RENESAS

16

H8S/2218

USB Function Module
Mass Storage Class (Bulk-Only Transport)
Application Note

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

Application Note

REU05B0064-0100
Rev.1.00
2007.10.01

Renesas Technology
www.renesas.com

Using the USB block as a Mass Storage Class device 2

1. Abstract 2

2. Introduction to USB 2

2.1 Transfer types: 3

2.1.1 Control transfers 3

2.1.2 Interrupt Transfers 3

2.1.3 Bulk transfers 4

2.1.4 Isochronous transfers 4

2.2 Storage media and the FAT File System: 4

3. The Mass Storage Class 5

3.1 Mass Storage Class Requirements: 5

3.2 The SCSI command Interface 5

3.3 The H8S/2218 USB Peripheral 6

4. Implementation 7

4.1 State Transition Diagram 7

4.2 USB Communication State 8

4.2.1 Control Transfer 9

4.2.2 Bulk Transfer 9

4.3 File Structure 9

4.4 Purposes of Functions 11

4.5 RAM Disk 14

4.6 Processing If an Error Occurs 15

5. Limitations 21

6. Data Sheet 21

7. References 21

Using the USB block as a Mass Storage Class device

1. Abstract

The following application note introduces the USB Mass Storage class and shows an example of how to configure the USB block on the H8S/2218 and use the microcontroller as a Mass Storage class device. This document refers to the 3DK2218 USB kit and specifically to the included Mass Storage Class 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. 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, so each pipe is 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 for the Mass Storage class 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 interface supported and power consumption are provided in this descriptor; most devices usually support only a single configuration, but multiple configurations are allowed.
c) Interface descriptor. (Specifying Mass Storage class): Each interface on the device has its own descriptor and subordinate descriptors (descriptors for endpoints used in the interface); the mass storage functionality is specified in this descriptor.
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) 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.

The Get_Descriptor command is used to retrieve standard, class and vendor specific requests depending on the setting in the descriptor type field which is the high byte of the request. 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.

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 bursts of 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. This type of transfer is supported only by Full and high speed devices.

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. The transfer mode is supported only by Full and High speed USB devices.

2.2 Storage media and the FAT File System:

There are many different kinds of storage media and they can be broadly grouped into hard drives and general flash memory from an embedded storage media point of view. While hard drives are preferred for larger capacities, flash memory is the media of choice in smaller capacity applications. While using flash memory, device firmware must implement "wear-leveling" in order to extend the life of the memory chip.

FAT file system:

The FAT (File Allocation Table) file system was created (and partially patented) by Microsoft for disk management purposes. The system essentially keeps a table of file names, location of contents, usable areas of memory etc. This file system is supported by virtually every OS, and hence is an ideal format in which to store data. The FAT file system is used extensively in embedded data storage systems that have to directly interface with an OS.

Implementing a file-system is entirely dependant on the application. In some cases where data is to be stored on the mass storage device and simply moved around by the OS, the firmware does not have to be aware of or understand the host file-system, it only has to respond to requests to transfer blocks of data.

In the provided application, a file diskimage.h includes the FAT file system; on startup, this image is copied into RAM, and thus the device enumerates as a preformatted FAT device.

For more information and articles on Embedded File Systems and Storage media refer to Jan Axelson's USB Mass Storage page

3. The Mass Storage Class

USB devices are categorized in to various classes based on common behavior and protocols for devices that serve similar functions.

3.1 Mass Storage Class Requirements:

In addition to supporting standard USB requirements, a mass storage class device must conform to the following as well:

1. Interface Descriptor with a class code of 08h.
2. A mass storage interface with Bulk-in and Bulk-out end points and endpoint zero for control transfer.
3. Storage media (hard drive, flash-memory cards, CD/DVD, MMC cards etc).
4. Firmware support for logical block addressing (LBA) of the storage media.
5. Firmware support for Mass Storage class requests.
6. Support for one or more industry-standard command-block sets to exchange control, data and status information. (e.g. SCSI)

File system support is not a requirement for Mass Storage class devices but, dependant on the application, the developer may choose to implement this as well. If the firmware supports Logical Block Addressing, then any required data can be readily accessed; however, if there was need for this storage media to be useful in other environments outside that controlled by the device firmware (e.g. if the storage media were to be used in a Windows environment), then additional support/interface as required by that environment will have to be provided. In most cases a file system provides the requisite interface.

3.2 The SCSI command Interface

The SCSI (Small Computer Standard Interface) standard contains definitions of command sets of specific peripheral device types and using it to transfer data between computers and the specific peripheral device. However the presence of “unknown” as one of the device types, theoretically allows SCSI to be used to interface with practically any device.

Devices are classified based on the type of SCSI commands they support; the SCSI Block Commands (SBC) document specifies commands used by flash drives and other direct access block devices. In SCSI terminology, communication takes place between an initiator and a target. The initiator sends a command to the target which then responds. SCSI commands are sent in a Command Descriptor Block (CDB). The CDB consists of a one byte operation code followed by five or more bytes containing command-specific parameters. At the end of the command sequence the target returns a Status Code byte which is usually 00h for success, 02h for an error (called a Check Condition), or 08h for busy. When the target returns a Check Condition in response to a command, the initiator usually then issues a SCSI Request Sense command in order to obtain the status.

The total number of commands defined by the protocol are about 60; but how many are implemented is dependant on the application. At a minimum, dependant on the application, the following primary commands have to be implemented:

1. INQUIRY: This command is used by the host to request information about the device. The response from the device is in the form of a structure (at least 36 bytes in length) where information such as

peripheral device type, vendor identification number, and other information about the devices capabilities is sent to the host. The response structure is sent in the data-transfer phase of the request.

2. READ CAPACITY: Tells the host the media sector information.
3. READ (10): Reads the specified sector volume data from a specified sector.
4. REQUEST SENSE: The host requests sense data via this command. In the event that the device experiences a problem, status information is filled into a structure; this data is called sense data.
5. TEST UNIT READY: This command is used to determine if the storage device is ready for use.
6. WRITE (10). Writes the specified sector volume data to a specified sector.

The following additional commands have been implemented by the sample program as well.

1. PREVENT/ALLOW MEDIUM REMOVAL: This command requests the device to prevent or allows removal of the storage media from the device. A 2 bit field is used to set/unset the option; support is optional for this command.
2. VERIFY: Verifies if the data in a medium can be accessed.
3. STOP/START UNIT: Controls installation and removal of media.
4. MODE SENSE (6): Tells the host the drive status.

3.3 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 3.1: 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 3.2: 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.

4. Implementation

In this section, features of the sample program and its structure are explained. This sample program runs on the H8S/2218, which works as a RAM disk, and initiates USB transfers by means of interrupts from the USB function module. Of the interrupts from modules in the H8S/2218, there are three interrupts related to the USB function module: EXIRQ0, EXIRQ1, and IRQ6, but in this sample program, only EXIRQ0 is used.

