Using the H8S/2215R 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 H8S/2215R to transfer data using bulk pipes only and use the microcontroller as a “classless” USB device. This document refers to the RSK H8S/2215R 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 H8S/2215R USB Peripheral

The H8S/2215R is a high performance 16-bit embedded microcontroller built around the high speed, 32-bit H8S/2000 CPU core.

The H8S incorporates up to 256 kbytes of FLASH memory and 20 kbytes of RAM. The on-chip peripherals include:

- DMA controller (DMAC)
- Data Transfer Controller (DTC)
- 8 bit timer (TMR)
- 16-bit timer-pulse unit (TPU)
- Watchdog timer (WDT)
- Real-time clock (RTC)
- Serial communication interface (SCI)
- Boundary scan
- Universal serial bus (USB)
- 10-bit A/D converter
- High-performance user debugging interface (H-UDI)
- Clock pulse generator

The main features of the USB peripheral are:

- On-chip UDC (USB Device Controller) conforming to USB 1.1
- USB standard version 2.0 full-speed (12 Mbps) transfer supported
- Automatic processing of USB protocol
- 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
- Maximum of 9 endpoints specifiable. (Endpoint Configuration is selectable)
- Total 1288-byte FIFO

- EP0s fixed: Control_setup FIFO, 8 bytes
- EP0i fixed: Control_in FIFO, 64 bytes
- EP0o fixed: Control_out FIFO, 64 bytes
- EPn selectable: Interrupt_in FIFO, variable 0 to 64 bytes
- EPn selectable: Bulk_in FIFO, 64 bytes × 2 (double-buffer configuration)
- EPn selectable: Bulk_out FIFO, 64 bytes × 2 (double-buffer configuration)
- EPn selectable: Isochronous_in FIFO, variable 0 to 128 bytes × 2 (double-buffer configuration)
- EPn selectable: Isochronous_out FIFO, variable 0 to 128 bytes × 2 (double-buffer configuration)
- EPn selectable: Bulk_in FIFO, 64 bytes × 2 (double-buffer configuration)
- EPn selectable: Bulk_out FIFO, 64 bytes × 2 (double-buffer configuration)
- EPn selectable: Interrupt_in FIFO, variable 0 to 64 bytes

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 1: 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 H8S microcontrollers, the majority of peripherals are in Module Stop Mode when the H8S 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. However, some pre-initialization has to be performed prior to this.

The USB peripheral is mapped from address H'C00000 to H'C000FF, which is part of the external addressable area. Therefore, the Bus State Controller (BSC) registers have to be correctly configured prior to the USB peripheral being enabled, otherwise it will not be possible to communicate to the USB peripheral. It is also necessary to configure the Interrupt Controller prior to enabling the USB peripheral as the USB peripheral is interrupt driven.

The function **HardwareSetup()** which is called as part of the Power-on Reset exception handler assigns interrupt priority levels to the peripherals and configures the Interrupt Controller for Mode 2 operation. Please refer to section 5 of the H8S/2215R Hardware User Manual for a detailed description of the interrupt controller.

The function **USBPreInitSetup()** configures the BSC and takes the USB peripheral out of Module Stop Mode. The BSC must be configured for 8-Bit, 3-State Access with 0 wait states. Once the USB peripheral has been taken out of Module Stop Mode the USB clock2 will start. When this clock is stable the CK48READY flag is set in the UIFR3 (USB Interrupt FlagRegister 3).

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 UISRs (USB Interrupt Select Registers). If the corresponding bit is cleared to 0, the interrupt request will be handled by interrupt vector 104, EXIRQ0. If the corresponding bit is set to 1, the interrupt request will be handled by interrupt vector 105, EXIRQ1. By default, the UISRs have a value of 0. Therefore, the interrupt generated in response to the CK48READY flag will be handled by EXIRQ0. 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 UIFRs (USB Interrupt Flag Registers).

The function **HandleClockOK()** is called by the EXIRQ0 ISR in response to the USB clock stabilization and performs further USB peripheral initialization, such as enabling and disabling the other USB interrupts and directing them to EXIRQ0 or EXIRQ1. All Bulk transfer requests are configured to be processed on the EXIRQ1 vector.

At this point the application could place the USB peripheral back into Module Stop Mode by setting Bit-0 of MSTPCRB, hence saving power. However, this application does not do this and now 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 EXIRQ0. 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-6 of Port 3. 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’ and issues a software reset to the UDC. 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 EXIRQ0 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 UEDR0s (USB Endpoint Data Register 0s) register. UEDR0s stores the 8-Byte command sent to the host during set up. The function **ReadSetupPacket()** assigns the 8-Byte command to the union **SetupData**. With the data assigned to **SetupData** 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 H8S/2215R provides information to the host PC. This information is held in the files **usbdescriptors.c** & **usbdescriptors.h**.

