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THIS SPEC IS OBSOLETE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: 002-05487

Spec Title: AN205487 - 16FX/FR80/FM3 USB FAMILY 16/32-BIT MICROCONTROLLER

Replaced by: None

AN205487

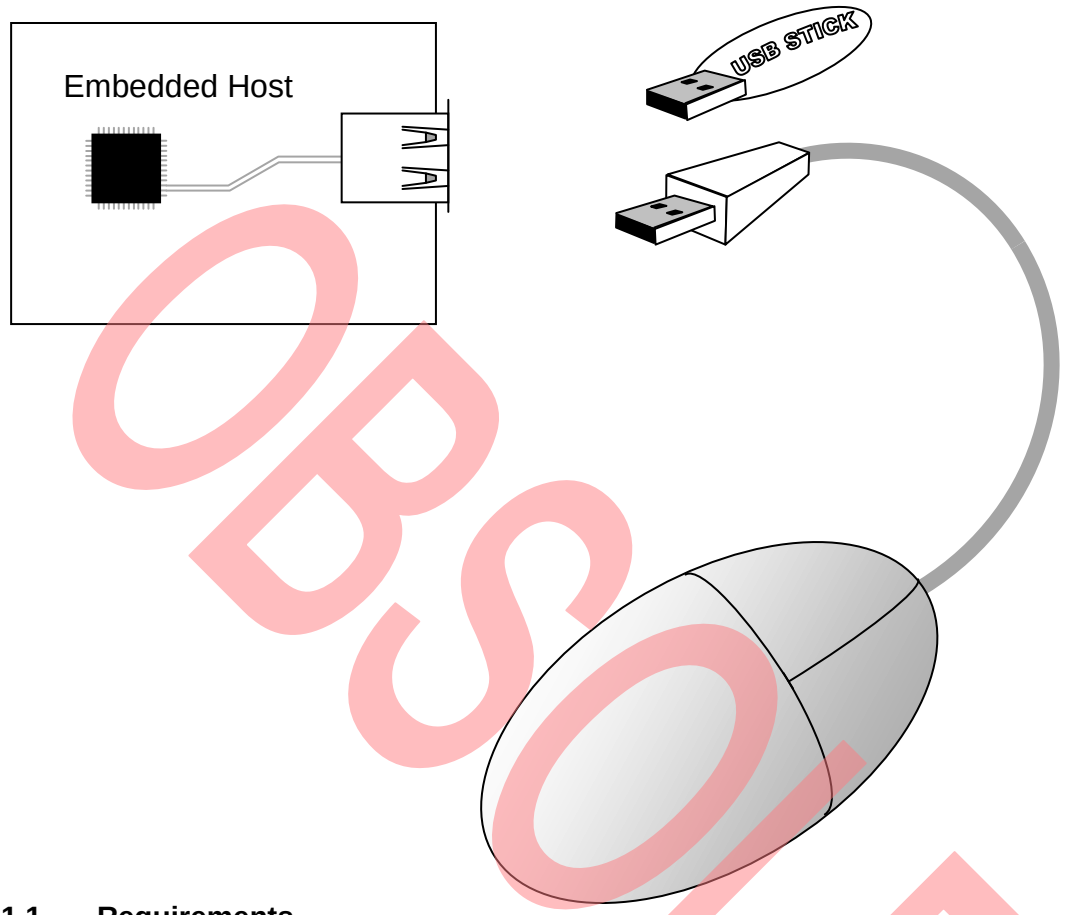
16FX/FR80/FM3 USB Family 16/32-Bit Microcontroller

The embedded USB host can be seen as a slim version of a computer USB host controller.USB host usage is reasonable for temporarily installed devices which need hot-plug and bus-powering. Also for devices which have only USB, it is a good reason to have an embedded USB host on chip.

Contents

1 Introduction

The embedded USB host can be seen as a slim version of a computer USB host controller. The hardware abstraction layer does only support as maximum full-speed and only one device while most computers supporting high-speed and USB hubs. This limitation makes sense for embedded one chip microcontrollers with clock frequencies about 24 to 200MHz. USB host functionalities in one chip embedded systems are mostly used for USB mass storage like USB flash memory sticks. Benefits are common data storages which can be used with the computer and the embedded system. Example applications are firmware update, configuration loading from USB stick, data logging, etc. But also other USB devices like keyboard, mouse and printer are of interest: For example barcode scanners, thermal printers, keyboards or touch matrices. Most of these devices also exist as non USB versions, which are much easier to implement in embedded one chip solutions. USB host usage is reasonable for temporarily installed devices which need hot-plug and bus-powering. Also for devices which have only USB, it is a good reason to have an embedded USB host on chip.



1.1 Requirements

For 16FX, FR80 and FM3 the USB driver lower than version 2.0 can be used. For Version 2.0 and greater there is only support for FM3 series and higher.

The whole USB Stack is included in the Fujitsu USB Assistant (for 16FX, FR80 and FM3) and Fujitsu USB Wizard (for FM3 and higher).

USB Stack < 2.0:

Supported Architectures:

16FX

FR80

FM3

Fujitsu USB Assistant: USB Configuration Tool

USB Stack >= 2.0

Supported Architectures:

FM3

Fujitsu USB Wizard: USB Configuration Tool

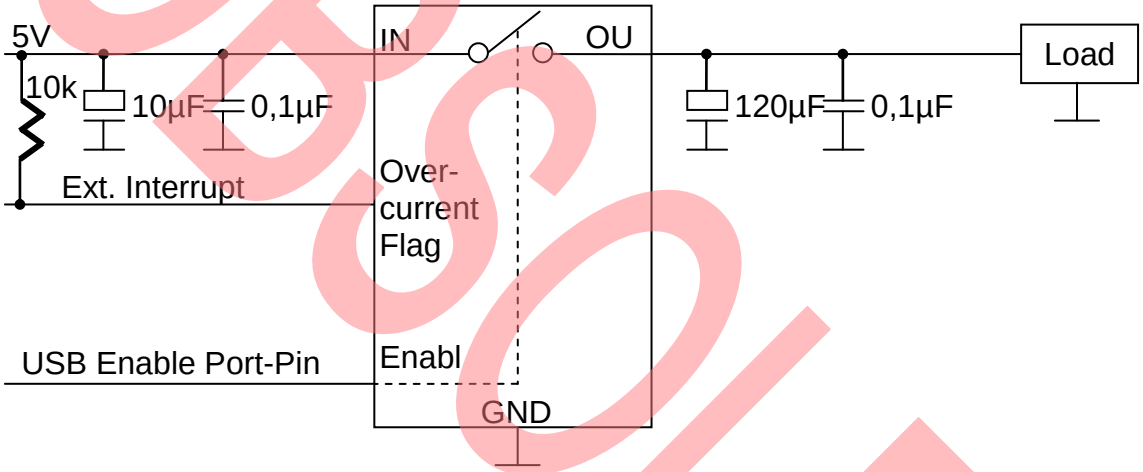
1.2 Electrical Connection

This chapter will explain different connection types and required components for USB host functionality.

1.2.1 VBUS Power Switch and Current-Overflow Detection

There are different ways to connect a USB device to the embedded USB host. For all of them an optional VBUS power switch and Over-Current protection is needed. There are different ICs on the market which handle this functionality. Diodes Inc.™ for example provides the AP2141 chip for example. The switch is used to enable the VBUS line to power a bus-powered device. As specified in the USB specification a device can use a maximum of 500mA current. If the device will need more current or the VBUS line will be short cut, the IC sets the over current flag. The capacitors on the ingoing and outgoing pins are used to stabilize the VBUS line. On the outgoing side a capacitor of about min. 120µF is needed.