Features of this program are as follows.

- Control transfer can be performed.
- Bulk-out transfer can be used to receive data from the host controller.
- Bulk-in transfer can be used to send data to the host controller.
- It operates as a RAM disk that supports SCSI commands.

4.1 State Transition Diagram

Figure 4.1 shows a state transition diagram for this sample program. In this sample program, as shown in figure 4.1, there are transitions between four states.

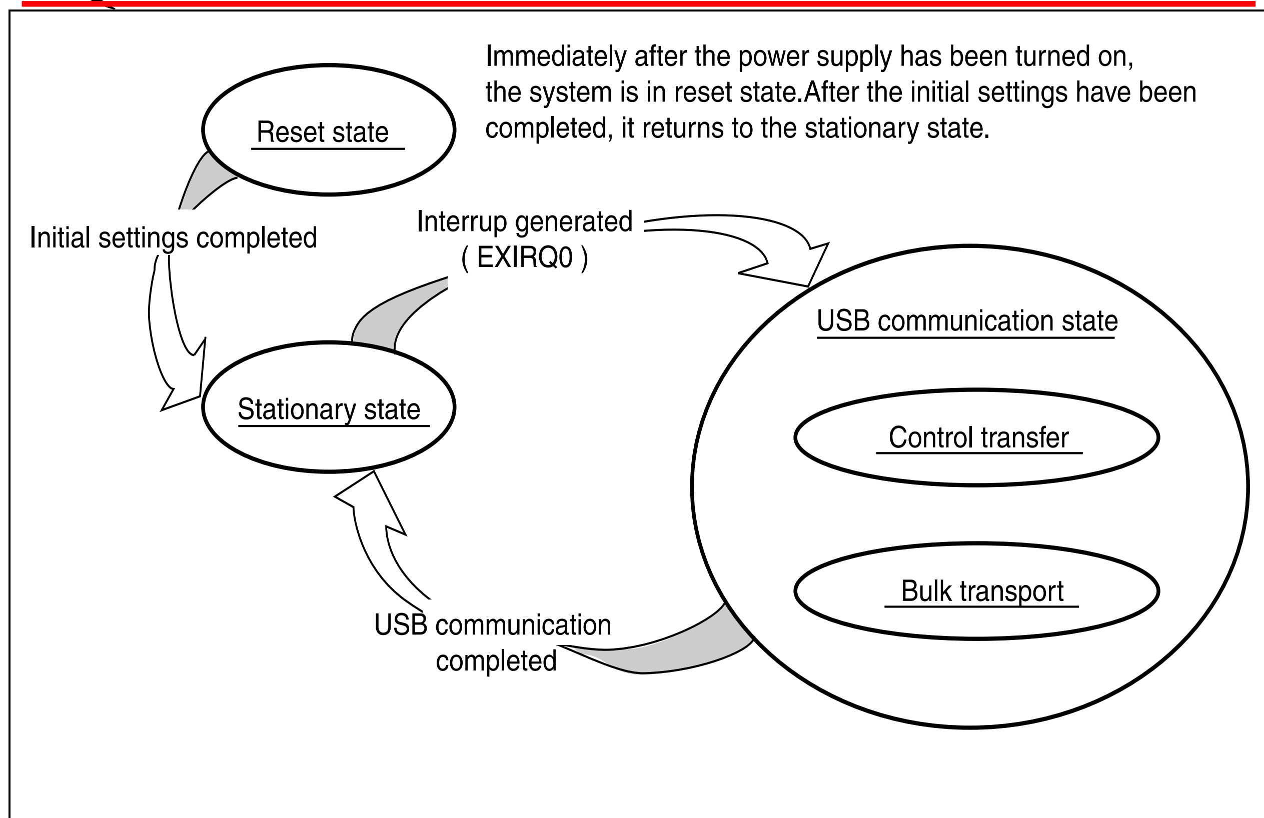

Figure 4.1: State Transition Diagram

- Reset State

  Upon power-on reset and manual reset, this state is entered. In the reset state, the H8S/2218 mainly performs initial settings.
- Stationary State

  When initial settings are completed, a stationary state is entered in the main loop.
- USB Communication State

  In the stationary state, when an interrupt from the USB module occurs, this state is entered. In the USB communication state, data transfer is performed by a transfer method according to the type of interrupt. The interrupts used in this sample program are indicated by interrupt flag registers 0 to 3 (UIFR0 to UIFR3), and there are nine interrupt types in all. When an interrupt factor occurs, the corresponding bits in UIFR0 to UIFR3 are set to 1.

4.2 USB Communication State

The USB communication state can be further divided into two states according to the transfer type (see figure 4.2). When an interrupt occurs, first there is a transition to the USB communication state, and then there is further branching to a transfer state according to the interrupt type. The branching method is explained in section 5, Sample Program Operation.

