**APPLICATION NOTE**

# USING THE ST7263 FOR DESIGNING A USB MOUSE

**by Microcontroller Division Application Team**

## INTRODUCTION

This application note describes the implementation of a cost-effective USB Mouse using the ST7263 microcontroller. A detailed description of low-consumption power management mode (resume mode) is given in section 5.

ST provides a complete architecture as well as firmware drivers to help you develop your application. A list of reference documents is provided at the end of the application note. It is assumed that the reader is familiar with the ST7263 microcontroller and USB.

# Table of Contents

## Table of Contents

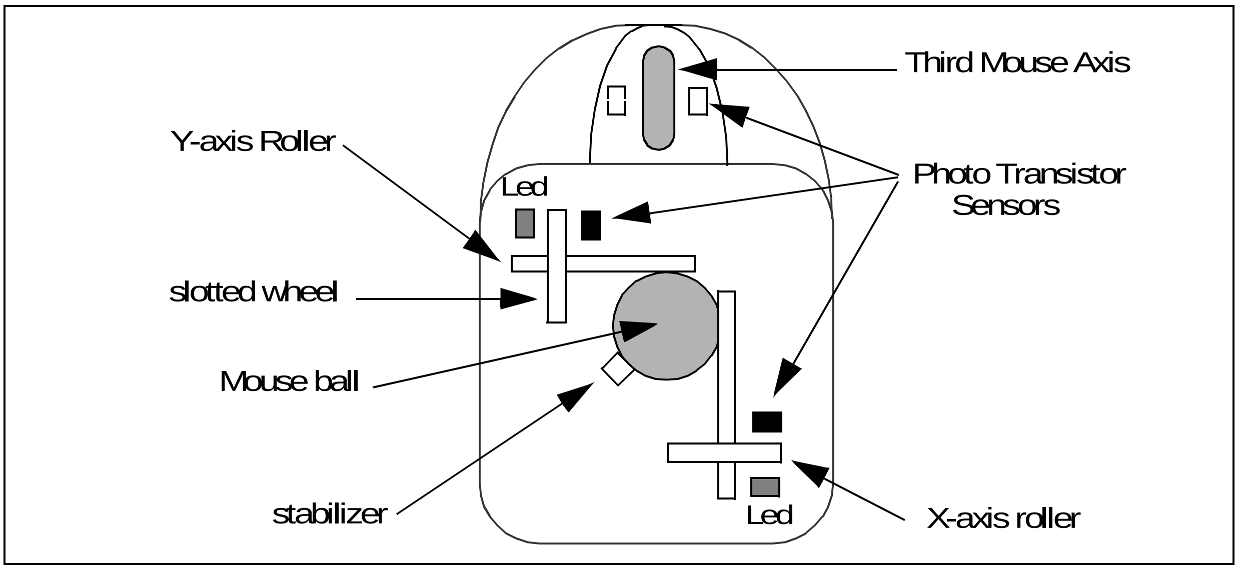
## 1 MOUSE BASIC

There are three ways of encoding the displacements of a mouse: mechanical, opto-mechanical and optical. Because of its relatively high resolution and reliable behaviour on a wide range of surfaces, the opto-mechanical mouse is the most common found on the market.

### 1.1 OPTO-MECHANICAL DESCRIPTION

An opto-mechanical mouse uses a rolling ball, slotted wheels, LEDs and encoders to translate the two-dimensional mouse movement into electrical signals.

**Figure 1. Mouse mechanical hardware**

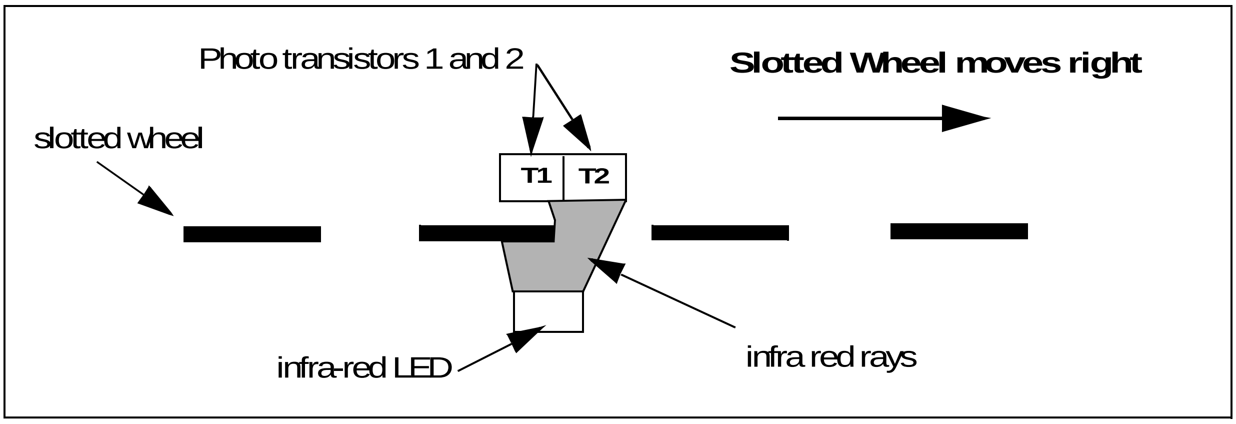


To track mouse movements, a rubber ball transmits the vertical and horizontal displacements to two perpendicular rollers, with a slotted wheel connected at one end. A double Photo transistor cell is associated with each slotted wheel to track the X-axis and Y-axis displacements.

### 1.2 PHOTO TRANSISTOR DETECTORS

The encoder consists of an infra-red LED on one side of the slotted wheel and two superimposed photo transistor detectors on the other side. During a mouse displacement, the slotted wheel crosses the infra-red signal so that the photo transistor detectors conduce alternatively one after the other. This is illustrated in Figure 2.

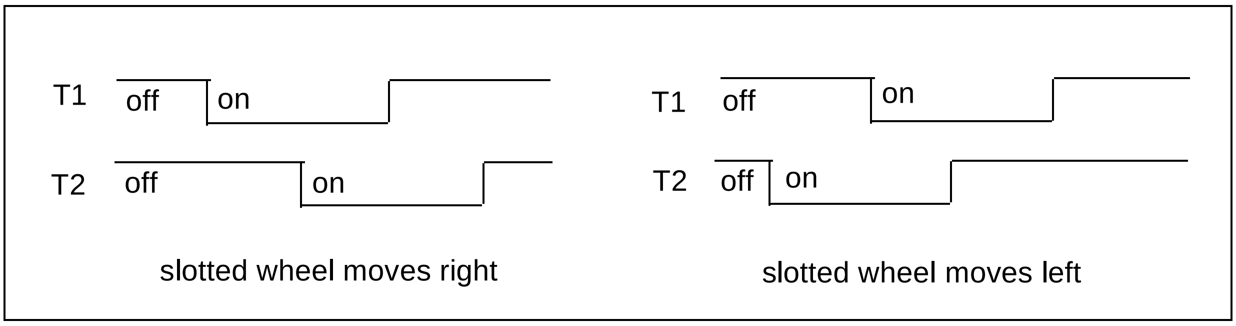
**Figure 2. X-Axis roller opto-mechanical detail**



## 1.3 DISPLACEMENT DIRECTION DETECTION

Two photo transistor sensors implemented on the same physical component enable the direction detection of the axis supporting the slotted wheel. They deliver quadrature signals as illustrated in Figure 3.

**Figure 3. Photo transistors output waveforms**



## 2 INTRODUCTION TO THE ST7263 MICROCONTROLLER

The ST7263 microcontrollers form a sub family of the ST7 dedicated to USB applications. The devices are based on an industry-standard 8-bit core and feature an enhanced instruction set. They operate a a 24MHz oscillator frequency. Under software control, the ST7263 MCUs may be placed in either Wait or Halt modes, thus reducing power consumption.

The S7263 Microcontrollers include a ST7 Core, up to 16K program memory, up to 512 bytes RAM, 19 I/O lines and the following on-chip peripherals:

- USB Low speed interface with 3 endpoints with programmable in/out configuration using DMA architecture with embedded 3.3V voltage regulator and transceivers (no external components are needed).
- 8-bit Analog-to-Digital converter (ADC) with 8 multiplexed analog inputs.
- Industry standard asynchronous SCI serial interface (8K & 16K ROM versions).
- Watchdog.
- 16-bit Timer featuring an External clock input, 2 Input Captures, 2 Output Compares with Pulse Generator capabilities.
- Fast I2C Multi Master interface (16K only).
- Low voltage reset ensuring proper power-on or power-off of the device.

The following application as been implemented on a ST72633, a microcontroller version with 4K ROM and 256 bytes RAM.

## 3 HARDWARE IMPLEMENTATION

The Figure 4 shows a typical circuit for a low-cost USB mouse using the ST7263.

A 24Mhz ceramic resonator is connected to the clock inputs of the microcontroller. The external oscillator frequency is divided by 3 to get the internal clock (8Mhz for CPU and Peripherals) and by 4 to get the 6 Mhz USB frequency. The resonator should be placed close to the OSCIN and OSCOUT pins.

The embedded voltage regulator generates the 3.3V reference for USB interface on the USBVCC pin. A 1.5 KΩ resistor is connected from the USBVCC pin to USBDM line to signal that the mouse as a low-speed device.

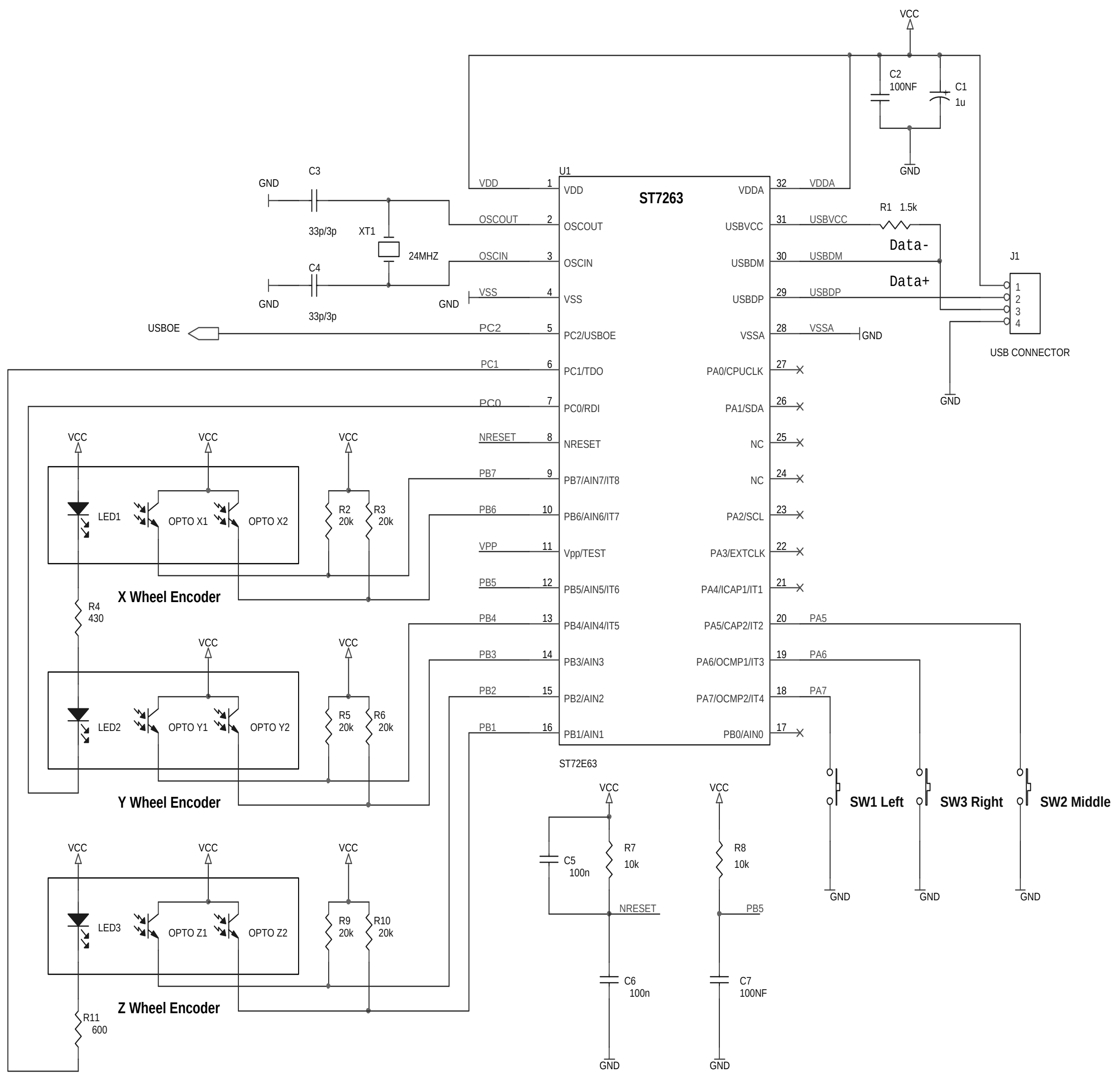
The three Mouse buttons (Left, middle and right) are connected to port A pins 7, 6 and 5 respectively. These pins are configured as input with 20kΩ Pull-up resistors. As the left and right buttons are connected to falling edge external interrupts, they can be used to exit suspend mode.

