AVR276: USB Software Library for AT90USBxxx Microcontrollers

8-bit AVR® Microcontrollers

Application Note

Features

- **Low Speed (1.5Mbit/s) and Full Speed (12Mbit/s) data rates**
- **Control, Bulk, Isochronuous and Interrupt transfer types**
- **Up to 6 data endpoints/pipes**
- **Single or double buffering**
- **Device mode:**
  - **Standard or custom USB device classes with AVR USB software library**
- **Reduced host mode:**
  - **Host pipes auto configuration with device descriptors**
  - **Supports for composite devices (multiple interfaces)**

1. Description

This document describes the AT90USBxxx USB software library and illustrate how to develop USB device or reduced host applications using this library.

This document is written for the software developers to help in the development of their applications (both device or reduced host mode) for the AT90USBxxx. It assumes that readers are familiar with the AT90USBxxx architecture. A minimum knowledge of chapter 9 of USB specification 2.0 (www.usb.org) is also required to understand the content of this document.

1.1 Overview

The AT90USBxxx software library is designed to hide the complexity of USB development (and especially enumeration stage) from sofware designers.

The aim of this document is to describe the USB firmware and give an overview of the architecture. The main files are described in order to give the user the easiest way to customize the firmware and build his own application.

The AT90USBxx software library also provides a dual role (device or reduced host) application example (template demonstration sofware) to illustrate the usage of this library.

7675A–AVR–01/07

1.2 Limitations

- **Full support for OTG (On the Go) compliance (SRP/HNP requests management) not yet integrated**

1.3 Terminology

VID: USB Vendor Identifier

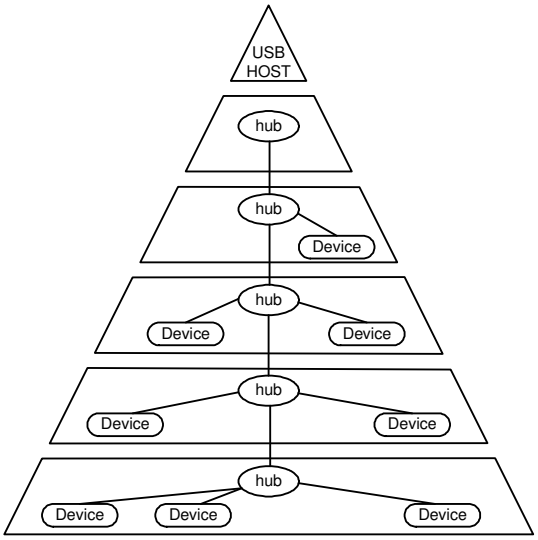
PID: USB Product Identifier

2. USB Bus

2.1 Bus topology

The USB specification defines two different types of nodes on the USB bus: host or device

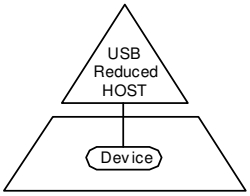
Figure 2-1. USB standard topology



- USB Host:
  - There is only one host in any USB system, and it operates as the “master” of the USB bus.
  - The USB interface to the host system is referred to as the Host Controller.
- USB Device:
  - A USB device operates a slave node on the USB bus.
  - Thanks to the USB hub (that also operates as USB device), up to 127 devices can be connected on the USB bus. Each device is uniquely identified by a device address.

AT90USBxxx parts can operate both as USB device or USB host, accurately in host mode AT90USBxxx operates as reduced host controller. A reduced host controller has a unique USB port and does not handle full USB tree with hub. It means that a reduced host controller is designed to handle a unique point to point connection with a unique USB device. A reduced host application supports a known targeted device list (VID/PID list). Only the devices listed within this list are supported by the application. In addition the AT90USBxxx USB software library is able to support a targeted list of CLASS/SUBCLASS/PROTOCOL.

Figure 2-2. Reduced host topology



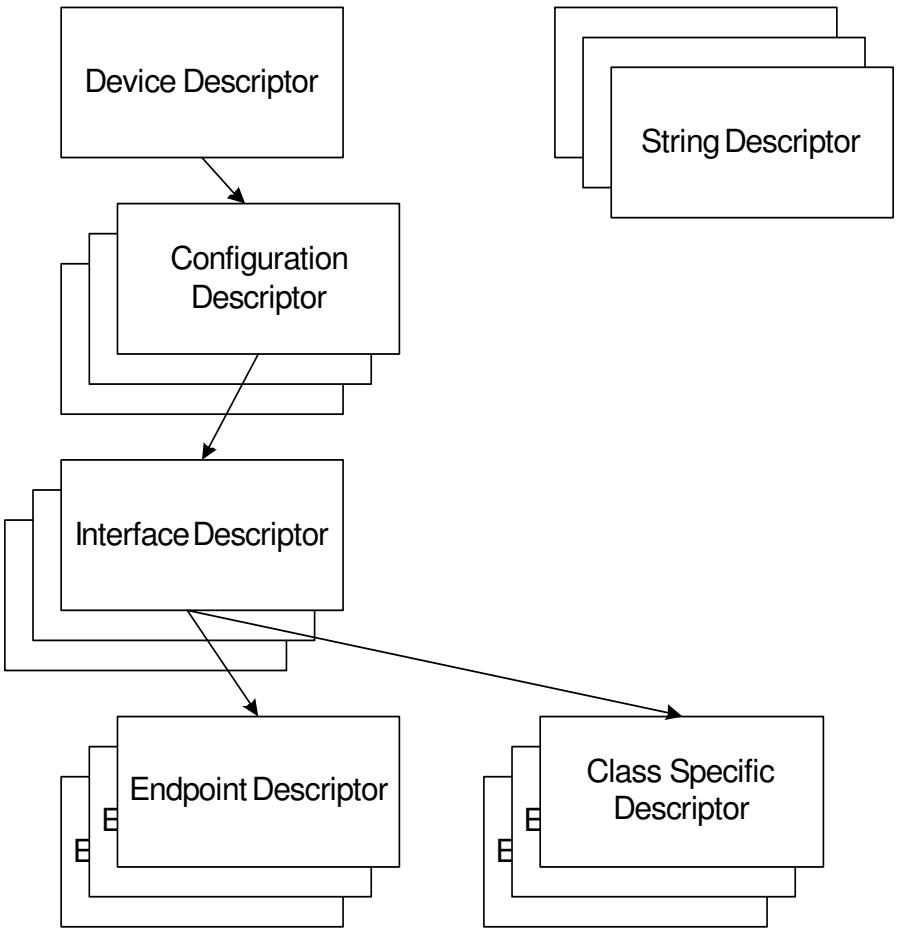
The AT90USBxxx software library can be configured to manage one of the following USB operating modes:

- USB device
- USB reduced host
- USB dual role device
  - This mode allows to support both previously defined modes. The operating mode is defined thanks to the external USB ID pin. Tied to ground (connected to a MiniA plug) the ID pin allows the AT90USBxxx to enter USB reduced host mode. If the ID pin is not connected (connected to a mini B plug) the AT90USBxxx operates in device mode.

