UM0463
User manual

ST7 full-speed USB library

Introduction

The "ST7 Full-Speed USB Library" (hereafter "the library") is designed for ST7 full-speed USB products. With the library, the user of the ST7265 and ST7SCR will have easy access to the embedded USB cell. The library is provided with full source code and it can be used for any USB application. The library also provides an optional feature to upgrade the firmware of the microcontroller through the USB. This document describes the use and the implementation of the library.

- Definitions:
  - USB Universal Serial Bus
  - HID Human Interface Devices
  - DFU Device Firmware Update
  - IAP In-Application Programming

Contents

List of tables

1 Overview

1.1 Functionality of the library

As a generic USB library, it provides the following functionalities:

- Initialize the USB hardware cell
- Answer all standard USB requests defined in Chapter 9 of USB specification [1]
- Enumerate the USB device to the host with less user code intervention
- Means for plug-in user code to handle nonstandard USB requests
- Functions for sending and/or receiving data on all the endpoints
- Optional DFU functionality without any user code intervention

1.2 Background of ST7 full-speed USB device

1.2.1 USB device

The ST7 full-speed USB device implements several USB endpoints. The number of endpoints varies among products. *Table 1* lists the endpoint configuration of each product:

Table 1. Endpoint configuration

Endpoint	0	1 IN	1 OUT	2 IN	2 OUT	3 IN	4 IN	5 IN
ST7265	16	16	16	64	64	N/A	N/A	N/A
ST7SCR	8	8	N/A	64	64	8	8	8

The figures in the above table represent the maximum packet size allowed for a particular endpoint. An endpoint is identified with its address and its direction. The endpoint in address 0 is the control endpoint and is always bidirectional. In the table, IN identifies that the direction of the endpoint is from the device to the host and OUT identifies that the direction of the endpoint is from the host to the device.

The data exchange on endpoints is done through endpoint DMA buffers. The endpoint DMA buffers are located in the fixed location of RAM area and their locations vary among products.

There is a pair of registers corresponding to each endpoint. They are the endpoint status register and the DMA counter register. The endpoint status register identifies four status of the endpoint:

Table 2. Endpoint status

Status	Meaning
DISABLE	The endpoint does not answer any bus traffic.
STALL	The endpoint stalls any IN token for IN direction or OUT token for OUT direction.

Table 2. Endpoint status (continued)

Status	Meaning
NAK(1)	The endpoint responds with NAK to any IN token for IN direction or NAK to any OUT token for OUT direction, and without data transfer.
ACK	– IN direction: The endpoint can send data to the host and the counter register gives the number of bytes to be sent. – OUT direction: The endpoint can receive data and responses with ACK after the data packet and the counter register indicates the number of bytes received after the device ACKs the data packet. At the end of the data transmission, the hardware machine switches to NAK.

1. In case of error on data toggling bit, the hardware machine does the data retransmission

There are two interrupt sources that inform the software about the USB status, generic USB interrupt and the End of Suspend interrupt. The generic USB interrupt integrates six USB events. They are defined in *Table 3*.

Table 3. USB events

Events	Meaning
Correct Transfer (CTR)	A correct USB transfer is complete on either endpoint. The software is responsible for checking what the endpoint is.
Setup Overrun (SOVR)	A SETUP transfer is complete when a previous CTR is pending. It happens when the software does not have enough time to process a CTR event.
Error (ERR)	The USB cell detects a bus error. Register ERRSR gives the type of error.
Suspend request (SUSP)	The USB cell detects that the bus is idle for more than 3ms, requiring the device to enter suspend mode.
Reset (RESET)	The USB cell detects a USB reset sequence on the bus.
Start of frame (SOF)	A SOF token is received.

The End Of Suspend (ESUSP) interrupt is asserted to wake up the device when bus activity is detected in suspend mode.

In summary, to send data from the device to the host, one has to copy the data to one of the IN endpoint buffers, then set the number of bytes to be sent in the corresponding endpoint counter register, and set the endpoint status register to ACK state. Afterward, the device waits for the USB host to send an IN token to that endpoint and the device sends the data bytes after the IN token. After the host receives the data packet, it sends an ACK token to acknowledge reception. The device sees the ACK token from the host and generates a CTR interrupt to inform the software that the data transmission is successful.

Contrary to receiving data from the host, the corresponding endpoint counter register has to be set as the number of bytes expected and the endpoint status register is set to ACK state. When the host sends an OUT token to that endpoint and followed by the data packet, the device copies the data packet to the endpoint DMA buffer. After the device receives the data packet correctly, it sends an ACK token to acknowledge reception and then generates a CTR interrupt to inform the software that the data has been successfully received. At this time, the counter register contains the difference of expected bytes and received bytes.

If the device wants to send or receive data bytes containing more than the maximum packet size of an endpoint, several sending or receiving procedures must be performed.

2 Programming model of the library

2.1 Handling control endpoint 0

The USB specification[1] defines four transfer types, Control, Interrupt, Bulk and Isochronous. The USB host sends requests to the device through the control endpoint (every USB device has only one control endpoint). The format of the requests is defined and is sent to the device as a SETUP packet. The meaning of the requests is classified into three categories: standard, class specific and vendor specific.

Since the standard requests are common and generic to all USB devices, the library receives and handles all standard requests on the control endpoint 0. The library also handles DFU specific requests by itself without any user code intervention. The library answers requests without the intervention of the user application if the library has enough information about the requests.

Otherwise, the library will call user application defined callback functions to process the requests when some application actions are needed or some information from the application is needed.

The format and the meaning of the class specific requests and the vendor specific requests are not common to all USB devices. The library does not handle any of the requests in these categories except DFU requests. Whenever the library receives a request that it does not know, the library calls a user defined callback function and passes the request to the user application code.

All the SETUP requests are processed with a state machine. Two models are implemented to run this state machine, the polling model and the interrupt model.

As stated previously in *Section 1.2.1: USB device*, an interrupt is generated at the end of the correct USB transfer. The library code receives this interrupt. In the interrupt process routine, the trigger endpoint is identified. If the event is Correct Transfer on endpoint 0, the state machine is started with the token received in the interrupt model immediately, or the token received is saved and the state machine is started from the main loop in the polling model afterward.

The advantage of the interrupt model is that the device answers the host requests to the control endpoint in the fastest way which improves the overall performance of the whole USB system. The disadvantage of the interrupt model is that the MCU global interrupt will be disabled for a slightly longer time and the other application defined interrupt events have to wait for the USB process finishes to be served. Some programming restrictions are required in the interrupt model because of the limitation of the particular compiler.