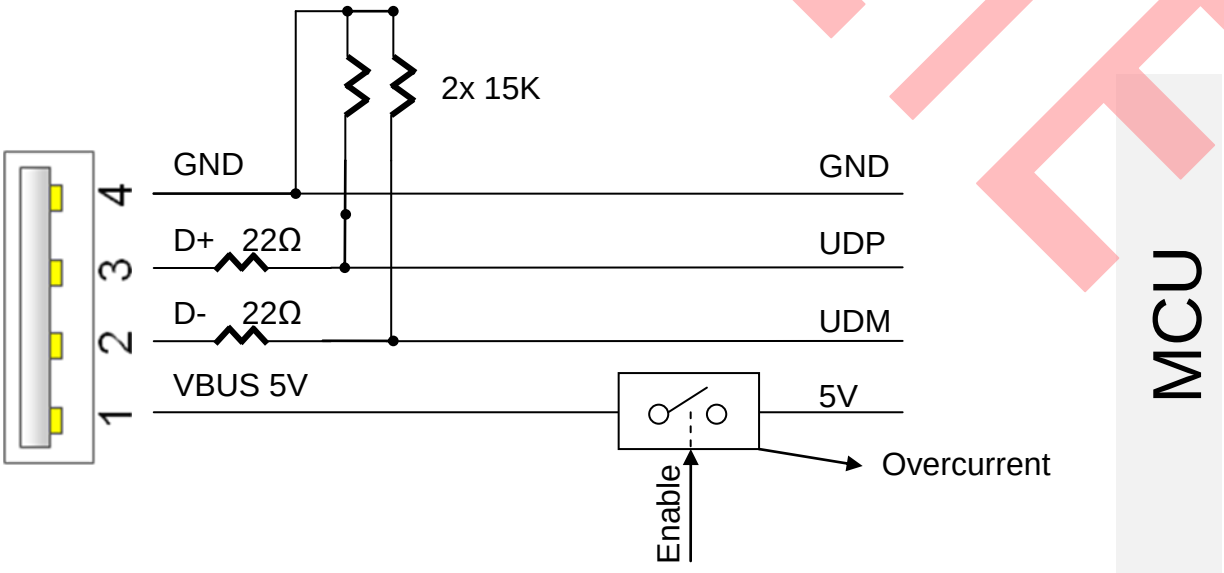
Figure 1. powering with Over-Current detection



1.2.2 Minimal Components for Host only Connection

As electrical connection the USB host normally uses the USB A Connector (female USB). Following schematic shows the minimal components:

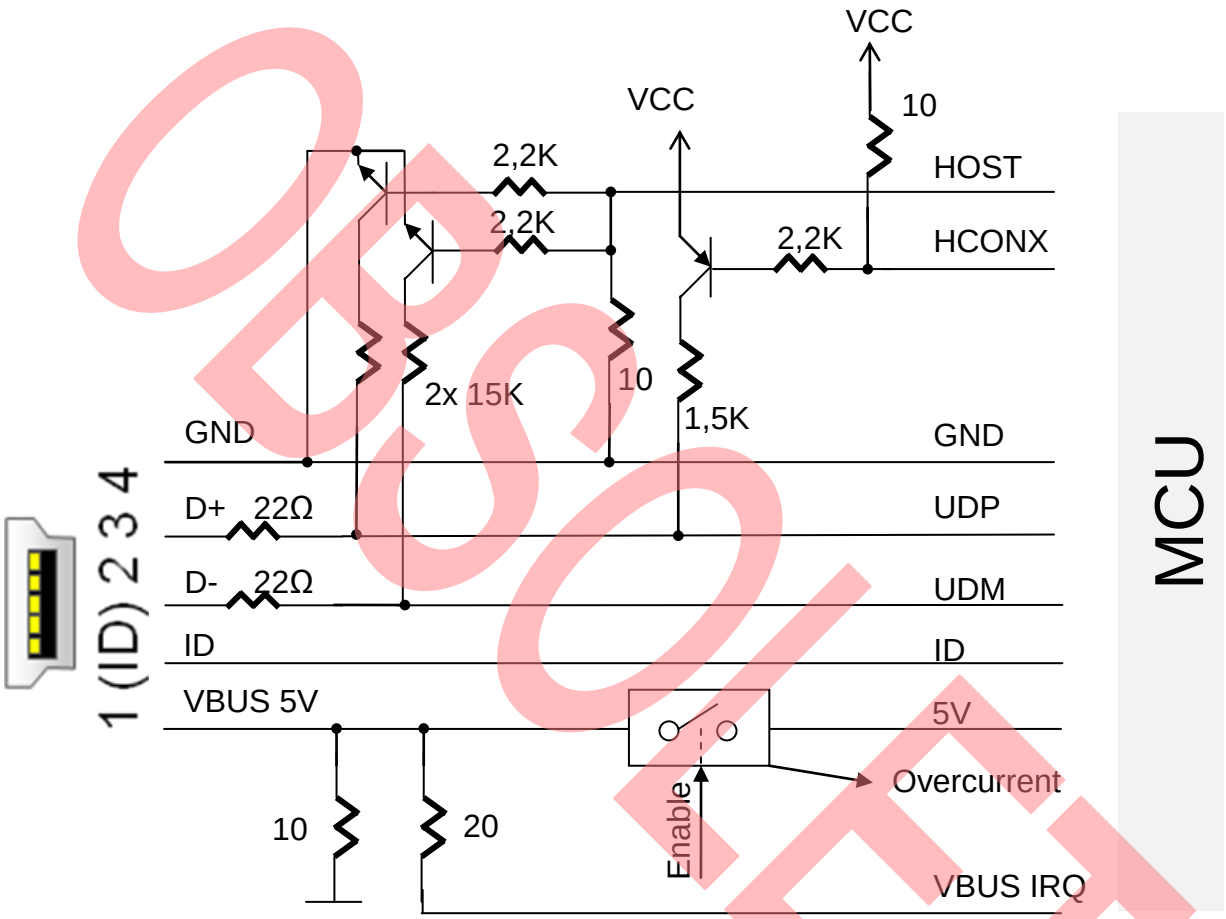
Figure 2. USB B connector - host schematic



1.2.3 USB On-The-Go

For USB On-The-Go the Mini-USB connector or Micro-USB connector can be used. An additional ID-Pin specifies if the MCU provides host or device functionality.

Figure 3. Mini-USB - USB On-The-Go schematic



USB On-The-Go integrates USB host and function with the same connector. In this example the device functionality does only support full speed (only pull-up via D+). In host mode this configuration supports low- and full-speed if both transfer types are supported by the MCU. The functionality of the USB hardware abstraction layer is defined by the ID-Pin which is internally connected in the USB cable connector to ground if the USB hardware abstraction layer shall provide host functionality. Otherwise the ID-Pin is left open. For using the ID-Pin, the port-pin on MCU side must have internal pull-ups enabled.

Following scenarios demonstrating the pin/port usage:

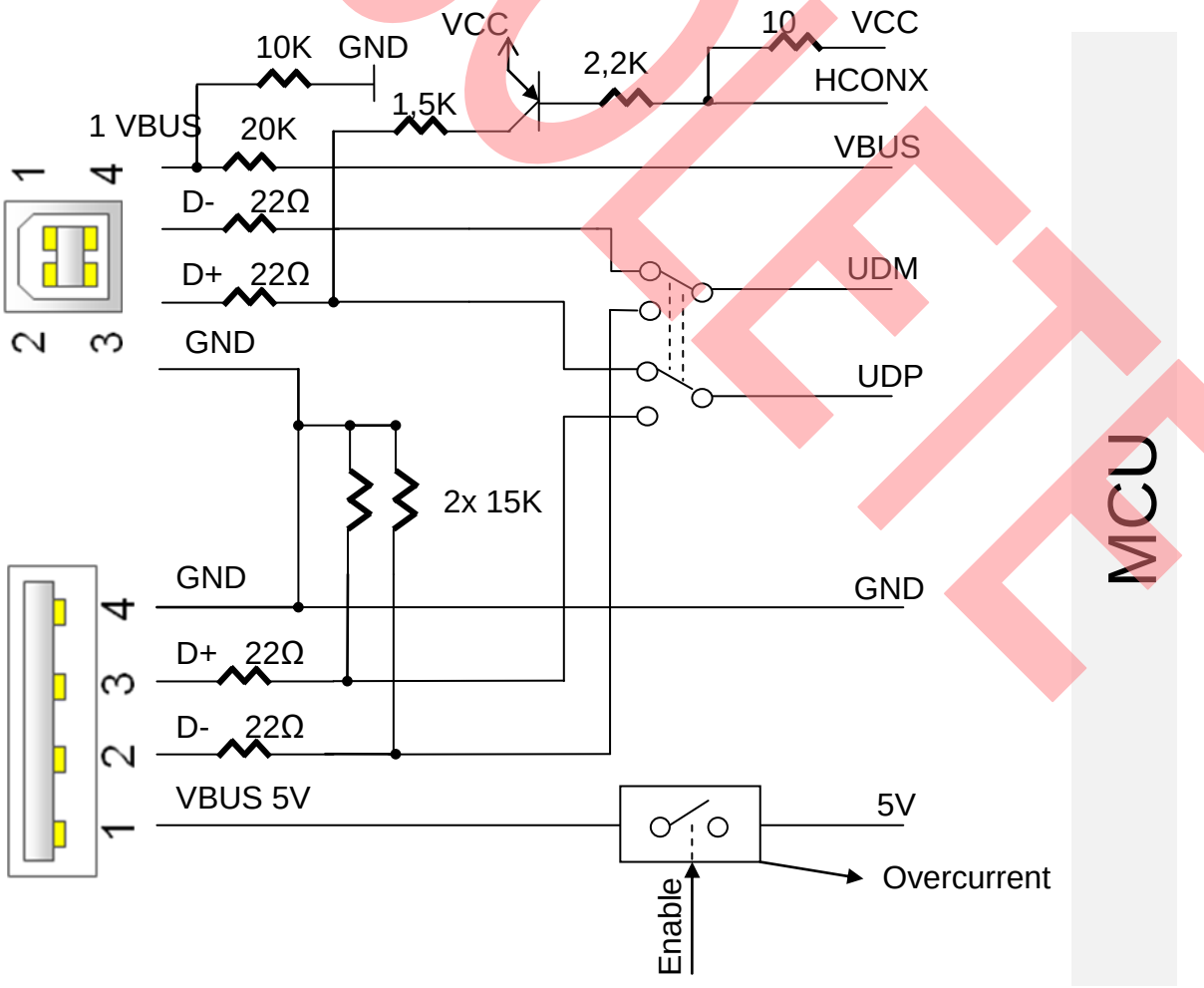
Table 1. On-The-Go Scenarios, X: don't care