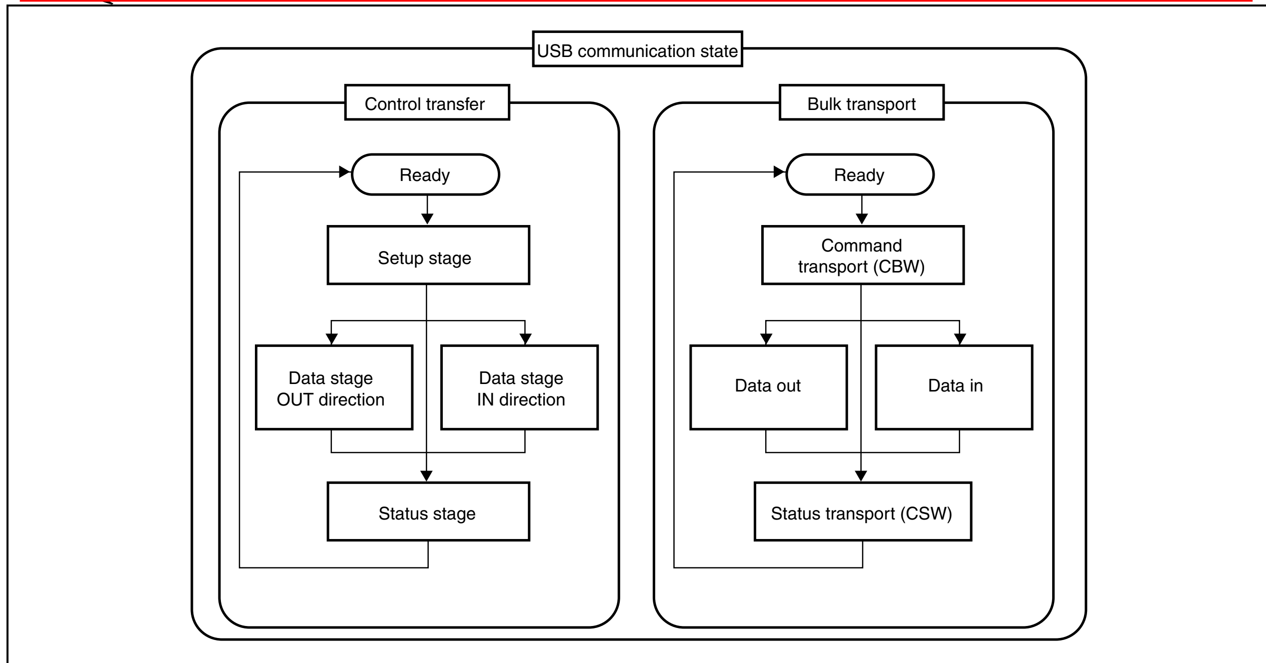

Figure 4.2: USB Communication State

4.2.1 Control Transfer

Control transfer is used mainly for functions such as obtaining device information and specifying device operating states. For this reason, when the function is connected to the host PC, control transfer is the first transfer to be carried out.

Transfer processing for control transfer is carried out in a series of two or three stages. These stages are a setup stage, a data stage, and a status stage.

4.2.2 Bulk Transfer

Bulk transfer has no time limitations, so it is used to send large volumes of data with no errors. The data transfer speed is not guaranteed, but the data contents are guaranteed. With USB Mass Storage Class (Bulk-Only Transport), bulk transfer is used to transfer storage data between the host PC and the function.

Transport processing for USB Mass Storage Class (Bulk-Only Transport) is carried out in a series of two or three stages. These stages are command transport (CBW), data transport, and status transport (CSW).

4.3 File Structure

This sample program consists of eight source files and eleven header files. The overall file structure is shown in table 4.1. Each function is arranged in one file by transfer method or function type. Figure 4.3 shows the layered configuration of these files.

File Name	Principle Role
StartUp.c	Microcomputer default settings
UsbMain.c	Judging the causes of interrupts Sending and receiving packets
DoRequest.c	Processing setup commands issued by the host
DoRequestBOT_StorageClass.c	Processing Mass Storage Class (Bulk-Only Transport) class commands
DoControl.c	Executing control transfer
DoBulk.c	Executing bulk transfer
DoBOTMSClass.c	Executing Mass Storage Class (Bulk-Only Transport)
DoSCSICommand.c	Analyzing and processing SCSI commands
h8s2218.h	Defining H8S/2218 registers
SysMemMap.h	Defining MS2218CP memory map addresses
CatProType.h	Prototype declarations
CatTypedef.h	Defining the basic structures used in USB firmware
CatBOTTypedef.h	Defining structures used for Bulk-Only Transport
CatSCSITypedef.h	Defining structures used for SCSI and macros for preparing FAT information
SetUsbInfo.h	Default settings of variables needed to support USB
SetBOTInfo.h	Default settings of variables needed to support Bulk-Only Transport
SetSCSIInfo.h	Default settings of variables needed to support SCSI commands
SetSystemSwitch.h	System operation settings
SetMacro.h	Defining macros
sct.src	Specifying variables to be used to copy initial values from RAM

Table 4.1: File Structure

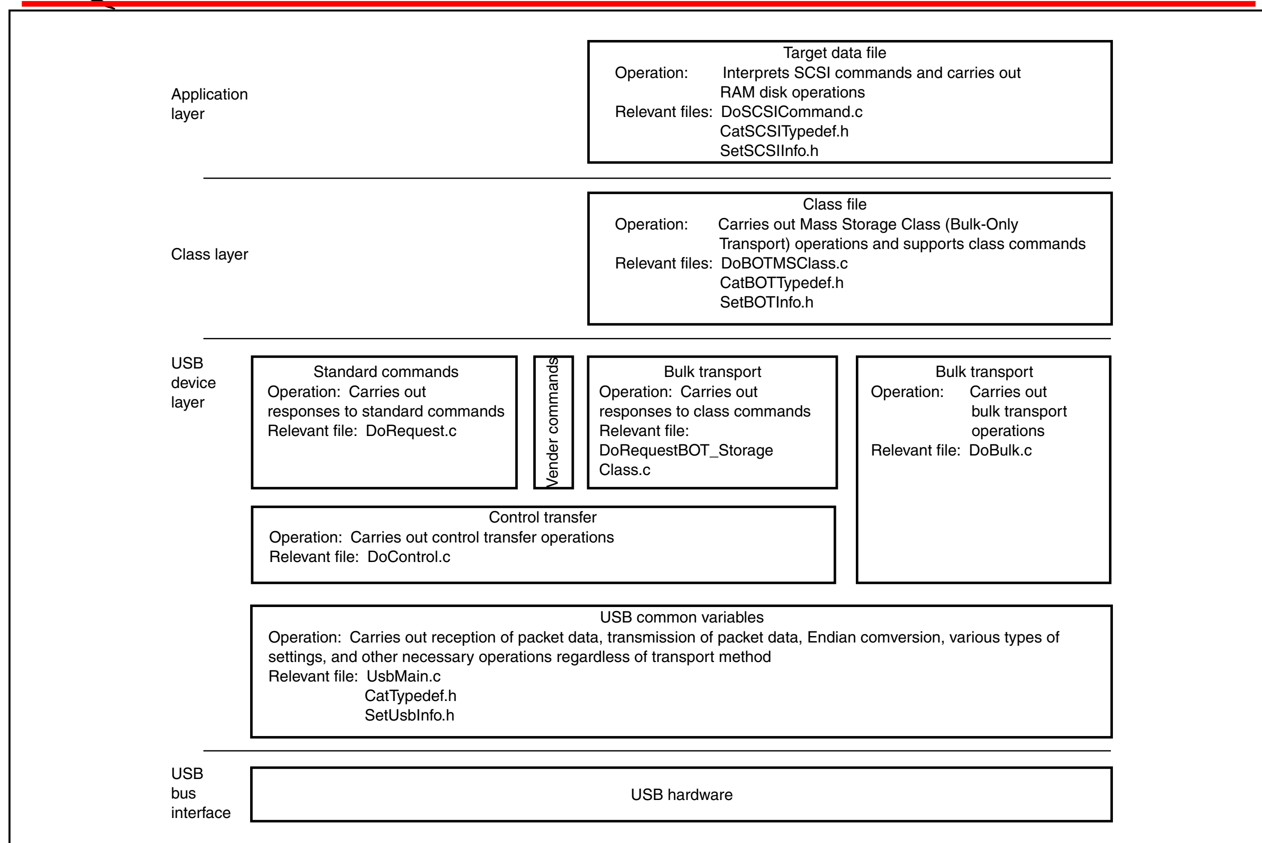

Figure 4.3: Layered Configuration of the Storage Class (BOT) firmware

4.4 Purposes of Functions

Table 4.2 to 4.9 shows functions contained in each file and their purposes.