On the other hand, the advantage and the disadvantage of the polling model are reversed. The running of the USB state machine would not interfere with the other part of the application. The application developer has the freedom to run the USB state machine at the time when he/she wants. In the polling model, the library responds to the USB requests quite slower than running in the interrupt model. This may slow down the bus in a fast system or in systems with a lot of transactions on the control endpoint.

In the polling model, USB_Polling () is used to run the USB state machine. The user code has to call this function from his/her main (see app_main () function) loop. The USB model is selected in the configuration file (mcu_conf.h).

2.2 Transactions on noncontrol endpoints

The user application uses noncontrol endpoints by calling a set of functions to send or receive data. The user application calls a function to pass the data buffer that contains the data or will receive the data from the library and calls another function to inquire if the transaction is finished. This inquiry checks an internal flag that represents the status of the transceiver.

The interrupt model and the polling model are implemented for the noncontrol endpoints too. The flag of the transceiver status is set in the interrupt routine as soon as the transaction is done in the interrupt model, but the flag is set in the USB polling routine if the polling model is selected. Obviously, the interrupt model improves overall system performance.

2.3 Special note on MCU interrupt

The USB events are very special versus other events in the MCU application. Interrupt of the USB events has to be answered as soon as possible, otherwise, the performance of the USB system would be decreased and some USB events may be lost.

The library configures the MCU to work on nested interrupt mode and set the USB interrupt as the highest interrupt level, so that the USB interrupt will be served in the fastest way.

In the interrupt model, the USB interrupt is configured as interrupt level 3 and the USB state machine will be running in interrupt level 2. All the other interrupt sources are configured as interrupt level 1.

In the polling model, the USB interrupt is configured as interrupt level 3 and the USB state machine will be running in interrupt level 1. All the other interrupt sources are configured as interrupt level 1. The user code can change the non-USB interrupt to level 2.

The user application has to be aware of this special configuration when using the interrupt peripheral of the MCU.

3 The USB state machine of the control endpoint

3.1 The states

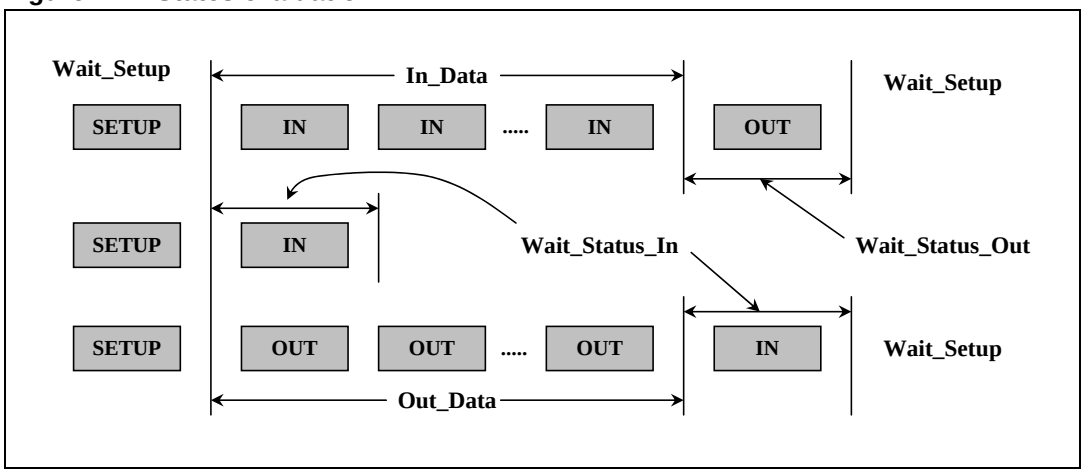
The USB state machine implemented in the library is used to handle the requests on the control endpoint only. The state machine does not process any transactions of noncontrol endpoints.

The USB state machine is designed to execute the SETUP requests. A SETUP request consists of three stages: SETUP stage, data stage and status stage. The states of the state machine are designed to match these three stages.

Table 4. States of USB state machine

State	Meaning
Wait_Setup	The state machine is in idle.
In_Data	The state machine is in data IN stage and is waiting for an IN data packet.
Out_Data	The state machine is in data OUT stage and is waiting to send an OUT data packet.
One_More_In	The state machine is in data IN stage and is waiting for a zero-length IN data packet when the length of the data stream in the data stage is multiple of the maximum packet size of the endpoint.
One_More_Out	The state machine is in data OUT stage and is waiting for sending a zero-length OUT data packet when the length of the data stream in the data stage is a multiple of the maximum packet size of the endpoint.
Wait_Status_In	The state machine is waiting for a status IN stage after the data OUT stage is finished, or after the SETUP stage if there is no data stage.
Wait_Status_Out	The state machine is waiting for a status OUT stage after the data IN stage is finished.
Address2Set	This is a special state to identify the process of a Set Address request.
State_Error	An error is detected, such as an invalid request.

Figure 1. States evaluation



3.2 Data structure for the SETUP packet

When a new SETUP packet arrives, all eight bytes of the SETUP packet are copied to an internal structure sUSB_vSetup, so that the next SETUP packet does not overwrite the previous one during processing. This internal structure is defined as:

```
typedef struct USB_vSetup {
   unsigned char  USBbmRequestType;
   unsigned char  USBbRequest;
   WORD_BYTE  USBwValues;
   WORD_BYTE  USBwIndexes;
   WORD_BYTE  USBwLengthes;
   unsigned char Flag;
} _USB_VSETUP;
_USB_VSETUP sUSB_vSetup;
```