Scenario	HOST (OUT)	HCONX (OUT)	ID (IN)	Enable (OUT)	Overcurrent Flag (IN)	VBUS IRQ (IN)
Device Mode	Low	Low	Open	Low	X	High
Device Mode Idle	Low	High	Open	Low	X	X
Host Mode	High	High	GND	High	Low	X
Host Mode > 500mA	High	High	GND	Low	High	X

1.2.4 USB On-Demand

The third case is using USB device and host functionality in the same application. This means switching USB host and function via switch, relay or special analogue IC. In this case both USB connectors will be implemented on the target system but only one mode at the same time can be realized. It is like USB On-The-Go with two connectors. This makes sense in size unlimited applications. But why not using directly USB On-The-Go? As negative Example: USB On-The-Go is mostly used in mobile devices because there is less space to implement connectors. For those applications the connector has to be as small as possible and less connectors will increase the mobility. But with these applications a special adapter is needed to convert the USB-Mini connector to the standard USB B female connector. In embedded industrial applications every adapter adds more problems and little connectors can be easily destroyed. If there is enough room for a second connector, it is much easier to connect USB devices directly and safely to the target system.

Figure 4. USB On-Demand schematic



2 USB Data Transfers Scenarios

To get the transition between USB specification and the hardware abstraction layer of the Fujitsu MCUs, the following chapters give a short overview about the different transfer types and how to implement them.

2.1 General Data Transfer

General transfers are IN and OUT token initiated data blocks. Fujitsu supports 2 physically integrated host endpoints with FIFO buffers for IN and OUT direction. All endpoint pipes are realized with these both endpoints.

Every data sent via USB is split into packages. These packages are limited by the FIFO buffer size depending on the host and device side. All packages are initiated via tokens.

2.1.1 IN transfers (Host IN)

Following table describes an IN token which is requested from Host side to read data from the Universal Serial Bus:

SYNC	IN	ADDR	EP	CRC5	EOP
00000001	0x69	0x02	0x02	0x01	2

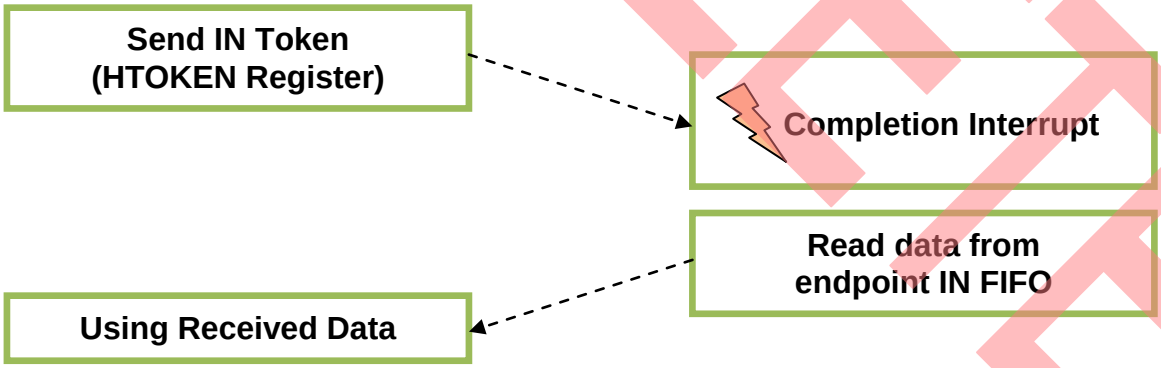
The USB Function device answers with the requested package:

SYNC	DATA0	DATA	CRC16	EOP
00000001	0xC3	00 11 22 33 44 55 66 77	0xCBA8	2

After this process the Host answers with an acknowledge (ACK):

SYNC	ACK	EOP
00000001	0xD2	2

The MCU handles this transfer as follows:



2.1.2 OUT transfers (Host OUT)

Following table describes an OUT token which is requested from host side to write data to the Universal Serial Bus:

SYNC	OUT	ADDR	EP	CRC5	EOP
00000001	0xE1	0x02	0x02	0x01	2

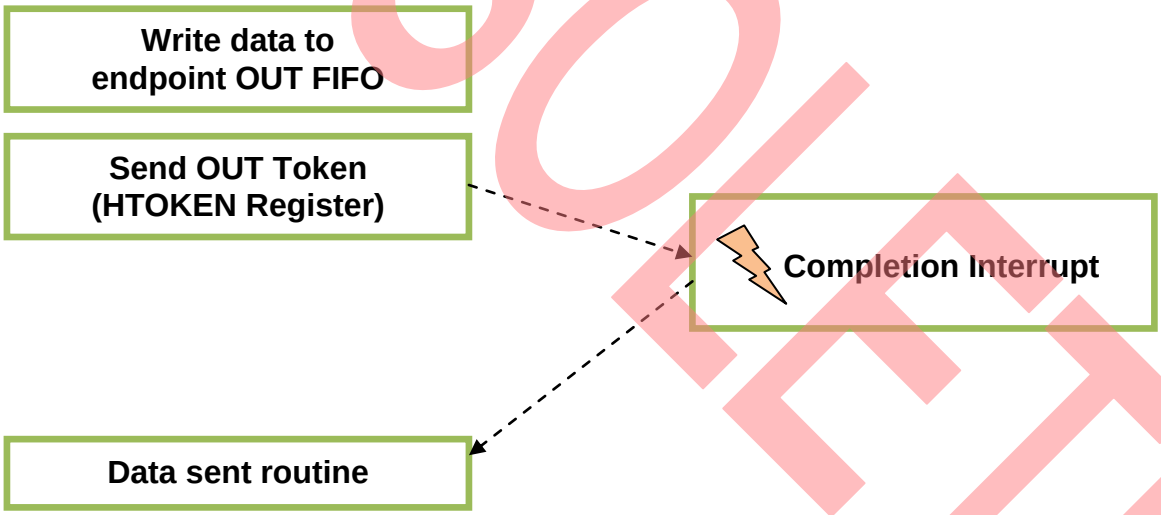
The USB function device answers with the requested package:

SYNC	DATA0	DATA	CRC16	EOP
00000001	0xC3	00 11 22 33 44 55 66 77	0xCBA8	2

After this process the host answers with an acknowledge (ACK):

SYNC	ACK	EOP
00000001	0xD2	2

The MCU handles this transfer as follows:



2.2 Control Transfers

Control Transfers have a special handshake data structure. Before a data package is requested (via IN / OUT token), a setup package is sent in the setup stage.

