To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

RENESAS

Application Note

PFESiP/V850EP1

32-bit Microcontroller Dedicated to PFESiP® EP-1

USB Function Sample Software

Document No. A19349EJ1V0AN00 (1st edition)
Date published September 2008 NS

[MEMO]

NOTES FOR CMOS DEVICES

① VOLTAGE APPLICATION WAVEFORM AT INPUT PIN

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (MAX) and V_{IH} (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (MAX) and V_{IH} (MIN).

② HANDLING OF UNUSED INPUT PINS

Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

③ PRECAUTION AGAINST ESD

A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

④ STATUS BEFORE INITIALIZATION

Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

⑤ POWER ON/OFF SEQUENCE

In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current.

The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

⑥ INPUT OF SIGNAL DURING POWER OFF STATE

Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.

Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

These commodities, technology or software, must be exported in accordance with the export administration regulations of the exporting country. Diversion contrary to the law of that country is prohibited.

M8E 02. 11-1

To obtain the latest documents when designing, contact an NEC sales office or a distributor.

PREFACE

Readers This manual is intended for users who understand the functions of the microcontroller function chip with an on-chip V850E2 CPU core (PFESiP/V850EP1) and wish to evaluate developing PFESiP EP-1 Series products using the chip.

Purpose This manual is intended to help users understand the USB function sample software of the PFESiP/V850EP1.

How to Read This Manual It is assumed that the readers of this manual have general knowledge of electrical engineering, logic circuits, microcontrollers, SRAM, page ROM, and SDRAM.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	xxxZ (Add Z after pin or signal name)
Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Binary ..... XXXX or XXXXB
	Decimal XXXX
	Hexadecimal ...XXXXH
Prefix indicating power of 2	
(Address space, memory capacity):	K (kilo): 2^{10} = 1,024
	M (mega): 2^{20} = $1,024^2$
	G (giga): 2^{30} = $1,024^3$
Data type:	Word ... 32 bits
	Halfword ... 16 bits
	Byte ... 8 bits

Related Documents The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such. Furthermore, some related documents may be intended for individual customers, because the documents are prepared in the development/planning stage of each core.

Documents related to PFESiP EP-1 Series

Document Name	Document No.
V850E2 Architecture User's Manual	U17135E
PFESiP EP-1 Series Design Manual	A19068E
PFESiP/V850EP1 Product Data User's Manual	A19069E
PFESiP/V850EP1 Hardware (CPU Function) User's Manual	A19070E
PFESiP/V850EP1 Hardware (USB Function) User's Manual	A19071E
PFESiP/V850EP1 USB Function Sample Software Application Note	This manual

Documents related to PFESiP EP-1 Evaluation Board

Document Name	Document No.
PFESiP EP-1 Evaluation Board Technical Information User's Manual	A19350E
PFESiP EP-1 Evaluation Board Lite Technical Information User's Manual	A19354E

Documents related to development tools

Document Name			Document No.
RX850 Pro User's Manual	Ver.3.21	Basics	U18165E
	Ver.3.20	Installation	U17421E
	Ver.3.21	Technical	U18164E
	Ver.3.20	Task Debugger	U17422E

CONTENTS

LIST OF FIGURE

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 Overview

The USB storage class driver is a sample program for the USB function controller built into the PFESiP/V850EP1. It complies with Universal Serial Bus Specification Revision 1.1 and Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0 and operates in the real-time operating system for embedded control, RX850 Pro (complies with the μ ITRON 3.0 specifications).

This sample program uses a control endpoint (endpoint number 0) and IN and OUT (endpoint numbers 1 and 2) of a bulk endpoint. It connects to the standard storage class host driver provided with Windows XP and controls storage devices (virtual devices). The Mass Storage class is defined as the class.

This sample program uses the development evaluation board PFESiP EP-1 Evaluation Board (including PFESiP EP-1 Evaluation Board Lite which is omitted hereinafter) as the hardware execution environment. To use the PFESiP EP-1 Evaluation Board and sample program as they are, create an execution object according to the procedure described in **CHAPTER 6 LOAD MODULE** and check the operation according to **CHAPTER 2 LOAD MODULE EXECUTION**.

To use this sample program by changing the PFESiP EP-1 Evaluation Board to another target board, do so by referring to **CHAPTER 3 SYSTEM BUILDUP**, **CHAPTER 4 PROCESSING PROGRAM DEPENDING ON RX850 Pro**, and **CHAPTER 5 LINK DIRECTIVE FILE** and observing the board specifications.

To change the PFESiP EP-1 Evaluation Board and sample program for use, do so by referring to **CHAPTER 3 SYSTEM BUILDUP**, **CHAPTER 4 PROCESSING PROGRAM DEPENDING ON RX850 Pro**, **CHAPTER 5 LINK DIRECTIVE FILE**, **CHAPTER 6 LOAD MODULE**, and **CHAPTER 7 USB STORAGE CLASS DRIVER FUNCTIONS** and making the required changes.

The positioning of the USB storage class driver is shown below.

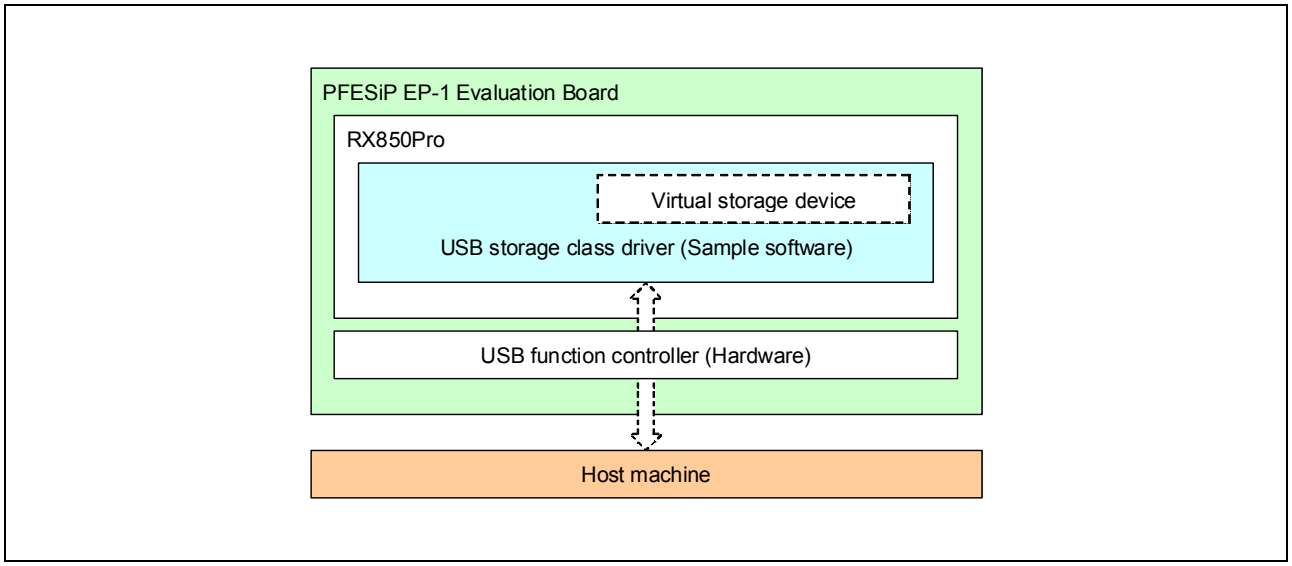
Caution **This sample program operates as a Mass Storage device (interface class: mass storage, interface subclass: SCSI, interface protocol: Bulk-Only Transport protocol). The storage device used in this manual has no logical unit connected and operates such that a removable disk is virtually connected by securing a memory area (block size: 512 bytes, number of logic blocks: 192, capacity: 96 KB).**

Remarks 1. See the following documents for details of the USB Mass Storage class.

Universal Serial Bus Mass Storage Class Specification Overview Revision 1.1
Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0
Universal Serial Bus Mass Storage Class UFI Command Specification Revision 1.0

2. **2.1 Load Module Execution Procedure** is described assuming the environment shown in **1.3 Execution Environment**.

Figure 1-1. Positioning of USB Storage Class Driver



1.2 Development Environment

This manual is described assuming the following environment as the hardware and software environments required for developing a system by using the sample program.

Hardware environment

Host machine:	A machine compatible with PC/AT (OS: Windows XP)

Software environment

Real-time OS:	RX850 Pro Version 3.21
USB storage class driver:	The sample program set explained in this chapter
Project Manager:	PM+ Version 6.30
C compiler package:	CA850 Version 3.10

Caution If the directory configuration differs from that handled by a sample project file, edit the project file according to the environment.

Remark See the project manager help for how to edit a project file.

1.3 Execution Environment

This manual is described assuming the following environment as the hardware and software environments required for executing a load module by using the sample program.

Hardware environment

Host machine:	A machine compatible with PC/AT (OS: Windows XP)
IE control machine:	A machine compatible with PC/AT (OS: Windows XP)
Target board:	PFESiP EP-1 Evaluation Board / PFESiP EP-1 Evaluation Board Lite
In-circuit emulator (IE):	RTE-2000-TP (made by Midas lab Inc.)
JTAG probe	
USB cable	

Software environment

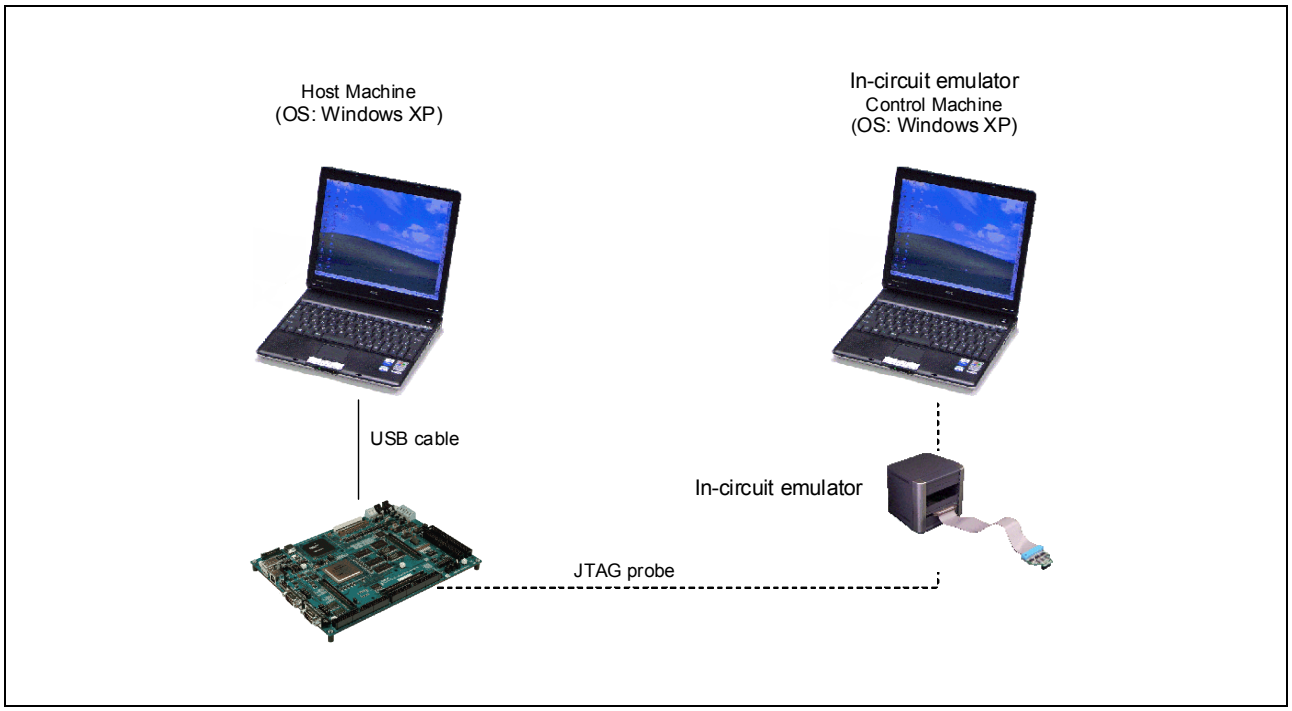
Project Manager:	PM+ Version 6.30
IE software:	ID850QB Version 3.40

Remarks 1. See the **PFESiP EP-1 Evaluation Board User's Manual** for details of how to set up an execution environment.

2. See the **RTE-2000-TP User's Manual** for details of how to set up the in-circuit emulator (RTE-2000-TP).

3. See the **ID850QB User's Manual** for details of the ID850QB.

Figure 1-2. Execution Environment



CHAPTER 2 LOAD MODULE EXECUTION

2.1 Load Module Execution Procedure

The procedure for executing a load module in the environment shown in **1.3 Execution Environment** is described below, taking a load module that uses this sample program as an example.

<1> In-circuit emulator (IE) control machine setup

Turn on and start the IE control machine.

Turn on and set up the in-circuit emulator as well.

<2> Host machine setup

Turn on and start the host machine. (The IE control machine can be used as the host machine, but it is strongly recommended to provide a separate host machine for development.)

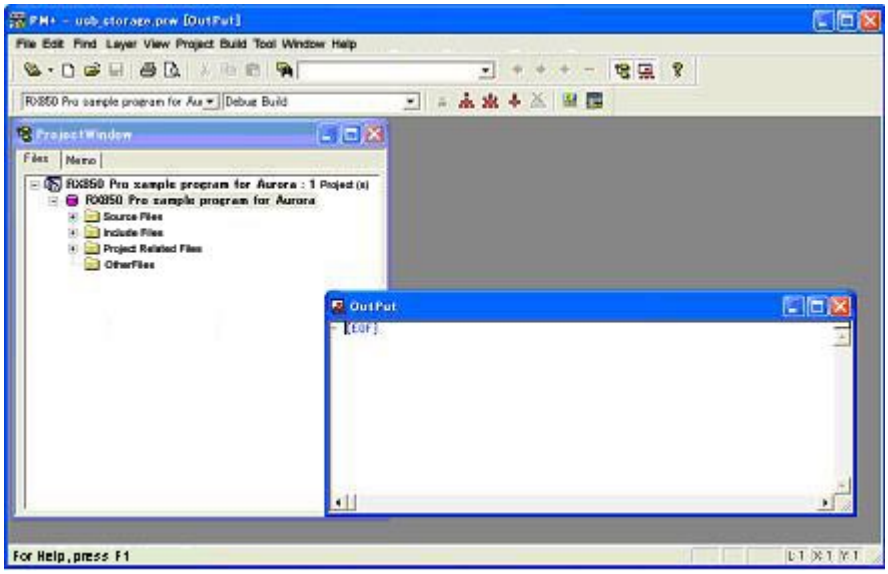
<3> Resetting PFESiP EP-1 Evaluation Board

Reset the PFESiP EP-1 Evaluation Board by pulling SW_RESET of the PFESiP EP-1 Evaluation Board towards you.

<4> Starting project manager (PM+)

Double-click the workspace file usb_storage.prw stored in the directory V850USB_Storage¥rx85p¥conf of the sample program in Windows Explorer and start PM+.

[PM+ start screen]



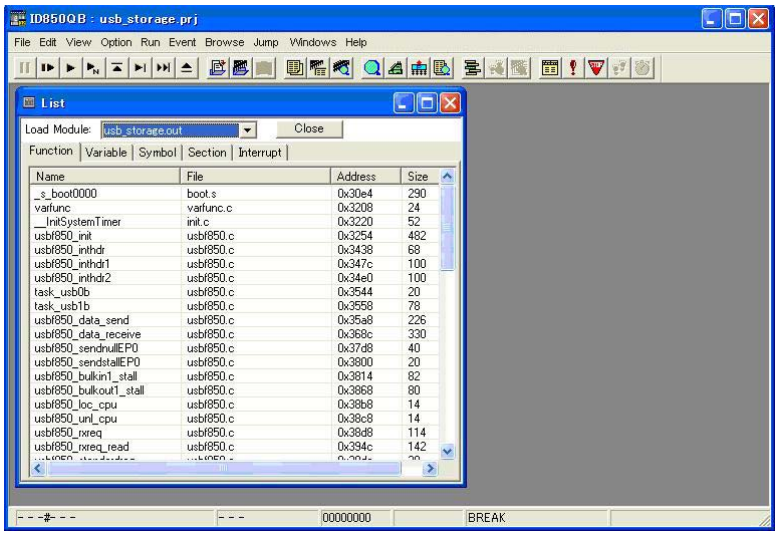
Remark See **2.2 Directory Configuration** for details of the sample program directory configuration.

<5> Starting debugger (ID850QB) and reading load module

Select [Build] → [Debug] on the PM+ toolbar and start the ID850QB.

Debug information is also read by the debugger as shown in the figure below when the load module is read by the on-chip instruction RAM of the PFESiP/V850EP1 of the PFESiP EP-1 Evaluation Board, because reading of a load module is set to be automatically performed in the sample program debugger information file usb_storage.pri.

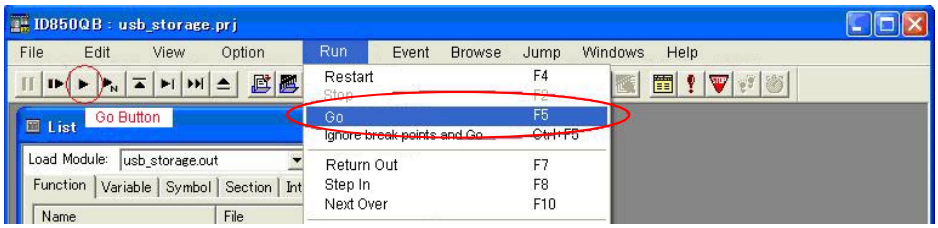
[ID850QB start screen]



<6> Execution

Execute the code read on-board by pressing the [F5] key, clicking the [Go] button, or selecting [Run] → [Go] on the toolbar.

[Program execution example]



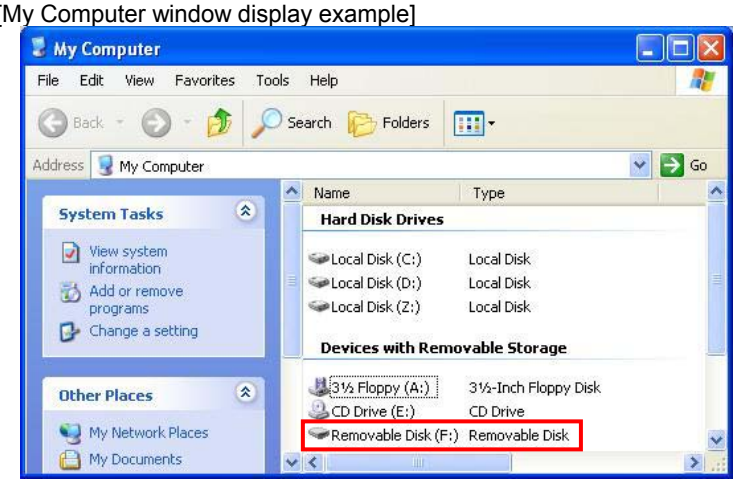
<7> USB connection

Connect the USB cable.

Connect the B connector to the board and the A connector to the host machine.

If the host machine recognizes a device, the standard USB storage class host driver of Windows XP will be automatically installed. If the device is normally recognized, it will be displayed as a “Removable Disk” in the My Computer window.

[My Computer window display example]

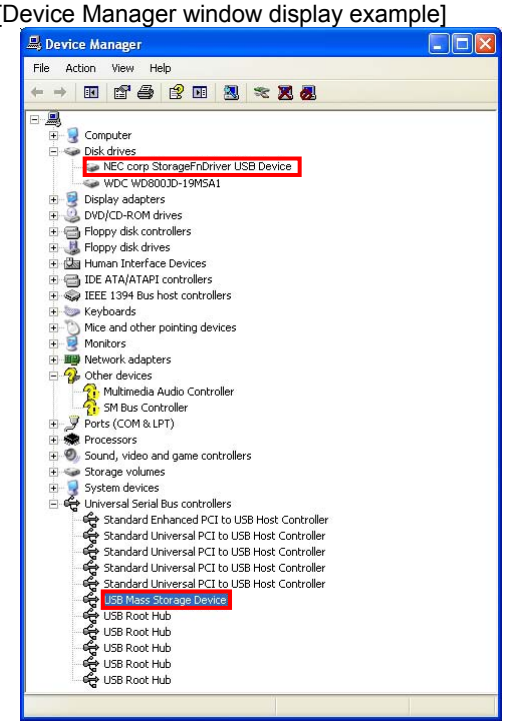


<8> Starting Device Manager

Select [My Computer] → [Properties] → [Hardware] tab.

Select [Device Manager] and start Device Manager.

[Device Manager window display example]



<9> USB device connection check

Check that “USB Mass Storage Device” is listed under “Universal Serial Bus controllers” and “NEC corp StorageFncDriver USB Device” under “Disk drives” in the Device Manager window.

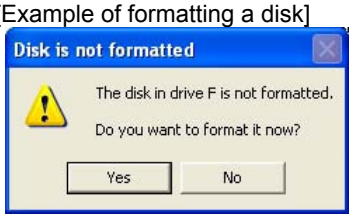
<10> Device use method

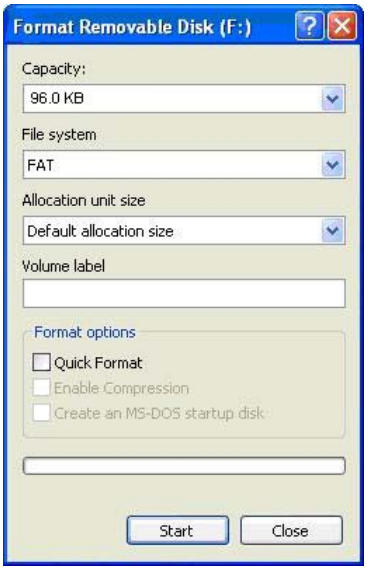
When [Open] is selected for the “Removable disk” in the My Computer window, formatting of the disk will be requested. Format the disk by following the instructions displayed on-screen.

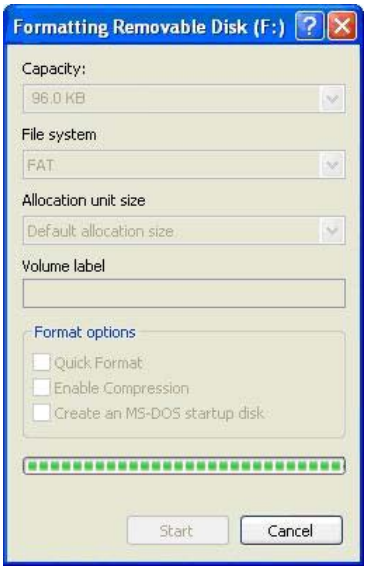
When formatting the disk is completed normally, files can be read, written, or deleted from the disk as with a normal disk device.

