

1. Abstract	2
2. Introduction to USB	2
2.1 <i>Transfer types</i>	3
2.1.1 Control transfers.....	3
2.1.2 Interrupt Transfers.....	4
2.1.3 Bulk transfers:.....	4
2.1.4 Isochronous transfers:	4
2.2 <i>The HID Class</i>	4
2.3 <i>HID Class Commands:</i>	4
3. The H8SX/1664 USB Peripheral	5
4. Implementation	6
4.1 <i>The HID application</i>	7
4.1.1 Overview.....	7
4.1.2 Operation.....	7
5. Using the HID application in your project	10
6. Limitations	11
7. Data Sheet	11
8. References	11

Using the H8SX/1664 as a HID class device

1. Abstract

The following application note introduces the USB HID class and shows an example of how to configure the USB block on the H8SX/1664 and use the microcontroller as a HID class device. This document refers to the RSK/H8SX1664 USB kit and specifically to the included HID application example.

2. Introduction to USB

The USB (Universal Serial Bus) is an interface and a protocol that allows a single host computer to communicate with a variety of peripheral devices. The USB 2.0 spec defines this interface. Although it is dependant on the application most USB projects will require a host side interface app and the device firmware. Every USB communication is between a host and a device, where the host controls the bus and initiates communication all the time, except in case of devices with the remote-wakeup feature (USB On-The-Go allows for devices to negotiate for the role of host and thus bus control). In comparison with other interfaces, USB offers a host of advantages which include, automatic configuration (enumeration), minimum IRQ lines used, hot pluggable, low cost, low power consumption, speed and reliability. Depending on the application, the developers can chose one (or more) of four USB transfer types for the project; Control, Bulk, Interrupt and Isochronous. These classifications are based on frequency of transfer, amount of data to be transferred and the kind of data being transferred.

In USB terminology, individual devices are referred to as *functions*, which are linked in series through *hubs*. The hubs are special-purpose devices that are not considered functions. There always exists one hub known as the root hub, which is attached directly to the host controller.

Endpoints:

Functions and hubs have associated *pipes* (logical channels). Pipes are connections from the host controller to a logical entity on the device named an *endpoint*. The end point thus serves as a data buffer; typically it is a block of data memory or a register in the device; each endpoint can transfer data in one direction only (except endpoint 0), either into or out of the device/function, thus making pipes unidirectional. Every device has endpoint zero configured for bidirectional control transfer. The number of available endpoints and supported transfer types vary with each device. The different kinds of endpoints are Bulk, Control, Interrupt and Isochronous. Since endpoints are unidirectional, they will be followed by an “in” or “out” specification (e.g. Bulk-In).

Device Information:

To identify itself as a USB device and to conform to the spec for a certain class, a device needs to have in its firmware certain elements of information that the host can access in order to successfully enumerate and then communicate with the device. These elements are broadly known as Descriptors and they are further classified into:

- a) Device descriptor: Information such as the device class, the device sub-class, number of configurations, max packet size and other info about the device as a whole are present in this descriptor.

- b) Configuration Descriptor: Information about the number of interfaces supported and power consumption is provided in this descriptor; most devices usually support only a single configuration, but multiple configurations are allowed.
- c) Interface descriptor: Each interface on the device has its own descriptor and subordinate descriptors (descriptors for endpoints used in the interface).
- d) Endpoint Descriptors (at least 2): Endpoint descriptors contain information about the endpoints to be used in that interface. This includes maximum packet size, polling rate, endpoint type (Interrupt, Bulk, Control or Isochronous) and endpoint direction (in or out).
- e) Report Descriptor: This descriptor is required only in case of HID class devices and contains information on the format of data being transmitted.
- f) String Descriptor: Human readable information i.e. messages to be displayed on device enumeration etc are stored in this descriptor. It is optional.

Enumeration:

Before the host can begin using a USB device, it has to learn about the device capabilities, resources and other features in order to assign a device driver. The procedure by which a device identifies itself (including all resources and capabilities available) to the host is known as enumeration. When a function or hub is attached to the host controller through any hub on the bus (including the root hub), it is given a unique 7 bit address on the bus by the host controller. On any USB system all communication is initiated by the host. The host uses a specific set of requests to retrieve required information from the device. These requests can be classified as standard requests and class-specific requests. There are eleven standard requests in the USB spec of which the HID class uses two; the Set_Descriptor and the Get_Descriptor requests.