If there is no more data to send or to receive, the status stage will be entered. In the status stage a zero byte is transferred to finish the transfer. If the last package was an OUT or SETUP package, the zero byte is transferred from the USB function device, otherwise from the host.

If there is data to send or read, the Setup Stage is followed by the data stage.

The Setup Stage is initiated with following Token:

SYNC	SETUP	ADDR	EP	CRC5	EOP
00000001	0xD2	0x00	0x00	0x08	2

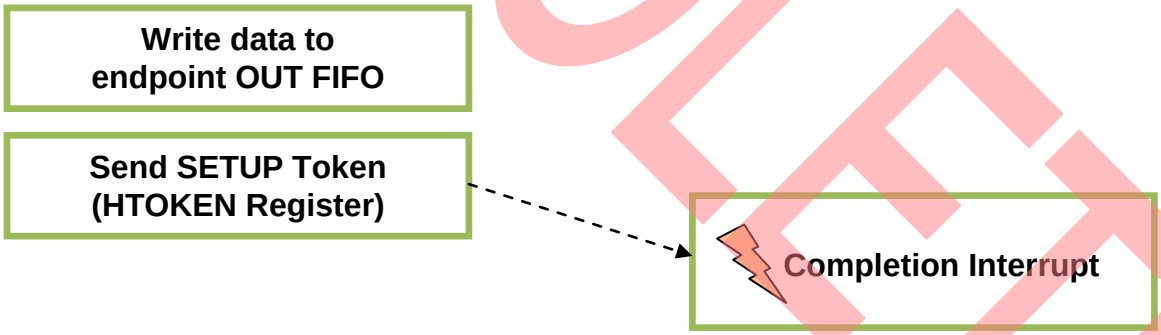
The host is sending 8 byte data containing the setup request. The data package is always of type DATA0:

SYNC	DATA0	DATA	CRC16	EOP
00000001	0xC3	00 05 02 00 00 00 00 00	0xD768	2

After this process the function answers with an acknowledge (ACK):

SYNC	ACK	EOP
00000001	0xD2	2

The MCU handles this transfer as follows:



If no more data has to be sent, the status stage is entered and the host has to request an IN token to receive the zero byte transfer:

SYNC	IN	ADDR	EP	CRC5	EOP
00000001	0x69	0x00	0x00	0x08	2

The host is receiving a zero byte. The data package is always of type DATA1:

SYNC	DATA1	DATA	CRC16	EOP
00000001	0x4B		0x0000	2

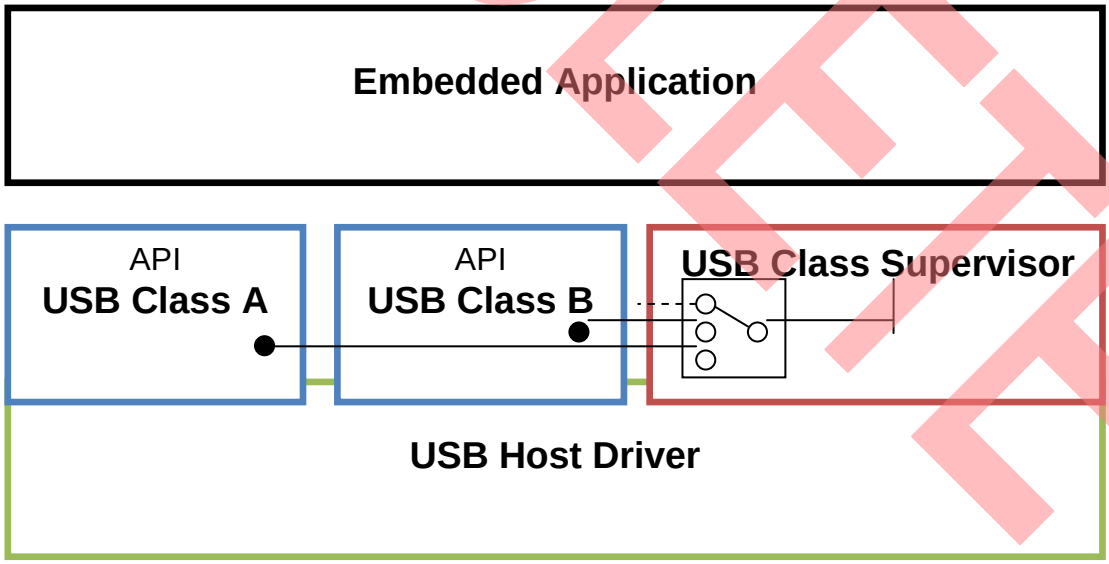
After this process the host answers with an acknowledge (ACK):

SYNC	ACK	EOP
00000001	0xD2	2

3 Embedded USB Host Concept

The driver part of USB has to establish the connection and the enumeration process. But for using an USB device there is more intelligence needed. Every device needs a kind of device driver using the USB host driver for its communication and building the interface to the embedded application. These device drivers (USB Classes) depend on the connected USB device. If different types of devices can be connected on the USB host, there must be a kind of device driver selector. This device driver selector is called “USB class supervisor” in the Fujitsu own USB concept. After choosing the matching device driver and initializing the USB device (Setting the USB configuration), the API of the active USB device can be used.

Figure 5. USB Host Stack Modules



3.1 USB Host Driver

The driver handles data transfers, the enumeration process and hot-plug. After a device was addressed, the USB Class Supervisor handles the next steps.

3.1.1 Virtual Endpoints

The USB host hardware abstraction layer supports only 2 endpoints for its communication: One endpoint for OUT transfers and the other for IN transfers.

To access the different endpoint pipes described in the USB specification as endpoint 0 to maximum 15, virtual endpoints must be built. These endpoints have to store different properties. Following `typedef` represents the different properties a virtual endpoint must support.

For USB Stack version < 2.0 (16FX, FR80, FM3):

Figure 6. Virtual USB Endpoints

```
typedef struct HostEndpoint
{
    uint8_t*  pu8Buffer;                    // buffer pointer
    uint8_t*  pu8BufferPos;                 // current pointer in buffer
    uint8_t*  pu8BufferPosNextPackage;  // temporary Pointer (set s current pointer
after ACK)
    uint32_t  u32DataSize;                  // data to send / data to receive
    uint32_t  u32BufferSize;                // size of buffer
    uint16_t  u16FifoSize;                  // virtual fifo size of this endpoint
    void      (* CompletionHandler)(void); // handler called after transfer is
complete
    boolean_t bToggle;                      // data package toggle
    uint8_t   u8LastToken;                  // last token executed with this endpoint
    uint8_t   u8Status;                     // endpoint status
    uint8_t   u8Stage;                      // endpoint status stage
    uint8_t   u8Address;                    // endpoint address
    uint8_t   u8Interval;                   // if interrupt endpoint this is the interval
    uint8_t   u8IntervalCount;              // used internally for interval generating
    uint8_t   u8Retries;                    // number of retried transfers
} HostEndpoint_t;
```

For USB Stack version >= 2.0 (FM3):

Figure 7. Virtual USB Endpoints

```
typedef struct stc_host_endpoint
{
    uint8 t*  pu8Buffer;                                // buffer pointer
    uint8 t*  pu8BufferPos;                             // current pointer in buffer
    uint8_t*  pu8BufferPosNextPackage;  // temporary Pointer (set s current pointer after ACK)
    uint32_t  u32DataSize;                              // data to send / data to receive
    uint32_t  u32BufferSize;                            // size of buffer
    uint16_t  u16FifoSize;                              // virtual fifo size of this endpoint
    void (* CompletionHandler)(stc_usbn_t* pstcUsb); // handler called after transfer is complete
    boolean_t bToggle;                                  // data package toggle
    uint8 t   u8LastToken;                              // last token executed with this endpoint
    volatile uint8 t   u8Status;                        // endpoint status
    volatile uint8_t   u8Stage;                         // endpoint status stage
    uint8_t   u8Address;                                // endpoint address
    uint8_t   u8Interval;                               // if interrupt endpoint this is the interval
    uint8 t   u8IntervalCount;                          // used internally for interval generating
    uint8 t   u8Retries;                                // number of retried transfers
    boolean_t bAbortTransfer;                           // force abort transfer
} stc_host_endpoint_t;
```

3.1.2 Data Transfers

Before data can be transferred, virtual endpoints have to be created. These endpoints can have up to 15 virtual addresses. Each of them can be of OUT or IN type.