The contents of the disk will be retained until executing the load module is stopped.

[Example of formatting a disk]





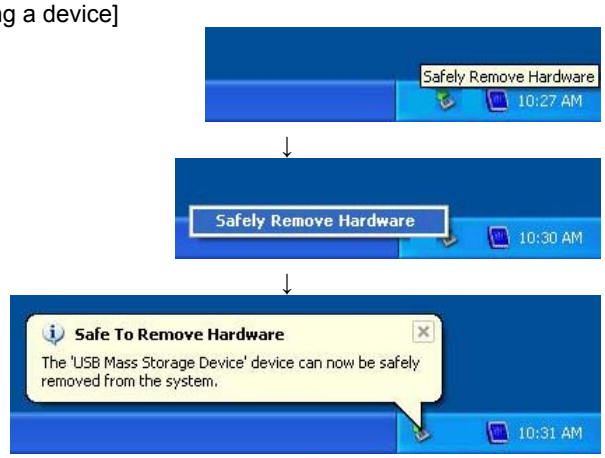


<11> Stopping using device

Left-click the [Safely Remove Hardware] icon in the Windows task tray.

Select the USB mass storage device of the drive used in step <10> and stop the device.

[Example of stopping using a device]



<12> USB cable removal

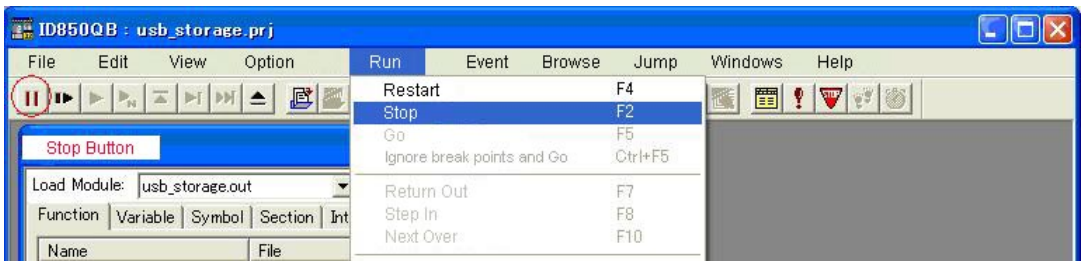
Check that using the device has been stopped in step <11> and remove the USB cable.

<13> Program termination method

End the program under execution.

Stop the execution of the program by pressing the [F2] key, clicking the [Stop] button, or selecting [Run] → [Stop] on the toolbar.

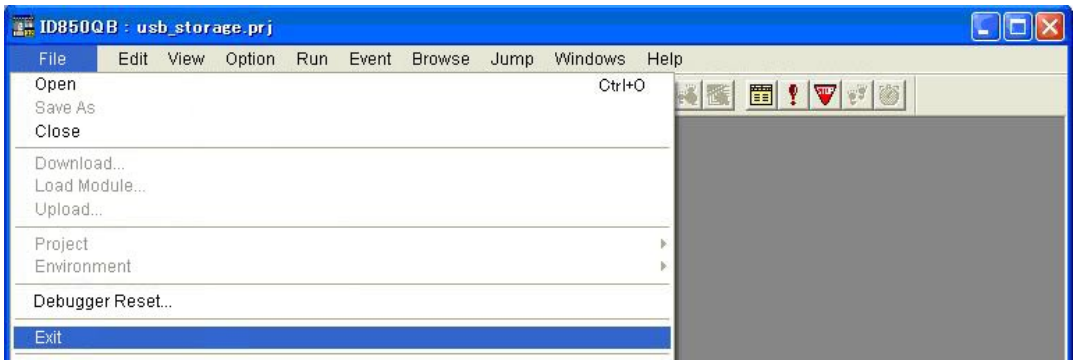
[Program termination example]



<14> ID850QB termination method

Select [File] → [Exit] on the toolbar to terminate the ID850QB.

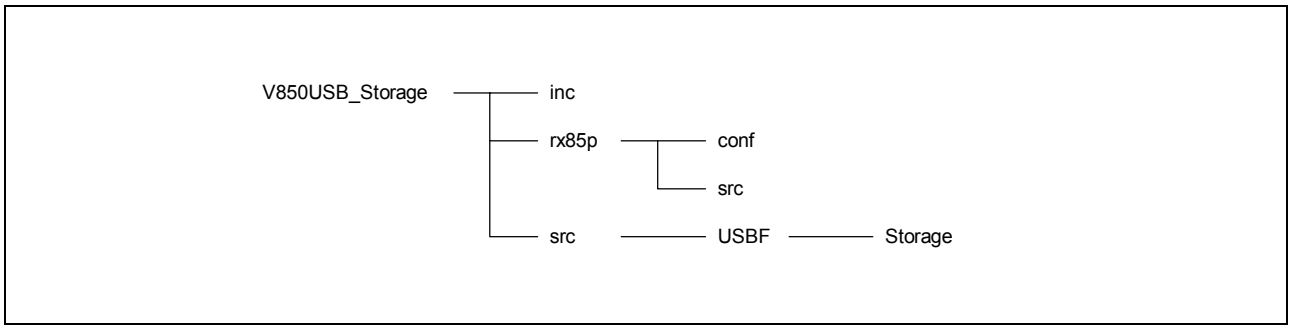
[ID850QB termination example]



2.2 Directory Configuration

The directory configuration of the files included in this sample program set is shown below.

Figure 2-1. Sample Program Directory Configuration



An overview of each directory is shown below.

(1) V850USB_Storage

This is the USB storage class driver directory.

(2) V850USB_Storage¥inc

This is the directory in which USB storage class driver header files are stored.

* aurora_usb_reg.h: USB-related register definition file of PFESiP/V850EP1
* errno.h: Return value header file
* types.h: Data type header file

(3) V850USB_Storage¥rx85p

This is the directory in which the files for the RX850 Pro are stored.

(4) V850USB_Storage¥rx85p¥conf

This is the directory in which the system files for the RX850 Pro are stored.

* usb_storage.prw: USB storage class driver workspace file
* usb_storage.prj: USB storage class driver project file
* usb_storage.pri: USB storage class driver debugger information file
* usb_storage.tcl: ID850QB start operation setting script file
* usb_storage.out: USB storage class driver load module
* sys.h: System information header file
* sys.s: System information table
* sct.s: System call table

Caution sys.h (system information header file), sys.s (system information table), and sct.s (system call table) are files generated during building. Normally, these files are generated via command manipulation performed by using a configurator. When a sample build file is used, however, these files are not required to be generated in advance, because commands are set to be executed during build execution.

(5) V850USB_Storage¥rx85p¥src

This is the directory in which the files for the RX850 Pro are stored.

* boot.s: Boot processing assembler file
* entry.s: Entry processing assembler file
* init.c: Hardware initialization block source file
* init.h: Hardware initialization block header file
* sys.cf: CF definition file
* varfunc.c: Software initialization block source file
* usb_storage.dir: Link directive file

(6) V850USB_Storage¥src

This is the directory in which the files of the board-dependent blocks of the USB storage class driver are stored.

* port.c: Port setting source file
* port.h: Port setting header file

(7) V850USB_Storage¥src¥USBF

This is the directory in which the files of the USB processing block of the USB storage class driver are stored.

* usbf850.c: USB device source file
* usbf850.h: USB device header file
* usbf850desc.h: USB descriptor definition file
* usbf850_dma.c: DMA control source file
* usbf850_dma.h: DMA control header file
* usbf850_storage.c: Source file for interface between USB and storage device
* usbf850_storage.h: Header file for interface between USB and storage device

(8) V850USB_Storage¥src¥USBF¥Storage

This is the directory in which the files of the storage device processing block of the USB storage class driver are stored.

* ata_ctrl.c: Storage device control source file
* ata.h: Storage device header file
* scsi_cmd.c: SCSI command processing source file
* scsi.h: SCSI command processing header file

CHAPTER 3 SYSTEM BUILDUP

3.1 Overview

System buildup refers to generating a load module by using the files installed in the user development environment (host machine) from a medium providing a USB storage class driver.

The procedure for building a USB storage class driver system is shown below.

<1> Describing processing blocks depending on the RX850 Pro

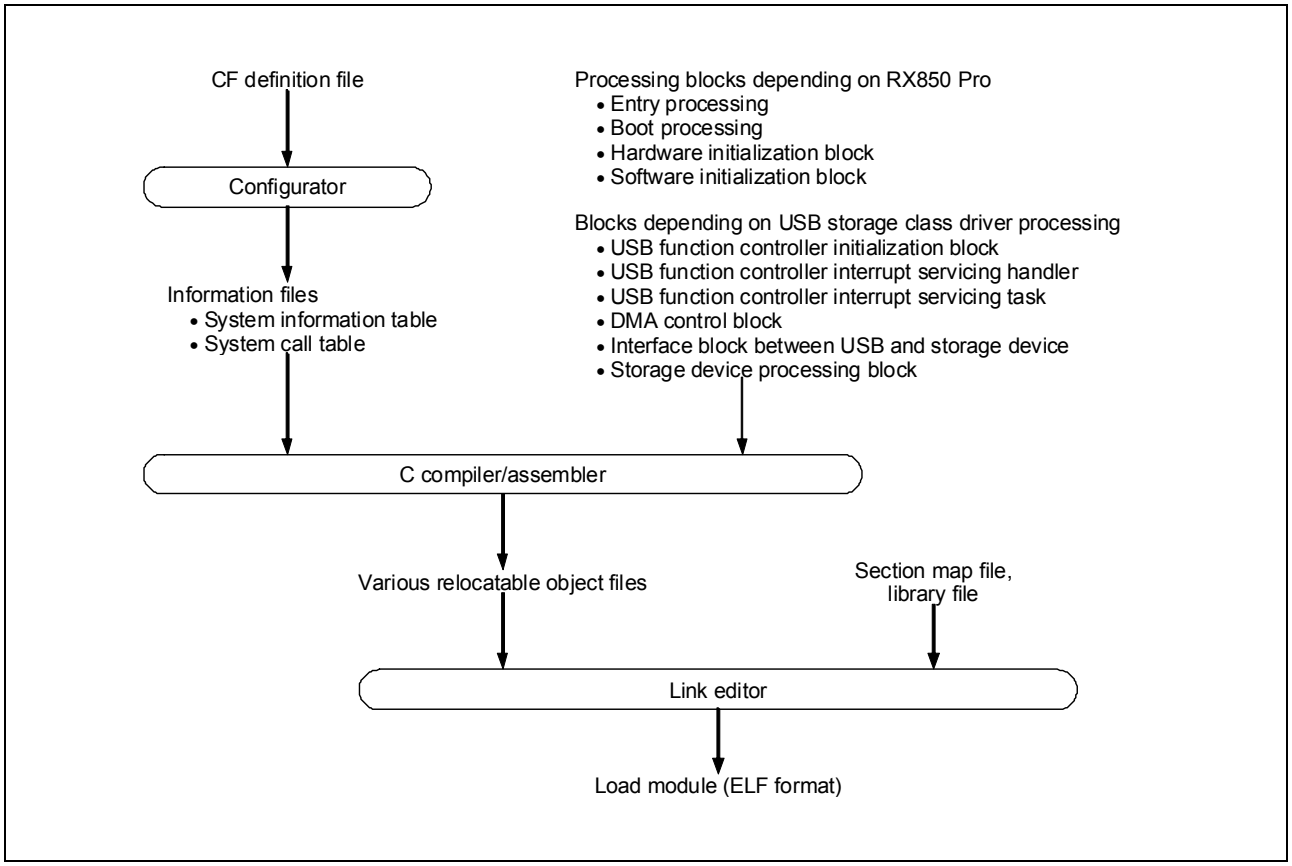
<2> Describing a board-dependent block

<3> Describing blocks depending on the USB storage class driver processing

<4> Describing a section map file

<5> Generating a load module

Figure 3-1. System Buildup Procedure



3.2 Describing Processing Blocks Depending on RX850 Pro

Functions of the real-time OS (RX850 Pro) are used as some functions provided by the USB storage class driver. Furthermore, a processing program described by a user will be executed under the management of the RX850 Pro.

Consequently, to operate the RX850 Pro normally, the processing blocks depending on the RX850 Pro must be described.

The processing blocks depending on the RX850 Pro are shown below.

- CF definition file
- Entry processing
- System initialization processing
  - Boot processing
  - Hardware initialization block
  - Software initialization block

Remark See **CHAPTER 4 PROCESSING PROGRAM DEPENDING ON RX850 Pro** for details of the processing program depending on the RX850 Pro.

3.3 Describing Board-Dependent Blocks

The USB storage class driver source program is the initialization processing related to the processing depending on the user execution environment/application system, cut out as a board-dependent block.

The board-dependent block is shown below.

- CPU-board-dependent block

  The I/O port manipulation required for the USB bus driver is cut out as a CPU-board-dependent block.

  The PFESiP EP-1 Evaluation Board can perform VBUS detection by VBUSDET only if the PFESiP/V850EP1 outputs a low level from P157. Consequently, it is described in the port setting source program (port.c) such that a low level is output from P157.

Caution A dedicated function is not provided, because port settings are handled similarly to the settings of other registers. See the boot processing block (boot.s) and the port setting source program (port.c) called from the software initialization block for the processing method.
Port registers are defined in the PFESiP/V850EP1 device file.

3.4 Describing Blocks Depending on USB Storage Class Driver Processing

In this sample program, the driver functions to perform the USB storage class driver functions are cut out as blocks depending on the USB storage class driver processing.

The blocks depending on the USB storage class driver processing are shown below.

- USB function controller initialization block
- USB function controller interrupt handler
- USB function controller interrupt servicing task
- USB function controller general-purpose functions
- DMA control block
- Interface block between USB and storage device
- Storage device processing block

Remark See **CHAPTER 7 USB STORAGE CLASS DRIVER FUNCTIONS** for details of the blocks depending on the USB storage class driver processing.

3.5 Describing Section Map File

A section map file is a file to be used by the user to fix the address assignment performed by the link editor. When using the RX850 Pro, the following five text areas are the required sections.

- Common-section placement area: .system section
- Interrupt servicing-related placement area: .system_int section
- Scheduler-related placement area: .system_cmn section
- System information area: .sit section
- Interface library/system call placement area: .text section

Remark See **CHAPTER 5 LINK DIRECTIVE FILE** for details of the section map file.

3.6 Generating Load Module

Generate an ELF-format load module by executing the C compiler, assembler, or linker for the coded processing program depending on the RX850 Pro, blocks depending on the USB storage class driver processing, and section map file.

Remark See **CHAPTER 6 LOAD MODULE** for details of the load module generation procedure.

CHAPTER 4 PROCESSING PROGRAM DEPENDING ON RX850 Pro

4.1 Overview

Functions of the real-time OS (RX850 Pro) are used as some functions provided by the USB storage class driver. Furthermore, a processing program described by a user will be executed under the management of the RX850 Pro.

Consequently, to operate the RX850 Pro normally, the processing blocks depending on the RX850 Pro must be described.

The processing blocks depending on the RX850 Pro are shown below.

- CF definition file
- Entry processing
- System initialization processing
  - Boot processing
  - Hardware initialization block
  - Software initialization block

4.2 CF Definition File

To build a system that uses the RX850 Pro, an information file (CF definition file) that retains the various data provided to the RX850 Pro is required.

The information required for performing the USB storage class driver functions is shown below.

- Real-time OS information
  - RX series information
- SIT information
  - System information
  - Maximum system value information
  - System memory information
  - Task information
  - Interrupt handler information
  - Initialization handler information
- SCT information
  - Task management/task-attached synchronization management function system call information
  - Interrupt servicing management function system call information
  - Time management function system call information

Caution **In this sample program, five tasks, three interrupt handlers, and seven system calls are used to perform the various functions. Consequently, in the CF definition file, the sta_tsk, ext_tsk, slp_tsk, and wup_tsk system calls must be defined to be used in the task management/task-attached synchronization management function system call information, the loc_cpu and unl_cpu system calls must be defined to be used in the interrupt servicing management function system call information, and the dly_tsk system call must be defined to be used in the time management function system call information, in addition to securing five tasks as the maximum number of maximum system value information tasks generated and securing three interrupt handlers as the maximum number of interrupt handlers generated for the USB storage class driver.**

Remark See the **RX850 Pro Installation User's Manual** and the sample CF definition file (sys.cf) for details of how to code a CF definition file.

4.2.1 Information file generation procedure

The procedure for generating information files (system information table, system call table, system information header file) is described below. Note that the information files are generated by using the Windows command prompt.

Caution When the sample build file is used, the information files are generated during building. Consequently, the information files are not required to be generated via the following procedure.

<1> Moving to directory

Use the Windows cd command to move to the directory in which the CF definition file is stored. An input example when the directory in which the CF definition file is stored is C:¥sample is shown below.

[Command input example]

```
C:>cd C:¥sample¥rx850<Enter>
```

<2> Information file generation

Use the configurator cf850pro.exe to generate the information files from the CF definition file created in a unique description format. An input example when generating three information files (system information table: sit.850, system call table: svc.850, system information header file: sys.h) from the input file (CF definition file name: sys.cf) is shown below.

[Command input example]

```
C:>cf850pro ?i sys.s ?c scy.s ?d sys.h sys.cf<Enter>
```

The information files can be generated from the CF definition file via the above-mentioned operation.

Caution This sample program is provided with a sample file (CF definition file) for generating the information files.

Remark See the **RX850 Pro Installation User's Manual** for the start options of and how to execute the configurator cf850pro.exe.

4.3 Entry Processing

A branch processing to an interrupt handler is assigned to the handler address to which the processor forcibly moves the control when a maskable interrupt occurs.

Assign the macro RTOS_IntEntry_Indirect (branch processing to the interrupt servicing management function provided by the RX850 Pro) to the handler address corresponding to the interrupt handler (defined by the interrupt handler information of the CF definition file) executed under the management of the RX850 Pro.

Remark See the included program entry.s for details of the entry processing coding method.

4.4 System Initialization Processing

The system initialization processing consists of a hardware initialization processing (boot processing, hardware initialization block) and software initialization processing (nucleus initialization block, initialization handler) which are required for the RX850 Pro to operate normally.

Consequently, the system initialization processing is the processing performed first when the system is started.

Caution Among the four types of system initialization processing, the nucleus initialization block is a part of the functions provided by the RX850 Pro and its processing is therefore not required to be described by the user.

The processing executed at the nucleus initialization block is shown below.

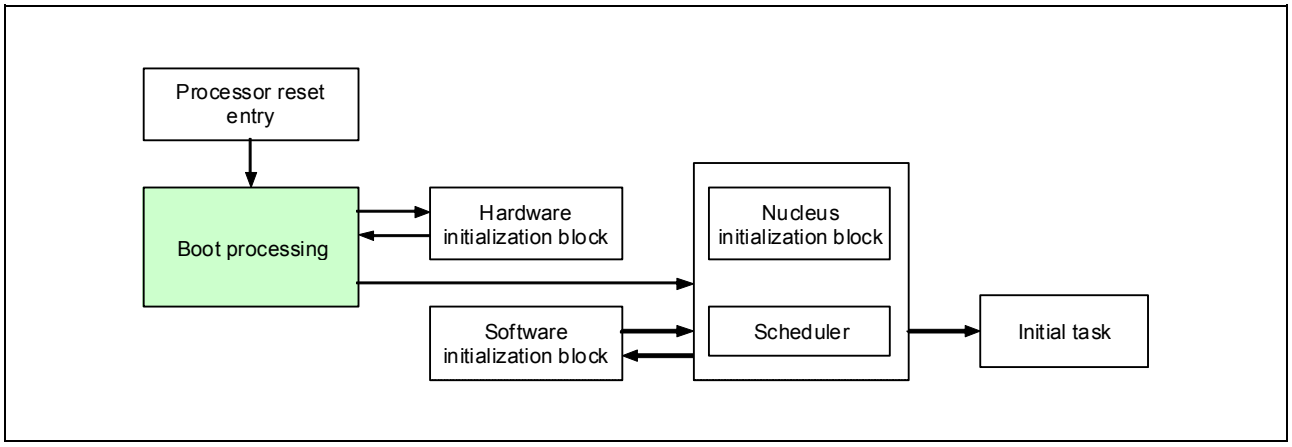
- Securing system memories defined by the CF definition file
  System Pool 0,1
  User Pool 0
- Generating/initializing management objects defined by the CF definition file
  Generating/starting a task
  Registering interrupt handlers
- Starting the initial task
- Generating/starting the idle task
- Calling the software initialization block
- Transferring the control to the scheduler

The idle task is a processing routine started by the scheduler when a processing program (task) under the management of the RX850 Pro is no longer in the run or ready state, i.e., when a processing program subject to the scheduling of the RX850 Pro no longer exists in the system, and issues the HALT instruction.

4.4.1 Boot processing

Boot processing is a function assigned to the reset entry of the processor and is executed first among the system initialization processing. The positioning of boot processing is shown below.

Figure 4-1. Positioning of Boot Processing



The processing to be executed in boot processing is described below.

Remark See the sample boot.s file for details of the boot processing coding method.

• tp, gp, ep register setting

When the system is started, the values of text pointer tp, global pointer gp, and stack pointer ep, which are required for executing the various processing programs (including boot processing) are undefined. Consequently, the initial setting of these registers is performed as the first processing of boot processing.

Caution In this chapter, it is recommended to set "0" to tp, "global pointer symbol_gp output by the compiler" to gp, and "element pointer symbol_ep output by the compiler" to ep.

• Calling the hardware initialization block

The function (hardware initialization block) provided for initializing the hardware in the target system is called. When executing the processing (internal-unit initialization) to be executed at the hardware initialization block somewhere else, the hardware initialization block is not required to be called.

Caution In this chapter, the hardware initialization block is not required to be called, because the processing (internal-unit initialization) to be executed at the hardware initialization block is executed at the software initialization block.
See the RX850 Pro Installation User's Manual for details.