File in Which Stored	Function Name	Purpose
StartUp.c	SetPowerOnSection	Sets BSC, terminals, and interrupt controller, calls initialization routines, and shifts to the main loop
	_INITSCT	Copies variables that have default settings to the RAM work area
	InitMemory	Clears the RAM area used in bulk communication
	InitSystem	Specifies the USB clock, system interrupts, and masks

Table 4.2: StartUp.c

When a power-on reset or manual reset is carried out, the SetPowerOnSection of the StartUp.c file is called. At this point, the H8S/2218 default settings are entered and the RAM area used for bulk transfer is cleared.

File in Which Stored	Function Name	Purpose
UsbMain.c	BranchOfInt	Discriminates interrupt factors, and calls function according to interrupt
	GetPacket	Writes data transferred from the host controller to RAM
	GetPacket4	Writes data transferred from the host controller to RAM in longwords (ring buffer supported; not used by the Mass Storage Class)
	GetPacket4S	Writes data transferred from the host controller to RAM in longwords (ring buffer not supported; fast-speed version)
	PutPacket	Writes data for transfer to the host controller to the USB module
	PutPacket4	Writes data for transfer to the host controller to the USB module in longwords (ring buffer supported; not used by the Mass Storage Class)

File in Which Stored	Function Name	Purpose
UsbMain.c	PutPacket4S	Writes data for transfer to the host controller to the USB module in longwords (ring buffer not supported; fast-speed version)
	SetControlOutContents	Overwrites data with that sent from the host
	SetUsbModule	Sets USB module initial settings
	ActBusReset	Clears FIFO on receiving bus reset
	ActBusVcc	Pulls up D+ and controls USB module when the USB cable is connected or disconnected (not used by this sample application)
	ConvRealn	Reads data of a specified byte length from a specified address
	ConvReflexn	Reads data of a specified byte length from specified addresses, in reverse order

Table 4.3: UsbMain.c

In UsbMain.c, interrupt factors are discriminated by the USB interrupt flag register, and functions are called according to the interrupt type. Also, packets are sent and received between the host controller and function modules.

File in Which Stored	Function Name	Purpose
DoRequest.c	DecStandardCommands	Decodes command issued by host controller, and processes standard commands
	DecVenderCommands	Processes vendor commands

Table 4.4: DoRequest.c

During control transfer, commands sent from the host controller are decoded and processed. In this sample program, a vendor ID of 045B (vendor: Renesas Technology Corp.) is used. When the customer develops a product, the customer should obtain a vendor ID at the USB Implementers' Forum. Because vendor commands are not used, DecVenderCommands does not perform any action. In order to use a vendor command, the customer should develop a program.

File in Which Stored	Function Name	Purpose
DoRequestBOT_ StorageClass.c	DecBOTClass Commands	Processes USB Mass Storage Class (Bulk-Only Transport) commands

Table 4.5: DoRequestBOT_StorageClass.c

This function carries out processing according to the Mass Storage Class (Bulk-Only Transport) commands (Bulk-Only Mass Storage Reset and Get Max LUN).

The Bulk-Only Mass Storage Reset command resets all of the interfaces used in Bulk-Only Transport.

The Get Max LUN command returns the largest logical unit number used by peripheral devices. In this sample program, there is one logical unit, so a value of 0 is returned to the host.

File in Which Stored	Function Name	Purpose
DoControl.c	ActControl	Controls the setup stage of control transfer
	ActControlIn	Controls the data stage and status stage of control IN transfer (transfer in which the data stage is in the IN direction)
	ActControlOut	Controls the data stage and status stage of control OUT transfer (transfer in which the data stage is in the OUT direction)
	ActControlInOut	Sorts the data stage and status stage of control transfers and direct them to ActControlIn and ActControlOut.

Table 4.6: DoControl.c

When control transfer interrupt SETUP TS is generated, ActControl obtains the command, and decoding is carried out by DecStandardCommands to determine the transfer direction. Next, when control transfer interrupt EP0o TS, EP0i TR, or EP0i TS is generated, ActControlInOut calls either ActControlIn or ActControlOut depending on the transfer direction, and the data stage and status stage are carried out by the called function.

File in Which Stored	Function Name	Purpose
DoBulk.c	ActBulkOut	Performs bulk-out transfer
	ActBulkIn	Performs bulk-in transfer
	ActBulkInReady	Performs preparations for bulk-in transfer

Table 4.7: DoBulk.c

These functions carry out processing involving bulk transfer.

File in Which Stored	Function Name	Purpose
DoBOTMS Class.c	ActBulkOnly	Divides Bulk-Only Transport into separate stages
	ActBulkOnlyCommand	Controls CBW for Bulk-Only Transport
	ActBulkOnlyIn	Controls data transport and status transport (when the data stage is in the IN direction)
	ActBulkOnlyOut	Controls data transport and status transport (when the data stage is in the OUT direction)

Table 4.8: DoBOTMSClass.c

With DoBOTMSClass.c, control of the two or three stages of the Mass Storage Class (Bulk-Only Transport) is carried out, and operation is carried out in accordance with the specifications.

File in Which Stored	Function Name	Purpose
DoSCSI Command.c	DecBotCmd	Processes SCSI commands sent from the host using Bulk-Only Transport
	SetBotCmdErr	Processes SCSI command errors

Table 4.9: DoSCSICommand.c

The DoSCSICommand.c function is used to analyze SCSI commands sent from the host PC and prepare for the next data transport or status transport.

Figure 4.4 shows the interrelationship between the functions explained in table 4.2 to 4.9. The upper-side functions can call the lower-side functions. Also, multiple functions can call the same function. In the stationary

state, SetPowerOnSection calls other functions, and in the case of a transition to the USB communication state which occurs on an interrupt, BranchOfInt calls other functions. Figure 4.4 shows the hierarchical relation of functions; there is no order for function calling. For information on the order in which functions are called, please refer to the flow charts of section 5, Sample Program Operation.