For data transfers the physical endpoints 1 and 2 of the MCU hardware abstraction layer are used. One of both has to be the OUT endpoint and the other the IN endpoint. The direction is set via the direction flag in the endpoint control register.

Figure 8. Set the direction of the endpoint

```
CLEAR_MASK(EP1C, _EP1C_DIR);  // EP1 is a HOST IN endpoint
SET_MASK(EP2C, _EP2C_DIR);    // EP2 is a HOST OUT endpoint
```

All data is transferred via the FIFO buffer of the physically endpoints. For OUT or SETUP transfers, data has to be written into the endpoint FIFO buffer, before the OUT or SETUP token is executed. As defined before endpoint 2 is used as OUT endpoint.

For USB Stack version < 2.0 (16FX, FR80, FM3):

Figure 9. Used to transfer data for an OUT or SETUP token

```
void UsbHost_TransferDataToFifo(uint8_t* pu8Buffer, uint16_t u16Size, uint16_t
u16FifoSize)
{
    boolean_t Odd = u16Size & 1;
    EP2S = (EP2S & (~0x1F)) | u16FifoSize; // setting fifo size
    u16Size = u16Size / 2; // transfer 16 Bit words;
    while(u16Size--)
    {
       EP2DT = *(uint16_t*)pu8Buffer;
       pu8Buffer += 2;
    }
    if(Odd == TRUE)
    {
        EP2DTL= *(uint8_t*)pu8Buffer; // transfer the first or last byte
    }
    CLEAR_MASK(EP2S, _EP2S_DRQ);    // now the OUT FIFO is valid for the next transfer
}
```

For USB Stack version >= 2.0 (FM3):

Figure 10. Used to transfer data for an OUT or SETUP token

```
void UsbHost TransferDataToFifo(stc usbn t* pstcUsb, uint8 t* pu8Buffer, uint16 t u16Size, uint16 t
u16FifoSize)
{
    boolean_t Odd = u16Size & 1;
    pstcUsb->EP2S = (pstcUsb->EP2S & (~0x1F)) | u16FifoSize; // setting fifo size

    u16Size = u16Size / 2; // transfer 16 Bit words;
    while(u16Size--)
    {
           pstcUsb->EP2DTL = *pu8Buffer++;
           pstcUsb->EP2DTH = *pu8Buffer++;
    }
    if(Odd == TRUE)  {
        pstcUsb->EP2DTL= *(uint8 t*)pu8Buffer; // transfer the first or last byte
    }
    pstcUsb->EP2S_f.DRQ = 0;   // now the OUT FIFO is valid for the next transfer
}
```

For receiving data via an IN token, first the IN token is sent. In the completion interrupt the data can be read from endpoint 1 previously defined as IN endpoint.

For USB Stack version < 2.0 (16FX, FR80, FM3):

Figure 11. Used to transfer data after a successful IN token

```
uint16_t UsbHost_TransferFifoToBuffer(uint8_t* pu8Buffer)
{
    uint16_t u16Size = (uint16_t)(EP1S & 0x00FF);
    uint16_t u16ReceivedSize = u16Size;
    u16Size = u16Size / 2; // transfer 16 Bit words;
    while(u16Size--)
    {
        *(uint16_t*)pu8Buffer = EP1DT;
        pu8Buffer += 2;
    }
    if(u16ReceivedSize & 1 == TRUE)  {
        *(uint8_t*)pu8Buffer = EP1DTL; // transfer the first or last byte
    }
    CLEAR_MASK(EP1S, _EP1S_DRQ);    // now the IN FIFO is valid for the next transfer
    return u16ReceivedSize;
}
```

For USB Stack version >= 2.0 (FM3):

Figure 12. Used to transfer data after a successful IN token

```
uint16 t UsbHost TransferFifoToBuffer(stc usbn t* pstcUsb, uint8 t* pu8Buffer)
{
    uint16_t u16Size = 0;
    uint16_t u16ReceivedSize;
    u16Size = (uint16 t)(pstcUsb->EP1S & 0x00FF);
    u16ReceivedSize = u16Size;
    u16Size = u16Size / 2; // transfer 16 Bit words;
    while(u16Size--)
    {
            *pu8Buffer++ = pstcUsb->EP1DTL;
            *pu8Buffer++ = pstcUsb->EP1DTH;
    }
    if(u16ReceivedSize & 1 == TRUE)  {
        *(uint8_t*)pu8Buffer = pstcUsb->EP1DTL; // transfer the first or last byte
    }
    pstcUsb->EP1S_f.DRQ = 0;   // now the IN FIFO is valid for the next transfer
    return u16ReceivedSize;
```

3.1.3 Token Handling

Tokens can be sent via the HTOKEN register.

Table 2. Host Token Endpoint Register (HTOKEN)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TGGL	TKEN2	TKEN1	TKEN0	EDPT3	EDPT2	EDPT1	EDPT0

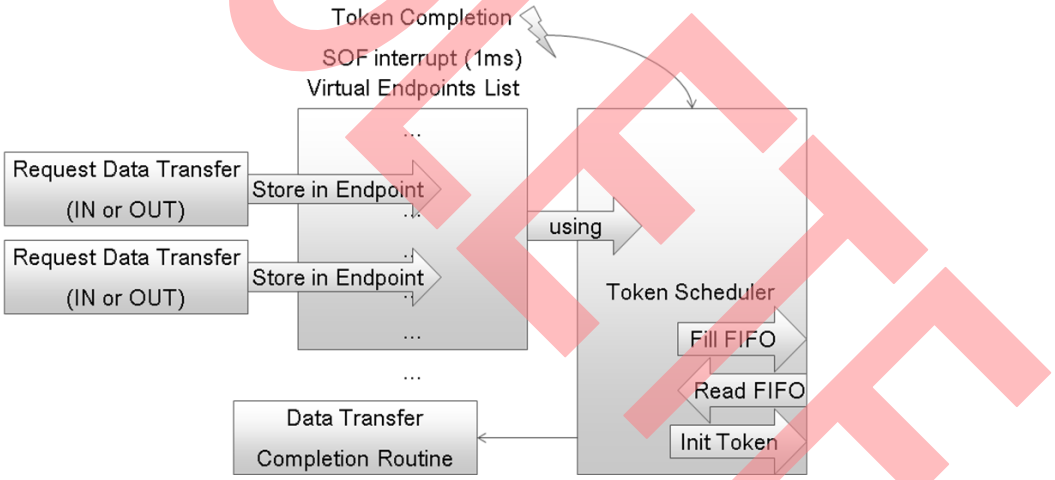
Each Token is linked to a virtual endpoint. The TGGL flag represents the different data packages: DATA0 or DATA1 (TGGL=1 → DATA1). These data packages are used rotational during data transfers. After a successful transfer (handshake was ACK) this flag is toggled in the virtual endpoint. For a new token the toggle status flag of the virtual endpoint is used as TGGL flag. The TKEN0..2 bits are used to define the type of token to be sent. This can be SOF, SETUP, IN, or OUT. The EPDT0..3 bits define the endpoint address without the direction bit.

After the TKEN0..2 bits are written, the token will be executed! The token is automatically executed with a SYNC. For SETUP or OUT transfers first the data has to be transferred to the physically OUT endpoint FIFO buffer. After filling the buffer the token can be initiated. For IN transfers the IN token has to be initiated first. After successful handshake (ACK) the data can be read from the physically IN endpoint FIFO buffer in the completion interrupt routine.

3.1.4 Token Scheduler

To prevent collisions a kind of scheduler for tokens is introduced. This Scheduler is executed by a SOF interrupt and the completion interrupt. If tokens are complete, next data can be transferred. If all data were transferred a data transfer complete event can be executed. All SOF-initiated interrupts are used to handle “USB interrupt transfers” and initiate new transfers. The scheduler also handles the priority and considers at last, which kind of token has to be sent.