• Transferring the control to the nucleus initialization block

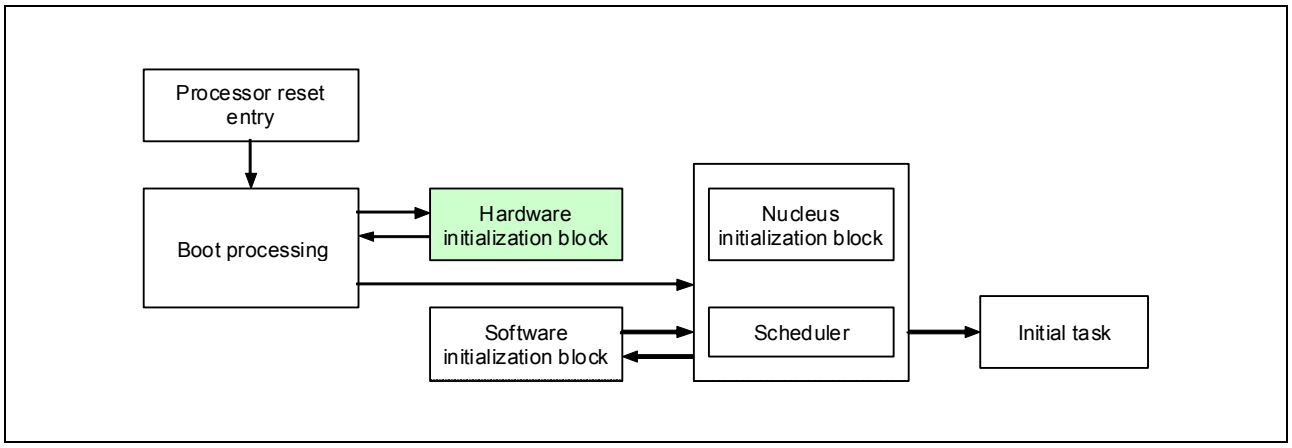
At the nucleus initialization block, system memories (System Pool 0, 1, User Pool 0) are secured and management objects are generated/initialized according to the information described in the system information table. Consequently, start address_sit of the system information table must be set to the r10 register before transferring the control to the nucleus initialization block.

Caution The system information table is a CF definition file created in a unique description format and converted into the assembly language format by using a utility tool (configurator cf850pro.exe) provided by the RX850 Pro.

4.4.2 Hardware initialization block

The hardware initialization block is a function provided for initializing the hardware in the target system and is called from boot processing. The positioning of the hardware initialization block is shown below.

Figure 4-2. Positioning of Hardware Initialization Block



The processing to be executed at the hardware initialization block is described below.

Caution 1. Maskable interrupts are not required to be disabled in particular, because they are masked by default during initialization.

2. In the sample program, the hardware is initialized at the software initialization block. See the RX850 Pro Installation User's Manual for details of the hardware initialization block.

- Returning the control to boot processing

Returning the control from the hardware initialization block, which is the boot processing call function, to boot processing is performed by issuing the "return();" instruction, because a return address is set for the lp register when the hardware initialization block is called in boot processing.

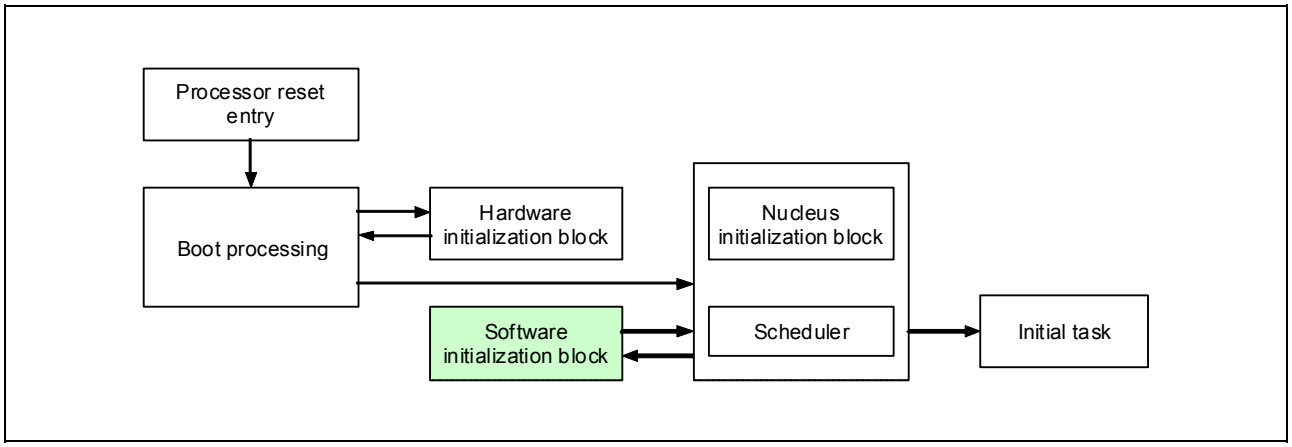
If the hardware initialization block is described in assembly language, the control will be returned to boot processing by issuing the "jmp [lp]" instruction.

4.4.3 Software initialization block

The initialization handler is a function provided for making the user software environment comfortable and is called from the nucleus initialization block.

The positioning of the software initialization block is shown below.

Figure 4-3. Positioning of Software Initialization Block



The processing to be executed at the software initialization block is described below.

Remark See the sample varfunc.c file for details of the software initialization block coding method.

- Internal-unit (real-time pulse unit (RPU)) initialization

The RX850 Pro performs timer operation functions (delayed task wakeup, starting the cyclic handler, timeout, etc.) by using timer interrupts generated at a fixed cycle. Consequently, the real-time pulse unit must be initialized before the RX850 Pro starts processing.

Note that a value such that a timer interrupt is generated at the base clock cycle defined by the system information in the CF definition file must be set to compare register CMD0 of the real-time pulse unit.

- Enabling timer interrupt acknowledgment

The acknowledgment of timer interrupts is enabled. Consequently, the timer operation functions (delayed task wakeup, timeout, starting the cyclic handler, etc.) provided by the RX850 Pro can be used when the processing of the nucleus initialization block ends.

- Returning the control to the nucleus initialization block

Returning the control from the initialization handler, which is the nucleus initialization block call function, to the nucleus initialization block is performed by issuing the “return();” instruction, because a return address is set for the lp register when the initialization handler is called at the nucleus initialization block.

If the initialization handler is described in assembly language, the control will be returned to the nucleus initialization block by issuing the “jmp [lp]” instruction.

4.5 Time Management Function

The time management in the RX850 Pro uses clock interrupts generated by hardware (such as the clock controller) at a fixed cycle.

When a clock interrupt is generated, the system clock processing of the RX850 Pro is called and processing related to time, such as updating the system clock, delayed task wakeup, and starting the cyclic handler is executed.

The system clock is a software timer that retains the time (48-bit width, unit: ms) that is used by the RX850 Pro for time management.

The system clock is updated in units of base clock cycles (specified during configuration) in system clock processing after it is set to “0H” in system initialization processing.

Caution The system clock managed by the RX850 Pro is configured of a 48-bit width. Consequently, the RX850 Pro ignores overflowed values (values that cannot be expressed in 48 bits). See the RX850 Pro Basics User’s Manual for details of the RX850 Pro time management functions.

CHAPTER 5 LINK DIRECTIVE FILE

5.1 Overview

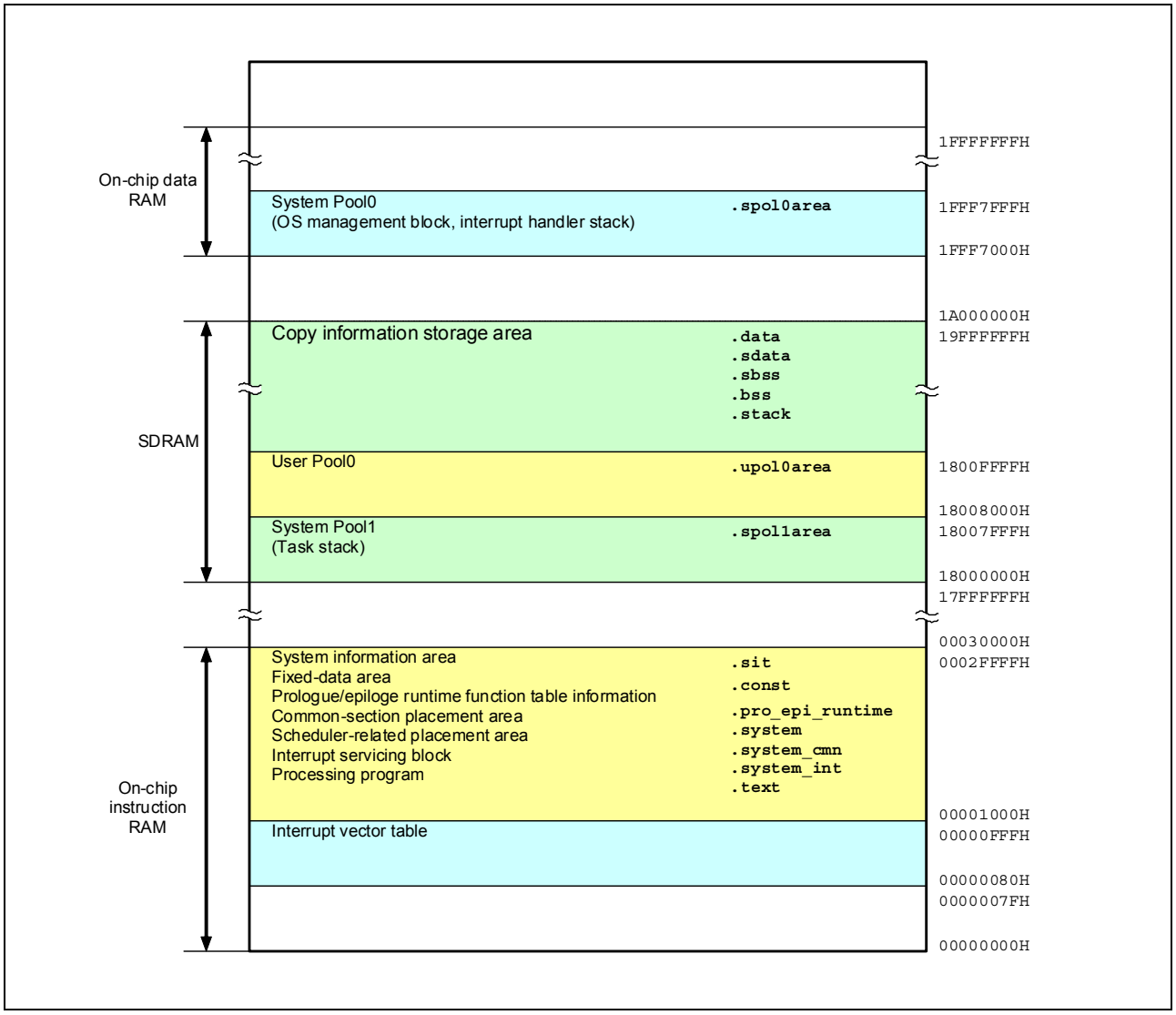
A link directive file is a file used by the user to fix the address assignment performed by the link editor.

The address assignments required other than the user processing programs (.data and .bss sections, etc) are described in **5.2 RX850 Pro Address Assignment** and **5.3 Other Address Assignment**.

The address assignment performed using the sample usb_storage.dir file is shown below.

Remark See the sample usb_storage.dir file for details of the link directive file coding method.

Figure 5-1. Address Assignment Example



5.2 RX850 Pro Address Assignment

The RX850 Pro consists of five text areas (common-section placement area, interrupt servicing-related placement area, scheduler-related placement area, system information area, interface library/system call placement area). This enables memory areas requiring a large capacity to be assigned to an external RAM and memory areas requiring high-speed accessing (interrupt servicing block, scheduling processing block) to be assigned to the on-chip instruction RAM (00000000H to 0002FFFFH).

Caution In the sample program, all five text areas are placed in the on-chip instruction RAM.

- Common-section placement area (.system section)

  This is an area to which the main body processing (task management function, task-attached synchronization function, etc.) of the RX850 Pro is assigned.

- Interrupt servicing-related placement area (.system_int section)

  This area consists of the pre-interrupt servicing performed when transferring the control to the interrupt handler and post-interrupt servicing performed when returning the control to the processing program in which a maskable interrupt occurred, among the interrupt servicing management functions provided by the RX850 Pro.

  Consequently, the performance of responding to the interrupt handler is improved by assigning the interrupt servicing block to the on-chip instruction RAM.

Caution In this chapter, it is recommended to assign the interrupt servicing block to the on-chip instruction RAM.

- Scheduler-related placement area (.system_cmn section)

  This area consists of the task wakeup processing and task scheduling processing among the scheduling functions provided by the RX850 Pro.

  Consequently, the system call processing accompanied by a scheduling processing is speeded up in addition to the task wakeup processing and task scheduling processing being speeded up by assigning the scheduling processing block to the on-chip instruction RAM.

Caution In this chapter, it is recommended to assign the scheduling processing block to the on-chip instruction RAM.

- System information area (.sit section)

  This is the area to which the system information table generated by executing configurator cf850.exe for the CF definition file is assigned.

  The system information table consists of the various data required for executing the nucleus initialization block (securing system memory, generating/initializing management objects).

- Interface library/system call placement area (.text section)

  This is the area to which instructions including system calls are assigned.

- System memory

  This area consists of various management blocks (task management block, semaphore management block, etc.) that will be required for performing the functions provided by the RX850 Pro, the area to which the interrupt handler stack will be assigned (System Pool 0), the area to which the task stack will be assigned (System Pool 1), and the area in which dynamic memory manipulation (memory block acquisition/release) from a processing program is enabled (User Pool 0).

Caution 1. The system memory start address must be specified when creating the CF definition file. Consequently, be sure to specify the address when defining the system memory in the section map file.

2. System memory section names can be freely specified by the user.

5.3 Other Address Assignment

The section requiring address assignment other than the RX850 Pro is described below.

- CA850 reserve area (.pro_epi_runtime section)
  This is the area to which the prologue/epilogue runtime function table is placed in the CA850.

Remark . See the **CA850 User's Manual** for details of the .pro_epi_runtime section.

CHAPTER 6 LOAD MODULE

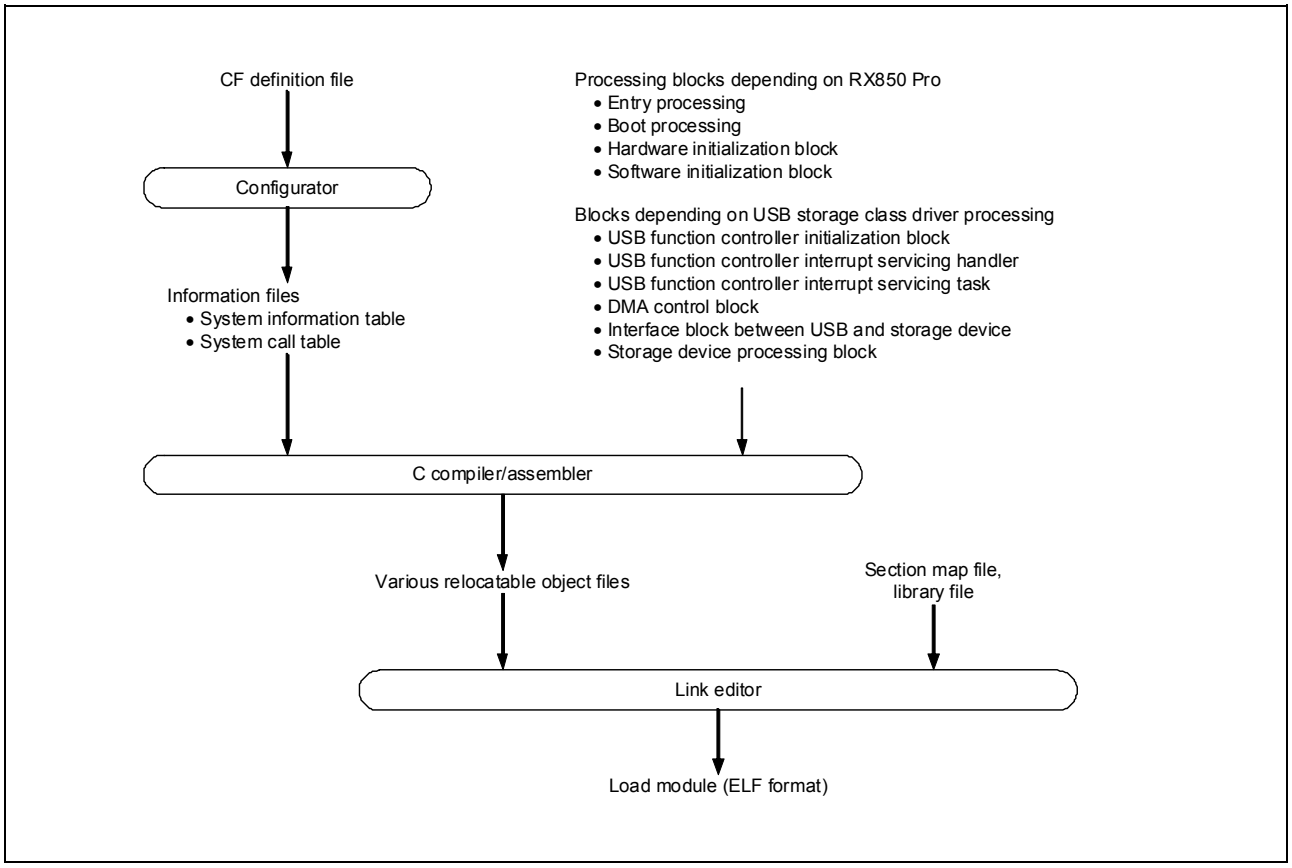
6.1 Overview

Create an ELF-format load module by executing the C compiler, assembler, or linker for the coded processing program depending on the RX850 Pro, blocks depending on the USB storage class driver processing, and section map file.

The load module generation procedure is shown below.

Caution A load module supporting the sample program can be generated by executing the sample .prj file. The .prj file path definition, however, must be corrected according to the user development environment.

Figure 6-1. Load Module Generation Procedure



6.2 Load Module Generation

The coded processing program depending on the RX850 Pro, blocks depending on the USB storage class driver processing, and section map file are turned into ELF-format load modules via the following procedure.

<1> System information table and system call table generation

The file format of the CF definition file created in a unique description format is not applicable to the link processing performed by the link editor when creating a load module.

Consequently, generate files that can be assembled (system information table, system call table) by using the utility tool (configurator cf850.exe) provided by the RX850 Pro.

Remark See **4.2.1 Information file generation procedure** for details of the system information table and system call table generation method.

<2> Object file generation

Generate a relocatable object file by executing the C compiler/assembler for the various processing programs (C language/assembly language format files) shown below.

- Processing programs depending on the RX850 Pro
  - System information table
  - System call table
  - Entry processing
  - Boot processing
  - Hardware initialization block
  - Initialization handler
- Blocks depending on the USB storage class driver processing

<3> Load module generation

Generate ELF-format load modules by executing the link editor for the relocatable object file generated in step <2>, various library files, and section map file.

libc.a	Standard CA850 library
rxtmcore.o	Nucleus common-section object
librxp.a	Nucleus library
libchp.a	Interface library

rxtmcore.o, librxp.a, and libchp.a are provided by RX850 Pro and libc.a by CA850.

CHAPTER 7 USB STORAGE CLASS DRIVER FUNCTIONS

7.1 Overview

The USB storage class driver requires tasks for executing the USB storage class driver processing, interrupt handlers, and descriptions of the USB function controller initialization processing.

The blocks depending on the USB storage class driver processing are shown below.

- USB function controller initialization processing

  This processing is called from the RX850 Pro software initialization block and performs initialization processing of the USB function controller.

- USB function controller interrupt handler

  This is a routine dedicated to interrupt servicing, which is called every time an interrupt of the USB function controller is generated and is defined in the CF definition file.

Caution In this sample program, interrupts other than those needed are masked.
The following four interrupts are used in this sample program.

CPUDEC interrupt reported by the INTUSBF0 signal
(This indicates that a request for decoding with FW is made to the UF0E0ST register.)
BKO1DT interrupt reported by the INTUSBF0 signal
(This indicates that data was normally received by the UF0BO1 register.)
EP1_ENDINT interrupt reported by the INTUSBF1 signal
(This indicates that the DMA transfer of endpoint 1 has ended.)
EP2_ENDINT interrupt reported by the INTUSBF2 signal
(This indicates that the DMA transfer of endpoint 2 has ended.)

- USB function controller interrupt servicing task

  This task is called from the USB function controller interrupt handler and performs processing (register setting, data transmission/reception processing, etc.) for each interrupt source.

- USB function controller general-purpose functions

  Functions such as for setting a STALL response for each endpoint and performing transmission and reception processing are provided as general-purpose functions used by the USB storage class driver.

Remark See the sample usbf850.c file for details of the coding method of the blocks depending on the USB storage class driver processing.

- DMA control block

  This block performs DMA initialization and DMA start processing.

  In the sample program, DMA transfer is used when the bulk endpoint data exceeds the MaxPacket size (40 bytes).

Remark See the sample usbf850_dma.c file for details of the DMA control block coding method.

- Bulk-Only Transport processing block

This block performs request processing, CBW processing, and CSW transmission processing, which are unique to the device class for the USB storage class.

Caution The following two requests unique to the device class are acknowledged in this sample program. See Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0 for details of each request.

Bulk-Only Mass Storage Reset request
Get Max LUN request

Remark See the sample usbf850_storage.c file for details of the Bulk-Only Transport processing block coding method.

- Storage device processing block

This block initializes the storage device and processes SCSI commands.

Caution 1. This sample program operates as a Mass Storage device (interface class: mass storage, interface subclass: SCSI, interface protocol: Bulk-Only Transport protocol). The storage device used in this manual has no logical unit connected and operates such that a removable disk is virtually connected by securing a memory area (block size: 512 bytes, number of logic blocks: 192, capacity: 96 KB). The memory area for the virtual device is secured as an array with the name storage_data.
See the sample ata.h file for details.
2. When changing the virtual device and controlling the real device, the data and data processing used in the sample program must be changed according to the environment.

Remark See the sample ata_ctrl.c and scsi_cmd.c files for the storage device processing block coding method.

- USB suspend/resume processing

The USB suspend/resume processing is not supported by this sample program, because it depends on the system. If the processing is required by the system, add the processing by noting the following.

Suspending/resuming is reported by an interrupt (INTUSBF0 signal) with the USB function controller built into the PFESiP/V850EP1. Consequently, if the UF0IS0.RSUSPD bit is checked whether it is set (1) in the interrupt handler (for the INTUSBF0 signal), a suspend state or resume state can be identified by checking the UF0EPS1.RSUM bit.