The Get_Descriptor command is used to retrieve descriptors. The Set_Descriptor request lets the host change descriptors in the device. The host controller then polls the bus for traffic, usually in a round-robin fashion, so no function can transfer any data on the bus without explicit request from the host controller.

Frames:

USB establishes a 1 millisecond time base called a frame on a full-/low-speed bus. A frame can contain several transactions. Each transfer type defines what transactions are allowed within a frame for an endpoint. Isochronous and interrupt endpoints are given opportunities to access the bus every N frames. This information is set in the “*Interval*” or Polling Interval field in the endpoint descriptor. In the provided sample code, it is set to 0x0a for the Interrupt endpoint used by the HID. Thus the device will be polled by the host every 10 milliseconds. For Bulk endpoints, this field is not applicable.

2.1 Transfer types

Four transfer types are supported by the USB spec:

2.1.1 Control transfers

Control transfers are facilitated by the device control endpoint (endpoint zero). The host uses control transfers to configure the device, request device information and other settings. Control transfers are different from other transfers in that they have stages; typically three stages. The host sends a request in the Setup stage; the Data stage is used by the host/device to send data (not all requests have this stage) and the device reports the status information in the Status stage. Control transfers may also be used to send vendor specific requests.

2.1.2 Interrupt Transfers

Interrupt transfers are typically periodic communication requiring bounded latency. An Interrupt request is queued by the device until the host polls the USB device asking for data. These transfers require an Interrupt-In endpoint on the device.

2.1.3 Bulk transfers:

Bulk transfers can be used for large bursty data. It is ideal in situations where the transfer rate is not critical. Data transfer using bulk transfers are very fast if the bus is idle; if the bus is busy, the transfers are delayed. This type of transfer is supported only by Full-Speed and High-Speed devices and require a Bulk-In endpoint and a Bulk-Out endpoint for data to and from the PC respectively.

2.1.4 Isochronous transfers:

Isochronous transfers occur continuously and periodically. They typically contain time sensitive information, such as an audio or video stream. There is no retry or guarantee of delivery, although for the kind of application it is designed for, loss of a packet or frame does not cause critical issues with application performance e.g. audio or video glitches too small to be noticed by the user. This transfer mode is supported only by Full and High speed USB devices.

Refer to Universal Serial Bus Specification Revision 2.0 on usb.org for more details.

2.2 The HID Class

USB devices are categorized into various classes based on common behavior and protocols for devices that serve similar functions. By definition the HID (Human Interface Device) Class consists of devices humans interact with in the course of operating a computer system and some examples of this type of devices are keyboards, mice, joysticks etc.

However the device does not necessarily have to be a “human interface” device to utilize the USB HID class driver; any device that functions within the specifications of the USB HID class can use the protocol for operation. The HID class can thus include any device that sends or receives data at moderate rates, including devices that define a maximum time between transfers. On the H8SX/1664 the theoretical maximum data transfer rate that can be achieved using Interrupt and Control end points (HID class endpoints) is 6.4 Kbytes/sec; however this number is dependant on a variety of factors including bus access time requested by the device.

2.3 HID Class Commands:

The HID class spec defines 6 commands which are listed below.

bRequest Field Value	Command	Meaning
0x01	GET_REPORT	Transfers HID data from the device to the host PC through control transfer.
0x02	GET_IDLE	Returns the current value for the rate of time for which interrupt transfer stops.
0x03	GET_PROTOCOL	Reports the current active protocol (boot protocol or report protocol).
0x09	SET_REPORT	Transfers HID data from the host PC to the device through control transfer
0x0A	SET_IDLE	Specifies the rate of time for which interrupt transfer stops.
0x0B	SET_PROTOCOL	Specifies the active protocol (boot protocol or report protocol).