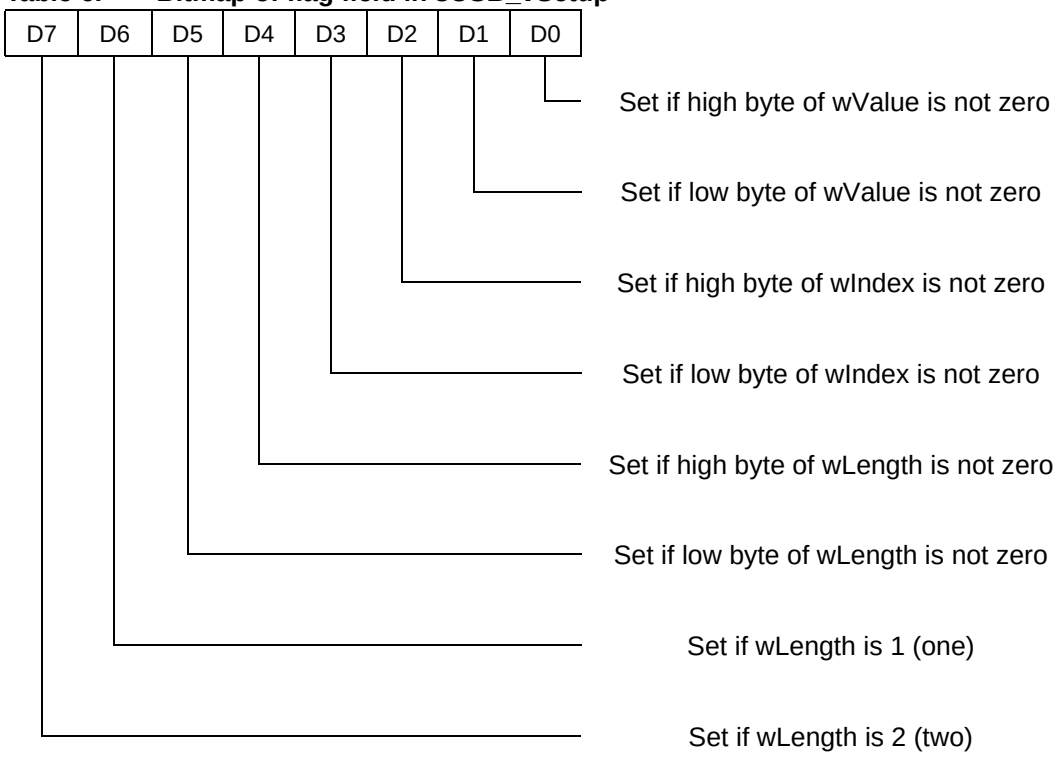
This structure is used in the library and is used in the user code of the USB callback functions. Referring to Table 9-2 of the USB specification [*1*], each field of the structure corresponds to a SETUP field.

Table 5. SETUP packet fields

Field in SETUP packet	Software name with the defined structure
bmRequestType	sUSB_vSetup.USBbmRequestType
bRequest	sUSB_vSetup.USBbRequest
wValue	sUSB_vSetup.USBwValue
High byte of wValue	sUSB_vSetup.USBwValue1
Low byte of wValue	sUSB_vSetup.USBwValue0
wIndex	sUSB_vSetup.USBwIndex
High byte of wIndex	sUSB_vSetup.USBwIndex1
Low byte of wIndex	sUSB_vSetup.USBwIndex0
wLength	sUSB_vSetup.USBwLength
High byte of wLength	sUSB_vSetup.USBwLength1
Low byte of wLength	sUSB_vSetup.USBwLength0

The bitmap of the flag field in the structure is designed to shorten the code size. *Table 6* shows the corresponding meaning in each bit and it is set during the copy of the SETUP packet.

Table 6. Bitmap of flag field in sUSB_vSetup



3.3 Standard requests

Most of the requests specified in Table 9-3 of the USB specification [1] are treated as standard requests in the library. *Table 7* lists all the standard requests and their valid parameters in the library. Requests that are not in the table below are considered as nonstandard requests.

Table 7. Standard requests

	State	bmRequest Type	Low byte of *wValue*	High byte of *wValue*	Low byte of *wIndex*	High byte of *wIndex*	wLength	Comments
GET_STATUS	A, C	80	00	00	00	00	2	Get Status of the Device.
	C	81	00	00	N	00	2	Get Status of Interface, N is the valid interface number.
	A, C	82	00	00	00	00	2	Get Status of Endpoint 0 OUT direction.
	A, C	82	00	00	80	00	2	Get Status of Endpoint 0 IN direction.
	C	82	00	00	EP	00	2	Get Status of Endpoint EP.

Table 7. Standard requests (continued)

	State	bmRequest Type	Low byte of *wValue*	High byte of *wValue*	Low byte of *wIndex*	High byte of *wIndex*	wLength	Comments
CLEAR_FEATURE	A, C	00	01	00	00	00	00	Clear the device remote wake-up feature.
	C	02	00	00	EP	00	00	Clear STALL condition of endpoint EP. EP does not refer to endpoint 0.
SET_FEATURE	A, C	00	01	00	00	00	00	Set the device remote wake-up feature.
	C	02	00	00	EP	00	00	Set STALL condition of endpoint EP. EP does not refer to endpoint 0.
SET_ADDRESS	D, A	00	N	00	00	00	00	Set the device address, N is the valid device address.
GET_DESCRIPTOR	All	80	00	01	00	00	Non-0	Get the device descriptor.
	All	80	N	02	00	00	Non-0	Get the configuration descriptor; N is the valid configuration index.
	All	80	N	03	LangID		Non-0	Get the string descriptor; N is the valid string index. This request is valid only when the string descriptor is supported.
GET_CONFIGURATION	A, C	80	00	00	00	00	1	Get the device configuration.
SET_CONFIGURATION	A, C	80	N	00	00	00	00	Set the device configuration; N is the valid configuration number.
GET_INTERFACE	C	81	00	00	N	00	1	Get alternate setting of the interface N; N is the valid interface number.
SET_INTERFACE	C	01	M	00	N	00	00	Set alternate setting M of the interface N; N is the valid interface number and M is the valid alternate setting of the interface N.

Note: Letters in the column "State": D=Default state; A=Address state; C=Configured state; All=All EP states.

EP value: D0-D3=endpoint address; D4-D6=reserved as zero; D7= 0: OUT endpoint, 1: IN endpoint.

All the nonstandard requests are passed to the user application code by means of a callback function. Every project that uses this library has to implement a callback function to receive nonstandard requests and return with success or with error. *Section 3.4* discusses the callback function.

3.4 Nonstandard requests

3.4.1 SETUP stage

The library passes all nonstandard requests to the user code with the callback USER_USB_Setup (). The nonstandard requests include the user-interpreted requests and the invalid requests. User-interpreted requests are class specific requests, vendor specific requests or the requests that the library considers as an invalid request but the application wants to interpret them as valid requests (for example, the library does not support the Halt feature on endpoint 0 but the user application may need this feature.) Invalid requests are the requests that are not standard requests and are not user-interpreted requests.

Since USER_USB_Setup () is called after the SETUP stage and before the data stage (*Section 3.1*), the user code is responsible, in the USER_USB_Setup (), to parse the content of the SETUP packet (sUSB_vSetup). If a request is invalid request, the user code has to call RequestError () and return to the caller of USER_USB_Setup ().

For a user-interpreted request, the user code prepares the data buffer for the following data stage if the request has a data stage, otherwise the user code executes the request and returns to the caller of USER_USB_Setup ().

3.4.2 Data stage

The process of the data stage has three categories: data IN stage with continued data buffer, data IN stage with noncontinued data buffer and data OUT stage.

Data IN stage with continued data buffer

The user code should perform the operation given in this section if a request needs a data IN stage and all data bytes that are going to be sent to the host are saved in a single and continued data buffer. Examples of requests in this category are: GET_REPORT & GET_DESCRIPTOR request of HID class .

