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

V850E/ME2

32-Bit Single-Chip Microcontroller

USB Function Drivers

μPD703111A

Document No. U17069EJ1V0AN00 (1st edition)
Date Published May 2004 N CP(K)

[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.

- **The information in this document is current as of March, 2004. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC 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 NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

  "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.

  "Special": 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, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC Electronics product in your application, please contact the NEC Electronics office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

[GLOBAL SUPPORT]
http://www.necel.com/en/support/support.html

NEC Electronics America, Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65030

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
- **Tyskland Filial**
  Taeby, Sweden
  Tel: 08-63 80 820
- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-558-3737

NEC Electronics Shanghai Ltd.
Shanghai, P.R. China
Tel: 021-5888-5400

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377

NEC Electronics Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 6253-8311

J04.1

INTRODUCTION

Readers

This application note is intended for users who wish to understand the functions of the V850E/ME2 to design application systems using the V850E/ME2.

Purpose

The purpose of this application note is to help the user understand the composition of the USB function drivers incorporated in the V850E/ME2, using three sample programs.

Organization

This application note is broadly divided into the following sections.

- V850E/ME2 introduction
- USB bus driver
- USB storage class driver
- USB communication class driver

Remark The sample programs of the drivers are available from the following website.

http://www.necel.com/micro/v850/devicedata/index.html#SAMPLE

How to Read This Manual

It is assumed that the readers of this application note have general knowledge in the fields of electrical engineering, logic circuits, and microcontrollers.

- To know the hardware functions and electrical specifications of the V850E/ME2
  → Refer to the **V850E/ME2 Hardware User's Manual** (separately provided).

- To know the instruction functions of the V850E/ME2
  → Refer to the **V850E1 Architecture User's Manual** (separately provided).

The "yyy bit of the xxx register" is described as the "xxx.yyy bit" in this manual. Note with caution that if "xxx.yyy" is described as is in a program, however, the compiler/ assembler cannot recognize it correctly.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	$\overline{xxx}$ (overscore over pin or signal name) or /xxx ("/" before signal name)
Memory map address:	Top: higher, bottom: lower
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} = $1,024^3$

Related documents The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Documents related to V850E/ME2

Document Name	Document No.
V850E1 Architecture User's Manual	U14559E
V850E/ME2 Hardware User's Manual	U16031E
V850E/ME2 Hardware Application Note	U16794E
V850E/ME2 USB Function Drivers Application Note	This manual

Documents related to development tools (user's manuals)

Document Name		Document No.
CA850 Ver. 2.50 C Compiler Package	Operation	U16053E
	C Language	U16054E
	Assembly Language	U16042E
PM plus Ver. 5.10		U16569E
ID850 Ver. 2.50 Integrated Debugger	Operation	U16217E
ID850NW Ver. 2.51 Integrated Debugger	Operation	U16454E
RX850 Ver. 3.13 or Later Real-Time OS	Basics	U13430E
	Installation	U13410E
	Technical	U13431E
RX850 Pro Ver. 3.15 Real-Time OS	Basics	U13773E
	Installation	U13774E
	Technical	U13772E
RD850 Ver. 3.01 Task Debugger		U13737E
RD850 Pro Ver. 3.01 Task Debugger		U13916E
AZ850 Ver. 3.10 System Performance Analyzer		U14410E

CONTENTS

CHAPTER 1 V850E/ME2 INTRODUCTION

The V850E/ME2 is a product of the NEC Electronics single-chip microcontroller “V850 Series”. This chapter gives a simple outline of the V850E/ME2.

1.1 Outline

The V850E/ME2 is a 32-bit single-chip microcontroller that integrates the V850E1 CPU, which is a 32-bit RISC-type CPU core for ASIC, newly developed as the CPU core central to system LSI for the current age of system-on-chip. This device incorporates a cache, data RAM, instruction RAM, and various peripheral functions such as memory controllers, a DMA controller, real-time pulse unit, serial interfaces, USB function controller (USBF), and an A/D converter for realizing high-capacity data processing and sophisticated real-time control.

1.2 Features

- Number of instructions: 83
- Minimum instruction execution time: 10 ns/7.5 ns/6.7 ns (at internal 100 MHz/133 MHz/150 MHz operation)
- General-purpose registers: 32 bits × 32
- Instruction set:
  V850E1 CPU
  Signed multiplication (16 bits × 16 bits → 32 bits or 32 bits × 32 bits → 64 bits): 1 to 2 clocks
  Saturated operation instructions (with overflow/underflow detection function)
  32-bit shift instructions: 1 clock
  Bit manipulation instructions
  Load/store instructions with long/short format
  Signed load instructions
- Memory space:
  256 MB linear address space (common program/data use)
  Chip select output function: 8 spaces
  Memory block division function: 2, 4, 6, 8, 64 MB/block
  Programmable wait function
  Idle state insertion function
- External bus interface:
  32-bit data bus (address/data separated)
  32-/16-/8-bit bus sizing function
  External bus division function: 1/1, 1/2, 1/3, 1/4 (66 MHz MAX.)
  Bus hold function
  External wait function
  Address setup wait function
  Endian control function
- Internal memory:
  Instruction RAM: 128 KB
  Data RAM: 16 KB
- Instruction cache: 8 KB 2-way set associative
- Interrupts/exceptions:
  External interrupts: 40 (including NMI)
  Internal interrupts: 59 sources
  Exceptions: 2 sources
  Eight levels of priorities can be set.
- Memory access controller:
  DRAM controller (compatible with SDRAM)
  Page ROM controller
  Speculative read/write buffer function

- DMA controller:
  4 channels
  Transfer unit: 8 bits/16 bits/32 bits
  Maximum transfer count: 65,536 (2^{16})
  Transfer type: Flyby (1-cycle)/2-cycle
  Transfer mode: Single/Single step/Block
  Transfer target: Memory ↔ memory, memory ↔ I/O
  Transfer request: External request/On-chip peripheral I/O/ Software
  DMA transfer terminate (terminal count) output signal
  Next address setting function

- I/O lines:
  Input ports: 1
  I/O ports: 77

- Real-time pulse unit:
  16-bit timer/event counter: 6 channels (no capture operation for 2 channels)
    16-bit timers: 6
    16-bit capture/compare registers: 12
  16-bit interval timer: 4 channels
  16-bit up/down counter/timer for 2-phase encoder input: 2 channels
    16-bit capture/compare registers: 4
    16-bit compare registers: 4

- Serial interfaces (SIO):
  Asynchronous serial interface B (UARTB)
  Clocked serial interface 3 (CSI3)
    CSI3/UARTB: 1 channel
    UARTB: 1 channel
    CSI3: 1 channel
    USB function controller (USBF): 1 channel
      Full speed (12 Mbps)
      Endpoint Control transfer: 64 bytes × 2
      Interrupt transfer: 8 bytes × 2
      Bulk transfer (IN): 64 bytes × 2 banks × 2
      Bulk transfer (OUT): 64 bytes × 2 banks × 2

- A/D converter:
  10-bit resolution A/D converter: 8 channels

- PWM (Pulse Width Modulation):
  16-bit resolution PWM: 2 channels

- Clock generator:
  ×8 function using SSCG

- Power-save function:
  HALT/IDLE/software STOP mode

- Package:
  176-pin plastic LQFP (fine pitch) (24 × 24)
  240-pin plastic FBGA (16 × 16)

- CMOS technology:
  Fully static circuits

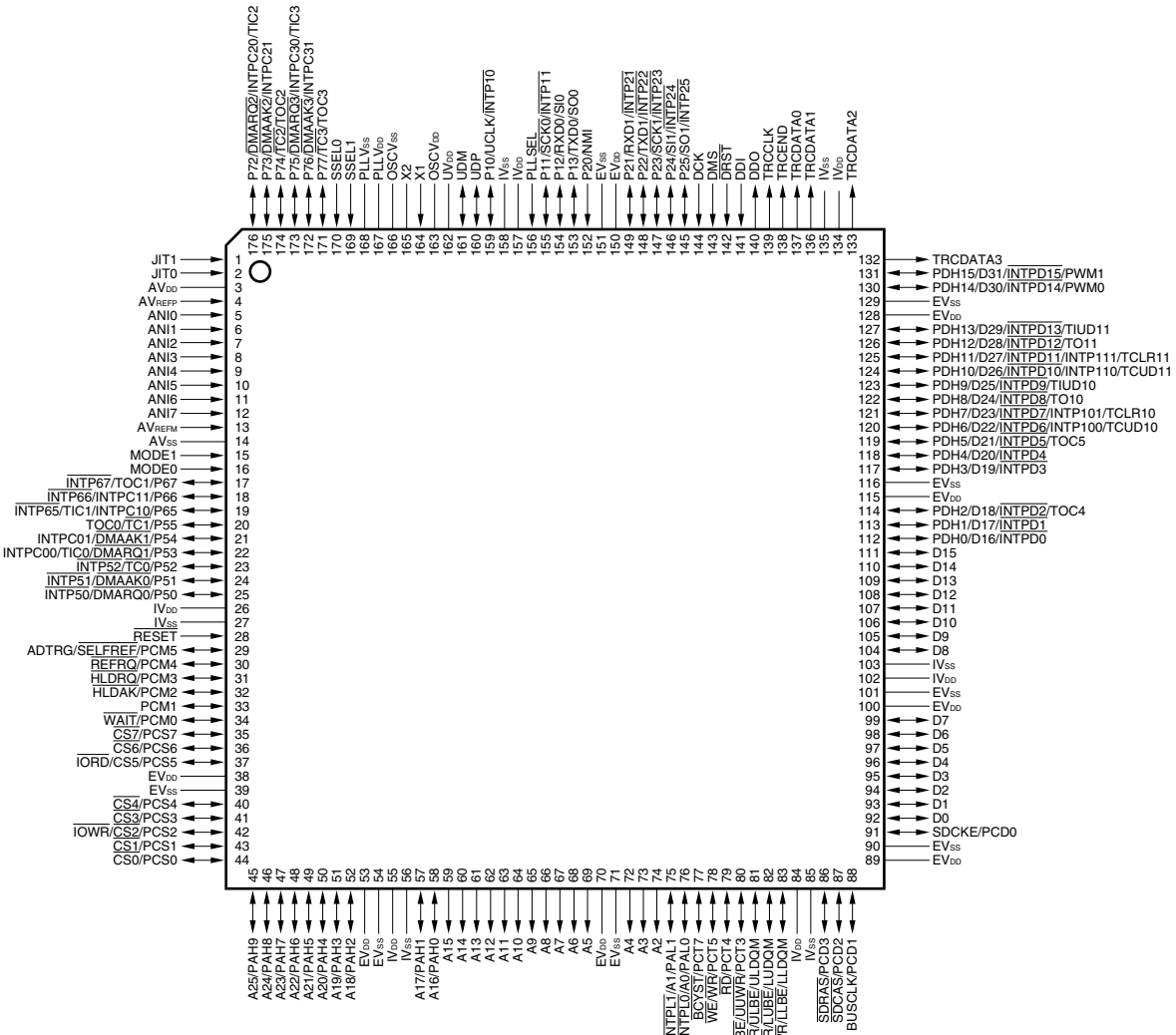
1.3 Ordering Information

Part Number	Package	Maximum Operating Frequency
μPD703111AGM-10-UEU	176-pin plastic LQFP (fine pitch) (24 × 24)	100 MHz
μPD703111AGM-13-UEU	176-pin plastic LQFP (fine pitch) (24 × 24)	133 MHz
μPD703111AGM-15-UEU	176-pin plastic LQFP (fine pitch) (24 × 24)	150 MHz
μPD703111AF1-10-GA3	240-pin plastic FBGA (16 × 16)	100 MHz
μPD703111AF1-13-GA3	240-pin plastic FBGA (16 × 16)	133 MHz
μPD703111AF1-15-GA3	240-pin plastic FBGA (16 × 16)	150 MHz

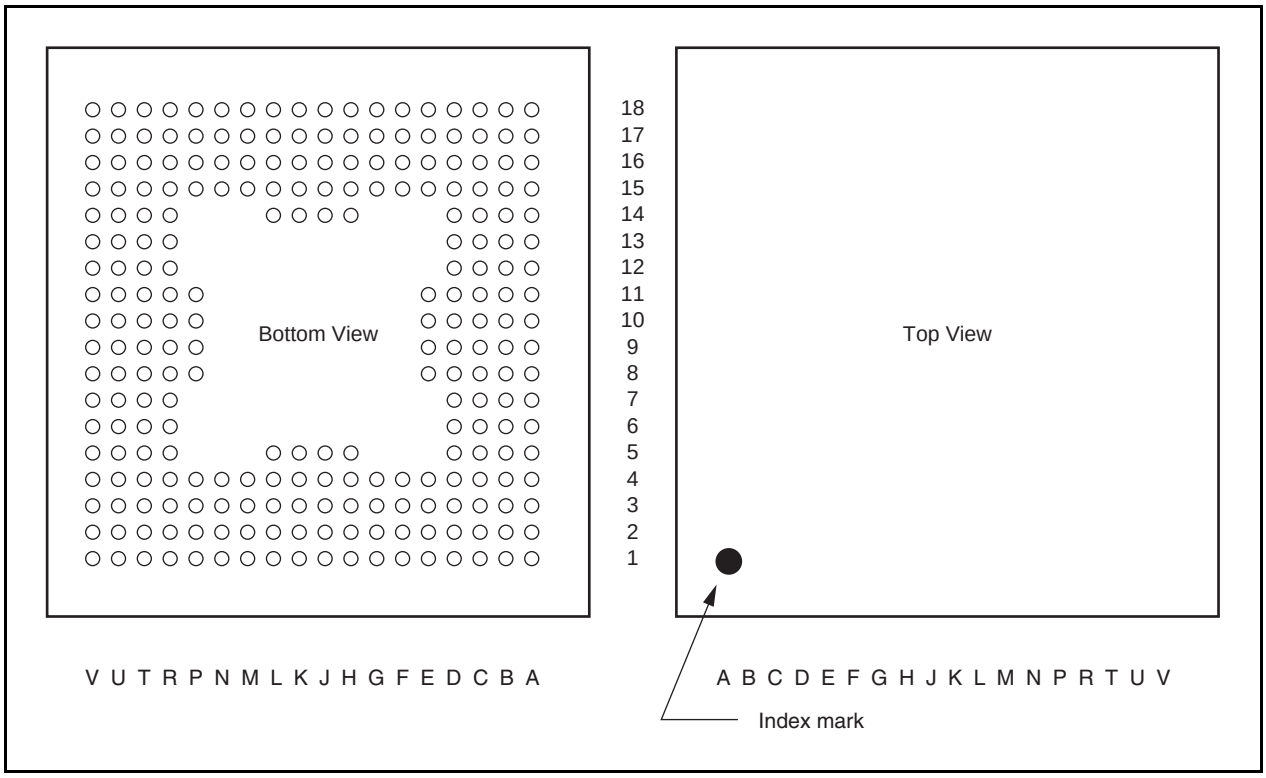
1.4 Pin Configuration

- 176-pin plastic LQFP (fine pitch) (24 × 24)
  μPD703111AGM-10-UEU
  μPD703111AGM-13-UEU
  μPD703111AGM-15-UEU

Top View



- 240-pin plastic FBGA (16 × 16)
  μPD703111AF1-10-GA3
  μPD703111AF1-13-GA3
  μPD703111AF1-15-GA3



(1/2)

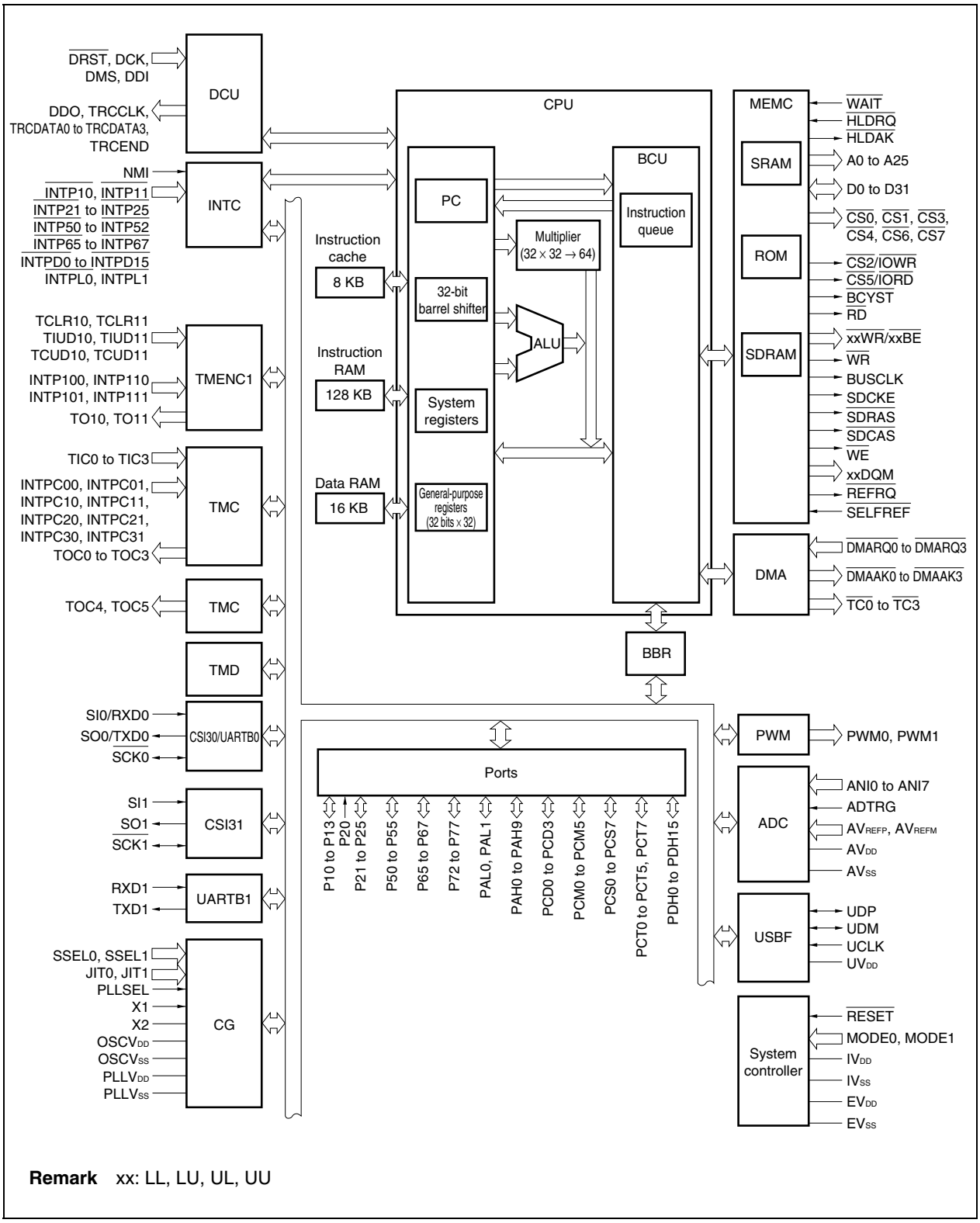
Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
A1	–	C12	IV_{DD}	G3	EV_{SS}
A2	IV_{SS}	C13	PAH2/A18	G4	D7
A3	PCT0/$\overline{LLWR}$/$\overline{LLBE}$/LLDQM	C14	PAH4/A20	G15	PCM1
A4	–	C15	PAH6/A22	G16	PCM3/$\overline{HLDRQ}$
A5	PCT4/$\overline{RD}$	C16	–	G17	PCM4/$\overline{REFRQ}$
A6	–	C17	PCS0/$\overline{CS0}$	G18	PCM5/ADTRG/$\overline{SELFREF}$
A7	–	C18	–	H1	–
A8	EV_{DD}	D1	D0	H2	D8
A9	A9	D2	EV_{SS}	H3	D9
A10	–	D3	PCD0/SDCKE	H4	D10
A11	A14	D4	EV_{DD}	H5	IV_{SS}
A12	IV_{SS}	D5	PCT1/$\overline{LUWR}$/$\overline{LUBE}$/LUDQM	H14	–
A13	EV_{DD}	D6	–	H15	$\overline{RESET}$
A14	–	D7	PAL0/$\overline{INTPL0}$/A0	H16	IV_{SS}
A15	PAH5/A21	D8	A4	H17	–
A16	PAH7/A23	D9	A6	H18	IV_{DD}
A17	PAH9/A25	D10	–	J1	–
A18	–	D11	A13	J2	D11
B1	–	D12	EV_{SS}	J3	D12
B2	PCD1/BUSCLK	D13	PAH3/A19	J4	–
B3	PCD2/$\overline{SDCAS}$	D14	–	J5	D13
B4	–	D15	–	J14	–
B5	PCT3/$\overline{UUWR}$/$\overline{UUBE}$/UUDQM	D16	PCS2/$\overline{CS2}$/$\overline{IOWR}$	J15	P50/$\overline{INTP50}$/$\overline{DMARQ0}$
B6	PCT7/$\overline{BCYST}$	D17	PCS3/$\overline{CS3}$	J16	P51/$\overline{INTP51}$/$\overline{DMAAK0}$
B7	A2	D18	EV_{DD}	J17	P52/$\overline{INTP52}$/$\overline{TC0}$
B8	–	E1	D3	J18	P53/INTPC00/TIC0/$\overline{DMARQ1}$
B9	A8	E2	D2	K1	D14
B10	A12	E3	D1	K2	D15
B11	PAH0/A16	E4	–	K3	PDH0/D16/$\overline{INTPD0}$
B12	–	E8	A3	K4	PDH1/D17/$\overline{INTPD1}$
B13	–	E9	A5	K5	PDH2/D18/$\overline{INTPD2}$/TOC4
B14	–	E10	A10	K14	P55/TOC0/$\overline{TC1}$
B15	–	E11	PAH1/A17	K15	P54/INTPC01/$\overline{DMAAK1}$
B16	PAH8/A24	E15	PCS4/$\overline{CS4}$	K16	P65/$\overline{INTP65}$/INTPC10/TIC1
B17	–	E16	EV_{SS}	K17	P66/$\overline{INTP66}$/INTPC11
B18	PCS1/$\overline{CS1}$	E17	PCS5/$\overline{CS5}$/$\overline{IORD}$	K18	–
C1	–	E18	PCS6/$\overline{CS6}$	L1	EV_{DD}
C2	–	F1	D6	L2	–
C3	PCD3/$\overline{SDRAS}$	F2	D5	L3	EV_{SS}
C4	IV_{DD}	F3	D4	L4	PDH3/D19/$\overline{INTPD3}$
C5	PCT2/$\overline{ULWR}$/$\overline{ULBE}$/ULDQM	F4	–	L5	PDH4/D20/$\overline{INTPD4}$
C6	PCT5/$\overline{WE}$/$\overline{WR}$	F15	–	L14	MODE1
C7	PAL1/$\overline{INTPL1}$/A1	F16	PCS7/$\overline{CS7}$	L15	–
C8	EV_{SS}	F17	PCM0/$\overline{WAIT}$	L16	MODE0
C9	A7	F18	PCM2/$\overline{HLDAK}$	L17	–
C10	A11	G1	IV_{DD}	L18	P67/$\overline{INTP67}$/TOC1
C11	A15	G2	EV_{DD}	M1	–

(2/2)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
M2	PDH5/D21/INTPD5/TOC5	R7	DCK	U4	–
M3	PDH6/D22/INTPD6/INTP100/TCUD10	R8	EV_{DD}	U5	TRCCLK
M4	–	R9	P11/INTP11/SCK0	U6	DRST
M15	ANI6	R10	IV_{SS}	U7	P25/INTP25/SO1
M16	AV_{REFM}	R11	UDM	U8	P22/INTP22/TXD1
M17	ANI7	R12	X2	U9	EV_{SS}
M18	AV_{SS}	R13	$PLLV_{DD}$	U10	IV_{DD}
N1	PDH7/D23/INTPD7/INTP101/TCLR10	R14	SSEL0	U11	–
N2	PDH8/D24/INTPD8/TO10	R15	–	U12	$OSCV_{DD}$
N3	PDH9/D25/INTPD9/TIUD10	R16	AV_{REFP}	U13	–
N4	PDH10/D26/INTPD10/INTP110/TCUD11	R17	AV_{DD}	U14	–
N15	ANI2	R18	–	U15	P76/INTPC31/DMAAK3
N16	ANI3	T1	EV_{DD}	U16	P73/INTPC21/DMAAK2
N17	ANI4	T2	TRCDATA3	U17	P72/INTPC20/TIC2/DMARQ2
N18	ANI5	T3	–	U18	–
P1	–	T4	TRCDATA1	V1	–
P2	PDH11/D27/INTPD11/INTP111/TCLR11	T5	TRCEND	V2	TRCDATA2
P3	PDH13/D29/INTPD13/TIUD11	T6	DDI	V3	IV_{SS}
P4	–	T7	–	V4	TRCDATA0
P8	P23/INTP23/SCK1	T8	P21/INTP21/RXD1	V5	–
P9	P12/SI0/RXD0	T9	P20/NMI	V6	DMS
P10	–	T10	–	V7	P24/INTP24/SI1
P11	UV_{DD}	T11	UDP	V8	–
P15	–	T12	X1	V9	P13/SO0/TXD0
P16	ANI0	T13	$OSCV_{SS}$	V10	PLLSEL
P17	ANI1	T14	SSEL1	V11	P10/INTP10/UCLK
P18	–	T15	P75/INTPC30/TIC3/DMARQ3	V12	–
R1	PDH12/D28/INTPD12/TO11	T16	–	V13	–
R2	EV_{SS}	T17	JIT1	V14	–
R3	PDH14/D30/INTPD14/PWM0	T18	JIT0	V15	$PLLV_{SS}$
R4	IV_{DD}	U1	PDH15/D31/INTPD15/PWM1	V16	P77/TOC3/TC3
R5	–	U2	–	V17	P74/TOC2/TC2
R6	DDO	U3	–	V18	–

Remark Leave the A1, A4, A6, A7, A10, A14, A18, B1, B4, B8, B12 to B15, B17, C1, C2, C16, C18, D6, D10, D14, D15, E4, F4, F15, H1, H14, H17, J1, J4, J14, K18, L2, L15, L17, M1, M4, P1, P4, P10, P15, P18, R5, R15, R18, T3, T7, T10, T16, U2 to U4, U11, U13, U14, U18, V1, V5, V8, V12 to V14, and V18 pins open.

1.5 Internal Block Diagram



Remark xx: LL, LU, UL, UU

1.6 Internal Memory

The V850E/ME2 has a 128 KB (64 KB × 2 banks) instruction memory, 8 KB (2-way set associative) instruction cache, and 16 KB data RAM.

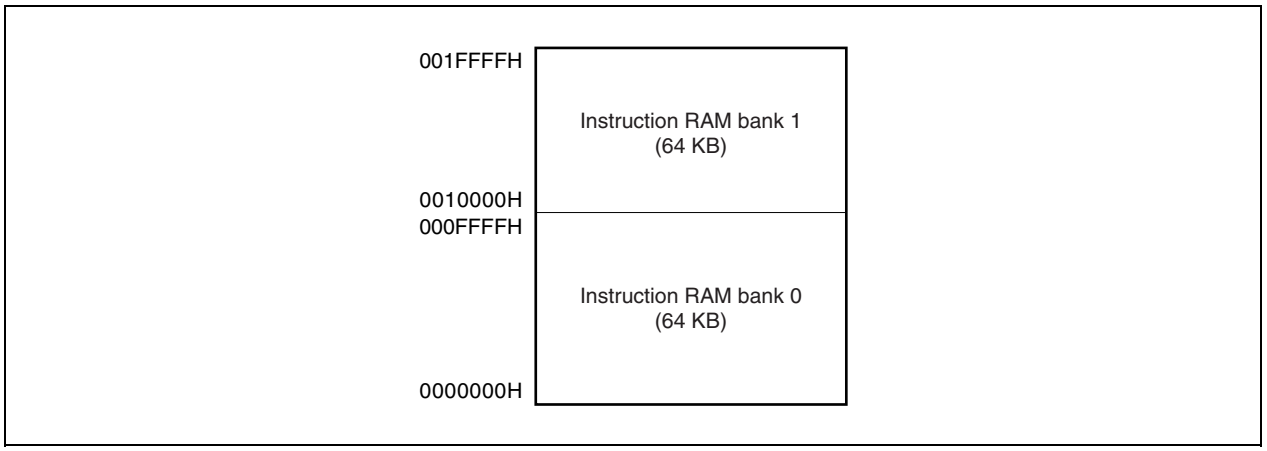
1.6.1 Internal instruction RAM

The internal instruction RAM has two modes: read mode and write mode. These modes are selected by the internal instruction RAM mode register (IRAMM).

After reset, the instruction RAM is initialized to the write mode. Therefore, the read mode is set after instruction data is transferred to the internal instruction RAM by program or the DMA controller. In the read mode, an instruction is fetched in one internal system clock.

Caution All interrupt/exception handlers, except the reset handler, are in bank 0 of the internal instruction RAM. Do not generate any interrupt/exception until a write operation to this bank is completed.