The X, Y and Z encoders are all connected to Port B pins configured as input without pull-up. Note that the outputs of the Photo transistors are connected to the microcontroller via 20kΩ Pull-up resistors. These resistors are not integrated inside the microcontroller. This is to allow you to modify the maximum range of the Photo transistor output signals.

Pin PB5 is connected to an external RC circuit to implement the wake up interrupt function. This is explained in more detail in the Power management chapter (section 5).

The Two pins, PC0 and PC1, drive the mouse LEDs for X-Y axis and for Z axis respectively.

## Figure 4. Hardware Implementation



# 4 SOFTWARE IMPLEMENTATION

## 4.1 DEVICE ENUMERATION AND CONFIGURATION

When a USB device is attached, the host issues a reset signal. When the reset signal is released, the device enters the enumerated state.

### 4.1.1 USB Reset

The USB reset is independent from the chip reset. A USB reset signal resets the USB interface peripheral but not the ST7 core and the other peripherals.

When a USB reset signal is detected on the bus, the RESET bit in the ISTR register is set and a USB interrupt is generated. All the USB interface registers are reset.

### 4.1.2 Enumeration

The host performs a bus enumeration to identify the attached device and to assign a unique address to it. The device responds to the requests sent by the host during the enumeration process on its default pipe (endpoint 0).

Enumeration steps:

**a. Get Device descriptor**

The host sends a get device descriptor request. The device replies with its device descriptor to report its attributes (Device Class, maximum packet size for endpoint zero).

**b. Set address**

A USB device uses the default address after reset until the host assigns a unique address using the set address request. The firmware writes the device address assigned by the host in the DADDR register.

**c. Get configuration**

The host sends a get configuration. The device replies with its configuration descriptor, interface descriptor and endpoint descriptor. The configuration descriptor describes the number of interfaces provided by the configuration, the power source (Bus or Self powered) and the maximum power consumption of the USB device from the bus. The Interface descriptor describes the number of endpoints used by this interface. The Endpoint descriptor describes the transfer type supported and the bandwidth requirements.

**d. Set Configuration**

The host assigns a configuration value to the device based on the configuration information. The device is then in configured state and can draw the amount of power described in the configuration descriptor. The device is now configured and ready to be used.

For more information, see also the USB specification, chapter 9, “USB Device Framework”.

## 4.2 USB MOUSE DESCRIPTORS

USB protocol can configure devices at start-up or when they are plugged-in at run time. These devices are divided into various device classes. Each device class defines the common behaviour and protocols for devices that serve similar functions.

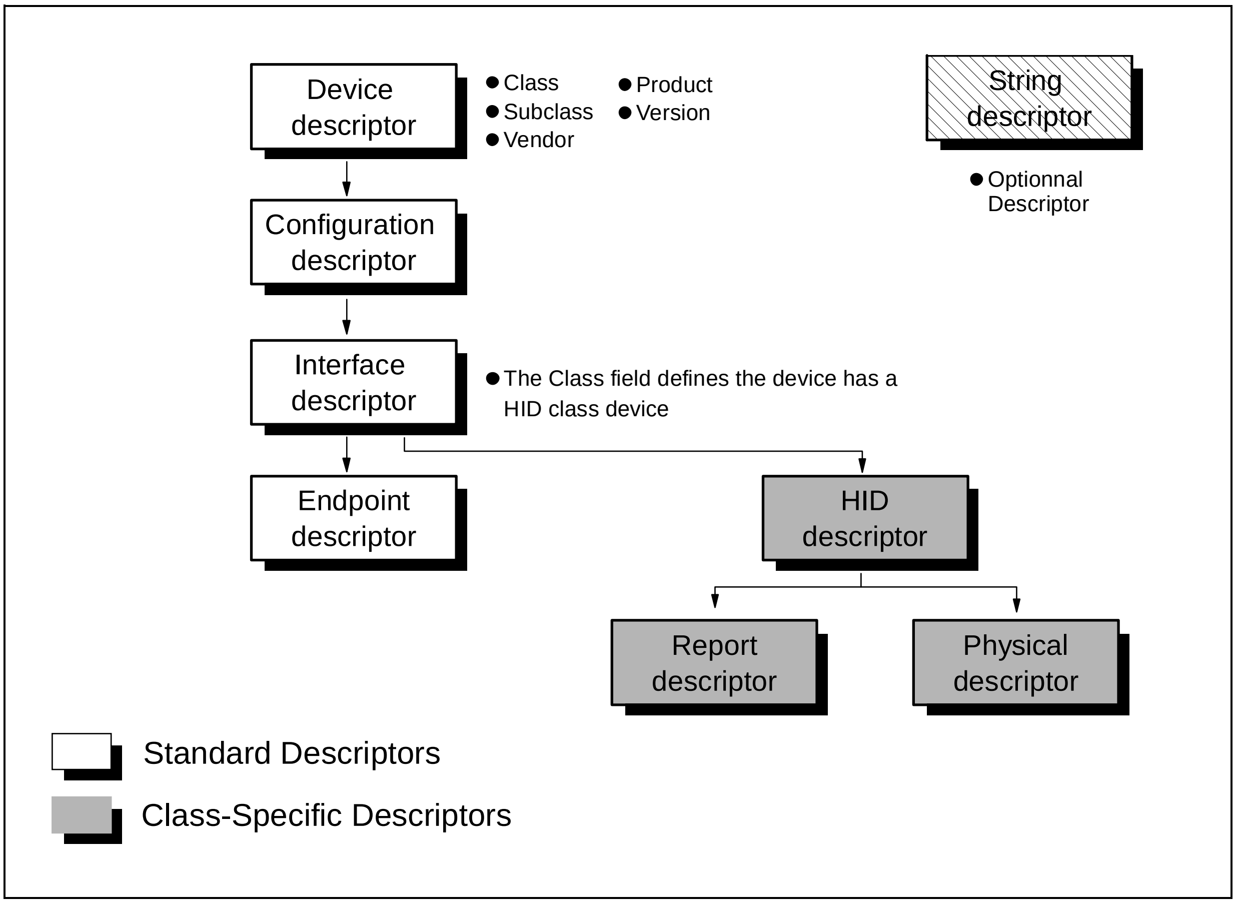
### 4.2.1 Descriptor Structure

The HID class consists primarily of devices that are used by humans to control the operation of computer systems. Mice, like all pointing devices, are typical examples of HID class devices.

The information about a USB mouse are stored in segments of its ROM. These segments are called Descriptors and are divided into several types: Device Descriptor, Configuration Descriptor, Interface Descriptor, HID Descriptor, Endpoint Descriptor, String Descriptor and Report Descriptor. All Descriptors (except String Descriptor) are mandatory in this application.

Figure 4 illustrates the Descriptor structure.

**Figure 5. Descriptor structure for a Mouse: Human Interface Device (HID)**



### 4.2.2 Device Descriptors

At the top level, a descriptor includes two tables of information referred to as the Device Descriptor and the String Descriptor. A standard USB Device Descriptor specifies the Product ID and other informations about the device. For example, Device Descriptor fields primarily include: Class, Subclass, Vendor, Product, Version.

The following code corresponds to USB Mouse Descriptors applied to a two-buttons, two- axis opto-mechanical mouse.

```
const Byte DeviceDescriptor[] = {
      0x12, // bLength
      0x01, // bDescriptorType
      0x00, // bcdUSB
      0x01, // bDeviceClass
      0x00,
      0x00, // bDeviceSubClass
      0x00, // bDeviceProtocolconst Byte DeviceDescriptor[] = {
      0x12, // bLength
      0x01, // bDescriptorType
      0x00, // bcdUSB
      0x01, // bDeviceClass
      0x00,
      0x00, // bDeviceSubClass
      0x00, // bDeviceProtocol
      0x08, // bMaxPacketSize0
      0x83, // idVendor
      0x04,
      0x02, // idProduc
      0x00,
      0x06, // bcdDevice
      0x00,
      INDEX_MANUFACT, // Index of string descriptor describing manufacturer
      INDEX_PRODUCT, // Index of string descriptor describing product
      INDEX_SERIALNUM, //Index of string descriptor describing the device’s se-
      rial number
      0x01 // bNumConfigurations
      };
```

### 4.2.3 Configuration Descriptor

This Descriptor divided into several segments includes Interface Descriptor, HID descriptor and Endpoint Descriptor:

```
const Byte ConfDescriptor[] = {
      // Configuration descriptor
      0x09, // bLength: Configuration Descriptor size
      0x02, // bDescriptorType: Configuration
      0x22, // wTotalLength: 34 Bytes returned
      0x00,
      0x01, // bNumInterfaces: 1 interface
      0x01, // bConfigurationValue: Configuration value
      0x00, // iConfiguration: Index of string descriptor describing the configu-
      ration
      0x60, // bmAttributes: Bus powered and Remote wake up
      0x0A // MaxPower 20 mA
      // Interface descriptor
      0x09, // bLength: Interface Descriptor size
      0x04, // bDescriptorType: Interface descriptor type
      0x01, // bInterfaceNumber: Number of Interface )
      0x00, // bAlternateSetting: Alternate setting
      0x01, // bNumEndpoints: one endpoint used
      0x03, // bInterfaceClass: HID
      0x01, // bInterfaceSubClass: No subclass
      0x02, // bInterfaceProtocol: Mouse
      0x00, // interface: Index of string descriptor
      // HID descriptor
      0x09, // bLength: HID Descriptor size
      0x21, // bDescriptorType: HID
      0x00, // bcdHID: HID Class Spec release number
      0x01,
      0x21, // bCountryCode: Hardware target country US
      0x01, // bNumDescriptors: Number of HID class descriptors to follow
      0x22, // bDescriptorType
      REPORTDESCSIZE_L, // wItemLength: Report descriptor length low bite
      REPORTDESCSIZE_H,
      // Endpoint descriptor
      0x07, // bLength: Endpoint Descriptor size
      0x05, // bDescriptorType: Endpoint descriptor type
      0x81, // bEndpointAddress: Endpoint Address (IN)
      0x03, // bmAttributes: Interrupt endpoint
      0x08, // wMaxPacketSize: 8 Byte max
      0x00,
      0x0A // bInterval: Polling Interval (8 ms)
      };
```