As an example, the processing is added by adding the above-mentioned state identification code to the interrupt handler (for the INTUSBF0 signal) from which the task performing the required processing is set to be awakened.

7.2 Processing Flow

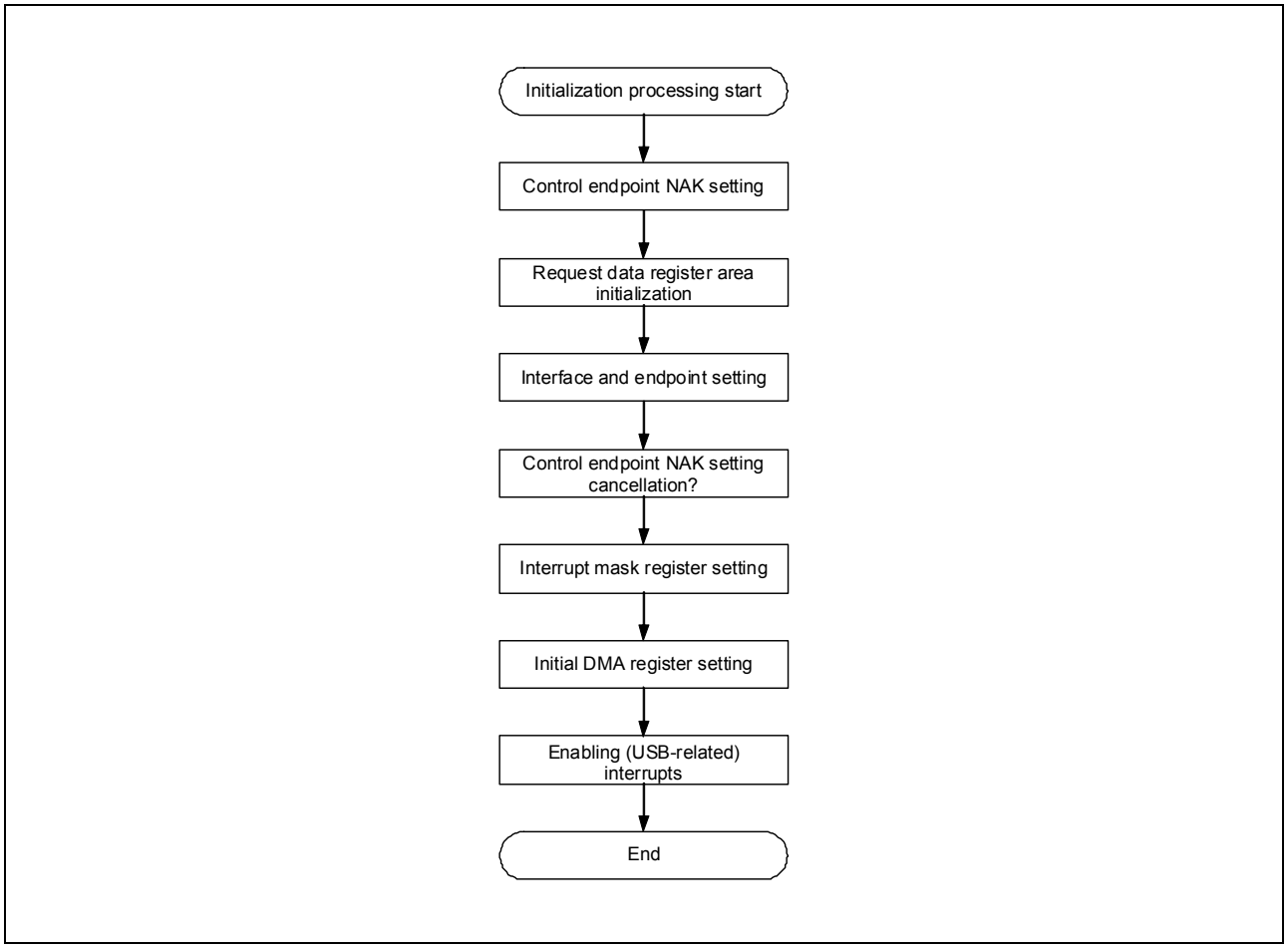
The processing flow of the initialization processing, interrupt servicing, and CBW data processing of the sample program is shown below.

7.2.1 Initializing Process

Initialization of the USB device is called from the software initialization block and executed.

The flow of the USB device initialization processing (when turning the power on) with the sample program is shown below.

Figure 7-1. Initialization Processing Flow Chart



The processing to be executed in the initialization processing is described below.

Caution The initialization processing is required except for port processing. The pin assignment may vary if the target board differs, so read the following according to the target board specifications.

- Control endpoint NAK setting

Perform a NAK response for all requests, including automatically executed requests.

Set so that the hardware returns no unintended data to automatically executed requests, until the data used with automatically executed requests is registered.

- Request data register area initialization

Register the descriptor data for responding to the Get Descriptor request to registers.

The data to be registered are the device status, the endpoint 0 status, Device Descriptor, Configuration Descriptor, Interface Descriptor, and Endpoint Descriptor.

Caution The descriptor for the class may have to be registered, depending on the class.
For the USB storage class, no descriptors other than the standard USB descriptor are used.

- Interface and endpoint setting

  Set information, such as the number of interfaces to be supported, alternative setting status, relationships between interfaces and endpoints, to registers.

- Control endpoint NAK setting cancellation

  Cancel the NAK setting of the control endpoint (endpoint number 0) when registering the data for automatically executed requests is completed.

- Interrupt mask register setting

  Set masking of each interrupt source indicated by the interrupt status register of the USB function controller.

- Initial DMA register setting

  Call the DMA initialization processing and perform initialization for each endpoint for which DMA is used.

- Enabling interrupts

  Enable the INTUSBF0, INTUSBF1, and INTUSBF2 interrupts.

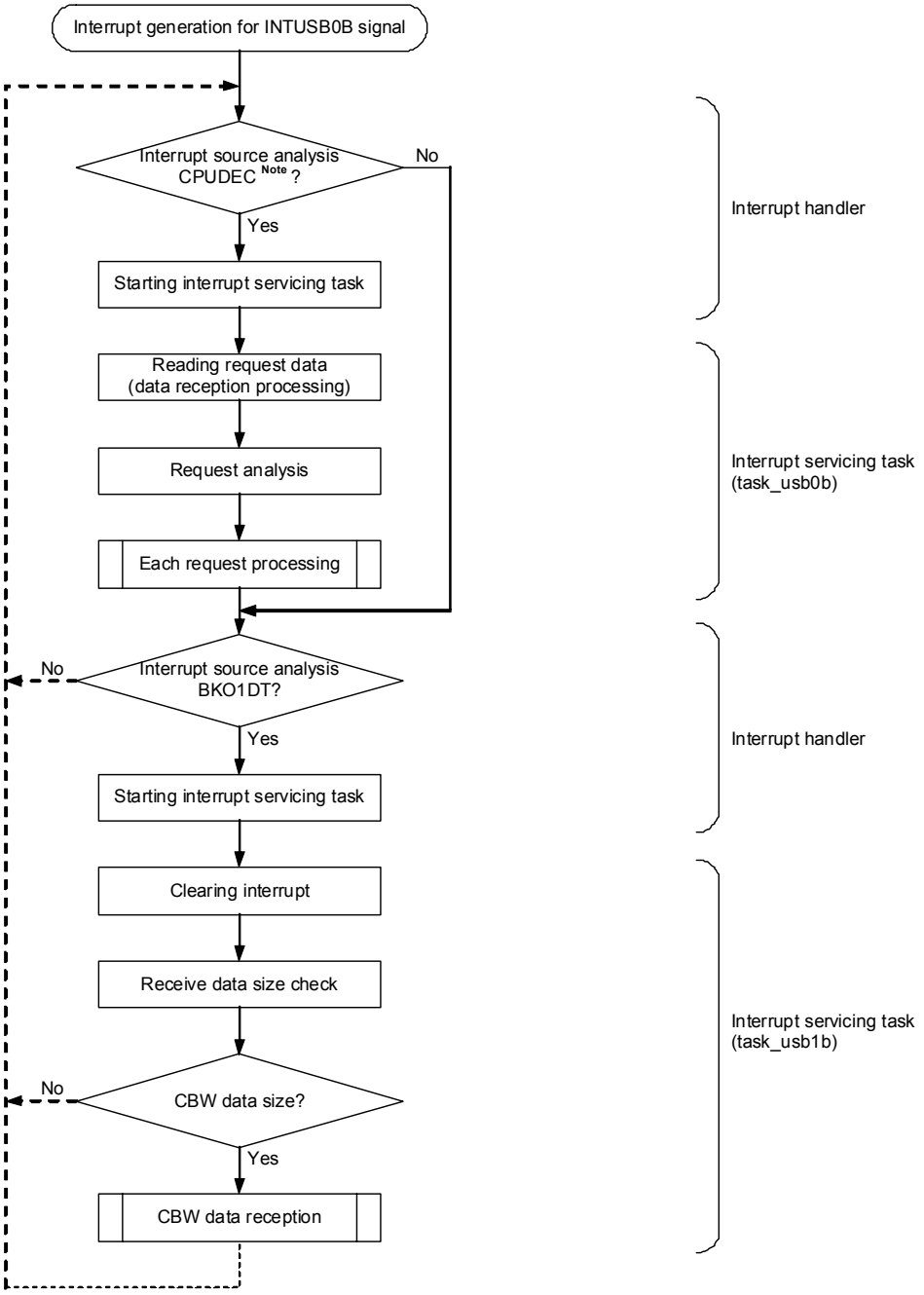
7.2.2 Interrupt servicing

With the sample program, the USB function controller operates by an interrupt event after initialization is completed. If no event is present, the USB function controller is always in the idle state. Furthermore, the USB function controller never generates an event from the storage device and operates by an event from any host driver.

Figures 7-2, 7-3, and 7-4 show the interrupt servicing flow with the sample program.

Caution The flow chart shown in Figure 7-2 shows the flow of the interrupt servicing reported by the INTUSBF0 signal of the USB function controller.
The flow chart shown in Figure 7-3 shows the flow of the interrupt servicing reported by the INTUSBF1 signal of the USB function controller.
The flow chart shown in Figure 7-4 shows the flow of the interrupt servicing reported by the INTUSBF2 signal of the USB function controller.

Figure 7-2. Interrupt Servicing Flow Chart (1)



Note The CPUDEC and BKO1DT interrupts are acknowledged with the INTUSBF0 signal of the USB storage class driver.

The processing of the sample program with the interrupt by the INTUSBF0 signal is described below.

[Processing in interrupt handler]

- Interrupt source analysis

With the sample program, the interrupt status to be analyzed differs depending on the interrupt handler to be executed.

The CPUDEC and BKO1DT interrupts are supported as the interrupts reported by the INTUSBF0 signal. If these interrupts are generated, the interrupt handler will be started by the INTUSBF0 signal. The UF0IS1 register is read and whether the interrupt source is the CPUDEC interrupt is checked in this interrupt handler. Furthermore, the UF0IS3 register is read and whether the interrupt source is the BKO1DT interrupt is checked.

Caution In this sample program, the interrupt handler to be used is pre-registered by the CF definition file.

- Starting the interrupt servicing task

If the interrupt source is CPUDEC, the task_usb0b task will be started.

Caution In this sample program, the task to be started is pre-registered by the CF definition file.

- Starting the interrupt servicing task

If the interrupt source is BKO1DT, the task_usb1b task will be started.

Caution In this sample program, the task to be started is pre-registered by the CF definition file.

[Processing with task_usb0b task]

- Reading the request data

The SETUP data is read from the UF0E0ST register.

- Request analysis

he read SETUP data is analyzed and the request content is checked.

- Each request processing

A processing corresponding to the analyzed request content is performed.

With the sample program, the Get Descriptor (String Descriptor) request of the standard device requests and the requests unique to the device class are processed.

[Processing with task_usb1b task]

- Interrupt source analysis

Only the BKO1DT interrupt is supported as an interrupt reported by the INTUSBF1 signal. If an interrupt is generated, the interrupt handler will be started by the INTUSBF1 signal.

The interrupt source is not checked in the interrupt handler, but whether the interrupt source is BKO1DT is checked with the started task.

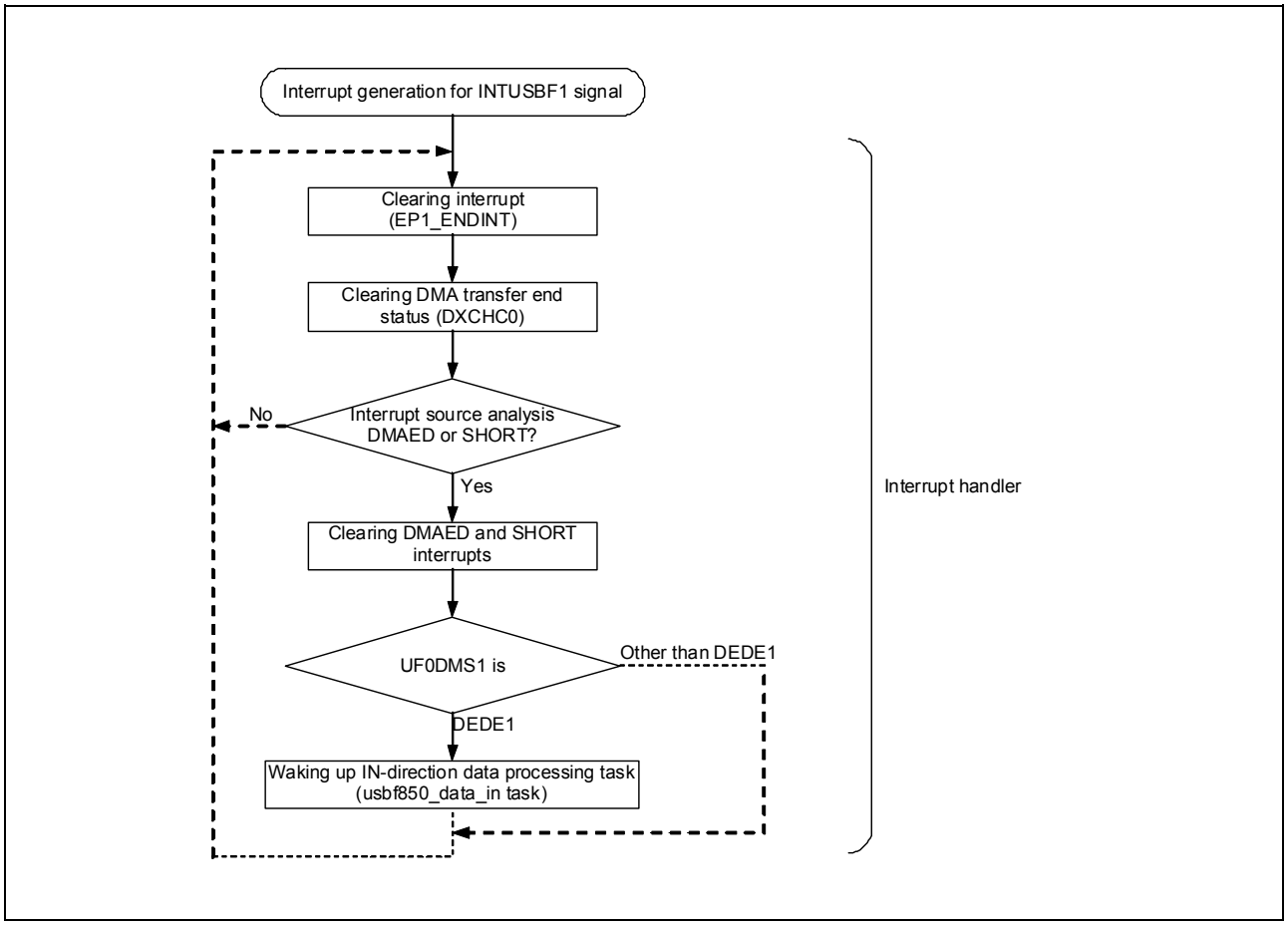
Caution In this sample program, the interrupt handler to be used is pre-registered by the CF definition file.

- Receive data size check

If the interrupt source is BKO1DT, the UF0BO1L register will be read and whether the receive data length is equal to the CBW data length will be checked. If the receive data length is equal to the CBW data length, the usbf850_rx_cbw function will be called and the CBW processing will be started.

Remark See **7.2.3 CBW data processing** for the CBW processing.

Figure 7-3. Interrupt Servicing Flow Chart (2)



The processing of the sample program with the interrupt by the INTUSBF1 signal is described below.

- Clearing DMA transfer end interrupt source of endpoint 1

  The interrupt source of the USB function Bridge is cleared.

- Clearing DMA transfer end status

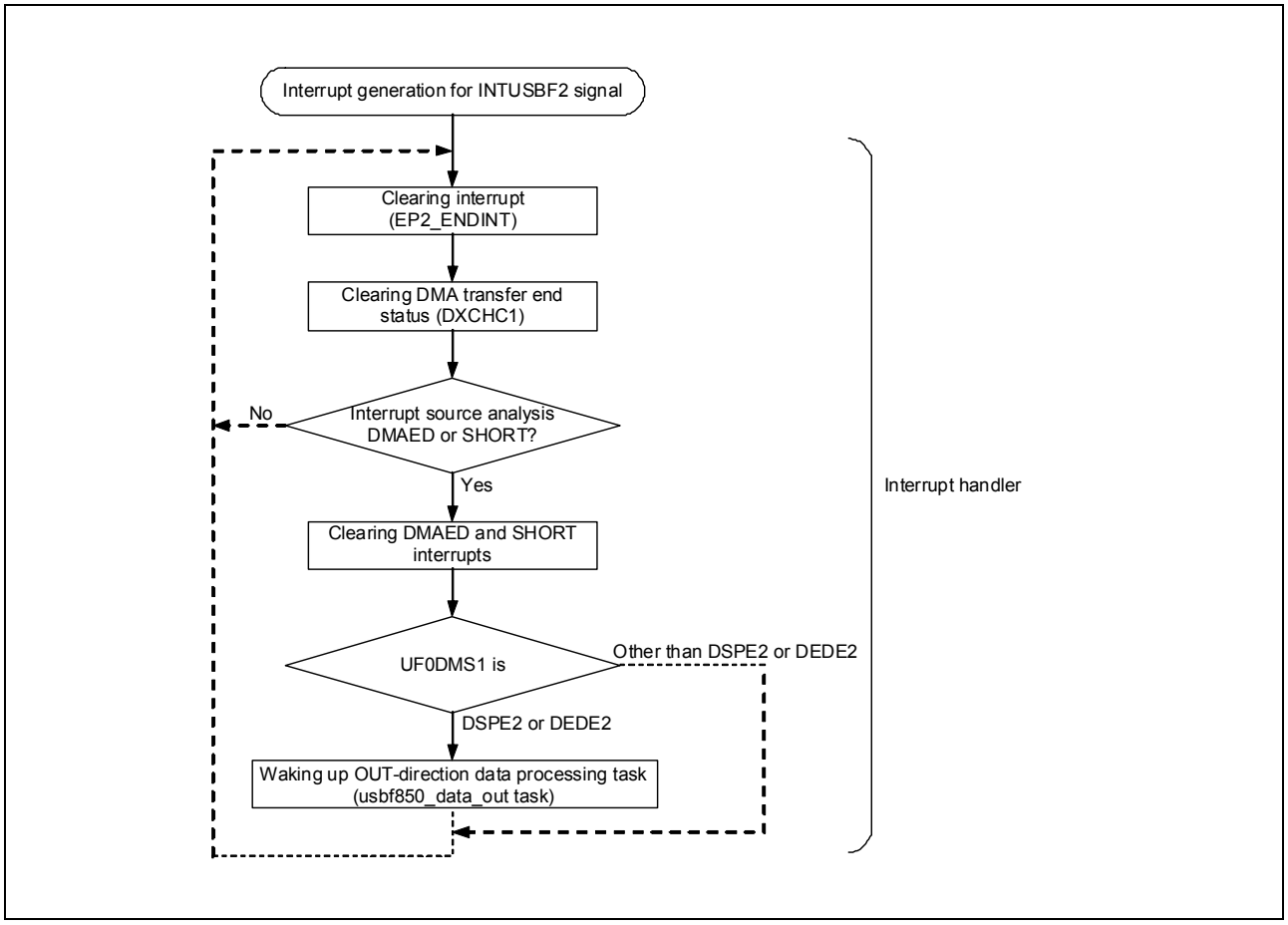
  The DMA transfer end status of the DMA controller is cleared.

- Waking up the usbf850_data_in task

  The usbf850_no_data task, usbf850_data_in task, and usbf850_data_out task perform SCSI command processing. These tasks are started by receiving a normal CBW with the BKO1DT interrupt. Among these tasks, usbf850_data_in task and usbf850_data_out task sleep after DMA is started. With the interrupt handler for the INTUSBF1 signal, the UF0DMS1 register will be read and whether the interrupt source is DEDE1 will be checked if the interrupt source is DMAED or SHORT.

  If the interrupt source is DEDE1, the usbf850_data_in task will be awakened.

Figure 7-4. Interrupt Servicing Flow Chart (3)



The processing of the sample program with the interrupt by the INTUSBF2 signal is shown below.

- Clearing DMA transfer end interrupt source of endpoint 2

  The interrupt source of the USB function Bridge is cleared.

- Clearing DMA transfer end status

  The DMA transfer end status of the DMA controller is cleared.

