16

H8S/2218

USB Function Module Communication Class Device Application Note

Renesas 16-Bit Single-Chip Microcomputer
H8S Family / H8S/2200 Series

Application Note

REU05B0065-0100
Rev.1.00
2007.10.01

Renesas Technology
www.renesas.com

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 CDC class 4
2.2.1 Communications Device Class descriptors 4
2.2.2 Device Models 5
2.2.3 Abstract Control Model 5

3. The H8S/2218 USB Peripheral 5

4. Implementation 6

4.1 H8S/2218 USB Operation 6

5. Limitations 6

6. Data Sheet 6

7. References 6

Using the H8S/2218 as a CDC device

1. Abstract

The following application note introduces the USB CDC class and shows an example of how to configure the USB block on the H8S/2218 and use the microcontroller as a Communications Class Device. This document refers to the 3DK2218 USB kit and specifically to the included CDC 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 filed 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 non-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.

1. Refer to

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

2.2 The CDC class

USB communications device class is a composite Universal Serial Bus device class. It provides a single device class, but there may be more than one interfaces implemented such as a custom control interface, data interface, audio and mass storage related interfaces etc. As the name implies, it is used for communication class of devices including modems, telephony (POTS), and computer networking (Ethernet, ATM).

In order to handle the different responsibilities of a communications device such as device management, call management and data transmission, the different interfaces are specified. A Communications class interface for e.g. can handle device management and call management, and a data interface could handle data transmission. The reason for having different interfaces is that while defining an interface in the USB Interface Descriptor for a device, specific endpoints are associated with that interface. Thus while an interrupt endpoint could be used for call management (Communications Class Interface), bulk endpoints could be used for data transmission (Data Interface) since a higher bandwidth is required for the latter.

The management element of a Communications class interface is handled via the control endpoint (endpoint 0), and the optional notification element via the interrupt endpoint.

2.2.1 Communications device class descriptors

Devices in this class use the standard USB Device descriptors, Configuration descriptors, Interface descriptors and Endpoint descriptors.

Functional descriptors are used to describe the content of the class-specific information within an Interface descriptor. Functional descriptors include, header functional descriptor, Call Management functional descriptor functional descriptors for all device models, the union functional descriptor etc.

2.2.2 Device Models

Depending on the call management and data transmission, the device would fall into one of the defined CDC models. While some devices provide extensive call management including notification over the communications interface, others would chose to multiplex call management along with data transmission over the data interface and have a minimal implementation of the communications interface.

2.2.3 Abstract Control Model

With an Abstract Control Model, the USB device understands standard V.25ter (AT) commands. The device contains a Data-pump and micro-controller that handles the AT commands and relay controls. The device uses both a Data Class interface and a Communication Class interface.

A Communication Class interface of type Abstract Control Model will consist of a minimum of two pipes; one is used to implement the management element and the other to implement a notification element (interrupt pipe).The management element will handle both call management and device management commands. In addition, the device can use two pipes to implement channels over which to carry unspecified data, typically over a Data Class interface. The data interface usually uses either the Bulk or Isochronous endpoints and are expected to exist in pairs i.e. Iso-in and Iso-out or Bulk-in and Bulk-out.

Since legacy communication systems use only a single channel for both data and commands, the ACM is used to multiplex both streams over the Data Interface. Thus while the communications class interface would still be implemented, actual call control would occur over the data interface. To describe this particular characteristic, the *bmCapabilities* filed in the Call Management Functional descriptor has to be set accordingly.

In the included sample application, the filed is set to support Requests listed in Table 1.

Request	Code	Description
SET_LINE_CODING	20h	Configures DTE rate, stop-bits, parity, and number-of-character bits.
GET_LINE_CODING	21h	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
SET_CONTROL_LINE_STATE	22h	RS-232 signal used to tell the DTE device is now present.

Table 1: Supported AT requests in sample application

2. Refer to

Device Class Definition for Communication Devices version1.1 on usb.org for more detail.