In the USER_USB_Setup (), once the request is identified and all the data bytes are ready in the buffer, the user code assigns the buffer pointer to the variable vUSB_DataToCopy and assigns the number of bytes to be sent to the variable vUSB_length. The library will send the data from the buffer without the user code intervention again.

Data IN stage with noncontinued data buffer

It is easy and convenient to save all data bytes in one single buffer, but in some applications, because of the limitation of the buffer RAM size or that required by the application, all data bytes cannot be saved in one single buffer. For example, if the application wants to send a

big chunk of data (larger than the MCU RAM size) located in an external memory, then the data stream has to be split into small pieces and read into RAM piece by piece.

In the USER_USB_Setup (), the user code should assign the variable vUSB_DataToCopy as NULL (char *0) and the variable vUSB_length as the size of the data stream. Then, when the library is going to send a data packet of the data IN stage, it calls a callback function USER_USB_CopydataIN (unsigned char CopyLength). In this function, the user code should copy CopyLength bytes of data from its own buffer to the endpoint DMA buffer EP0_IN. The variable vUSB_offset indicates where to start copying the data.

In this way, the user code can generate the data stream, for example reports of GET REPORT request, in the DMA buffer directly and dynamically.

Data OUT stage

When a data OUT stage is necessary, the user code should assign the number of bytes that is going to receive to the variable vUSB_length in the USER_USB_Setup () and return to the caller of USER_USB_Setup (). Every time the library receives a data packet, it then calls callback USER_USB_CopydataOUT (unsigned char CopyLength). In the function USER_USB_CopydataOUT (), the user code has to copy the data to its own buffer from the endpoint DMA buffer EP0_OUT. The variable vUSB_offset indicates the offset of this data packet in the data stream. Of course, the user code can process the received data in the endpoint buffer EP0_OUT directly if the data stream fits in the buffer.

Before returning to the caller of USER_USB_CopydataOUT (), the library will not go on to receive the next data packet or to the status stage. Please do not stay in the function USER_USB_CopydataOUT () for a long time as it will slow down the performance of the USB system.

3.4.3 Status stage

The status stage of a control endpoint identifies the end of a SETUP transaction. To the user application, the status stage means that all data bytes have been sent/received. It is time to release the data buffer and prepare another data stream for the next data IN stage or to process the received data and prepare a new buffer for the next data OUT stage.

The library calls callback function USER_USB_Status_In() after the host acknowledges a status IN stage and calls callback function USER_USB_Status_Out() after it acknowledges a status OUT stage. In these two callback functions, the user code can call RequestError() if the request is an invalid request or there is any error in the data stage (for example received invalid data or failed to collect data to send).

3.5 Processing of standard requests

After the SETUP packet is copied into sUSB_vSetup, the request and its parameters are parsed. Standard requests defined in *Section 3.3* are handled in the library with less user code intervention. The following sections will discuss how the library processes each standard request and when the user code will be involved.

3.5.1 Get status

Recipient of the device

The library exports a variable, vUSB_Current_Feature. This variable is defined as: unsigned char vUSB_Current_Feature;

The meaning of this variable is:

- D7: Reserved as one
- D6: Power feature; 1=Self-Powered, 0=Bus-Powered
- D5: Remote-wake-up: 1=Support, 0=Not support
- D4-D2: Reserved as zero
- D1: Current Remote-wakeup: 1=Enable, 0=Disable
- D0: Current power feature: 1=Self-Powered, 0=Bus-Powered

This variable has the features of a device. The library requires that the user code copies the bmAttributes field of the configuration descriptor to this variable in the USER_USB_Set_Configuration() callback function.

With the help of the above variable, the library answers the request without any callback to the user code.

Recipient of the interface

In the configured state, the library ensures that the interface index is valid by checking the low byte of wIndex with the number of interfaces (vUSB_Num_Interface) given by the user code. The library replies with two bytes of zero without any callback to the user code if the interface index is valid.

Recipient of the endpoint

The library checks if the endpoint referred to is valid. In the address state, the valid endpoint is endpoint address 0 on both directions only. In the configured state, the endpoint is valid if its corresponding status register is not set as DISABLE (*Section 1.2.1*).

The library answers the request without the user code intervention if the endpoint referred to is valid.

3.5.2 Clear feature and set feature

3.5.3 Recipient of device

The valid requests are set/clear the remote wakeup feature. The library sets or clears the feature if the feature is supported (bit 5 of vUSB_Current_Feature is 1, see *Section 3.5.1: Get status*)

The library answers the request without any callback to the user code.

Recipient of interface

The USB specification [*1*] specifies that this request is valid in the configured state but the behavior is not specified. The library checks if the request parameters are valid and passes the control to the user code. It is up to the user code to decide what to do.

Recipient of endpoint

The library does not support the Halt feature for the endpoint 0. In the configured state, the library checks if status register of the specified endpoint is not set as DISABLE (see *Section 1.2.1*). Then the endpoint is set to STALL status (Set Feature) or is set to NAK status if it was in the STALL status (Clear Feature.)

After the status of the endpoint is changed correctly, the user code implemented callback function USER_USB_Clear_Feature_EP() is called for the Clear Feature request and the USER_USB_Set_Feature_EP() is called for the Set Feature request. In USER_USB_Clear_Feature_EP(), the user code knows that a STALL condition is cleared for certain endpoint and it is up to the user code to set the endpoint to ACK or stay in NAK status. In USER_USB_Set_Feature_EP(), the user code knows that a STALL condition is set for certain endpoint so that the state machine of that endpoint can be paused or stopped.

3.5.4 Set address

The library checks the parameters of the request and set the device to the new address if the request is valid. There is no user code callback for this request.

3.5.5 Get descriptor

A structure is defined for the user code to pass the descriptors to the library.

```
typedef struct OneDescriptor {
  char *Descriptor;
  unsigned short Size;
} ONE_DESCRIPTOR;
```

The field Descriptor is the pointer that points to the descriptor and the field Size gives the length of the descriptor.

An Application Descriptor table (Appli_Desc_Tab) is defined in the user area to keep track of descriptors. The number of string descriptors in the table has to be configured by the user through NUM_APP_STR_DESC macro (descriptor.h file) and the descriptor table has to be changed accordingly. This descriptor table is accessed through a pointer variable (DescTabPtr) which is present at a fixed location in the RAM. The user has to pass the address of the table to this pointer (DescTabPtr) in his main (App_main) function. The rest will be taken care of by the library.

For StringDescriptors, the 3rd descriptor in the descriptor table must list all the language ID supported. The remaining string descriptors will follow language ID descriptor in the descriptor table.