Figure 13. Token Scheduler



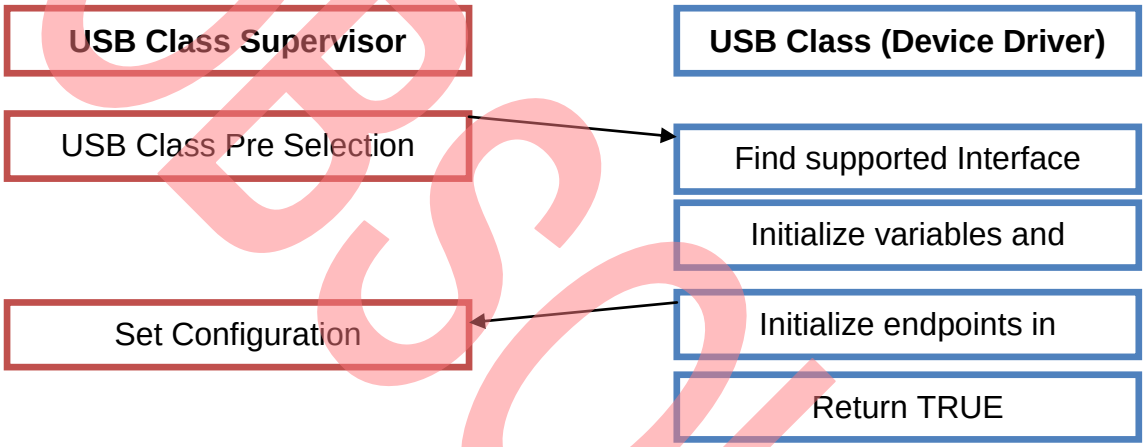
3.2 USB Class Supervisor

The class supervisor is called after the enumeration process was finished successfully. This case is reached by the device addressed state. The next intention is to reach a configured device. But this can only be done after the right device driver was found and the device driver has initialized all necessary endpoints for its configuration.

All information about the device type is found in the USB descriptors which will be requested from the connected device. First the device descriptor is requested. The device descriptor can contain information about the pre-selection of an existing device driver. To get detailed information about the connected device, the configuration descriptor is requested too. Now the USB Class Supervisor can select matching device drivers. If the driver can handle the specific device, it returns `TRUE` after initialization. All necessary endpoints and buffers should be now safe (pointers set correctly to the initialized buffers) and the configuration can be set (device configured state).

3.3 USB Classes

A USB class has the duty to build the missing part between USB host driver and main application. The USB Class is a kind of device driver. The pre-selection is handled by the USB class supervisor. If the pre-selection matches to the USB class type, the initialization routine located in the class driver is called with the received configuration descriptor data buffer as parameter. In the USB class initialization process all necessary endpoints are set, if the configuration and the interface descriptors are matching to the USB class supported USB device.



Following code gives an example how the pre-selection in the USB Class Supervisor can be realized:

Figure 14. Find the matching USB class driver

```
for(i=0;i<MAX_CLASSHANDLERS;i++)
{
   if (UsbSupervisor_UsbClassMatch(&UsbClassHandlers[i],&stcMatchClass))
   {
       if (UsbClassHandlers[i].InitializeClass(pu8Configuration,u32ConfigurationSize)
== TRUE)
       {
           dbg("Setting Configuration (1)\n");
           UsbHost_SetConfigurationDescriptor(1);
           CurrentDriver = UsbClassHandlers[i].stcDriverType;
           break;
       }
   }
}
```

After the initialization process the USB class is dynamically connected with the different event handlers for all data pipes and handles the data transfers and device specific protocol. The different protocols are described in the USB specification.

3.3.1 Setting up own classes

First an initialization handler must be built. This handler gets as a parameter a buffer pointer containing the data of the pre-selected configuration and the length of this configuration. The USB Host library offers some procedures to get a pointer to the different descriptors in the configuration.

For USB Stack version < 2.0 (16FX, FR80, FM3):

Figure 15. Example USB Class Initialization Handler

```
static HostEndpoint_t *EndpointIN;
static uint8_t        u8MouseData[5];

boolean_t HidMouseInitHandler(uint8_t* pu8Configuration, uint32_t u32Length)
{
}
```

For USB Stack version >= 2.0 (FM3):

Figure 16. Example USB Class Initialization Handler

```
static stc_host_endpoint_t *pstcEndpointIN;
static uint8_t        pu8MouseData[5];
static stc_usbn_t* pstcUsbHandle = NULL;

boolean_t UsbHostHidMouse_InitHandler(stc_usbn_t* pstcUsb, uint8_t* pu8Configuration, uint32_t u32Length)
{
    ...
    if (pstcUsbHandle != NULL)
    {
        return FALSE;
    }
    pstcUsbHandle = pstcUsb;
    ...
}
```

The number of interface descriptors in the configuration descriptor can be easily extracted by:

Figure 17. Reading number of interface descriptors

```
u8NumberOfInterfaces = pu8Configuration[4];
```

Now all interfaces can be scanned if they are supported or not:

Figure 18. Scan interface descriptors

```
for(u8InterfaceNumber=0;u8InterfaceNumber < u8NumberOfInterfaces;u8InterfaceNumber++)
{
  pu8Buffer =
UsbHost_GetUsbInterfaceDescriptor(pu8Configuration,u32Length,u8InterfaceNumber);
  if (pu8Buffer == 0)
  {
    /* Error: Could not Request Interface Descriptor */
    return FALSE;
  }
  u8NumberOfEndpoints = pu8Buffer[4];
  if ((pu8Buffer[6] == 0x01) && (pu8Buffer[7] == 0x2)) // SubClass & Protocol is
supported?
  {
    /* process next steps here */
  }
}
```

In `/* process next steps here */` the code for finding all necessary endpoints with their addresses must be written.

For a USB mouse with only one endpoint it could look like this:

```
for(u8EndpointNumber=1;u8EndpointNumber<=u8NumberOfEndpoints;u8EndpointNumber++)
{
    pu8Buffer = UsbHost_GetUsbEndpointDescriptor(
                    pu8Configuration,
                    u32Length,
                    u8InterfaceNumber,
                    u8EndpointNumber);
    if (pu8Buffer[2] & USB_IN_DIRECTION)
    {
        u8EndpointIN = pu8Buffer[2];
        u16MaxPackageSizeIN = (uint16_t)(pu8Buffer[4] + (pu8Buffer[5] << 8));
        u8Interval = pu8Buffer[6];
    }
    if (u8EndpointIN != 0)
    {
        break; // Endpoint found, exit loop
    }
}
```

After searching for endpoints, they have to be initialized. This is done by calling the `UsbHost_AddHostEndpoint` procedure. Here also the data received handler can be initialized for static receive handlers.

For USB Stack version < 2.0 (16FX, FR80, FM3):

```
if (u8EndpointIN != 0)
{
    UsbHost_AddHostEndpoint(
        &EndpointIN,
        u8EndpointIN,
        u16MaxPackageSizeIN,
        u8Interval,
        HidMouse_DataReceivedHandler
    );
    EndpointIN->pu8Buffer = pu8MouseData;
    EndpointIN->u32DataSize = 5;
    return TRUE;
}
else
{
    return FALSE;
}
```

For USB Stack version >= 2.0 (FM3):

```
if (u8EndpointIN != 0)
{
    UsbHost_AddHostEndpoint(
        pstcUsbHandle,
        &pstcEndpointIN,
        u8EndpointIN,
        u16MaxPackageSizeIN,
        u8Interval,
        HidMouse_DataReceivedHandler
    );
    EndpointIN->pu8Buffer = pu8MouseData;
    EndpointIN->u32DataSize = 5;
    return TRUE;
}
```

```
else
{
    return FALSE;
```

Now the initialization process is handled but the handler procedure is not defined. The receive handler can look like the following procedure:

For USB Stack version < 2.0 (16FX, FR80, FM3):

```
void HidMouse_DataReceivedHandler()
{
    EndpointIN->CompletionHandler = HidMouse_DataReceivedHandler;
}
```

For USB Stack version >= 2.0 (FM3):

```
void HidMouse_DataReceivedHandler(stc_usbn_t* pstcUsb)
{
    EndpointIN->CompletionHandler = HidMouse_DataReceivedHandler;
}
```

The received data is stored within the virtual endpoint structure. To use the received data, the buffer can be extracted as followed:

```
u8MouseButtons = EndpointIN->pu8Buffer[0];
```

3.3.2 Class Driver Table

The USB Class Driver Table is defined in the file *ClassDriverTable.h*. It contains constants for all necessary information to initialize the correct class driver via the `UsbClassSupervisor`.