### 4.2.4 Report Descriptor

The Report Descriptor is different from the other descriptors in that it is not simply a table of values. It is made up of items that provide information about the data provided by each control in a device. Input items are used to tell the host what type of data will be returned as input to the host for interpretation, wether the data is absolute or relative and other pertinent information. By looking at a Report Descriptor alone, an application knows how to handle incoming data, as well as what the data could be used for. The following descriptor describes a two-buttons, two-axis USB Mouse.

```
const Byte ReportDescriptor[] = {
      0x05, 0x01, // USAGE_PAGE (Generic Desktop)
      0x09, 0x02, // USAGE (Mouse)
      0xa1, 0x01, // COLLECTION (Application)
      0x09, 0x01, // USAGE (Pointer)
      0xa1, 0x00, // COLLECTION (Physical)
      0x05, 0x09, // USAGE PAGE (Buttons)
      0x19, 0x01, // USAGE_MINIMUM (01)
      0x29, 0x03, // USAGE_MAXIMUM (03)
      0x15, 0x00, // LOGICAL_MINIMUM (0)
      0x25, 0x01, // LOGICAL_MAXIMUM (1)
      0x95, 0x03, // REPORT_COUNT (3)
      0x75, 0x01, // REPORT_SIZE (1)
      0x81, 0x02, // INPUT (Data,Var,Abs)
      0x95, 0x01, // REPORT_COUNT (1)
      0x75, 0x05, // REPORT_SIZE (5)
      0x81, 0x01, // INPUT (Cnst)
      0x05, 0x01, // USAGE PAGE (Generic Desktop)
      0x09, 0x30, // USAGE (X)
      0x09, 0x31, // USAGE (Y)
      0x15, 0x81, // LOGICAL_MINIMUM (-127)
      0x25, 0x7F, // LOGICAL_MAXIMUM (127)
      0x75, 0x08, // REPORT_SIZE (8)
      0x95, 0x02, // REPORT_COUNT (2)
      0x81, 0x06, // INPUT (Data,Var,Relative)
      0xc0, // END_COLLECTION
      0xc0, // END_COLLECTION
      };
```

### 4.2.5 String Descriptor

The previous descriptors can contain references to string Descriptors that provide displayable information describing a descriptor in human-readable form. The inclusion of string Descriptors is optional. String Descriptors use UNICODE encoding.

In the following String Descriptor example, all fields can be modified to enter your own manufacturer index, product index and serial number index. In this case, some fields have to be replaced by a #define NAME at the top of the <descript.h> file, these defines are needed by the USB protocol layer to handle the report sending transaction.

```
#define STRINGDESCSIZE 82 // String descriptor Length
#define INDEX_MANUFACT 0x04 // Index of String Descriptor describing manufacturer
#define LENGTH_MANUFACT 0x26 // Manufacturer length
#define INDEX_PRODUCT 0x2A // Index of String Descriptor describing product
#define LENGTH_PRODUCT 0x22 // Product length
#define INDEX_SERIALNUM 0x4C // Index of String Descriptor describing serial
number
#define LENGTH_SERIALNUM 0x06 // Serial number length

const Byte StringDescriptor[] = {
      0x04, // bLength : Length of String descriptor
      0x03, // bDescriptorType
      0x09, // bString US English
      0x04,
      LENGTH_MANUFACT,  // bLength: Length of String descriptor
      0x03, // bDescriptorType
      uS, uT, // Manufacturer
      uM, ui, uc, ur, uo, ue, ul, ue, uc, ut, ur, uo, un, ui, uc, us,
      LENGTH_PRODUCT, // bLength: Length of String descriptor
      0x03, // bDescriptorType
      uS, uT, u7, u2, u6, u3, uSPACE, uU, uS, uB, uSPACE, uM, uo, uu, us, ue,
      LENGTH_SERIALNUM,
      0x03, // bDescriptorType
      uS, uT,
      LENGTH_ADDSTRING0,
      0x03,
      // Add your string
      LENGTH_ADDSTRING1,
      0x03
      // Add your string
      };
```

## 4.3 DATA TRANSFER

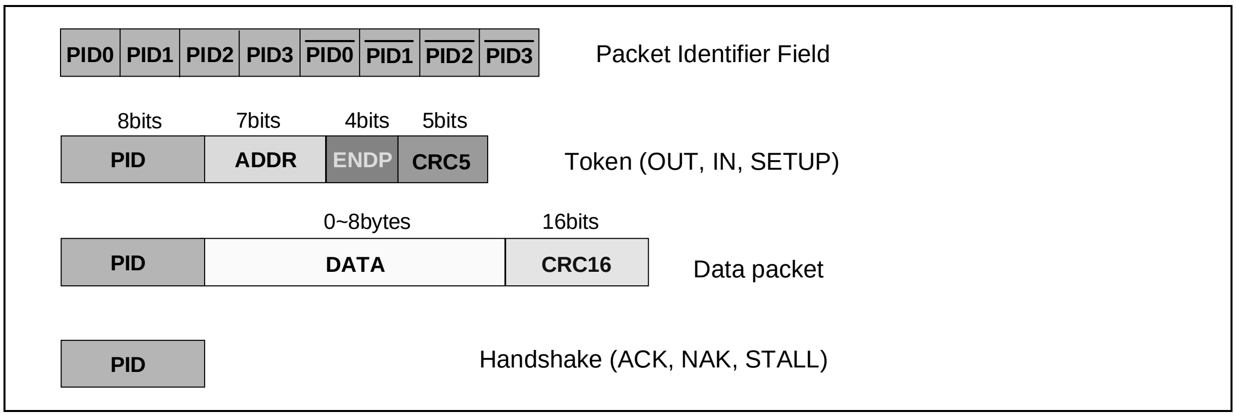
The USB Mouse should be able to receive data on endpoint 0 and send data through endpoint 0 and endpoint 1. The transfer types supported by this application are:

**1. Control transfer** with endpoint 0 (SETUP and IN and OUT tokens)

**2. Interrupt transfer with endpoint 1** (IN token)

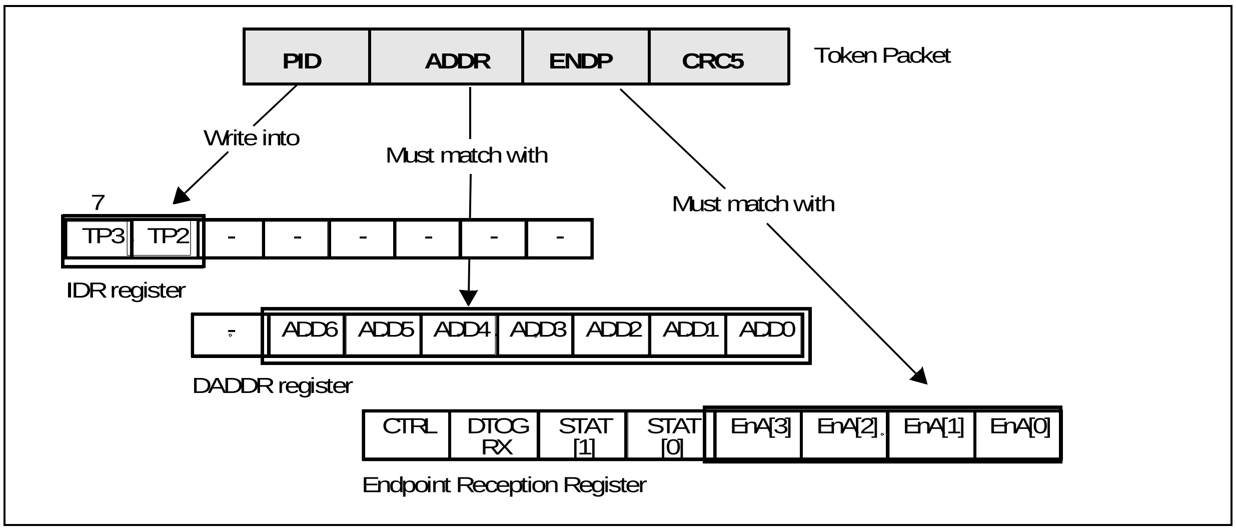
Figure 6 presents the different USB packet types encountered in this application.

**Figure 6. USB Low-speed packet formats**



All the decoding / encoding operations on the USB frames are handled by firmware. The following figure describes the interaction between the received token packet and the hardware registers of the ST7263 USB interface.

**Figure 7. Token packet reception**



The firmware must determine the data transfer direction and the endpoint number which has sent or received data by reading the IDR and PIDR registers.

The following table shows the data transfer direction corresponding to each PID.

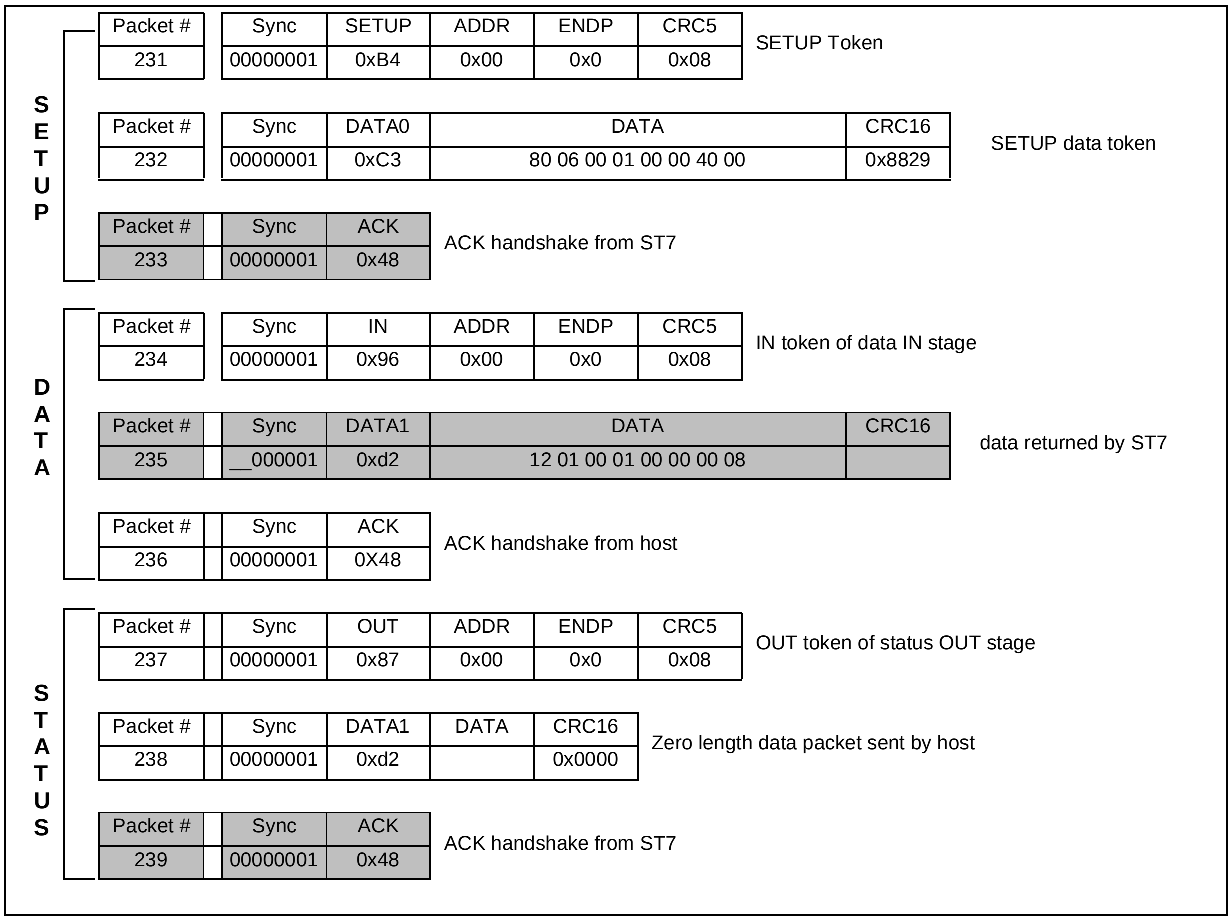
| PID name | Data transfer direction |
|---|---|
| SETUP, OUT | from host to device |
| IN | from device to host |