Table 1: HID class commands

A HID class device that requires BIOS support so that it can use a simple protocol and function, before the Operating System and the full fledged HID driver is loaded, is known as a boot device e.g. keyboards, mice. All HID class devices have to support the GET_REPORT command while all HID boot devices have to support the SET_PROTOCOL and GET_PROTOCOL commands. The remaining commands are optional and need to be implemented only if necessary for the application.

Refer to Device Class Definition for Human Interface Devices (HID) version 1.11 on usb.org for more details.

3. The H8SX/1664 USB Peripheral

The H8SX CPU is a high-speed CPU with an internal 32-bit architecture that is upward compatible with the H8/300, H8/300H, and H8S CPUs.

The main features of the USB peripheral are:

- On-chip UDC (USB Device Controller) conforming to USB 2.0
- USB standard version 2.0 full-speed (12 Mbps) transfer supported
- Automatic processing of USB standard commands for endpoint 0. (Some commands need to be processed through the firmware)
- Four transfer modes supported (Control, Bulk, Interrupt and Isochronous)
- 16 interrupt signals
- On-chip bus transceiver
- Power mode: Self power mode or bus power mode can be selected by the power mode bit (PWMD) in the control register (CTLR).

Endpoint	Name	Transfer Type	Max Packet Size (bytes)	FIFO Capacity	Buffer	DMA Transfer
0	EP0s	Setup	8	8 bytes		
	EP0i	Control-in	8	8 bytes		
	EP0o	Control-out	8	8 bytes		
1	EP1	Bulk-Out	64	128 bytes		Available
2	EP2	Bulk-In	64	128 bytes		Available
3	EP3	Interrupt-in	8	8 bytes		

Table 2: Endpoint Configurations

Commands decoded by hardware	Commands not decoded by hardware
Clear Feature	Get descriptor
Get Configuration	Synch Frame
Get Interface	Set Descriptor
Get Status	Class/Vendor command
Set address	
Set Configuration	
Set Feature	
Set Interface	

Table 3: Standard USB command support

The applications included use commonly used USB functions and procedures. These functions and general code flow are described in the following sections.

4. Implementation

Power On:

As with all H8SX microcontrollers, the majority of peripherals are in Module Stop Mode when the device comes out of reset. To use the peripherals they have to be taken out of Module Stop Mode. This is no different for the USB peripheral.

The function **HardwareSetup()** which is called as part of the Power-on Reset exception configures the system clock, configures port pins and interrupts for switches SW1, SW2 and. The function **USBPreInitSetup()** enables the USB module, and configures the USB port pins and interrupt vectors to be used. the function **SetEPInfo()** configures the endpoints by filling in the EPIR register array, and enables different USB interrupt sources

When an Interrupt Flag is set an interrupt will be generated by the USB peripheral. The USB peripheral can 'direct' this to 1 of 2 interrupt vectors, depending on the settings of the ISRn registers (USB Interrupt Select Registers). If the corresponding bit is cleared to 0, the interrupt request will be handled by interrupt vector 234, USBINTN2. If the corresponding bit is set to 1, the interrupt request will be handled by interrupt vector 235, USBINTN3. As many Interrupt Flags can generate the same interrupt, the Interrupt Service Routine (ISR) has to determine which interrupt has occurred. This is done by interrogating the IFRn registers (USB Interrupt Flag Registers).

Once the initial setup is complete, the application waits for a USB cable to be connected, which will generate a VBUS interrupt.

USB Cable Plugged In/Out:

The VBUS interrupt is handled by USBINTN2. The interrupt interrogates the USB Interrupt Flag Registers and in response to the VBUS interrupt, calls the **HandleVBus()** function.

If a USB cable has been connected, the VBUS interrupt clears all of the USB FIFOs and outputs a logical '1' via Bit-4 of Port M. The output of this logic pulls the D+ line high via a 1.5kΩ resistor to indicate that the USB interface is Full Speed. The application will now wait for the next interrupt, which should be the Bus Reset Interrupt from the host PC. If the USB cable has been disconnected, the VBUS interrupt clears Bit-6 of Port 3 to '0'. The application will now wait for the USB cable to be connected.

The function **HandleBusReset()** simply clears all of the FIFOs and ensures that Stall conditions for all Endpoints are cleared. The application now waits for a Setup Command from the host and which will generate the USBINT2 interrupt.