In this file 3 sections are used to specify existing driver classes. First the class driver header is included (1). If the class type is completely different than the specified types, it has to be defined here (2). With the constant `struct array UsbClassHandlers` detection conditions of the different class drivers can be specified (3). `MAX_CLASSHANDLERS` defines the maximum number of classes in the `UsbClassHandlers` array. If a condition shall be ignored, it has to be set to `0`.

```
/*****************************************************************************/
/* Include files                                                             */
/*****************************************************************************/

#include "UsbMassStorage.h"
#include "HidMouse.h"
/* (1) OTHER USB CLASS DRIVER INCLUDES CAN BE ADDED HERE */

/*****************************************************************************/
/* Global pre-processor symbols/macros ('#define')                           */
/*****************************************************************************/

#define USBCLASSDRIVER_MASSSTORAGE 1
#define USBCLASSDRIVER_MOUSE       2
#define USBCLASSDRIVER_JOYSTICK    3
/* (2) OTHER USB CLASS DRIVER TYPES CAN BE ADDED HERE */

/*****************************************************************************/
/* Global variable declarations ('extern', definition in C source)           */
```

```
/*****************************************************************************/

#ifdef __USBCLASSSUPERVISOR_C__
    #define MAX_CLASSHANDLERS 2
    const UsbClassHandler_t UsbClassHandlers[] =
    {
        {
            MassStorageInitHandler,    // Handler called to initialize the driver
            MassStorageDeinitHandler,  // Handler called after a device disconnection
            0,0,    // 0 = ignored     // Vendor ID, Product ID
            0,0,0,  // 0 = ignored     // Device Class, Device SubClass, Device
Protocol
            0x08,0x06,0x50,                // Interf. Class, Interf. SubClass, Interf.
Protocol
            USBCLASSDRIVER_MASSSTORAGE // Driver Type
        },
        {
            HidMouseInitHandler,       // Handler called to initialize the driver
            HidMouseDeinitHandler,     // Handler called after a device disconnection
            0,0,    // 0 = ignored     // Vendor ID, Product ID
            0,0,0,  // 0 = ignored     // Device Class, Device SubClass, Device
Protocol
            0x03,0x01,0x02,            // Interf. Class, Interf. SubClass, Interf.
Protocol
            USBCLASSDRIVER_MOUSE       // Driver Type
        }
        /* (3) OTHER USB CLASS DRIVER SETTINGS FOR DETECTION CAN BE ADDED HERE */
    };
#endif
```

4 USB Host API (Version < 2.0)

HOW TO USE THE USB HOST STACK API FOR 16FX, FR80 AND FM3

4.1 Files

Filename	Directory	Description
UsbClassSupervisor.c	*Fujitsu/UsbHost/*	The Class Supervisor selects the class supported by the current device.
UsbClassSupervisor.h	*Fujitsu/UsbHost/*	Header File
UsbHost.c	*Fujitsu/UsbHost/*	USB Host Driver. Handles: Data Transfers Token Scheduling Enumeration Hotplug
UsbHost.h	*Fujitsu/UsbHost/*	Header File
UsbInit.c	*Fujitsu/Usb/*	MCU specific initializations needed for USB. For example USB clock.
UsbInit.h	*Fujitsu/Usb/*	Intelligent environment and MCU recognition
UsbRegisters.h	*Fujitsu/Usb/*	Register translation between different MCU series, country specific header files and different environments
UsbSofTimeout.c	*Fujitsu/Usb/*	SOF interrupt handling and ms Timeout handling
UsbSofTimeout.h	*Fujitsu/Usb/*	Header File
UsbSpec.h	*Fujitsu/Usb/*	USB Spec defines and type definitions
ClassDriverTable.h	*Fujitsu/UsbHost/Classes/*	USB Class Driver definition table
Optional*:*		
HidMouse.c	*Fujitsu/UsbHost/Classes/HidMouse/*	USB Mouse Class Driver
HidMouse.h	*Fujitsu/UsbHost/Classes/HidMouse/*	Header File
StorageApi.c	*Fujitsu/UsbHost/Classes/MassStorage/*	Storage API for File system
StoraheApi.h	*Fujitsu/UsbHost/Classes/MassStorage/*	Header File
UsbMassStorage.c	*Fujitsu/UsbHost/Classes/MassStorage/*	Mass Storage Class Driver
UsbMassStorage.h	*Fujitsu/UsbHost/Classes/MassStorage/*	Header File

4.2 Type definitions

4.2.1 UsbRequest_t (UsbHost.h)

Definition

```
typedef struct UsbRequest
{
    uint8_t  bmRequestType;
    uint8_t  bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
} UsbRequest_t;
```

4.2.2 SetupPackage_t (UsbHost.h)

Definition

```
typedef struct SetupPackage
{
    UsbRequest_t* pstcRequest;
    uint8_t*      pu8Buffer;
} SetupPackage_t;
```

4.2.3 HostEndpoint_t (UsbHost.h)

Definition

```
typedef struct HostEndpoint
{
    uint8_t*  pu8Buffer;
    uint8_t*  pu8BufferPos;
    uint8_t*  pu8BufferPosNextPackage;
    uint32_t  u32DataSize;
    uint32_t  u32BufferSize;
    uint16_t  u16FifoSize;
    void      (* CompletionHandler)(void);
    boolean_t bToggle;
    uint8_t   u8LastToken;
    uint8_t   u8Status;
    uint8_t   u8Stage;
    uint8_t   u8Address;
```

```
    uint8_t    u8Interval;
  uint8_t  u8IntervalCount;
  uint8_t  u8Retries;
} HostEndpoint_t;
```

4.3 API Functions

4.3.1 UsbHost_Init

This procedure is used to initialize the USB Host in general. Note that on some MCUs the USB hardware abstraction layer clock must be set. This can be done by calling `UsbInit()` before initializing the UsbHost.

Definition

```
void UsbHost_Init(void);
```

4.3.2 UsbHost_GetDeviceStatus

The function returns the current device status.

Definition

```
uint8_t UsbHost_GetDeviceStatus(void);
```

Return Value

USBHOST_DEVICE_IDLE	1
USBHOST_DEVICE_ADDRESSED	2
USBHOST_DEVICE_ADDRESSING	3
USBHOST_DEVICE_ENUMERATED	4
USBHOST_DEVICE_CONFIGURATING	5
USBHOST_DEVICE_CONFIGURED	6

4.3.3 UsbHost_SetConfigurationDescriptor

This function is used by the Usb Class Supervisor to set the configuration of a supported device.

Definition

```
void UsbHost_SetConfigurationDescriptor(
uint8_t u8Configuration
);
```

Parameter

`u8Configuration` Configuration to set (normally 1)

4.3.4 UsbHost_SetupRequest

The procedure initializes an USB setup request with a callback function which can also handle possible received data.

Definition

```
void UsbHost_SetupRequest(
UsbRequest_t* pstcSetup,
void(*SetupCompletion)(
uint8_t* pu8Buffer,
uint32_t u32DataSize
)
);
```

Parameter

UsbRequest_t	Setup Request
SetupCompletion	Setup Completion Routine

4.3.5 UsbHost_GetUsbInterfaceDescriptor

Definition

```
uint8_t* UsbHost_GetUsbInterfaceDescriptor(
uint8_t* pu8Buffer,
uint16_t u16Size,
uint8_t u8InterfaceNumber,
);
```

Parameter

pu8Buffer	Buffer containing the whole configuration data
u16Size	maximum size of the buffer
u8InterfaceNumber	Index number of interface

Return Value

Bufferpointer to the interface descriptor raw data

4.3.6 UsbHost_GetUsbEndpointDescriptor

Definition

```
uint8_t* UsbHost_GetUsbEndpointDescriptor(
```

```
uint8_t* pu8Buffer,
uint16_t u16Size,
uint8_t u8InterfaceNumber,
uint8_t u8EndpointNumber
);
```

Parameter

pu8Buffer	Buffer containing the whole configuration data
u16Size	maximum size of the buffer
u8InterfaceNumber	Index number of interface
u8EndpointNumber	Index number of endpoint

Return Value

Bufferpointer to the endpoint descriptor raw data

4.3.7 UsbHost_TransferData

The procedure is used to initiate an in- or outgoing transfer.