Limitation: The library supports the language ID of US English (0x0409) only, so the element 0 of StringDescriptor should list one language ID of US English only.

Constant descriptors

A descriptor is constant if the descriptor is stored in a single continued data buffer when it is going to be sent to the host. With the constant descriptors, the application initializes the Descriptor table pointer with the table address. The library will send the constant descriptors when they are required.

Nonconstant descriptors

Some descriptors may not be constant and they can be generated by the user code or can be read from outside of the MCU (such as an E2PROM.) For these descriptors, the corresponding pointer variable defined in the above section should be set as value zero. If a pointer is NULL, the library passes the request to the user code through the callback function USER_USB_Setup(). In USER_USB_Setup(), the user code has to identify the request by checking the parameters in sUSB_vSetup and performs the actions described in *Data IN stage with noncontinued data buffer*.

Standard request with constant descriptors

For the request of device descriptor, the library replies with the data referred by the DeviceDescriptor.

For the requests of the configuration descriptor, the library checks the index of configuration descriptor and then replies with the data from the corresponding index in the ConfigDescriptor[] array.

For the requests of configuration descriptors, the library checks the configuration index that is smaller than Num_Configuration given by the user code. Then the library replies with the data referred by the array of ConfigDescriptor[] accordingly.

For the requests of string descriptors, the library checks that the language ID is 0x0409 (US English) and that the string index is smaller than NUM_APP_STR_DESC given by the user code. Then the library replies with the data referred by the array of Appli_Desc_Tab[ ] accordingly.

For the requests that are not one of the above discussed, the library calls callback function USER_USB_Setup() to pass the request to the user code. This gives the user code an opportunity to answer, for example, the HID descriptor of an HID device.

3.5.6 Set configuration and get configuration

A variable, vUSB_Configuration, is defined in the library to keep track of the current configuration value. The content of this variable is returned to answer the Get Configuration request.

In the process of Set Configuration request, the library checks the configuration value versus the user variable Num_Configuration to ensure a valid configuration value is set. Then the library calls a callback function USER_Set_Configuration() to let the user code configure the device. In the USER_Set_Configuration(), the user code has to check the value of vUSB_Configuration and configure the device to the specified configuration correspondingly. The user code returns REQ_ERROR if the configuration value is invalid or the device cannot be configured correctly, otherwise the user code returns a value of REQ_SUCCESS.

Note: When the device is unconfigured (vUSB_Configuration is zero), the user code has to set the status register of all non-0 endpoints to DISABLE state. In any configured state, the user code has to keep the status register of all unused endpoints in DISABLE state. This requirement ensures that the Set Feature and the Clear Feature requests respond correctly.

3.5.7 Set interface and get interface

The library checks the parameters of the request and calls one of the two callback functions USER_USB_Set_Interface() or USER_USB_Get_Interface().

For the Set Interface request, the user code in the USER_USB_Set_Interface() should take the variable sUSB_vSetup.USBwIndex0 as the interface index and the variable sUSB_vSetup.USBwValue0 as the alternative setting value. The user code validates these two variables and configures the specified interface accordingly. The function USER_USB_Set_Interface() returns REQ_ERROR if there is any error, otherwise returns REQ_SUCCESS.

For the Get Interface request, the user code in the function USER_USB_Get_Interface() should take the variable sUSB_vSetup.USBwIndex0 as the interface index and returns the alternative setting value of the specified interface if there is no error. The user code calls RequestError() if there is any error.

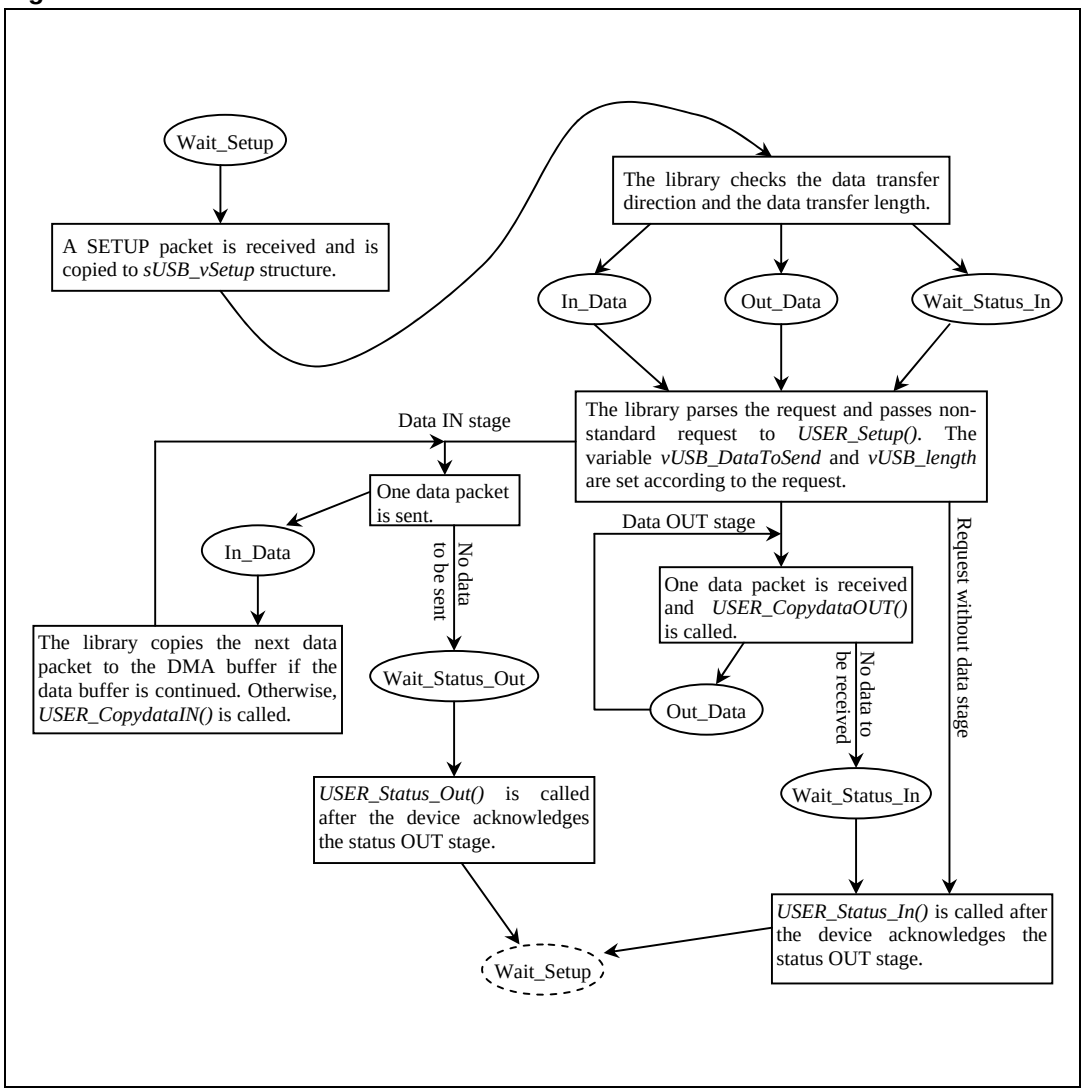
3.5.8 Status stage of the standard requests

As specified in *Section 3.4.3*, the callback functions USER_USB_Status_In() and USER_USB_Status_Out() will be called respectively at the end of all the standard requests except Set Address request. In these two callback functions, the user code is able to know what request is processed in the library.