Enumeration:

In response to the SetupTS flag being set the function **HandleSetupCmd()** is called. This function calls **ReadSetupPacket()** which reads the data from the UEPDR0s (USB Endpoint Data Register 0s) register. UEDPR0s stores the 8-Byte command sent to the host during set up. The function **ReadSetupPacket()** assigns the 8-Byte command to the union **g_oSetupData**. With the data assigned to **g_oSetupData** the function **DecodeSetupPacket()** determines what has been requested. The functions **DecodeStandardSetupPacket()**, **GetDescriptorString()**, **DecodeClassSetupPacket()** and **WriteControlInPacket()** are subsequently called enabling the USB peripheral to successfully enumerate with

the host PC. As part of the enumeration process, the H8SX/1664 provides device information to the host PC by way of descriptors. The descriptors are defined in the files **usbdescriptors.c** & **usbdescriptors.h**.

4.1 The HID application

4.1.1 Overview

The sample HID application allows the user to control the state of the four LEDs on the RSK/H8SX1664 and read switch status via a PC application. Commands to toggle the LEDs and read switch status are transmitted to and from the host PC using Interrupt Transfer and HID Reports.

Figure 1 shows a screen shot of the PC application which is used to control the H8SX/1664.

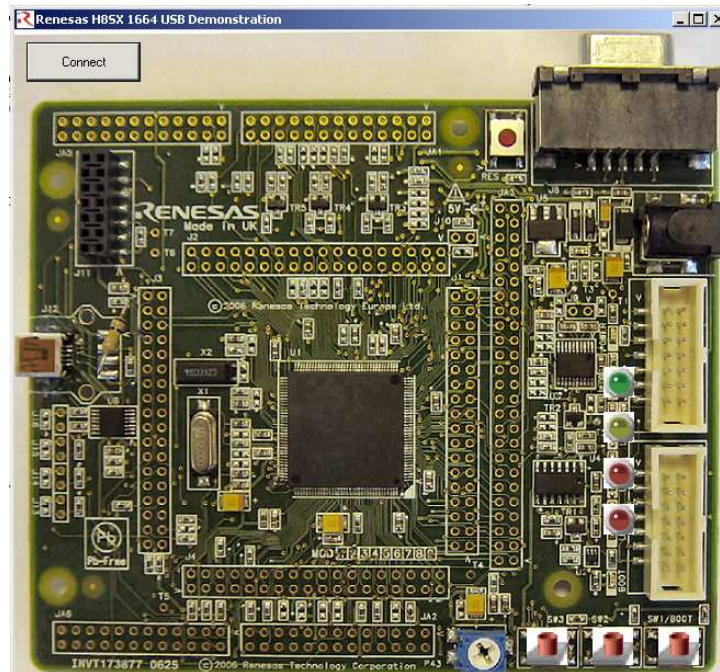


Figure 1: HID application GUI

When the USB cable is connected the device will enumerate. Since the application enumerates as a standard HID class device, an inf file is not required; standard Windows HID class drivers are used by the host. All of the information describing the H8SX/1664 is sent to the host during the enumeration process. The information is held in the files **usbdescriptors.c** and **usbdescriptors.h**. One of the parameters, which the enumeration process provides, is the report size. In this application, the report size is specified as 2 bytes for In (INPUT_REPORT_SIZE) & Out transfers (OUTPUT_REPORT_SIZE).

If the enumeration process is successful, clicking the “Connect” button will gray it out.

4.1.2 Operation

When the H8SX/1664 is connected to the host PC, the device will perform enumeration. When the device has enumerated, the data can be transferred to and from the host PC.

When data is successfully received by the device, the function **HandleEP0oTS()** will be called. This copies the data from Endpoint EP0o, Control_out transfer to a local store, **g_OutputReport.Data[]** and sets a software flag **g_OutputReport.ReceivedFlag**. The software flag can be monitored by an application if it needs to know when HID data is received, but is not used in this example.

The received data in a local variable **g_OutputReport.Data[]** which is a OUTPUT_REPORT_SIZE byte array; where the first byte contains LED status and the second byte contains switch status. On receiving data from the host, the value in **g_OutputReport.Data[0]** is used to toggle the state of the LEDs on the board.