Definition

```
void UsbHost_TransferData(
HostEndpoint_t* Handler,
uint8_t* pu8Buffer,
uint32_t u32BufferSize,
void (* CompletionHandler)(void)
);
```

Parameter

`Handler`	Host endpoint to transfer data with
`pu8Buffer`	Buffer to transfer from or to transfer to
`u32BufferSize`	Data Size to send or maximum data to receive
`CompletionHandler`	Handler called after data was sent or received

4.3.8 Usb_AddTimeOut

This procedure can be used to add a delayed called function. The clock for the timeout is generated via SOF. This means: Timeout is only supported while a device is connected!

Definition

```
boolean_t Usb_AddTimeOut(
```

```
void (* Handler)(void),
uint16_t u16TimeOut
);
```

Parameter

Handler	Handler called delayed
u16TimeOut	Timeout interval in ms interval

Return Value

`TRUE` on success, otherwise `FALSE`

4.3.9 Usb_RemoveTimeOut

This procedure can be used to remove a delayed called function. The clock for the timeout is generated via SOF. This means: Timeout is only supported while a device is connected!

Definition

```
boolean_t Usb_RemoveTimeOut(
void (* Handler)(void)
);
```

Parameter

Handler	Handler to remove from list

Return Value

`TRUE` on success, otherwise `FALSE`

5 USB Host API (Version >= 2.0)

HOW TO USE THE USB HOST STACK API FOR FM3 AND HIGHER

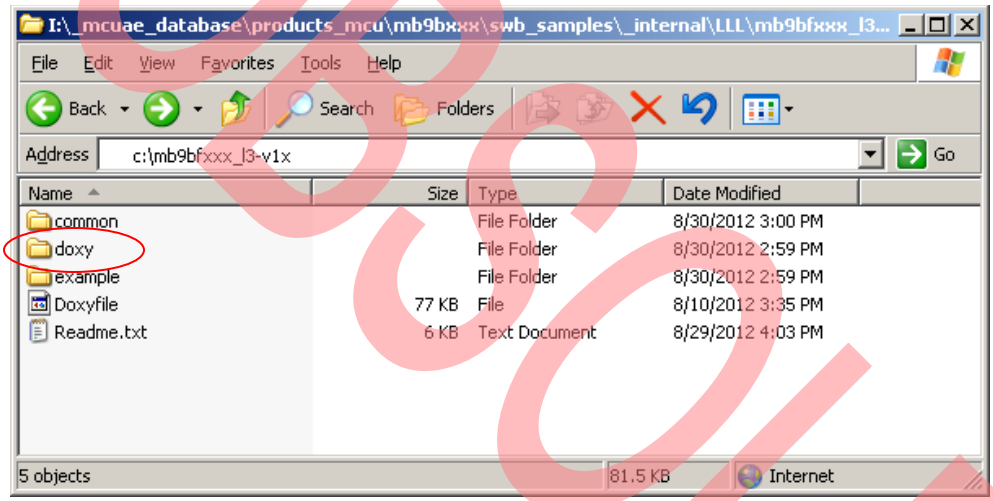
The API of the USB stack is now described within the Low Level Library (L3) for FM3.

See also (mb9bfxxx_l3):

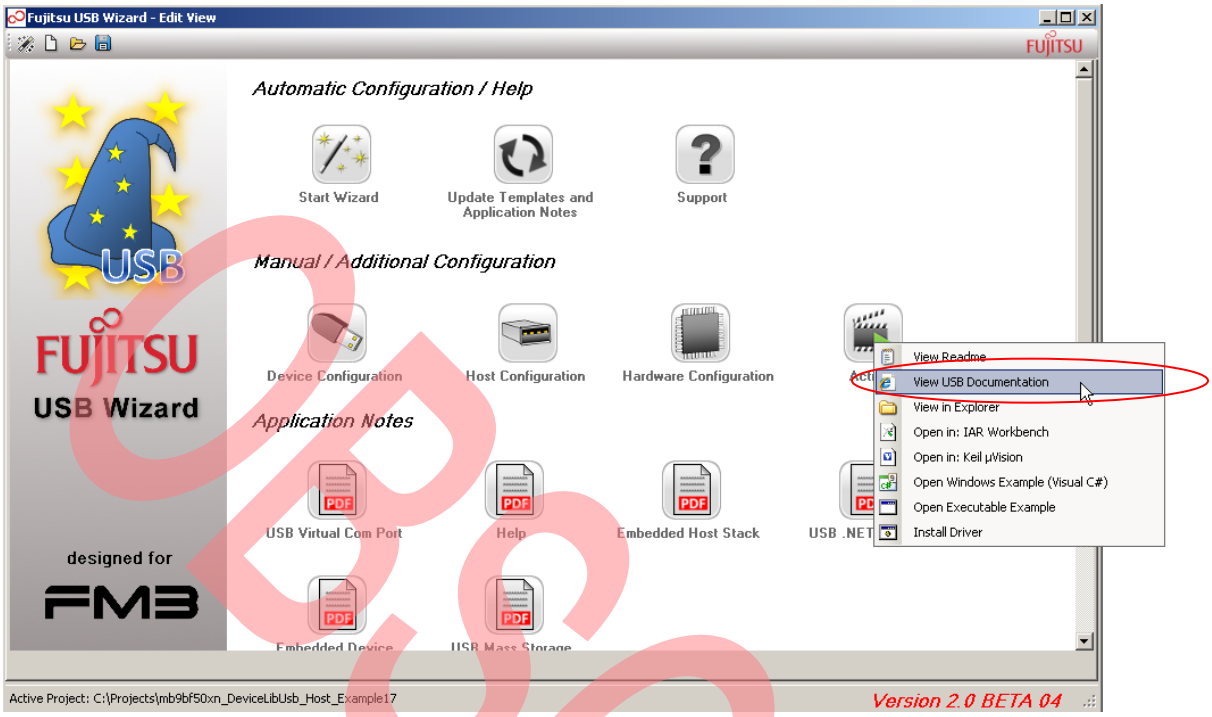
http://www.spansion.com/Products/microcontrollers/Pages/default.aspx

or

http://www.spansion.com/Support/microcontrollers/sampleprogram/Pages/fm3.aspx



If you are using the Fujitsu USB Wizard, the doxy directory will be automatically created inside of the project directory.



5.1 Files

Filename	Directory	Description
UsbClassSupervisor.c	*UsbMiddleware*	The Class Supervisor selects the class supported by the current device.
UsbClassSupervisor.h	*UsbMiddleware*	
UsbHost.c	*L3*	USB Host Driver. Handles: Data Transfers Token Scheduling Enumeration Hotplug
UsbHost.h	*L3*	
usbethernetclock.c	*L3*	USB Clock Initialization
usbethernetclock.h	*L3*	
usb.c	*L3*	General USB support
usb.h	*L3*	
base_types_l3.h	*L3*	Base Types
ClassDriverTable.h	*UsbConfig*	USB Class Driver definition table
Optional:		
UsbHost<NAME>.c	*UsbMiddleware*	Device Driver
UsbHost<NAME>.h	*UsbMiddleware*	

5.2 Examples

5.2.1 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 Fujitsu MCU 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 19. Switch to USB device

```
/* switch USB device */
Usb_SwitchUsb(&USB0, UsbSwitchToDevice, 0);
```

Figure 20. Switch to USB host

```
/* switch USB host */
Usb_SwitchUsb(&USB0, UsbSwitchToHost, 0);
```

Figure 21. Switch depending USB device VBUS

```
/* switch USB depending USB device VBUS */
Usb_SwitchUsb(&USB0, UsbSwitchDependingDeviceVbus, 0);
```

Figure 22. Switch USB off

```
/* switch USB off */
Usb_SwitchUsb(&USB0, UsbSwitchAllOff, 0);
```

6 Appendix

6.1 Information in the WWW

Dedicated information on Cypress Microcontroller products including datasheets and manuals, software examples, application notes and tools can be found here:

http://www.spansion.com/Products/microcontrollers/Pages/default.aspx

7 Document History

Document Title: AN205487 - 16FX/FR80/FM3 USB Family 16/32-Bit Microcontroller

Document Number: 002-05487

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	–	MSCH	21/07/2011	V1.0, MSc, First Version
			13/08/2012	V1.1, MSc, Description for USB Stack < 2.0 and >= 2.0 added
*A	5041727	MSCH	12/08/2015	Converted Spansion Application Note “MCU-AN-300126-E-V11”to Cypress format

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless
Spansion Products	spansion.com/products

PSoC® Solutions

psoc.cypress.com/solutions

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

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

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