5.1 H8S/2215R USB Operation

When the H8S/2215R 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 H8S/2215R. When data is sent to the device it is transmitted via Bulk Transfer. Endpoint 2o is used on the H8S/2215R for Bulk Out transfers. When data is successfully received in UEDR2o (USB endpoint data register 2out) the EP2oREADY flag generates EXIRQ1. In response to the EP2oREADY interrupt, the function **HandleEP2READY()** is called. This function copies the received data from the EndPoint2o 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 2.

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 2: Received data format

It is also possible for the H8S to send data to the host PC. Bulk-In Transfer via End Point 2i 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 array 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 software buffer to the USB buffers UEDR2i (USB Data Register 2in). The USB peripheral now performs the necessary USB protocol handling to transmit the data.

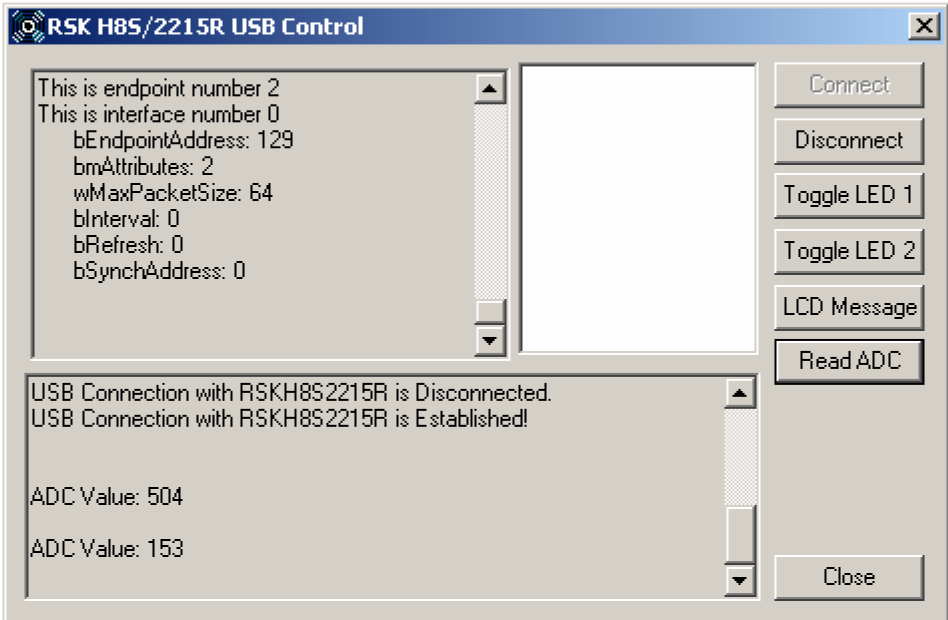

Figure 1: USB_Connect.

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.