2.2 USB Descriptors

During the enumeration process, the host asks the device several descriptor values to identify it and load the correct drivers. Each USB device should have at least the descriptors shown in the figure below to be recognized by the host:

Figure 2-3. USB Descriptors



The most difficult part of any USB application is determining what the device descriptors should be. Every USB device communicates its requirements to the host through a process called enu-

meration. The AT90USBxxx software library provides full enumeration process for both device or reduced host mode.

During enumeration, the device descriptors are transferred to the host which assigns a unique address to the device. The descriptors are described in detail in Chapter 9 of the USB 2.0 specification.

2.2.1 Device Descriptor

The USB device can have only one device descriptor. This descriptor displays the entire device. It gives information about the USB version, the maximum packet size of the endpoint 0, the vendor ID, the product ID, the product version, the number of the possible configurations the device can have, etc.

The table hereunder shows the format of this descriptor

Table 2-1. Device descriptor

Field	Description
bLength	Descriptor size
bDescriptorType	Device descriptor
bcdUSB	USB version
bDeviceClass	Code class (If 0, the class will be specified by each interface, if 0xFF, it is specified by the vendor)
bDeviceSubClass	Sub class code (assigned by USB org)
bDeviceProtocol	Code protocol (assigned by USB org)
bMaxPacketSize	The maximal packet size in bytes of the endpoint 0. It has to be 8 (Low Speed), 16, 32 or 64 (Full Speep)
idVendor	Vendor identification (assigned by USB org)
idProduct	Product identification (assigned by the manufacturer)
bcdDevice	Device version (assigned by the manufacturer)
iManufacturer	Index into a string array of the manufacturer descriptor
iProduct	Index into a string array of the product descriptor
iSerialNumber	Index into a string array of the serial number descriptor
bNumConfiguration	Number of configurations

2.2.2 Configuration Descriptor

The USB device can have more than one configuration descriptor, however the majority of devices use a single configuration. This descriptor specifies the power-supply mode (self_powered or bus-powered), the maximum power that can be consumed by the device, the interfaces belonging to the device, the total size of all the data descriptors , etc.

For example one device can have two configurations, one when it is powered by the bus and the other when it is self-powered. We can imagine also configurations which use different transfer modes.

The table hereunder shows the format of this descriptor:

Table 2-2. Configuration descriptor

Field	Description
bLength	Descriptor size
bDescriptor	Configuration descriptor
wTotalLength	Total descriptors size
bNuminterface	Number of interfaces
bConfigurationValue	Number of the configuration
bmAttributes	self-powered or bus-powered, remote wake up
bMaxpower	2mA by unit

2.2.3 Interface Descriptor

A single device can have more than one interface. The main information given by this descriptor is the number of endpoints used by this interface and the USB class and subclass.

The table hereunder shows the format of this descriptor:

Table 2-3. Interface descriptor

Field	Description
bLength	Descriptor size
bDescriptorType	Interface descriptor
bInterfaceNumber	interface number
bAltenativeSetting	Used to select the replacing interface
bNumEndpoint	Number of endpoints (excluding endpoint 0)
bInterfaceClass	Class code (assigned by USB org)
bInterfaceSubClass	Subclass code (assigned by USB org) 0 No subclass 1 Boot interface subclass
iInterface	Index into a string array to describe the used interface

2.2.4 Endpoint Descriptor

This descriptor is used to describe the endpoint parameters such as: the direction (IN or OUT), the transfer type supported (Interrupt, Bulk, Isochronuous), the size of the endpoint, the interval of data transfer in case of interrupt transfer mode, etc.

The table hereunder shows the format of this descriptor:

Table 2-4. Endpoint descriptor

Field	Description
bLength	Descriptor size
bDescriptorType	Endpoint descriptor
bEndpointAdress	Endpoint adress Bits[0..3] Number of the endpoint Bits[4..6] reserved, set to 0 Bit 7: Direction: 0 = OUT, 1 = IN
bmAttributes	Bits[0..1] Transfer type: 00=Control, 01=Isochronous, 10=Bulk, 11=Interrupt Bits [2..7] reserved, except for Isochronous transfer Only control and interrupt modes are allowed in Low Speed
wMaxPacketSize	The maximum data size that this endpoint support
bInterval	It is the time interval to request the data transfer of the endpoint. The value is given in number of frames (ms). Ignored by the bulk and control transfers. Set a value between 1and 16 (ms) for isochronous Set a value between 1 and 255 (ms) for the interrupt transfer in Full Speed Set a value between 10 and 255 (ms) for the interrupt transfer in Low Speed