Figure 1-1. Memory Map of Internal Instruction RAM



1.6.2 Instruction cache function

The $\overline{CS0}$ to $\overline{CS2}$ spaces are cacheable areas. It can be specified by the cache configuration register (BHC) whether each space is used as a cacheable area or uncacheable area. The cache lock status of way 0, auto fill of way 0, and tag clear of ways 0 and 1 are specified by using the instruction cache control register (ICC).

Cautions 1. Write the BHC register after reset. After writing a value to this register, do not change it.

2. In an ordinary system, memories located in the $\overline{CS0}$ to $\overline{CS2}$ spaces are used as cacheable areas. In a system where a program is downloaded by a boot program, they are set as cacheable areas after downloading is completed.

An area where the instruction that sets the BHC register exists cannot be changed from an uncacheable area to cacheable area, or vice versa.

To set this space as a cacheable area, first set it as another uncacheable area and then change it to a cachable area, or set it using the internal instruction RAM area.

1.6.3 Internal data RAM

The internal data RAM area is allocated to the 16 KB area of addresses FFFB000H to FFFEFFFH. Instruction codes cannot be allocated to this area.

1.7 Speculative Read Function (Read Buffer Function)

The V850E/ME2 has a 4-word (128-bit) read buffer used for a speculative read function to enable high-speed CPU processing. The speculative timing can be set for each $\overline{CSn}$ space by line buffer control registers 0 and 1 (LBC0 and LBC1) (n = 0 to 7).

Caution Generally, use of the speculative read function is prohibited for units that are not located at contiguous addresses (such as I/O devices), or memory whose contents change asynchronously to the CPU (such as a dual-port memory that is written by another bus master).

1.8 Initialization Pins

The V850E/ME2 has initialization pins that set various operation modes.

1.8.1 MODE0 and MODE1 pins

The operation mode is specified according to the status of the MODE0 and MODE1 pins. In an application system, fix the specification of these pins and do not change them during operation. Operation is not guaranteed if these pins are changed during operation.

Table 1-1. Setting of Data Bus

MODE1	MODE0	Operating Mode	
L	L	Normal operation mode	32-bit data bus
L	H		16-bit data bus
Other than above		Setting prohibited	

Remark L: Low-level input
H: High-level input

1.8.2 PLLSEL, SSEL0, and SSEL1 pins

These input pins are set according to the frequency (F_X) input to the X1 and X2 pins. Set the PLLSEL, SSEL0, and SSEL1 pins in accordance with the value of $F_X \times 8 = f_X$ (main clock).

Table 1-2. Frequency List

Multiplication Factor	PLLSEL Pin	SSEL1 Pin	SSEL0 Pin	Input Frequency (MHz) (Target Value)	Main Clock (f_X) Frequency (MHz)
8	H	L	H	Setting prohibited	Setting prohibited
		H	L	10.00 to 10.19	80.00 to 81.59
		H	H	10.20 to 11.99	81.60 to 95.99
	L	L	L	12.00 to 14.39	96.00 to 115.19
		L	H	14.40 to 17.39	115.20 to 139.19
		H	L	17.40 to 18.75	139.20 to 150.00
		H	H	Setting prohibited	Setting prohibited

Caution The maximum value of f_{CLK} is 100 MHz in a 100 MHz product, 133 MHz in a 133 MHz product, and 150 MHz in a 150 MHz product.
The operation is not guaranteed if f_{CLK} (MAX.) < f_X.
Make sure that f_X does not exceed the guaranteed operating frequency of each product.

Remark L: Low-level input
H: High-level input

1.8.3 JIT0 and JIT1 pins

These input pins specify the frequency modulation rate (f_{DIT}) of SSCG output. The default values (after reset) of the ADJON and ADJ2 to ADJ0 bits of the SSCG control register (SSCGC) are changed as follows, depending on the setting of these pins.

Table 1-3. Default Values of SSCGC.ADJON and SSCGC.ADJ2 to SSCGC.ADJ0 Bits

JIT1 Pin	JIT0 Pin	Default Value			
		ADJON Bit	ADJ2 Bit	ADJ1 Bit	ADJ0 Bit
L	L	0	0	0	0
L	H	1	0	0	1
H	L	1	0	1	1
H	H	1	1	0	1

Remark L: Low-level input
H: High-level input

CHAPTER 2 USB BUS DRIVER

2.1 General

2.1.1 Overview

The USB bus driver is a sample program for the USB function controller that is incorporated in the V850E/ME2. It conforms to Universal Serial Bus Specification Revision 1.1 and operates on the embedded real-time control operating system RX850 Pro (conforms to the μITRON 3.0 specifications).

This sample program uses the control endpoint (endpoint 0) only. The vendor-specific class is defined as the class, and the driver performs enumeration processing (standard device request processing) when a USB device is connected.

This sample program uses the emulation board SolutionGear™ MINI (SG-703111-1) as the hardware execution environment. When using the SolutionGear MINI and sample program as is, create the execution object by following the procedure described in **2.6 Load Module** and confirm its operation by following the procedure described in **2.2 Execution of Load Module**.

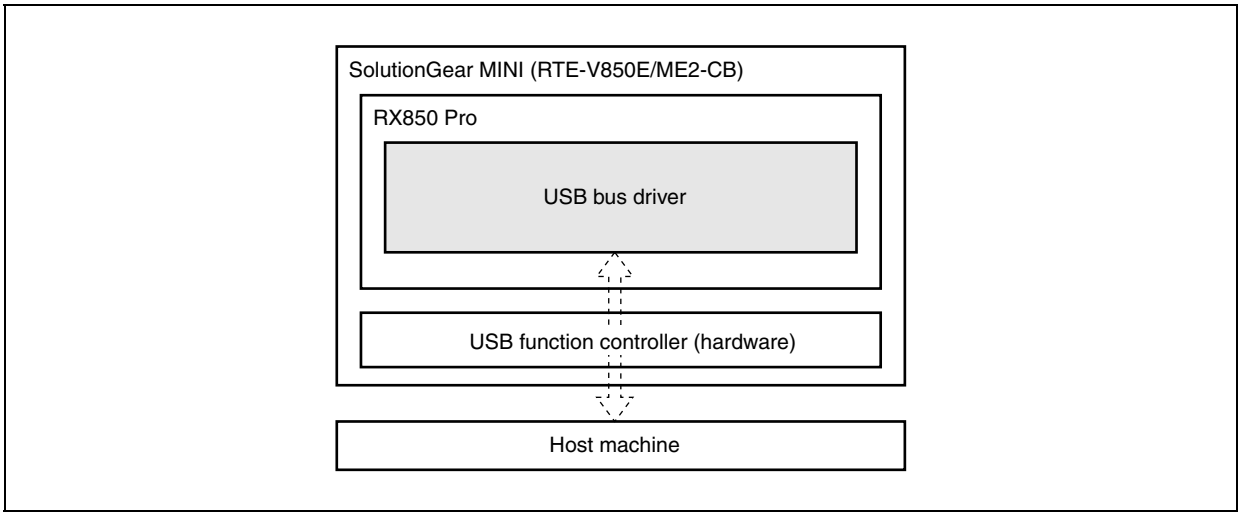
When using another target board instead of SolutionGear MINI, change the board referring to **2.3 System Configuration**, **2.4 RX850 Pro-Dependent Processing Modules**, and **2.5 Section Map File**, in accordance with the board specifications.

When changing both SolutionGear MINI and sample program, change them referring to **2.3 System Configuration**, **2.4 RX850 Pro-Dependent Processing Modules**, **2.5 Section Map File**, **2.6 Load Module**, and **2.7 USB Driver Functions**.

The positioning of the USB bus driver is shown below.

Remark The descriptions in **2.2.1 Execution procedure of load module** assume the user environment described in **2.1.3 Execution environment**.

Figure 2-1. Positioning of USB Bus Driver



2.1.2 Development environment

This section assumes the following hardware and software environments are used for system development using the sample program.

- Hardware environment
  Host machine: PC/AT™-compatible machines (OS: Windows™ XP)
- Software environment
  Real-time OS: RX850 Pro Version 3.15
  USB bus driver: Sample program set described in this section
  C compiler package: MULTI2000
  (CCV850 Version 3.5 (made by Green Hills Software, Inc.))

Caution If the directory configuration of the user environment differs from that handled in the build file of the sample program, adjust the build file to the user environment.

Remark Refer to the help of MULTI™ (made by Green Hills Software, Inc.) for the description of the build file.

2.1.3 Execution environment

This section assumes the following hardware and software environments are used for load module execution using the sample program.

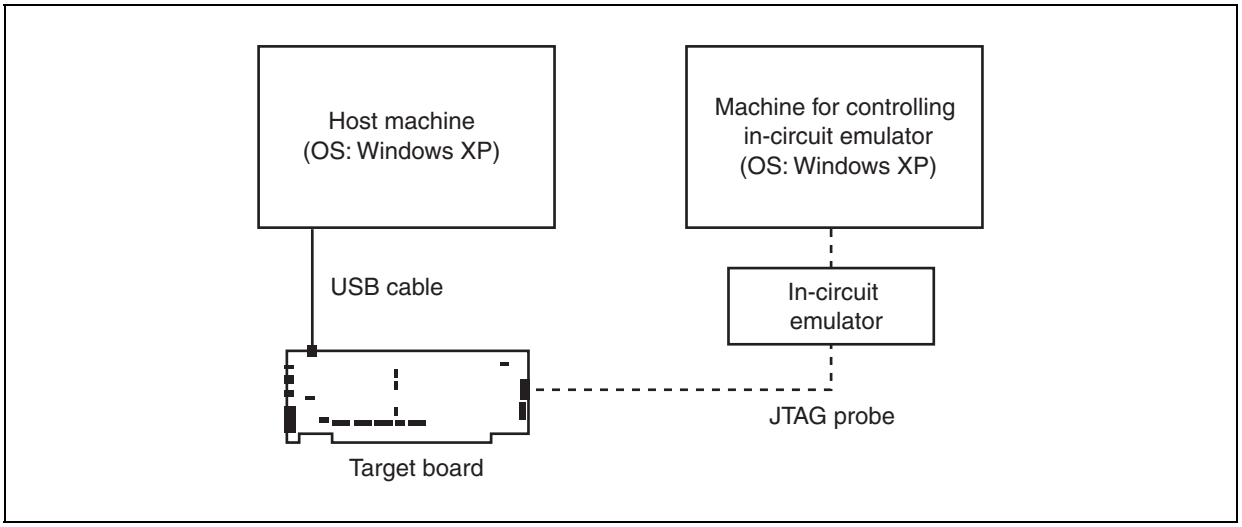
- Hardware environment
  - Host machine: PC/AT-compatible machines (OS: Windows XP)
  - IE control machine: PC/AT-compatible machines (OS: Windows XP)
  - Target board: SolutionGear MINI (SG-703111-1)
  - In-circuit emulator (IE): N-wire IE (RTE-2000-TP) (made by Midas Lab Inc.)
  - JTAG probe
  - USB cable
- Software environment
  - Software for IE: PARTNER Setup Program Version 1.242

Remarks 1. Refer to **APPENDIX A SG-703111-1 BOARD** and the **SG-703111-1 User's Manual** for details of how to set up the execution environment.

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

3. Refer to the **PARTNER User's Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for details of PARTNER.

Figure 2-2. Execution Environment



2.2 Execution of Load Module

2.2.1 Execution procedure of load module

The following shows the procedure for executing the load module under the environment described in **2.1.3 Execution environment**, taking the load module using the sample program as an example.

(1) Preparation of machine for controlling in-circuit emulator (IE)

Turn on the power and start up the IE control machine and the in-circuit emulator.

(2) Preparation of host machine

Turn on the power and start up the host machine (the IE control machine can be used as the host machine, but it is strongly recommended to provide an independent machine for development).

(3) Reset SG-703111-1 board

Press the RESET button of the SG-703111-1 board to reset the SG-703111-1 board.

(4) Startup of software for IE

Start up software for IE.

Select the [Start] button → “All Programs” → “PARTNER” → “RPTSETUP (NB85ET)” in Windows.

Click the [Open] button and specify a project file; the [Run] button is then selectable. Click the [Run] button to start up PARTNER. Make the board settings after startup. It is useful to create at this time the setting file loaded at startup. Refer to **APPENDIX A SG-703111-1 BOARD**, **PARTNER User’s Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for setup files for the sample described in this section.

Cautions 1. Be sure to apply power to the target board before starting up the in-circuit emulator.

2. If you want to load the setting file for resetting the target board after the in-circuit emulator is started up, load the setting file (init.mcr in the example below) by inputting a command to the command window, as shown below.

[Command input example]

```
><init.mcr<Enter>
```

(5) Loading the load module

Load the load module to the board using the in-circuit emulator function.

The load module (usb_bus.out in the example below) can be loaded by selecting [Load] in the [File] menu on the toolbar, or input the L command (loading file) in the command window.

[Command input example]

```
>l usb_bus.out<Enter>
```

(6) Execution

The code loaded to the board is executed by pressing the F5 key or the [Go] button.

Remark The same operation is performed by selecting [Go] in the [Run] menu on the toolbar.

(7) USB connection

Connect the USB cable.

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

Cautions 1. The USB cable can be connected before/after starting up the target board.

2. When the device is detected by the host machine, the software installation screen appears. Since no dedicated host driver is provided in this sample program, select the [Cancel] button here.

(8) Startup of Device Manager

Open the Properties window from My computer and select the Hardware tab. Select the Device Manager to start up the Device Manager.

Remark The Device Manager can also be started up from [Manage] menu of My computer or the Control Panel.

(9) Confirmation of USB device connection

Make sure that “USB Device” is displayed under “Other devices” in the Device Manager screen.

Caution The driver included in this sample program only performs processing up to enumeration. Therefore, the driver performs no more operations.

(10) Exiting program

Terminate the program under execution.

Click the forcible break button on the PARTNER screen, or select “Forcible Break” in the [Run] menu on the toolbar to stop program execution.

(11) Shutting down in-circuit emulator

Shut down the in-circuit emulator and reset the target board by following the procedure described in **(1)**.

Select [Exit] in the [File] menu on the toolbar to terminate PARTNER.

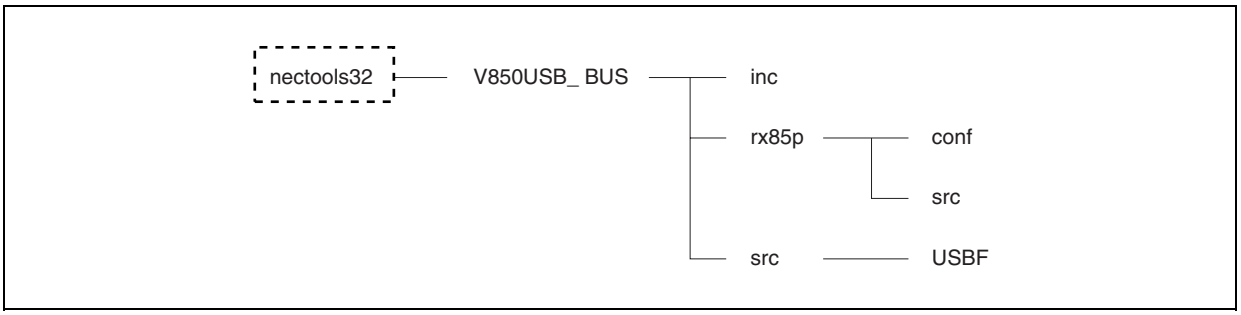
After terminating PARTNER, reset the target board by following the procedure described in **(1)**.

2.2.2 Directory configuration

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

Caution It is recommended to place the directory of the USB bus driver files directly under the directory where the RX850 Pro is installed (\nectools32).

Figure 2-3. Sample Program Directory Configuration



The outline of each directory is shown below.

(1) nectools32

A directory created when the RX850 Pro is installed. Place the directory (directory name: V850USB_BUS) of the driver directly under this directory.

(2) nectools32\V850USB_BUS

A directory for the USB bus driver.

- usb_bus.bld: Build file of USB bus driver
- common.lx: Section map file

(3) nectools32\V850USB_BUS\inc

A directory in which header files for the USB bus driver are stored.

- errno.h: Header file for return value
- types.h: Header file for data type
- sys.h: Header file for system information

Caution sys.h (header file for system information) is usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create this file because the command is automatically executed when build is executed.

(4) nectools32\V850USB_BUS\rx85p

A directory in which files for the RX850 Pro are stored.

(5) nectools32\V850USB_BUS\rx85p\conf

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

- sit.850: System information table
- svc.850: System call table
- sysi.tbl: System information table
- sysc.tbl: System call table

Cautions 1. Files in this directory are usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create these files because the command is automatically executed when build is executed.

2. sit.850 and sysi.tbl, svc.850 and sysc.tbl differ only in their file extension.

(6) nectools32\V850USB_BUS\rx85p\src

A directory in which files for RX850 Pro are stored.

- boot.850: Assembler file for boot processing
- entry.850: Assembler file for entry processing
- init.c: Source file for hardware initialization module
- init.h: Header file for hardware initialization module
- sys.cf: CF definition file
- varfunc.c: Source file for software initialization module

(7) nectools32\V850USB_BUS\src

A directory in which files of the USB bus driver board-dependent module are stored.

- port.c: Source file for port setting
- port.h: Header file for port setting

(8) nectools32\V850USB_BUS\src\USBF

A directory in which files of the USB bus driver USB processing module are stored.

- usbf850.c: Source file for USB bus driver
- usbf850.h: Header file for USB bus driver
- usbf850desc.h: USB bus driver descriptor definition file

2.3 System Configuration

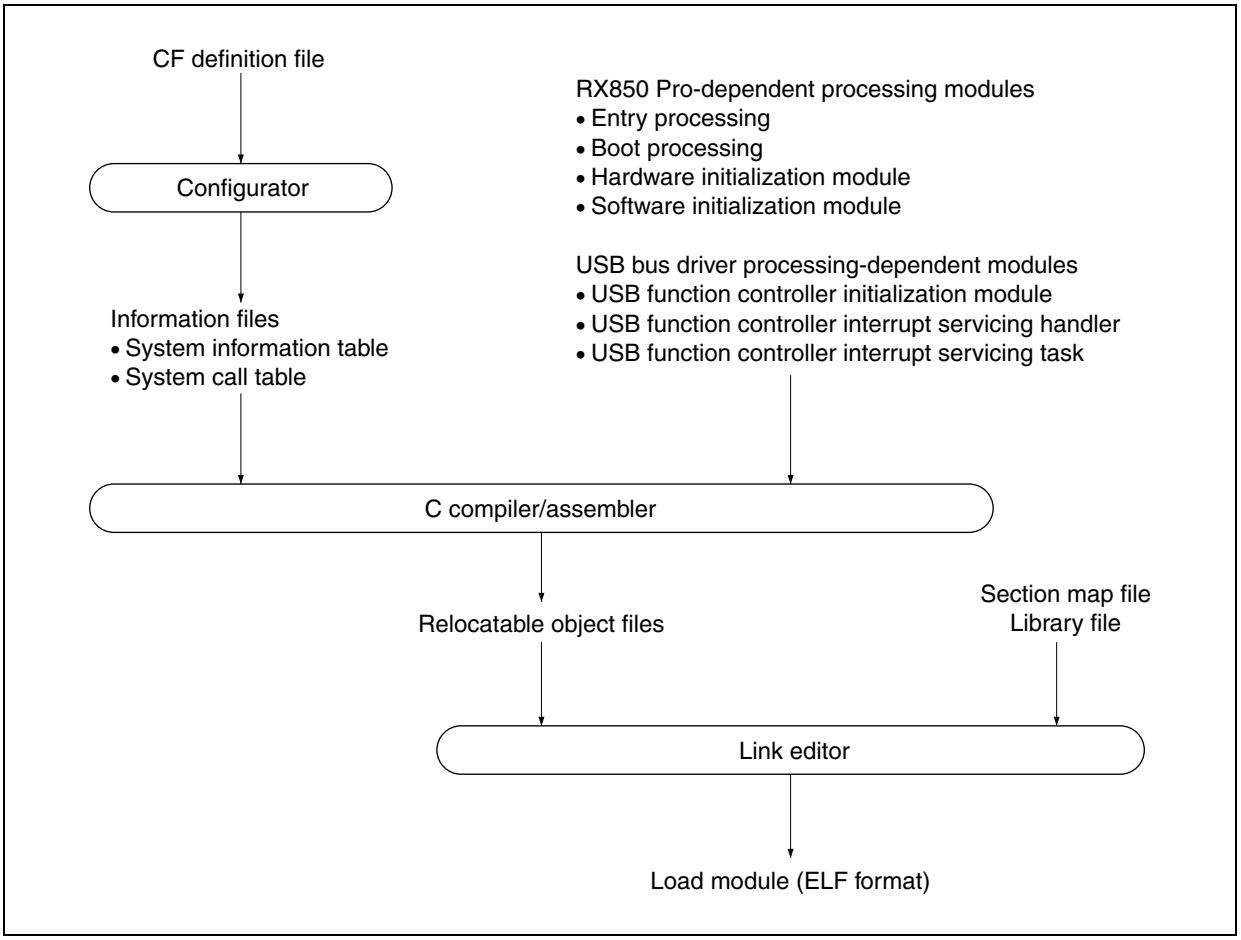
2.3.1 Overview

System configuration means creation of the load module using files that are installed in the user's development environment (the host machine) from the USB bus driver supply medium.

The system configuration procedure of USB bus drivers is shown below.

(1) Describing RX850 Pro-dependent processing module
(2) Describing board-dependent module
(3) Describing USB bus driver processing-dependent module
(4) Describing section map file
(5) Creating load module

Figure 2-4. System Configuration Procedure



2.3.2 Describing RX850 Pro-dependent processing module

Some functions provided by the USB bus driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
  - Boot processing
  - Hardware initialization module
  - Software initialization module

Remark Refer to **2.4 RX850 Pro-Dependent Processing Modules** for details of the RX850 Pro-dependent processing module.

2.3.3 Describing board-dependent module

The initialization processing, which is related to the processing dependent on the user's execution environment and application system, is provided as a board-dependent module in the USB bus driver source program.

The board-dependent module is as follows.

- CPU board-dependent module

  The port input/output manipulation required for the USB bus driver is provided as a CPU board-dependent module.

Caution Since port setting is handled in the same manner as setting of other registers, no dedicated function is provided.
Refer to the RX850 Pro standard header file *SFR.h* stored in \nectools32\inc850\common\ for the register definition. For detailed processing, refer to the source program for port setting (port.c) called from the boot processing module (boot.850) and software initialization module.

2.3.4 Describing USB bus driver processing-dependent module

The driver functions, which are used to implement the USB bus driver functions, are provided as the USB bus driver processing-dependent module in this sample program.

The USB bus driver processing-dependent modules are listed below.

- USB function controller initialization module
- USB function controller interrupt handlers
- USB function controller interrupt servicing tasks
- USB function controller general-purpose functions

Remark Refer to **2.7 USB Bus Driver Functions** for details of the USB bus driver processing-dependent module.

2.3.5 Describing section map file

The section map file is used by the user to fix address assignment performed by the link editor.

The following five text areas are essential sections when using the RX850 Pro.

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

Remark Refer to **2.5 Section Map File** for details of the section map file.

2.3.6 Creating load module

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing modules, USB bus driver processing-dependent module, and section map file, that have been coded.

Remark Refer to **2.6 Load Module** for details of how to create the load module.

2.4 RX850 Pro-Dependent Processing Modules

2.4.1 Overview

Some functions provided by the USB bus driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
  - Boot processing
  - Hardware initialization module
  - Software initialization module

2.4.2 CF definition file

An information file (CF definition file) that contains data provided to the RX850 Pro is required to configure the system in which the RX850 Pro is used.

The following information is required for using the USB bus driver function.

○ Real-time OS information
  - RX Series information
○ SIT information
  - System information
  - System maximum value information
  - System memory information
  - Task information
  - Interrupt handler information
  - Initialization handler information
○ SCT information
  - Task management/task-associated synchronization system call information
  - Interrupt servicing management system call information
  - Time management system call information

Caution This sample program implements each functions using three tasks, three interrupt handlers, and seven system calls. Therefore, the CF definition file, the maximum number of tasks to be created must be set to three as the system's maximum value information and the maximum number of interrupt handlers to be created must be set to three for the USB bus driver and use of *sta_tsk*, *ext_tsk*, *slp_tsk*, and *wup_tsk* system calls must be defined as task management/task-associated synchronization system call information, use of the *loc_cpu* and *unl_cpu* system calls as interrupt servicing management system call information, and use of the *dly_tsk* system call as time management function system call information.

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

(1) Procedure for creating information files

A procedure for creating information files (system information table, system call table, and system information header file) is shown below.

The information file can be created from the Windows command prompt.

Caution If a build file in the sample program is used, users are not required to create information files in this procedure because they are automatically executed when build is executed.

<1> Change current directory

Move the current directory to the directory in which the CF definition file is stored using the cd command of Windows.

A command 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> Creating information files

Create the information file from the CF definition file that has been created in the specific description format, using the configurator cf850pro.exe.

A command input example when creating three information files (system information table: sit.850, system call table: svc.850, and system information header file: sys.h) from an input file (CF definition file name: sys.cf) is shown below.

[Command input example]

```
C:>cf850pro -i sit.850 -c svc.850 -d sys.h sys.cf<Enter>
```

The information files are created from the CF definition file.

Caution A sample file (CF definition file) used for creating the information files is provided in the sample program.

Remark Refer to the **RX850 Pro Installation User's Manual** for details of the option to activate the configurator cf850pro.exe and execution method.

2.4.3 Entry processing

This processing assigns a branch instruction to an interrupt handler to the handler address where control is forcibly passed by the processor when a maskable interrupt occurs.

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

Remark Refer to sample program *entry.850* for details of how to code the entry processing.

2.4.4 System initialization processing

The system initialization processing includes initialization processing (boot processing and hardware initialization module) of hardware required for operating the RX850 Pro normally, and software initialization processing (nucleus initialization module and Initialization handler).

The system initialization processing is performed first when the system is activated.

Caution Among the four types of system initialization processing, users are not required to describe the nucleus initialization module because it is a function provided by the RX850 Pro.

The processing performed by the nucleus initialization module is shown below.

○ Securement of system memory defined by CF definition file
- System pool 0
- User pool 0

○ Generation and activation of management object defined by CF definition file
- Generation and activation of task
- Registration of interrupt handler

○ Activation of initial task
○ Generation and activation of idle task
○ Calling software initialization module
○ Passing control to scheduler

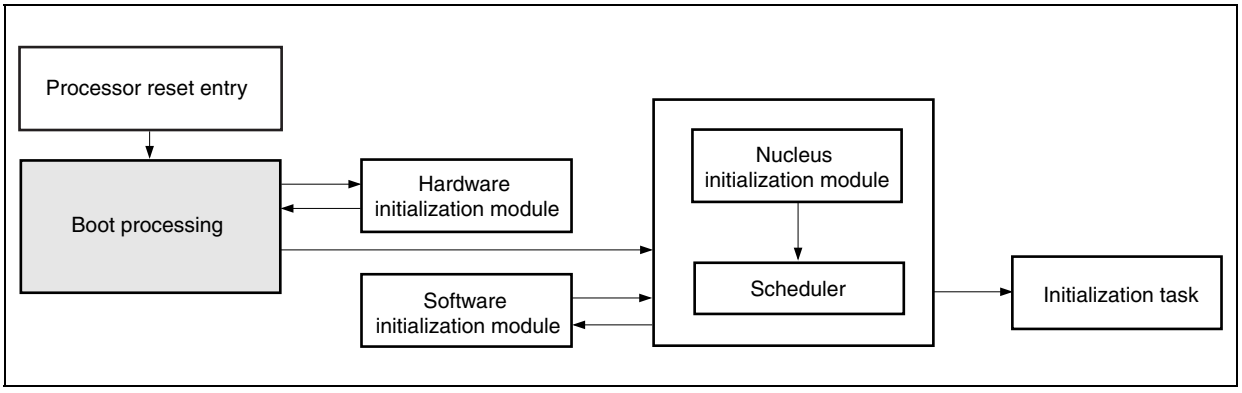
The idle task is a processing routine that is activated by the scheduler when a processing module (task) executed by the RX850 Pro is no longer in the run or ready state, that is, no processing module targeted to the scheduling by the RX850 Pro exist in the system. The idle task issues the HALT instruction.

(1) Boot processing

Boot processing is the function assigned to the processor reset entry, so it is executed first in the system initialization processing.

The positioning of boot processing is shown below.

Figure 2-5. Positioning of Boot Processing



The processing executed by boot processing is shown below.

Remark Refer to sample program *boot.850* for details of how to code boot processing.

- Setting tp, gp, and ep registers
  Values of the text pointer tp, global pointer gp, and stack pointer ep, which are required for execution of each processing module (including boot processing), are undefined when a system is activated. Boot processing first performs initial setting of these registers.