### 4.3.1 Control transfer with endpoint 0

All control transfers are supported by endpoint 0. There can be control transfers with data phase and control transfers without data phase. As a consequence, a control transfer may have three transaction stages: a Setup stage, a Data stage (not for no-data control transfer) and a Status stage.

Figure 8 shows an example of control transfer with one data IN stage.

**Figure 8. Control transfer**



| Packet # | Sync | SETUP | ADDR | ENDP | CRC5 |
|---|---|---|---|---|---|
| 231 | 00000001 | 0xB4 | 0x00 | 0x0 | 0x08 |

| Packet # | Sync | DATA0 | DATA | CRC16 |
|---|---|---|---|---|
| 232 | 00000001 | 0xC3 | 80 06 00 01 00 00 40 00 | 0x8829 |

| Packet # | Sync | ACK |
|---|---|---|
| 233 | 00000001 | 0x48 |

| Packet # | Sync | IN | ADDR | ENDP | CRC5 |
|---|---|---|---|---|---|
| 234 | 00000001 | 0x96 | 0x00 | 0x0 | 0x08 |

| Packet # | Sync | DATA1 | DATA | CRC16 |
|---|---|---|---|---|
| 235 | __000001 | 0xd2 | 12 01 00 01 00 00 00 08 | |

| Packet # | Sync | ACK |
|---|---|---|
| 236 | 00000001 | 0X48 |

| Packet # | Sync | OUT | ADDR | ENDP | CRC5 |
|---|---|---|---|---|---|
| 237 | 00000001 | 0x87 | 0x00 | 0x0 | 0x08 |

| Packet # | Sync | DATA1 | DATA | CRC16 |
|---|---|---|---|---|
| 238 | 00000001 | 0xd2 | | 0x0000 |

| Packet # | Sync | ACK |
|---|---|---|
| 239 | 00000001 | 0x48 |

The use of DMA architecture allows the endpoint definition to be completely flexible. The received and send data are stored in RAM buffers assigned to the DMA of the USB interface. The DMA buffers are in a contiguous RAM area. The firmware must write the starting address of the DMA memory area in the DMAR register and in the DA7 and DA6 bits of the IDR register. The six least significant bits of the IDR register are managed by hardware. For detailed information on the mapping, see the USB interface chapter of the ST7263 data sheet.

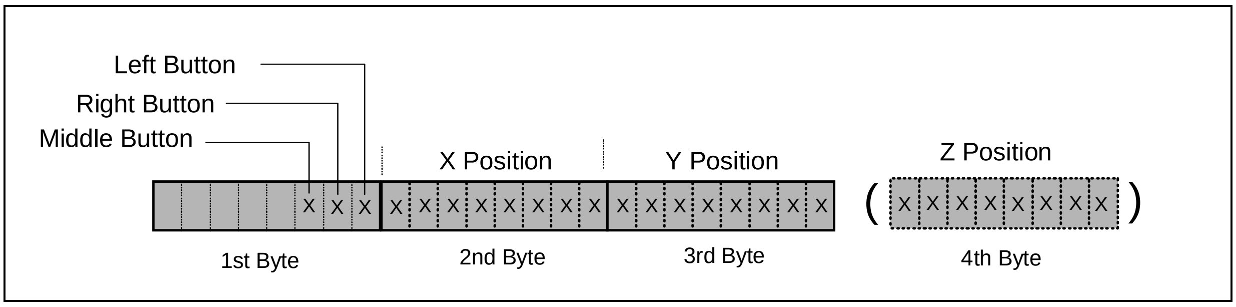
### 4.3.2 Interrupt transfer with endpoint 1

After the enumeration phase, the host continuously issues IN tokens through the endpoint 1 interrupt pipe. As the mouse has some data to return to the host, it returns three bytes of data (for a 2-axis mouse) or four bytes of data (for a 3-axis mouse).

These bytes are in format used for the boot report format for USB Mouse so that the data can be correctly interpreted by the BIOS. As shown in Figure 9, the first byte contains Left, middle and right buttons states, the second and the third bytes contain the relative displacement of the X and Y Axis respectively, since the last transaction. An optional fourth byte contains the Z relative displacement.

When the mouse has some data to return to the host through the interrupt pipe, it must write this data in the DMA buffer and enable endpoint 1 in transmission by setting the EP1RA to VALID. The host polls endpoint 1 with a polling interval given in the endpoint descriptor by sending an IN token. The hardware interface replies with STALL, NAK or data.

**Figure 9. Byte definition of boot report for USB Mouse**



## 5 POWER MANAGEMENT

The Suspend mode has been introduced in the USB specification to make an attached device enter low-power mode if no bus activity is detected for more than 3.0 ms.

Moreover, an attached device must be able to leave the suspend state if the Host issues a Reset signal, a Resume signal or if the device issues a remote wake-up sequence.

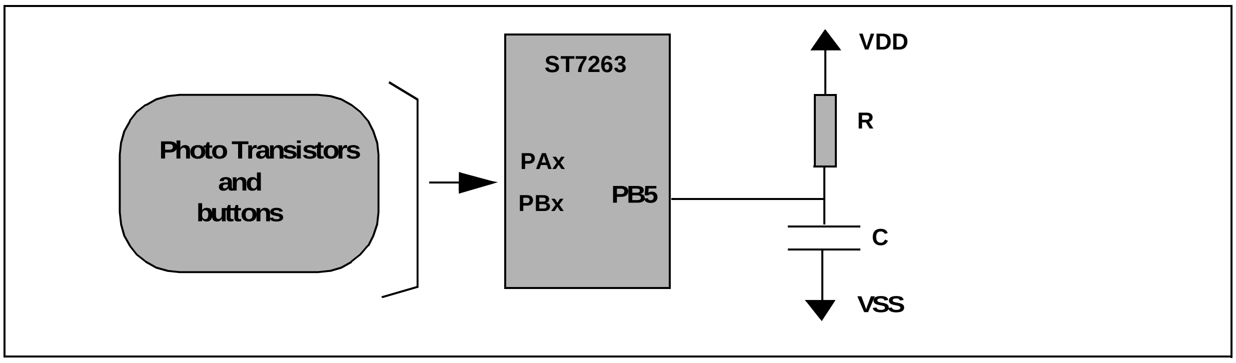
### 5.1 USB SUSPEND MODE IMPLEMENTATION

Depending on the USB mouse functionality, The application must issue a remote wake-up sequence if a movement has been detected on one of the Photo transistor outputs or if one of the mouse buttons has been pressed. Then the application must periodically check the port states without consumming more than 500 µA average.

To support the suspend mode specification, the ST7263 microcontroller enters low-power mode by executing the HALT instruction. The internal oscillator is then turned off, causing all internal processing to be stopped, including the operation of the on-chip peripherals. We can exit HALT mode by receiving a reset signal, an End Suspend Interrupt coming from the USB peripheral or an external interrupt (edge detection).

In order to be within the 500 µA suspend mode specification, the most common solution consists of using an external RC circuit.

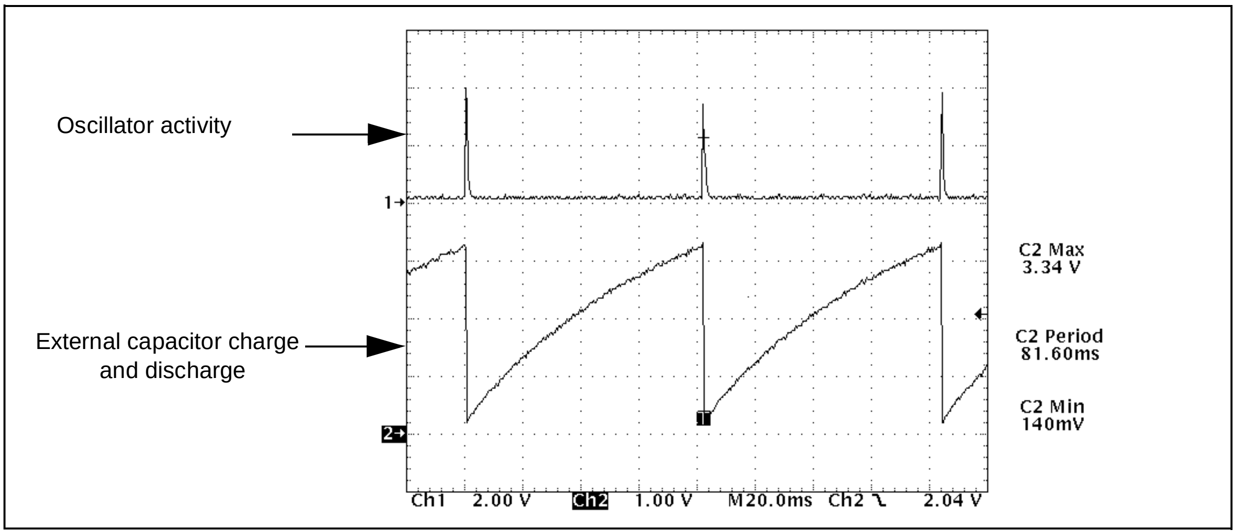
**Figure 10. RC external circuitry**



An external capacitor loads trough an external resistor. This capacitor is connected to the PB5 external interrupt pin of the microcontroller. As soon as the capacitor load has reached the Low-to-High trigger level, the external interrupt vector is set. Then, the entire microcontroller is woken-up by hardware (Oscillator, core and Peripherals). If the mouse has moved or if the buttons have been pressed, the application must perform a Remote Wake-up sequence, otherwise, we discharge the capacitor and re-enter suspend state.