3. Firmware Architecture

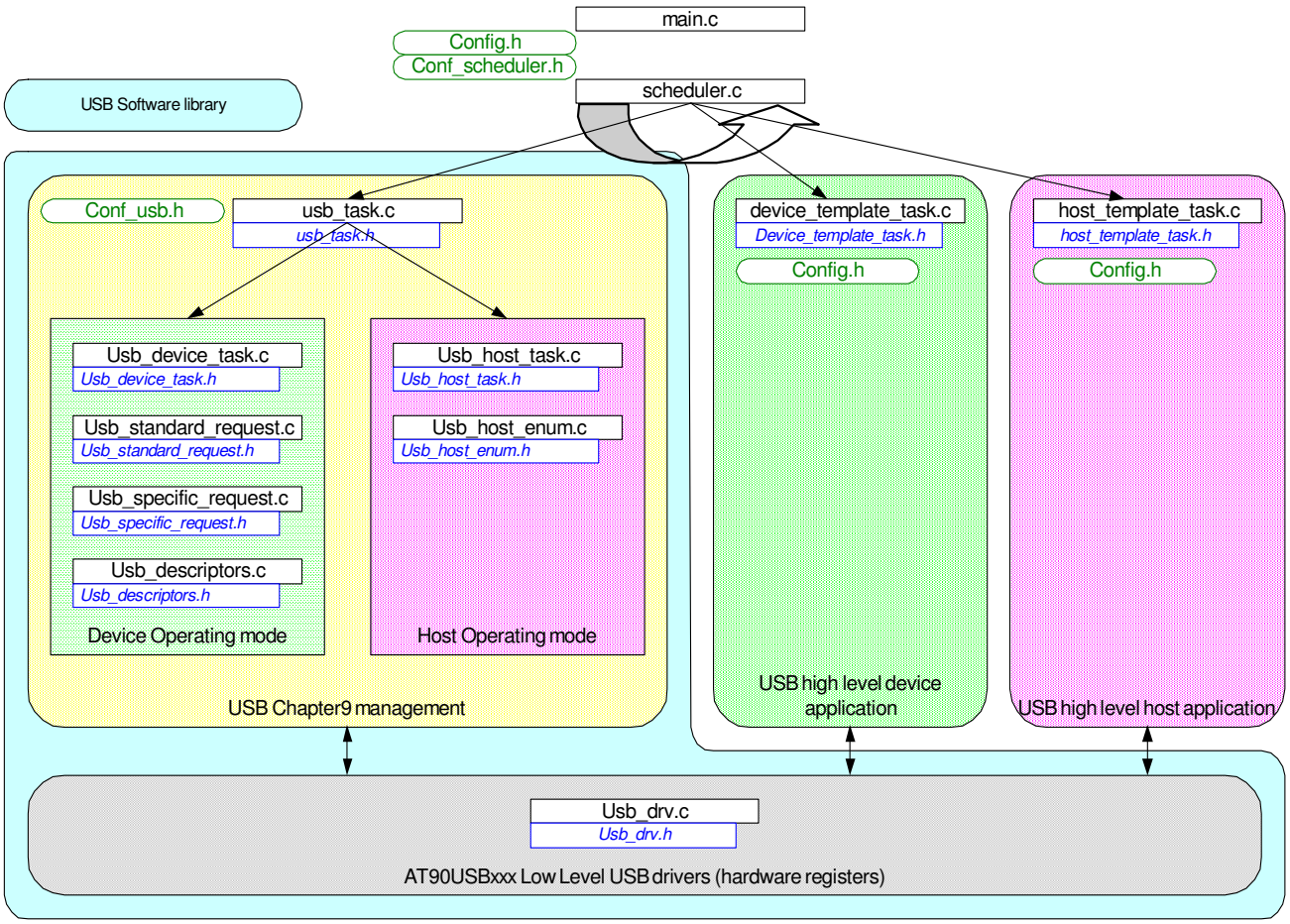
As shown in Figure 3-1, the architecture of the USB firmware is designed to avoid any hardware interfacing (drivers layer should not be modified by the user). The USB software library can manage both device or host USB chapter 9 enumeration process.

The global USB firmware architecture is illustrated thanks to a dual role sample application “template” that allows the AT90USBxxx to operate as a device or a host depending on the USB ID pin.

The sample dual role application is based on three different tasks:

- The usb_task (usb_task.c), is the task performing the USB low level enumeration process in device or host mode. Once this task has detected that the USB connection is fully operationnal, it updates different status flags that can be checked within the high level application tasks.
- The device template task (device_template_task.c) performs the high level device application operation. This task contains the USB device user application that can be executed once the device is enumerated.
- The host template task (host_template_task.c) performs the high level host application operation once the connected device is enumerated.

Figure 3-1. AT90USBxxx USB Firmware Architecture for dual role application



3.1 About the sample application

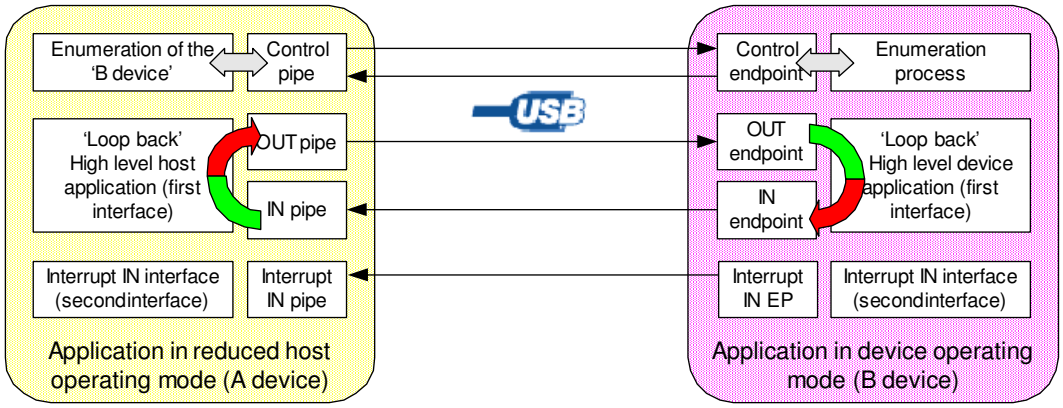
The sample application used to illustrate the USB software library can operate in both USB operating modes (device or host).

The device operating mode presents two interfaces:

- The first bullk IN/OUT interface operates in loop back data transfer (all data received on the OUT endpoint are sent back with the IN endpoint).
- The second interface transmits data with an interrupt IN endpoint.

The host operating mode of the sample application recognizes, enumerates and uses the both interfaces of the device application.