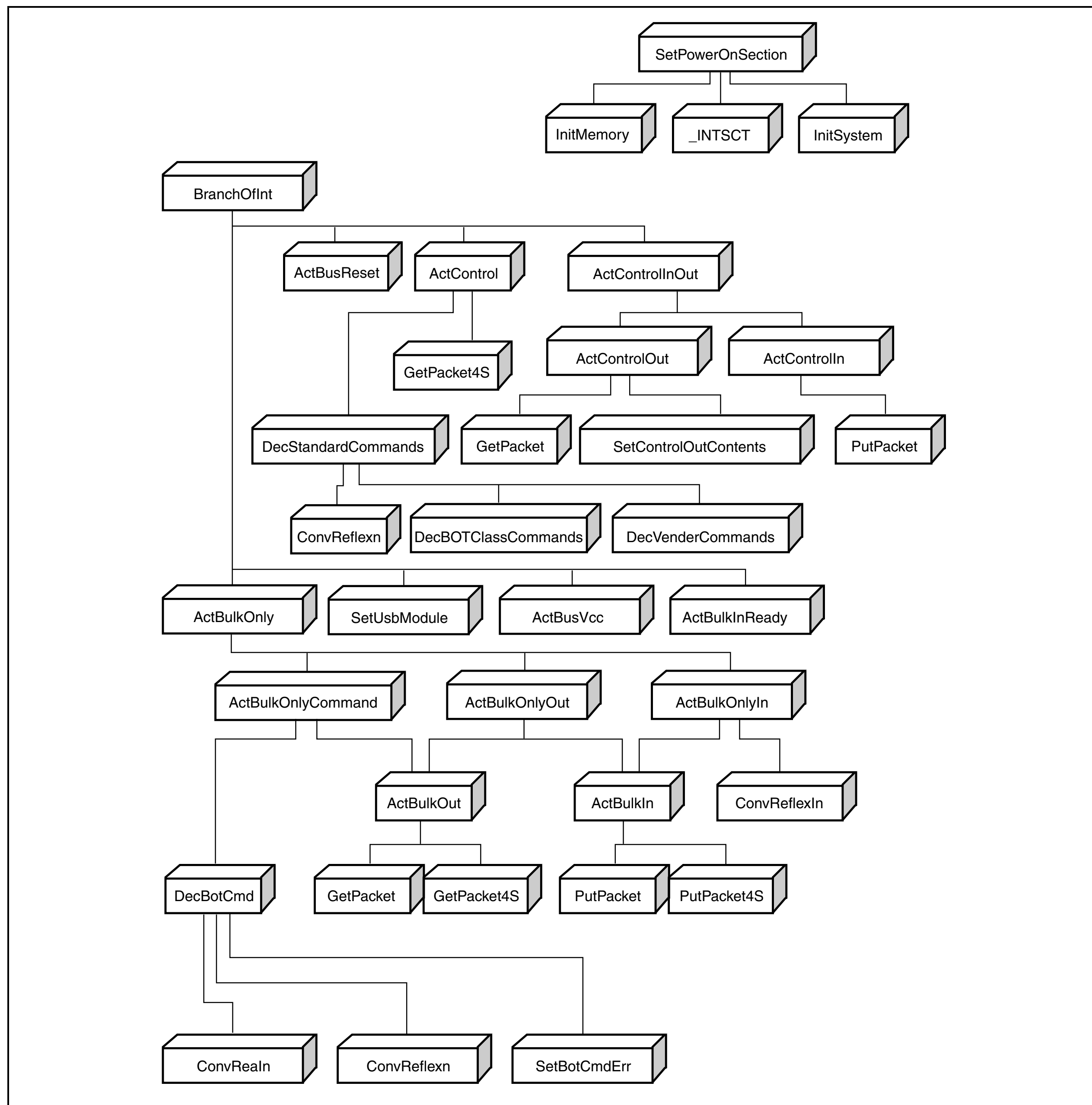

Figure 4.4: Interrelationships between Functions

4.5 RAM Disk

In the sample program provided here, the Internal RAM in the 2218 is selected as the disk device, and the host PC is notified that the 2218 (function) is a disk.

As shown in figure 4.5, the disk device of the function has a master boot block and a partition boot block. When the system is booted, an initialization routine is used to write the master boot block and the partition boot block to the RAM disk area on the Internal RAM.

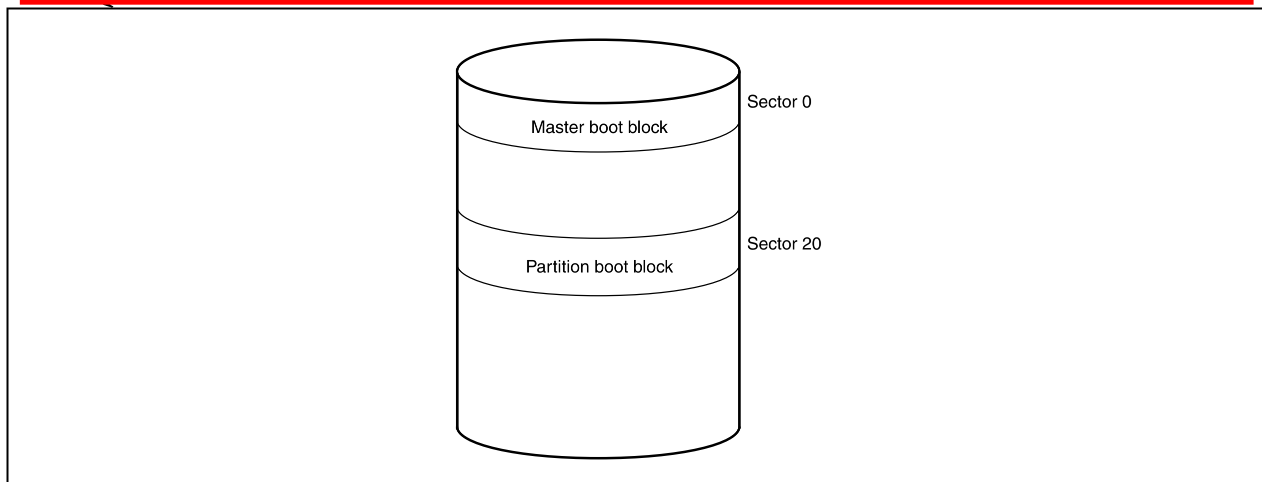

Figure 4.5: Disk Construction

SCSI commands are used to allow function access from the host PC (saving and loading data). In order to work with SCSI commands, the user needs to understand the construction shown in figure 4.5 and then write the operation.

4.6 Processing If an Error Occurs

The errors that may occur during a Mass Storage Class (Bulk-Only Transport) transmission between the host PC and function, and how the function operates when an error occurs are described below.

The Bulk-Only Transport standard defines the following two types of errors:

- Invalid CBW
- Operation expected by the host PC and operation planned by the function (operation specified by the SCSI command) do not match (10 cases)

The Bulk-Only Transport standard does not cover any other states.

There are 13 states for data transfer between the host PC and a function as shown in Tables 4.10 and 4.11. Cases 1, 6 and 12 are normal states.