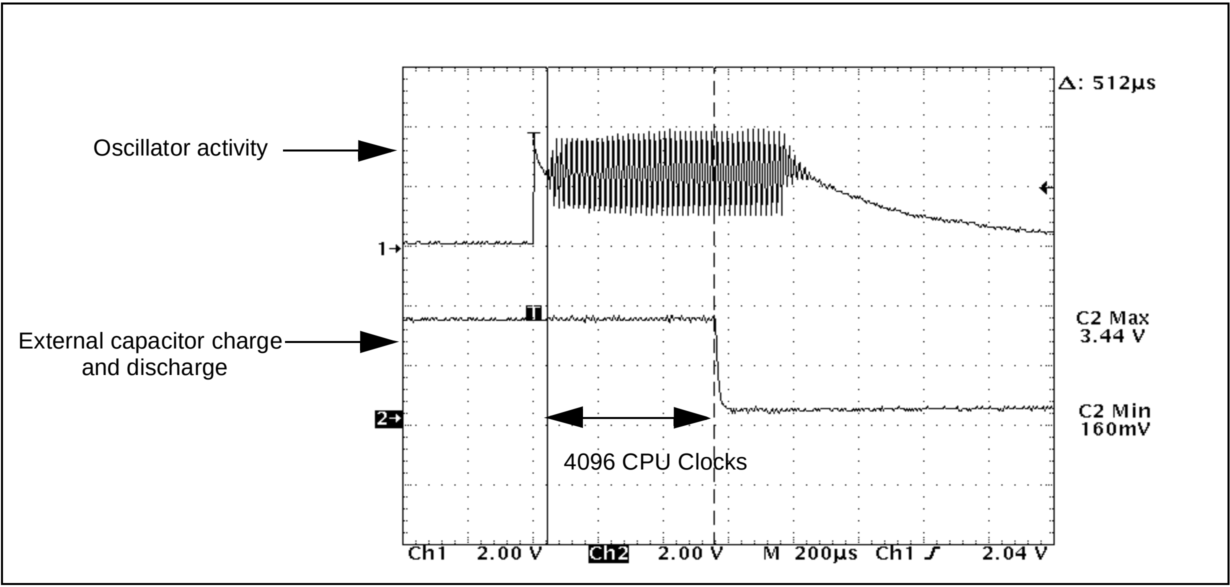
Figure 11 shows the RC circuit behaviour when the mouse is not moved.

**Figure 11. RC circuit behaviour in suspend mode**



During the charge of the capacitor, the microcontroller is in low-power mode, it draws only about 250 µA. As soon as the charge of the capacitor reaches the interrupt trigger level, the microcontroller wakes-up. The oscillator is then turned on and a stabilisation time is required before releasing CPU clock cycles. This stabilization time is 4096 CPU clock cycles. During this activity phase, the microcontroller draws about 20 mA. This oscillator wake-up phase is shown in Figure 12.

**Figure 12. Oscillator Waking-up sequence**

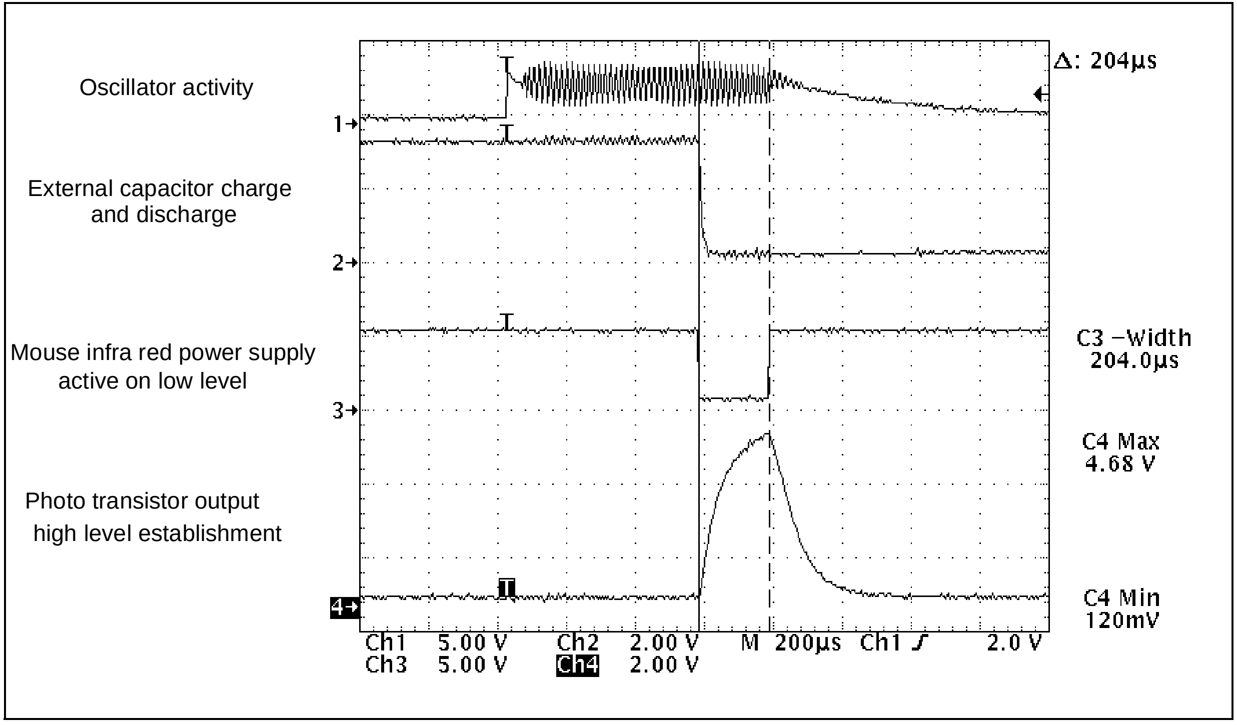


### 5.1.1 Special requirements on Photo transistors

To check if the mouse has moved, it is necessary to compare the current state of the Photo transistor outputs with the previous ones (recorded before entering suspend mode).

As soon as the microcontroller performs program code operations, the Mouse infra-red LED is turned on. However, to establish the previous Photo transistor values (especially in case of high level values) a certain amount of time is required. This period is lower than 200 µs for the components used. This is due to current establishment in the infra-red diode.

**Figure 13. Photo transistors behaviour**



### 5.1.2 RC value proposal

To summarize, the microcontroller is active for 800 µs and it draws 20 mA during this period. Then it is in low-power mode for the rest of the period (drawing about 250 µA). This means it is quite easy to calculate the correct value of the RC time period to enter suspend mode 500 µA average max current specification:

$$Specvalue = \frac{(Imax \times WakeupTime) + (Imin \times (Period - WakeupTime))}{Period}$$

Example:

If you choose 450 µA to stay well within the specification and with the following parameters:

Imax = 20 mA - Wakeup Time = 800 µs - Imin = 250 µA

A Period value of 80 ms is obtained.

The RC value can then be calculated to get this period.

You need to pay attention to 2 parameters:

– The high level voltage imposed by the high trigger level (about 3.4V)

– The input impedance of the pin which modifies the equivalent resistance of the RC circuit.

The following table presents some RC values and the corresponding time periods and consumption measured on a ST7263 microcontroller.

Depending on the application supporting the suspend mode, the developer will have to find a compromise between the suspend interval time and the suspend average current.

**Table 1. RC Value proposal**

| RC Values | Suspend time period | Suspend average current |
|---|---|---|
| C= 330 nF - R= 1 MΩ | t= 306 ms | I= 400 µA |
| C= 330 nF - R= 670 KΩ | t= 200 ms | I= 405 µA |
| C= 330 nF - R= 330 KΩ | t= 97 ms | I= 440 µA |
| C= 100 nF - R= 670 KΩ | t= 80 ms | I= 450 µA |
| C= 100 nF - R= 330 KΩ | t= 37 ms | I= 540 µA **(out of spec.)** |

**Note 1:** The suspend mode consumption has been measured with Ports A and C configured as outputs. This has been done to avoid additional current consumption in their pull-up resistors.

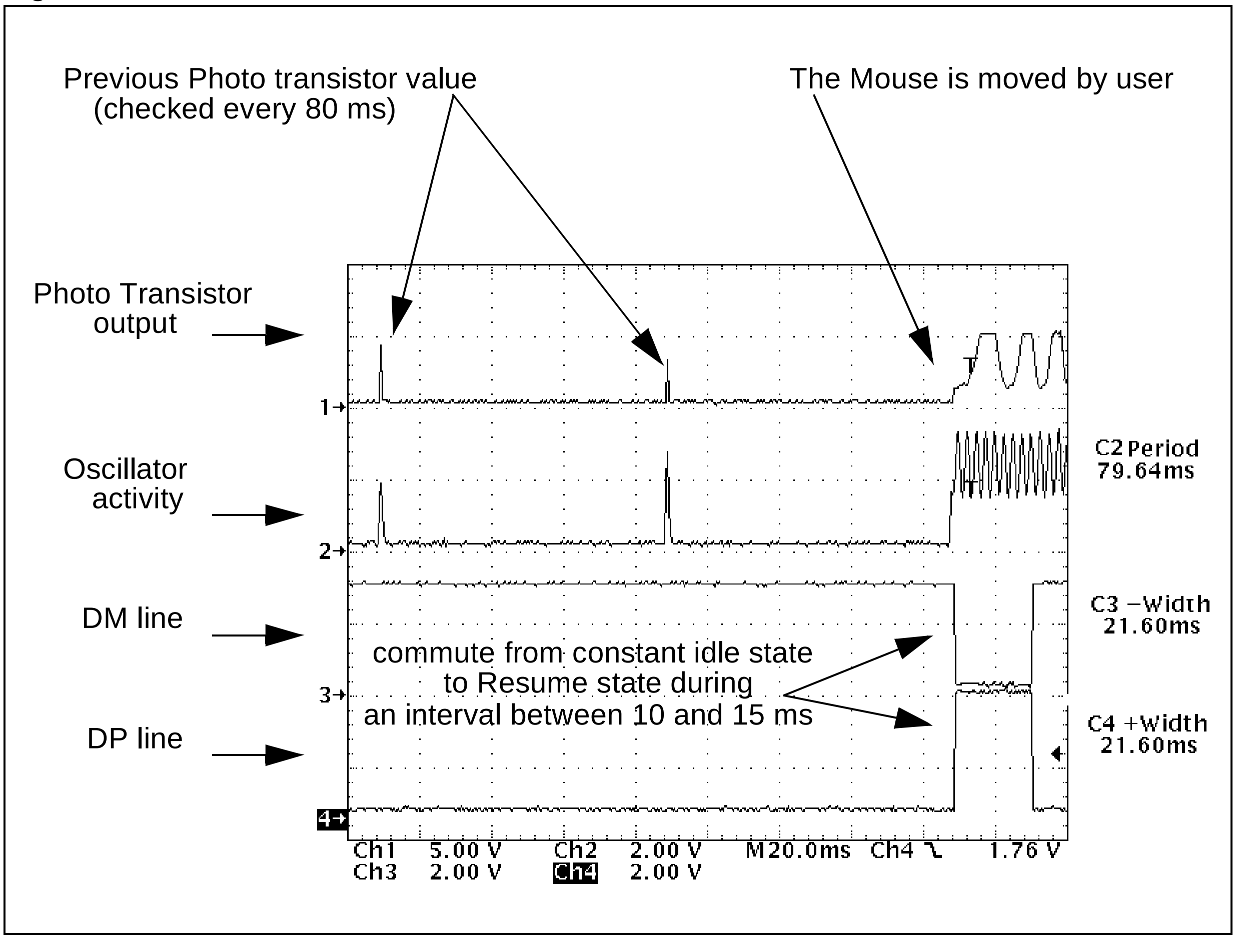
**Note 2:** A USB Mouse is a Human Interface Device. This means a device manipulated at quite low frequencies. The best compromise is obtained by choosing a RC value which allows you to enter in remote wake-up mode and to check the device sensor states frequently (more than 10 times per second).