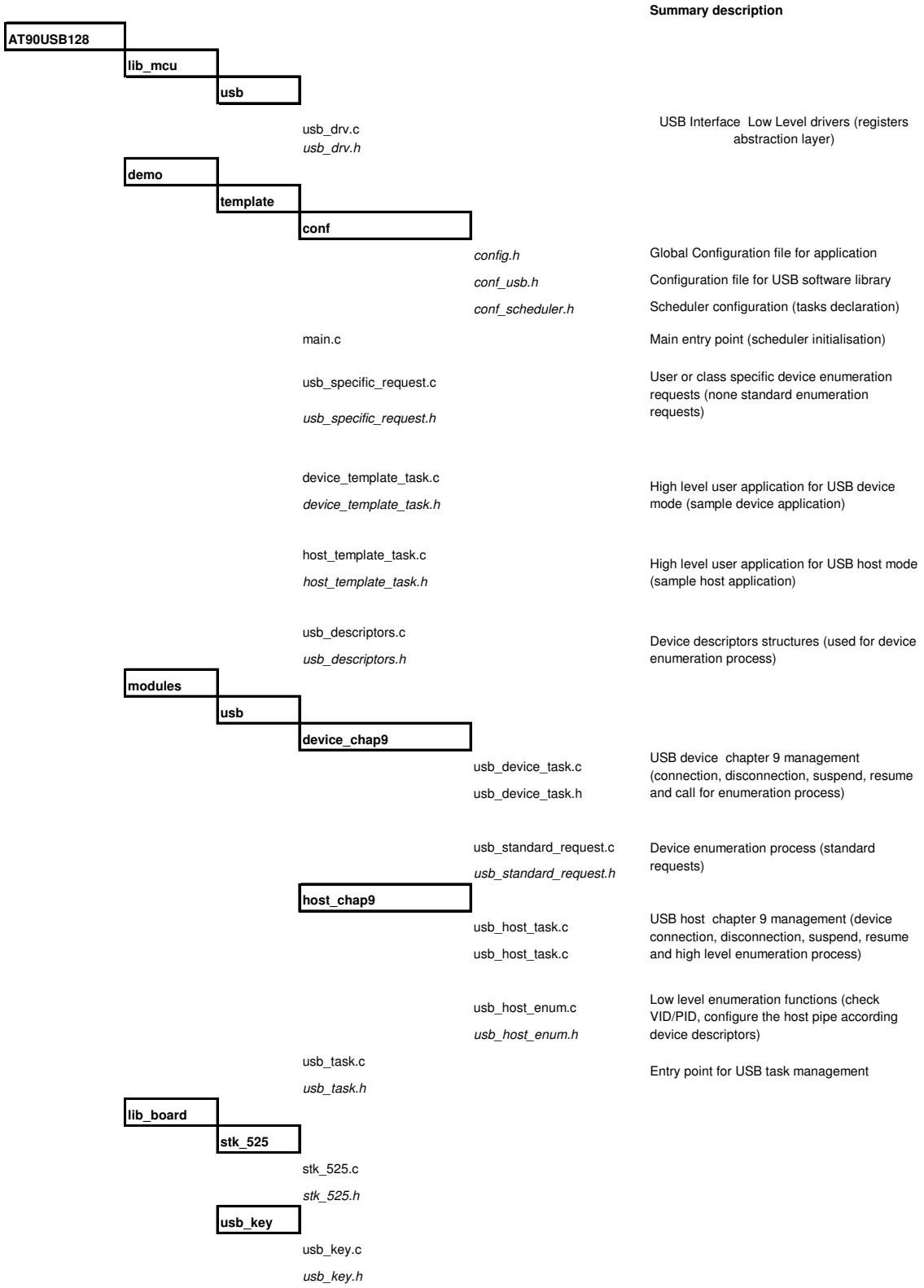
Figure 3-2. Sample application overview



Note: The B device descriptors used for this sample application are not directly usable for enumeration with a standard Pc host system. Please refer to Atmel website for "real" device applications examples (HID mouse, HID keyboard, MassStorage, CDC ...).

3.2 Source files architecture

Figure 3-3. Source files organisation



4. Configuring the USB software library

The sample application is pre-configured to implement both host and device functionalities. It can also be configured to be used in device or reduced host mode only. Depending on the USB operating mode selected, the USB task will call either the usb_host_task, or the USB device task to manage chapter 9 requests. In such case, the corresponding template_device_task or template_host_task can be removed from the scheduled tasks.

4.1 Global configuration

All common configuration parameters for the application are defined in the “config.h” file (XTAL frequency, CPU type...). Module specific parameters are defined in their respective configuration files.

4.2 Scheduler configuration

The sample application provides a simple tasks scheduler that allows the user to create and add applicative tasks without modifying the global application architecture and organisation. This scheduler calls all predefined tasks in a predefined order without any preemption. A task is executed until it is finished, then the scheduler calls the next task.

The scheduler tasks are defined in the “conf_scheduler.h” file, where the user can declare his tasks.

For the sample USB dual role application the following scheduler configuration parameters are used:

```
#define Scheduler_task_1_init   usb_task_init
#define Scheduler_task_1        usb_task
#define Scheduler_task_2_init   device_template_task_init
#define Scheduler_task_2        device_template_task
#define Scheduler_task_4_init   host_template_task_init
#define Scheduler_task_4        host_template_task
```

The scheduler_task_X_init functions are executed only once upon scheduler startup whereas the Scheduler_task_X functions are executed in a infinite loop.

4.3 USB library configuration

The USB library can be configured thanks to the “conf_usb.h” file. This file contains both USB modes configuration parameters for device and host. The configuration file is split into three sections for global, device and host configuration parameters.

The global configuration section allows to select if the library uses the device and/or host USB mode and if the internal USB pads regulator should be enable for the application (depending on the application power supply range).

4.4 Device configuration

The USB library manages USB chapter 9 for a B device:

- Connection/Disconnection (VBUS monitoring)
- Suspend
- Resume
- Enumeration requests

Asynchronous USB events (connection, suspend, resume, reset) are managed directly within the USB interrupt subroutine located in the “usb_tak.c” file. The user application can execute specific functions upon these events thanks to the user defined actions of the “conf_usb.h” file.

Enumeration requests from the host are managed in polling mode with the “usb_device_task.c” and “usb_standard_request.c files”. The usb_task that belongs to the scheduler tasks periodically check for new control requests from the host.

4.4.1 Configuring the USB library

To enable the USB device mode of the library the USB_DEVICE_FEATURE should be defined as ENABLED.

The device specific configuration section of “conf_usb.h” file contains the physical endpoints numbers definition used by the device application and a set of user specific actions that can be executed upon special events during the USB communication.

For the sample application:

```
#define NB_ENDPOINTS 4 //number of EP in the application including EP0
#define EP_TEMP_IN 1
#define EP_TEMP_OUT 2
#define EP_TEMP_INT_IN 3


// write here the action to associate to each USB event
// be carefull not to waste time in order not disturbing the functions
#define Usb_sof_action()         sof_action();
#define Usb_wake_up_action()
#define Usb_resume_action()
#define Usb_suspend_action()
#define Usb_reset_action()
#define Usb_vbus_on_action()
#define Usb_vbus_off_action()
#define Usb_set_configuration_action()
```

The user action defines the users high level application to execute specific functions. For example the user can map a function executed upon each USB start of frame event or USB bus reset.

4.4.2 About composite devices

A composite device allows a USB device to be detected as a multiple peripherals on the USB bus. Thus a composite device declares more than one interface in its configuration descriptor. Each interface has its own Class/SubClass/Protocol and associated high level application behavior (for example a composite device can operate as an HID interface in association with a device Mass Storage application).

The sample composite devices that illustrate the USB software library declares two differrent interfaces:

- A first interface with two bulk IN/bulk OUT endpoints.
- A second interface with a single interrupt IN endpoint.

4.4.3 Configuring the device descriptors

The device descriptors used for the device enumeration process are store in “usb_descriptors.c and usb_descriptors.h” files.

The descriptors structures type are declared in “usb_descriptors.h” file, the user should declare here all the enumeration parameters for his device configuration.