		What the Host PC Expects		
		No Data Transfer	**Data Reception from Function**	**Data Send to Function**
What the function plans	No data transfer	(1) Hn = Dn	(4) Hi > Dn	(9) Ho > Dn
	Data send to host PC	(2) Hn < Di	(5) Hi > Di (6) Hi = Di (7) Hi < Di	(10) Ho < > Di
	Data reception from host PC	(3) Hn < Do	(8) Hi < > Do	(11) Ho > Do (12) Ho = Do (13) Ho < Do

Table 4.10: Data Transfer States between Host PC and Function

Case No.	Relation between Host PC and Function
1	The host PC expects no data transfer and the function plans no data transfer.
2	The host PC expects no data transfer but the function plans to send data to the host PC
3	The host PC expects no data transfer but the function plans to receive data from the host PC.
4	The host PC expects to receive data from the function but the function plans no data transfer to the host PC.
5	The amount of data the function sends to the host PC is less than the amount of data the host PC expected to receive from the function.
6	The amount of data the function sends to the host PC is equal to the amount of data the host PC expected to receive from the function.
7	The amount of data the function sends to the host PC is greater than the amount of data the host PC expected to receive from the function.
8	The host PC expects to receive data from the function but the function plans to receive data from the host PC.
9	The host PC expects to send data to the function but the function plans no data transfer to the host PC.
10	The host PC expects to send data to the function but the function plans to send data to the host PC.
11	The amount of data the function receives from the host PC is less than the amount of data the host PC expected to send to the function.
12	The amount of data the function receives from the host PC is equal to the amount of data the host PC expected to the function.
13	The amount of data the function receives from the host PC is greater than the amount of data the host PC expected to send to the function.

Table 4.11: Explanation of Data Transfer States between Host PC and Function

Table 4.12 shows sample error conditions that may be generated.

Case No.	Relation between Host PC and Function
2	When a READ command is issued from the host PC, the amount of data to be transported in the USB data transport is 0 while the amount of data specified by the SCSI command is a value other than 0.
3	When a WRITE command is issued from the host PC, the amount of data to be transported in the USB data transport is 0 while the amount of data specified by the SCSI command is a value other than 0.
4	When a READ command is issued from the host PC, the amount of data to be transported in the USB data transport is 0 while the amount of data specified by the SCSI command is 0.
5	When a READ command is issued from the host PC, the amount of data specified by the SCSI command is less than the amount of data to be transported in the USB data transport.
7	When a READ command is issued from the host PC, the amount of data specified by the SCSI command is greater than the amount of data to be transported in the USB data transport.
8	Even though a WRITE command has been issued from the host PC, the host PC requests for data in the USB data transport.
9	When a WRITE command is issued from the host PC, the amount of data to be transported in the USB data transport is a value other than 0 while the amount of data specified by the SCSI command is 0.
10	Even though a READ command has been issued from the host PC, the host PC sends data in the USB data transport.
11	When a WRITE command is issued from the host PC, the amount of data specified by the SCSI command is less than the amount of data to be transported in the USB data transport.
13	When a WRITE command is issued from the host PC, the amount of data specified by the SCSI command is greater than the amount of data to be transported in the USB data transport.

Table 4.12: Sample Error Conditions

Table 4.13 shows how a function operates when each error condition occurs.

Case No.	Relation between Host PC and Function
2, 3	• Set 0x02 as the CSW status.
4, 5	• The function adds data to become equal to the data length set in dCBWDataTransferLength and then sends data to the host PC. • Set the amount of data added in the data transport in dCBWDataResidue of CSW. • Set 0x00 as the CSW status.
7, 8	• The function sends data to the host PC up to the data length set in dCBWDataTransferLength. • Set 0x02 as the CSW status.
9, 11	• The function receives data from the host PC up to the data length set in dCBWDataTransferLength. • Set the difference between the amount of data received in the data transport and the amount of data processed by the function in dCBWDataResidue of CSW. • Set 0x01 as the CSW status.
10, 13	• The function receives data from the host PC up to the data length set in dCBWDataTransferLength. • Set 0x02 as the CSW status.

Table 4.13: Function Operation for Each Error Condition

Figures 4.6 to 4.8 show the processing when a data transfer error occurs.

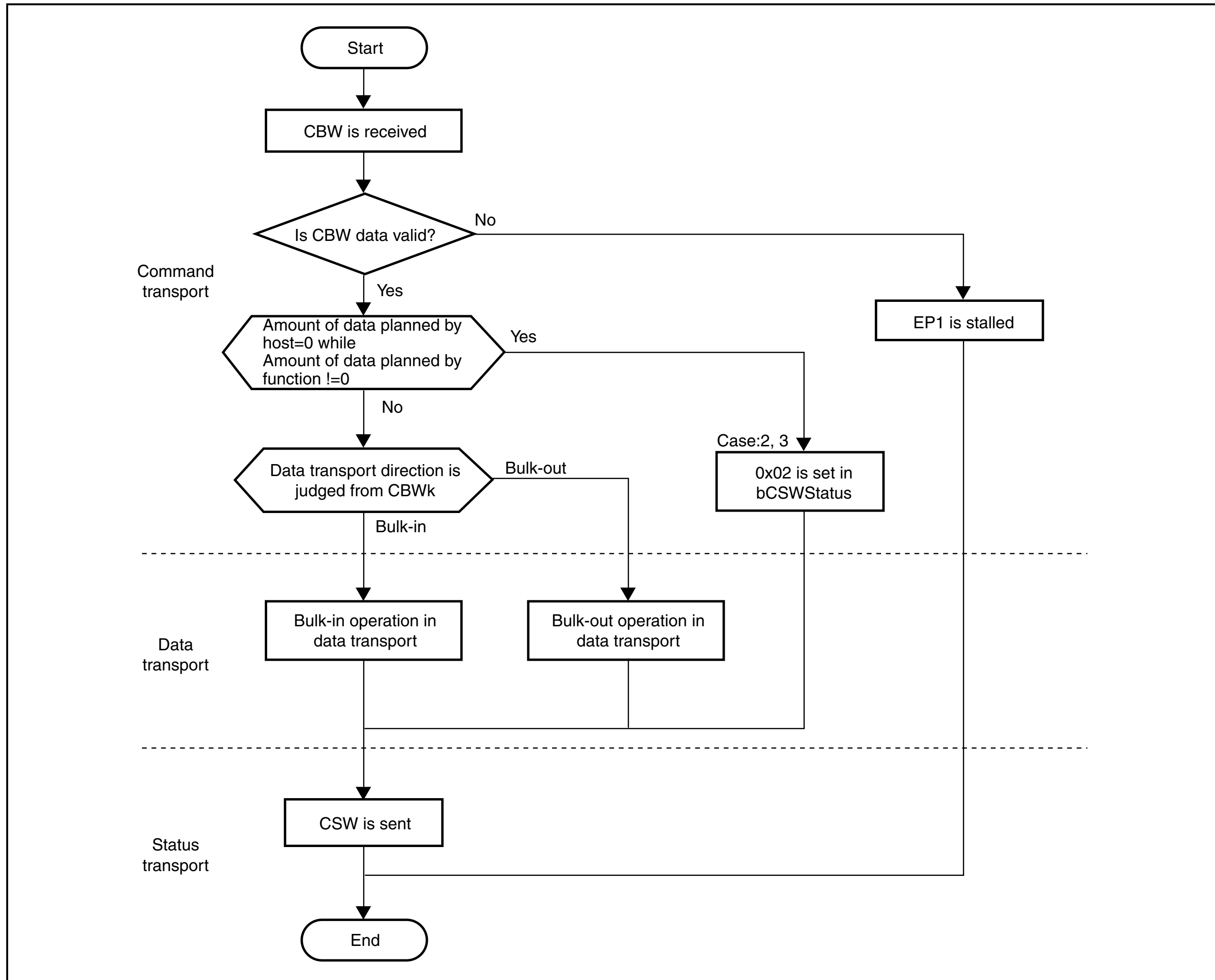

Figure 4.6: Error Processing Flow in Data Transfer (1)