## 5.2 REMOTE WAKE-UP MODE

As soon as a different value has been read from the buttons or photo transistor outputs, the microcontroller performs a Remote Wake-up sequence to resume the communication flow between the Device and the Host.

An example of such a sequence is shown on Figure 14.

**Figure 14. Resume state behaviour**



Note that the DM and DP lines are maintained in Resume state for 21.6 ms while the USB specification release 1.1 requires the device to impose resume signalling for a period between 10 ms and 15 ms. In fact the device imposes the suspend state for only 12 ms and then the host maintain this state for 21.6 ms. This 20 ms minimum time of resume signalling insures that all devices in the network that are enabled to see the resume are woken-up.

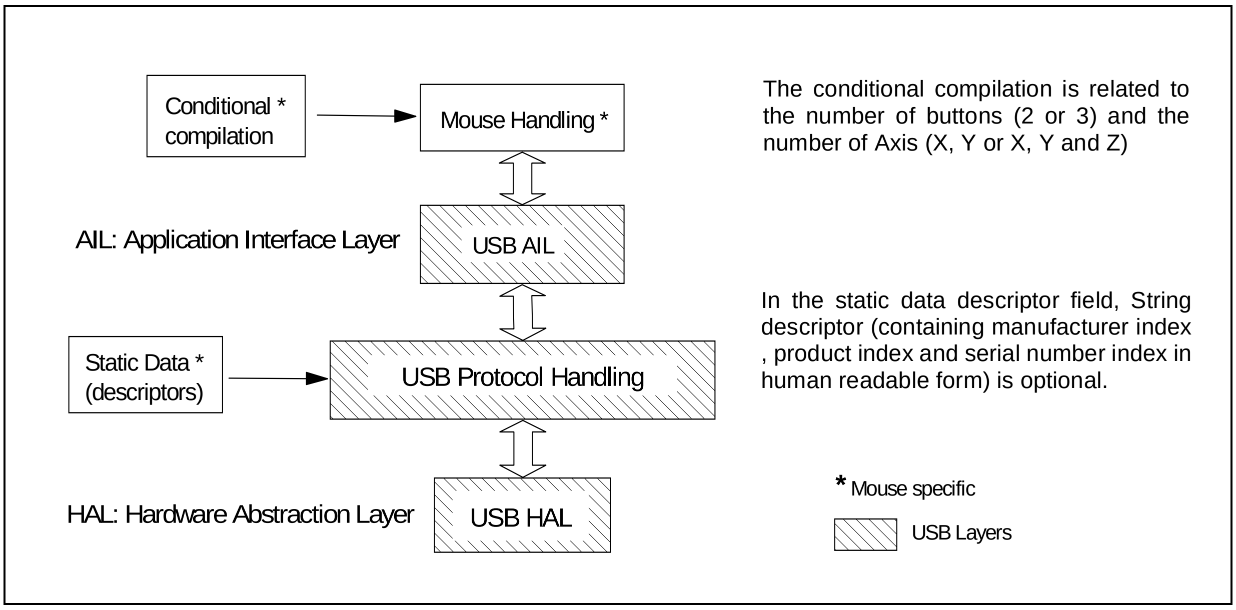
# 6 PROGRAM ARCHITECTURE

## 6.1 FIRMWARE LAYERS

The program architecture is divided in several layers from the hardware register low level layer to the mouse handling high level layer. This makes modifications easier at any level.

This is shown by the Architecture Overview in Figure 15.

**Figure 15. Architecture Overview**


Conditional * compilation
Mouse Handling *
The conditional compilation is related to the number of buttons (2 or 3) and the number of Axis (X, Y or X, Y and Z)
AIL: Application Interface Layer
USB AIL
In the static data descriptor field, String descriptor (containing manufacturer index , product index and serial number index in human readable form) is optional.
Static Data * (descriptors)
USB Protocol Handling
HAL: Hardware Abstraction Layer
USB HAL
* Mouse specific
USB Layers

***USB HAL:*** The USB Hardware Abstraction Layer (HAL) exports the major functions which directly access the ST7263 hardware registers. It handles all registers from the SIE (Serial Interface Engine) macrocell which is the peripheral that handles USB exchanges at the lowest level.

***USB Protocol Handling:*** This layer manages the USB protocol. It handles the different transaction during USB SETUP, DATA and STATUS stages. It replies to the standard requests used for device enumeration (SET_ADDRESS, GET_CONFIGURATION...).

***USB AIL:*** The USB Application Interface Layer (AIL) exports all functions needed for any HID application. It allows the application to receive specific requests from Host software through a Control Pipe (Endpoint 0). It also contains functions for sending data through the Control Pipe (Endpoint 0) or the Interrupt Input Pipe (Endpoint 1).

## 6.2 MOUSE HANDLING ROUTINES

In the following section, all the programs have been developed using the Hiware C compiler Toolchain. ST provides a complete architecture as well as firmware drivers to help you develop your application quickly and easily. A list of reference documents is provided at the end of the application note.

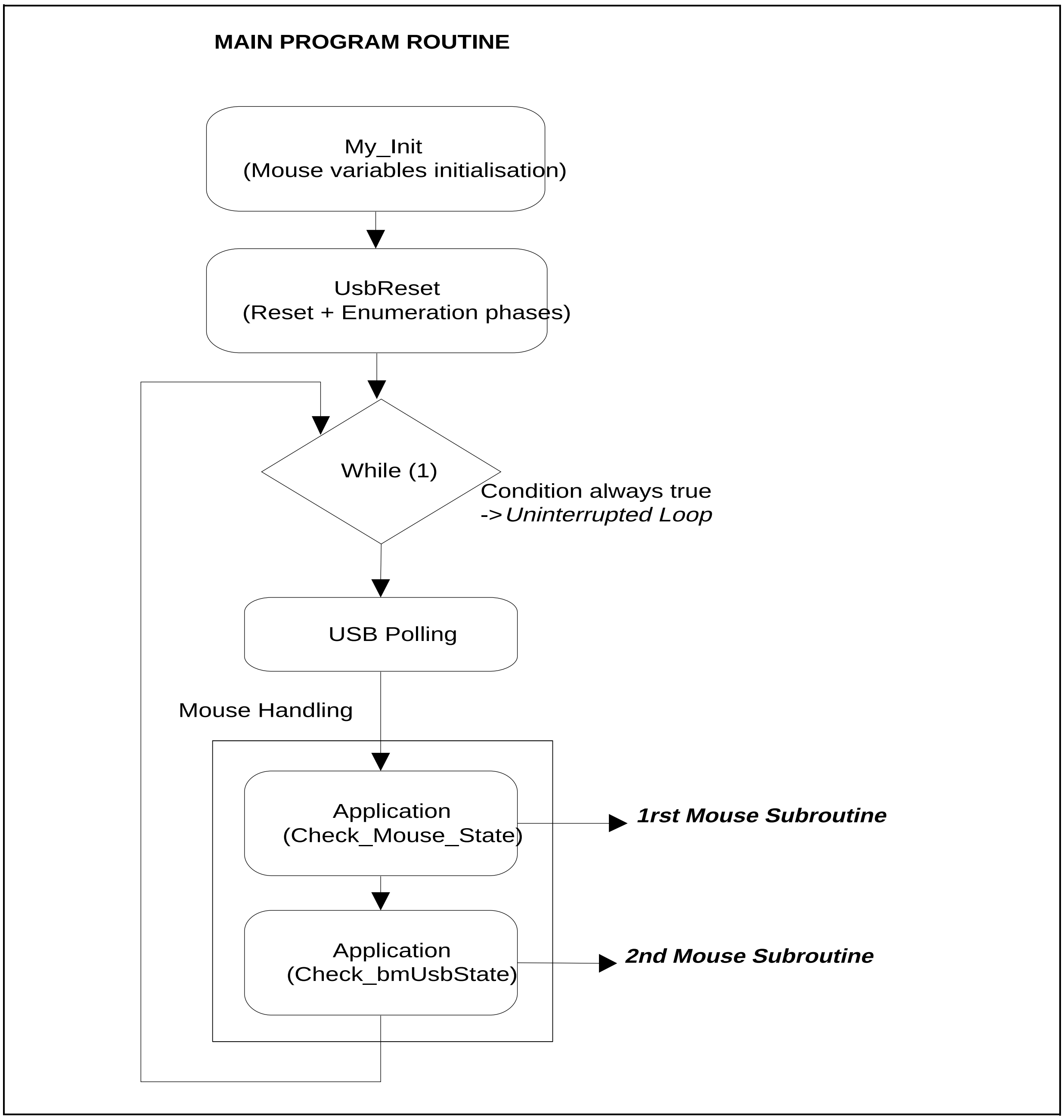
According to the ST7263 microcontroller, all USB events are managed by interrupt. When an USB event occurs, a flag of the Interrupt State Register (ISTR) is set by hardware. Then, the firmware determines the interrupt origin by reading the ISTR register, sets the corresponding bit in a software register (bmUsbIntFlag), and clears the interrupt flag. The USB polling routine reads the software register (bmUsbIntFlag) to determine the USB interrupt source and jumps to the corresponding interrupt routine.

Each interrupt routine sets the global variable “bmUsbState” to the corresponding USB state: SOF, Ennumerated, Suspended or Remote wake-up.

The Mouse handling routine is divided into two subroutines continuously executed by the microcontroller. A first subroutine is affexted to the mouse sensor management. The second subroutines is dedicated to the BmUsbState management.

These two subroutines can be interrupted by USB or other microcontroller interrupt vectors. Some conditional events involve the execution of other subroutines.

