1. Abstract ..... 2
2. Introduction to USB ..... 2
2.1 Transfer types ..... 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 The “Bulk Only” class ..... 4
3. libusb-win32 ..... 5
4. The H8SX/1664 USB Peripheral ..... 5
5. Implementation ..... 6
5.1 H8SX/1664 USB Operation ..... 7
5.2 Windows application ..... 8
6. Using the “Bulk-Only” application in your project ..... 11
7. Limitations ..... 11
8. Data Sheet ..... 11
9. References ..... 11

# Using the H8SX/1664 as a Bulk only device

## 1. Abstract

The following application note introduces USB and shows an example of how to configure the USB block on the H8SX/1664 to transfer data using bulk pipes only and use the microcontroller as a "classless" USB device. This document refers to the RSK H8SX/1664 USB kit and specifically to the included Bulk 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 of four USB transfer types for his 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 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 interface 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. An example of this is Get_Descriptor.

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 "***bInterval"*** or Polling Interval field in the endpoint descriptor. 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 “Bulk Only” class

USB does not specify a Bulk Only class. During enumeration, the host attempts to load the appropriate driver based on the device class, which is specified in the Device descriptor. However, if the descriptor does not specify a defined USB class, then no drivers will be loaded. Enumeration will proceed as expected, and the device will declare its resources including endpoints etc. Thus the host is aware of the device capabilities, but does not load a Windows USB class driver, since no class was specified; the user has to provide the driver in this case. In the included Bulk-Only application, the device descriptors have been modified to do just that. A driver is required to access the USB device, which is provided via the libusb-win32 library.

The “Bulk-Only” class is like a psuedo-UART, in the sense that it allows high speed transfer of data over USB without having to enumerate as a defined USB class device. The data to be transferred from the device is loaded into the appropriate driver and the transmit flag is set. When the host polls that endpoint the next time, the data is transmitted. Data reception occurs similarly on the device. The data transfer occurs over the bulk endpoints and is not limited by USB in terms of format or function. It is up to the user application to make sense of data format. Thus operation is very similar to a Mass Storage class device, except for the fact that in Mass Storage devices, the data transferred over the bulk pipes have to conform to a specific format (SCSI); whereas in the Bulk-Only class, raw data can be transferred, since the user writes the host application that makes sense of the data.

## 3. libusb-win32

libusb-win32 is an open source library that allows users to access USB devices in their application without having to write OS kernel code. The library has been ported to run on most operating systems including Microsoft Windows. The website also includes many useful applications, including one that allows creation of a custom .inf file for any device. The function calls are simplistic

The library has been used in the included Bulk-Only application to access the specific USB device.

Refer to the libusb-win32 homepage and libusb homepage for more information.

## 4. 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 Buffer Capacity | 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 1: 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 2: 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.

# 5. 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 USBINTN2 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**.

## 5.1 H8SX/1664 USB Operation

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

Using the included PC application it is possible to transmit and receive data to and from the H8SX/1664 via Bulk Transfers. Endpoint 1 is used on the H8SX/1664 for Bulk Out transfers. When data is successfully received in EPDR1 (USB endpoint data register 1) the EP1FULL flag generates USBINTN3. In response to the EP1FULL interrupt, the function **HandleEP1FULL()** is called. This function copies the received data from the EndPoint1 Bulk-Out data register to a local store, BULKDataOUT.Data[] and sets BULKDataOUT.ReceivedFlag. The main loop then decodes the host request and performs the requested action based on received data. The format of the received data is as shown in Table 3.

| Local Store for received data | Header Byte | Data bytes |
|---|---|---|
| | Byte[0] | Byte[1--RX_BUFFER_SIZE] |
| BULKDataOUT.Data[RX_BUFFER_SIZE] | ‘06’ Toggle LED | ‘01’ Toggle LED1 |
| | | ‘02’ Toggle LED2 |
| | ‘08’ LCD Message | LCD Message |
| | ‘09’ Read ADC | --- |

**Table 3: Received data format**

It is also possible for the H8SX to send data to the host PC. Bulk-In Transfer via End Point 2 transmits the data to the host PC. This can be demonstrated by clicking on the Read ADC button. The device will respond to this request by sending back the current ADC reading which can be modified by tweaking the potentiometer on the RSK.