3.6 The execution of the state machine

The state machine is in the state Wait_Setup when it is in idle. Once a SETUP packet is received, the request is copied to the sUSB_vSetup structure and the field sUSB_vSetup.flag is evaluated. Then the state machine is started. *Figure 2* illustrates the execution of the state machine. The state machine returns to the idle state (Wait_Setup) after USER_USB_Status_In() or USER_USB_Status_Out() returns.As stated in *Section 2.1*, the state machine of the control endpoint has two execution models: the polling model and the interrupt model. Since all callback functions specified in this chapter are part of the state machine, they can be part of the interrupt process routine if the state machine is working in the interrupt model. The application developers have to be aware that there are some limitations concerning writing code of the interrupt routines. This topic is not the scope of this document; please refer to other documents for this issue.

Figure 2. State machine execution



In summary, the callback function USER_USB_Setup() opens the door for the user code to parse the class specific or vendor specific requests or the Get Descriptor requests for nonconstant descriptors. The callback functions USER_USB_CopydataIN() and USER_USB_CopydataOUT() transfer data for those requests. The callback functions USER_USB_Status_In() and USER_USB_Status_Out() inform the user code that the process of a SETUP request is finished.

Note: For a complete list of callback functions please refer to Section 7.3.

4 Data transfer on noncontrol endpoints

Noncontrol endpoints are identified by the endpoint address and the endpoint direction, for example endpoint 2 IN and endpoint 2 OUT are two different endpoints. The endpoint of IN direction is used to send data from the device to the host. It can be either a bulk endpoint or an interrupt endpoint. The endpoint of OUT direction is used to receive data from the host to the device. It can be either a bulk endpoint or an interrupt endpoint too.

The library provides a set of functions for transferring data along endpoints of both directions. Each endpoint of either direction has two functions.

4.1 Sending data to the host

There are two functions for sending data to an endpoint.

```
char USB_SendDataEP?( unsigned char *DataAddress,
                      unsigned char LengthToXmit);
char USB_EP?_isSent(void);
```

Where? is a digit that represents the endpoint address. The digit ranges from 1 to 5 for ST7SCR, and the digit is 1 or 2 for ST7265. For example, the sending functions for endpoint 2 are USB_SendDataEP2() and USB_EP2_isSent().

The function USB_SendDataEP?() is used to start a sending procedure. The parameter DataAddress is a pointer which points to the data buffer and the parameter LengthToXmit gives the length of the data package. After the sending procedure is started, this function returns REQ_SUCCESS to the caller immediately. The function returns REQ_ERROR if the previous sending procedure is not finished or the specified endpoint is in the state of DISABLE or STALL. The user code should call function USB_EP?_isSent() some time later to enquire that the data transmission is finished. The function USB_EP?_isSent() returns a nonzero value if the data is sent, otherwise it returns a value zero. The user code has to wait that a sending procedure is finished before starting another sending procedure.

If the data package to be sent is longer than 255 bytes, the user code has to split the data package into a few small pieces that shorter than 255 bytes. The length of these few small data pieces, except the last data piece, have to be in the multiple of the maximum packet size of the used endpoint. Refer to *Section 1.2.1* for the maximum packet size of each endpoint.

4.2 Receiving data from the host

There are two functions for receiving data from an endpoint.

```
char USB_RecvDataEP?(unsigned char *DataAddress,
                     unsigned char Length);
unsigned char USB_TakeDataEP?(void);
```

Where? is a digit that represents the endpoint address. The digit is 1 or 2 for ST7265 and the digit is 2 for ST7SCR only. For example, the receiving functions for endpoint 2 are USB_RecvDataEP2() and USB_TakeDataEP2().

The function USB_RecvDataEP?() is used to start a receiving procedure. The parameter DataAddress is a pointer which points to the data buffer and the parameter Length gives the length of the data package. After the receiving procedure is started, this function returns REQ_SUCCESS to the caller immediately. The function returns REQ_ERROR if the previous receiving procedure is not finished or the specified endpoint is in the state of DISABLE or STALL. After starting the receiving procedure, the user code should call function USB_TakeDataEP?() some time later to enquire that the data package is received. The function USB_TakeDataEP?() returns 0xFF if the receiving procedure is not finished, otherwise it returns a value that represents the length of the valid data bytes in the buffer. The user code has to wait that a receiving procedure is finished before starting another receiving procedure.

If the data packet to be received is longer than 254 bytes, the user code has to split the data packet into smaller pieces that are shorter than 254 bytes. The length of these small data pieces, except the last data piece, must be a multiple of the maximum packet size of the endpoint used. Refer *Section 1.2.1* for the maximum packet size of each endpoint.

4.3 Handling noncontrol endpoints directly

The library provides the standard procedure to transfer data on the noncontrol endpoints. The user code can handle the data transfer on the noncontrol endpoints by access the endpoint status registers directly. The library defines a set of operations to manipulate the status registers.

4.3.1 Endpoint status registers

Refer to *Section 1.2.1*, each endpoint has four possible states and they are defined as EP_DISABLE, EP_STALL, EP_NAK and EP_ACK respectively.