**Figure 16. Mouse software subroutines overview**



### 6.2.1 Check_mouse_state subroutine

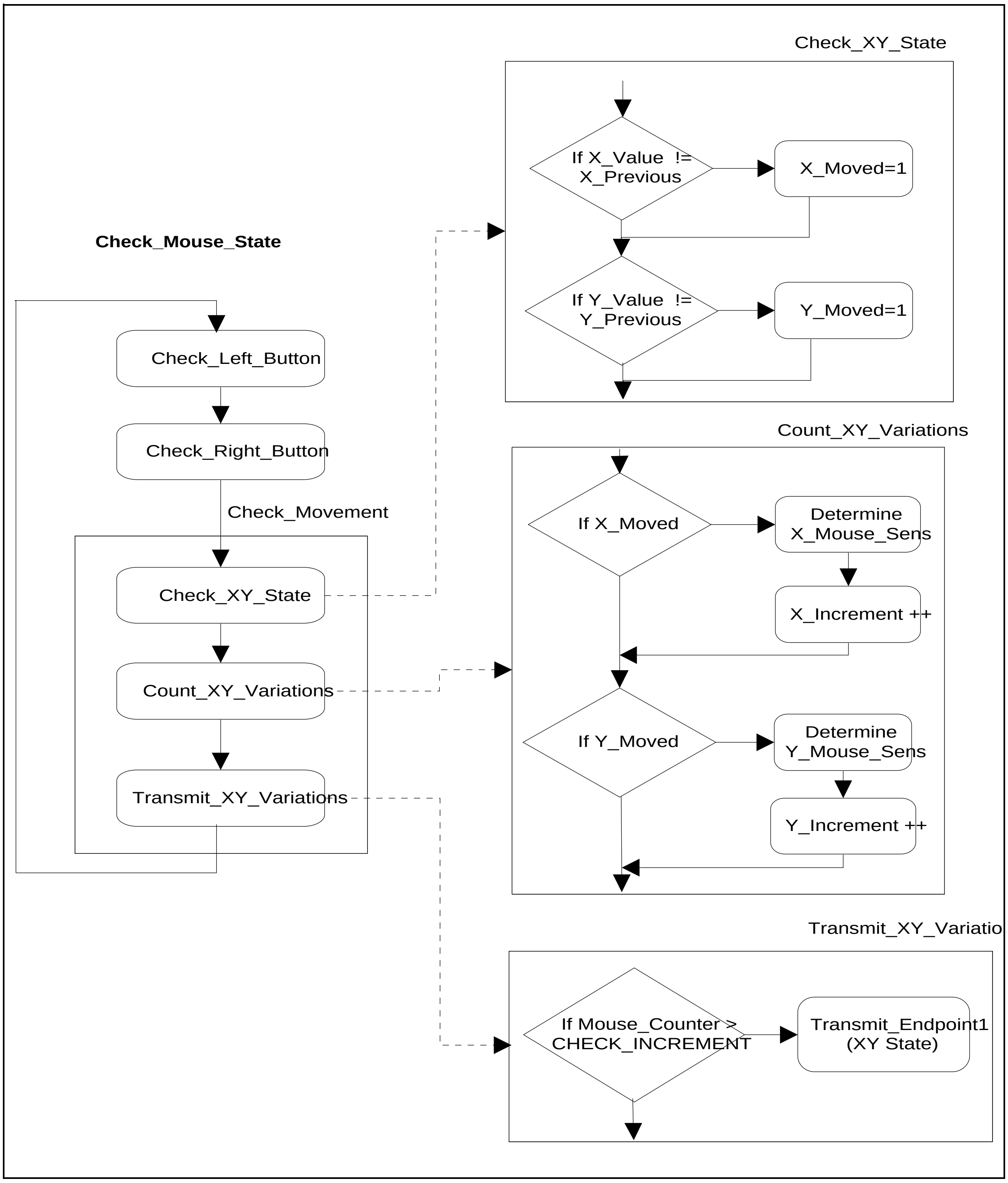
The first subroutine called *check_mouse_state* (*See Figure 17*) handles all the variations on the external sensors (buttons and photo transistors).

This subroutine handles the different states of the buttons and of the photo transistors on the mouse. The mouse state informations are sent only if one of the different sensors has been moved. In the other case, the mouse answers the host IN-tokens by issuing non-acknowledge (NAK) tokens.

If one of the mouse axis has moved, a first subroutine determines the X-Y direction, a second subroutine counts the variation increments and a third subroutine transmits this variation only if the Mouse_counter variable has reached the Check_increment constant value.

The Mouse_counter value is incremented as soon as a Start of Frame (SOF) is encountered. It is refreshed at the end of the transmission subroutine. By default this value is set to 5 in the *usr_var.h* file. The modification of the Check_increment constant value affects the amplitude of the cursor's displacements on the screen of your PC. Then, depending on your display unit format, you can either choose a small or a high amplitude cursor displacement.

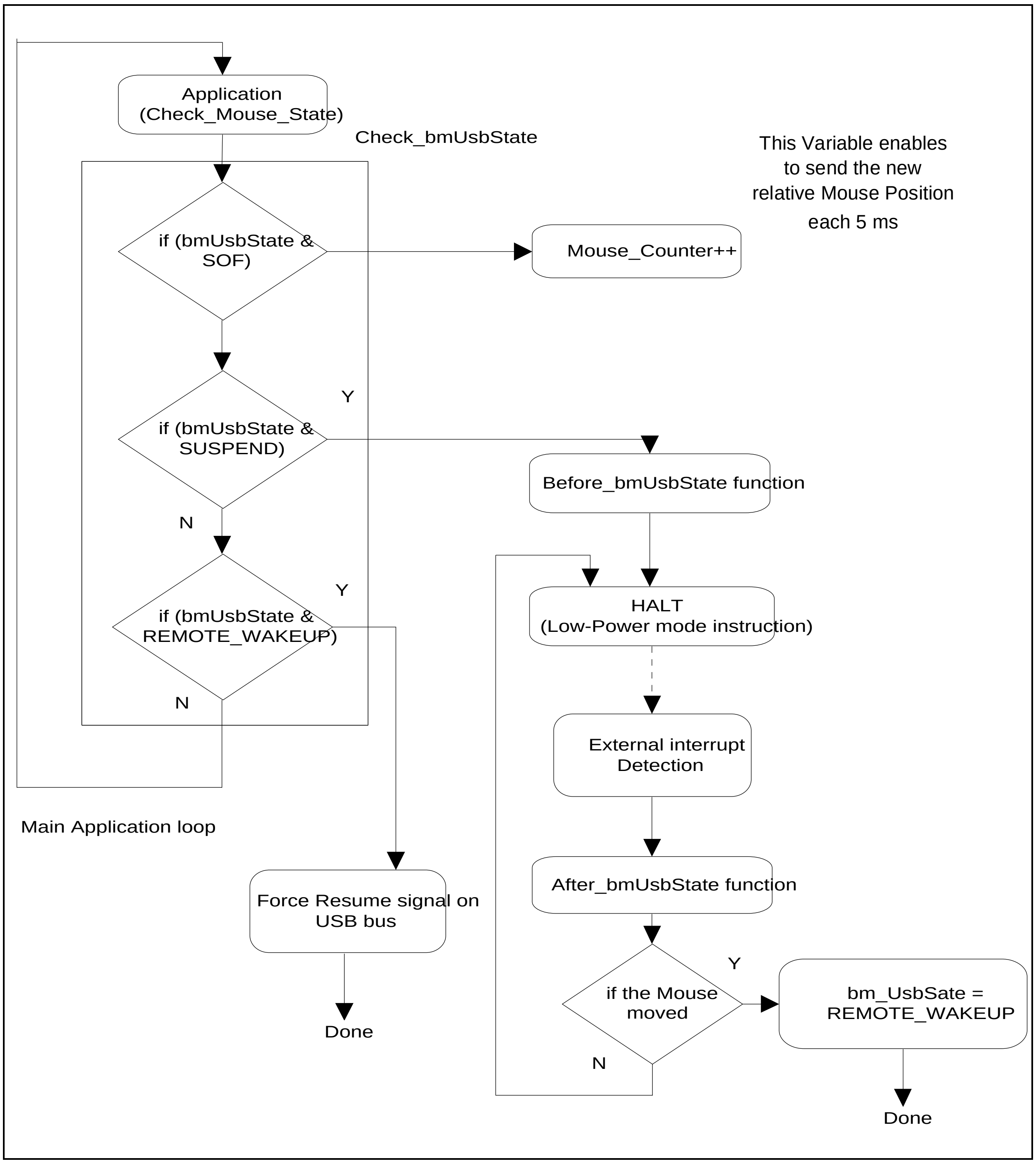
**Figure 17. Check_mouse_state subroutine**



### 6.2.2 Check_bmUsbstate subroutine

The second subroutine called *check_bmUsbstate* (*See Figure 18*) is specifically for the USB bus state variations like Start of Frame (SOF), device Enumerated, suspend mode and remote wake-up mode.

**Figure 18. Check_bmUsbstate subroutine**


Application (Check_Mouse_State)
Check_bmUsbState
This Variable enables to send the new relative Mouse Position each 5 ms
if (bmUsbState & SOF)
Mouse_Counter++
Y
if (bmUsbState & SUSPEND)
Before_bmUsbState function
N
Y
if (bmUsbState & REMOTE_WAKEUP)
HALT (Low-Power mode instruction)
N
External interrupt Detection
Main Application loop
After_bmUsbState function
Force Resume signal on USB bus
Y
if the Mouse moved
bm_UsbSate = REMOTE_WAKEUP
Done
N
Done

Special care must be taken to the port configuration that the microcontroller must perform before enter and after exit Suspend State. This means that all the external devices that can draw current must be disabled. Two dedicated subroutines have been created to handle previous and post Suspend states.

Two time routines have been included in the suspend state condition subroutine:

- The first 50µs time routine is mandatory to completely discharge the external capacitor.

- The second 200µs time routine is mandatory to let the photo transistor reach its previous value after having woken-up the microcontroller. (This last period concerns the establishment of the current in the infra-red diodes).

These two time values can be modified by users according to the characteristics of the external components used in the application (especially external capacitor value and infra-red current establishment time).

## CONCLUSION

Because of their functionality and hot insertion properties, Universal Serial Bus Devices are capable of replacing all previous serial and parallel devices. Low speed Human Interface Devices like mice are especially directed by this protocol.

The above application note describes the implementation of a low cost USB Mouse using the ST7263 microcontroller. All USB requirements like transceiver, Serial Interface Engine, Voltage regulator and DMA architecture have been implemented on the chip around an industry-standard 8-bit optimized core. All these features make the USB Mouse developed with the ST7263 microcontroller efficient, rapid and low-cost.

## REFERENCE DOCUMENTS

– ST7263 Data Sheet

– AN 1017 Using the ST7 Universal Serial Bus Microcontroller

– AN 1069 Developing an ST7 USB Apllicaton

– AN 989 Starting with ST7 Hiware C.

– AN 1064 Writing Optimized Hiware C Language for ST7

A general training for ST7 (hardware and development tools) is available on the ST7 CD-ROM.