The configuration descriptor type is defined at the end of the “usb_descriptors.h” file:

```
// Configuration descriptor template
// The device has two interfaces
// - First interface has 2 bulk endpoints
// - Second interface has 1 interrupt IN endpoint
typedef struct
{
   S_usb_configuration_descriptor cfg_temp;
   S_usb_interface_descriptor     ifc_temp;
   S_usb_endpoint_descriptor      ep1_temp;
   S_usb_endpoint_descriptor      ep2_temp;
   S_usb_interface_descriptor     ifc_second_temp;
   S_usb_endpoint_descriptor      ep3_temp;
} S_usb_user_configuration_descriptor;
```

The associated interfaces and endpoints parameters are declared at the begining of the “usb_descriptors.h” file.

```
// USB Device descriptor
#define USB_SPECIFICATION     0x0200
#define DEVICE_CLASS          0      //! each configuration has its own class
#define DEVICE_SUB_CLASS      0      //! each configuration has its own sub-class
#define DEVICE_PROTOCOL       0      //! each configuration has its own protocol
#define EP_CONTROL_LENGTH     64
#define VENDOR_ID             0x03EB // Atmel vendor ID = 03EBh
#define PRODUCT_ID            0x0000
#define RELEASE_NUMBER        0x1000
#define MAN_INDEX             0x01
#define PROD_INDEX            0x02
#define SN_INDEX              0x03
#define NB_CONFIGURATION      1

               // CONFIGURATION
#define NB_INTERFACE       2     //! The number of interface for this configuration
#define CONF_NB            1     //! Number of this configuration
#define CONF_INDEX         0
#define CONF_ATTRIBUTES    USB_CONFIG_SELFPOWERED
#define MAX_POWER          50    // 100 mA (2mA unit !)
```

```
            // USB Interface descriptor gen
#define INTERFACE_NB_TEMP        0        //! The number of this interface
#define ALTERNATE_TEMP           0        //! The alt settting nb of this interface
#define NB_ENDPOINT_TEMP         2        //! The number of endpoints this this
interface have
#define INTERFACE_CLASS_TEMP     0x00     //! Class
#define INTERFACE_SUB_CLASS_TEMP 0x00     //! Sub Class
#define INTERFACE_PROTOCOL_TEMP  0x00     //! Protocol
#define INTERFACE_INDEX_TEMP     0

           // USB Endpoint 1 descriptor FS
#define ENDPOINT_NB_TEMP1       (EP_TEMP_IN | 0x80)
#define EP_ATTRIBUTES_TEMP1     0x02              // BULK = 0x02, INTERUPT = 0x03
#define EP_IN_LENGTH_TEMP1      64
#define EP_SIZE_TEMP1           EP_IN_LENGTH_TEMP1
#define EP_INTERVAL_TEMP1       0x00  // Interrupt polling interval from host

           // USB Endpoint 2 descriptor FS
#define ENDPOINT_NB_TEMP2       EP_TEMP_OUT
#define EP_ATTRIBUTES_TEMP2     0x02              // BULK = 0x02, INTERUPT = 0x03
#define EP_IN_LENGTH_TEMP2      64
#define EP_SIZE_TEMP2           EP_IN_LENGTH_TEMP2
#define EP_INTERVAL_TEMP2       0x00  // Interrupt polling interval from host

            // USB Second Interface descriptor gen
#define INTERFACE_NB_SECOND_TEMP        1        //! The number of this interface
#define ALTERNATE_SECOND_TEMP           0        //! The alt settting nb of this
interface
#define NB_ENDPOINT_SECOND_TEMP        1       //! The number of endpoints this this
interface have
#define INTERFACE_CLASS_SECOND_TEMP     0x00     //! Class
#define INTERFACE_SUB_CLASS_SECOND_TEMP 0x55     //! Sub Class
#define INTERFACE_PROTOCOL_SECOND_TEMP  0xAA     //! Protocol
#define INTERFACE_INDEX_SECOND_TEMP     0

           // USB Endpoint 2 descriptor FS
#define ENDPOINT_NB_TEMP3       (EP_TEMP_INT_IN | 0x80)
#define EP_ATTRIBUTES_TEMP3     0x03              // BULK = 0x02, INTERUPT = 0x03
#define EP_IN_LENGTH_TEMP3      64
#define EP_SIZE_TEMP3           EP_IN_LENGTH_TEMP2
#define EP_INTERVAL_TEMP3       20  // Interrupt polling interval from host
```

All these enumeration parameters are used to fill-up the descriptor fields declared in “usb_descriptors.c” file. When the host controller performs enumeration processs, its requests are decoded thanks to the “standard_request.c” enumeration functions and the predefined descriptors are sent to the host controller.

4.5 Reduced host configuration

The USB library manages USB chapter 9 for a reduced host controller:

- VBUS generation and monitoring
- Connection of the peripheral
- Disconnection of the peripheral

- Enumeration and identification of the connected peripheral
- Configuration of the host controller pipes according to the device descriptors of the connected peripheral
- Suspend the USB activity
- Resume and remote wake-up management

Similar to the device mode of the library, asynchronous USB events (connection, disconnection, remote wake-up detection) are managed directly within the USB interrupt subroutine located in the “usb_tak.c” file. The user application can execute specific functions upon these events thanks to the user defined actions of the “conf_usb.h” file.

The connected device state and its enumeration process are managed in polling mode with the “usb_host_task.c” and “usb_host_enum.c files”. Only critical and asynchronous device events such as device disconnection and remote wake up detection are managed under interrupt (“usb_task.c“ file).

To enable the USB host mode of the library the USB_HOST_FEATURE should be defined as ENABLED.

The host specific configuration section of “conf_usb.h” file allows to select the following main parameters:

- The VID/PID table of known devices supported by the host application
- The class/subclass/protocol table aof interfaces supported by the host application
- The maximum number of interfaces supported for a connected composite device
- The maximum number of endpoints associated to an interface
- The timeout parameters for a USB transfer (number of NAK or time delay)