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

- Calling hardware initialization module
  Functions (hardware initialization module) are called to initialize the hardware on the target system.
  This step is not required if initialization of internal units is performed by other module.

  Caution In this chapter, this step is not required because initialization of internal units is performed by the software initialization module. Refer to the RX850 Pro Installation User's Manual for details.

- Passing control to nucleus initialization module
  The nucleus initialization module secures the system memory (system pool 0, user pool 0) and creation/initialization of management objects, based on information described in the system information table. Therefore, start address_sit of the system information table must be set to the r10 register before passing control to the nucleus initialization module.

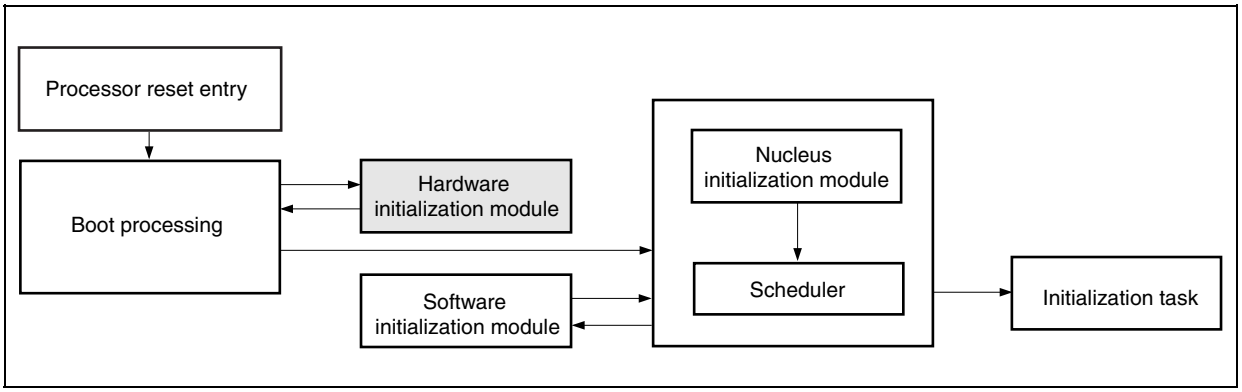
  Caution The system information table is a table in which the CF definition file created in a specific description format is converted to the assembly language format, using the utility tool (configurator cf850pro.exe) provided by the RX850 Pro.

(2) Hardware initialization module

The hardware initialization module is a function to initialize the hardware on the target system, and is called from boot processing.

The positioning of the hardware initialization module is shown below.

Figure 2-6. Positioning of Hardware Initialization Module



The processing executed by the hardware initialization module is shown below.

Cautions 1. Users are not required to disable the maskable interrupts because they are masked at initialization by default.

2. Hardware initialization is performed by the software initialization module in the sample program. Refer to the RX850 Pro Installation User's Manual for details of the hardware initialization module.

- Returning control to boot processing

  Control can be returned from the hardware initialization module to boot processing by issuing the "return( );" instruction, because the return address to the lp register is set when the hardware initialization module is called from boot processing.

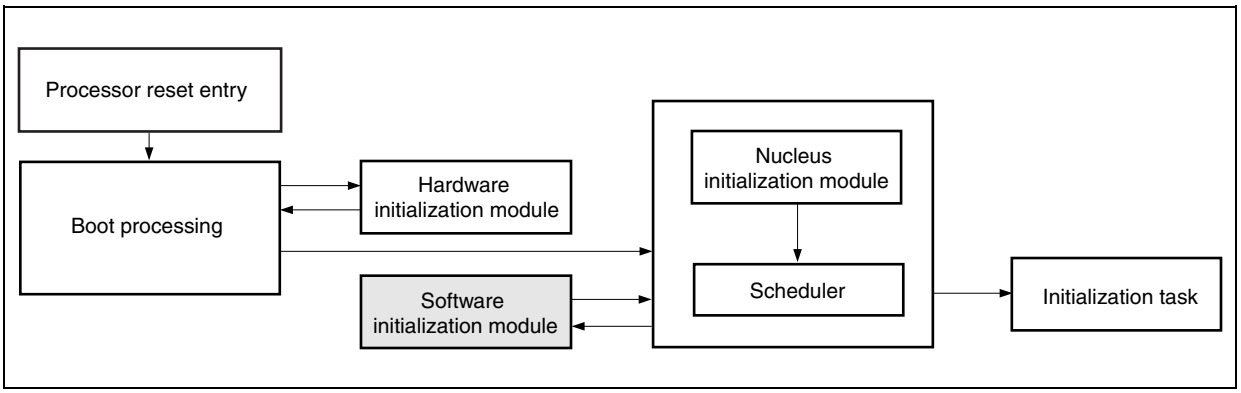
  If the hardware initialization module is described with the assembly language, this processing is implemented by issuing the "jmp [lp]" instruction.

(3) Software initialization module

The initialization handler is a function provided to enhance operability of the user software environment, and is called from the nucleus initialization module.

The positioning of the software initialization module is shown below.

Figure 2-7. Positioning of Software Initialization Module



The processing executed by the software initialization module is shown below.

Remark Refer to sample program *varfunc.c* for how to code the software initialization module.

- Initialization of internal unit (real-time pulse unit (RPU))

  The RX850 Pro implements the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) using the timer interrupt that occurs in a constant cycle. Therefore, the real-time pulse unit must be initialized before the RX850 Pro starts processing.

  The compare register CMD0 included in the real-time pulse unit must be set so that timer interrupts occur in a base clock cycle defined in system information in the CF definition file.

- Enabling timer interrupt acknowledgment

  Acknowledgment of timer interrupts is enabled. In addition, this enables the use of the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) provided by the RX850 Pro when processing by the nucleus initialization module ends.

- Passing control to nucleus initialization module

  Control can be returned from the initialization handler to the nucleus initialization module by issuing the “return( );” instruction, because the return address lp register is set when the initialization handler is called from the nucleus initialization module.

  If the initialization handler is described with the assembly language, this processing is implemented by issuing the “jmp [lp]” instruction.

2.4.5 Time management function

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

The RX850 Pro calls system clock processing when a clock interrupt occurs, and performs processing related to the time such as updating the system clock, task delay wake-up, and activation of the cyclic handler.

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

After the system clock is set to “0H” by system initialization processing, it is updated by system clock processing in base clock cycle units (specified at configuration).

Caution The system clock managed by the RX850 Pro is configured as 48 bits wide. Therefore, overflowed numeric values (numeric values that cannot be expressed by 48 bits) are ignored by the RX850 Pro. Refer to the RX850 Pro Basics User’s Manual for details of the time management function of the RX850 Pro.

2.5 Section Map File

2.5.1 Overview

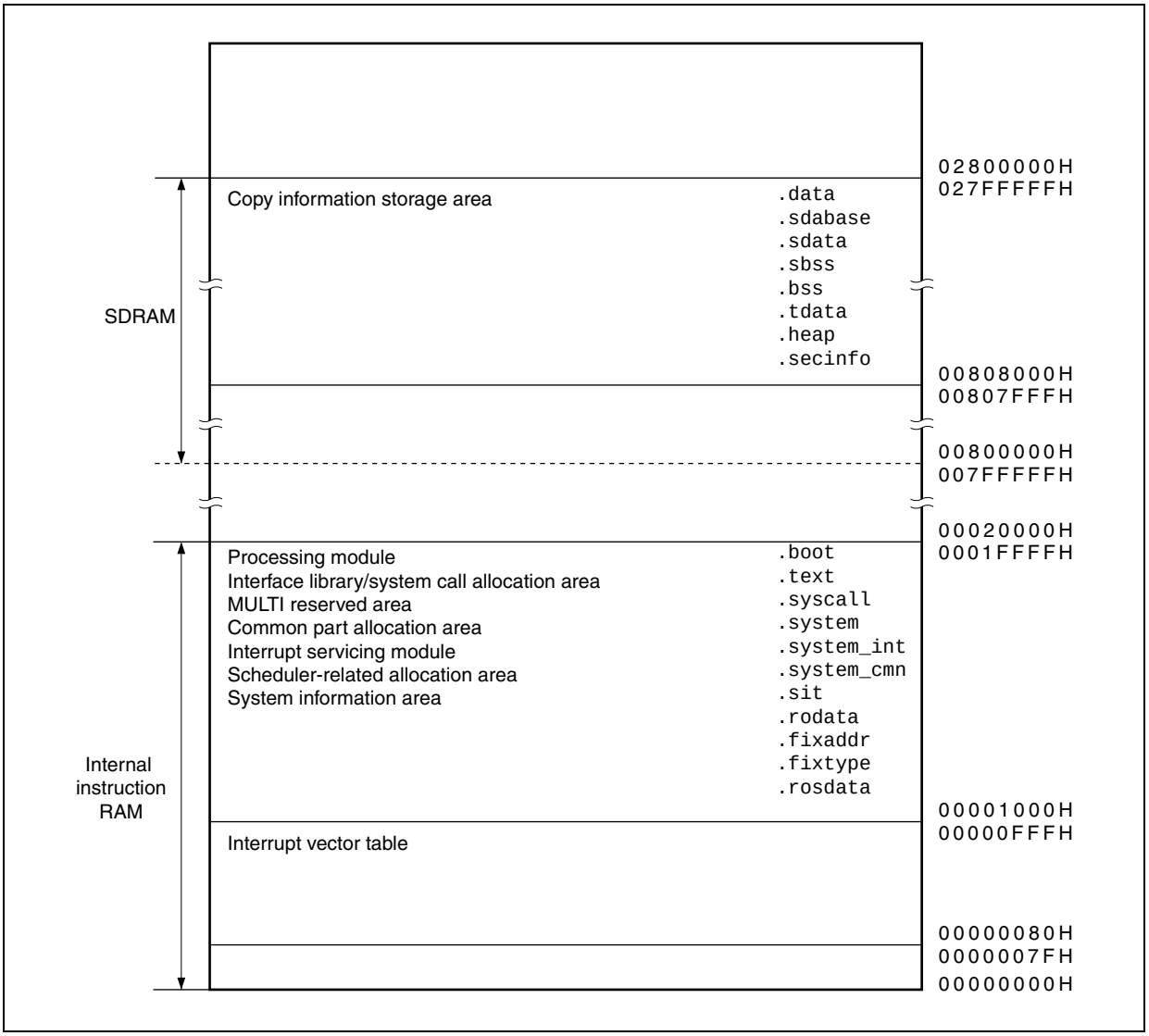
The section map file is used by the user to fix address assignment performed by the link editor.

Required assignments for addresses other than the user processing program (such as .data and .bss sections) are described in **2.5.2 Address assignment by RX850 Pro** and **2.5.3 Other address assignment**.

Address assignment performed in sample program *common.lx* is shown below.

Remark Refer to sample program *common.lx* for how to code the section map file.

Figure 2-8. Address Assignment Example



2.5.2 Address assignment by RX850 Pro

The RX850 Pro consists of five text areas: common part allocation area, interrupt servicing-related allocation area, scheduler-related allocation area, system information area, and interface library/system call allocation area. Using these areas, memory areas for which a large space is required can be assigned to the external RAM, and memory areas for which a high-speed access is required (interrupt servicing module, scheduling processing module) can be assigned to the internal instruction RAM (00000000H to 0001FFFFH).

Caution All five text areas are allocated to the internal instruction RAM in the sample program.

- Common part allocation area (.system section)
  Processing of the RX850 Pro (such as task management function, task-associated synchronization function) is assigned to this area.

- Interrupt servicing-related allocation area (.system_int section)
  Among the interrupt servicing management functions provided by the RX850 Pro, interrupt preprocessing that is performed when control is passed to the interrupt handler and interrupt postprocessing that is performed when control is handed back to the processing module in which a maskable interrupt occurs are assigned to this area.
  By assigning the interrupt servicing module to the internal instruction RAM, therefore, response performance to the interrupt handler can be improved.

  Caution It is recommended to assign the interrupt servicing module to the internal instruction RAM.

- Scheduler-related allocation area (.system_cmn section)
  Among the scheduling function provided by the RX850 Pro, task wake-up processing and task scheduling processing are assigned to this area.
  By assigning the scheduling processing section to the internal instruction RAM, therefore, task wake-up processing and task scheduling processing are accelerated, as well as system call processing involving scheduling processing.

  Caution It is recommended to assign the scheduling module to the internal instruction RAM.

- System information area (.sit section)
  The system information table created by executing the configurator cf850.exe on the CF definition file is assigned to this area.
  The system information table includes various data required for executing the nucleus initialization module (securement of the system memory and creation/initialization of management objects).

- Interface library/system call allocation area (.text section)
  The instructions including system calls are assigned to this area.

- System memory
  Various management block required for implementing functions provided by the RX850 Pro (such as the task management block, semaphore management block), area in which the stack used by the interrupt handler or task is assigned (system pool 0), and area in which dynamic memory manipulation (such as acquisition/release of memory blocks) from the processing module is enabled (user pool 0), are assigned to this area.

Cautions 1. The ”system memory start address” must be specified when creating the CF definition file. Be sure to specify the address when defining the system memory in the section map file.
2. The user can specify any section name in the system memory.

2.5.3 Other address assignment

The other sections for which address assignment is required are described below.

- MULTI reserved area (.syscall section)
  This area is used as a work area by the debugger MULTI (made by Green Hills Software, Inc.).

Cautions 1. The .syscall section must be defined regardless of whether or not MULTI is used.
2. Be sure to specify 4-byte alignment when defining the .syscall section.

- Copy information storage area (.secinfo section)
  This area is used by the link editor to output information (start address, size) required for transferring program (data, text) of a section for which the ROM identifier is specified in the section map file from ROM to RAM.
  Specification of the ROM identifier is required when performing ROMization of a processing module. Therefore, definition of the .secinfo section is not required when ROMization is not performed.

Caution This section is empty in the sample program because ROM identifier specification is not performed.

2.6 Load Module

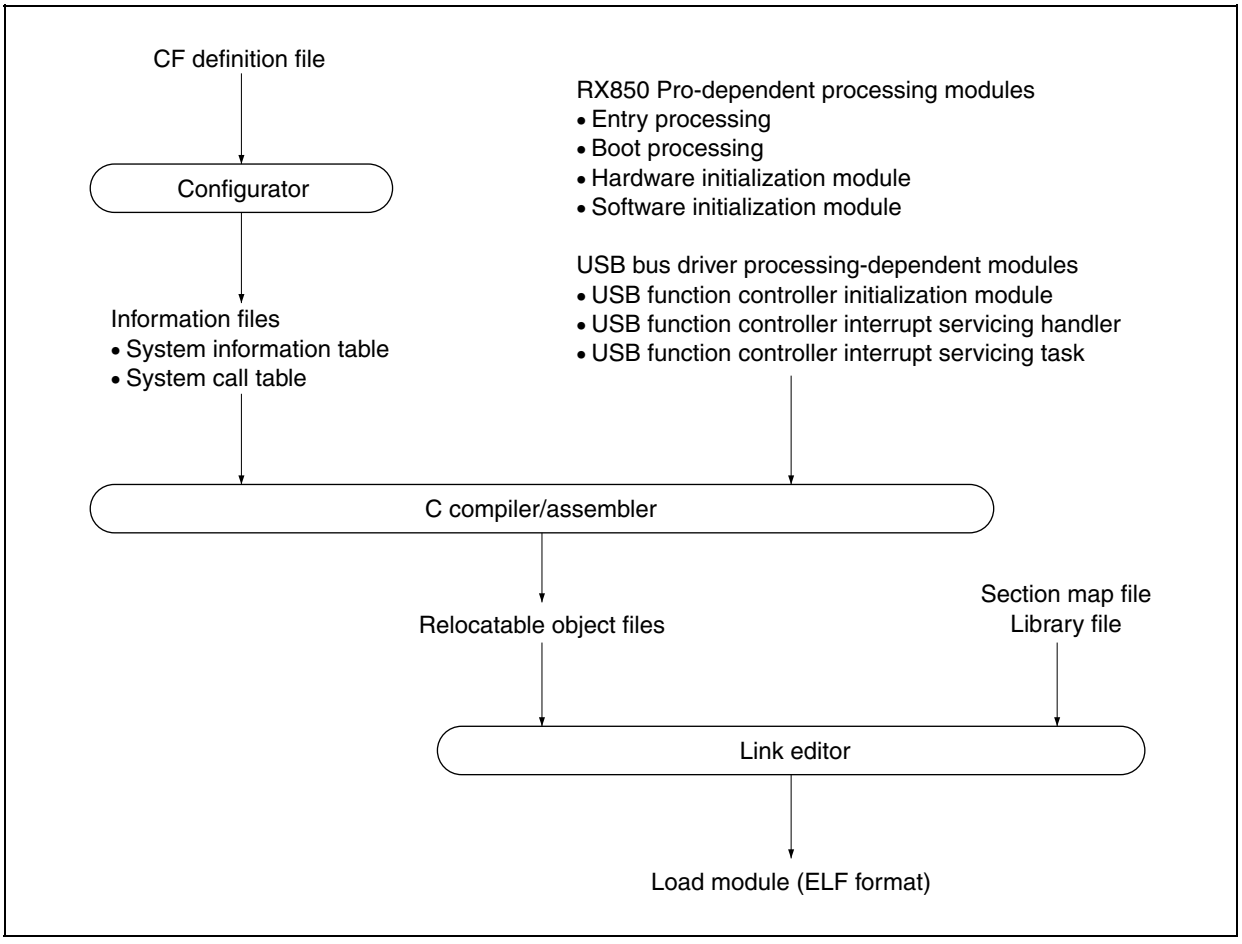
2.6.1 Overview

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing module, USB bus driver processing-dependent module, section map file, that have been coded.

The procedure for creating load modules is shown below.

Caution The load module corresponding to the sample program can be created by executing the .bld file in the sample program. However, definition of the .bld file must be adjusted to the user development environment.

Figure 2-9. Load Module Creation Procedure



2.6.2 Creating load module

An ELF-format load module can be created from the RX850 Pro-dependent processing module, USB bus driver processing-dependent module, and section map file, that have been coded, using the following procedure.

(1) Creation of system information table and system call table

Original CF definition file formats are excluded from the link processing performed by the link editor when creating a load module.

Therefore, a file that can be assembled (system information table or system call table) must be created using the utility tool (configurator cf850.exe) provided by the RX850 Pro.

Remark Refer to **2.4.2 (1) Procedure for creating information file** for how to create the system information table and system call table.

(2) Creation of object file

A relocatable object file is created by executing the C compiler/assembler for the processing module (file described in the C language/assembly language) shown below.

○ RX850 Pro-dependent processing module
- System information table
- System call table
- Entry processing
- Boot processing
- Hardware initialization module
- Initialization handler

○ USB bus driver processing-dependent module

(3) Creation of load module

An ELF-format load module is created by executing the link editor for relocatable object file created in **(2)**, library files, and section map file.

libansi.a	ANSI C library
libind.a	C library made by Green Hills Software, Inc. (routines independent of target CPU)
libarch.a	C library made by Green Hills Software, Inc. (routines dependent of target CPU)
libsys.a	C library made by Green Hills Software, Inc. (system call, initialization routines)

rxcore.o	Nucleus common part object
librxp.a	Nucleus library
libchp.a	Interface library

rxcore.o, *librxp.a*, and *libchp.a* are provided by the RX850 Pro, and *libansi.a*, *libind.a*, *libarch.a*, and *libsys.a* are provided by the CCV850 (made by Green Hills Software, Inc.).

2.7 USB Bus Driver Functions

2.7.1 Overview

Initialization processing performed by the USB function controller, as well as tasks and interrupt handlers to implement USB bus driver processing, must be described in the USB bus driver.

A list of USB bus driver processing-dependent modules is shown below.

- USB function controller initialization processing
  This module is called from the RX850 Pro software initialization module and initializes the USB function controller.

- USB function controller interrupt handlers
  This is an interrupt servicing-dedicated routine that is called each time an interrupt by the USB function controller occurs, and is defined in the CF definition file.

  Caution Interrupts other than required are masked in this sample program.
  Only the CPUDEC interrupt reported by the INTUSB0B signal (which indicates that there is a request that is decoded by FW in the UF0E0ST register) is used in this sample program.

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

- USB function controller general-purpose function
  This is a general-purpose function used by the USB bus driver to perform the STALL response setting for each endpoint and transmission/reception processing.

  Remark Refer to sample program *usbf850.c* for how to code the USB bus driver processing-dependent module.

- USB suspend/resume processing
  Since the USB suspend/resume processing depends on the system, it is not supported in this sample program. If this processing is necessary in your system, add the processing making allowances for the following points.
  The suspend/resume state is reported to the USB function controller incorporated in the V850E/ME2 by an interrupt (INTUSB0B signal). Therefore, whether the current status is suspend or resume can be judged by checking the UF0IS0. RSUSPD bit in the interrupt handler (for the INTUSB0B signal); if this bit is 1, the UF0EPS1.RSUM bit is checked to judge the status.
  Processing can be added by adding the above code to judge the status to the interrupt handler (for the INTUSB0B signal) and wakes up a task to perform necessary processing from the code.

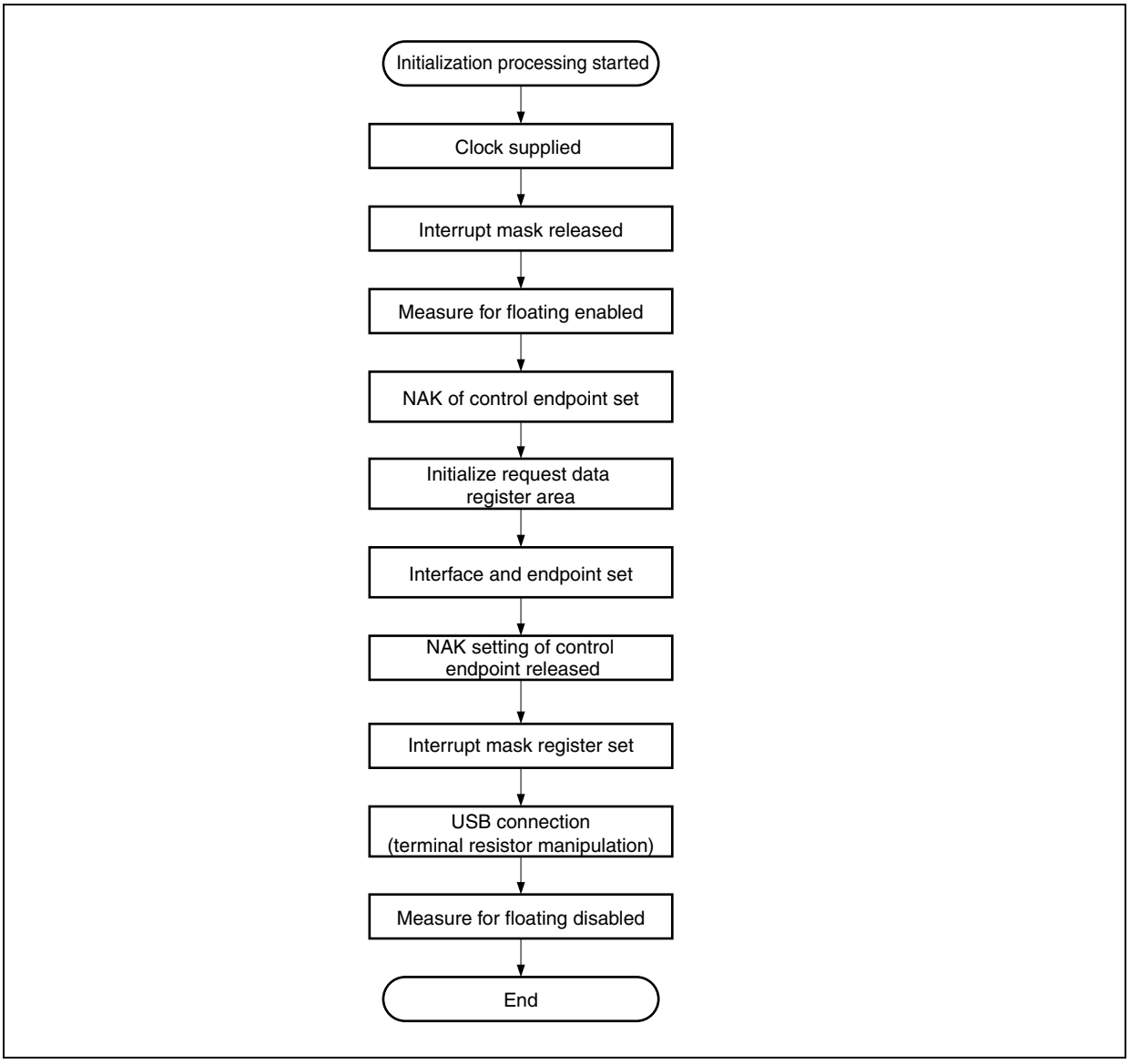
2.7.2 Processing flows

The processing flows of initialization processing and interrupt servicing in the sample program are shown below.

(1) Initialization processing

Initialization processing of the USB device is called and executed by the software initialization module.
The flow of USB device initialization processing (at power application) in the sample program is shown below.

Figure 2-10. Flowchart of Initialization Processing



The processing executed by the initialization processing is shown below.

Caution Initialization processing is required except for processing of ports. The pin assignment may differ if another target board is used. In such a case, read the descriptions in this manual making changes as necessary to match the specifications of the target board to be used.

- Clock supply
  Be sure to set the UCKC.UCKCNT bit to 1 before setting the USB function controller register. A clock to USB is supplied by setting this bit to 1.
  The P10 pin is used for inputting a clock, so set the P10 pin to input mode to enable clock input.

- Release of interrupt mask
  Masking of the USB-related interrupt signal is released using the interrupt control register.

- Enabling floating measure
  The UF0BC.UBFIOR bit is cleared to 0 to prevent mis-recognition due to a bus reset caused by an undefined value when the cable is disconnected.

- Setting of NAK for control endpoint
  A NAK response is sent to all the requests including automatic execution requests.
  This setting is made so that hardware does not return unexpected data in response to an automatic execution request until registration of data used for the automatic execution request is complete.

- Initialization of request data register area
  Descriptor data used to respond to a Get Descriptor request is registered in a register.
  Data such as device status, endpoint 0 status, device descriptor, configuration descriptor, interface descriptor, and endpoint descriptor are registered.

  Caution Registration of the descriptor for the class may be required depending on the class. The vendor-specific class is defined in this sample program, and only the USB standard descriptor is used.

- Setting of interface and endpoint
  Information such as the number of supported interfaces, the state of alternative settings, relationship between the interface and endpoints are set to a register.

- Release of NAK setting at control endpoint
  The NAK setting at control endpoint (endpoint 0) is released when registration of data for an automatic execution request is complete.

- Setting of interrupt mask register
  Masking for each interrupt source shown in the interrupt status register of the USB function controller.

- USB connection (terminal resistor manipulation)
  The D+ signal is pulled up.

- Disabling floating measure
  The floating measure is disabled by setting the UF0BC.UBFIOR bit to 1.

(2) Interrupt servicing

The sample program operates by interrupt events after initialization. The device is in the idle state as long as no event occurs.

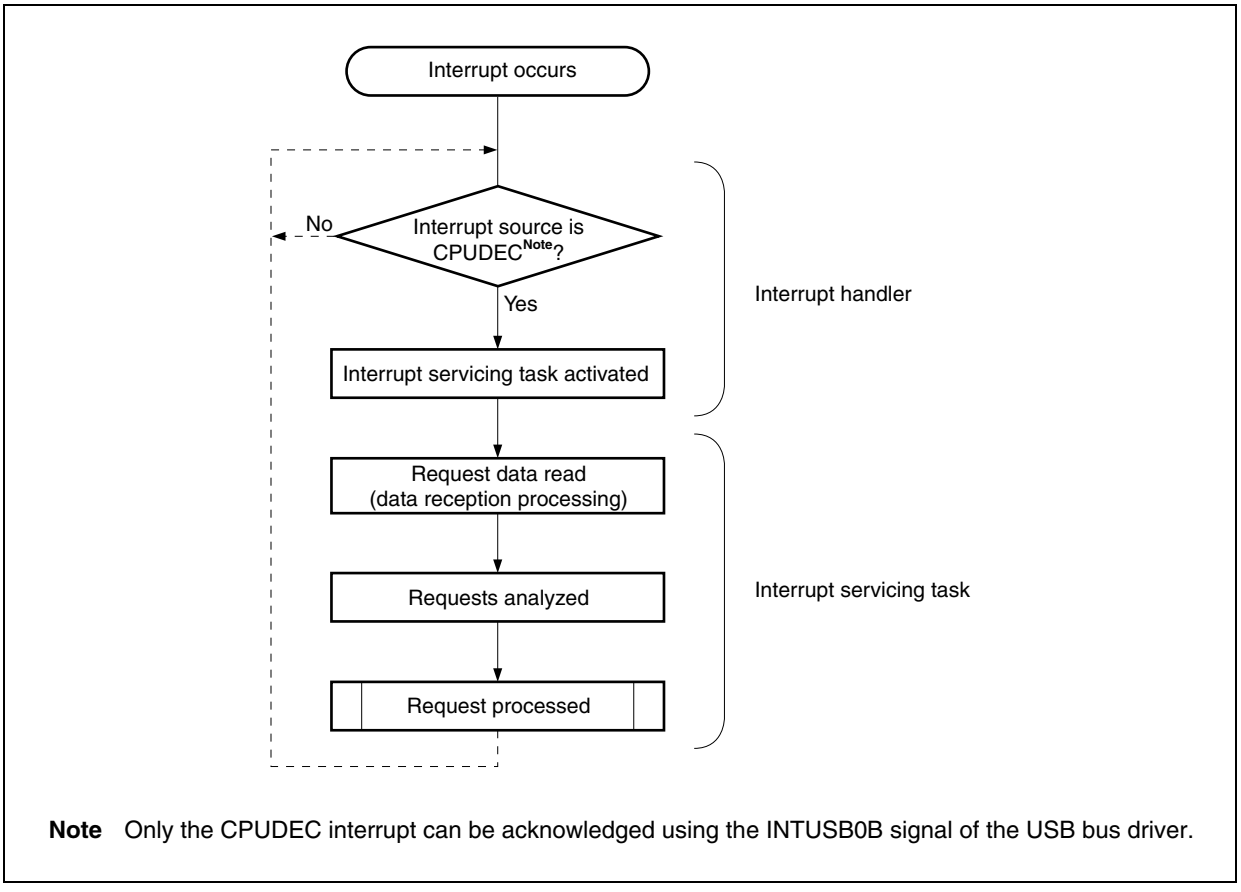
The following shows the interrupt servicing flow in the sample program.