In response to the Read ADC button being clicked, the host sends the command 0x09 to the device which is read into the array BulkDataOut.Data[]. The main loop then parses this array and sets the pointers BulkDataIn.pStart and BulkDataIn.pEnd to the start and end address of the buffer holding the ADC values and calls the function **TransmitData()** which sets a flag notifying the USB core that data is ready to be transmitted.

On the next request from the host for Bulk In data, the **HandleEP2EMPTY()** function writes a maximum of BULK_IN_PACKET_SIZE data from the buffer to the USB buffers UEDR2i (USB Data Register 2in). The USB peripheral now performs the necessary USB protocol handling to transmit the data.

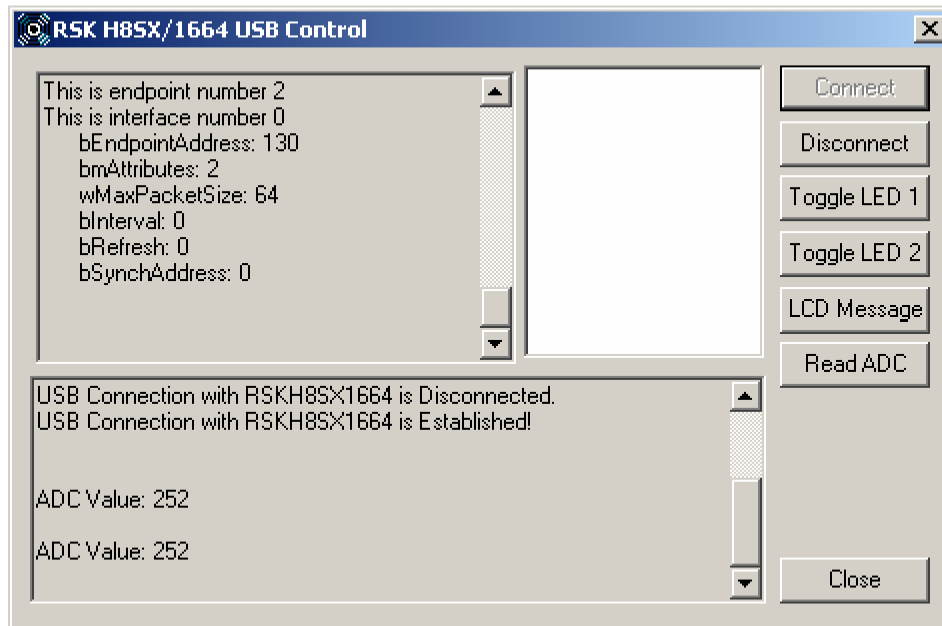


**Figure 1: USB_Connect.**

## 5.2 Windows application

The Windows application uses the libusb-win32 library in a VC++ project to interface with the USB device.

The library function calls used are **usb_bulk_read()**, and **usb_bulk_write(),** which take bulk endpoint numbers, buffer addresses and timeout values as arguments. The source code is made available only as an example of how to use these functions. The library also provides functions **usb_interrupt_read()** and **usb_interrupt_write()** which allow access to interrupt endpoints.