For ST7265, there are four operations available to read status registers:

- USB_GetRx1Status() // Get the endpoint 1 receiving status
- USB_GetTx1Status() // Get the endpoint 1 sending status
- USB_GetRx2Status() // Get the endpoint 2 receiving status
- USB_GetTx2Status() // Get the endpoint 2 sending status

For ST7SCR, there are six operations available to read status registers:

- USB_GetTx1Status() // Get the endpoint 1 sending status
- USB_GetRx2Status() // Get the endpoint 2 receiving status
- USB_GetTx2Status() // Get the endpoint 2 sending status
- USB_GetTx3Status() // Get the endpoint 3 sending status
- USB_GetTx4Status() // Get the endpoint 4 sending status
- USB_GetTx5Status() // Get the endpoint 5 sending status

For ST7265, there are four operations available to set status registers:

- USB_SetTxEP1Status(Status) // Set the endpoint 1 sending status
- USB_SetRxEP1Status(Status) // Set the endpoint 1 receiving status
- USB_SetTxEP2Status(Status) // Set the endpoint 2 sending status
- USB_SetRxEP2Status(Status) // Set the endpoint 2 receiving status

For ST7SCR, there are six operations available to set status registers:

- USB_SetTxEP1Status(Status) // Set the endpoint 1 sending status
- USB_SetTxEP2Status(Status) // Set the endpoint 2 sending status
- USB_SetRxEP2Status(Status) // Set the endpoint 2 receiving status
- USB_SetTxEP3Status(Status) // Set the endpoint 3 sending status
- USB_SetTxEP4Status(Status) // Set the endpoint 4 sending status
- USB_SetTxEP5Status(Status) // Set the endpoint 5 sending status

In the above operations, Status is the state that will be set.

4.3.2 DMA counter registers

The sending DMA counter register for each endpoint is defined as CNTxTXR and the receiving DMA counter register for each endpoint is defined as CNTxRXR, where x represents the endpoint number.

4.3.3 DMA buffers

The DMA buffer for each sending endpoint is defined as EPx_IN and the DMA buffer for each receiving endpoint is defined as EPx_OUT, where x represents the endpoint number.

4.3.4 Sending data to the host

Follow the steps below to send data to the host:

1. Check if the endpoint status is EP_NAK to ensure that the previous data in the buffer is sent.
2. Copy the data to be sent to the DMA buffer of the endpoint.
3. Set the length of data to be sent to the corresponding endpoint DMA counter register.
4. Set the sending endpoint status to EP_VALID.
5. Check that the endpoint status becomes EP_NAK again to ensure that the data is sent.

4.3.5 Receiving data from the host

Follow the steps below to receive data from the host

1. Check if the endpoint status is EP_NAK to ensure that the EP is free to receive data.
2. Ensure that the data in the endpoint DMA buffer is useless to avoid the loss of useful data.
3. Set the length of data expected to the corresponding endpoint DMA counter register.
4. Set the receiving endpoint status to EP_VALID.
5. Check that the endpoint status becomes EP_NAK from EP_VALID.
6. Subtract the current value of the endpoint DMA counter register from the value set in step 2 to get the actual length of the received data.
7. Process the received data.

5 USB control functions for nondata transfer

The user application should be aware of a few other USB functions. Same as the data transfer function, these functions are part of the USB library.

5.1 Library initialization

To use the library certain initializations need to be done. If the USB support is needed in the application, then the application has to initialize the USB by calling the Init_USB function and the descriptor table pointer by the descriptor table address. The Init_USB() function initializes the USB state machine and the configuration state. This function also enables the Reset interrupt of the USB along with suspend and End of suspend interrupts. After calling this function, the library will be able to receive the reset from USB.

5.2 Device reset

When the device receives a reset request from the USB, the library initializes the endpoint 0 to acknowledge the SETUP token from host. It then calls the callback function USER_USB_Reset(). In this function, the user code should reset and initialize the application on both software and hardware. This function is part of the interrupt routine. The restriction of an interrupt routine is applied.

5.3 Start of frame

If the start of frame interrupt is enabled, the callback function USER_USB_SOF() is called every time a start of frame (SOF) token arrives. In this function, the user code can perform some timing related operations. This function is part of the interrupt routine. The restriction of an interrupt routine is applied.

Two functions are provided to enable and disable the interrupt of arrival of SOF:

```
void Enable_SOF(void);

void Disable_SOF(void);
```

6 Device firmware upgrade support

The library provides the support for the IAP which is used to change the application code (present in sector 1 and sector 2) through the USB without disturbing the code of sector 0. The Boot loader (USB + DFU) code of the library is placed in the sector 0. The user has to take care that his code is always placed in sector 1 and sector 2 and the code in sector 0 should not be disturbed. Use only those pragmas which are used in user code file or new defined pragmas. Don't use pragmas of bootloader. Refer to the lkf file for pragmas of each sector.

To upgrade the firmware the user has to go to DFU mode and after upgrading he has to return to the application mode. There can be two possible scenarios to go to DFU mode:

1. Composite mode: DFU Detach command is issued on the DFU interface to jump from application mode to DFU mode
2. Noncomposite mode: In this mode the user has to enter the DFU mode by means of checking the hardware switch after reset (e.g. Check the port status, if the pin level is low then boot in DFU mode, else go to the application mode). The current code is using PF3. If it's LOW, DFU mode is entered, otherwise it will boot in an application mode.

Firmware Upgrade: In the DFU mode, the user can upgrade the firmware through the PC GUI. For upgrading the firmware the user first needs to convert .s19 file into .dfu file through the GUI. Once converted, this dfu file will be used by the GUI to upgrade the firmware. For changing the code, the flash of microcontroller needs 12 V during erasing and writing. The supply for this 12 V should be available on the board (e.g. using charge pump ST662A).

Caution: 12 V should be supplied to the VPP pin of the microcontroller only during erasing or writing the flash.

7 The library package and its configuration

7.1 The contents of the package

The library is provided as a software package. All the source code files are included. *Table 8* lists all the files in the package.

Table 8. Code structure

File name	Comments	User changeable
Config\Metrowerks\map_7265.h Config\Metrowerks\map_7scr.h	Definition of hardware registers specific for each microcontroller. Use corresponding file while using Metrowerks compiler	No
Config\Metrowerks\map_7265.c Config\Metrowerks\map_7scr.c	Reserve the memory space for hardware registers.	No
Config\Cosmic\IO7265.h Config\Cosmic\IO7scr.h	Definition of hardware registers specific for each microcontroller. Use corresponding file while using Cosmic compiler.	No
Config\Cosmic\Vect_7265.c Config\Cosmic\Vect_7scr.c	Vector table files specific for each microcontroller.	No
Sources\main.c	Contain the main routine of the sample project.	Yes
Sources\mcu_conf.h	Contain the configuration information for configuring the library	Yes
Sources\USB\usb_lib.c	Code of control endpoint state machine.	No
Sources\USB\usb_int.c	Code to receive the USB interrupts.	No
Sources\USB\usb_ep1-5.c	Code related to the endpoint 1 to 5.	No
Sources\USB\usb_lib.h Sources\USB\usb_def.h Sources\USB\usb_reg.h Sources\USB\Vec_dec.h	Definitions of USB functions, structures, constants and macros that are used by the library. These three header files will be included in the user code files. Vec_dec contains declarations of vector functions	No
Sources\user\user_usb.c Sources\user\descriptor.c Sources\user\application.c Sources\user\user_usb.h Sources\user\descriptor.h Sources\user\appli.h Sources\user Int_7265.c	User Files. User need to write/integrate his/her code in/with these files.	Yes
Sources\HID\hidlayer.c Sources\HID\hidlayer.h	Files of a sample project.	Yes

Table 8. Code structure (continued)

File name	Comments	User changeable
Sources\generic\define.h Sources\generic\lib_bits.h	Generic files.	Yes
Source\DFU\dfu.c Source\DFU\dfu.h Source\DFU\DFU_Desc.c Source\DFU\DFU_Desc.h Source\DFU\Flashing.h	The Files for IAP.	No

7.2 Configuration of the library package

The file "mcu_conf.h" under directory "sources/" contains the information to configure the library.