Cautions 1. No dedicated host driver is provided in this sample program, so the driver only performs processing up to enumeration of the USB (device inquiry). Since the host only issues an automatic response request via the USB function controller incorporated in the V850E/ME2, the device is always in the idle state.

2. The flowchart in Figure 2-11 illustrates processing of the Get Descriptor (String Descriptor) request, to which the USB function controller does not respond automatically, and the device class-specific request.

Refer to CHAPTER 3 USB STORAGE CLASS DRIVER and CHAPTER 4 USB COMMUNICATION CLASS DRIVER for the transmit/receive processing at the bulk endpoint.

Figure 2-11. Flowchart of Interrupt Servicing



Note Only the CPUDEC interrupt can be acknowledged using the INTUSB0B signal of the USB bus driver.

The processing of an interrupt is shown below.

- Confirmation of interrupt source
  The analyzed interrupt status varies depending on the executed interrupt handler.
  Since only the CPUDEC interrupt can be acknowledged in this sample program, the interrupt handler is activated by the INTUSB0B signal. This interrupt handler reads the UF0IS1 register and judges if the interrupt source is CPUDEC interrupt or not.

  Caution In this sample program, the interrupt handlers to be used are registered in the CF definition file in advance.

- Activation of interrupt servicing task
  The *task_usb0b* task is activated if the interrupt source is CPUDEC.

  Caution In this sample program, the tasks to be activated are registered in the CF definition file in advance.

- Reading request data
  SETUP data is read from the UF0E0ST register.

- Analysis of request
  SETUP data that has been read is analyzed and the purpose of the request is confirmed.

- Processing of requests
  Processing of the analyzed request is performed.
  In the sample program, only the standard device request Get Descriptor (String Descriptor) is handled.

2.7.3 USB bus driver descriptor information

The USB standard descriptors defined in this sample program are shown below.
Descriptors described in (a) to (d) are the minimum required descriptors.

Remark Refer to **Universal Serial Bus Specification Revision 1.1** for details.

(a) Device descriptor

This descriptor holds general information of the device. One device descriptor must be prepared for each device. The information contained in this descriptor is used for identifying a unique in the device configuration. In the sample program, the driver performs enumeration processing (standard device request processing) when a USB device is connected. The vendor-specific class is defined as the class.

Table 2-1. Device Descriptor

Offset	Size (Byte)	Value	Description
0	1	12H	Length value of this descriptor (byte)
1	1	01H	Descriptor type (device)
2	2	10H/01H	USB version (USB 1.1)
4	1	FFH	Class code (vendor-specific class)
5	1	00H	Sub-class code
6	1	00H	Protocol code
7	1	40H	Maximum packet size at endpoint 0
8	2	09H/04H	Vendor ID (NEC Electronics)
10	2	FBH/FFH	Product ID
12	2	01H/00H	Device release number
14	1	01H	Index to string descriptor (Manufacturer)
15	1	00H	Index to string descriptor (Product)
16	1	00H	Index to string descriptor (Serial Number)
17	1	01H	Number of devices that can be configured

(b) Configuration descriptor

This descriptor holds information on concrete device configuration.

Table 2-2. Configuration Descriptor

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	02H	Descriptor type (configuration)
2	2	12H/00H	Total length value of descriptor returned together with configuration descriptor in response to the Get Descriptor request
4	1	01H	Number of interfaces supported in the configuration
5	1	01H	Configuration value
6	1	00H	Index to string descriptor (configuration)
7	1	C0H	Configuration of device (self-powered/remote wakeup function)
8	1	00H	Maximum power consumption of device

(c) Interface descriptor

This descriptor holds concrete interface information in the configuration.

The configuration provides one interface in this sample program.

This descriptor is always returned as a part of the configuration descriptor, and is not accessed directly by a Get Descriptor request or Set Descriptor request.

Table 2-3. Interface Descriptor

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	04H	Descriptor type (interface)
2	1	00H	Interface value
3	1	00H	*Alternate* set value
4	1	00H	Endpoint number (excluding endpoint 0)
5	1	FFH	Interface class (vendor-specific class)
6	1	00H	Interface sub-class
7	1	00H	Interface protocol
8	1	00H	Index to string descriptor (interface)

(d) Endpoint descriptor

This descriptor holds information required by the host for determining the bandwidth requirements for each endpoint.

This descriptor is always returned as a part of the configuration information by a Get Descriptor (configuration) request.

This descriptor is not accessed directly by a Get Descriptor request or Set Descriptor request.

Caution Since the driver in this sample program only performs processing up to enumeration, only the control endpoint is used. Therefore, the endpoint descriptor is not defined in the sample program.

(e) String descriptor

This descriptor holds information required by the host for determining the bandwidth requirement for each endpoint.

Table 2-4. String Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	04H	Length value of this descriptor (byte)
1	1	03H	Descriptor type (string)
2	2	09H/04H	Language type used by string descriptor (English/US)

Table 2-5. String Descriptor (2)

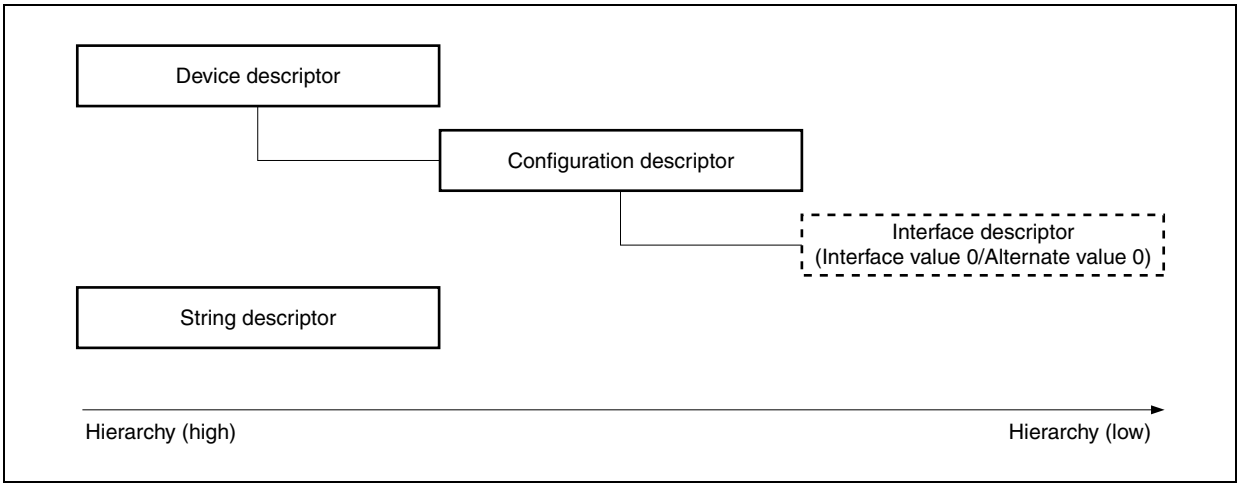
Offset	Size (Byte)	Value	Description
0	1	2AH	Length value of this descriptor (byte)
1	1	03H	Descriptor type (string)
2	40	'N','E','C','','E','l','e','c','t','r','o','n','i','c','s',' ','C','o','.'	Manufacturer: NEC Electronics Co.

- Descriptor configuration

  The descriptor configuration in this sample program is shown below. This configuration consists of the five descriptors described before.

Caution The device descriptor, configuration descriptor, and string descriptor are accessed by an independent Get Descriptor request. The interface descriptor is accessed as part of the configuration descriptor.

Figure 2-12. Descriptor Configuration



2.7.4 Data macro

The data macros (data type, return value, etc.) used by the USB bus driver are shown below.

(1) Data type

Data type macro is defined in the header file *types.h* in nectools32\USB_BUS\inc.
A list of the data types is shown below.

Caution No special data type is used in the sample program.

(2) Return value

Macro of the return value is defined in the header file *errno.h* in nectools32\USB_BUS\inc.
A list of the return values is shown below.

Caution No special data type is used in the sample program.

Table 2-6. List of Return Values

Macro	Type	Description
DEV_OK	0	Normal termination
DEV_ERROR	−1	Abnormal termination

2.7.5 Data structure

The data structure used by the USB bus driver is shown below.

- USB device request structure

  The USB device request structure is defined in USB header file *usbf850.h* in nectools32\V850USB_BUS\src \USBF. 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;
```

2.7.6 Description of functions

(1) Overview

A list of the processing modules described in this chapter is shown below.

Table 2-7. List of Processing Modules in Sample Program (1/2)

Processing Module Name	Function Name	File Name	Remark
RX850 Pro-dependent processing module			
CF definition file	–	sys.cf	–
Entry processing	–	entry.850	Assembly language
Boot processing	boot	boot.850	Assembly language
Hardware initialization module	__InitSystemTimer	init.c	C language
Initialization handler	varfunc	varfunc.c	C language
Header file	–	init.h	–
Board-dependent processing module			
Port initialization	port850_reset	port.c	C language
Header file	–	port.h	–

Table 2-7. List of Processing Modules in Sample Program (2/2)

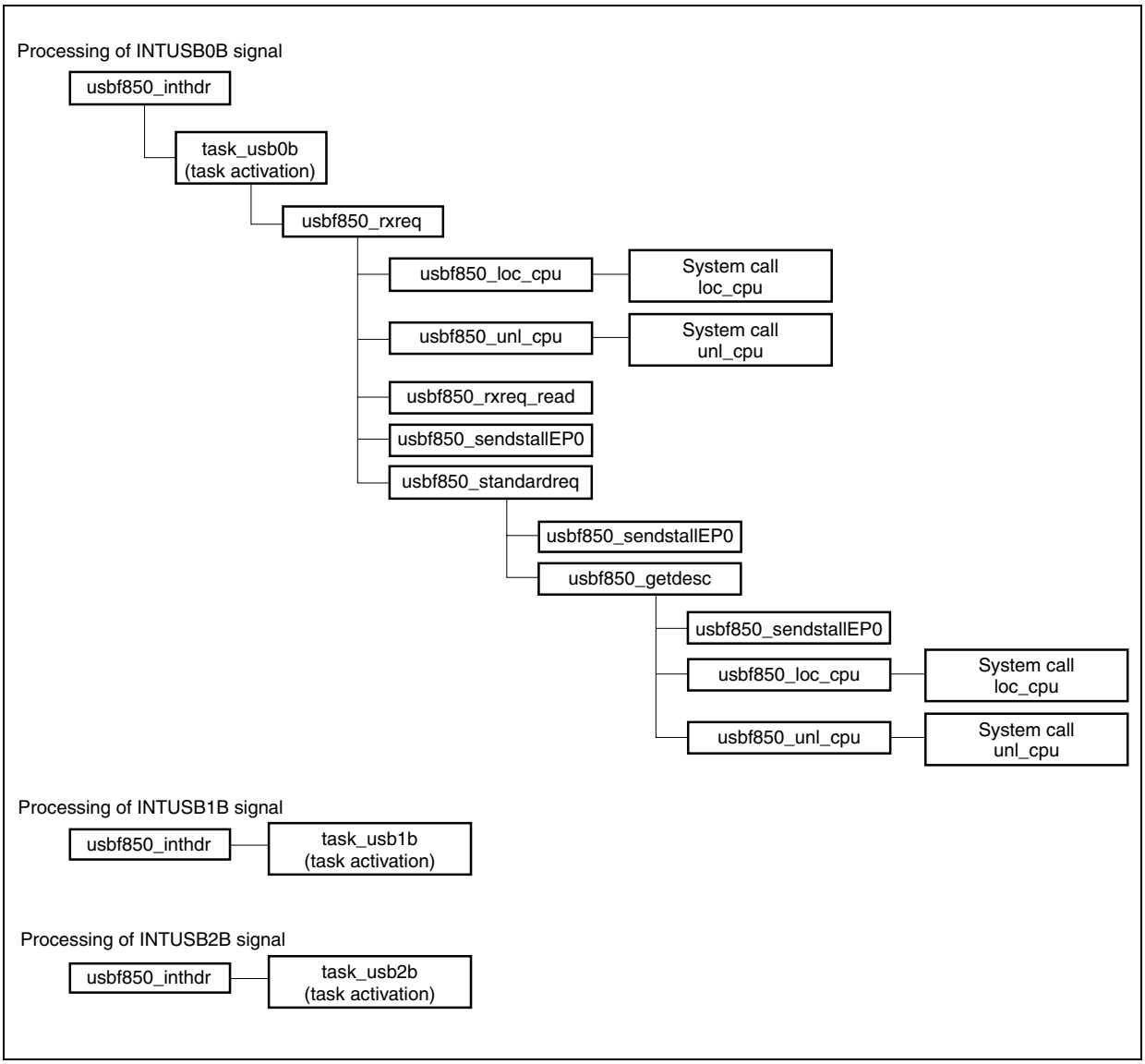
Processing Module Name	Function Name	File Name	Remark
USB bus driver processing module			
Initialization function	usbf850_init	usbf850.c	C language
Interrupt handler (INTUSB0B signal)	usbf850_inthdr	usbf850.c	C language
Interrupt handler (INTUSB1B signal)	usbf850_inthdr1	usbf850.c	C language
Interrupt handler (INTUSB2B signal)	usbf850_inthdr2	usbf850.c	C language
Interrupt servicing task (INTUSB0B signal)	task_usb0b	usbf850.c	C language
Interrupt servicing task (INTUSB1B signal)	task_usb1b	usbf850.c	C language
Interrupt servicing task (INTUSB2B signal)	task_usb2b	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 calling function (loc_cpu)	usbf850_loc_cpu	usbf850.c	C language
System call calling 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	–
Header file			
Data type declaration	–	types.h	–
Return value declaration	–	errno.h	–
Build file	–	usb_bus.bld	–
Section map file	–	common.lx	–

(2) Function tree

The calling relationship (function tree) in the sample program is illustrated below.

Caution *usbf850_init* is called from the initialization handler.

Figure 2-13. Sample Program Function Tree



(3) Description of functions

The functions in this sample program are explained in the following format.

xxxx … <1>	Valid caller: – – – – … <2>

[Outline] … <3>

– –
– –

[C language format] … <4>

– – – – – – – –

[Parameter] … <5>

I/O	Parameter	Description

[Operation] … <6>

– –
– –

[Return value] … <7>

– –
– –

<1> Name

Indicates the function name.

<2> Valid caller

Indicates the type of the processing module from which a function can be called.

Task:	The function can be called only from a task.
Non-task:	The function can be called only from a non-task.
Non-task | Task:	The function can be called from a task or non-task.
–:	Interrupt handler or task, and is not used to call functions.

<3> Outline

Shows the outline of a function operation.

<4> C language format

Shows the description format when calling a function from the processing module described in the C language.

<5> Parameter

Shows the function parameter 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 parameter

<6> Operation

Describes detailed operation of the function.

<7> Return value

Indicates the return value from a function using the data macro or numeric value.

usbf850_init	Valid caller: Non-task I Task

[Outline]

This is a function that initializes the USB function controller incorporated in the V850E/ME2.

[C language format]

```
void usbf850_init (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called from the software initialization module and performs processing to initialize the USB function controller incorporated in the V850E/ME2.

Remark Refer to **2.7.2 (1) Initialization processing** for details of initialization processing.

[Return value]

None

usbf850_inthdr	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB0B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This is the interrupt handler activated by the INTUSB0B signal (USB function status 0).

In this sample program, the interrupt handler checks the interrupt source and activates the interrupt servicing task (task_usb0b) only when the source is the CPUDEC interrupt. This handler is defined in the CF definition file.

[Return value]

Object ID number (task ID number)

usbf850_inthdr1	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB1B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr1 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB1B signal (USB function status 1).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb1b). This handler is defined in the CF definition file.

Caution This function is not used in the sample program.

[Return value]

Object ID number (task ID number)

usbf850_inthdr2	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB2B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr2 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB2B signal (USB function status 2).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb2b). This handler is defined in the CF definition file.

Caution This function is not used in the sample program.

[Return value]

Object ID number (task ID number)

task_usb0b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB0B signal.

[C language format]

```
void task_usb0b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB0B interrupt signal (USB function status 0 interrupt). In the sample program, this task calls the *usbf850_rxreq* function and performs processing of the USB standard device request.

Caution In this sample program, the standard device request Get Descriptor (String Descriptor) that is not responded automatically by the USB function controller incorporated in the V850E/ME2 is handled.

[Return value]

None

task_usb1b Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB1B signal.

[C language format]

```
void task_usb1b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB1B interrupt signal (USB function status 1 interrupt). This processing is not provided in the sample program, so the program returns without processing.

Caution This function is not used in the sample program.

[Return value]

None

task_usb2b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB2B signal.

[C language format]

```
void task_usb2b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB2B interrupt signal (USB function status 2 interrupt). This processing is not provided in the sample program, so the program returns without processing.

Caution This function is not used in the sample program.

[Return value]

None

usbf850_data_send Valid caller: Non-task | Task

[Outline]

This is a data transmit function used by the USB function controller.

[C language format]

```
long usbf850_data_send (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of transmit data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function transmits from the endpoint specified by *ep* data whose size is specified by *len* starting from the address specified by *data*.

Caution This function is not used in the sample program.

[Return value]

Status upon transmission

DEV_ERROR: Endpoint number is illegal

DEV_OK: Normal termination

usbf850_data_receive Valid caller: Non-task | Task

[Outline]

This is a data receive function used by the USB function controller.

[C language format]

```
long usbf850_data_receive (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of the buffer for receive data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function reads data whose size is specified by *len* from the buffer at the endpoint specified by *ep* and stores it to the address specified by specified *data*.

Caution This function is not used in the sample program.

[Return value]

Status upon reception

DEV_ERROR: Receive data size is illegal, or endpoint number is illegal.

DEV_OK: Normal termination

usbf850_sendnullEP0	Valid caller: Non-task I Task

[Outline]

This is a function that transmits Null data from the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendnullEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function transmits Null data (whose data size is 0) from the control endpoint (endpoint 0).

[Return value]

None

usb850_sendstallEP0	Valid caller: Non-task I Task

[Outline]

This is a function that sends a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendstallEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the control endpoint (endpoint 0).

[Return value]

None

usbf850_bulkin1_stall	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 1).

[C language format]

```
void usbf850_bulkin1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 1).

Caution This function is not used in the sample program.

[Return value]

None

usbf850_bulkout1_stall	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 2).

[C language format]

```
void usbf850_bulkout1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 2).

Caution This function is not used in the sample program.

[Return value]

None

usbf850_loc_cpu	Valid caller: Task

[Outline]

This is a function that disables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_loc_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *loc_cpu* system call.

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

[Return value]

None

usbf850_unl_cpu	Valid caller: Task

[Outline]

This is a function that enables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_unl_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *unl_cpu* system call.

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

[Return value]

None

usbf850_rxreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs USB request processing.

[C language format]

```
void usbf850_rxreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *task_usb0b* task that is activated by the INTUSB0B interrupt signal. This function calls SETUP data read processing, analyzes the read data, and calls USB request processing based on the analysis result.

[Return value]

None

usbf850_rxreq_read	Valid caller: Non-task I Task

[Outline]

This is a function that reads USB request data.

[C language format]

```
void usbf850_rxreq_read (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function reads SETUP data received subsequently to the Setup token at the control endpoint (endpoint 0). The SETUP data is distinguished from normal data and is stored in a dedicated register. It is always read in 8-byte units.

[Return value]

None

usbf850_standardreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs the USB standard request.

[C language format]

```
void usbf850_standardreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called if the standard request is read from SETUP data and calls the *usbf850_getdesc* function when the request type is confirmed as the Get Descriptor request.

[Return value]

None

usbf850_getdesc	Valid caller: Non-task I Task

[Outline]

This is a function that performs the USB standard request Get Descriptor (String Descriptor) processing.

[C language format]

```
void usbf850_getdesc (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *usbf850_standardreq* function and performs the USB standard request Get Descriptor (String Descriptor) processing. This function sets a STALL response for a request other than the Get Descriptor (String Descriptor) request.

[Return value]

None

usbf850_sstall_ctrl	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sstall_ctrl (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response at the control endpoint (endpoint 0).

Caution This function is not used in the sample program.

[Return value]

None

CHAPTER 3 USB STORAGE CLASS DRIVER

3.1 General

3.1.1 Overview

The USB storage class driver is a sample program for the USB function controller that is incorporated in the V850E/ME2. It conforms to Universal Serial Bus Specification Revision 1.1 and Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0, and operates on the embedded real-time control operating system RX850 Pro (conforms to the μITRON 3.0 specifications).

This sample program uses the control endpoint (endpoint 0) and IN and OUT of the bulk endpoint (endpoints 1 and 2), and is connected to the Windows XP standard storage class host driver to control a storage device (virtual device). The Mass Storage class is defined as the class.

This sample program uses the emulation board SolutionGear MINI (SG-703111-1) as the hardware execution environment. When using the SolutionGear MINI and sample program as is, create the execution object by following the procedure described in **3.6 Load Module** and confirm its operation by following the procedure described in **3.2 Execution of Load Module**.

When using another target board instead of SolutionGear MINI, change the board referring to **3.3 System Configuration**, **3.4 RX850 Pro-Dependent Processing Modules**, and **3.5 Section Map File**, in accordance with the board specifications.

When changing both SolutionGear MINI and sample program, change them referring to **3.3 System Configuration**, **3.4 RX850 Pro-Dependent Processing Modules**, **3.5 Section Map File**, **3.6 Load Module**, and **3.7 USB Driver Functions**.

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

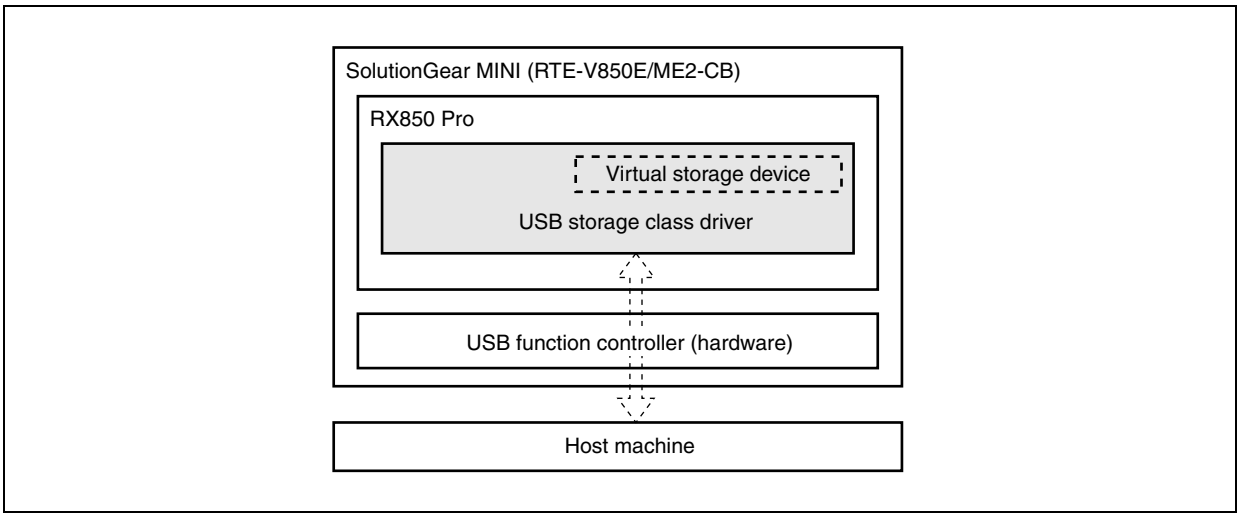
Caution This sample program operates as a mass-storage device (interface class: Mass Storage, interface sub-class: SCSI, interface protocol: Bulk-Only Transport protocol). The storage device used in this sample operates under the assumption that there are no logical units connected, the memory area is secured, and a removable disk is connected (block size: 512 bytes, number of logical blocks: 192, capacity: 96 KB).

Remarks 1. Refer to the following 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. The descriptions in **3.2.1 Execution procedure of load module** assume the user environment described in **3.1.3 Execution environment**.

Figure 3-1. Positioning of USB Storage Class Driver



3.1.2 Development environment

This section assumes the following hardware and software environments are used for system development using the sample program.

- Hardware environment
  Host machine: PC/AT-compatible machines (OS: Windows XP)
- Software environment
  Real-time OS: RX850 Pro Version 3.15
  USB storage class driver: Sample program set described in this section
  C compiler package: MULTI2000
  (CCV850 Version 3.5 (made by Green Hills Software, Inc.))

Caution If the directory configuration of the user environment differs from that handled in the build file of the sample program, adjust the build file to the user environment.

Remark Refer to the help of MULTI (made by Green Hills Software, Inc.) for the description of the build file.

3.1.3 Execution environment

This section assumes the following hardware and software environments are used for load module execution using the sample program.

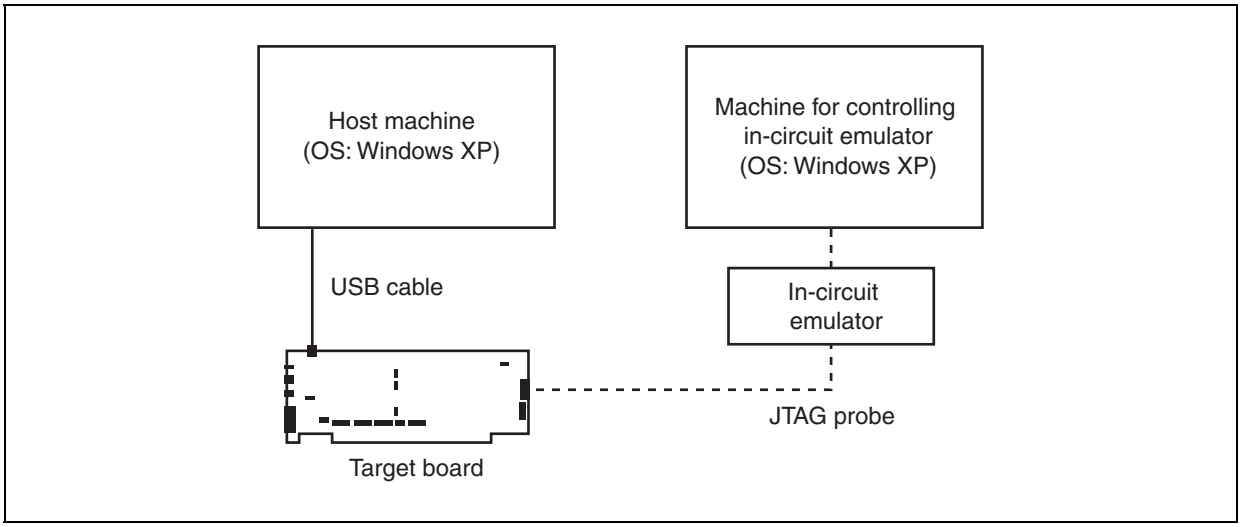
- Hardware environment
  - Host machine: PC/AT-compatible machines (OS: Windows XP)
  - IE control machine: PC/AT-compatible machines (OS: Windows XP)
  - Target board: SolutionGear MINI (SG-703111-1)
  - In-circuit emulator (IE): N-wire IE (RTE-2000-TP) (made by Midas Lab Inc.)
  - JTAG probe
  - USB cable
- Software environment
  - Software for IE: PARTNER Setup Program Version 1.242

Remarks 1. Refer to **APPENDIX A SG-703111-1 BOARD** and the **SG-703111-1 User's Manual** for details of how to set up the execution environment.

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

3. Refer to the **PARTNER User's Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for details of PARTNER.

Figure 3-2. Execution Environment



3.2 Execution of Load Module

3.2.1 Execution procedure of load module

The following shows the procedure for executing the load module under the environment described in **3.1.3 Execution environment**, taking the load module using the sample program as an example.

(1) Preparation of machine for controlling in-circuit emulator (IE)

Turn on the power and start up the IE control machine and the in-circuit emulator.

(2) Preparation of host machine

Turn on the power and start up the host machine (the IE control machine can be used as the host machine, but it is strongly recommended to provide an independent machine for development).

(3) Reset SG-703111-1 board

Press the RESET button of the SG-703111-1 board to reset the SG-703111-1 board.

(4) Startup of software for IE

Start up software for IE.

Select the [Start] button → "All Programs" → "PARTNER" → "RPTSETUP (NB85ET)" in Windows.

Click the [Open] button and specify a project file; the [Run] button is then selectable. Click the [Run] button to start up PARTNER. Make the board settings after startup. It is useful to create at this time the setting file loaded at startup. Refer to **APPENDIX A SG-703111-1 BOARD**, **PARTNER User's Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for setup files for the sample described in this section.

Cautions 1. Be sure to apply power to the target board before starting up the in-circuit emulator.

2. If you want to load the setting file for resetting the target board after the in-circuit emulator is started up, load the setting file (init.mcr in the example below) by inputting a command to the command window, as shown below.

[Command input example]