Example for the sample application:

```
//!   This table contains the VID/PID that are supported by the reduced host application
   //!   VID_PID_TABLE format definition:
   //!
   //!   #define VID_PID_TABLE      {VID1, number_of_pid_for_this_VID1, PID11_value,...,
PID1X_Value \n
   //!                              ...\n
   //!                              ,VIDz, number_of_pid_for_this_VIDz, PIDz1_value,...,
PIDzX_Value}
   #define VID_PID_TABLE            {0x03EB, 2, 0x201C, 0x2014 \
                                    ,0x0123, 3, 0x2000, 0x2100, 0x1258}

   //!   @brief CLASS/SUBCLASS_PROTOCOL supported table list
   //!
   //!   This table contains the CLASS/SUBCLASS/PROTOCOL that is supported by the reduced host
application
   //!   This table definition allows to extended the reduced application device support to an
entire Class/
   //!   /subclass/protocol instead of a simple VID/PID table list.
   //!
   //!   CLASS_SUBCLASS_PROTOCOL format definition: \n
   //!   #define CLASS_SUBCLASS_PROTOCOL  {CLASS1, SUB_CLASS1,PROTOCOL1, \n
   //!                                     ...\n
   //!                                     CLASSz, SUB_CLASSz,PROTOCOLz}
   #define CLASS_SUBCLASS_PROTOCOL     {\
                                       0x00, 0x00, 0x00,\
                                       0x00,0x55,0xAA}

   //! The size of RAM buffer reserved of descriptors manipulation
   #define SIZEOF_DATA_STAGE        250

   //! The address that will be assigned to the connected device
```

```
#define DEVICE_ADDRESS           0x05

//! The maximum number of interface that can be supported (composite device)
#define MAX_INTERFACE_SUPPORTED  0x02

//! The maximum number of endpoints per interface supported
#define MAX_EP_PER_INTERFACE     3

//! The host controller will be limited to the strict VID/PID list.
//! When enabled, if the device PID/VID does not belongs  to the supported list,
//! the host controller library will not go to deeper configuration, but to error state.
#define HOST_STRICT_VID_PID_TABLE      DISABLE

//! Try to configure the host pipes according to the device descriptors received
#define HOST_AUTO_CFG_ENDPOINT         ENABLE

//! Host start of frame interrupt always enable
#define HOST_CONTINUOUS_SOF_INTERRUPT  DISABLE

//! When Host error state detected, goto unattached state
#define HOST_ERROR_RESTART             ENABLE

//! USB host pipes transfers use USB communication interrupt (allows to use none blocking
functions)
#define USB_HOST_PIPE_INTERRUPT_TRANSFER  ENABLE

//! Force WDT reset upon ID pin change
#define ID_PIN_CHANGE_GENERATE_RESET   ENABLE

//! Enable Timeout delay (time) for host transfer
#define TIMEOUT_DELAY_ENABLE           ENABLE

//! delay 1/4sec (250ms) before timeout value
#define TIMEOUT_DELAY                  1

//! Enable cpt NAK Timeout for host transfer
#define NAK_TIMEOUT_ENABLE          ENABLE

//! Number of NAK handshake before timeout for transmit functions (up to 0xFFFF)
#define NAK_SEND_TIMEOUT            0x0010

//! NAKNumber of NAK handshake before timeout for receive functions (up to 0xFFFF)
#define NAK_RECEIVE_TIMEOUT         0x0010

//! For reduced host only allows to control VBUS generator with PIO PE.7
#define SOFTWARE_VBUS_CTRL          ENABLE

#if (HOST_AUTO_CFG_ENDPOINT==FALSE)
   //! If no auto configuration of EP, map here user function
   #define        User_configure_endpoint()
#endif

//! @defgroup host_cst_actions USB host custom actions
//!
//! @{
// write here the action to associate to each USB host event
// be carefull not to waste time in order not disturbing the functions
#define Usb_id_transition_action()
#define Host_device_disconnection_action()
#define Host_device_connection_action()
#define Host_sof_action()
#define Host_suspend_action()
#define Host_hwup_action()
#define Host_device_not_supported_action()
#define Host_device_class_not_supported_action()
#define Host_device_supported_action()
#define Host_device_error_action()
//! @}
```

5. Using the USB software library within high level USB application

5.1 Device application

The device user application task knows that the device is properly enumerated thanks to the “Is_device_enumerated()” function that returns TRUE once the SET_CONFIGURATION request has been received from the host.

```
void device_template_task(void)
{
    //.. FIRST CHECK THE DEVICE ENUMERATION STATE
    if (Is_device_enumerated())
    {
       //.. HERE START THE USB DEVICE APPLICATIVE CODE
    }
}
```

The “device_template_task.c” file included in the sample application code illustrates how to use both bulk IN/OUT and interrupt endpoints associated to the interfaces declared in the configuration descriptors.

5.2 Host application

The host user application communicates with the host chapter 9 library thanks to event specific functions that allows:

- Detection of device connection/disconnection
- Get the main device characteristics (VID, PID, Class, SubClass, Protocol, Number of interfaces, Max power supply...)
- Suspend/resume the USB bus

In addition to USB chapter 9 management, the host library also provides a set of generic functions that allows to manage a bulk IN bullk OUT data flow (send and receive functions in both polling (blocking) mode or none blocking mode under interrupt).

5.2.1 Device detection

5.2.1.1 Device connection

The function “Is_new_device_connection_event()” returns TRUE, when a new device is enumerated and configured (Set Configuration request sent) with the USB host chapter 9 library.

5.2.1.2 Device disconnection

The function “Is_device_disconnection_event()” returns TRUE, when the previous configured device just disconnect from the USB host port.

5.2.2 Configured device characteristics

The host library provides a set of functions to get the USB characteristics of the configured device.

5.2.2.1 Generic information

- “Get_VID()”: returns the VID of the configured device.
- “Get_PID()”: returns the PID of the configured device.
- “Get_maxpower()” returns max power required for the configured device (2mA unit)
- “Is_device_self_powered()” returns true when the configured device is self powered.
- “Is_device_supports_remote_wakeup()” returns TRUE when the configured device supports remote wake up feature.
- “Is_host_full_speed()” returns TRUE when the configured device is connected in full speed mode. Connected to a low speed device, the function returns FALSE.

5.2.2.2 Interfaces, Endpoints

The host library supports devices with multiple interfaces declaration (composite devices) and can configure the host pipes according to the receive device descriptors.

The interface parameters of the configured device are stored in an array of structures which contains the interface characteristics.

```
S_interface interface_supported[MAX_INTERFACE_SUPPORTED]
// with
typedef struct
{
   U8  interface_nb;
   U8  altset_nb;
   U16 class;
   U16 subclass;
   U16 protocol;
   U8  nb_ep;
   U8  ep_addr[MAX_EP_PER_INTERFACE];
} S_interface;
```

The host library provides a set of functions to access these parameters:

- “Get_nb_supported_interface()” returns the number of supported interfaces configured for the connected device.
- “Get_class(interface)” returns the class of the specified interface.
- “Get_subclass(interface)” returns the subclass of the specified interface.
- “Get_protocol(interface)” returns the protocol of the specified interface.
- “Get_nb_ep(s_interface)” returns the number of endpoints associeted to the specified interface number.
- “Get_interface_number(s_interface)” returns device descriptor interface number for the specified supported interface.
- “Get_alts_s(s_interface)” returns the number of the alternate setting value for the specified interface.
- “Get_ep_addr(s_interface,n_ep)” returns the logical endpoint address associated to the specified interface and endpoint number.

The host library allows “on the fly” host pipe configuration. According to the device descriptors received, the host library allocates pipe associated to each device endpoint. A look up table (“ep_table”) allows to link a phisical host pipe number to logical endpoint address.

```
U8 ep_table[MAX_EP_NB];
```

The following functions can be used to get the crossreferences:

- “Get_ep_addr(s_interface,n_ep)” returns the logical endpoint address associated to the specified interface and endpoint number.
- “host_get_hwd_pipe_nb(ep_addr)” returns the physical pipe number associated to a logical endpoint address.

5.2.3 Bus activity management

5.2.3.1 Suspending the USB bus

Called from the user host application, the “Host_request_suspend()” function makes the USB bus enter suspend mode. If the configured device supports remote wake up, the USB host library will sent a “Set Feature Enable Remote Wake Up” request before entering suspend mode.