- Waking up the usbf850_data_out task

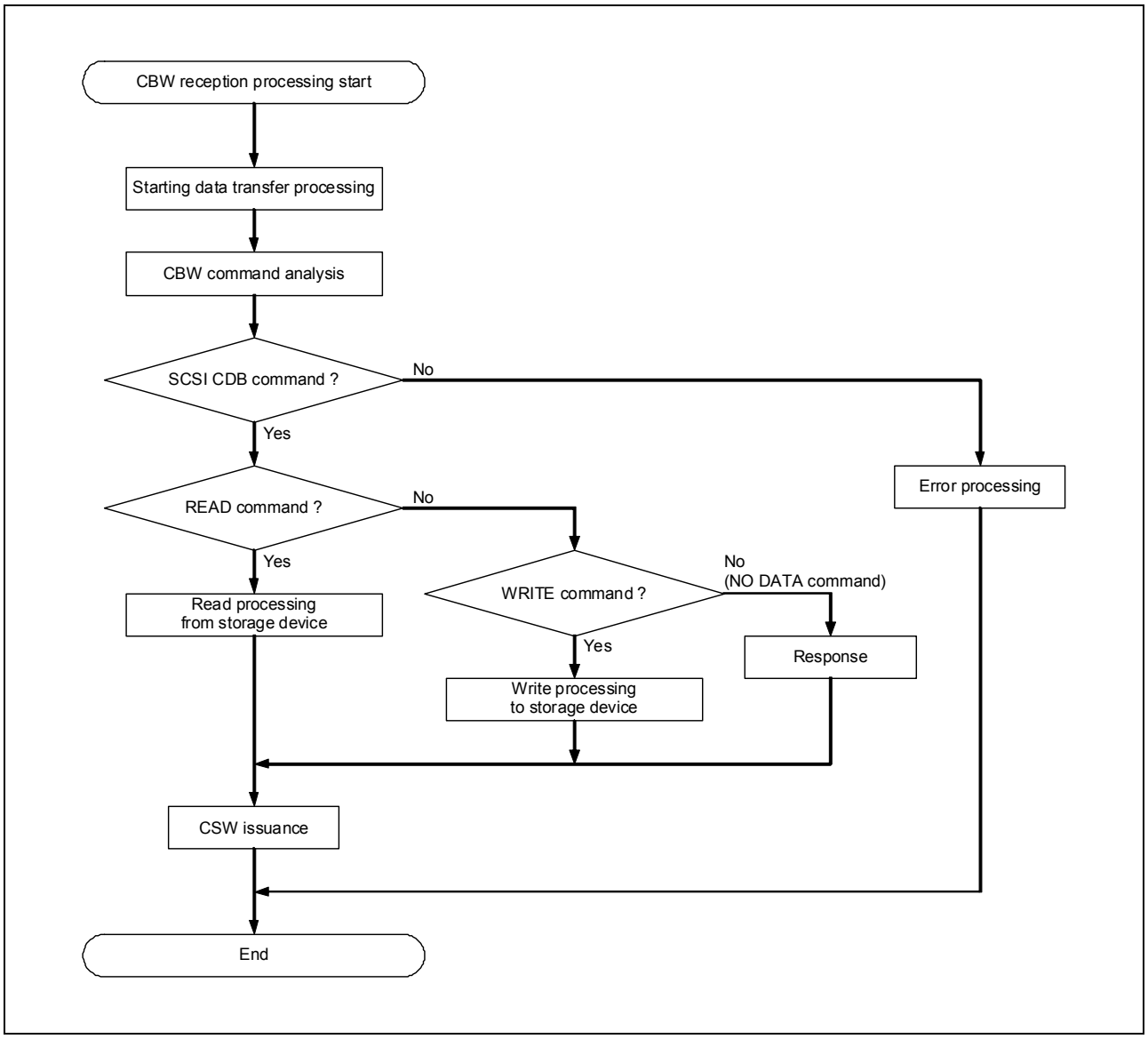
  The usbf850_no_data task, usbf850_data_in task, and usbf850_data_out task perform SCSI command processing. These tasks are started by receiving a normal CBW with the BKO1DT interrupt. Among these tasks, usbf850_data_in task and usbf850_data_out task sleep after DMA is started. With the interrupt handler for the INTUSBF2 signal, the UF0DMS1 register will be read and whether the interrupt source is DSPE2 or DEDE2 will be checked if the interrupt source is DMAED or SHORT.

  If the interrupt source is DSPE2 or DEDE2, the usbf850_data_out task will be awakened.

7.2.3 CBW data processing

The CBW data processing is started when CBW data is received via the USB. The CBW data processing flow is shown below. The CBW data format is shown in Table 7-1 and the CSW data format in Table 7-2.

Figure 7-5. CBW Data Processing Flow Chart



The processing of the CBW data with the sample program is described below.

- CBW command analysis

  After the CBW data is received, the contents of the CBW are analyzed.

  In this analysis, the CBW tag is saved, the valid number of data of the CBWCB and the command direction are checked, and each processing task of the READ, WRITE, and NO DATA commands are started.

- READ command processing

  The task (usbf850_data_in) processing the READ commands of the SCSI commands is started.

  This task calls the SCSI command processing block, identifies the CSW transmission status from the execution result, and calls the CSW issue processing.

- WRITE command processing

  The task (usbf850_data_out) processing the WRITE commands of the SCSI commands is started.

This task calls the SCSI command processing block, identifies the CSW transmission status from the execution result, and calls the CSW issue processing.

- NO DATA command processing

The task (usbf850_no_data) processing the NO DATA commands of the SCSI commands is started.

This task calls the SCSI command processing block, identifies the CSW transmission status from the execution result, and calls the CSW issue processing.

- CSW issuance

The command execution result is called as an argument from each command processing task.

CSW data is generated from the arguments and transmission processing is performed.

[CBW format]

A CBW consists of the following 31 bytes of data.

The processing from the host can be identified based on the CBWCB value.

Table 7-1. CBW Data Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	dCBWSignature (55H)							
1	dCBWSignature (53H)							
2	dCBWSignature (42H)							
3	dCBWSignature (43H)							
4 to 7	dCBWTag (CBW tag to be processed)							
8 to 11	dCBWDataTransferLength (transfer data length)							
12	bmCBWFlag (Data-OUT/IN specification)							
13	Reserved				bCBWLUN (target device number)			
14	Reserved			bCBWCBLength (valid number of bytes of CBWCB)				
15 to 30	CBWCB (command)							

[CBW format]

A CSW consists of the following 13 bytes of data.

Table 7-2. CSW Data Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	dCSWSignature (53H)							
1	dCSWSignature (42H)							
2	dCSWSignature (53H)							
3	dCSWSignature (55H)							
4 to 7	dCSWTag (CBW tag to be processed)							
8 to 11	dCSWDataResidue (difference between CBW specification transfer data length and processed data length)							
12	bmCSWStatus (CBW processing result status)							

7.2.4 SCSI command processing

The processing of the SCSI commands is started after the CBW data is received via the USB.

The sample program supports the 19 types of SCSI CDB commands shown in Table 7-3.

Figure 7-6 shows the SCSI command (READ command) processing flow.

Caution The commands provided with the sample program are only the minimum of commands required for operating the sample program. Add commands not provided with the sample program, and processing required for generating response data and regarding the processing method of response data according to the user environment.

Table 7-3. SCSI Commands

Command	Code	Operation
READ commands		
REQUEST SENSE	03H	Transfers SENSE data to the host.
READ (6)	08H	Transfers logic data block data in the specified range to the host.
INQUIRY	12H	Reports the configuration information and attribute of the target and logical unit to the host.
MODE SENSE (6)	1AH	Reads the mode select parameter value and attribute of the logical unit.
READ FORMAT CAPACITIES	23H	Reports the capacity (number of blocks, block length) of the logical unit to the host.
READ CAPACITY	25H	Reports the data capacity in the logical unit to the host.
READ (10)	28H	Same as READ (6)
MODE SENSE (10)	5AH	Same as MODE SENSE (6)
WRITE commands		
WRITE (6)	0AH	Writes data from the host to a specified block in a medium.
MODE SELECT (6)	15H	Sets and changes various parameters of such as the data format of the logical unit.
WRITE (10)	2AH	Same as WRITE (6)
WRITE VERIFY	2EH	Reads the data written to a medium and checks the normality of the data.
VERIFY	2FH	Checks the normality of the data in a medium in the drive unit.
WRITE_ BUFF	3BH	Writes any data to the target memory.
MODE_SELECT (10)	55H	Same as MODE SELECT (6)
NO DATA commands		
TEST UNIT READY	00H	Reports the state of the logical unit to the initiator (host device).
SEEK	0BH	Performs a seek operation to a position in the specified storage medium.
START STOP UNIT	1BH	Enables or disables accessing the medium of the logical unit.
SYNCHRONIZE CACHE	35H	Matches the values of the cache memory and medium for a block in the specified range.

Figure 7-6. READ Command Processing Flow Chart (1/2)

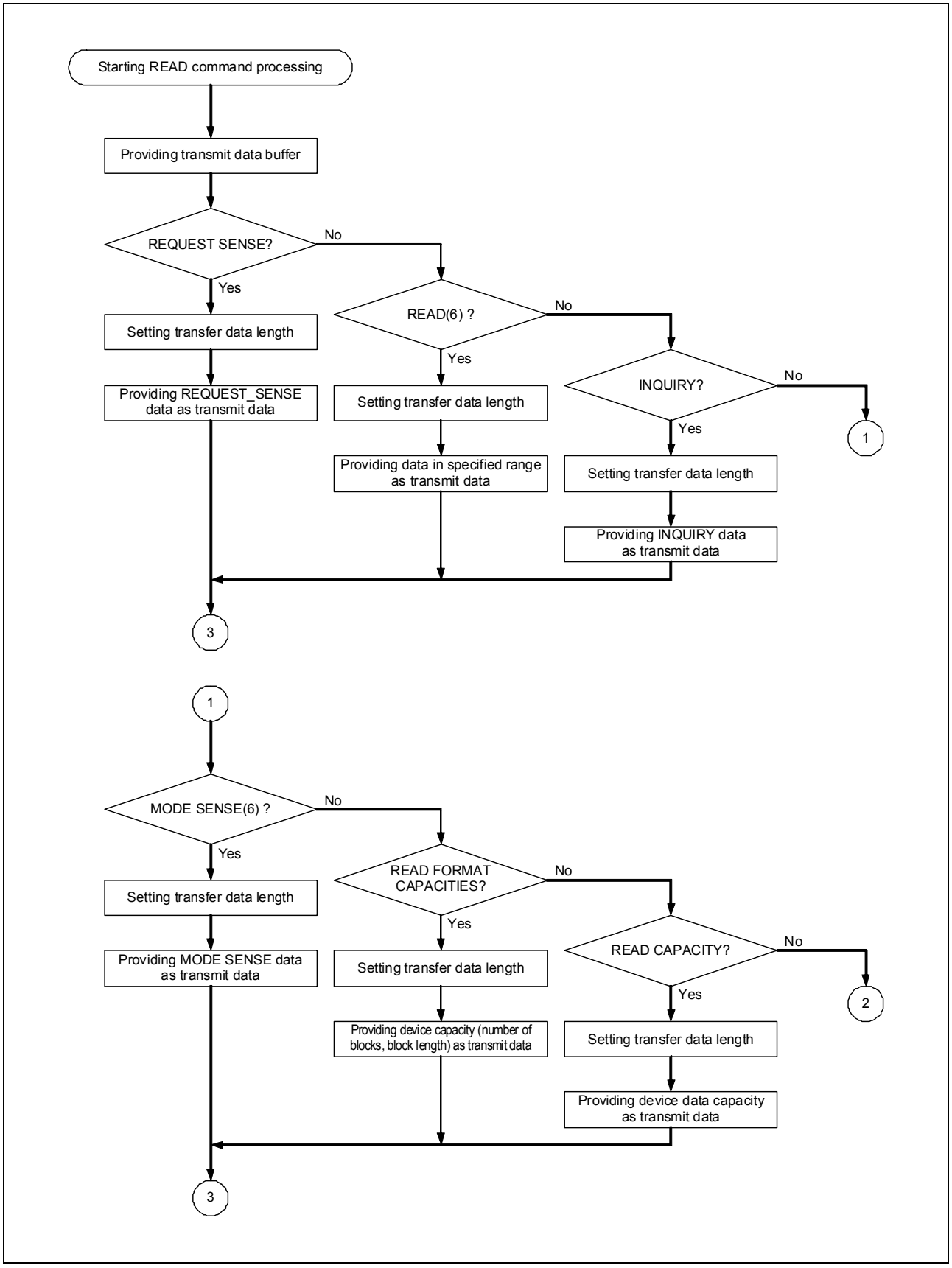
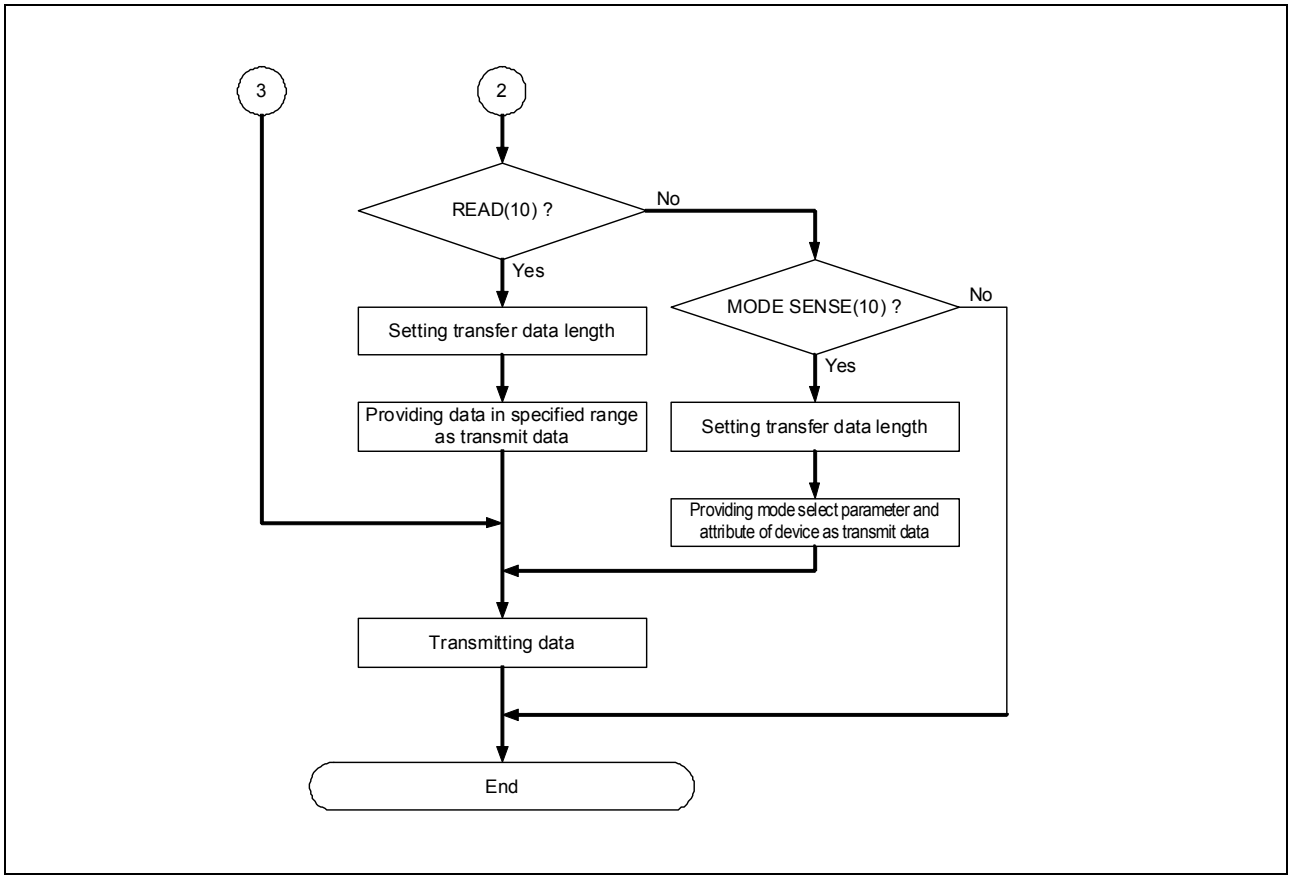


Figure 7-6. READ Command Processing Flow Chart (2/2)



[REQUEST SENSE command processing]

The REQUEST SENSE command reports sense data to the host.

The sense data format is shown below. Furthermore, the sense data used in the sample program are shown as data values. Provided data is returned as the following data values, because a virtual device is used with the sample program.

Table 7-4. Sense Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	VALID	Error code							70H
1	Reserved								00H
2	Reserved		ILI	Reserved	Sense key				00H
3	Information								00H
4	Information								00H
5	Information								00H
6	Information								00H
7	Additional sense data length (n – 7)								0AH
8	Command-specific information								00H
9	Command-specific information								00H
10	Command-specific information								00H
11	Command-specific information								00H
12	Additional sense code (ASC)								00H
13	Additional sense code qualifier (ASCQ)								00H
14	FRU (Field Replaceable Unit) code								00H
15	SKSV	Sense-key-specific information							00H
16	Sense-key-specific information								00H
17	Sense-key-specific information								00H

The sense keys transmitted to the host in the sample driver are shown below.

Table 7-5. Sense Keys

Sense Key	ASC	ASCQ	Description of Error
00	00	00	NO SENSE
05	00	00	ILLEGAL REQUEST.
05	20	00	INVALID COMMAND OPERATION CODE
05	24	00	INVALID FIELD IN COMMAND PACKET

[READ (6) command processing]

The READ (6) command reads the data in the specified range from the storage device and transmits the read data to the host.

With the sample program, the data read from the virtual device is transmitted to the host.

[INQUIRY command processing]

The INQUIRY command reports information of the device to the host.

The INQUIRY data format is shown below. Provided data is returned as the following data values, because a virtual device is used with the sample program.

Table 7-6. INQUIRY Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Qualifier			Device type code					00H
1	RMB	Device type modifier							80H
2	Version		ECMA version			00H			00H
3	AENC	TrmIOP	01H			01H			02H
4	Additional data length (n – 4 bytes)								1FH
5	Reserved								00H
6	Reserved								00H
7	Reserved								00H
8	Vendor ID (ASCII)								Note 1
:	:								Note 1
15	Vendor ID (ASCII)								Note 1
16	Product ID (ASCII)								Note 2
:	:								Note 2
31	Product ID (ASCII)								Note 2
32	Product edition (ASCII)								Note 3
:	:								Note 3
35	Product edition (ASCII)								Note 3
36	Unique to vendor								none
:	:								none
55	Unique to vendor								none
56	Reserved								none
:	:								none
95	Reserved								none
96	Unique to vendor								none
:	:								none
n	Unique to vendor								none

Notes 1. NEC Corp ASCII character code

2. StorageFncDriver ASCII character code

3. 0.12 ASCII character code

[MODE SENSE (6) command processing]

The MODE SENSE (6) command reports the mode select parameter and attribute of the device to the host.

The MODE SENSE data format is shown below. Provided data is returned as the following data values, because a virtual device is used with the sample program. 01H is the only supported page code. This data is returned regardless of the command page code.

Table 7-7. MODE SENSE Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Mode parameter length								Note 1
1	Media type								00H
2	Device-specific parameter								00H
3	Block descriptor length								08H
4	Density code								00H
5	00H								00H
6	00H								00H
7	C0H								C0H
8	Reserved								00H
9	00H								00H
10	00H								02H
11	00H								00H
12	PS	Reserved	Page code						Note 2
13	Page length (n-13)								0AH
14	Mode parameter								Note 3
:	:								Note 3
n	Mode parameter								Note 3

Notes 1. The number of bytes of the parameter list specified by the DBD of the CDB and page code, or a parameter list for the data specified by the allocation length, whichever is less

2. CDB page code

3. 08H, 0BH, 00H, 00H, 00H, 00H, 00H, 00H, 00H, 00H

[READ FORMAT CAPACITY command processing]

The READ FORMAT CAPACITY command reports the device capacity (number of blocks, block length) to the host. The READ FORMAT CAPACITY data format is shown below. Provided data is returned as the following data values, because a virtual device is used with the sample program.

Table 7-8. READ FORMAT CAPACITY Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Reserved								00H
1	Reserved								00H
2	Reserved								00H
3	Capacity list length (bytes)								08H
4	Number of blocks								00H
5	Number of blocks								00H
6	Number of blocks								00H
7	Number of blocks								C0H
8	Reserved						Descriptor Code		01H
9	Block length								00H
10	Block length								02H
11	Block length								00H
12	Number of blocks								00H
13	Number of blocks								00H
14	Number of blocks								00H
15	Number of blocks								C0H
16	Reserved								00H
17	Block length								00H
18	Block length								02H
19	Block length								00H

[READ CAPACITY command processing]

The READ CAPACITY command reports the device data capacity to the host. The READ CAPACITY data format is shown below. Provided data is returned as the following data values, because a virtual device is used with the sample program.

Table 7-9. READ CAPACITY Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Logical block address								00H
1	Logical block address								00H
2	Logical block address								00H
3	Logical block address								BFH
4	Block length								00H
5	Block length								00H
6	Block length								02H
7	Block length								00H

[READ (10) command processing]

The READ (10) command reads the data of the specified range from the storage device and transmits the read data to the host.

With the sample program, the data read from the virtual device is transmitted to the host.

[MODE SENSE (10) command processing]

The MODE SENSE (10) command reports the mode select parameter and attribute of the device to the host.

The MODE SENSE (10) data format is shown below. Provided data is returned as the following data values, because a virtual device is used with the sample program. 01H is the only supported page code. This data is returned regardless of the command page code.