```
><init.mcr<Enter>
```

(5) Loading the load module

Load the load module to the board using the in-circuit emulator function.

The load module (usb_storage.out in the example below) can be loaded by selecting [Load] in the [File] menu on the toolbar, or input the L command (loading file) in the command window.

[Command input example]

```
>l usb_storage.out<Enter>
```

(6) Execution

The code loaded to the board is executed by pressing the F5 key or the [Go] button.

Remark The same operation is performed by selecting [Go] in the [Run] menu on the toolbar.

(7) USB connection

Connect the USB cable.

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

Cautions 1. The USB cable can be connected before/after starting up the target board.

2. When the device is detected by the host machine, the Windows XP standard USB storage class host driver is automatically installed. After the driver has been installed normally, the device is displayed under "Removable disk" in My Computer.

(8) Startup of Device Manager

Open the Properties window from My computer and select the Hardware tab. Select the Device Manager to start up the Device Manager.

Remark The Device Manager can also be started up from [Manage] menu of My computer or the Control Panel.

(9) Confirmation of USB device connection

Make sure that "USB Mass Storage Device" is displayed under "Universal Serial Bus controllers", and "NEC corp StorageFncDriver USB Device" is displayed under "Disk Drives" in the Device Manager screen.

(10) How to use device

Select "Removable disk" in My Computer and execute "Open" on the right-click menu; a screen to prompt disk formatting appears. Execute formatting following the instructions on the subsequent screens.

After formatting is completed normally, the disk can be used in the same manner as using ordinary disk device, such as reading, writing or deleting files.

In addition, the disk contents are held until execution of a load module is stopped.

(11) Exiting program

Terminate the program under execution.

Click the forcible break button on the PARTNER screen, or select "Forcible Break" in the [Run] menu on the toolbar to stop program execution.

(12) Shutting down in-circuit emulator

Shut down the in-circuit emulator and reset the target board by following the procedure described in **(1)**.

Select [Exit] in the [File] menu on the toolbar to terminate PARTNER.

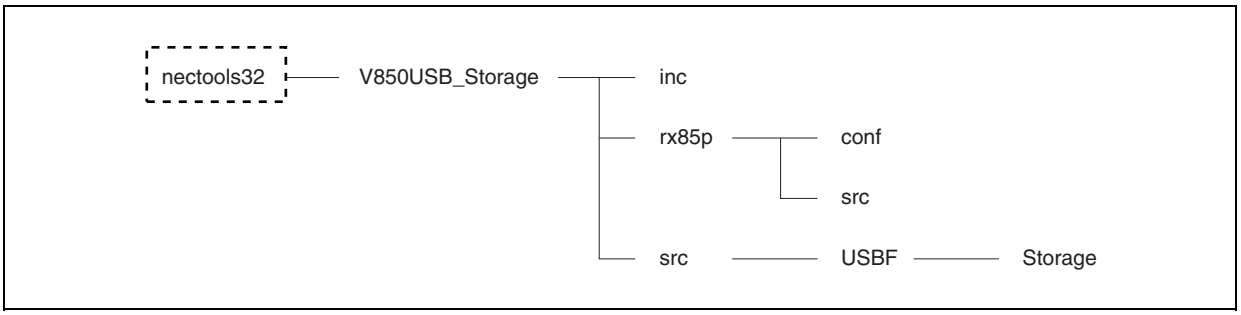
After terminating PARTNER, reset the target board by following the procedure described in **(1)**.

3.2.2 Directory configuration

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

Caution It is recommended to place the directory of the USB storage class driver files directly under the directory where the RX850 Pro is installed (\nectools32).

Figure 3-3. Sample Program Directory Configuration



The outline of each directory is shown below.

(1) nectools32

A directory created when the RX850 Pro is installed. Place the directory (directory name: V850USB_Storage) of the driver directly under this directory.

(2) nectools32\V850USB_Storage

A directory for the USB storage class driver.

- usb_storage.bld: Build file of USB storage class driver
- common.lx: Section map file

(3) nectools32\V850USB_Storage\inc

A directory in which header files for the USB storage class driver are stored.

- errno.h: Header file for return value
- types.h: Header file for data type
- sys.h: Header file for system information

Caution sys.h (header file for system information) is usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create this file because the command is automatically executed when build is executed.

(4) nectools32\V850USB_Storage\rx85p

A directory in which files for the RX850 Pro are stored.

(5) nectools32\V850USB_Storage\rx85p\conf

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

- sit.850: System information table
- svc.850: System call table
- sysi.tbl: System information table
- sysc.tbl: System call table

Cautions 1. Files in this directory are usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create these files because the command is automatically executed when build is executed.

2. sit.850 and sysi.tbl, svc.850 and sysc.tbl differ only in their file extension.

(6) nectools32\V850USB_Storage\rx85p\src

A directory in which files for RX850 Pro are stored.

- boot.850: Assembler file for boot processing
- entry.850: Assembler file for entry processing
- init.c: Source file for hardware initialization module
- init.h: Header file for hardware initialization module
- sys.cf: CF definition file
- varfunc.c: Source file for software initialization module

(7) nectools32\V850USB_Storage\src

A directory in which files of the USB storage class driver board-dependent module are stored.

- port.c: Source file for port setting
- port.h: Header file for port setting

(8) nectools32\V850USB_Storage\src\USBF

A directory in which files of the USB storage class driver USB processing module are stored.

- usbf850.c: Source file for USB device
- usbf850.h: Header file for USB device
- usbf850desc.h: USB descriptor definition file
- usbf850_dma.c: Source file for DMA control
- usbf850_dma.h: Header file for DMA control
- usbf850_storage.c: Source file for USB-storage interface
- usbf850_storage.h: Header file for USB-storage interface

(9) nectools32\V850USB_Storage\src\USBF\Storage

A directory in which files of the USB storage class driver storage device processing module are stored.

- ata_ctrl.c: Source file for storage device control
- ata.h: Header file for storage device
- scsi_cmd.c: Source file for SCSI command processing
- scsi.h: Header file for SCSI command processing

3.3 System Configuration

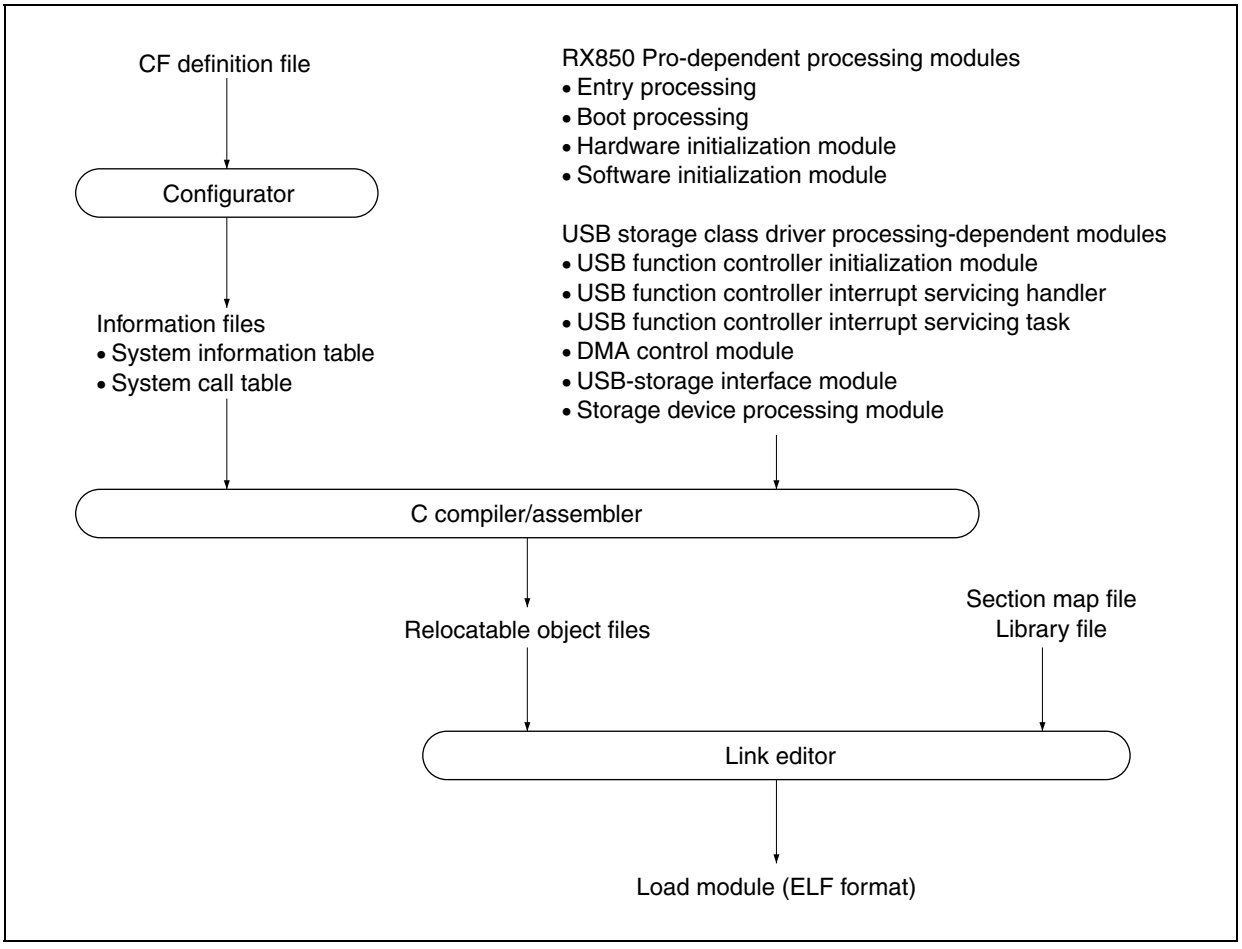
3.3.1 Overview

System configuration means creation of the load module using files that are installed in the user's development environment (the host machine) from the USB storage class driver supply medium.

The system configuration procedure of USB storage class drivers is shown below.

(1) Describing RX850 Pro-dependent processing module
(2) Describing board-dependent module
(3) Describing USB storage class driver processing-dependent module
(4) Describing section map file
(5) Creating load module

Figure 3-4. System Configuration Procedure



3.3.2 Describing RX850 Pro-dependent processing module

Some functions provided by the USB storage class driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
- Boot processing
- Hardware initialization module
- Software initialization module

Remark Refer to **3.4 RX850 Pro-Dependent Processing Modules** for details of the RX850 Pro-dependent processing module.

3.3.3 Describing board-dependent module

The initialization processing, which is related to the processing dependent on the user's execution environment and application system, is provided as a board-dependent module in the USB storage class driver source program.

The board-dependent module is as follows.

- CPU board-dependent module

  The port input/output manipulation required for the USB storage class driver is provided as a CPU board-dependent module.

Caution Since port setting is handled in the same manner as setting of other registers, no dedicated function is provided.
Refer to the RX850 Pro standard header file *SFR.h* stored in \nectools32\inc850\common\ for the register definition. For detailed processing, refer to the source program for port setting (port.c) called from the boot processing module (boot.850) and software initialization module.

3.3.4 Describing USB storage class driver processing-dependent module

The driver functions, which are used to implement the USB storage class driver functions, are provided as the USB storage class driver processing-dependent module in this sample program.

The USB storage class driver processing-dependent modules are listed below.

- USB function controller initialization module
- USB function controller interrupt handlers
- USB function controller interrupt servicing tasks
- USB function controller general-purpose functions
- DMA control module
- USB-storage interface module
- Storage device processing module

Remark Refer to **3.7 USB Storage Class Driver Functions** for details of the USB storage class driver processing-dependent module.

3.3.5 Describing section map file

The section map file is used by the user to fix address assignment performed by the link editor.

The following five text areas are essential sections when using the RX850 Pro.

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

Remark Refer to **3.5 Section Map File** for details of the section map file.

3.3.6 Creating load module

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing modules, USB storage class driver processing-dependent module, and section map file, that have been coded.

Remark Refer to **3.6 Load Module** for details of how to create the load module.

3.4 RX850 Pro-Dependent Processing Modules

3.4.1 Overview

Some functions provided by the USB storage class driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
- Boot processing
- Hardware initialization module
- Software initialization module

3.4.2 CF definition file

An information file (CF definition file) that contains data provided to the RX850 Pro is required to configure the system in which the RX850 Pro is used.

The following information is required for using the USB storage class driver function.

○ Real-time OS information
- RX Series information

○ SIT information
- System information
- System maximum value information
- System memory information
- Task information
- Interrupt handler information
- Initialization handler information

○ SCT information
- Task management/task-associated synchronization system call information
- Interrupt servicing management system call information
- Time management system call information

Caution This sample program implements each functions using six tasks, three interrupt handlers, and seven system calls. Therefore, the CF definition file, the maximum number of tasks to be created must be set to six as the system's maximum value information and the maximum number of interrupt handlers to be created must be set to three for the USB storage class driver and use of *sta_tsk*, *ext_tsk*, *slp_tsk*, and *wup_tsk* system calls must be defined as task management/task-associated synchronization system call information, use of the *loc_cpu* and *unl_cpu* system calls as interrupt servicing management system call information, and use of the *dly_tsk* system call as time management function system call information.

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

(1) Procedure for creating information files

A procedure for creating information files (system information table, system call table, and system information header file) is shown below.

The information file can be created from the Windows command prompt.

Caution If a build file in the sample program is used, users are not required to create information files in this procedure because they are automatically executed when build is executed.

<1> Change current directory

Move the current directory to the directory in which the CF definition file is stored using the cd command of Windows.

A command 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> Creating information files

Create the information file from the CF definition file that has been created in the specific description format, using the configurator cf850pro.exe.

A command input example when creating three information files (system information table: sit.850, system call table: svc.850, and system information header file: sys.h) from an input file (CF definition file name: sys.cf) is shown below.

[Command input example]