Switch	External Interrupt	Switch Status modification
SW1	IRQ2 (SW1InterruptHandler())	<code>g_InputReport.Data[g_ucSwitchPosition] ^= 0x01</code>
SW2	IRQ4 (SW2InterruptHandler())	<code>g_InputReport.Data[g_ucSwitchPosition] ^= 0x02</code>
SW3	IRQ7 (SW3InterruptHandler())	<code>g_InputReport.Data[g_ucSwitchPosition] ^= 0x04</code>

Table 4: Switch triggered events

When any one of the switches (SW1-SW3) is pressed, an IRQ will be generated which calls one of the associated **SWInterruptHandler()** ISR and toggles the associated bit for that switch in the variable **g_InputReport.Data[g_ucSwitchPosition]** (refer to Table 4). It then calls **SwitchHandler()** which re-enables the de-bounce timer and calls **WriteEP3INData()** which copies the data from the local variable to the EP3 FIFO and then sets the EP3PKTE bit. This generates a trigger to enable transmission from EP3 FIFO. The USB peripheral then performs the necessary USB protocol handling; on receiving an IN token for endpoint 3 from the USB host the data in the EP3 FIFO will be transmitted

Refer to the following flowcharts for a function hierarchy and code flow.

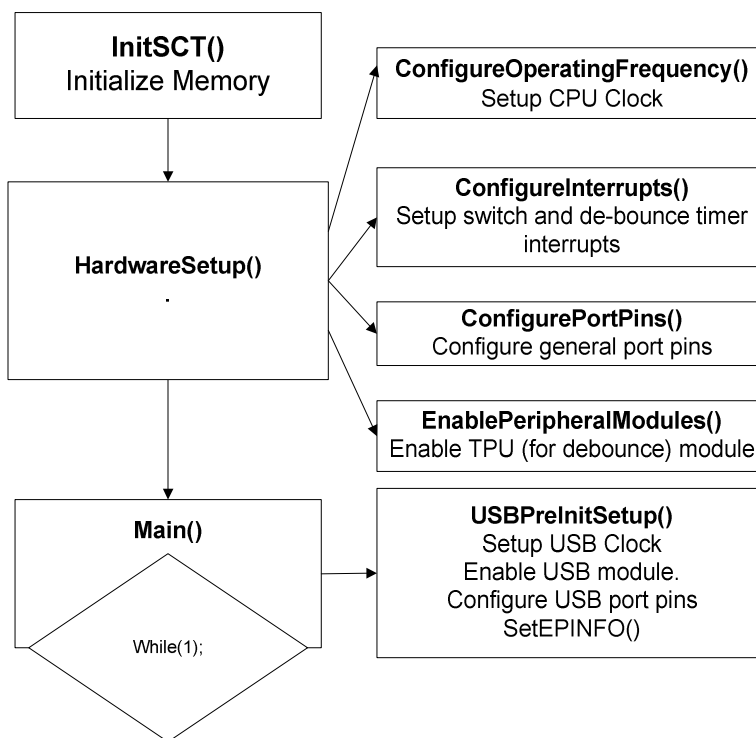


Figure 2: System Initialization

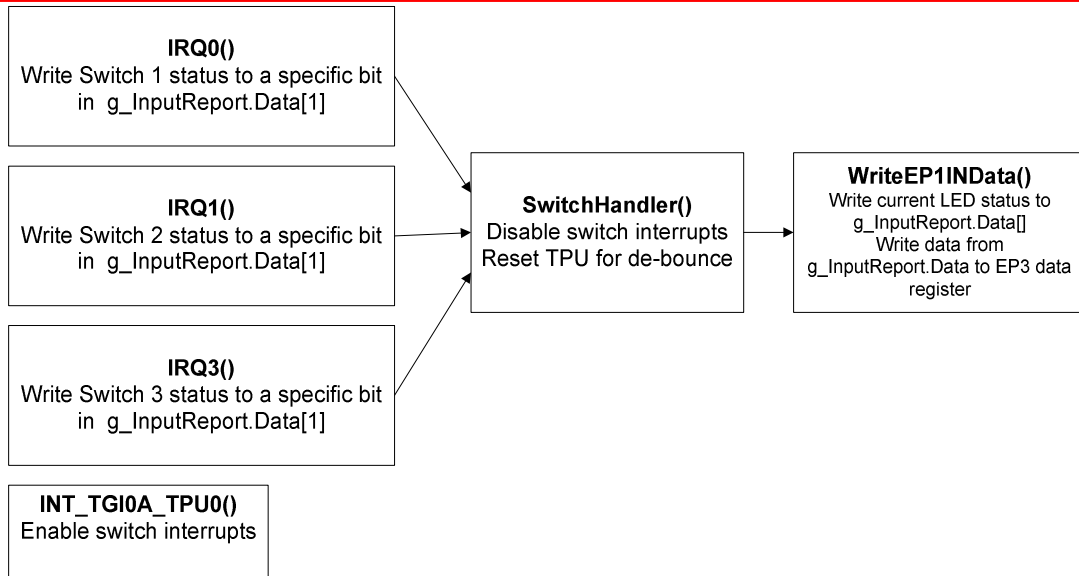


Figure 3: Switch and Timer Interrupts

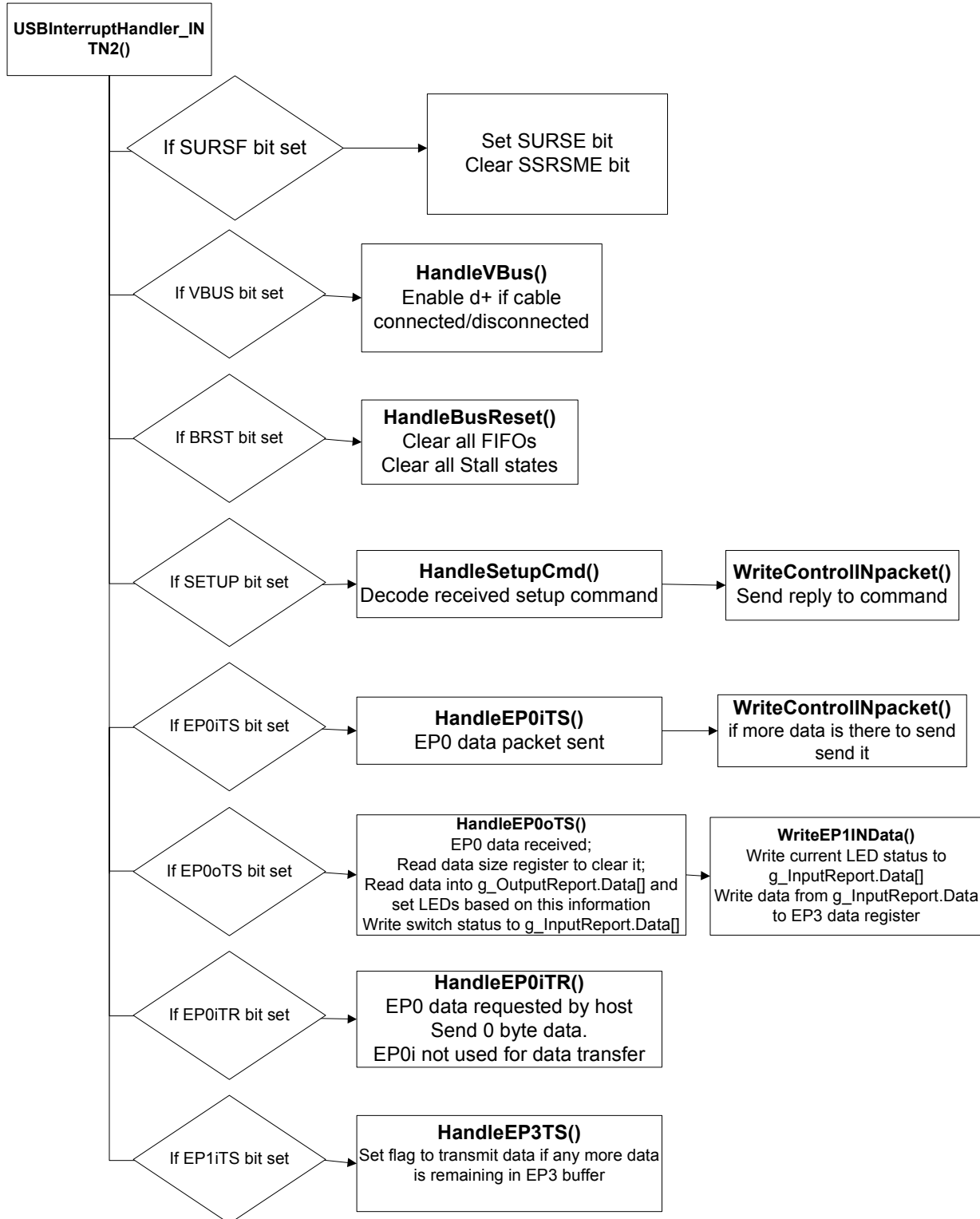


Figure 4: USBInterruptHandler INTNT2

5. Using the HID application in your project

To implement HID communication in your application, use the files `usb.c` & `usb.h` and `usbdescriptors.c` & `usbdescriptors.h` in your project. To receive and send data, your application only needs to access the below mentioned buffers and functions.