The “Is_host_suspended()” functions returns TRUE when the USB is in suspend state and may be used from the host high level application to detect the USB state.

5.2.3.2 Resuming the USB bus

The host application can resume the USB activity calling the “Host_request_resume()” function.

5.2.3.3 Remote Wake up

If the configured device supports remote wake up, this function allows to resume the USB host activity.

5.2.4 Data transfer functions

The host library provides two type of generic functions that allow to send and receive data with bulk IN/ bulk out pipes either in polling or interrupt mode.

5.2.4.1 Polling transfer functions

- Transmit function

```
U8 host_send_data(U8 pipe, U16 nb_data, U8 *buf);
```

This function send nb_data pointed with *buf with the physical pipe number specified.

- Receive function

```
U8 host_get_data(U8 pipe, U16 *nb_data, U8 *buf);
```

This function receives nb_data pointed with *buf with the pipe number specified. The number of bytes to process is passed as parameter, thus when the functions returns it contains the number of data effectively received by the function.

5.2.4.2 None blocking transfer functions

To use these functions, “USB_HOST_PIPE_INTERRUPT_TRANSFER” should be defined as ENABLE in the host configuration section. These functions are similar to the previous blocking one, but they return imediately without active wait for the end of the data transfer. When the data transfer is complete or an error occurs, the callback function passed as pointer parameter is colled with a return status and the number of bytes processed as parameters of this callback function.

- Transmit function

```
U8 host_send_data_interrupt(U8 pipe, U16 nb_data, U8 *buf, void
(*handle)(U8 status, U16 nb_byte));
```

This function send nb_data pointed with *buf with the physical pipe number specified.

- Receive function

```
U8 host_get_data_interrupt(U8 pipe, U16 nb_data, U8 *buf, void  (*handle)(U8
status, U16 nb_byte));
```

This function receives nb_data pointed with *buf with the pipe number specified.

5.2.4.3 Returned values

The communication functions return the following status values:

```
#define PIPE_GOOD             0 //Transfer OK
#define PIPE_DATA_TOGGLE   0x01 //Data toggle error
#define PIPE_DATA_PID      0x02 //PID error
#define PIPE_PID           0x04
#define PIPE_TIMEOUT       0x08 //Time out error (no handshake received)
#define PIPE_CRC16         0x10 //CRC error
#define PIPE_STALL         0x20 //STALL handshake received
#define PIPE_NAK_TIMEOUT   0x40 //Maximum number of NAK handshake received
#define PIPE_DELAY_TIMEOUT 0x80 //Timeout error
```

5.2.5 Sample high level host application

The following code extracted from the host application task illustrates a typical high level host application management using the USB host sofware library:

```
void host_template_task(void)
{
  U16 *ptr_nb;
  U16 nb;
  U8 i;
  ptr_nb=&nb;
  // First check the host controller is in full operating mode with the B device
  // attached and enumerated
  if(Is_host_ready())
  {
    // New device connection (executed  only one time after device connection)
    if(Is_new_device_connection_event())
    {
      for(i=0;i<Get_nb_supported_interface();i++)
      {
      // First interface with two bulk IN/OUT pipes
        if(Get_class(i)==0x00 && Get_subclass(i)==0x00 && Get_protocol(i)==0x00)
        {
          //Get correct physical pipes associated to IN/OUT Endpoints
          if(Is_ep_addr_in(Get_ep_addr(i,0)))
          {  //Yes associate it to the IN pipe
              pipe_in=host_get_hwd_pipe_nb(Get_ep_addr(i,0));
              pipe_out=host_get_hwd_pipe_nb(Get_ep_addr(i,1));
          }
          else
          {  //No, invert...
              pipe_in=host_get_hwd_pipe_nb(Get_ep_addr(i,1));
              pipe_out=host_get_hwd_pipe_nb(Get_ep_addr(i,0));
          }
        }
```

```
      // Seconf interface with interrupt IN pipe
      if(Get_class(i)==0x00 && Get_subclass(i)==0x55 && Get_protocol(i)==0xAA)
      {
          pipe_interrupt_in=host_get_hwd_pipe_nb(Get_ep_addr(i,0));
          Host_select_pipe(pipe_interrupt_in);
          Host_continuous_in_mode();
          Host_unfreeze_pipe();
      }
    }
  }
  // Firt interface (bulk IN/OUT) management
  // In polling mode
  // The sample task sends 64 byte through pipe nb2...
  sta=host_send_data(pipe_out,64,tab);
  // And receives 64bytes from pipe nb 1...
  *ptr_nb=64;
  sta=host_get_data(pipe_in,ptr_nb,tab);
  // Second interface management (USB interrupt IN pipe)
  Host_select_pipe(pipe_interrupt_in);
  if(Is_host_in_received())
  {
    if(Is_host_stall()==FALSE)
    {
      i=Host_read_byte();
      Host_read_byte();
    }
    Host_ack_in_received(); Host_send_in();
  }
  // Here an example of an applicative request to go to USB suspend ...
  if(Is_hwb())
  {
    Host_request_suspend();
  }
}
  // Here an applicative example of resume request...
  if(Is_host_suspended() && Is_joy_select())
  {
    Host_request_resume();
  }

   //Device disconnection...
   if(Is_device_disconnection_event())
   {
      //Put here code to be executed upon device disconnection...
   }
}
```

6. FAQ

6.1 Device mode

6.1.1 How to change the VID & the PID?

The VID (Vendor ID) and the PID (Product ID) allow the identification of the device by the host. Each manufacturer should have its own VID, which will be the same for all products (it is assigned by USB org). Each product should have its own PID (it is assigned by the manufacturer).

The value of the VID and the PID are defined in *usb_descriptor.h.* To change them you have to change the values below:

// USB Device descriptor

```
#define VENDOR_ID             0x03EB // Atmel vendor ID = 03EBh
#define PRODUCT_ID            0x201C
```

6.1.2 How can I change the string descriptors value?

The value of the string descriptors are defined in *usb_descriptor.h.* For example to change the product name value, you have to change the following values:

The length of the string value:

```
#define USB_PN_LENGTH         18
```

The String value:

```
#define USB_PRODUCT_NAME \
{ Usb_unicode('A') \
 ,Usb_unicode('V') \
 ,Usb_unicode('R') \
 ,Usb_unicode(' ') \
 ,Usb_unicode('U') \
 ,Usb_unicode('S') \
 ,Usb_unicode('B') \
 ,Usb_unicode(' ') \
 ,Usb_unicode('M') \
 ,Usb_unicode('O') \
 ,Usb_unicode('U') \
 ,Usb_unicode('S') \
 ,Usb_unicode('E') \
 ,Usb_unicode(' ') \
 ,Usb_unicode('D') \
 ,Usb_unicode('E') \
 ,Usb_unicode('M') \
 ,Usb_unicode('O') \
}
```