Table 7-10. MODE SENSE (10) Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Mode parameter length								Note 1
1	Mode parameter length								Note 1
2	Media type								00H
3	Device-specific parameter								00H
4	Reserved								00H
5	Reserved								00H
6	Block descriptor length								00H
7	Block descriptor length								08H
8	Density code								00H
9	Number of blocks								00H
10	Number of blocks								00H
11	Number of blocks								C0H
12	Reserved								00H
13	Block length								00H
14	Block length								02H
15	Block length								00H
16	PS	Reserved	Page code						Note 2
17	Page length (n-17)								0AH
18	Mode parameter								Note 3
:	:								Note 3
N	Mode parameter								Note 3

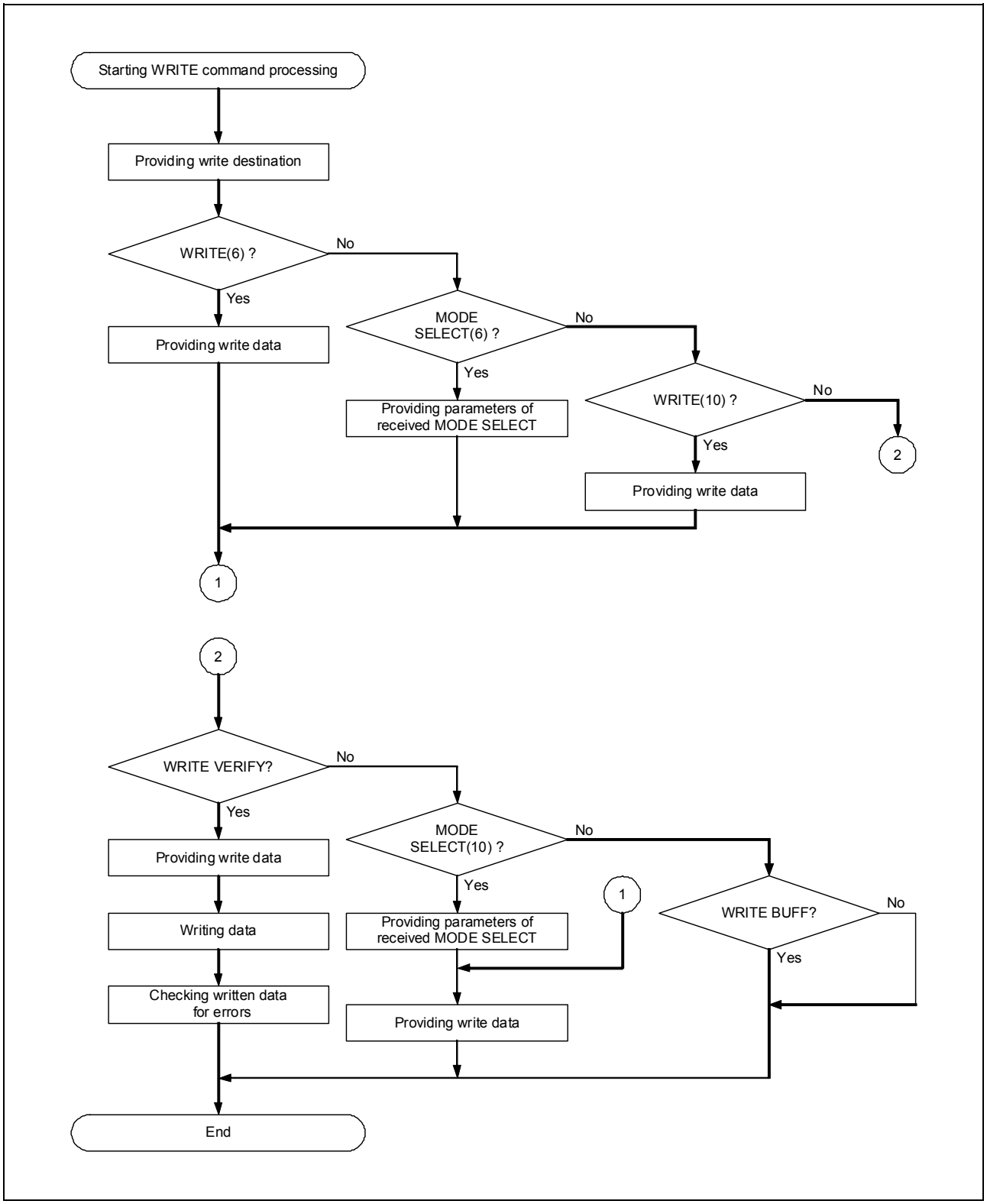
Notes 1. The number of bytes of the parameter list specified by the DBD of the CDB and page code, or a parameter list for the data specified by the allocation length, whichever is less

2. CDB page code

3. 08H, 0BH, 00H, 00H, 00H, 00H, 00H, 00H, 00H, 00H

The SCSI command (WRITE command) processing flow is shown below.

Figure 7-7. WRITE Command Processing Flow Chart



[WRITE (6) command processing]

The WRITE (6) command writes the received data to the specified area of the storage device.

With the sample program, the received data is written to the specified area of the virtual device.

[MODE SELECT (6) processing]

The MODE SELECT (6) command sets and changes various parameters, such as the physical attribute of the logical unit, data format in the storage medium, and error recovery method and procedure.

The MODE SELECT data format is shown below. The entire processing ends normally by only writing the received data to the MODE SELECT TABLE, regardless of the command page code, because a virtual device is used with the sample program. The data values are assumed to be used as the initial table values.

Table 7-11. MODE SELECT (6) Data Format

Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Mode parameter length								17H
1	Media type								00H
2	Device-specific parameter								00H
3	Block descriptor length								08H
4	Density code								00H
5	00H								00H
6	00H								00H
7	C0H								C0H
8	Reserved								00H
9	00H								00H
10	00H								02H
11	00H								00H
12	PS	1	Page code						Note 1
13	Page length (n-13)								0AH
14	Mode parameter								Note 2
:	:								Note 2
n	Mode parameter								Note 2

Notes 1. CDB page code

2. 08H, 0BH, 00H, 00H, 00H, 00H, 00H, 00H, 00H, 00H

[WRITE (10) command processing]

The WRITE (10) command writes the received data to the specified area of the storage device.

With the sample program, the received data is written to the specified area of the virtual device.

[WRITE VERIFY command processing]

The WRITE VERIFY command writes the received data to the storage device and checks the written data for errors. With the sample program, the processing ends normally after writing the received data to the virtual device, without checking the written data for errors.

[VERIFY command processing]

The VERIFY command checks the normality of the data in the storage device. The processing ends normally without performing any processing, because a virtual device is used with the sample program.

[WRITE BUFF command processing]

The WRITE BUFF command writes data to a memory (data buffer). The processing ends normally without performing any processing, because a virtual device is used with the sample program.

[MODE SELECT (10) command processing]

The MODE SELECT (10) command sets and changes various parameters, such as the physical attribute of the logical unit, data format in the storage medium, and error recovery method and procedure.

The MODE SELECT (10) data format is shown below. The entire processing ends normally by only writing the received data to the MODE SELECT (10) TABLE, regardless of the command page code, because a virtual device is used with the sample program. The data values are assumed to be used as the initial table values.

Table 7-12. MODE SELECT (10) Data Format

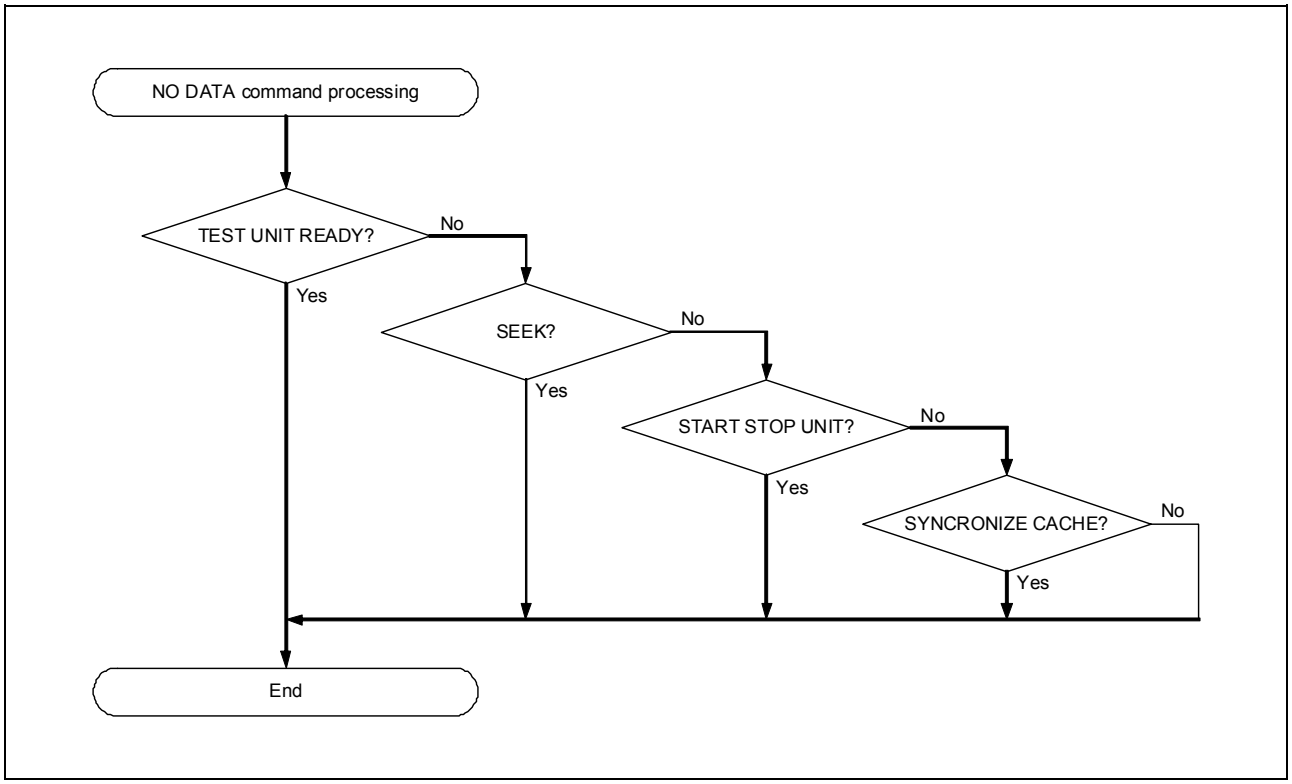
Byte \ Bit	7	6	5	4	3	2	1	0	Data Value
0	Mode parameter length								00H
1	Mode parameter length								1AH
2	Media type								00H
3	Device-specific parameter								00H
4	Reserved								00H
5	Reserved								00H
6	Block descriptor length								00H
7	Block descriptor length								08H
8	Density code								00H
9	Number of blocks								00H
10	Number of blocks								00H
11	Number of blocks								C0H
12	Reserved								00H
13	Block length								00H
14	Block length								02H
15	Block length								00H
16	PS	Reserved	Page code						Note 1
17	Page length (n-17)								0AH
18	Mode parameter								Note 2
:	:								Note 2
n	Mode parameter								Note 2

Notes 1. CDB page code

2. 08H, 0BH, 00H, 00H, 00H, 00H, 00H, 00H, 00H, 00H

The SCSI command (NO DATA command) processing flow is shown below.

Figure 7-8. NO DATA Command Processing Flow Chart



[TEST UNIT READY command processing]

The TEST UNIT READY command reports the state of the unit. The processing ends normally without performing any processing, because a virtual device is used with the sample program.

[SEEK command processing]

The SEEK command performs a seek operation to the specified block position. The processing ends normally without performing any processing, because a virtual device is used with the sample program.

[START STOP UNIT command processing]

The START STOP UNIT command sets restrictions for accessing the unit. The processing ends normally without performing any processing, because a virtual device is used with the sample program.

[SYNCHRONIZE CACHE command processing]

The SYNCHRONIZE CACHE command matches the data of the unit and CACHE. The processing ends normally without performing any processing, because a virtual device is used with the sample program.

7.3 USB Storage Class Driver Descriptor Information

The standard USB descriptors defined by the sample program are described below.
Be sure to use the descriptors of (a) through (d).

Remark See **Universal Serial Bus Specification Revision 1.1** for details.

(a) Device Descriptor

The Device Descriptor holds general information of a device. Provide a Device Descriptor for each device. The information is used to recognize a unique device in the device configuration. Concrete information of the current class at the device level is not used for the USB storage class.

Table 7-13. Device Descriptor

Offset	Size (Byte)	Value	Description
0	1	12H	Length value of this descriptor (bytes)
1	1	01H	Descriptor type (device)
2	2	10H/01H	USB version (USB1.1)
4	1	00H	Class code
5	1	00H	Subclass code
6	1	00H	Protocol code
7	1	40H	Maximum packet size of endpoint 0
8	2	09H/04H	Vender ID (NEC Electronics)
10	2	FCH/FFH	Product ID
12	2	01H/00H	Device release number
14	1	01H	Index to the string descriptor (manufacturer)
15	1	00H	Index to the string descriptor (Product)
16	1	00H	Index to the string descriptor (Serial Number)
17	1	01H	Number of possible configurations

(b) Configuration Descriptor

The Configuration Descriptor holds concrete device configuration information.

Concrete information of the current class at the configuration level is not used for the USB storage class.

Table 7-14. Configuration Descriptor

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (bytes)
1	1	02H	Descriptor type (configuration)
2	2	20H/00H	Total length value of the descriptor returned together with the configuration descriptor by the Get Descriptor request
4	1	01H	Supported number of interfaces with this configuration
5	1	01H	Configuration value
6	1	00H	Index to the string descriptor (configuration)
7	1	C0H	Device configuration (self-powered/remote wakeup function)
8	1	00H	Maximum power consumption of the device

(c) Interface Descriptor

The Interface Descriptor holds concrete interface information in the configuration.

With the sample program, the configuration provides one interface. Furthermore, this interface supports two endpoints and has two Endpoint Descriptors.

The Interface Descriptor is always returned as a part of the Configuration Descriptor and not directly accessed by the Get Descriptor or Set Descriptor request.

Table 7-15. Interface Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (bytes)
1	1	04H	Descriptor type (interface)
2	1	00H	Interface value
3	1	00H	Alternate setting value
4	1	02H	Number of endpoints (excluding endpoint 0)
5	1	08H	Interface class (mass storage class)
6	1	06H	Interface subclass (SCSI)
7	1	50H	Interface protocol (Bulk-Only)
8	1	00H	Index to the string descriptor (interface)

(d) Endpoint Descriptor

The Endpoint Descriptors hold the information required by the host for determining the band width requirements of each endpoint.

The Endpoint Descriptors are always returned as a part of the Configuration Descriptor and not directly accessed by the Get Descriptor or Set Descriptor request.

Table 7-16. Endpoint Descriptor (Bulk IN)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (bytes)
1	1	05H	Descriptor type (endpoint)
2	1	81H	Endpoint address value
3	1	02H	Endpoint transfer type
4	2	40H/00H	Maximum packet size of the endpoint
6	1	00H	Interval (ms): Only isochronous and interrupt endpoints are valid

Table 7-17. Endpoint Descriptor (Bulk OUT)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (bytes)
1	1	05H	Descriptor type (endpoint)
2	1	02H	Endpoint address value
3	1	02H	Endpoint transfer type
4	2	40H/00H	Maximum packet size of the endpoint
6	1	00H	Interval (ms): Only isochronous and interrupt endpoints are valid

(e) String Descriptor

With this sample program, the String Descriptors hold information of the device manufacturer name.

Table 7-18. String Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	04H	Length value of this descriptor (bytes)
1	1	03H	Descriptor type (string)
2	2	09H/04H	Language type used with the string descriptor (English/U.S.)

Table 7-19. String Descriptor (2)

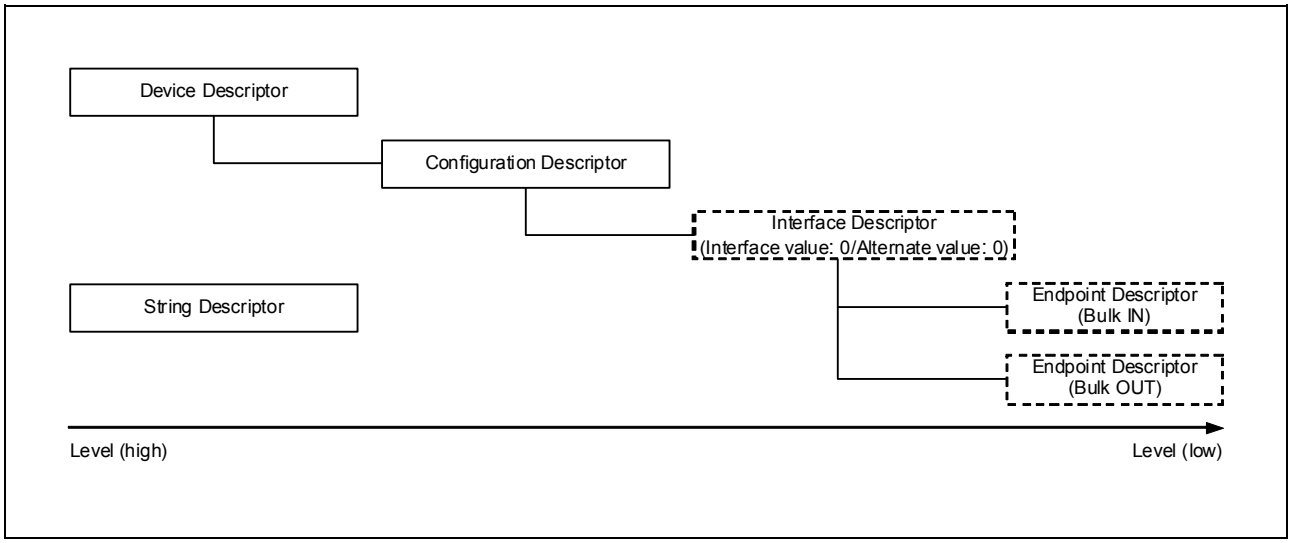
Offset	Size (Byte)	Value	Description
0	1	2AH	Length value of this descriptor (bytes)
1	1	03H	Descriptor type (string)
2	40	'N','E','C','','E','l','e','c','t','r','o','n','i','c','s',' ','C','o','.'	Manufacturer name (manufacturer) NEC Electronics Co.

7.3.1 Descriptor configuration

The configuration of the descriptors in the sample program is shown below.
Provide the above-mentioned descriptors in the following configuration.

Caution The Device Descriptor, Configuration Descriptor, and String Descriptors are accessed by different Get Descriptor requests. The Interface Descriptor and Endpoint Descriptors are accessed as parts of the Configuration Descriptor.

Figure 7-9. Descriptor Configuration



7.4 Data Macro

The various data macros (data type, return value, etc.) used in the USB storage class driver are described below.

7.4.1 Data type

The macros of the data types of the various parameters to be specified when issuing the USB storage class driver function are defined by the header file nectools32¥USB_Storage¥inc¥types.h.

The data types are shown below.

Table 7-20. Data Types

Macro	Type	Meaning
ULONG	unsigned long	Unsigned 32-bit integer
WORD	unsigned long	Unsigned 32-bit integer
HWORD	unsigned short	Unsigned 16-bit integer
BYTE	unsigned char	Unsigned 8-bit integer
(*PFV) ()	void	Processing program start address

7.4.2 Return value

The macros of the return values from the USB storage class driver function are defined by the header file nectools32¥USB_BUS¥inc¥errno.h.

The return values are shown below.

Table 7-21. Return Values

Macro	Numerical Value	Meaning
DEV_OK	0	Normal termination
DEV_ERROR	-1	Abnormal termination
DEV_ERR_NODATA	-2	Transfer direction error with NO DATA command
DEV_ERR_READ	-3	Transfer direction error with READ command
DEV_ERR_WRITE	-4	Transfer direction error with WRITE command
DEV_ERR_VERIFY	-5	Verification error
DEV_ERR_CBWLENGTH	-6	CBW length error
DEV_ERR_CBWCBW	-7	Reception of CBW during CBW processing (error)

7.5 Data Structures

The data structures used by the USB storage class driver are described below.

7.5.1 USB device request structure

The USB device request structure is defined by the USB header file nectools32 ¥ V850USB_Storage ¥ src ¥ USBF¥usbf850.h. The USB device request structure USB_SETUP is shown below.

```
typedef struct {
      unsigned char RequstType;         /*bmRequestType */
      unsigned char Request;            /*bRequest */
      unsigned short Value;             /*wValue */
      unsigned short Index;             /*wIndex */
      unsigned short Length;            /*wLength */
      unsigned char* Data;              /*index to Data */
} USB_SETUP;
```

7.5.2 CBW data structure

The CBW (command block wrapper) data structure handled by the USB storage class driver is defined by the header file nectools32¥USB_Storage¥inc¥types.h. The CBW data structure is shown below.

```
typedef struct {
      unsigned char dCBWSignature[4];             /*CBW signature*/
      unsigned char dCBWTag[4];                   /*CBW tag*/
      unsigned char dCBWDataTransferLength[4];    /*Transfer data length*/
      unsigned char bmCBWFlags;                   /*Data direction (OUT/IN) specification*/
      unsigned char bCBWLUN;                      /*Target device number*/
      unsigned char bCBWCBLength;                 /*Valid number of bytes of CBWCB*/
      unsigned char CBWCB[16];                    /*CBWCB(command)*/
} CBW_INFO,*PCBW_INFO;
```

7.5.3 CSWdata structure

The CSW (command status wrapper) data structure handled by the USB storage class driver is defined by the header file nectools32¥USB_Storage¥inc¥types.h. The CSW data structure is shown below.

```
typedef struct {
      unsigned char dCSWSignature[4];      /*CSW signature */
      unsigned char dCSWTag[4];            /*CSW tag */
      unsigned char dCSWDataResidue[4];    /*Difference between CBW specification transfer data
                                             length and processed data length */
      unsigned char bmCSWStatus;           /*CBW processing result status */
} CSW_INFO,*PCSW_INFO;
```

7.6 Explanation of Functions

7.6.1 Overview

The processing programs explained in this chapter are shown below.