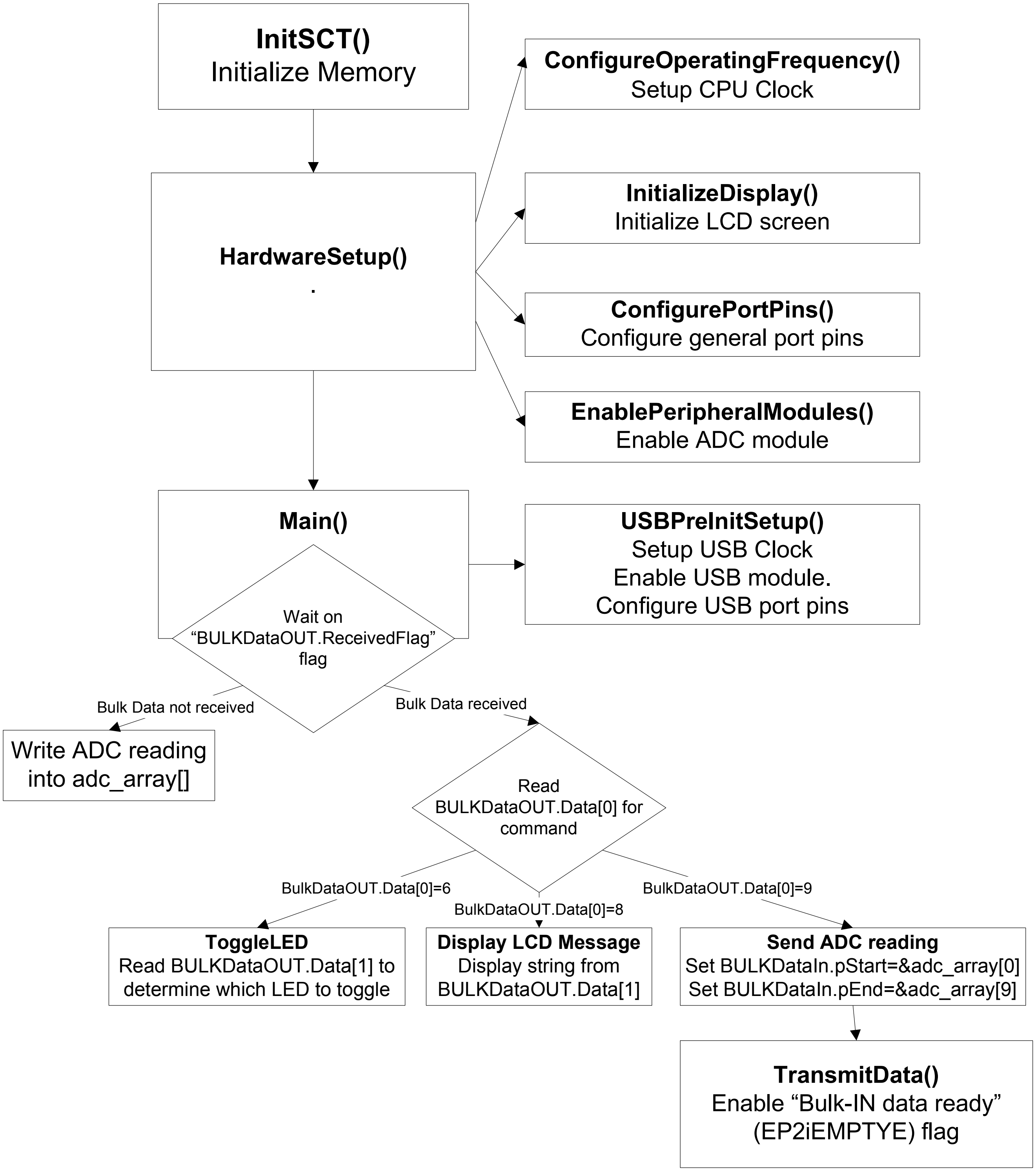

Figure 2: System Initialization

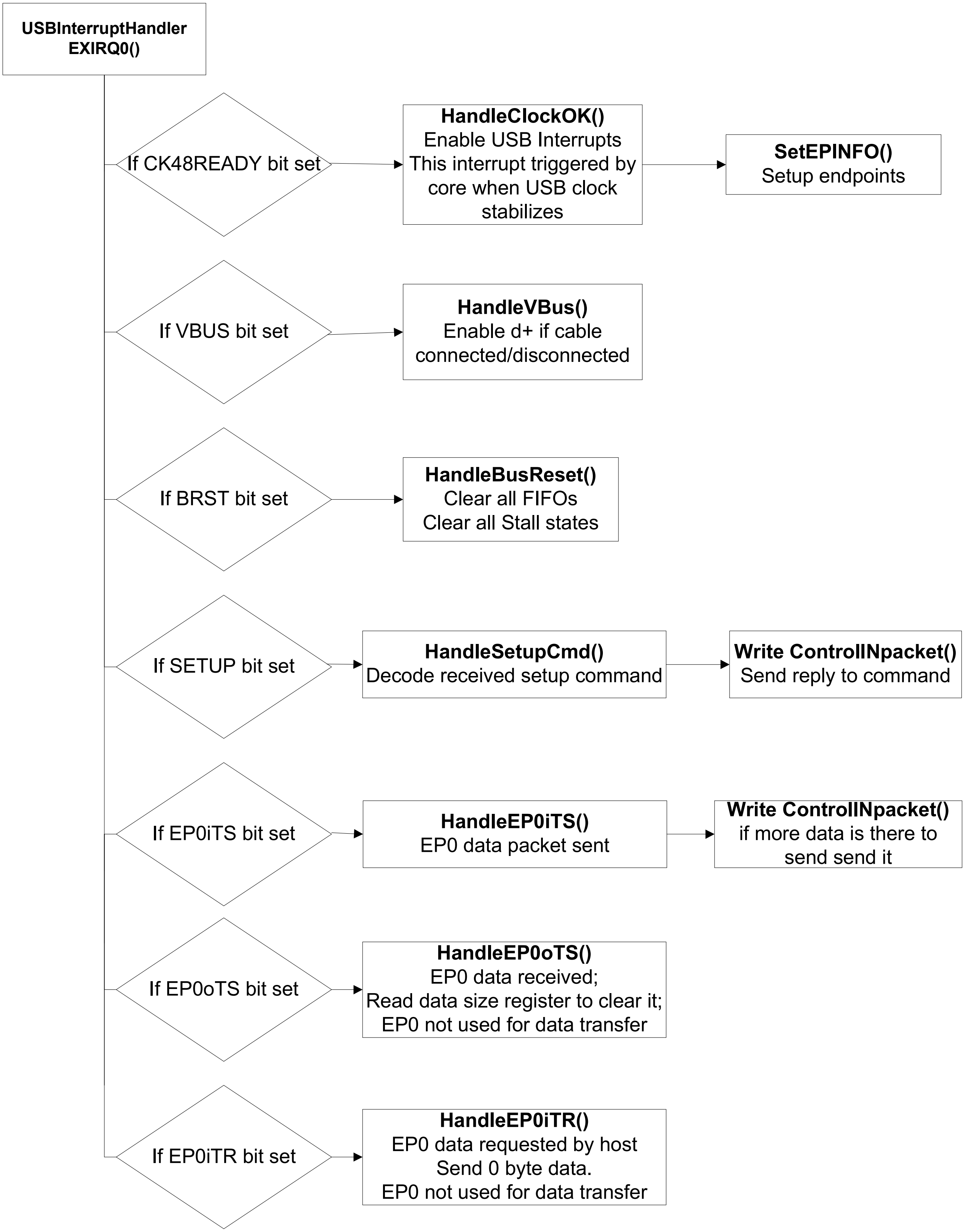

Figure 3: USBInterruptHandler EXIRQ0

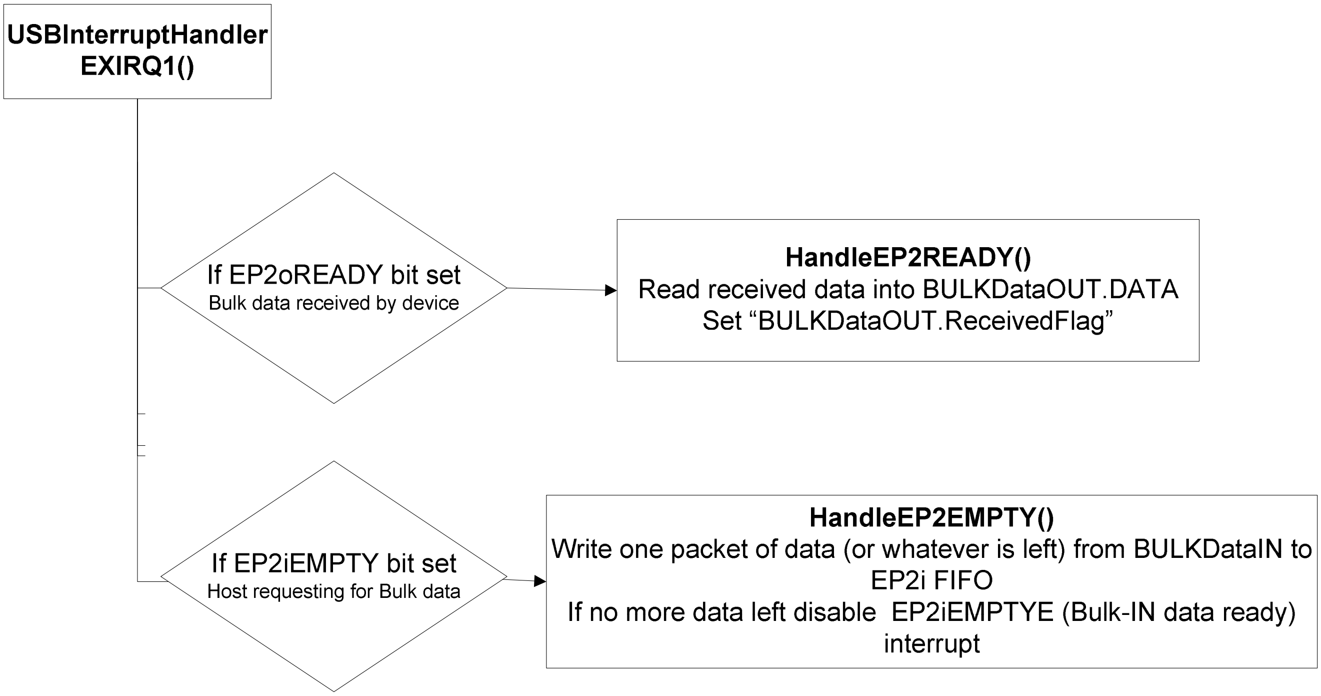

Figure 4: USBInterruptHandler EXIRQ1

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 EP2iEMPTYE 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. H8S/2215R group manual. Document number: REJ09B0140-0600
   (Use the latest version on the home page: http://www.renesas.com)

9. References

1. H8S/2215R group manual. Document number: REJ09B0140-0600
2. Universal Serial Bus Specification Revision 2.0
3. Device Class Definition for Communication Devices version1.1

4. The RSK H8S/2215R 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.