One definition, USB_POLLING_MODEL, is used to select the model of execution of the control endpoint state machine. Please comment out the definition if the user code wants to use the interrupt running model, otherwise define it.

Two definitions (MCU_ST7SCR & MCU_ST7265) are used to select the product to be used. Please comment the unused definition lines. When selecting the MCU_ST7265, you also need to select MCU_ST72651 or MCU_ST72652 which have different code memory size.

After selecting the microcontroller, the user has to select the endpoint which the application is going to use, for example, enable macro DECLARE_EP1_IN if the user code wants to use the endpoint 1 IN. Similarly enable macro DECLARE_EP1_OUT to use the endpoint 1 in OUT direction. Enable the macro of DECLARE_EP2_IN and DECLARE_EP2_OUT if the user code wants to use the endpoint 2 in corresponding direction, and so on. The library uses these definitions to include or exclude the endpoint functions. For example, for using ST7265, open the workspace for this microcontroller. Customize the library for ST7265 by changing configuration in mcu_conf.h file. Select the microcontroller in that file and choose endpoints with proper directions which you want to support.

If the user code wants to transfer the data stream that is longer than 255 bytes along the control endpoint, the line to define the macro LARGE_EP0 has to be included in the code. One macro, DFU_ENABLE, is used to select the DFU feature of the library. If the user doesn't want to use DFU, just comment DFU_ENABLE macro. The DFU_COMPOSITE_DEVICE has to be enabled if the user wants to use DFU in composite mode. In Composite mode there are application interfaces and DFU interface exposed. When using in composite mode, then entry to DFU mode is done through the DFU Detach command issued on the DFU interface through the GUI. If the composite mode is not used, then the user has to make arrangements to enter the DFU mode (e.g. Check the port status during reset, if the pin level is low then boot in DFU mode, else go to the application mode).

Warning: In composite mode, the DFU driver will be loaded every time the device is plugged to a PC (not only during firmware upgrades) .

7.3 Application interface to the library

The application code has to provide some values to the library to execute the state machine. The library uses callback functions to perform the operation and all callback functions are implemented by direct function calls which are defined in user_usb.h. These callback functions are listed in *Table 9*.

Table 9. Call back functions

Function name	Description
USER_USB_Reset()	Called when a USB Reset is received by the device.
USER_USB_Setup()	All user-interpreted requests are parsed in this function.
USER_USB_CopydataIN()	Used to copy data to the endpoint DMA buffer EP0_IN for each data packet of data IN stage.
USER_USB_CopydataOUT()	Used to copy data from the endpoint DMA buffer EP0_OUT for each data packet of data OUT stage.
USER_USB_Status_In()	Called when a status IN stage is executed for all the requests except Set Address.
USER_USB_Status_Out()	Called when a status OUT stage is executed for all the requests.
USER_USB_Set_Configuration()	The user code has to configure the device as specified.
USER_USB_Set_Interface()	The user code has to set the alternative setting of an interface as specified.
USER_USB_Get_Interface()	The user code has to return the alternative setting of an interface as specified.
USER_USB_Clear_Feature_EP()	The user code is informed that the specified endpoint STALL condition is cleared.
USER_USB_Set_Feature_EP()	The user code is informed that the specified endpoint is set as STALL.
USER_USB_SOF()	Called every time a start of frame (SOF) token arrives on USB given SOF interrupt is enabled.
USER_USB_Suspend()	Called when suspend request is received by the device from the USB.
USER_USB_ESuspend()	Called when the device is in suspended state and activity on USB bus is detected

Four variables are there for the application to pass some configuration values to the library:

- USB_Num_Configuration: number of configuration
- USB_Num_Interface: number of interface
- USB_Interrupt_Mask: the value for the USB interrupt mask register
- DescTabPtr: pointer to actual descriptor table

These variables must be set in (or prior to) the "USER_USB_Reset" function.

8 Comments about the sample project

The sample project is built on the ST72651 product. The project demonstrates how to use the library in an application.

The polling model is selected to run the control endpoint state machine.

The project implements a USB HID (Human Interface Device) class to establish the communication of the board with the PC. HID requests such as Get Report descriptor, Get Report and Set Report, Get Idle and Set Idle, Get Feature and Set Feature are implemented on EP0 and these requests have the same meaning as requests in HID specification [2.] with the same name. All these requests are implemented in HIDLayer.c and .h file.

The application replies Get Descriptor (Report descriptor) request with the content in the array Report_Descriptor[].

The implementation of Get Descriptor, Get Idle, Get Output and Get Feature requests shows how to send data through a control endpoint upon the request and the implementation of Set Idle, Set Output and Set Feature requests shows how to receive data from the control endpoint.

The sample project also shows how to send and receive data on a noncontrol endpoint in file "application.c". In this sample, the data transfer on the EP2_IN and EP2_OUT is demonstrated. Please use the USB Demonstration GUI for the evaluation which is freely available on the internet.

The application also supports Loop Back feature. One macro, LOOP_BACK in appli.h is used to enable/disable this feature. If it is enabled, then sent data to the hardware will be received back from the hardware in the application mode. If it is disabled, then sent data will not be received back automatically by the GUI.

When USB Demonstration GUI is used and LOOP_BACK is not enabled, then pressing the LED button on the GUI allows the LED to glow on the hardware board present on PD0, otherwise pressing the LED button on the GUI status allows the button to glow on the GUI itself. A similar functionality can be tested for the scroll bar on the GUI.

9 References

1. "Universal Serial Bus Specification", Revision 1.1, 23 September 1998
2. "Device Class Definition for Human Interface Devices (HID)", version 1.0 Draft #4
3. "Universal Serial Bus Device Class Specification for Device Firmware Upgrade" (Version 1.1 Aug 5, 2004)
4. "ST7265 Datasheet" (version 3.0 Sep 10, 2006)
5. "ST7265 Mass Storage Evaluation Kit"

10 Revision history

Table 10. Document revision history

Date	Revision	Changes
04-Jan-2008	1	Initial release

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2008 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com