6.1.3 How can I configure my device in self-powered or bus-powered mode?

The parameter to configure the device in self-powered or bus-powered mode is defined in *usb_descriptor.h* file. Hereunder the definition of each mode.

bus-power mode:

// USB Configuration descriptor

```
#define CONF_ATTRIBUTES    USB_CONFIG_BUSPOWERED
```

Self-power mode:

// USB Configuration descriptor

```
#define CONF_ATTRIBUTES    USB_CONFIG_SELFPOWERED
```

6.1.4 How can I add a new descriptor?

To add a new descriptor to your application, you have to follow the steps below:

1. Define the values of the descriptor parameters and its stucture type in *usb_descriptors.h* file.

For example the HID descriptor and its structure should be defined in *usb_descriptors.h* file as shown below:

```
/* ____HID descriptor_____________________________________________________*/
#define HID                   0x21
#define REPORT                0x22
#define SET_REPORT      0x02
#define HID_DESCRIPTOR        0x21
#define HID_BDC               0x1001
#define HID_COUNTRY_CODE      0x00
#define HID_CLASS_DESC_NB     0x01
#define HID_DESCRIPTOR_TYPE   0x22

/*_____ U S B   H I D   D E S C R I P T O R
_____________________________________*/
typedef struct {
  U8  bLength;               /* Size of this descriptor in bytes */
  U8  bDescriptorType;       /* HID descriptor type */
  U16 bscHID;                /* Binay Coded Decimal Spec. release */
  U8  bCountryCode;          /* Hardware target country */
  U8  bNumDescriptors;      /* Number of HID class descriptors to follow */
  U8  bRDescriptorType;      /* Report descriptor type */
  U16 wDescriptorLength;     /* Total length of Report descriptor */
} S_usb_hid_descriptor;
```

2. Add the new descriptor to the *s_usb_user_configuration_descriptor* structure in *usb_descriptors.h* file:

```
typedef struct
{
   S_usb_configuration_descriptor cfg_mouse;
   S_usb_interface_descriptor     ifc_mouse;
   S_usb_hid_descriptor           hid_mouse;
   S_usb_endpoint_descriptor      ep1_mouse;
} S_usb_user_configuration_descriptor;
```

3. In the File *usb_descriptors.c,* add the size of the new descriptor to the *wTotalLength* parameter of the configuration descriptor and add the descriptor value (see the example below):

```
code S_usb_user_configuration_descriptor usb_conf_desc = {
 { sizeof(S_usb_configuration_descriptor)
 , CONFIGURATION_DESCRIPTOR
 , Usb_write_word_enum_struc(sizeof(S_usb_configuration_descriptor)\
       +sizeof(S_usb_interface_descriptor)   \
       +sizeof(S_usb_hid_descriptor)         \
       +sizeof(S_usb_endpoint_descriptor)    \
                                )
 , NB_INTERFACE
 , CONF_NB
 , CONF_INDEX
 , CONF_ATTRIBUTES
 , MAX_POWER
 }
 ,
 { sizeof(S_usb_interface_descriptor)
 , INTERFACE_DESCRIPTOR
 , INTERFACE_NB_MOUSE
 , ALTERNATE_MOUSE
 , NB_ENDPOINT_MOUSE
 , INTERFACE_CLASS_MOUSE
 , INTERFACE_SUB_CLASS_MOUSE
 , INTERFACE_PROTOCOL_MOUSE
 , INTERFACE_INDEX_MOUSE
 }
 ,
 { sizeof(S_usb_hid_descriptor)
 , HID_DESCRIPTOR
 , HID_BDC
 , HID_COUNTRY_CODE
 , HID_CLASS_DESC_NB
 , HID_DESCRIPTOR_TYPE
 , Usb_write_word_enum_struc(sizeof(S_usb_hid_report_descriptor_mouse))
 }
 ,
 { sizeof(S_usb_endpoint_descriptor)
 , ENDPOINT_DESCRIPTOR
 , ENDPOINT_NB_1
 , EP_ATTRIBUTES_1
 , Usb_write_word_enum_struc(EP_SIZE_1)
 , EP_INTERVAL_1
 }
};
```

4. Do not forget to add all functions related to manage this new descriptor.

6.1.5 How can I add a new endpoint?

To configure a new endpoint, follow the steps below:

1. As explained in the USB descriptors section, an endpoint belongs to an interface. The first thing to do to add a new endpoint, is to increment by one to the ***NB_ENDPOINT*** parameter value. This value is defined in *usb_descriptor.h* file and it belong to the interface descriptor prameters.

// USB Interface descriptor

```
#define INTERFACE_NB xx
#define ALTERNATE xx
#define NB_ENDPOINT xx//This parameter=the endpoints number of the
interface
#define INTERFACE_CLASS xx
#define INTERFACE_SUB_CLASS xx
#define INTERFACE_PROTOCOL xx
#define INTERFACE_INDEX xx
```

2. The next step is to define the endpoint descriptor values . These values have to be defined in *usb_descriptors.h*.

```
// USB Endpoint 1 descriptor FS
#define ENDPOINT_NB_1       (EP_MOUSE_IN | 0x80)
#define EP_ATTRIBUTES_1     0x03             // BULK = 0x02, INTERUPT = 0x03
#define EP_IN_LENGTH_1      8
#define EP_SIZE_1           EP_IN_LENGTH_1
#define EP_INTERVAL_1       0x02  // Interrupt polling interval from host
```

EP_MOUSE_IN is defined in *conf_USB.h* to specify the endpoint number used by the application.

3. Add the new endpoint descriptor to the configuration descriptor (proceed as explained in the previous FAQ point).
4. Add the hardware initialization call in *usb_specific_request.c*

```
void usb_user_endpoint_init(U8 conf_nb)
{
  usb_configure_endpoint(EP_MOUSE_IN,      \
                         TYPE_INTERRUPT,     \
                         DIRECTION_IN,  \
                         SIZE_8,       \
                         ONE_BANK,     \
                         NYET_ENABLED);
}
```

7. Coding Style

The coding style explained hereunder are important to understand the firmware:

- Defined contants use caps letters.

```
#define FOSC 8000
```

- Macros Functions use the first letter as cap

```
#define Is_usb_sof() ((UDINT &   MSK_SOFI)    ? TRUE: FALSE)
```

- The user application can execute its own specific instructions upon each USB events thanks to hooks defined as following in *usb_conf.h.*

```
#define Usb_sof_action()         sof_action();
Note: The hook function should perform only short time requirement
operations !
```

- Usb_unicode() macro function should be used everywhere (String descriptors...) an unicode char is exchanged on the USB protocol.

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature

Printed on recycled paper.

7675A–AVR–01/07