduct,MaximumReceivingSize);
    UsbHshk = new UsbHandshake(USB,1,5);  //uses endpoint 1 and 5 for handshaking
}
```

The following method will send the data packages `DataA`, `DataB` and `DataC` via endpoint 1 and will wait for a handshaking (any 2 bytes from endpoint 5) before sending the next package.

Figure 4-26: Adding packages to the handshake queue

```
private void Button_Click(object sender,EventArgs e)
{
    byte[] DataA = {1,2,3,4,5};
    byte[] DataB = {5,4,3,2,1};
    byte[] DataC = {A,B,C,D,E};
    UsbHshk.WriteDataUsb(DataA);
    UsbHshk.WriteDataUsb(DataB);
    UsbHshk.WriteDataUsb(DataC);
}
```

A Appendix

A.1 List of litrature

Jan Axelson:

USB Complete - Everything You Need to Develop Custom USB Peripherals

Jan Axelson – USB Central

janaxelson.com/usb.htm

USB Implementers Forum (www.usb.org)

USB 2.0 Specification (http://www.usb.org/developers/docs)

Cypress Semiconductor

MB96300 Super Series Hardware Manual
MB96330 Series Microcontroller Datasheet

LibUsb

http://www.libusb.org

LibUsbDotNet

http://libusbdotnet.sourceforge.net

Thesycon

FUFA User Manual

USB in a NutShell

http://www.beyondlogic.org/usbnutshell/

USB made simple

http://www.usbmadesimple.co.uk/

A.2 Information on the Web

For more information about Cypress products, visit to the following website:

http://www.cypress.com/cypress-microcontrollers

The software examples related to this application note is:

mcu-an-300251-e-v10-16fx_usb_function_library

96330_usb_function_oscilloscope

96330_usb_function_audioplayer

96330_usb_function_template

Document History

Document Title: AN204832 – F²MC-16FX Family MB96330 Series 16-Bit Microcontroller USB Function Library

Document Number: 002-04832

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MSCH	04/14/2010	Original version
			04/15/2010	Flow Charts added
*A	5242922	MSCH	06/24/2016	Migrated Spansion Application Note MCU-AN-300251-E-V1.1 to Cypress format. Document obsoleted.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Lighting & Power Control	cypress.com/powerpsoc
Memory	cypress.com/memory
PSoC	cypress.com/psoc
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless/RF	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com