```
C:>cf850pro -i sit.850 -c svc.850 -d sys.h sys.cf<Enter>
```

The information files are created from the CF definition file.

Caution A sample file (CF definition file) used for creating the information files is provided in the sample program.

Remark Refer to the **RX850 Pro Installation User's Manual** for details of the option to activate the configurator cf850pro.exe and execution method.

3.4.3 Entry processing

This processing assigns a branch instruction to an interrupt handler to the handler address where control is forcibly passed by the processor when a maskable interrupt occurs.

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

Remark Refer to sample program *entry.850* for details of how to code the entry processing.

3.4.4 System initialization processing

The system initialization processing includes initialization processing (boot processing and hardware initialization module) of hardware required for operating the RX850 Pro normally, and software initialization processing (nucleus initialization module and Initialization handler).

The system initialization processing is performed first when the system is activated.

Caution Among the four types of system initialization processing, users are not required to describe the nucleus initialization module because it is a function provided by the RX850 Pro.

The processing performed by the nucleus initialization module is shown below.

○ Securement of system memory defined by CF definition file
- System pool 0
- User pool 0

○ Generation and activation of management object defined by CF definition file
- Generation and activation of task
- Registration of interrupt handler

○ Activation of initial task
○ Generation and activation of idle task
○ Calling software initialization module
○ Passing control to scheduler

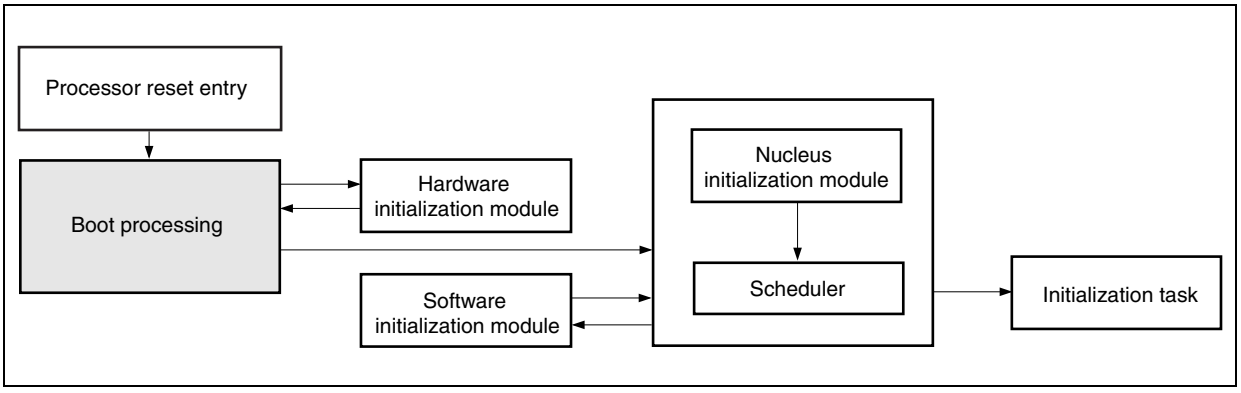
The idle task is a processing routine that is activated by the scheduler when a processing module (task) executed by the RX850 Pro is no longer in the run or ready state, that is, no processing module targeted to the scheduling by the RX850 Pro exist in the system. The idle task issues the HALT instruction.

(1) Boot processing

Boot processing is the function assigned to the processor reset entry, so it is executed first in the system initialization processing.

The positioning of boot processing is shown below.

Figure 3-5. Positioning of Boot Processing



The processing executed by boot processing is shown below.

Remark Refer to sample program *boot.850* for details of how to code boot processing.

- Setting tp, gp, and ep registers
  Values of the text pointer tp, global pointer gp, and stack pointer ep, which are required for execution of each processing module (including boot processing), are undefined when a system is activated. Boot processing first performs initial setting of these registers.

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

- Calling hardware initialization module
  Functions (hardware initialization module) are called to initialize the hardware on the target system.
  This step is not required if initialization of internal units is performed by other module.

  Caution In this chapter, this step is not required because initialization of internal units is performed by the software initialization module. Refer to the RX850 Pro Installation User's Manual for details.

- Passing control to nucleus initialization module
  The nucleus initialization module secures the system memory (system pool 0, user pool 0) and creation/initialization of management objects, based on information described in the system information table. Therefore, start address_sit of the system information table must be set to the r10 register before passing control to the nucleus initialization module.

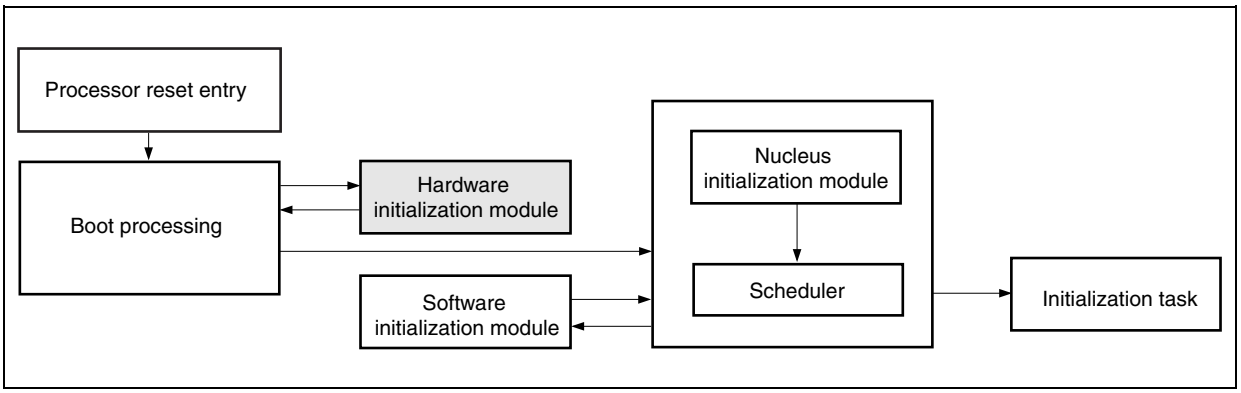
  Caution The system information table is a table in which the CF definition file created in a specific description format is converted to the assembly language format, using the utility tool (configurator cf850pro.exe) provided by the RX850 Pro.

(2) Hardware initialization module

The hardware initialization module is a function to initialize the hardware on the target system, and is called from boot processing.

The positioning of the hardware initialization module is shown below.

Figure 3-6. Positioning of Hardware Initialization Module



The processing executed by the hardware initialization module is shown below.

Cautions 1. Users are not required to disable the maskable interrupts because they are masked at initialization by default.

2. Hardware initialization is performed by the software initialization module in the sample program. Refer to the RX850 Pro Installation User's Manual for details of the hardware initialization module.

- Returning control to boot processing

  Control can be returned from the hardware initialization module to boot processing by issuing the "return( );" instruction, because the return address to the lp register is set when the hardware initialization module is called from boot processing.

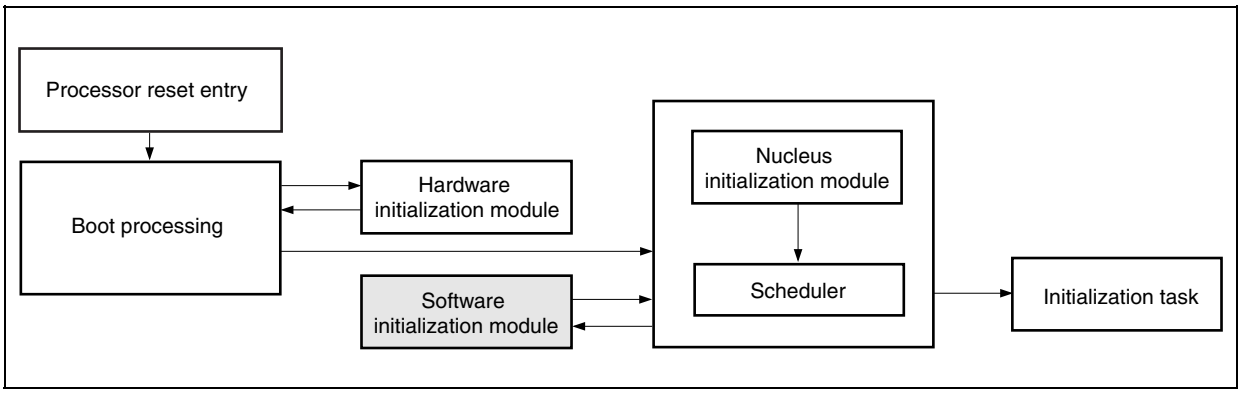
  If the hardware initialization module is described with the assembly language, this processing is implemented by issuing the "jmp [lp]" instruction.

(3) Software initialization module

The initialization handler is a function provided to enhance operability of the user software environment, and is called from the nucleus initialization module.

The positioning of the software initialization module is shown below.

Figure 3-7. Positioning of Software Initialization Module



The processing executed by the software initialization module is shown below.

Remark Refer to sample program *varfunc.c* for how to code the software initialization module.

- Initialization of internal unit (real-time pulse unit (RPU))

  The RX850 Pro implements the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) using the timer interrupt that occurs in a constant cycle. Therefore, the real-time pulse unit must be initialized before the RX850 Pro starts processing.

  The compare register CMD0 included in the real-time pulse unit must be set so that timer interrupts occur in a base clock cycle defined in system information in the CF definition file.

- Enabling timer interrupt acknowledgment

  Acknowledgment of timer interrupts is enabled. In addition, this enables the use of the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) provided by the RX850 Pro when processing by the nucleus initialization module ends.

- Passing control to nucleus initialization module

  Control can be returned from the initialization handler to the nucleus initialization module by issuing the “return( );” instruction, because the return address lp register is set when the initialization handler is called from the nucleus initialization module.

  If the initialization handler is described with the assembly language, this processing is implemented by issuing the “jmp [lp]” instruction.

3.4.5 Time management function

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

The RX850 Pro calls system clock processing when a clock interrupt occurs, and performs processing related to the time such as updating the system clock, task delay wake-up, and activation of the cyclic handler.

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

After the system clock is set to “0H” by system initialization processing, it is updated by system clock processing in base clock cycle units (specified at configuration).

Caution The system clock managed by the RX850 Pro is configured as 48 bits wide. Therefore, overflowed numeric values (numeric values that cannot be expressed by 48 bits) are ignored by the RX850 Pro. Refer to the RX850 Pro Basics User’s Manual for details of the time management function of the RX850 Pro.

3.5 Section Map File

3.5.1 Overview

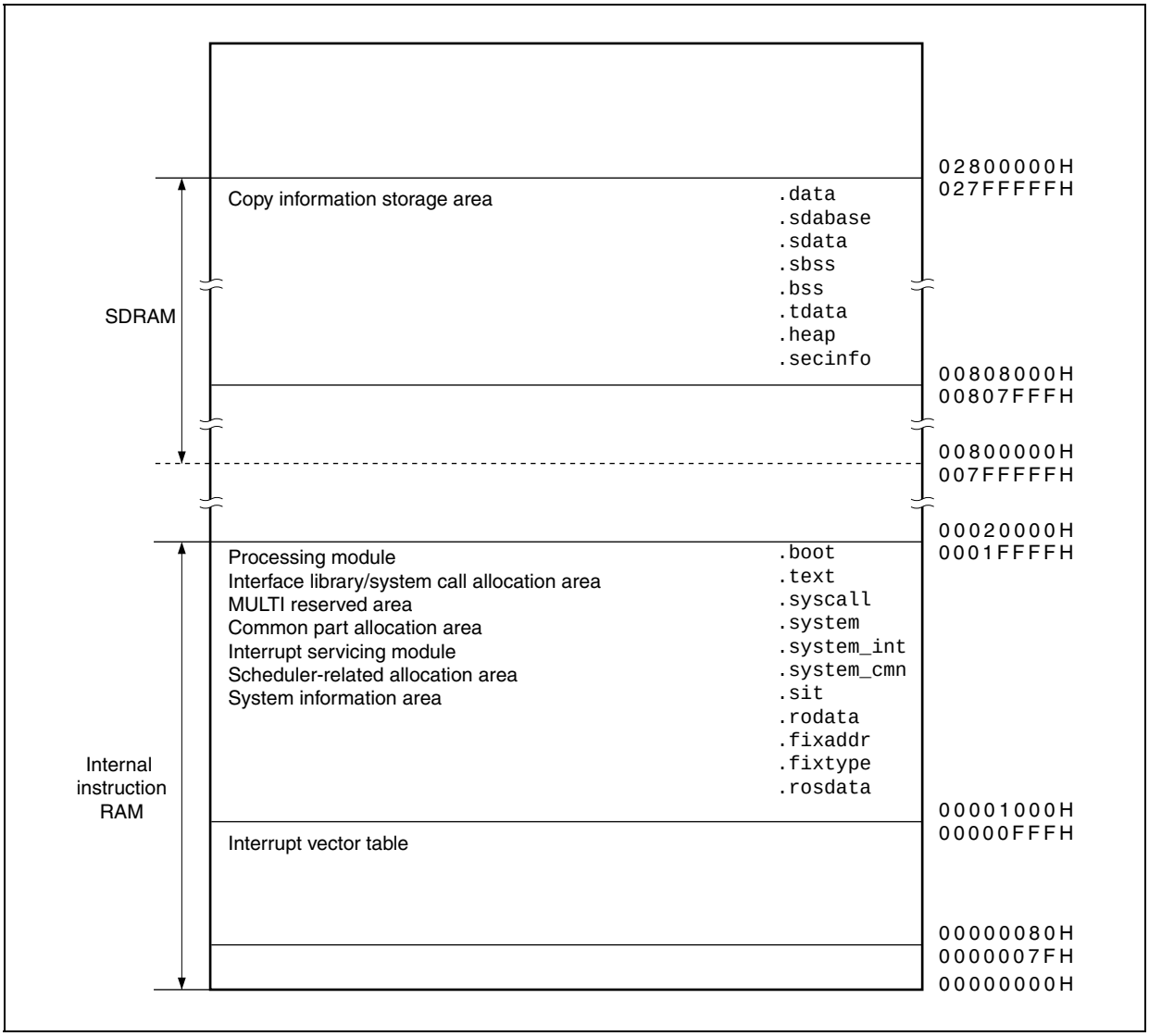
The section map file is used by the user to fix address assignment performed by the link editor.

Required assignments for addresses other than the user processing program (such as .data and .bss sections) are described in **3.5.2 Address assignment by RX850 Pro** and **3.5.3 Other address assignment**.

Address assignment performed in sample program *common.lx* is shown below.

Remark Refer to sample program *common.lx* for how to code the section map file.

Figure 3-8. Address Assignment Example



3.5.2 Address assignment by RX850 Pro

The RX850 Pro consists of five text areas: common part allocation area, interrupt servicing-related allocation area, scheduler-related allocation area, system information area, and interface library/system call allocation area. Using these areas, memory areas for which a large space is required can be assigned to the external RAM, and memory areas for which a high-speed access is required (interrupt servicing module, scheduling processing module) can be assigned to the internal instruction RAM (00000000H to 0001FFFFH).

Caution All five text areas are allocated to the internal instruction RAM in the sample program.

- Common part allocation area (.system section)
  Processing of the RX850 Pro (such as task management function, task-associated synchronization function) is assigned to this area.

- Interrupt servicing-related allocation area (.system_int section)
  Among the interrupt servicing management functions provided by the RX850 Pro, interrupt preprocessing that is performed when control is passed to the interrupt handler and interrupt postprocessing that is performed when control is handed back to the processing module in which a maskable interrupt occurs are assigned to this area.
  By assigning the interrupt servicing module to the internal instruction RAM, therefore, response performance to the interrupt handler can be improved.

  Caution It is recommended to assign the interrupt servicing module to the internal instruction RAM.

- Scheduler-related allocation area (.system_cmn section)
  Among the scheduling function provided by the RX850 Pro, task wake-up processing and task scheduling processing are assigned to this area.
  By assigning the scheduling processing section to the internal instruction RAM, therefore, task wake-up processing and task scheduling processing are accelerated, as well as system call processing involving scheduling processing.

  Caution It is recommended to assign the scheduling module to the internal instruction RAM.

- System information area (.sit section)
  The system information table created by executing the configurator cf850.exe on the CF definition file is assigned to this area.
  The system information table includes various data required for executing the nucleus initialization module (securement of the system memory and creation/initialization of management objects).

- Interface library/system call allocation area (.text section)
  The instructions including system calls are assigned to this area.

- System memory
  Various management block required for implementing functions provided by the RX850 Pro (such as the task management block, semaphore management block), area in which the stack used by the interrupt handler or task is assigned (system pool 0), and area in which dynamic memory manipulation (such as acquisition/release of memory blocks) from the processing module is enabled (user pool 0), are assigned to this area.

Cautions 1. The ”system memory start address” must be specified when creating the CF definition file. Be sure to specify the address when defining the system memory in the section map file.
2. The user can specify any section name in the system memory.

3.5.3 Other address assignment

The other sections for which address assignment is required are described below.

- MULTI reserved area (.syscall section)
  This area is used as a work area by the debugger MULTI (made by Green Hills Software, Inc.).

Cautions 1. The .syscall section must be defined regardless of whether or not MULTI is used.
2. Be sure to specify 4-byte alignment when defining the .syscall section.

- Copy information storage area (.secinfo section)
  This area is used by the link editor to output information (start address, size) required for transferring program (data, text) of a section for which the ROM identifier is specified in the section map file from ROM to RAM.
  Specification of the ROM identifier is required when performing ROMization of a processing module. Therefore, definition of the .secinfo section is not required when ROMization is not performed.

Caution This section is empty in the sample program because ROM identifier specification is not performed.

3.6 Load Module

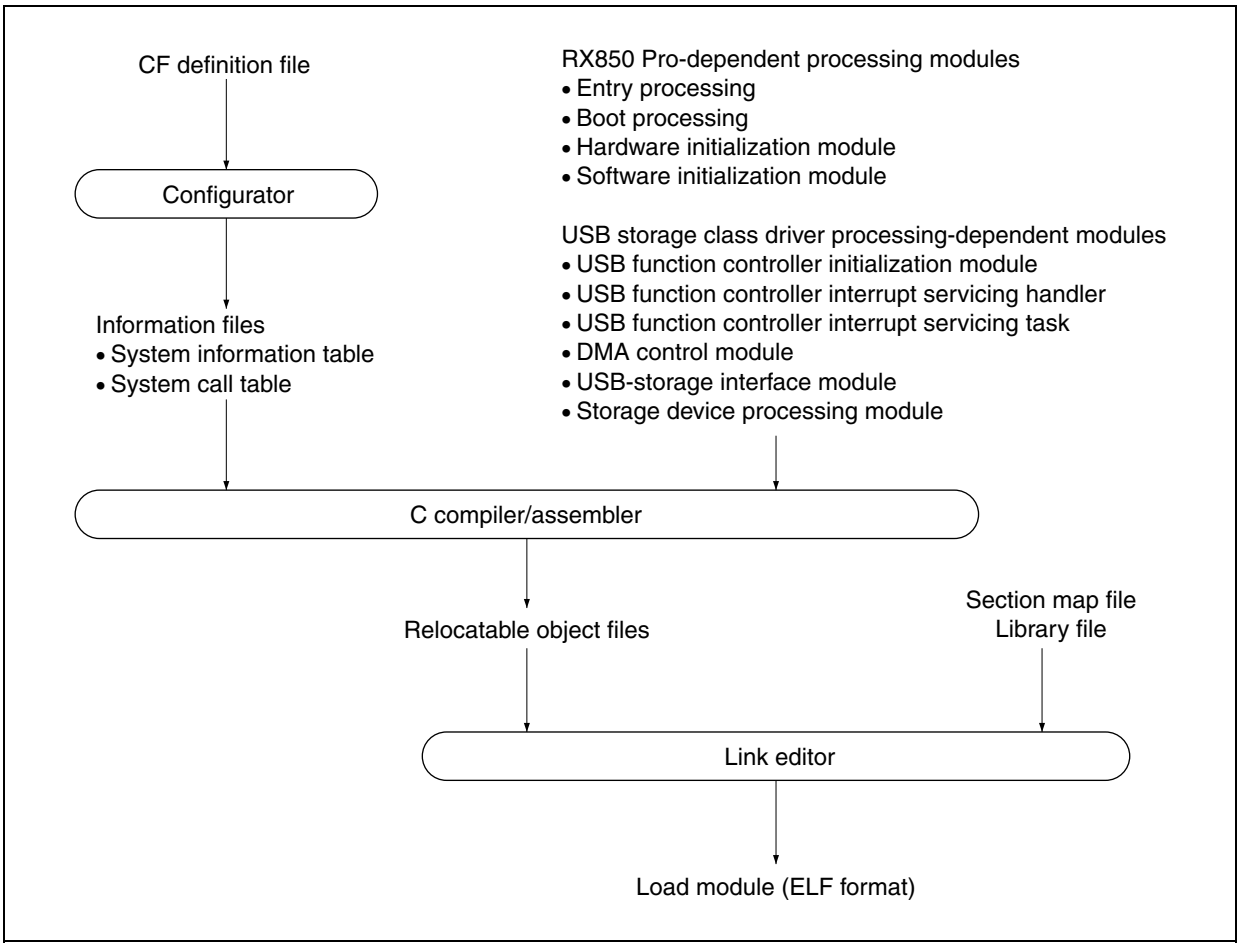
3.6.1 Overview

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing module, USB storage class driver processing-dependent module, section map file, that have been coded.

The procedure for creating load modules is shown below.

Caution The load module corresponding to the sample program can be created by executing the .bld file in the sample program. However, definition of the .bld file must be adjusted to the user development environment.

Figure 3-9. Load Module Creation Procedure



3.6.2 Creating load module

An ELF-format load module can be created from the RX850 Pro-dependent processing module, USB storage class driver processing-dependent module, and section map file, that have been coded, using the following procedure.

(1) Creation of system information table and system call table

Original CF definition file formats are excluded from the link processing performed by the link editor when creating a load module.

Therefore, a file that can be assembled (system information table or system call table) must be created using the utility tool (configurator cf850.exe) provided by the RX850 Pro.

Remark Refer to **3.4.2 (1) Procedure for creating information file** for how to create the system information table and system call table.

(2) Creation of object file

A relocatable object file is created by executing the C compiler/assembler for the processing module (file described in the C language/assembly language) shown below.

○ RX850 Pro-dependent processing module
- System information table
- System call table
- Entry processing
- Boot processing
- Hardware initialization module
- Initialization handler

○ USB storage class driver processing-dependent module

(3) Creation of load module

An ELF-format load module is created by executing the link editor for relocatable object file created in **(2)**, library files, and section map file.

libansi.a	ANSI C library
libind.a	C library made by Green Hills Software, Inc. (routines independent of target CPU)
libarch.a	C library made by Green Hills Software, Inc. (routines dependent of target CPU)
libsys.a	C library made by Green Hills Software, Inc. (system call, initialization routines)

rxcore.o	Nucleus common part object
librxp.a	Nucleus library
libchp.a	Interface library

rxcore.o, *librxp.a*, and *libchp.a* are provided by the RX850 Pro, and *libansi.a*, *libind.a*, *libarch.a*, and *libsys.a* are provided by the CCV850 (made by Green Hills Software, Inc.).

3.7 USB Storage Class Driver Functions

3.7.1 Overview

Initialization processing performed by the USB function controller, as well as tasks and interrupt handlers to implement USB storage class driver processing, must be described in the USB storage class driver.

A list of USB storage class driver processing-dependent modules is shown below.

- USB function controller initialization processing
  This module is called from the RX850 Pro software initialization module and initializes the USB function controller.

- USB function controller interrupt handlers
  This is an interrupt servicing-dedicated routine that is called each time an interrupt by the USB function controller occurs, and is defined in the CF definition file.

  Caution Interrupts other than required are masked in this sample program.
  The following four interrupts are used in this sample program.

  - **SHORT interrupt reported by INTUSB0B signal**
    (Indicates that data was read from the FIFO of the UF0BO1 or UF0BO2 register when the FIFO was not full in DMA mode, and the USBSPnB signal (n = 2 or 4) has been activated.)
  - **DMAED interrupt reported by INTUSB0B signal**
    (Indicates that the DMA end signal for endpoint n (n = 1 to 4, 7, or 8) has been activated.)
  - **CPUDEC interrupt reported by INTUSB0B signal**
    (Indicates that there is a request that is decoded by FW in the UF0E0ST register)
  - **BKO1DT interrupt reported by INTUSB1B signal**
    (Indicates that data has been received normally by the UF0BO1 register)

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

- USB function controller general-purpose function
  This is a general-purpose function used by the USB storage class driver to perform the STALL response setting for each endpoint and transmission/reception processing.

  Remark Refer to sample program *usbf850.c* for how to code the USB storage class driver processing-dependent module.

- DMA control module
  This module performs DMA initialization and activation processing.
  In this sample program, DMA transfer is used if the size of data at the bulk endpoint exceeds the MaxPacket size (40 bytes).

  Remark Refer to sample program *usbf850_dma.c* for details of how to code the DMA control module.

- Bulk-Only Transport processing module
  This module performs processing of USB storage class requests specific to device class, CBW, and CSW transmission.

  Caution The following two requests specific to device class can be acknowledged in this sample program. Refer to 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 Refer to sample program *usbf850_storage.c* for how to code the Bulk-Only Transport processing module.

- Storage device processing module
  This module performs storage device initialization and SCSI command processing.

  Cautions 1. This sample program operates as a mass-storage device (interface class: Mass Storage, interface sub-class: SCSI, interface protocol: Bulk-Only Transport protocol). The storage device used in this sample operates under the assumption that there are no logical units connected, the memory area is secured, and a removable disk is connected (block size: 512 bytes, number of logical blocks: 192, capacity: 96 KB). The memory area for the virtual device is secured as an array named *storage_data*. Refer to sample program *ata.h* for details.

  2. When controlling the actual device instead of the virtual device, the data and data processing used in the sample program must be modified. Adjust the data to the user environment.

  Remark Refer to sample programs *ata_ctrl.c* and *scsi_cmd.c* for details of how to code the storage device processing module.

- USB suspend/resume processing
  Since the USB suspend/resume processing depends on the system, it is not supported in this sample program. If this processing is necessary in your system, add the processing making allowances for the following points.
  The suspend/resume state is reported to the USB function controller incorporated in the V850E/ME2 by an interrupt (INTUSB0B signal). Therefore, whether the current status is suspend or resume can be judged by checking the UF0IS0. RSUSPD bit in the interrupt handler (for the INTUSB0B signal); if this bit is 1, the UF0EPS1.RSUM bit is checked to judge the status.
  Processing can be added by adding the above code to judge the status to the interrupt handler (for the INTUSB0B signal) and wakes up a task to perform necessary processing from the code.

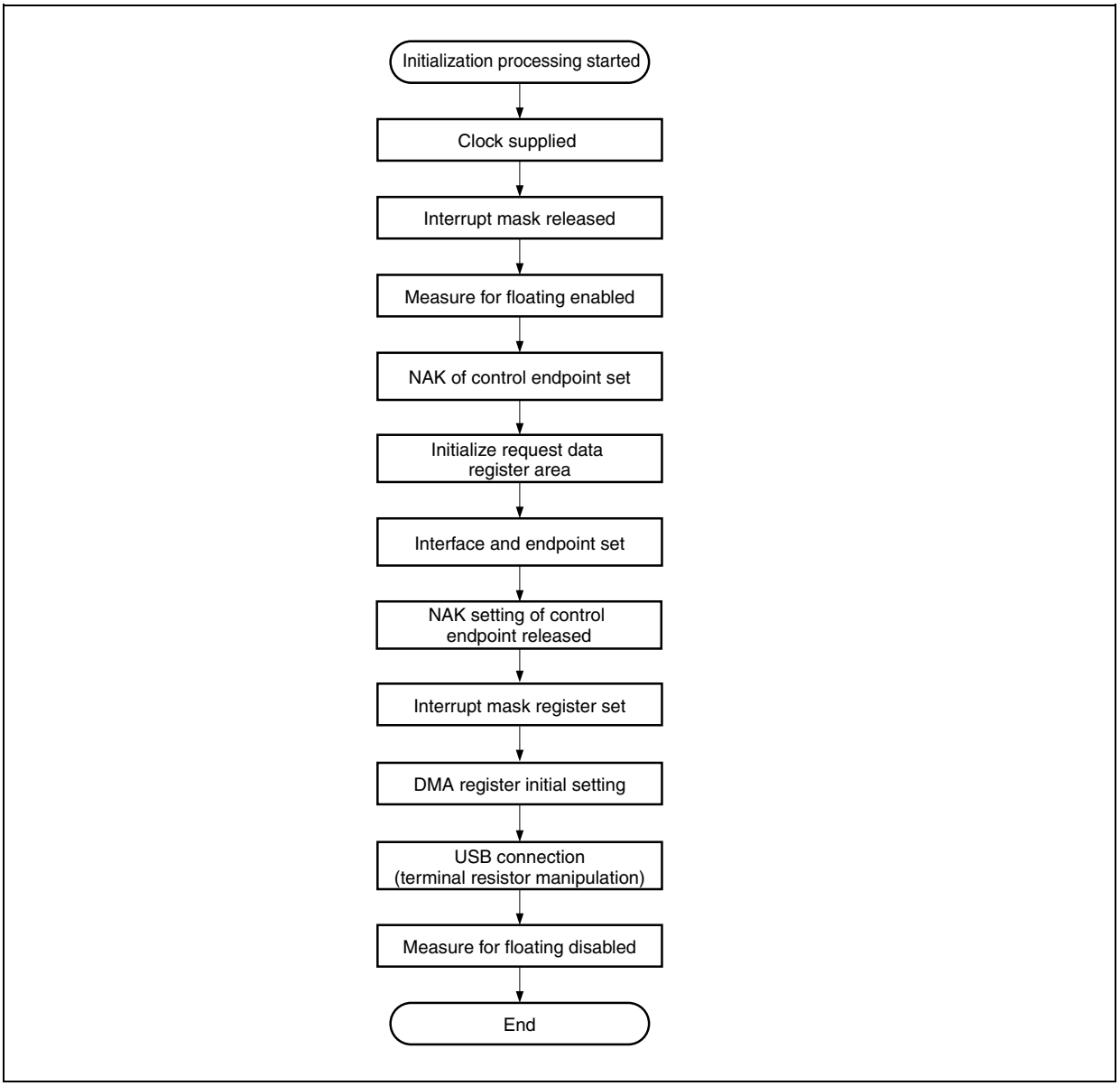
3.7.2 Processing flows

The processing flows of initialization processing, interrupt servicing, and CBW data processing in the sample program are shown below.

(1) Initialization processing

Initialization processing of the USB device is called and executed by the software initialization module.
The flow of USB device initialization processing (at power application) in the sample program is shown below.

Figure 3-10. Flowchart of Initialization Processing



The processing executed by the initialization processing is shown below.

Caution Initialization processing is required except for processing of ports. The pin assignment may differ if another target board is used. In such a case, read the descriptions in this manual making changes as necessary to match the specifications of the target board to be used.

- Clock supply
  Be sure to set the UCKC.UCKCNT bit to 1 before setting the USB function controller register. A clock to USB is supplied by setting this bit to 1.
  The P10 pin is used for inputting a clock, so set the P10 pin to input mode to enable clock input.

- Release of interrupt mask
  Masking of the USB-related interrupt signal is released using the interrupt control register.

- Enabling floating measure
  The UF0BC.UBFIOR bit is cleared to 0 to prevent mis-recognition due to a bus reset caused by an undefined value when the cable is disconnected.

- Setting of NAK for control endpoint
  A NAK response is sent to all the requests including automatic execution requests.
  This setting is made so that hardware does not return unexpected data in response to an automatic execution request until registration of data used for the automatic execution request is complete.

- Initialization of request data register area
  Descriptor data used to respond to a Get Descriptor request is registered in a register.
  Data such as device status, endpoint 0 status, device descriptor, configuration descriptor, interface descriptor, and endpoint descriptor are registered.

  Caution Registration of the descriptor for the class may be required depending on the class. Descriptors other than the USB standard descriptors are not used by the USB storage class.

- Setting of interface and endpoint
  Information such as the number of supported interfaces, the state of alternative settings, relationship between the interface and endpoints are set to a register.

- Release of NAK setting at control endpoint