Table 7-22. Sample Program Processing Programs (1/3)

Processing Program Name	Function Name	File Name	Remark
Processing programs depending on RX850 Pro			
CF definition file	-	sys.cf	-
Entry processing	-	entry.s	Assembly language
Boot processing	boot	boot.s	Assembly language
Hardware initialization block	__InitSystemTimer	init.c	C language
Initialization handler	varfunc	varfunc.c	C language
Header file	-	init.h	-
Link directive file	-	usb_storage.dir	-
Board-dependent block processing programs			
Port initialization	port850_reset	port.c	C language
Header file	-	port.h	-
header file			
USB-related register definition file of PFESiP/V850EP1		aurora_usb_reg.h	
Data type declaration	-	types.h	-
Return value declaration	-	errno.h	-

Table 7-22. Sample Program Processing Programs (2/3)

Processing Program Name	Function Name	File Name	Remark
USB storage class driver processing programs			
Initialization function	usbf850_init	usbf850.c	C language
Interrupt handler (for INTUSBF0 signal)	usbf850_inthdr	usbf850.c	C language
Interrupt handler (for INTUSBF1 signal)	usbf850_inthdr1	usbf850.c	C language
Interrupt handler (for INTUSBF2 signal)	usbf850_inthdr2	usbf850.c	C language
Control transfer processing task (endpoint 0)	task_usb0b	usbf850.c	C language
Bulk-out processing task (endpoint 2)	task_usb1b	usbf850.c	C language
Data transmission function	usbf850_data_send	usbf850.c	C language
Data reception function	usbf850_data_receive	usbf850.c	C language
Null data transmission function (endpoint 0)	usbf850_sendnullEP0	usbf850.c	C language
Stall response processing function (endpoint 0)	usbf850_sendstallEP0	usbf850.c	C language
Stall response processing function (endpoint 1)	usbf850_bulkin1_stall	usbf850.c	C language
Stall response processing function (endpoint 2)	usbf850_bulkout1_stall	usbf850.c	C language
System call call function (loc_cpu)	usbf850_loc_cpu	usbf850.c	C language
System call call function (unl_cpu)	usbf850_unl_cpu	usbf850.c	C language
Request processing function	usbf850_rxreq	usbf850.c	C language
Request data read function	usbf850_rxreq_read	usbf850.c	C language
Standard request processing function	usbf850_standardreq	usbf850.c	C language
Get Descriptor request processing function	usbf850_getdesc	usbf850.c	C language
Stall response processing function for setting request processing function (endpoint 0)	usbf850_sstall_ctrl	usbf850.c	C language
USB header file	-	usbf850.h	-
USB descriptor declaration	-	usbf850desc.h	-
Bulk-Only Mass Storage Reset request processing function (device-class-specific request processing)	usbf850_blkonly_mass_storage_reset	usbf850_storage.c	C language
Max LUN request processing function (device-class-specific request processing)	usbf850_max_lun	usbf850_storage.c	C language
Function for registering request processing function specific to device class for USB storage class	usbf850_setfunction_storage	usbf850_storage.c	C language
CBW reception processing function	usbf850_rx_cbw	usbf850_storage.c	C language
CBW check function	usbf850_storage_cbwchk	usbf850_storage.c	C language
CBW error processing function	usbf850_cbw_error	usbf850_storage.c	C language
CBW NO DATA command processing function	usbf850_no_data	usbf850_storage.c	C language
CBW DATA IN command processing function	usbf850_data_in	usbf850_storage.c	C language
CBW DATA OUT command processing function	usbf850_data_out	usbf850_storage.c	C language
CSW transmission processing function	usbf850_csw_ret	usbf850_storage.c	C language
Header file for interface function between USB and storage device	-	usbf850_storage.h	-
DMA initialization processing function for USB	usbf850_dma_init	usbf850_dma.c	C language
DMA start processing function for USB	usbf850_dma_start	usbf850_dma.c	C language
DMA header file	-	usbf850_dma.h	-

Table 7-22. Sample Program Processing Programs (3/3)

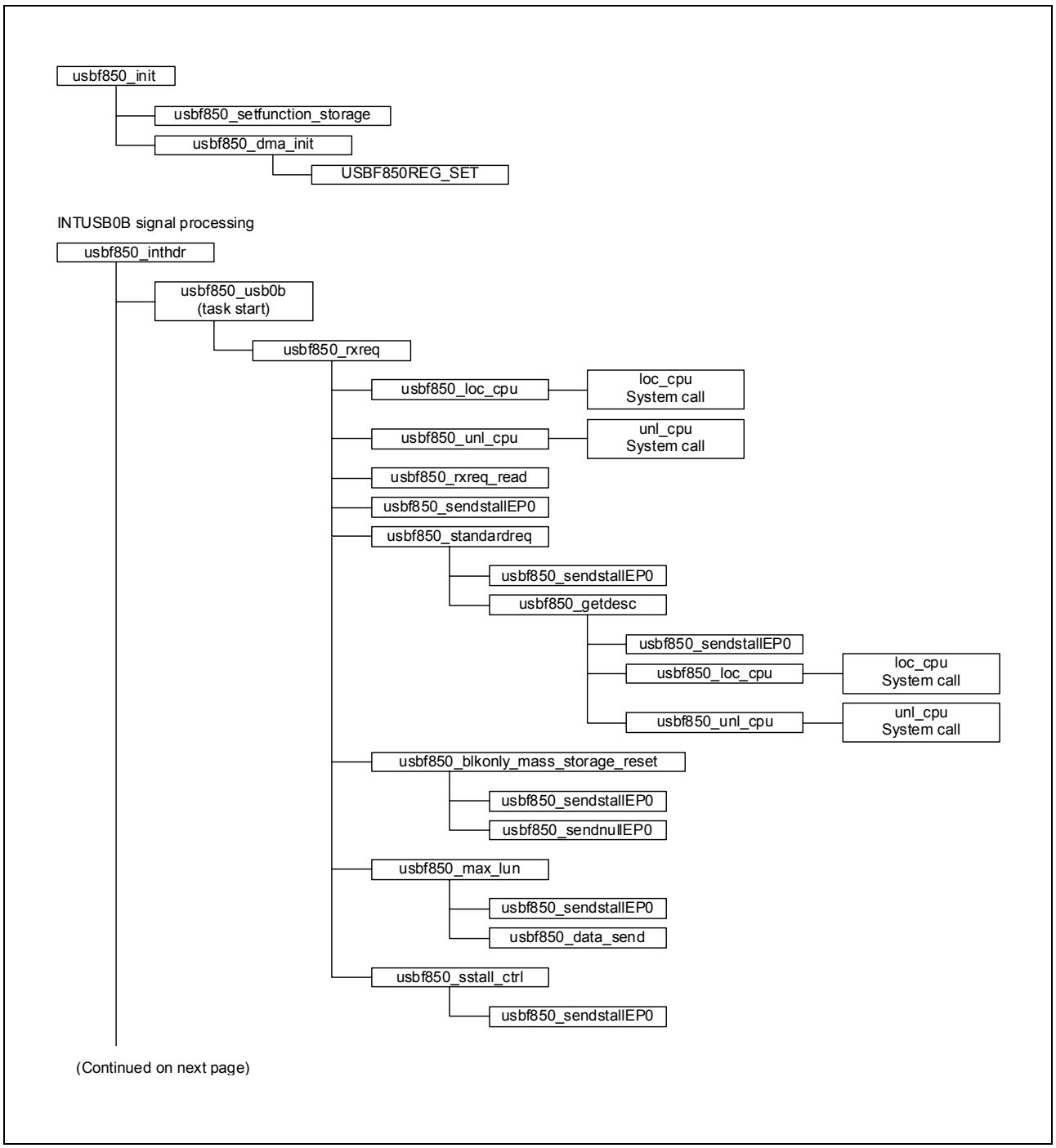
Processing Program Name	Function Name	File Name	Remark
Storage device processing programs			
Storage device initialization function	storageDev_Init	ata_ctrl.c	C language
Storage device header file	-	ata.h	-
CBWCB command analysis processing function	scsi_command_to_ata	scsi_cmd.c	C language
TEST UNIT READY command processing function	ata_test_unit_ready	scsi_cmd.c	C language
SEEK command processing function	ata_seek	scsi_cmd.c	C language
START STOP UNIT command processing function	ata_start_stop_unit	scsi_cmd.c	C language
SYNCHRONIZE CACHE command processing function	ata_synchronize_cache	scsi_cmd.c	C language
REQUEST SENSE command processing function	ata_request_sense	scsi_cmd.c	C language
INQUIRY command processing function	ata_inquiry	scsi_cmd.c	C language
MODE SELECT command processing function	ata_mode_select	scsi_cmd.c	C language
MODE SELECT (10) command processing function	ata_mode_select10	scsi_cmd.c	C language
MODE SENSE command processing function	ata_mode_sense	scsi_cmd.c	C language
MODE SENSE (10) command processing function	ata_mode_sense10	scsi_cmd.c	C language
READ FORMAT CAPACITIES command processing function	ata_read_format_capacities	scsi_cmd.c	C language
READ CAPACITY command processing function	ata_read_capacity	scsi_cmd.c	C language
READ (6) command processing function	ata_read6	scsi_cmd.c	C language
READ (10) command processing function	ata_read10	scsi_cmd.c	C language
WRITE (6) command processing function	ata_write6	scsi_cmd.c	C language
WRITE (10) command processing function	ata_write10	scsi_cmd.c	C language
VERIFY command processing function	ata_verify	scsi_cmd.c	C language
WRITE VERIFY command processing function	ata_write_verify	scsi_cmd.c	C language
WRITE BUFF command processing function	ata_write_buff	scsi_cmd.c	C language
SCSI-to-USB data transmission processing function	scsi_to_usb	scsi_cmd.c	C language
SCSI command processing header file	-	scsi.h	-
Function macros			
PFESiP/V850EP1 peripheral I/O register setting function (1-byte units: 8 bits)	USBF850REG_SET	usbf850.h	C language
PFESiP/V850EP1 peripheral I/O register read function (1-byte units: 8 bits)	USBF850REG_READ	usbf850.h	C language
PFESiP/V850EP1 peripheral I/O register setting function (1-word units: 16 bits)	USBF850REG_SET_W	usbf850.h	C language
PFESiP/V850EP1 peripheral I/O register read function (1-word units: 16 bits)	USBF850REG_READ_W	usbf850.h	C language

7.6.2 Function tree

A function tree of the functions related to calls of the sample program is shown below.

Caution usbf850_init and storageDev_Init are called from the initialization handler.
usbf850_dma_init is called from usbf850_init.

Figure 7-10. Sample Program Function Tree (1/4)



(Continued on next page)

Figure 7-10. Sample Program Function Tree (2/4)

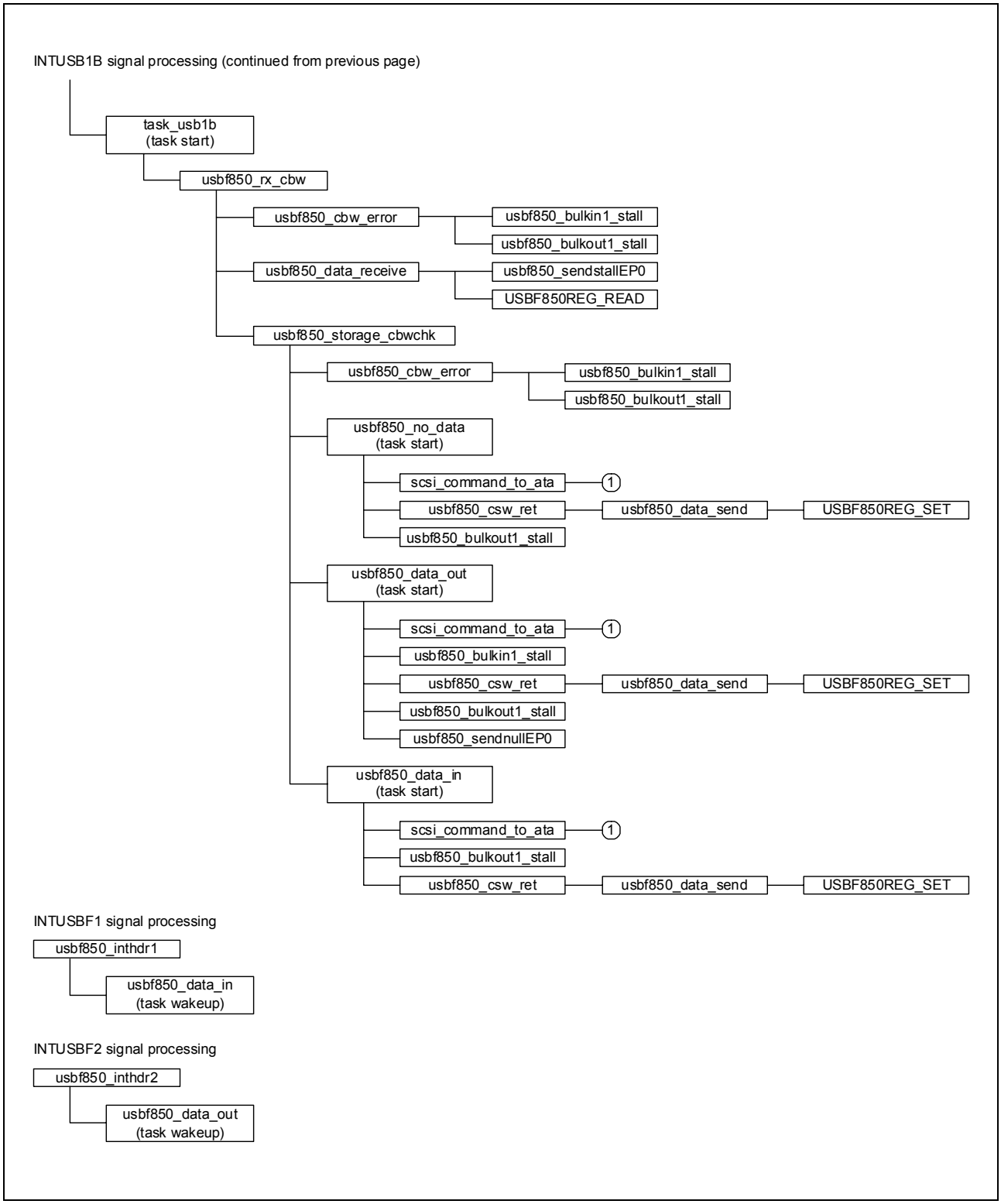


Figure 7-10. Sample Program Function Tree (3/4)

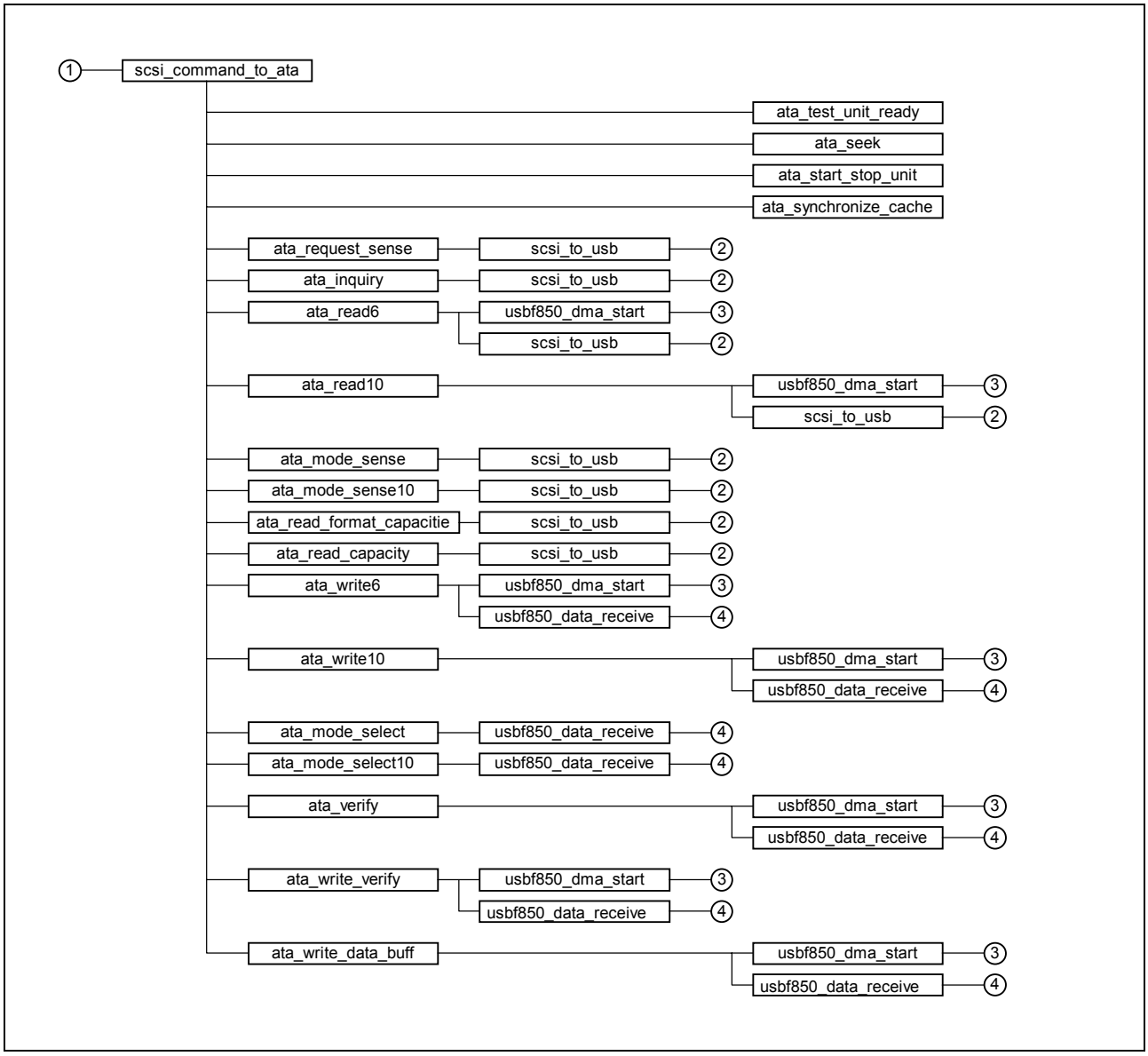
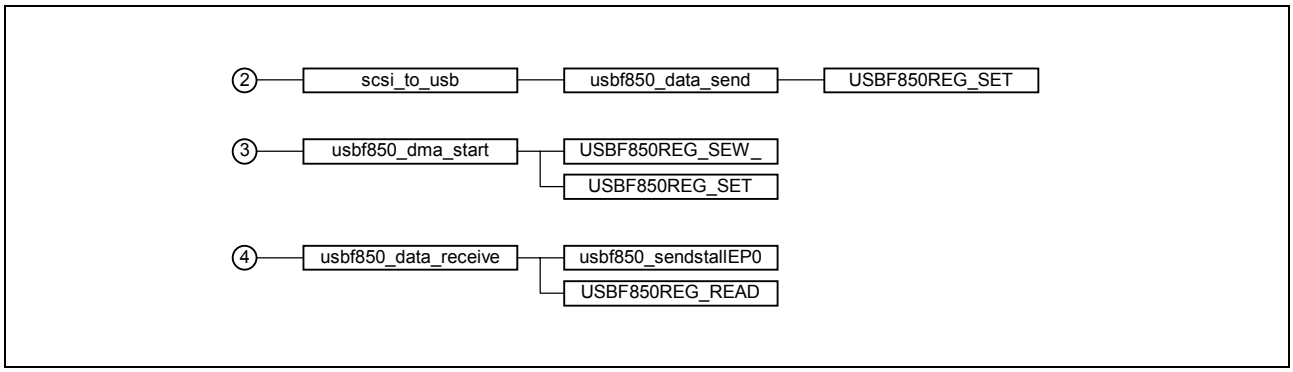


Figure 7-10. Sample Program Function Tree (4/4)



7.6.3 Explanation of functions

The functions of this sample program are explained based on the following description format.

xxxx <1>	Valid issue range:---- <2>

<1> Name: Function name

<2> V Valid issue range: Type of processing program for which a function can be issued

Task:	The function can be issued only from a task-type processing program.
Non-task:	The function can be issued only from a non-task-type processing program.
Task | Non-task:	The function can be issued from both a task- and a non-task-type processing program.
-:	The function cannot be called because it is an interrupt handler or task.

[Overview] <3> Functional overview of the function

[C format] <4> Description format when issuing the function from a processing program described in C language

[Parameter] <5> The parameter(s) of the function are shown in the following format.

I/O	Parameter	Description
A	B	C

A: Parameter type

- I: Parameter input to the USB function controller
- O: Parameter output from the USB function controller

B: Parameter data type

C: Description of the parameter

[Function] <6> Functional details of the function

[Return value] <7> Return value(s) from the function indicated as a data macro or numerical value

usbf850_init	Valid issue range: Non-task | Task

[Overview] This function initializes the USB function controller built into the PFESiP/V850EP1.

[C format] void usbf850_init (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is called from the software initialization block and initializes the USB function controller built into the PFESiP/V850EP1.

Remark See **7.2.1 Initialization processing** for details of the initialization processing.

[Return value] None

usbf850_inthdr	Valid issue range: -

[Overview] Interrupt handler (for the INTUSBF0 signal) for the USB function controller built into the PFESiP/V850EP1

[C format] ID usbf850_inthdr (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is an interrupt handler started by the INTUSBF0 signal (USB function status 0).
With the sample program, the interrupt source is examined and if it is the CPUDEC interrupt, the control transfer processing task (task_usb0b) will be started.
If the interrupt source is the BKO1DT interrupt, the bulk-out processing task (task_usb1b) will be started.
This handler is defined by the CF definition file.

Remark See **7.2.2 Interrupt servicing** for details of the interrupt servicing.

[Return value] Object ID No. (task ID No.)

usbf850_inthdr1	Valid issue range: -

[Overview] Interrupt handler (for the INTUSBF1 signal) for the USB function controller built into the PFESiP/V850EP1

[C format] ID usbf850_inthdr1 (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is an interrupt handler started by the INTUSBF1 signal (USB function status 1).
The UF0IS0 register (UF0INT status 0 register) is read and the interrupt source is examined and if it is the DMAED or SHORT interrupt, the UF0DMS1 register (DMA status 1 register) will be read, the interrupt source will be examined, and the usbf850_data_in task will be awakened. (usbf850_data_in enters the sleep state after DMA is started.)
This handler is defined by the CF definition file.

Remark See **7.2.2 Interrupt servicing** for details of the interrupt servicing.

[Return value] Object ID No. (task ID No.)

usbf850_inthdr2	Valid issue range: -

[Overview] Interrupt handler (for the INTUSBF2 signal) for the USB function controller built into the PFESiP/V850EP1

[C format] ID usbf850_inthdr2 (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is an interrupt handler started by the INTUSBF2 signal (USB function status 2).
The UF0IS0 register (UF0INT status 0 register) is read and the interrupt source is examined and if it is the DMAED or SHORT interrupt, the UF0DMS1 register (DMA status 1 register) will be read, the interrupt source will be examined, and the usbf850_data_out task will be awakened. (usbf850_data_out enters the sleep state after DMA is started.)
This handler is defined by the CF definition file.

[Return value] Object ID No. (task ID No.)

task_usb0b	Valid issue range: -

[Overview] This function performs interrupt servicing by the INTUSBF0 signal.

[C format] void task_usb0b (VP exinf)

[Parameter]

I/O	Parameter	Description
I	VP	exinf expansion information

This parameter is an area that stores user-specific information related to the target task and the user can freely use.
The information set to exinf can be dynamically acquired by issuing the ref_tsk system call from a processing program (task, non-task).

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Function] This function is a task started from the interrupt handler for the INTUSBF0 interrupt signal (USB function status 0 interrupt). With the sample program, the usbf850_rxreq function is called and a standard USB device request and device-class-specific requests are processed.

Caution With this sample program, only the standard device request Get Descriptor (String Descriptor) that does not automatically respond with the USB function controller built into the PFESiP/V850EP1 is processed.

Remark See **7.2.2 Interrupt servicing** for details of the interrupt servicing.

[Return value] None

task_usb1b	Valid issue range: -

[Overview] This function performs interrupt servicing by the INTUSBF1 signal.

[C format] void task_usb1b (VP exinf)

[Parameter]

I/O	Parameter	Description
I	VP	exinf expansion information

This parameter is an area that stores user-specific information related to the target task and the user can freely use.
The information set to exinf can be dynamically acquired by issuing the ref_tsk system call from a processing program (task, non-task).