The USB Descriptor tool DT2.4 is available on the USB website at URL http//www.usb.org

## C CODE SOURCE

## 1. Check_Mouse_State.c

```
/*---------------------------------------------------------------------------
ROUTINE NAME : Check_Left_Button
INPUT/OUTPUT :
DESCRIPTION :
-------------------------------------------------------------------------*/
void Check_Left_Button(void)
      {
      if(!(PADR & 0x80) && !(Left_Pressed)) // Left Button pressed
      {
      Left_Pressed = 1;
      ReportIdLength = 3;
      ReportIdArray[0] = 0x01; // Report ID for Switches
      ReportIdArray[1] = 0x00; // Report ID for for X Axis
      ReportIdArray[2] = 0x00; // Report ID for for Y Axis
      TransmitEP1(&ReportIdArray[0], ReportIdLength);
      }
      else if((PADR & 0x80) && (Left_Pressed)) // Left Button relaxed
      {
      Left_Pressed = 0;
      ReportIdLength = 3;
      ReportIdArray[0] = 0x00; // Report ID for Switches
      ReportIdArray[1] = 0x00; // Report ID for for X Axis
      ReportIdArray[2] = 0x00; // Report ID for for Y Axis
      TransmitEP1(&ReportIdArray[0], ReportIdLength);
      }
      }

/*---------------------------------------------------------------------------
ROUTINE NAME : Check_Right_Button
INPUT/OUTPUT :
DESCRIPTION :
-------------------------------------------------------------------------*/
void Check_Right_Button(void)
{
      if(!(PADR & 0x40) && !(Right_Pressed)) // Right Button pressed
      {
      Right_Pressed = 1;
      ReportIdLength = 3;
      ReportIdArray[0] = 0x02; // Report ID for the Right Switch
      ReportIdArray[1] = 0x00; // Report ID for X Axis
      ReportIdArray[2] = 0x00; // Report ID for Y Axis
      TransmitEP1(&ReportIdArray[0], ReportIdLength);
```

```
        }
        else if((PADR & 0x40) && (Right_Pressed)) // Right Button Relaxed
        {
        Right_Pressed = 0;
        ReportIdLength = 3;
        ReportIdArray[0] = 0x00; // Report ID for Switches
        ReportIdArray[1] = 0x00; // Report ID for X Axis
        ReportIdArray[2] = 0x00; // Report ID for Y Axis
        TransmitEP1(&ReportIdArray[0], ReportIdLength);
        }
}

/*---------------------------------------------------------------------------
ROUTINE NAME : X_Mouse_Sens
INPUT/OUTPUT :
DESCRIPTION  : checking if the mouse as moved
---------------------------------------------------------------------------*/
void X_Mouse_Sens(void) // Mouse routine for X sign detection (opto sensors on one
axis)
{
        switch (X_Value)
        {
        case 0x80 :
        {
        if (X_Previous == 0x00) X_Sens = 1; // on the right
        else if (X_Previous == 0xC0) X_Sens = 0; // on the left
        break;
        }
        case 0xC0 :
        {
        if (X_Previous == 0x80) X_Sens = 1; // on the right
        else if (X_Previous == 0x40) X_Sens = 0; // on the left
        break;
        }
        case 0x40 :
        {
        if (X_Previous == 0xC0) X_Sens = 1; // on the right
        else if (X_Previous == 0x00) X_Sens = 0 // on the left
        break;
        }
        case 0x00 :
        {
        if (X_Previous == 0x40) X_Sens = 1; // on the right
        else if (X_Previous == 0x80) X_Sens = 0; // on the left
        break;
        }
        default :
        break;
        }
}

/*---------------------------------------------------------------------
ROUTINE NAME : Y_Mouse_Sens
INPUT/OUTPUT :
DESCRIPTION  : checking if the mouse as moved
---------------------------------------------------------------------*/
void Y_Mouse_Sens(void) // Mouse routine for X sign detection (opto sensors on one
axis)
{
        switch (Y_Value)
        {
        case 0x10 :
        {
        if (Y_Previous == 0x00) Y_Sens = 1;// go up
        else if (Y_Previous == 0x18) Y_Sens = 0;// go down
        break;
        }
        case 0x18 :
        {
        if (Y_Previous == 0x10) Y_Sens = 1; // go up
        else if (Y_Previous == 0x08) Y_Sens = 0; // go down
        break;
        }
        case 0x08 :
        {
        if (Y_Previous == 0x18) Y_Sens = 1;// go up
        else if (Y_Previous == 0x00) Y_Sens = 0; // go down
        break;
        }
        case 0x00 :
        {
        if (Y_Previous == 0x08) Y_Sens = 1;// go up
        else if (Y_Previous == 0x10) Y_Sens = 0; // go down
        break;
        }
        default :
        break;
        }
}
```

```
/*---------------------------------------------------------------------------
ROUTINE NAME : Check_Movement
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/
void Check_XY_State(void)
{
      X_Value = (PBDR & 0xC0);  // (PB6 and PB5)
      if ((X_Value != X_Previous) && (X_Moved == 0)) // X-Axis mouse has moved
                              since last routine access
      X_Moved = 1;
      else // ((X_Value == X_Previous) && (X_Moved == 1))
      X_Moved = 0;

      Y_Value = (PBDR & 0x18);  // (PB3 and PB4)
      if ((Y_Value != Y_Previous) && (Y_Moved == 0))// Y-Axis mouse has moved
                              since last routine access
      Y_Moved = 1;
      else
      Y_Moved = 0;
}

/*---------------------------------------------------------------------------
ROUTINE NAME : Count_XY_Variations
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/
void Count_XY_Variations(void)
{
      if (X_Moved)
      {
      X_Mouse_Sens();
      X_Increment++;
      X_Previous = X_Value;
      }

      if (Y_Moved)
      {
      Y_Mouse_Sens();
      Y_Increment++;
      Y_Previous = Y_Value;
      }
}

/*---------------------------------------------------------------------------
```

```
ROUTINE NAME : Transmit_XY_Variations
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/

void Transmit_XY_Variations(void)
{
 ReportIdLength = 3;
      if (Mouse_Counter > 2000)// Reset Mouse_Counter before negative value.
      Mouse_Counter=0;

      if ((Mouse_Counter > CHECK_INCREMENT) && ((X_Increment != 0) ||
                              (Y_Increment != 0)))
      {
            ReportIdLength = 3;
            if ((X_Sens == 0) && (Y_Sens == 0)) // Mouse go on left and down
            {
            ReportIdArray[1] = ~X_Increment; // Negative X Increment Value
            ReportIdArray[2] = Y_Increment;  // Positive Y Increment Value
            }

            if ((X_Sens == 1) && (Y_Sens == 0)) // Mouse go on right and down
            {
            ReportIdArray[1] = X_Increment;  // Positive X Increment Value
            ReportIdArray[2] = Y_Increment;  // Positive Y Increment Value
            }

            if ((X_Sens == 0) && (Y_Sens == 1))  // Mouse go on left and up
            {
            ReportIdArray[1] = ~X_Increment; // Negative X Increment Value
            ReportIdArray[2] = ~Y_Increment; // Negative Y Increment Value
            }

            if ((X_Sens == 1) && (Y_Sens == 1))  // Mouse go on right and up
            {
            ReportIdArray[1] = X_Increment;  // Positive X Increment Value
            ReportIdArray[2] = ~Y_Increment; // Negative Y Increment Value
            }

      TransmitEP1(&ReportIdArray[0], ReportIdLength);
      X_Increment = 0;
      Y_Increment = 0;
      Mouse_Counter = 0;
      }
}
```

```
/*---------------------------------------------------------------------------
ROUTINE NAME : Check_Movement
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/
void Check_Movement(void)
{
      Check_XY_State();
      Count_XY_Variations();
      Transmit_XY_Variations();
}

/*---------------------------------------------------------------------------
ROUTINE NAME : Check_Mouse_State
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/
void Check_Mouse_State(void)
{
      Check_Left_Button();
      Check_Right_Button();
      Check_Movement();
}

/*---------------------------------------------------------------------------
ROUTINE NAME : MOUSE_HANDLING
INPUT/OUTPUT :
DESCRIPTION :
---------------------------------------------------------------------------*/
void MOUSE_HANDLING(void)
{
      Check_Mouse_State();
      Check_BmUsbState();
}
```

## 2. Check_bmUsbState.c

```
/*-------------------------------------------------------------------------------
ROUTINE NAME : Before_Enter_Suspend
INPUT/OUTPUT :
DESCRIPTION : This function details what we have to do before entering in suspend
mode
-----------------------------------------------------------------------------*/
void Before_Enter_Suspend(void)
{
unsigned int k;// variable used in the capacity discharge loop

      PADDR = 0xFF; // The 2 Ports are configured in output for min current con-
                              sumption care
      PCDDR = 0xFF;

      SetBit(PBDDR,5); // PB5 in output mode (external RC circuit)
      ClrBit(PBDR,5);  // Clr PB5 : Discharge the external capacity

      for (k=17; k>0;k --){ // 50us loop: This time interval is mandatory to
      asm nop;// completly discharge the external capacity
      }

      Previous_Suspend_State = (PBDR & 0xD8); // Read the Phototransistor value
                              before entering in SUSPEND
      Current_Suspend_State = Previous_Suspend_State;
      SetBit(PCDR,0);   // Switch OFF the Mouse LED
      ClrBit(PBDDR,5);  // PB5 configured in input mode (We let the capacity
                              charge)
      Wake_Up_Flag = 0;
      ITRFRE |= 0x20;   // Enable IT6 (PB5)
}

/*-------------------------------------------------------------------------------
ROUTINE NAME : After_End_Suspend
INPUT/OUTPUT :
DESCRIPTION :
-----------------------------------------------------------------------------*/
void After_End_Suspend(void)
{
unsigned int j; // variable used in the 200 us loop

      if (Wake_Up_Flag == 1)
      {
      ClrBit(PCDR,0); // switch on the Mouse LED
```

```
        SetBit(PBDDR,5); // PB5 in output mode (external RC circuit)
        ClrBit(PBDR,5);  // Clr PB5 : Discharge the external capacity

        ITRFRE |= 0x20; // Re enable IT6

                for (j=68; j>0; j--){ // 200us loop: This time interval is mandatory
                                        to let
                asm nop; // each Phototransistor reach his previous value
                }
        }
}

/*--------------------------------------------------------------------------
ROUTINE NAME : Check_BmUsbState
INPUT/OUTPUT :
DESCRIPTION :
--------------------------------------------------------------------------*/
void Check_BmUsbState(void)
{
        if(bmUsbState & SOF) // Start of frame occurs
                {
                bmUsbState &= ~SOF; // Reset the SOF bit
                Mouse_Counter++; // This variable is used to refresh the Cursor Posi-
                                        tion
                }

        if(bmUsbState & ENUMERATED) // Device is enumerated
                asm nop;

        if(bmUsbState & SUSPEND)
                {
                Before_Enter_Suspend();
                asm
                {
                Loop: halt; // Enters suspend mode
                }
                /***** return from interrupt in SUSPEND Mode *******/

                After_End_Suspend();
                Current_Suspend_State = (PBDR & 0xD8); // Filter on the 4 X-Y axis
                                        Optocoupler sensors
                if (Current_Suspend_State == Previous_Suspend_State)
                        {
                        Current_Suspend_State = Previous_Suspend_State;
                        SetBit(PCDR,0);  // switch OFF the Mouse Led
```

```
                  ClrBit(PBDDR,5); // PB5 configured in input (We let the capaci-
                             ty charge)
                  ITRFRE |= 0x20;  // Re enable IT5
                  Wake_Up_Flag = 0;
                  asm jp Loop;
                  }
            else
                  {
                  ITRFRE = 0x00; // Disable IT6 (PB5)
                  bmUsbState &= ~SUSPEND; // Reset "Go Suspend"
                  bmUsbState |= REMOTE_WAKEUP; // Set "Remote Wake-up"
                  ClrBit(PBDDR,5); // PB5 configured in input
                  }
            }

      if(bmUsbState & REMOTE_WAKEUP)
            {
            RemoteWakeup();
            }
}
```

## Int7263.c

```
/*----------------------------------------------------------------------------
ROUTINE NAME : INT_IT1IT8
INPUT/OUTPUT : None
DESCRIPTION  : Rising edge interrupt
COMMENTS     :
----------------------------------------------------------------------------*/
#pragma TRAP_PROC SAVE_REGS
void INT_IT1IT8(void)
{
       if(bmUsbState & SUSPEND) // We are in suspend mode
      {
      if((PBDR & 0x20) == 0x20)
      Wake_Up_Flag =1;
      ITRFRE |= 0x20;
      }
}
```