  The NAK setting at control endpoint (endpoint 0) is released when registration of data for an automatic execution request is complete.

- Setting of interrupt mask register
  Masking for each interrupt source shown in the interrupt status register of the USB function controller.

- Setting of DMA register initialization
  The DMA initialization module is called for each endpoint that uses DMA and initialization is performed.

- USB connection (terminal resistor manipulation)
  The D+ signal is pulled up.

- Disabling floating measure
  The floating measure is disabled by setting the UF0BC.UBFIOR bit to 1.

(2) Interrupt servicing

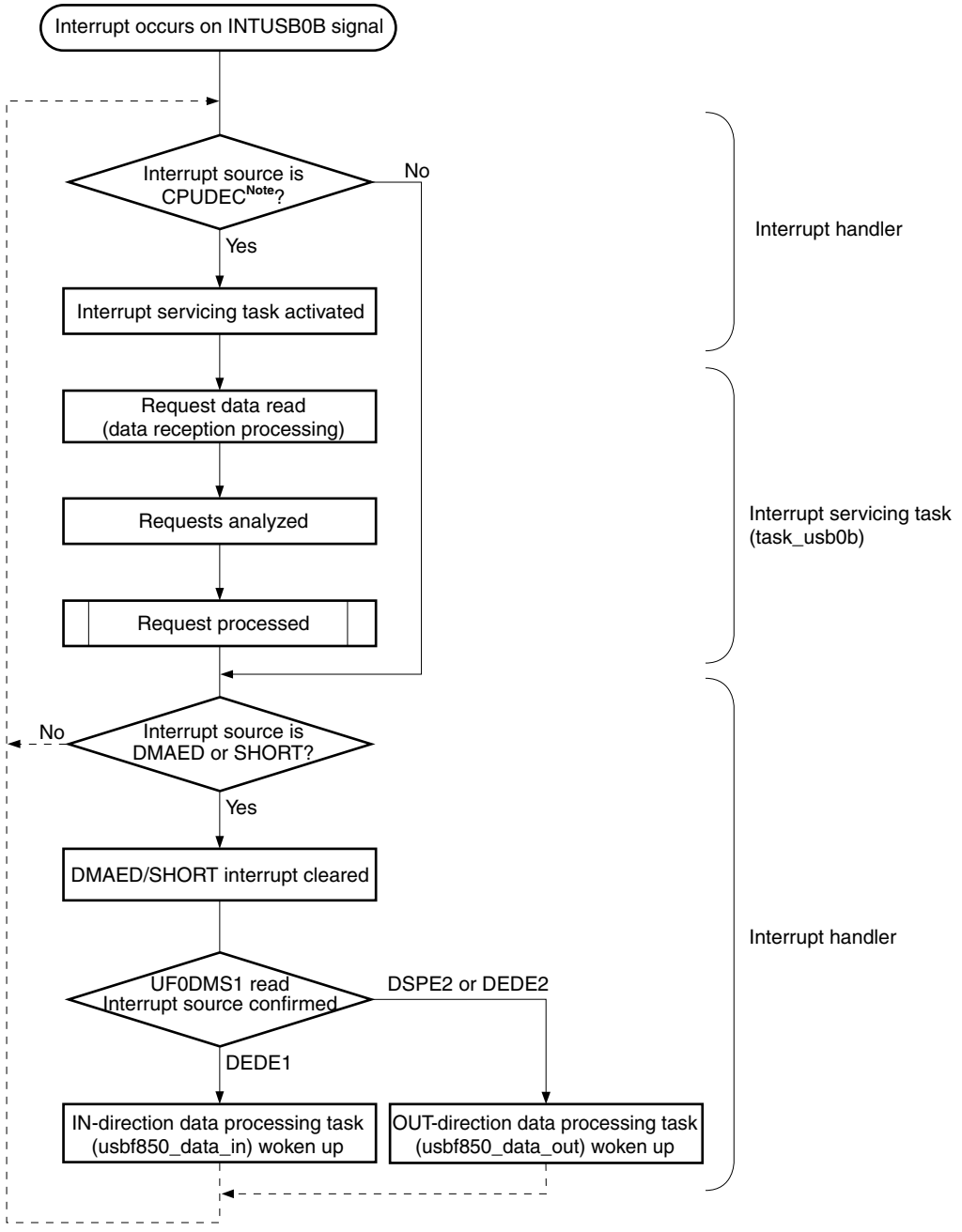
The sample program operates by interrupt events after initialization. The device is in the idle state as long as no event occurs. No events are activated from the storage device; all the events are activated by the host driver.

Figures 3-11 and 3-12 show the interrupt servicing flows in the sample program.

Caution The flowchart in Figure 3-11 illustrates the flow of interrupt servicing reported by the INTUSB0B signal of the USB function controller.

The flowchart in Figure 3-12 illustrates the flow of interrupt servicing reported by the INTUSB1B signal of the USB function controller.

Figure 3-11. Flowchart of Interrupt Servicing (1)



Note The CPUDEC, DMAED, and SHORT interrupts can be acknowledged using the INTUSB0B signal of the USB storage class driver.

The processing of an interrupt by the INTUSB0B signal in the sample program is shown below.

[Processing in interrupt handler]

- Confirmation of interrupt source
  In this sample program, the analyzed interrupt status varies depending on the executed interrupt handler.
  The CPUDEC, DMAED, and SHORT interrupts can be acknowledged using the INTUSB0B signal. When these interrupts occur, the interrupt handler is activated by the INTUSB0B signal. This interrupt handler reads the UF0IS1 register and judges if the interrupt source is CPUDEC interrupt or not. Furthermore, the interrupt handler reads the UF0IS0 register to confirm whether or not the interrupt source is DMAED or SHORT.

  Caution In this sample program, the interrupt handlers to be used are registered in the CF definition file in advance.

- Activation of interrupt servicing task
  The *task_usb0b* task is activated if the interrupt source is CPUDEC.

  Caution In this sample program, the tasks to be activated are registered in the CF definition file in advance.

- Wake-up of *usbf850_data_in* and *usbf850_data_out* tasks
  The *usbf850_no_data*, *usbf850_data_in*, and *usbf850_data_out* tasks perform SCSI command processing. These tasks are activated when a normal CBW is received by the BKO1DT interrupt. Among them, the *usbf850_data_in* and *usbf850_data_out* tasks enter sleep mode after DMA is activated. If the interrupt source is DMAED or SHORT, the interrupt handler for the INTUSB0B signal reads the UF0DMS1 register and checks if the source is DEDE1, DSPE2, or DEDE2.
  The *usbf850_data_in* task is woken up if the source is DEDE1, and the *usbf850_ data_out* task is woken up if the source is DSPE2 or DEDE2.

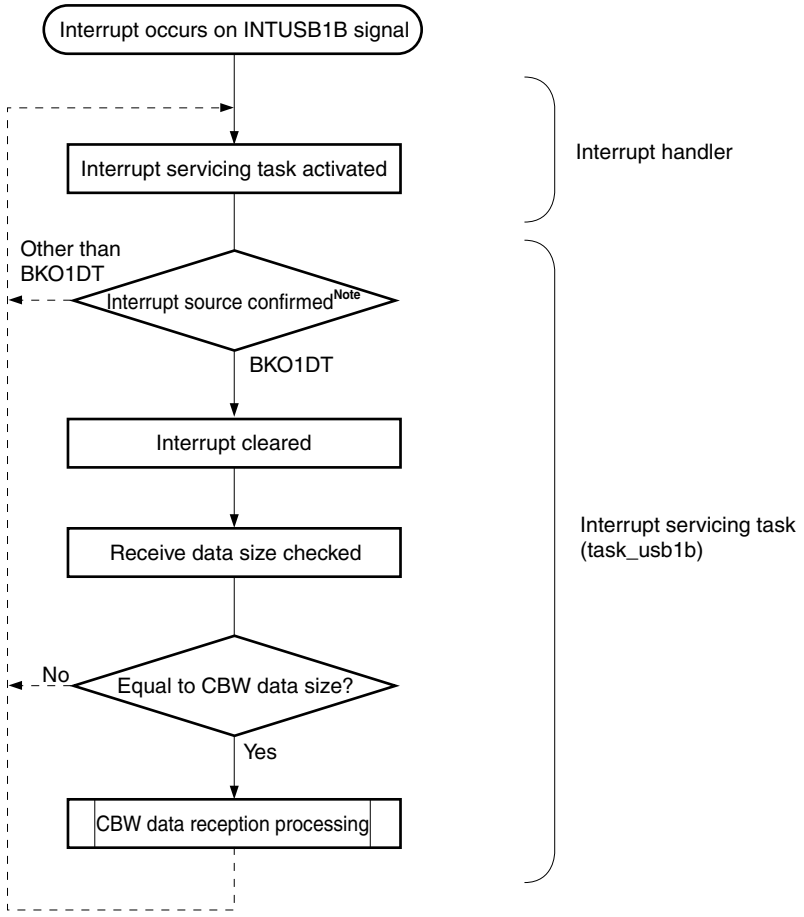
[Processing in task_usb0b task]

- Reading request data
  SETUP data is read from the UF0E0ST register.

- Analysis of request
  SETUP data that has been read is analyzed and the purpose of the request is confirmed.

- Processing of requests
  Processing of the analyzed request is performed.
  In the sample program, the standard device request Get Descriptor (String Descriptor) and device class-specific request are handled.

Figure 3-12. Flowchart of Interrupt Servicing (2)



Note Only the BKO1DT interrupt can be acknowledged using the INTUSB1B signal of the USB storage class driver.

The processing of an interrupt by the INTUSB1B signal in the sample program is shown below.

- Activation of interrupt servicing task
  The *task_usb1b* task is activated without confirming the interrupt source.

  Caution In this sample program, the tasks to be activated are registered in the CF definition file in advance.

- Confirmation of interrupt source
  Only the BKO1DT interrupt can be acknowledged using the INTUSB1B signal. When this interrupt occurs, the interrupt handler is activated by the INTUSB1B signal.
  The interrupt handler does not check the interrupt source, but the activated task confirms that the interrupt source is BKO1DT.

  Caution In this sample program, the interrupt handlers to be used are registered in the CF definition file in advance.

- Checking receive data size

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

  Remark Refer to **(3) CBW data processing** for details of CBW processing.

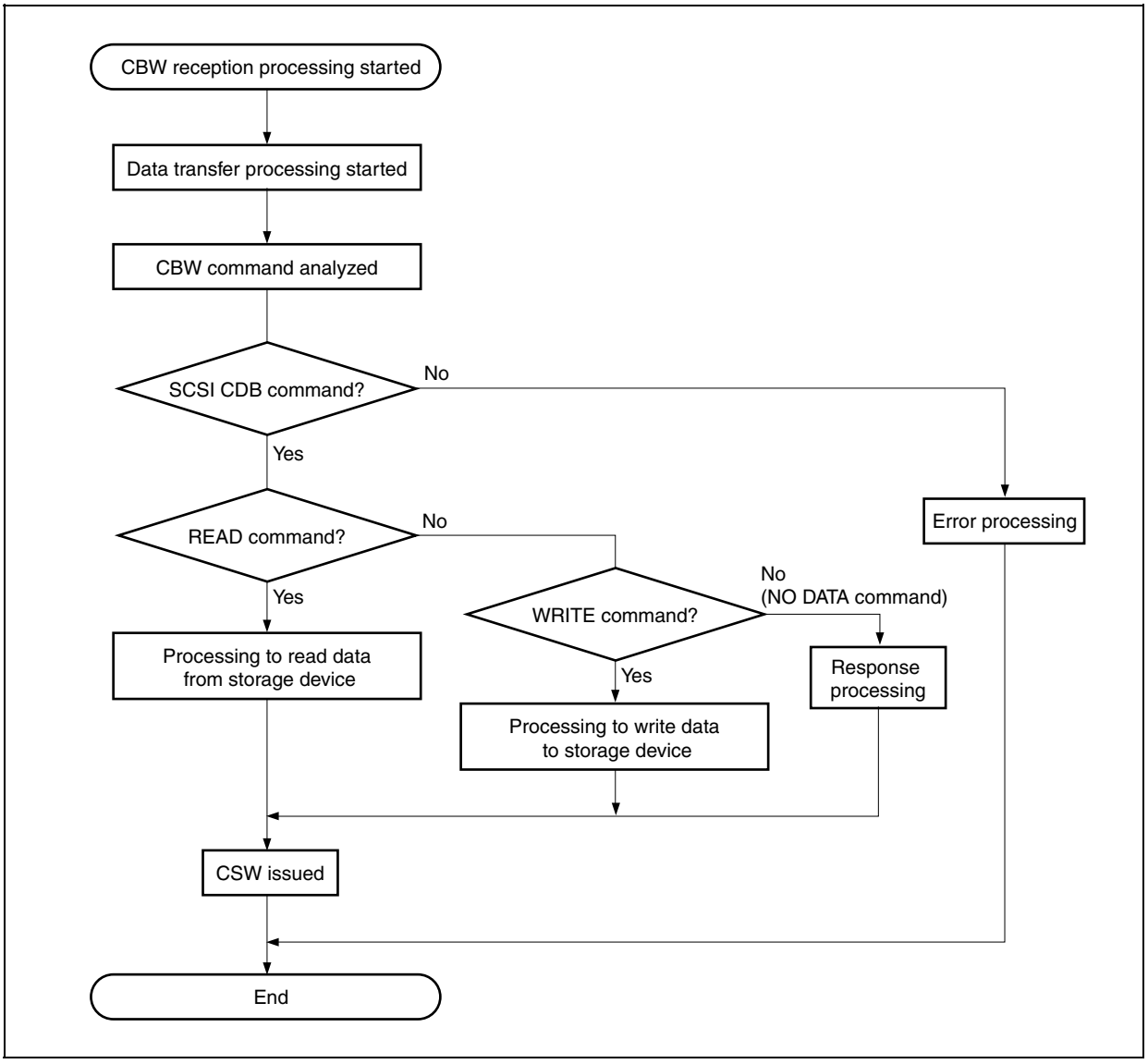
(3) CBW data processing

CBW data processing is started when CBW data is received via the USB.

A flow of CBW data processing is shown below.

Table 3-1 shows the CBW data format, and Table 3-2 shows the CSW data format.

Figure 3-13. Flowchart of CBW Data Processing



CBW data processing in the sample program is shown below.

- Analysis of CBW command
  After CBW data reception, the contents of CBW data are analyzed.
  The CBW tag is saved, the number of valid CBWCB data and the command direction is checked, and a task for processing the READ, WRITE, or NO DATA command is activated.

- READ command processing
  A task (usbf850_data_in) that performs processing of an SCSI READ command is activated.
  This task calls the SCSI command processing module, judges the CSW data transmit status according to the execution result, and calls CSW issuance processing.

- WRITE command processing
  A task (usbf850_data_out) that performs processing of an SCSI WRITE command is activated.
  This task calls the SCSI command processing module, judges the CSW data transmit status according to the execution result, and calls CSW issuance processing.

- NO DATA command processing
  A task (usbf850_no_data) that performs processing of an SCSI NO DATA command is activated.
  This task calls the SCSI command processing module, judges the CSW data transmit status according to the execution result, and calls CSW issuance processing.

- Issuance of CSW
  This processing is called by each command processing task with the command execution result as an argument.
  This processing generates CSW data from the argument and transmits it.

[CBW format]

CBW consists of 31-byte data as shown below.

The processing performed by the host can be judged using the CBWCB value.

Table 3-1. 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 (tag of CBW subject to processing)							
8 to 11	dCBWDataTransferLength (transfer data length)							
12	bmCBWFlag (Data-OUT/IN specification)							
13	Reserved				bCBWLUN (target device number)			
14	Reserved			bCBWCBLength (number of valid bytes of CBWCB)				
15 to 30	CBWCB (command)							

[CSW format]

CSW consists of 13-byte data as shown below.

Table 3-2. 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 (tag of CBW subject to processing)							
8 to 11	dCSWDataResidue (difference between transfer data length specified for CBW and processed data length)							
12	bmCSWStatus (status after CBW processing)							

(4) SCSI command processing

SCSI command processing is started when CBW data reception processing is performed on the USB side.
In the sample program, the 19 types of SCSI CDB commands shown in Table 3-3 are supported.
Figure 3-14 shows the flow of SCSI command (READ command) processing.

Caution The sample program only provides the minimum commands required for the sample program operation. Add the necessary processing concerning commands not included in the sample program and how to generate response data and its processing according to the user environment.

Table 3-3. SCSI Command List

Command	Code	Operation
READ		
REQUEST SENSE	03H	Transfers SENSE data to the host.
READ (6)	08H	Transfers data in the specified range of the logical data block to the host.
INQUIRY	12H	Reports configuration information and attributes of the target and logical unit to the host.
MODE SENSE (6)	1AH	Reads the mode select parameter value and attributes of the logical unit.
READ FORMAT CAPACITIES	23H	Reports the logical unit capacity (number of blocks and block length) 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		
WRITE (6)	0AH	Writes data sent from the host to a specified block in the medium.
MODE SELECT (6)	15H	Sets and changes parameters such as the data format of the logical unit.
WRITE (10)	2AH	Same as WRITE (6).
WRITE VERIFY	2EH	Writes data to the medium and reads it to check the data validity.
VERIFY	2FH	Checks the validity of data in the medium of the drive unit.
WRITE_ BUFF	3BH	Writes data to the memory in the target.
MODE_SELECT (10)	55H	Same as MODE SELECT (6).
NO DATA		
TEST UNIT READY	00H	Reports the logical unit status to the initiator (host device).
SEEK	0BH	Performs seek operation on the specified position in the record medium.
START STOP UNIT	1BH	Enables/disables access to the logical unit medium.
SYNCHRONIZE CACHE	35H	Matches the value of the cache memory and medium for the specified range of the block.

Figure 3-14. Flowchart of READ Command Processing (1/2)

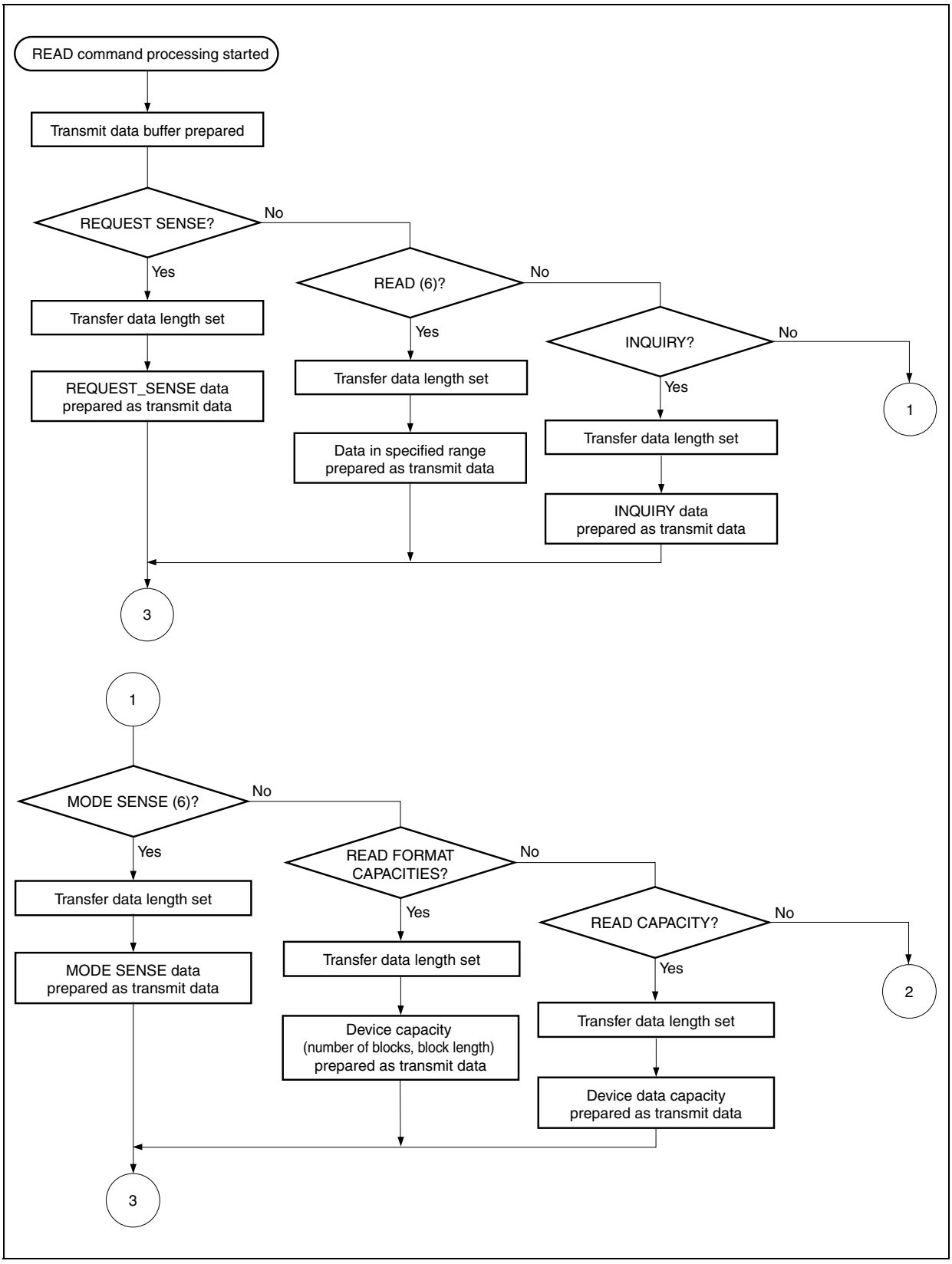
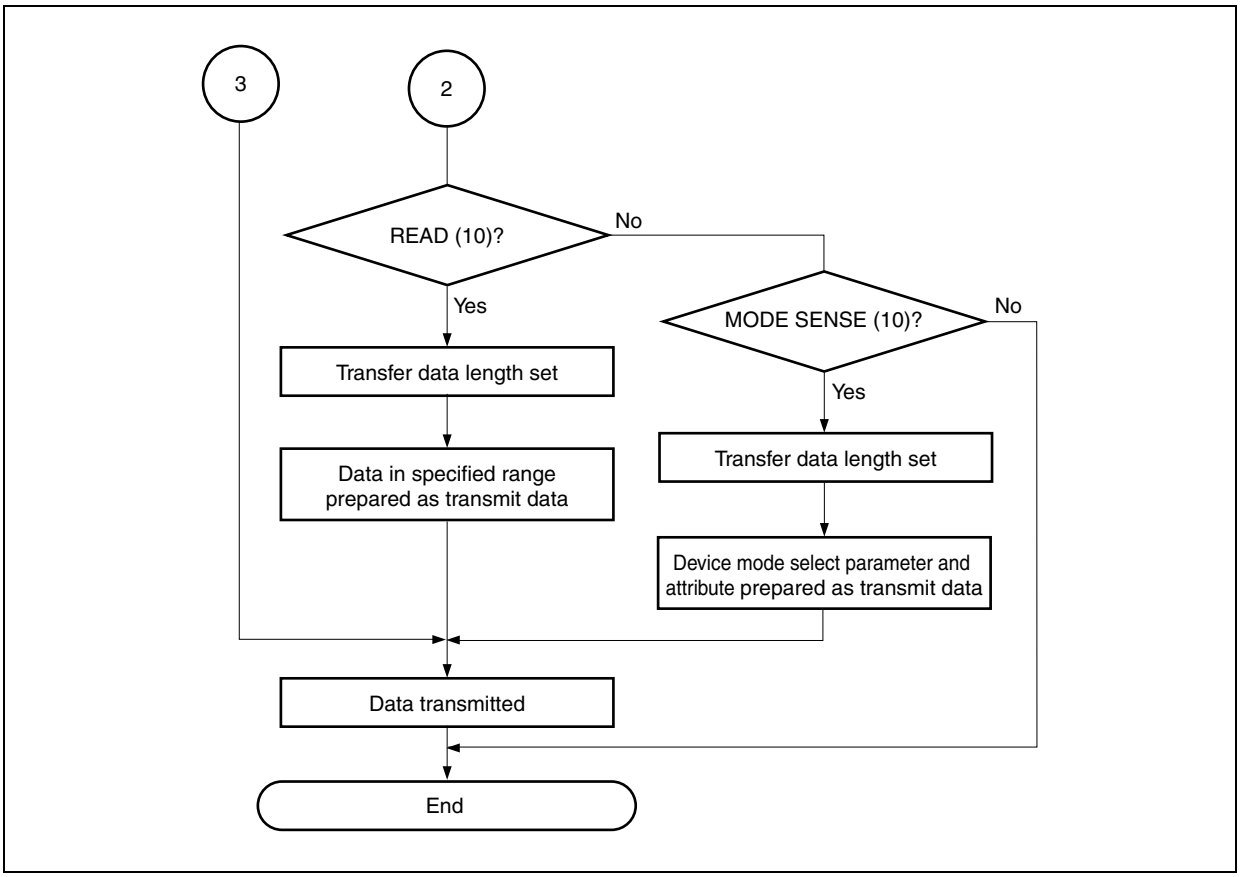


Figure 3-14. Flowchart of READ Command Processing (2/2)



[REQUEST SENSE command processing]

Sense data is reported to the host.

The sense data format and data values used in the sample program are shown below. Since the virtual device is used in the sample program, the data values shown below are returned.

Table 3-4. 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

A list of sense keys transmitted to the host in the sample program is shown below.

Table 3-5. Sense Key List

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]

Data is read from the specified range in the storage device and sent to the host.

In the sample program, data read from the virtual device is sent to the host.

[INQUIRY command processing]

Information on the device is reported to the host.

The INQUIRY data format is shown below. Since the virtual device is used in the sample program, the data values shown below are returned.

Table 3-6. 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	ISO 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 version (ASCII)								**Note 3**
:	:								**Note 3**
35	Product version (ASCII)								**Note 3**
36	Vendor-specific information								None
:	:								None
55	Vendor-specific information								None
56	Reserved								None
:	:								None
95	Reserved								None
96	Vendor-specific information								None
:	:								None
n	Vendor-specific information								None

Notes 1. ASCII character code for “NEC Corp”

2. ASCII character code for “StorageFncDriver”

3. ASCII character code for “0.12”

[MODE SENSE (6) command processing]

The mode select parameters and attributes for the device are reported to the host.

The MODE SENSE data format is shown below. Since the virtual device is used in the sample program, the data values shown below are returned. The supported page code is 01H only, so 01H is returned regardless of the command page code.

Table 3-7. MODE SENSE Data Format

Bit / Byte	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 smaller number of bytes between the parameter list specified by the page code of CDB and DBD and the parameter list specified by the allocation length

2. Page code for CDB

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

[READ FORMAT CAPACITY command processing]

The capacity (number of blocks and block length) of the device is reported to the host.

The READ FORMAT CAPACITY data format is shown below. Since the virtual device is used in the sample program, the data values shown below are returned.

Table 3-8. 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 (byte)								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 data capacity of the device is reported to the host.

The READ CAPACITY data format is shown below. Since the virtual device is used in the sample program, the data values shown below are returned.

Table 3-9. READ CAPACITY Data Format

Bit / Byte	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]

Data is read from the specified range in the storage device and sent to the host.

In the sample program, data read from the virtual device is sent to the host.

[MODE SENSE (10) command processing]

The mode select parameters and attributes for the device are reported to the host.

The MODE SENSE (10) data format is shown below. Since the virtual device is used in the sample program, the data values shown below are returned. The supported page code is 01H only, so 01H is returned regardless of the command page code.

Table 3-10. 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**

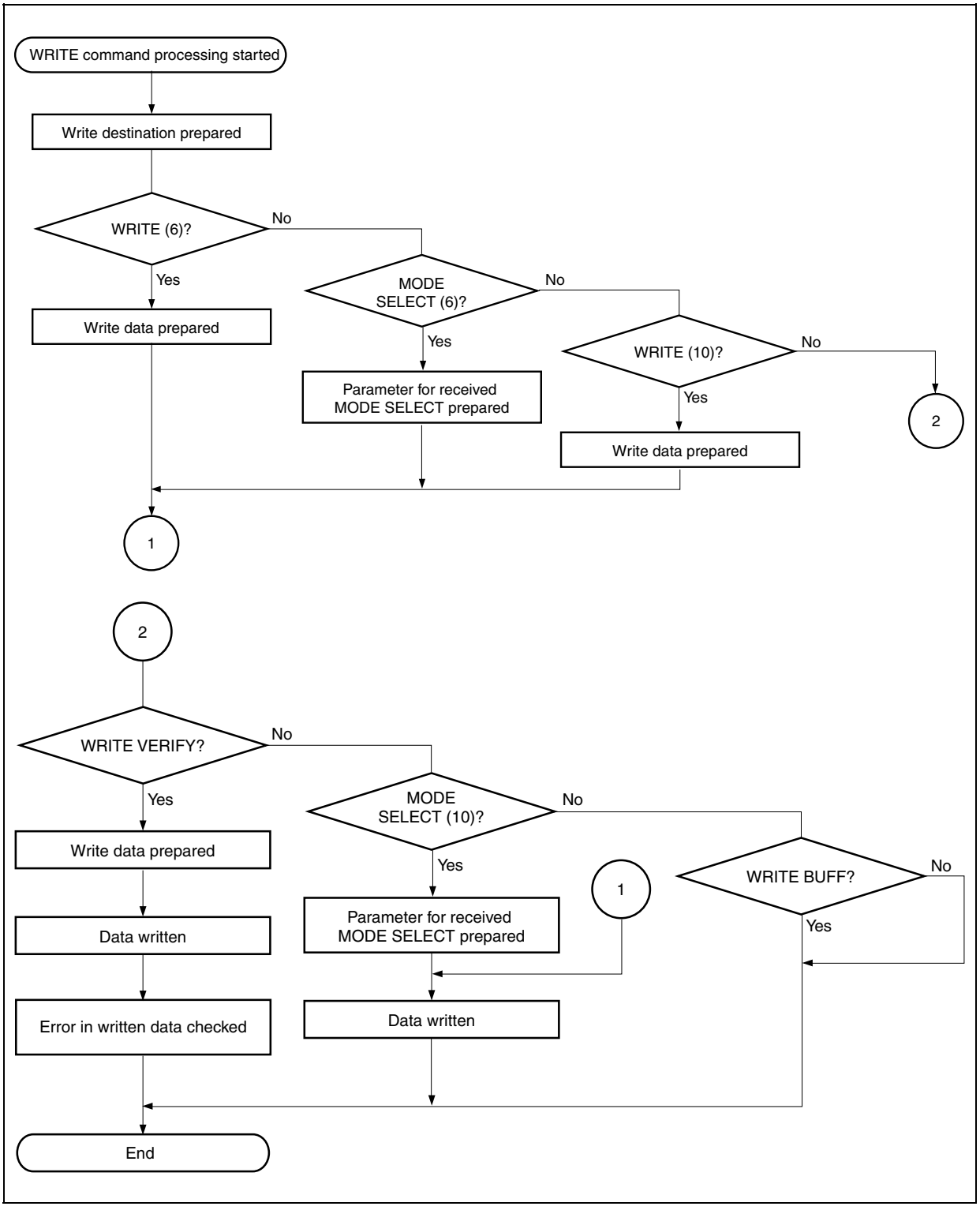
Notes 1. The smaller number of bytes between the parameter list specified by the page code of CDB and DBD and the parameter list specified by the allocation length

2. Page code for CDB

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

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

Figure 3-15. Flowchart of WRITE Command Processing



[WRITE (6) command processing]

Receive data is written to the specified area in the storage device.

In the sample program, receive data is written to the specified area in the virtual device.

[MODE SELECT (6) command processing]

Parameters such as the physical attribute of the logical unit, data format on the recording medium, and how to recover from errors are set or changed

The MODE SELECT data format is shown below. Since the virtual device is used in the sample program, the program just writes the receive data to the MODE SELECT TABLE regardless of the command page code, and terminates normally. The program supposes the data values shown below are used as the initial values of the table.

Table 3-11. 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. Page code for CDB

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

[WRITE (10) command processing]

Receive data is written to the specified area in the storage device.

In the sample program, receive data is written to the specified area in the virtual device.

[WRITE VERIFY command processing]

Receive data is written to the storage device. The written data is checked for errors. In the sample program, receive data is written to the virtual device but a data error check is not performed and the program terminates normally.

[VERIFY command processing]

The validity of data in the storage device is checked. Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

[WRITE BUFF command processing]

Data is written to the memory (data buffer). Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

[MODE SELECT (10) command processing]

Parameters such as the physical attribute of the logical unit, data format on the recording medium, and how to recover from errors are set or changed.

The MODE SELECT (10) data format is shown below. Since the virtual device is used in the sample program, the program just writes the receive data to the MODE SELECT (10) TABLE regardless of the command page code, and terminates normally. The program supposes the data values shown below are used as the initial values of the table.

Table 3-12. MODE SELECT (10) Data Format

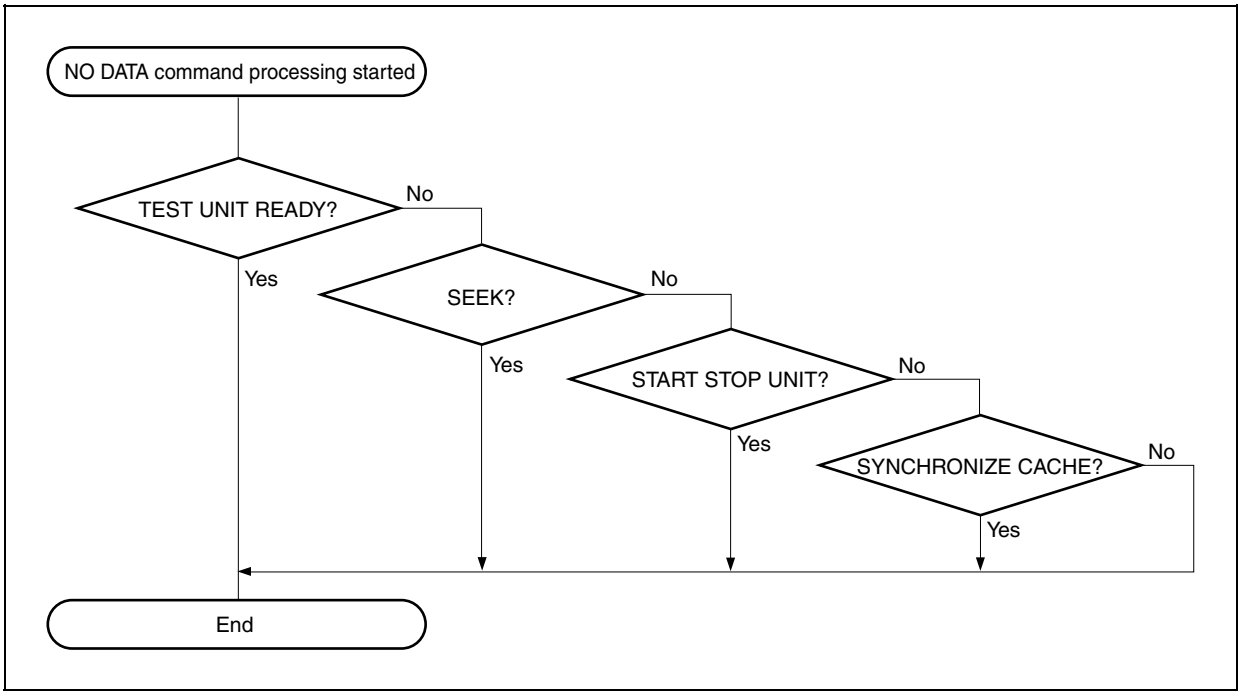
Bit / Byte	7	6	5	4	3	2	1	0	Data Value
0	Mode parameter length								00H
1	Mode parameter length								1AH
2	Media type								00H
3	Device-specific parameter								00H
4	Reserved								00H
5	Reserved								00H
6	Block descriptor length								00H
7	Block descriptor length								08H
8	Density code								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. Page code for CDB

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

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

Figure 3-16. Flowchart of NO DATA Command Processing



[TEST UNIT READY command processing]

The unit status is reported. Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

[SEEK command processing]

A seek operation is performed on the specified block position. Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

[START STOP UNIT command processing]

A setting to restrict access to the unit is performed. Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

[SYNCHRONIZE CACHE command processing]

Data in the unit and cache are matched. Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

3.7.3 USB storage class driver descriptor information

The USB standard descriptors defined in this sample program are shown below.
Descriptors described in (a) to (d) are the minimum required descriptors.

Remark Refer to **Universal Serial Bus Specification Revision 1.1** for details.

(a) Device descriptor

This descriptor holds general information of the device. One device descriptor must be prepared for each device. The information contained in this descriptor is used for identifying a unique in the device configuration. Concrete information is not used at the device level in the current USB storage class driver.

Table 3-13. Device Descriptor

Offset	Size (Byte)	Value	Description
0	1	12H	Length value of this descriptor (byte)
1	1	01H	Descriptor type (device)
2	2	10H/01H	USB version (USB 1.1)
4	1	00H	Class code
5	1	00H	Sub-class code
6	1	00H	Protocol code
7	1	40H	Maximum packet size at endpoint 0
8	2	09H/04H	Vendor ID (NEC Electronics)
10	2	FCH/FFH	Product ID
12	2	01H/00H	Device release number
14	1	01H	Index to string descriptor (Manufacturer)
15	1	00H	Index to string descriptor (Product)
16	1	00H	Index to string descriptor (Serial Number)
17	1	01H	Number of devices that can be configured

(b) Configuration descriptor

This descriptor holds information on concrete device configuration.

Concrete information is not used at the configuration level in the current USB storage class driver.

Table 3-14. Configuration Descriptor

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

(c) Interface descriptor

This descriptor holds concrete interface information in the configuration.

The configuration provides one interface in this sample program. This interface supports two endpoints, and therefore has two endpoint descriptors.

This descriptor is always returned as a part of the configuration descriptor, and is not accessed directly by a Get Descriptor request or Set Descriptor request.

Table 3-15. Interface Descriptor

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	04H	Descriptor type (interface)
2	1	00H	Interface value
3	1	00H	*Alternate* set value
4	1	02H	Endpoint number (excluding endpoint 0)
5	1	08H	Interface class (mass storage class)
6	1	06H	Interface sub-class (SCSI)
7	1	50H	Interface protocol (Bulk-Only)
8	1	00H	Index to string descriptor (interface)

(d) Endpoint descriptor

This descriptor holds information required by the host for determining the bandwidth requirements for each endpoint.

This descriptor is always returned as a part of the configuration descriptor, and is not accessed directly by a Get Descriptor request or Set Descriptor request.

Table 3-16. Endpoint Descriptor (Bulk IN)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (byte)
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 at endpoint
6	1	00H	Interval (ms): Valid only for isochronous and interrupt endpoints

Table 3-17. Endpoint Descriptor (Bulk OUT)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (byte)
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 at endpoint
6	1	00H	Interval (ms): Valid only for isochronous and interrupt endpoints

(e) String descriptor

This descriptor holds information on the manufacturer of the device in this sample program.

Table 3-18. String Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	04H	Length value of this descriptor (byte)
1	1	03H	Descriptor type (string)
2	2	09H/04H	Language type used by string descriptor (English/US)

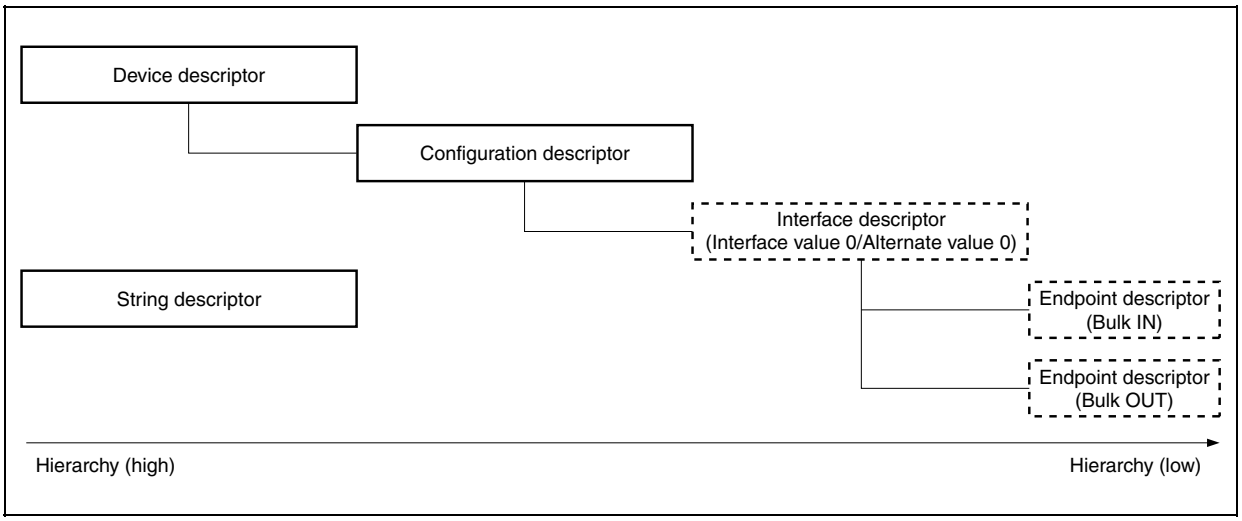
Table 3-19. String Descriptor (2)

Offset	Size (Byte)	Value	Description
0	1	2AH	Length value of this descriptor (byte)
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: NEC Electronics Co.

- Descriptor configuration
  The descriptor configuration in this sample program is shown below. This configuration consists of the five descriptors described before.

Caution The device descriptor, configuration descriptor, and string descriptor are accessed by an independent Get Descriptor request. The interface descriptor and endpoint descriptor are accessed as part of the configuration descriptor.

Figure 3-17. Descriptor Configuration



3.7.4 Data macro

The data macros (data type, return value, etc.) used by the USB storage class driver are shown below.

(1) Data type

Data type macro for parameters specified when a USB storage class driver function is called is defined in the header file *types.h* in nectools32\USB_Storage\inc.

A list of the data types is shown below.

Table 3-20. List of Data Types

Macro	Type	Description
ULONG	unsigned long	32-bit unsigned integer
WORD	unsigned long	32-bit unsigned integer
HWORD	unsigned short	16-bit unsigned integer
BYTE	unsigned char	8-bit unsigned integer
(*PFV) ()	void	Processing module activation address

(2) Return value

Macro of the return value from USB storage class driver function is defined in the header file *errno.h* in nectools32\USB_Storage\inc.

A list of the return values is shown below.

Table 3-21. List of Return Values

Macro	Type	Description
DEV_OK	0	Normal termination
DEV_ERROR	–1	Abnormal termination
DEV_ERR_NODATA	–2	Error in transfer direction for NO DATA command
DEV_ERR_READ	–3	Error in transfer direction for READ command
DEV_ERR_WRITE	–4	Error in transfer direction for WRITE command
DEV_ERR_VERIFY	–5	Verify error
DEV_ERR_CBWLENGTH	–6	CBW length error
DEV_ERR_CBWCBW	–7	CBW is received during CBW processing (error)

3.7.5 Data structure

The data structure used by the USB storage class driver is shown below.

(1) USB device request structure

The USB device request structure is defined in USB header file *usbf850.h* in nectools32\V850USB_Storage\src\USBF. 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;
```