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Function] This function is a task started from the interrupt handler for the INTUSBF1 interrupt signal (USB function status 1 interrupt). With the sample program, the interrupt source is checked and if the interrupt source is BKO1DT and the receive data length is equal to the CBW data size, the usbf850_rx_cbw function will be called.

Remark See **7.2.2 Interrupt servicing** for details of the interrupt servicing.

[Return value] None

usbf850_data_send	Valid issue range: Non-task | Task

[Overview] USB function controller data transmission function

[C format] long usbf850_data_send (unsigned char* data, long len, char ep)

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Transmit data start address
I	long len	Data size
i	char ep	Endpoint number

[Function] This function transmits the data of the size specified by “len” from the address specified by “data” at the endpoint specified by “ep”.

[Return value] Status during transmission

DEV_ERROR: Illegal endpoint number

DEV_OK: Normal termination

usbf850_data_receive	Valid issue range: Non-task | Task

[Overview] USB function controller data reception function

[C format] long usbf850_data_receive (unsigned char* data, long len, char ep)

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Receive data buffer start address
I	long len	Data size
i	char ep	Endpoint number

[Function] This function reads the data of the size specified by “len” from the buffer of the endpoint specified by “ep” and stores the data in the address specified by “data”.

[Return value] Status during reception

DEV_ERROR: Illegal receive data size or endpoint number

DEV_OK: Normal termination

usbf850_sendnullEP0	Valid issue range: Non-task | Task

[Overview] Control endpoint (endpoint 0) Null data transmission function

[C format] void usbf850_sendnullEP0 (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function transmits Null data (data of data size 0) at a control endpoint (endpoint 0).

[Return value] None

usb850_sendstallEP0	Valid issue range: Non-task | Task

[Overview] Control endpoint (endpoint 0) STALL response function

[C format] void usbf850_sendstallEP0 (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function sets a STALL response at a control endpoint (endpoint 0).

[Return value] None

usbf850_bulkin1_stall	Valid issue range: Non-task | Task

[Overview] Bulk endpoint (endpoint 1) STALL response function

[C format] void usbf850_bulkin1_stall (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function sets a STALL response at a bulk endpoint (endpoint 1).

[Return value] None

usbf850_bulkout1_stall	Valid issue range: Non-task | Task

[Overview] Bulk endpoint (endpoint 2) STALL response function

[C format] void usbf850_bulkout1_stall (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function sets a STALL response at a bulk endpoint (endpoint 2).

[Return value] None

usbf850_loc_cpu	Valid issue range: Task

[Overview] This function disables acknowledging maskable interrupts and dispatch processing.

[C format] void usbf850_loc_cpu (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function calls the loc_cpu system call.

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Return value] None

usbf850_unl_cpu	Valid issue range: Task

[Overview] This function enables acknowledging maskable interrupts and dispatch processing.

[C format] void usbf850_unl_cpu (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function calls the unl_cpu system call.

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Return value] None

usbf850_rxreq	Valid issue range: Non-task | Task

[Overview] This function performs USB request processing.

[C format] void usbf850_rxreq (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is called by the task_usb0b task started by the INTUSBF0 interrupt signal. This function calls the SETUP data read processing, analyzes the read data, and calls the USB request processing based on the analysis result.

[Return value] None

usbf850_rxreq_read	Valid issue range: Non-task | Task

[Overview] This function reads the USB request data.

[C format] void usbf850_rxreq_read (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function reads the SETUP data received following the Setup token in a control transfer (endpoint 0). The SETUP data is distinguished from normal data, stored in a dedicated register, and always read in 8-byte units.

[Return value] None

usbf850_standardreq	Valid issue range: Non-task | Task

[Overview] This function processes standard USB requests.

[C format] void usbf850_standardreq (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function will be called if the request is a standard request after the SETUP data is read.
The request is checked whether it is a Get Descriptor request and the usbf850_getdesc function is called.

[Return value] None

usbf850_getdesc	Valid issue range: Non-task | Task

[Overview] This function processes the Get Descriptor (String Descriptor) request of the standard USB requests.

[C format] void usbf850_getdesc (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is called by the usbf850_standardreq function and processes the Get Descriptor (String Descriptor) request of the standard USB requests.
This function performs a STALL response for requests other than the Get Descriptor (String Descriptor) request.

[Return value] None

usbf850_sstall_ctrl	Valid issue range: Non-task | Task

[Overview] Control endpoint (endpoint 0) STALL response processing function

[C format] void usbf850_sstall_ctrl (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function sets a STALL response at a control endpoint (endpoint 0).
This function uses a request code for the suffixes of an array when providing the class request processing function as a function pointer for the array with the usbf850_setfunction_storage function. By registering this function to a block without a corresponding request, the function is set to perform a STALL response when an unsupported request code is received.

[Return value] None

usbf850_blkonly_mass_storage_reset	Valid issue range: Non-task | Task

[Overview] Function for processing a request (Bulk-Only Mass Storage Reset) specific to the USB Mass Storage class

[C format] void usbf850_blkonly_mass_storage_reset (void)

[Parameter]

I/O	Parameter	Description
_	_	_

[Function] This function processes the Bulk-Only Mass Storage Reset request. If this request is received, this function initializes the storage device. With the sample program, this function clears the buffers of the bulk endpoints (endpoint numbers 1 and 2) and sets a STALL response.

[Return value] None

usbf850_max_lun	Valid issue range: Non-task | Task

[Overview] Function for processing a request (Get Max LUN) specific to the USB Mass Storage class

[C format] void usbf850_max_lun (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function processes the Get Max LUN request.
If this request is received, this function returns the total number of logical units supported by the device as one byte of data. With the sample program, the storage device is a virtual device, so this function provides 00H as data and transmits the data at a control endpoint (endpoint number 0).

[Return value] None

usbf850_setfunction_storage	Valid issue range: Non-task | Task

[Overview] This function performs the processing for registering a request processing function specific to the USB Mass Storage class to an array as a function pointer.

[C format] void usbf850_setfunction_storage (void)

[Parameter]

I/O	Parameter	Description
_	_	_

[Function] This function is called from the USB initialization processing and registers a request processing function specific to the USB Mass Storage class to an array (array name: Req_Func_C) as a function pointer.
For an unsupported request code, the usbf850_sstall_ctrl function is registered and a STALL response is set to be performed when an unsupported request is received.

[Return value] None

usbf850_rx_cbw	Valid issue range: Non-task | Task

[Overview] CBW data reception processing function

[C format] void usbf850_rx_cbw (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function is called from the interrupt servicing task and reads the CBW data.
Afterward, this function calls the usbf850_storage_cbwchk function.

Remark See **7.2.3 CBW data processing** for details of the CBW reception processing.

[Return value] None

usbf850_storage_cbwchk	Valid issue range: Non-task | Task

[Overview] CBW data command analysis processing function

[C format] int usbf850_storage_cbwchk (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function analyzes the read CBW data and starts the processing task of the corresponding data direction.

Remark See **7.2.3 CBW data processing** for details of the CBW reception processing.

[Return value] Status during CBW check

DEV_ERROR: Illegal CBWCB length

DEV_OK: Normal termination

usbf850_cbw_error	Valid issue range: Non-task | Task

[Overview] CBW data error processing function

[C format] void usbf850_cbw_error (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function sets a STALL response for bulk endpoints (endpoint numbers 1 and 2) when a CBW error is detected.

[Return value] None

usbf850_no_data	Valid issue range: Non-task | Task

[Overview] SCSI NO DATA command processing task

[C format] void usbf850_no_data (VP exinf)

[Parameter]

I/O	Parameter	Description
I	VP	exinf expansion information

This parameter is an area that stores user-specific information related to the target task and the user can freely use.
The information set to exinf can be dynamically acquired by issuing the ref_tsk system call from a processing program (task, non-task).

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Function] This function is a task for SCSI NO DATA command processing.
This function calls the scsi_command_to_ata function and passes the CBW processing status (GOOD, FAIL, PHASE) from the execution result to the CSW response processing function.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] None

usbf850_data_in	Valid issue range: Non-task | Task

[Overview] SCSI DATA IN command processing task

[C format] void usbf850_data_in (VP exinf)

[Parameter]

I/O	Parameter	Description
I	VP	exinf expansion information

This parameter is an area that stores user-specific information related to the target task and the user can freely use.
The information set to exinf can be dynamically acquired by issuing the ref_tsk system call from a processing program (task, non-task).

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Function] This function is a task for SCSI DATA IN command processing.
This function calls the scsi_command_to_ata function and passes the CBW processing status (GOOD, FAIL, PHASE) from the execution result to the CSW response processing function.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] None

usbf850_data_out	Valid issue range: Non-task | Task

[Overview] SCSI DATA OUT command processing task

[C format] void usbf850_data_out (VP exinf)

[Parameter]

I/O	Parameter	Description
I	VP	exinf expansion information

This parameter is an area that stores user-specific information related to the target task and the user can freely use.
The information set to exinf can be dynamically acquired by issuing the ref_tsk system call from a processing program (task, non-task).

Remark See the **RX850 Pro Basics User's Manual** for details of system calls.

[Function] This function is a task for SCSI DATA OUT command processing.
This function calls the scsi_command_to_ata function and passes the CBW processing status (GOOD, FAIL, PHASE) from the execution result to the CSW response processing function.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] None

usbf850_csw_ret	Valid issue range: Non-task | Task

[Overview] CSW response processing function

[C format] long usbf850_csw_ret (BYTE status)

[Parameter]

I/O	Parameter	Description
I	BYTE status	Status to be transmitted (GOOD, FAIL, PHASE)

[Function] CSW response processing function
This function transmits the CBW processing status (GOOD, FAIL, PHASE) passed by an argument to the host.

[Return value] Status during transmission
DEV_OK: Normal termination

usbf850_dma_init	Valid issue range: Non-task | Task

[Overview] DMA initialization function

[C format] void usbf850_dma_init (char ep)

[Parameter]

I/O	Parameter	Description
I	char ep	Number of endpoint using DMA

[Function] This function initializes the DMA to be used for the endpoint specified by an argument.

[Return value] None

usbf850_dma_start	Valid issue range: Non-task | Task

[Overview] DMA start function

[C format] void usbf850_dma_start (unsigned char* data, long len, char ep)

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Pointer of the buffer storing transmit and receive data
I	long len	Data length
I	char ep	Number of endpoint using DMA

[Function] If “ep” is 1, this function transfers data of the length specified by “len” via DMA from the buffer specified by “data” to the EP1_BULK_IN register.
If “ep” is 2, this function reads data of the length specified by “len” via DMA from the EP1_BULK_OUT register to the buffer specified by “data”.

[Return value] None

storageDev_Init	Valid issue range: Non-task | Task

[Overview] Storage device initialization processing function

[C format] void storageDev_Init (void)

[Parameter]

I/O	Parameter	Description
–	–	–

[Function] This function initializes the storage device.
This function only clears the secured memory area to 0, because a virtual device for which only a storage area is secured in the memory is used with the sample program.

[Return value] None

scsi_command_to_ata	Valid issue range: Non-task | Task

[Overview] SCSI command processing function

[C format] long scsi_command_to_ata
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Transfer direction

[Function] This function calls the command processing function from the SCSI command reported with the CBW.

[Return value] Status during command processing

DEV_ERR_NODATA:	Transfer direction error with NO DATA command
DEV_ERR_READ:	Transfer direction error with READ command
DEV_ERR_WRITE:	Transfer direction error with WRITE command
DEV_ERROR:	Status other than the above three statuses in the execution result of each command or illegal request
DEV_OK:	Normal termination

ata_test_unit_ready	Valid issue range: Non-task | Task

[Overview] TEST UNIT READY command processing function

[C format] long ata_test_unit_ready (long TransFlag)

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Function] This function performs the TEST UNIT READY command processing. This function executes no processing, returns OK, and ends, because a virtual device is used with the sample program.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_NODATA: Transfer direction error with NO DATA command

DEV_OK: Normal termination

ata_seek	Valid issue range: Non-task | Task

[Overview] SEEK command processing function

[C format] long ata_seek (long TransFlag)

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Function] This function performs the SEEK command processing.
This function executes no processing, returns OK, and ends, because a virtual device is used with the sample program.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_NODATA:	Transfer direction error with NO DATA command
DEV_OK:	Normal termination

ata_start_stop_unit	Valid issue range: Non-task | Task

[Overview] START STOP UNIT command processing function

[C format] long ata_start_stop_unit (long TransFlag)

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Function] This function performs the START STOP UNIT command processing.
This function executes no processing, returns OK, and ends, because a virtual device is used with the sample program.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_NODATA:	Transfer direction error with NO DATA command
DEV_OK:	Normal termination

ata_synchronize_cache	Valid issue range: Non-task | Task

[Overview] SYNCHRONIZE CACHE command processing function

[C format] long ata_synchronize_cache (long TransFlag)

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Function] This function performs the SYNCHRONIZE CACHE command processing.
This function executes no processing, returns OK, and ends, because a virtual device is used with the sample program.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_NODATA: Transfer direction error with NO DATA command
DEV_OK: Normal termination

ata_request_sense	Valid issue range: Non-task | Task

[Overview] REQUEST SENSE command processing function

[C format] long ata_request_sense
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the REQUEST SENSE command processing.
If the transmit data size specified by the command is 0, this function executes no processing, returns OK, and ends, because a virtual device is used with the sample program.
If the data size specified by the command is not 0, REQUEST SENSE data of that data size will be provided and transmitted.
If the data length of the provided REQUEST SENSE data is exceeded, data of the data length of only the provided REQUEST SENSE data will be transmitted.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing
DEV_ERR_NODATA: Transfer direction error with NO DATA command
DEV_ERR_READ: Transfer direction error with READ command
DEV_OK: Normal termination

ata_inquiry	Valid issue range: Non-task | Task

[Overview] INQUIRY command processing function

[C format] long ata_inquiry (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the INQUIRY command processing.
With the sample program, this function provides and transmits INQUIRY data of the data size specified by the command. If the data length of the provided INQUIRY data is exceeded, data of the data length of only the provided INQUIRY data will be transmitted.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Illegal request or abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_mode_select	Valid issue range: Non-task | Task

[Overview] MODE SELECT (6) command processing function

[C format] long ata_mode_select (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the MODE SELECT (6) command processing.
With the sample program, this function reads data of the specified data size to the MODE SELECT table.
If the data length of the provided MODE SELECT table is exceeded, data of the data length of only the provided MODE SELECT table will be read.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_WRITE: Transfer direction error with WRITE command
DEV_ERROR: Illegal CDB content
DEV_OK: Normal termination

ata_mode_select10	Valid issue range: Non-task | Task

[Overview] MODE SELECT (10) command processing function

[C format] long ata_mode_select10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the MODE SELECT (10) command processing.
With the sample program, this function reads data of the specified data size to the MODE SELECT (10) table. If the data length of the provided MODE SELECT (10) table is exceeded, data of the data length of only the provided MODE SELECT (10) table will be read.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_WRITE: Transfer direction error with WRITE command
DEV_ERROR: Illegal CDB content
DEV_OK: Normal termination

ata_mode_sense	Valid issue range: Non-task | Task

[Overview] MODE SENSE (6) command processing function

[C format] long ata_mode_sense (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the MODE SENSE (6) command processing.
With the sample program, this function provides and transmits MODE SENSE data of the specified data size. If the data length of the provided MODE SENSE data is exceeded, data of the data length of only the provided MODE SENSE data will be read.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_mode_sense10	Valid issue range: Non-task | Task

[Overview] MODE SENSE (10) command processing function

[C format] long ata_mode_sense10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the MODE SENSE (10) command processing.
With the sample program, this function provides and transmits MODE SENSE (10) data of the specified data size. If the data length of the provided MODE SENSE (10) data is exceeded, data of the data length of only the provided MODE SENSE (10) data will be read.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_read_format_capacities	Valid issue range: Non-task | Task

[Overview] READ FORMAT CAPACITIES command processing function

[C format] long ata_read_format_capacities
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the READ FORMAT CAPACITIES command processing.
With the sample program, this function provides and transmits READ FORMAT CAPACITIES data of the specified data size. If the data length of the provided READ FORMAT CAPACITIES data is exceeded, data of the data length of only the provided READ FORMAT CAPACITIES data will be transmitted.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ:	Transfer direction error with READ command
DEV_ERROR:	bnormal termination of the scsi_to_usb execution result
DEV_OK:	Normal termination

ata_read_capacity	Valid issue range: Non-task | Task

[Overview] READ CAPACITY command processing function

[C format] long ata_read_capacity (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the READ CAPACITY command processing.
With the sample program, this function provides and transmits READ CAPACITY data of the specified data size. If the data length of the provided READ CAPACITY data is exceeded, data of the data length of only the provided READ CAPACITY data will be transmitted.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_read6	Valid issue range: Non-task | Task

[Overview] READ (6) command processing function

[C format] long ata_read6 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the READ (6) command processing.
This function reads storage device data of the specified size from the specified location.
With the sample program, this function reads the data in the virtual device.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Illegal CDB content or abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_read10	Valid issue range: Non-task | Task

[Overview] READ (10) command processing function

[C format] long ata_read10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the READ (10) command processing.
This function reads storage device data of the specified size from the specified location.
With the sample program, this function reads the data in the virtual device.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command
DEV_ERROR: Illegal CDB content or abnormal termination of the scsi_to_usb execution result
DEV_OK: Normal termination

ata_write6	Valid issue range: Non-task | Task

[Overview] WRITE (6) command processing function

[C format] long ata_write6 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the WRITE (6) command processing.
This function writes the data (received in the storage device) of the specified size from the specified location. With the sample program, this function writes the data to the virtual device.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_WRITE: Transfer direction error with WRITE command
DEV_ERROR: Illegal CDB content
DEV_OK: Normal termination

ata_write10	Valid issue range: Non-task | Task

[Overview] WRITE (10) command processing function

[C format] long ata_write10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the WRITE (10) command processing.
This function writes the data (received in the storage device) of the specified size from the specified location. With the sample program, this function writes the data to the virtual device.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_WRITE: Transfer direction error with WRITE command
DEV_ERROR: Illegal CDB content
DEV_OK: Normal termination

ata_verify	Valid issue range: Non-task | Task

[Overview] VERIFY command processing function

[C format] long ata_verify (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the VERIFY command processing.
This function checks storage device data of the specified size from the specified location.
With the sample program, this function does not check data, because a virtual device is used.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing

DEV_ERR_NODATA:	Transfer direction error with NO DATA command
DEV_ERROR:	Illegal CDB content
DEV_OK:	Normal termination

ata_write_verify	Valid issue range: Non-task | Task

[Overview] WRITE VERIFY command processing function

[C format] long ata_write_verify (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the WRITE VERIFY command processing.
This function writes data of the specified size from the specified location to the storage device and checks whether the written data is valid. With the sample program, this function writes data only to the specified memory and does not check the data, because a virtual device is used.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing
DEV_OK: Normal termination

ata_write_buff	Valid issue range: Non-task | Task

[Overview] WRITE BUFF command processing function

[C format] long ata_write_buff (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *ScsiCommandBuf	CBWCB when the SCSI protocol is used
I	BYTE *pbData	Address of each endpoint data register
I	long lDataSize	Transmit/receive data size
I	long TransFlag	Data transfer direction

[Function] This function performs the WRITE BUFF command processing.
This function writes data to a memory (data buffer).
With the sample program, this function executes no processing and ends normally, because a virtual device is used.

Remark See **7.2.4 SCSI command processing** for details of the SCSI command processing.

[Return value] Status during command processing
DEV_OK: Normal termination

scsi_to_usb	Valid issue range: Non-task | Task

[Overview] Processing of the function transmission data from a virtual device into the direction of the USB

[C format] long scsi_to_usb (BYTE *pbData, long TransFlag)

[Parameter]

I/O	Parameter	Description
I	BYTE *pbData	Transmit data start address
I	long TransFlag	Data transfer direction

[Function] This function transmits data from a virtual device into the direction of the USB.

[Return value] Status during command processing

DEV_ERR_READ: Transfer direction error with READ command

DEV_OK: Normal termination

USBF850REG_SET	Valid issue range: Non-task | Task

[Overview] PFESiP/V850EP1 peripheral I/O register setting function (1-byte units: 8 bits)

[C format] USBF850REG_SET (offset, val)

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Setting data

[Function] This function sets the data specified by “val” to the PFESiP/V850EP1 peripheral I/O register (register address specified by “offset”). Note that this macro can be used only with registers accessible in 1-byte (8-bit) units.

[Return value] None

USBF850REG_READ Valid issue range: Non-task | Task

[Overview] PFESiP/V850EP1 peripheral I/O register read function (1-byte units: 8 bits)

[C format] USBF850REG_READ (offset)

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address

[Function] This function reads the value of the PFESiP/V850EP1 peripheral I/O register (register address specified by “offset”). Note that this macro can be used only with registers accessible in 1-byte (8-bit) units.

[Return value] None

USBF850REG_SET_W	Valid issue range: Non-task | Task

[Overview] PFESiP/V850EP1 peripheral I/O register setting function (1-word units: 16 bits)

[C format] USBF850REG_SET_W (offset, val)

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Setting data

[Function] This function sets the data specified by “val” to the PFESiP/V850EP1 peripheral I/O register (register address specified by “offset”). Note that this macro can be used only with registers accessible in 1-word (16-bit) units.

[Return value] None

USBF850REG_READ_W	Valid issue range: Non-task | Task

[Overview] PFESiP/V850EP1 peripheral I/O register read function (1-word units: 16 bits)

[C format] USBF850REG_READ_W (offset)

[Parameter]

I/O	Parameter	Description
I	offset	peripheral I/O register address

[Function] This function reads the value of the PFESiP/V850EP1 peripheral I/O register (register address specified by “offset”). Note that this macro can be used only with registers accessible in 1-word (16-bit) units.

[Return value] None

For further information, please contact:

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211-65030
http://www.eu.necel.com/

Hanover Office
Podbielskistrasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02-667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
Tel: 010-8235-1155
http://www.cn.necel.com/

Shanghai Branch
Room 2509-2510, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai, P.R.China P.C:200120
Tel:021-5888-5400
http://www.cn.necel.com/

Shenzhen Branch
Unit 01, 39/F, Excellence Times Square Building,
No. 4068 Yi Tian Road, Futian District, Shenzhen,
P.R.China P.C:518048
Tel:0755-8282-9800
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
Unit 1601-1613, 16/F., Tower 2, Grand Century Place,
193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: 2886-9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-8175-9600
http://www.tw.necel.com/

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253-8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737
http://www.kr.necel.com/

G0706