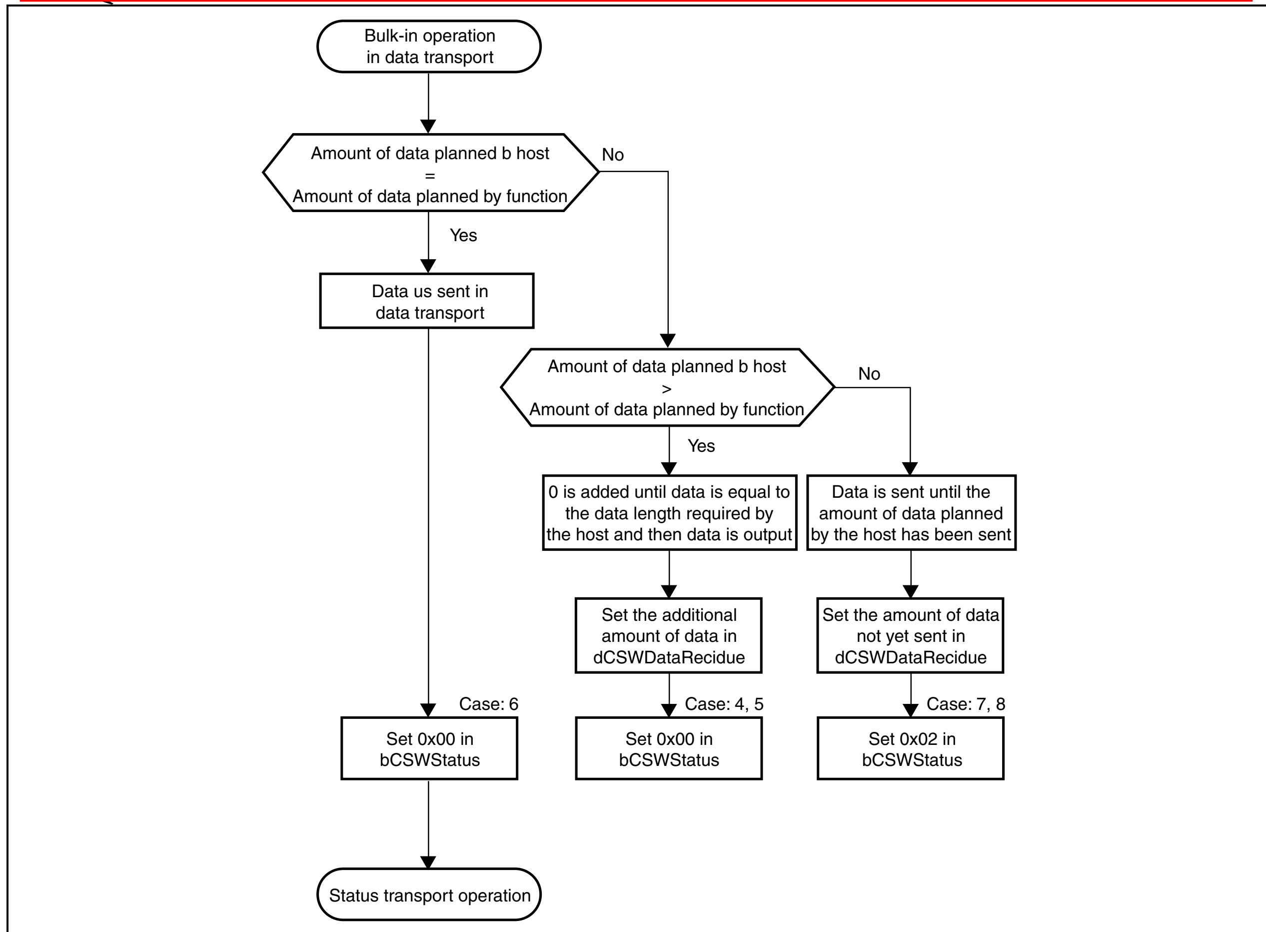

Figure 4.7: Error Processing Flow in Data Transfer (2)