3. The H8S/2218 USB Peripheral

The H8S/2218 is a high performance 16-bit embedded microcontroller built around the high speed, 32-bit H8S/2000 CPU core.

The H8S incorporates 128kBytes of FLASH memory and 12kBytes of RAM. The on-chip peripherals include:

- DMA controller (DMAC)
- 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
- Automatic processing of USB protocol
- Automatic processing of USB standard commands for endpoint 0. (Some commands need to be processed through the firmware)
- Full-speed (12 Mbps) transfer supported
- Three transfer modes supported (Control, Bulk, and Interrupt)
- 16 interrupt signals
- On-chip bus transceiver
- 4 Endpoints

Endpoint	Name	Transfer Type	Max Packet Size (bytes)	FIFO Buffer Capacity	DMA Transfer
0	EP0s	Setup	8	8 bytes	
	EP0i	Control-in	64	64 bytes	
	EP0o	Control-out	64	64 bytes	
1	EP1	Bulk-in	64	64 x 2 bytes	Available
2	EP2	Bulk-out	64	64 x 2 bytes	Available
3	EP3	Interrupt-in	64	64 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 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/2218 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.

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/2218 provides information to the host PC. This information is held in the files **usbdescriptors.c** & **usbdescriptors.h**.

4.1 H8S/2218 USB Operation

When the H8S/2218 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 HyperTerminal application it is possible to transmit and receive data to and from the H8S/2218. When data is sent to the device it is transmitted via Bulk Transfer. Endpoint 2 is used on the H8S/2218 for Bulk Out transfers. When data is successfully received in UEDR2 (USB endpoint data register 2) the EP2READY flag generates EXIRQ1. In response to the EP2READY interrupt, the function **HandleEP2Ready()** is called. This function copies the received data form the EndPoint2 Bulk-In data register to a local store, BULKDataOUT.Data[]. The received data is used to set the state of the LEDs on the 3DK2218.

It is also possible for the H8S to send data to the host PC. Bulk Transfer via End Point 1 transmits the data to the host PC. This can be demonstrated by pressing any of the three switches. The three switches are connected to three of the IRQ lines. In response to a switch being pressed, a constant text string is transmitted to the host PC.

Switch	External Interrupt	Text String Transmitted
SW1	IRQ2	"Switch 1 has been pressed. This Data Packet is 58 bytes.\r\n"
SW2	IRQ4	"Switch 2 has been pressed. This Data Packet is larger than 'Switch 1', at 85 bytes!\r\n"

SW3	IRQ7	"Switch 3 has been pressed. This is an even bigger Data Packer than 'Switch 2'. This is to demonstrate that the process of transmitting larger data packets is a 'transparent' procedure.\r\n"

Table 4: Switch triggered events

The IRQx ISR initiates two data pointers, one to the start of the data string and the other to the end of the data string. The IRQx ISR then calls a function **TransmitData()**; This function enables the EP1EMPTYE (End Point 1 FIFO Empty) interrupt. The EP1EMPTY flag indicates that the EP1 FIFO is empty and can accept data.

In response to the EP1EMPTYE flag being set the function **WriteBULKINPacket()** is called. This function copies data into the EP1 FIFO and then sets the EP1PKTE bit. This generates a trigger to enable the transmission to EP1 FIFO. The USB peripheral now performs the necessary USB protocol handling to transmit the data.

On receiving data from the host, the Handle EP2READY function sets s software flag, which allows the main loop to configure the BULKDataIN structure pointers so that the received data is sent back to the host and thus shows up as "echoes" on the hyperterminal.

5. Limitations

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

6. Data Sheet

1. H8S/2218 group manual. Document number: REJ09B0074-0400O

   (Use the latest version on the home page: http://www.renesas.com)

7. References

3. H8S/2218 group manual. Document number: REJ09B0074-0400O
4. Universal Serial Bus Specification Revision 2.0
5. Device Class Definition for Communication Devices version1.1
6. The 3DK2218 User Manual
7. "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.