The function **HandleEP0oTS()** moves data received from the host over the control pipe to the structure **g_OutputReport**. The function **WriteEP3INData()** moves data from the structure **g_InputReport** to the USB hardware buffers so that they can be transmitted to the host over the Interrupt pipe.

To customize the code so that your project can use the HID interface to communicate with a host, modify **WriteEP3INData()** so that the appropriate data to be transmitted is written into **g_InputReport.Data**. This structure also has a flag **g_InputReport.TransmittedFlag** that can be set to indicate status to a higher layer.

The data received from the host is in the buffer **g_OutputReport.Data**. This structure also has a flag **g_OutputReport.ReceivedFlag** that can be set to indicate status to a higher layer.

Data is transferred from the device to the host via EndPoint1i (Interrupt Pipe) using report descriptors. The current size of the data in the output report is specified by "OUTPUT_REPORT_SIZE", and is declared in the report descriptor. The size of the input report is specified in the same structure as well. Depending on the application requirements, the size of the report descriptor and the output report size have to be changed. The macros OUTPUT_REPORT_SIZE and INPUT_REPORT_SIZE should not have a value more than maximum packet size for the respective endpoints.

The polling interval is currently set to 10 ms (0x0A), in the Endpoint descriptor. This value can be further reduced or increased depending on the application requirements although operation at a selected speed is subject to the capabilities of the host USB controller.

In order to use vendor specific commands in the code, a function **DecodeVendorSetupPacket** has to be written to decode the vendor setup command. The **DecodeSetupPacket** function in **USB.c** has to be modified to call **DecodeVendorSetupPacket** when a vendor setup command is received.