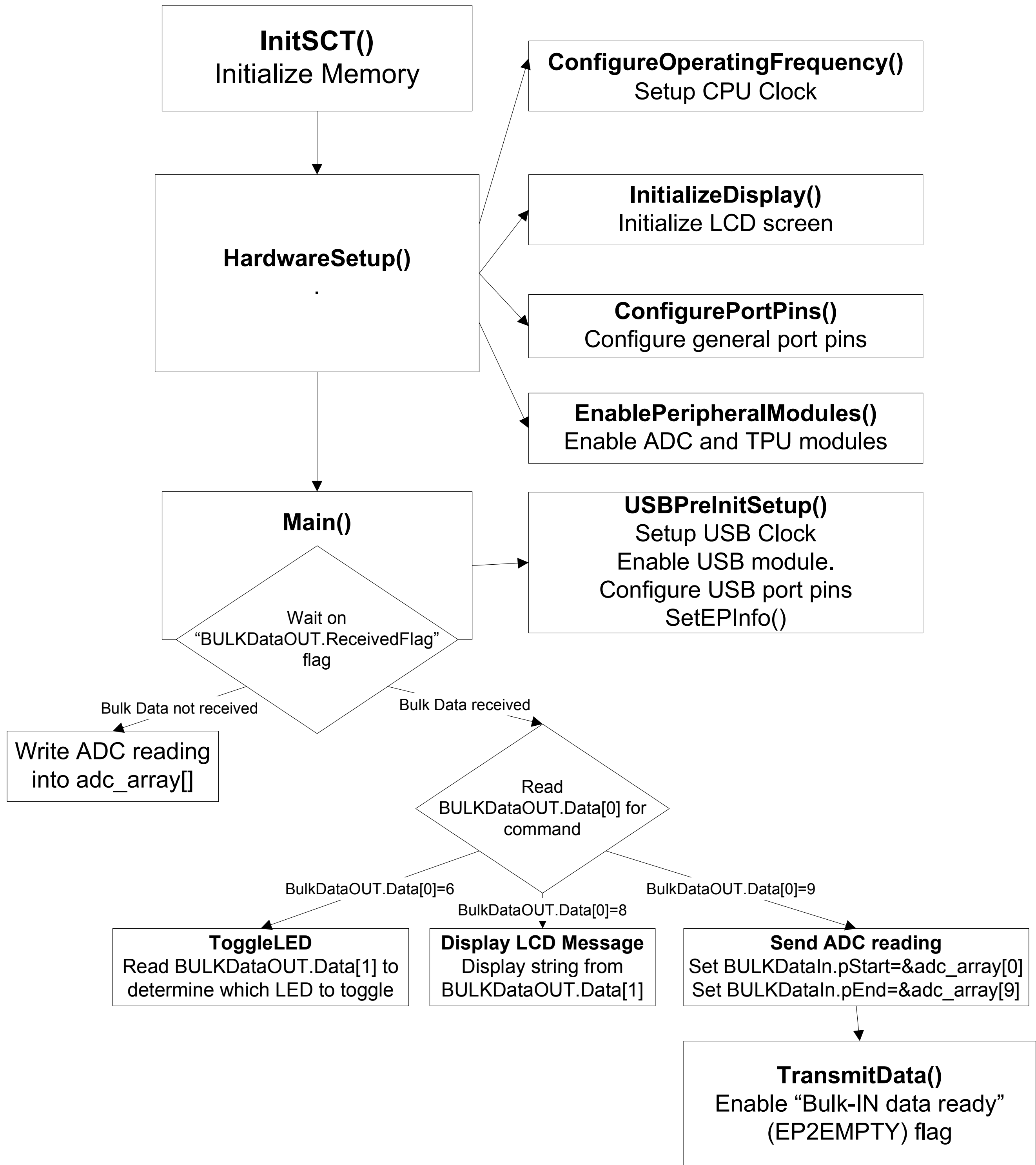


**Figure 2: System Initialization**

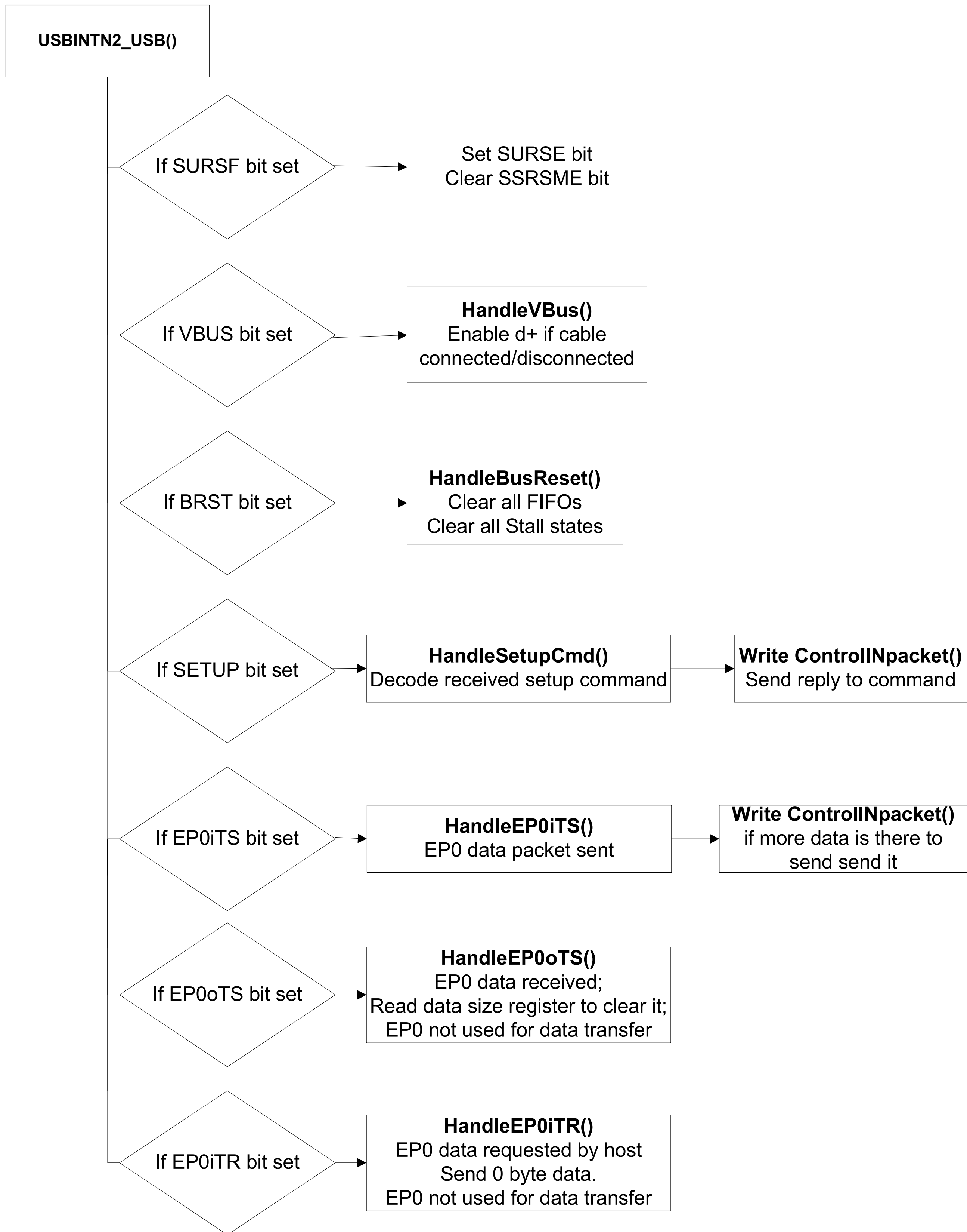


**Figure 3: USBINTN2_USB**

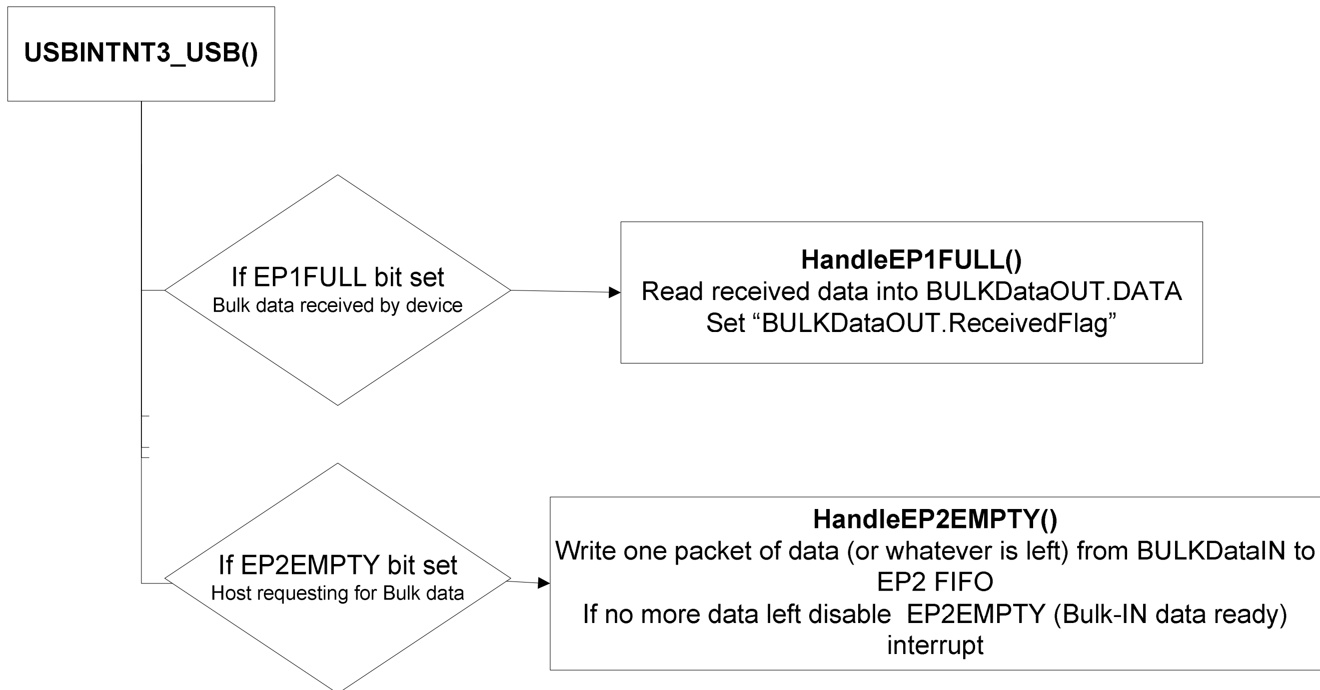


**Figure 4: USBINTN3_USB**

# 6. Using the "Bulk-Only" application in your project

After reading the above section on implementation, the sample application can be easily modified to transfer any amount of data between the device and host. Some parameters that may need to be modified are the size of the BULKDataOUT.Data[] which is typedef'd from RxDataBuff[] in usb.h. To use the system to transfer data, remove demo-specific code like switch ISR etc, configure BULKDataIN.pStart and BULKDataIN.pEnd pointers to point to the start and end addresses of any buffer that contains data to be transmitted and enable the EP2EMPTYE flag by calling the TransmitData() function. Data received from the host will be available in BULKDataOUT.Data[]

# 7. Limitations

The sample application supports only the GET_DESCRIPTOR standard request. Refer to the **DecodeStandardSetupPacket()** function for more details on this.

# 8. Data Sheet

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

# 9. References

1. H8SX/1664 group manual. Document number: REJ09B0294-0100
2. Universal Serial Bus Specification Revision 2.0
3. Device Class Definition for Communication Devices version1.1

4. The RSK H8SX/1664 User Manual
5. libusb-win32 homepage
6. libusb homepage
7. “USB Complete: Everything You Need to Develop Custom USB Peripherals” by Jan Axelson.
8. Jan Axelson’s USB Mass Storage page

## 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.