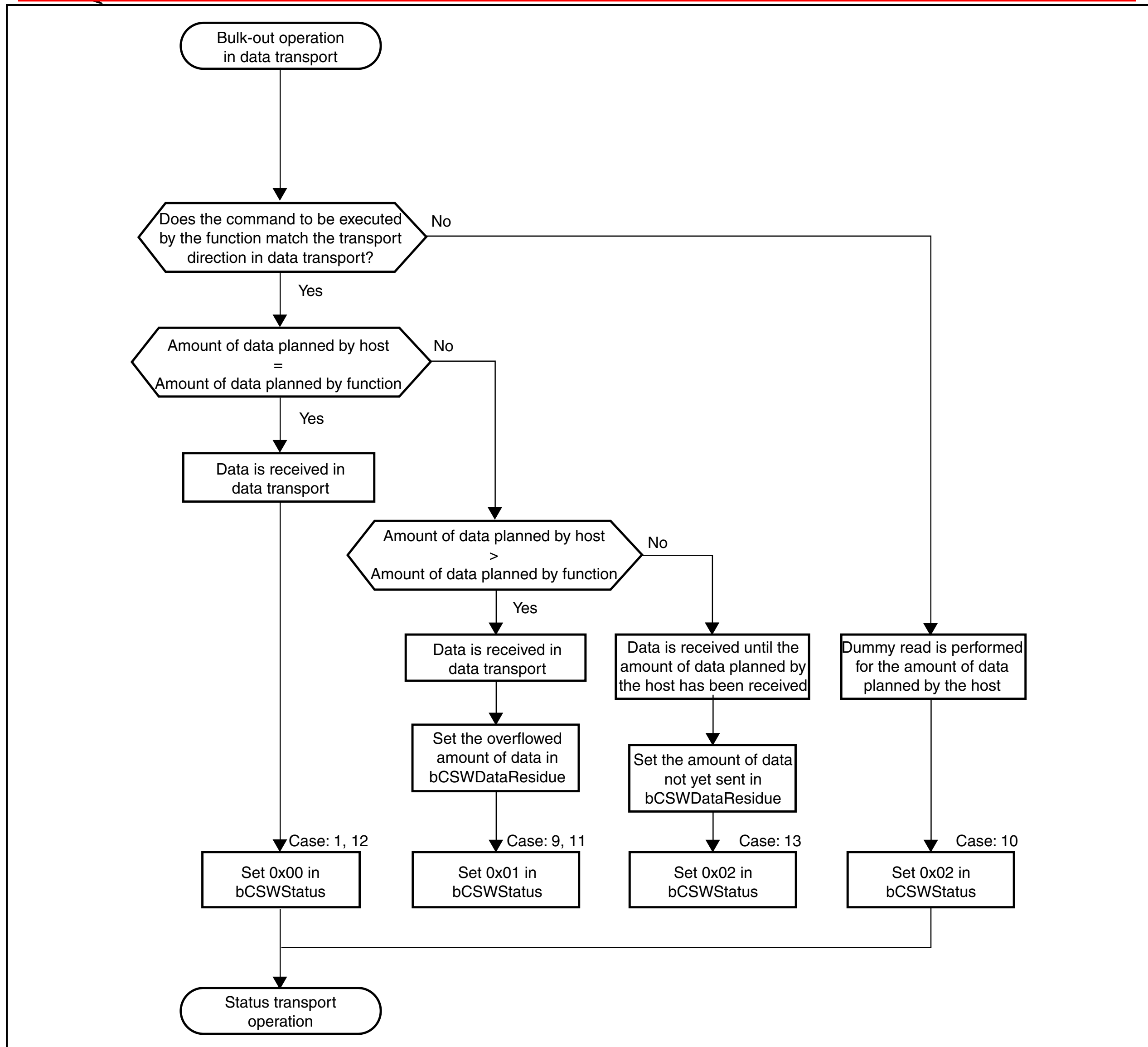

Figure 4.8: Error Processing Flow in Data Transfer (3)

When a Mass Storage Class (Bulk-Only Transport) transmission is carried out, transport of the CBW initiates a series of data transfers, and when the CSW is transported to the host PC, a series of data transfers is processed. This status contains two items: dCSWStatus that indicates the transport result, and dCSWDataResidue that indicates the number of error bytes.

In this sample program, the following two fields are used to create these two items.

- dCBWDataTransferLength field of CBW packet
- dCSWDataTransferResidue field of CSW packet

The dCBWDataTransferLength field of the CBW packet is used as the variable in which the number of data bytes the host PC specifies to be handled in the data transport is entered.

The dCSWDataTrasferResidue field of the CSW packet is used as the variable in which the number of data bytes the function handles in the data transport is entered.

When the CBW transport has been completed, the number of data bytes planned to be handled in the data

transport by the host PC and the function are stored in the dCBWDataTransferLength and dCSWDataTransferResidue fields, respectively.

Data is transferred in the data transport according to the flowcharts.

If data transport between the host PC and function has been processed without errors, the values in the dCBWDataTransferLength and dCSWDataTransferResidue fields are both subtracted by the number of bytes that have been transferred for every data transfer in the data transport. For other cases, the difference between the number of data bytes the host PC requires to be handled in the data transport and the number of data bytes the function has handled in the data transport is stored in the dCSWDataTransferResidue field of the CSW packet, and operation then moves to the status transport.

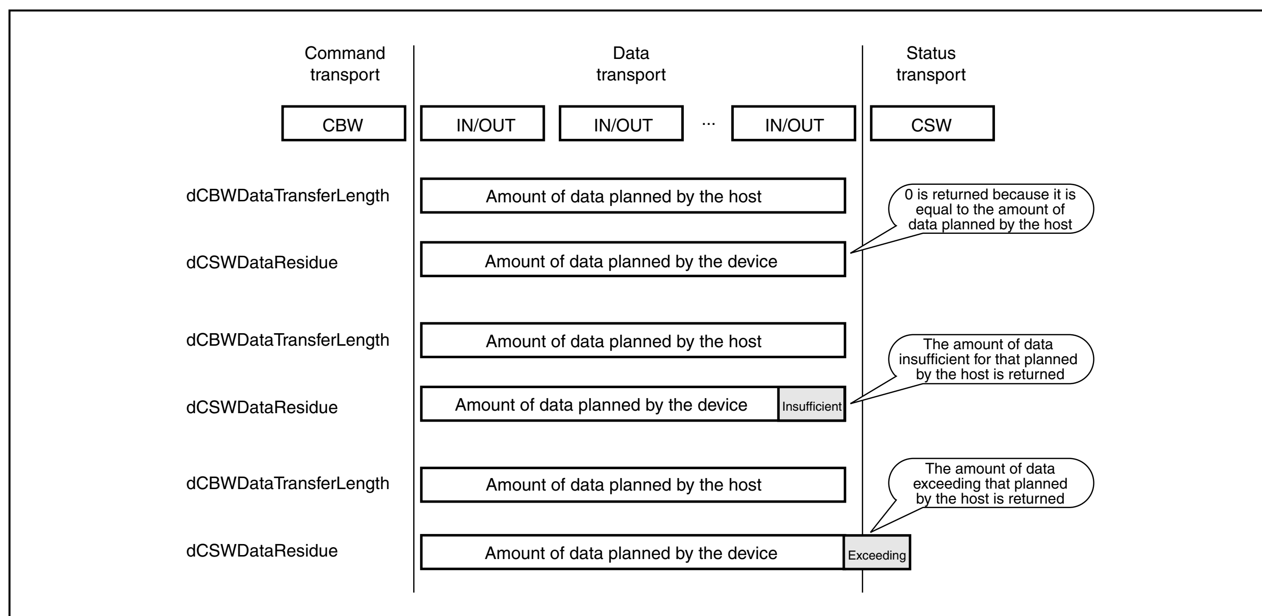

Figure 4.9: Each Stage in Bulk-Only Transport

5. Limitations

One of the limitations with the current implementation is that since CPU-Rewrite is not implemented, data stored on the chip will be lost on power recycling. In order to prevent this, CPU-rewrite will have to be implemented so that data is written to the ROM as opposed to the on chip RAM. Implementing this functionality will also overcome the storage area limitation (6.6 Kb) in the current design.

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

1. H8S/2218 group manual. Document number: REJ09B0074-0400O
2. Universal Serial Bus Specification Revision 2.0
3. Universal Serial Bus Mass Storage Class (Bulk Only Transport)Revision1.0
4. The 3DK2218 User Manual
5. “USB Complete: Everything You Need to Develop Custom USB Peripherals” by Jan Axelson.
6. 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.