6. Limitations

This implementation is not a full fledged HID class driver. Currently only the SET_REPORT HID class request is supported and on receiving other class specific requests the system will enter a STALL state. Refer to the **DecodeClassSetupPacket()** function for more details on this.

It also supports only the GET_DESCRIPTOR standard request. Refer to the **DecodeStandardSetupPacket()** function for more details on this.

The maximum supported Input and Output report sizes are limited to the maximum packet size for the respective endpoints.

On Lenovo/IBM T43 and T60 series of laptops, the Windows GUI has been known to give problems and displays a "Communication Failure Error". The application has been successfully tested for Chapter9 compliance and works as expected on other PC and laptop brands. If you encounter this issue with Lenovo/IBM laptops, please use another make and model PC for the GUI application.

7. Data Sheet

1. H8SX/1664 group manual. Document number: REJ09B0294-0100
(Use the latest version on the home page: <http://www.renesas.com>)

8. References

1. H8SX/1664 group manual. Document number: REJ09B0294-0100

2. Universal Serial Bus Specification Revision 2.0
3. Device Class Definition for Human Interface Devices (HID) version1.11
4. *"USB Complete: Everything You Need to Develop Custom USB Peripherals"* by Jan Axelson

Keep safety first in your circuit designs!

- Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
- Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (<http://www.renesas.com>).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.