(2) CBW data structure

The CBW (Command Block Wrapper) data structure handled in the USB storage class driver is defined in header file *types.h* in nectools32\USB_Storage\inc. 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;              /*number of valid bytes of
                                                     CBWCB*/
         unsigned char  CBWCB[16];                 /*CBWCB (command) */
} CBW_INFO,*PCBW_INFO;
```

(3) CSW data structure

The CSW (Command Status Wrapper) data structure handled in the USB storage class driver is defined in header file *types.h* in nectools32\USB_Storage\inc. 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 transfer data
                                           length specified for CBW and
                                           processed data length */
      unsigned char  bmCSWStatus;         /*status after CBW processing*/
} CSW_INFO,*PCSW_INFO;
```

3.7.6 Description of functions

(1) Overview

A list of the processing modules described in this chapter is shown below.

Table 3-22. List of Processing Modules in Sample Program (1/3)

Processing Module Name	Function Name	File Name	Remark
RX850 Pro-dependent processing module			
CF definition file	–	sys.cf	–
Entry processing	–	entry.850	Assembly language
Boot processing	boot	boot.850	Assembly language
Hardware initialization module	__InitSystemTimer	init.c	C language
Initialization handler	varfunc	varfunc.c	C language
Header file	–	init.h	–
Board-dependent processing module			
Port initialization	port850_reset	port.c	C language
Header file	–	port.h	–
Header file			
Data type declaration	–	types.h	–
Return value declaration	–	errno.h	–
Build file	–	usb_bus.bld	–
Section map file	–	common.lx	–

Table 3-22. List of Processing Modules in Sample Program (2/3)

Processing Module Name	Function Name	File Name	Remark
USB storage class driver processing module (USB processing module)			
Initialization function	usbf850_init	usbf850.c	C language
Interrupt handler (INTUSB0B signal)	usbf850_inthdr	usbf850.c	C language
Interrupt handler (INTUSB1B signal)	usbf850_inthdr1	usbf850.c	C language
Interrupt handler (INTUSB2B signal)	usbf850_inthdr2	usbf850.c	C language
Interrupt servicing task (INTUSB0B signal)	task_usb0b	usbf850.c	C language
Interrupt servicing task (INTUSB1B signal)	task_usb1b	usbf850.c	C language
Interrupt servicing task (INTUSB2B signal)	task_usb2b	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 calling function (loc_cpu)	usbf850_loc_cpu	usbf850.c	C language
System call calling 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 (processing of device class-specific request)	usbf850_blkonly_mass_ storage_reset	usbf850_storage.c	C language
Max LUN request processing function (processing of device class-specific request)	usbf850_max_lun	usbf850_storage.c	C language
Registration processing function of USB storage class device class-specific request processing function	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
USB-storage interface function header file	–	usbf850_storage.h	–
DMA initialization module 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 3-22. List of Processing Modules in Sample Program (3/3)

Processing Module Name	Function Name	File Name	Remark
Storage device processing modules			
Storage device initialization module	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
Function for processing of data transmission from SCSI to USB	scsi_to_usb	scsi_cmd.c	C language
SCSI command processing header file	–	scsi.h	–
Function macro			
V850E/ME2 peripheral I/O register setting function (1-byte units: 8 bits)	USBF850REG_SET	usbf850.h	C language
V850E/ME2 peripheral I/O register read function (1-byte units: 8 bits)	USBF850REG_READ	usbf850.h	C language
V850E/ME2 peripheral I/O register setting function (1-word units: 16 bits)	USBF850REG_SET_W	usbf850.h	C language
V850E/ME2 peripheral I/O register read function (1-word units: 16 bits)	USBF850REG_READ_W	usbf850.h	C language

(2) Function tree

The function calling relationship in the sample program is illustrated below.

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

Figure 3-18. Sample Program Function Tree (1/4)

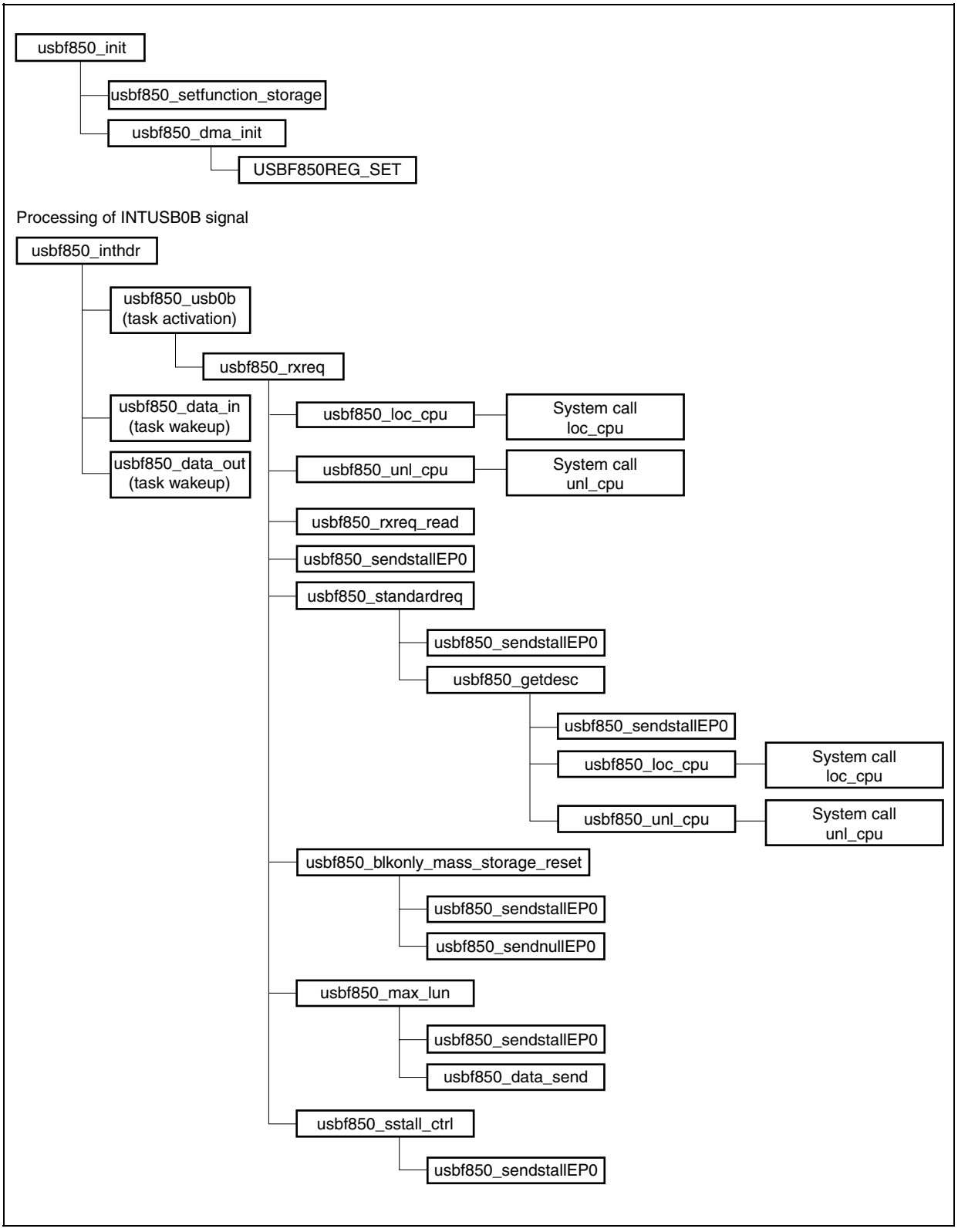


Figure 3-18. Sample Program Function Tree (2/4)

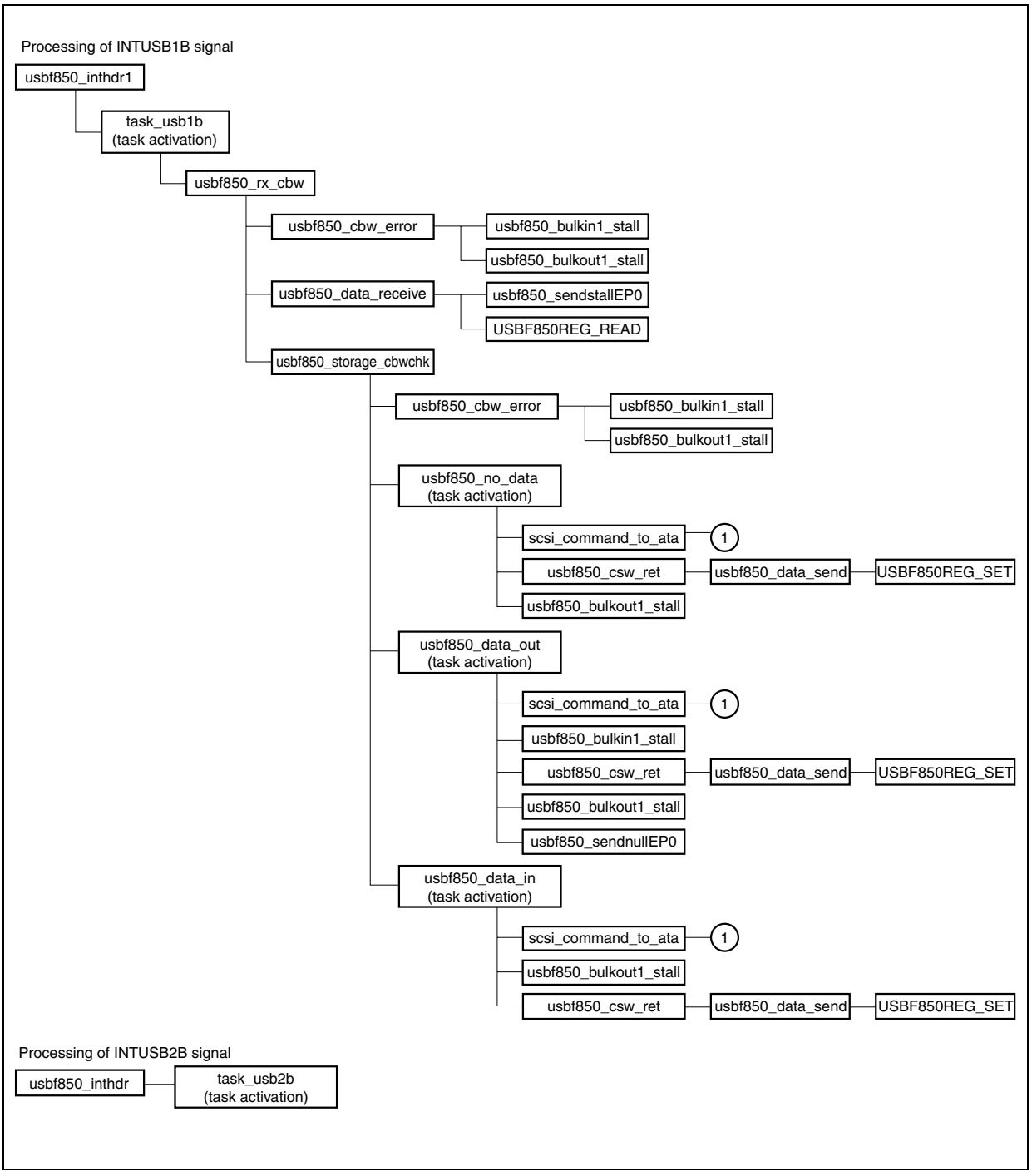


Figure 3-18. Sample Program Function Tree (3/4)

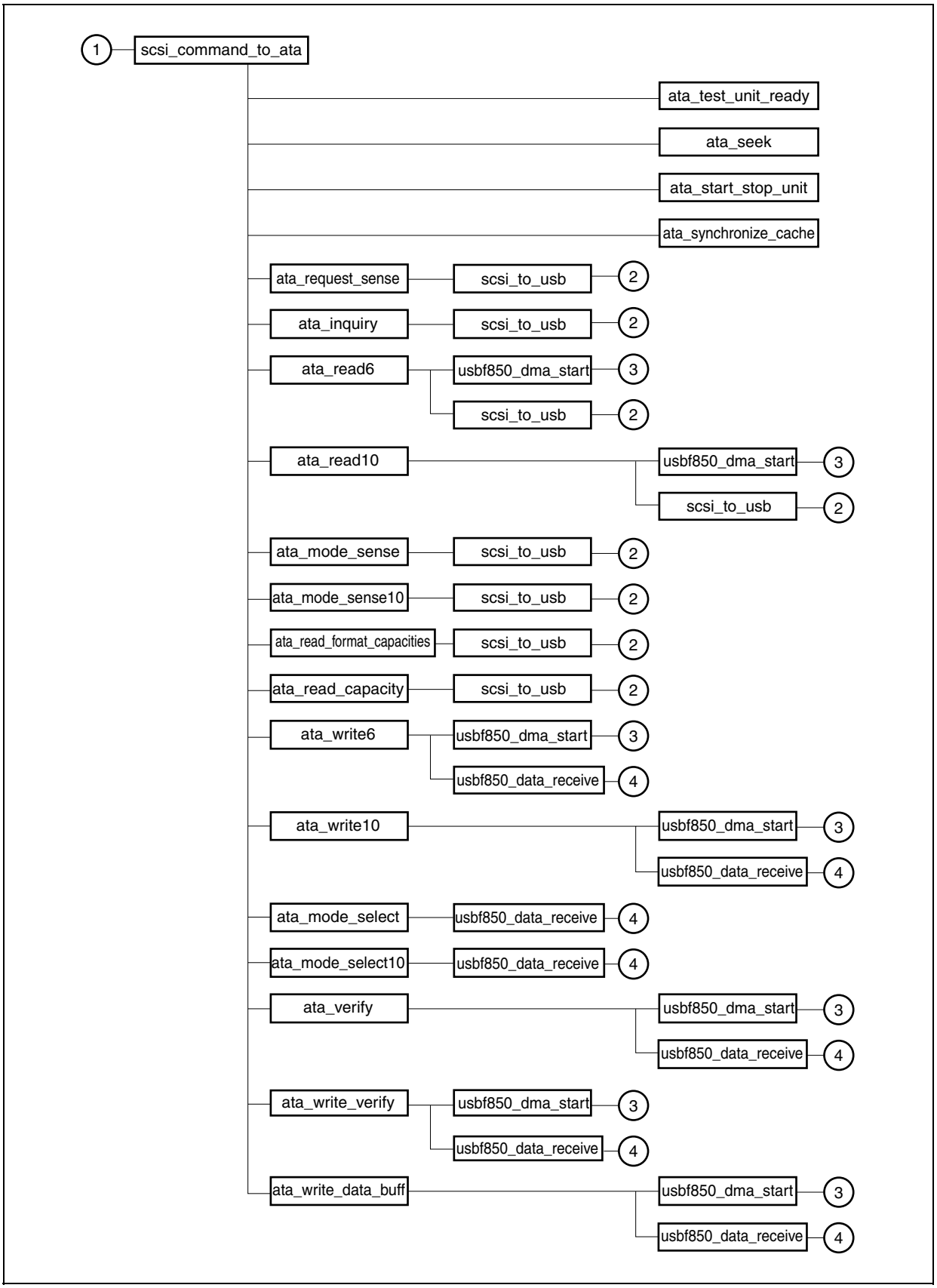
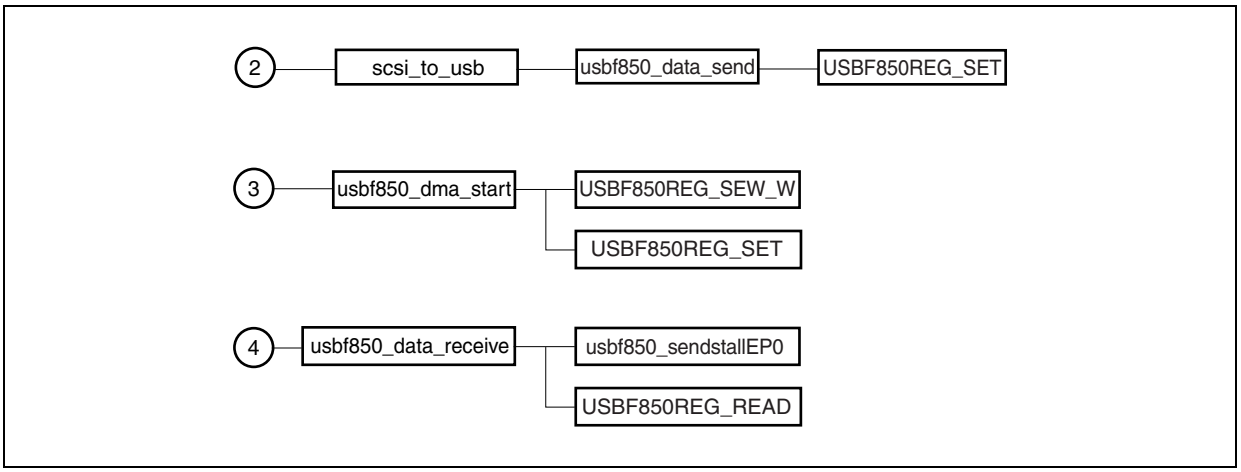


Figure 3-18. Sample Program Function Tree (4/4)



(3) Description of functions

The functions in this sample program are explained in the following format.

xxxx … <1>	Valid caller: – – – – … <2>

[Outline] … <3>

– –
– –

[C language format] … <4>

– – – – – – – –

[Parameter] … <5>

I/O	Parameter	Description

[Operation] … <6>

– –
– –

[Return value] … <7>

– –
– –

<1> Name

Indicates the function name.

<2> Valid caller

Indicates the type of the processing module from which a function can be called.

Task:	The function can be called only from a task.
Non-task:	The function can be called only from a non-task.
Non-task | Task:	The function can be called from a task or non-task.
–:	Interrupt handler or task, and is not used to call functions.

<3> Outline

Shows the outline of a function operation.

<4> C language format

Shows the description format when calling a function from the processing module described in the C language.

<5> Parameter

Shows the function parameter 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 parameter

<6> Operation

Describes detailed operation of the function.

<7> Return value

Indicates the return value from a function using the data macro or numeric value.

usbf850_init	Valid caller: Non-task I Task

[Outline]

This is a function that initializes the USB function controller incorporated in the V850E/ME2.

[C language format]

```
void usbf850_init (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called from the software initialization module and performs processing to initialize the USB function controller incorporated in the V850E/ME2.

Remark Refer to **3.7.2 (1) Initialization processing** for details of initialization processing.

[Return value]

None

usbf850_inthdr	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB0B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This is the interrupt handler activated by the INTUSB0B signal (USB function status 0).

In this sample program, the interrupt handler checks the interrupt source and activates the interrupt servicing task (task_usb0b) only when the source is the CPUDEC interrupt. If the source is the DMAED or SHORT interrupt, the interrupt handler reads the UF0DMS1 register (DMA status 1 register) to confirm the interrupt source, and wakes up the corresponding task (*usbf850_data_in* and *usbf850_data_out* are in the sleep state after DMA is started up). This handler is defined in the CF definition file.

Remark Refer to **3.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

usbf850_inthdr1	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB1B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr1 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB1B signal (USB function status 1).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb1b). This handler is defined in the CF definition file.

Remark Refer to **3.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

usbf850_inthdr2	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB2B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr2 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB2B signal (USB function status 2).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb2b). This handler is defined in the CF definition file.

Caution This handler is not called because all the interrupts reported by the INTUSB2B signal are masked in this sample program.

[Return value]

Object ID number (task ID number)

task_usb0b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB0B signal.

[C language format]

```
void task_usb0b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB0B interrupt signal (USB function status 0 interrupt). In the sample program, this task calls the *usbf850_rxreq* function and performs processing of the USB standard device request and device class-specific request.

Caution In this sample program, the standard device request Get Descriptor (String Descriptor) that is not responded automatically by the USB function controller incorporated in the V850E/ME2 is handled.

Remark Refer to **3.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

None

task_usb1b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB1B signal.

[C language format]

```
void task_usb1b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB1B interrupt signal (USB function status 1 interrupt). In the sample program, this task confirms the interrupt source and calls the function (usbf850_rx_cbw) if the interrupt source is BKO1DT and the length of the receive data is equal to the size of the CBW data.

Remark Refer to **3.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

None

task_usb2b Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB2B signal.

[C language format]

```
void task_usb2b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB2B interrupt signal (USB function status 2 interrupt). This processing is not provided in the sample program, so the program returns without processing.

Caution This function is not used in the sample program.

[Return value]

None

usbf850_data_send Valid caller: Non-task | Task

[Outline]

This is a data transmit function used by the USB function controller.

[C language format]

```
long usbf850_data_send (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of transmit data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function transmits from the endpoint specified by *ep* data whose size is specified by *len* starting from the address specified by *data*.

[Return value]

Status upon transmission

DEV_ERROR: Endpoint number is illegal

DEV_OK: Normal termination

usbf850_data_receive Valid caller: Non-task | Task

[Outline]

This is a data receive function used by the USB function controller.

[C language format]

```
long usbf850_data_receive (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of the buffer for receive data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function reads data whose size is specified by *len* from the buffer at the endpoint specified by *ep* and stores it to the address specified by specified *data*.

[Return value]

Status upon reception

DEV_ERROR: Receive data size is illegal, or endpoint number is illegal.

DEV_OK: Normal termination

usbf850_sendnullEP0	Valid caller: Non-task I Task

[Outline]

This is a function that transmits Null data from the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendnullEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function transmits Null data (whose data size is 0) from the control endpoint (endpoint 0).

[Return value]

None

usb850_sendstallEP0	Valid caller: Non-task I Task

[Outline]

This is a function that sends a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendstallEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the control endpoint (endpoint 0).

[Return value]

None

usbf850_bulkin1_stall	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 1).

[C language format]

```
void usbf850_bulkin1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 1).

[Return value]

None

usbf850_bulkout1_stall	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 2).

[C language format]

```
void usbf850_bulkout1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 2).

[Return value]

None

usbf850_loc_cpu	Valid caller: Task

[Outline]

This is a function that disables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_loc_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *loc_cpu* system call.

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

[Return value]

None

usbf850_unl_cpu	Valid caller: Task

[Outline]

This is a function that enables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_unl_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *unl_cpu* system call.

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

[Return value]

None

usbf850_rxreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs USB request processing.

[C language format]

```
void usbf850_rxreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *task_usb0b* task that is activated by the INTUSB0B interrupt signal. This function calls SETUP data read processing, analyzes the read data, and calls USB request processing based on the analysis result.

[Return value]

None

usbf850_rxreq_read	Valid caller: Non-task I Task

[Outline]

This is a function that reads USB request data.

[C language format]

```
void usbf850_rxreq_read (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function reads SETUP data received subsequently to the Setup token at the control endpoint (endpoint 0). The SETUP data is distinguished from normal data and is stored in a dedicated register. It is always read in 8-byte units.

[Return value]

None

usbf850_standardreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs the USB standard request.

[C language format]

```
void usbf850_standardreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called if the standard request is read from SETUP data and calls the *usbf850_getdesc* function when the request type is confirmed as the Get Descriptor request.

[Return value]

None

usbf850_getdesc	Valid caller: Non-task | Task

[Outline]

This is a function that performs the USB standard request Get Descriptor (String Descriptor) processing.

[C language format]

```
void usbf850_getdesc (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *usbf850_standardreq* function and performs the USB standard request Get Descriptor (String Descriptor) processing. This function sets a STALL response for a request other than the Get Descriptor (String Descriptor) request.

[Return value]

None

usbf850_sstall_ctrl	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sstall_ctrl (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response at the control endpoint (endpoint 0).

With the *usbf850_setfunction_storage* function, when a class request processing function is prepared as the function pointer for an array, this function uses a request code as a subscript for array. If this function is registered to a location where is no relevant request, a STALL response can be set when an unsupported request code is sent.

[Return value]

None

usbf850_blkonly_mass_storage_reset Valid caller: Non-task | Task

[Outline]

This is a function that handles the USB Mass Storage class-specific request Bulk-Only Mass Storage Reset.

[C language format]

```
void usbf850_blkonly_mass_storage_reset (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs Bulk-Only Mass Storage Reset request processing. When this request is received, this function initializes the storage device. In the sample program, this function clears buffers at the bulk endpoint (endpoints 1 and 2) and sets a STALL response for the bulk endpoint.

[Return value]

None

usbf850_max_lun	Valid caller: Non-task I Task

[Outline]

This is a function that handles the USB Mass Storage class-specific request Get Max LUN.

[C language format]

```
void usbf850_max_lun (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs Get Max LUN request processing. When this request is received, this function returns 1-byte data for the total number of logical units supported by the device. Since the virtual storage device is used in the sample program, 00H is prepared as data and transmitted from the control endpoint (endpoint 0).

[Return value]

None

usbf850_setfunction_storage	Valid caller: Non-task | Task

[Outline]

This is a function that registers a USB Mass Storage class-specific request processing function to an array as the function pointer.

[C language format]

```
void usbf850_setfunction_storage (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called from USB initialization processing, and registers a USB Mass Storage class-specific request processing function as a function pointer to an array (array name: Req_Func_C).

This function registers the *usbf850_sstall_ctrl* function for an unsupported request code to send a STALL response when an unsupported request is sent.

[Return value]

None

usbf850_rx_cbw	Valid caller: Non-task I Task

[Outline]

This is a function that performs CBW data reception processing.

[C language format]

```
void usbf850_rx_cbw (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the interrupt servicing task, reads CBW data, and then calls the *usbf850_storage_cbwchk* function.

Remark Refer to **3.7.2 (3) CBW data processing** for details of CBW reception processing.

[Return value]

None

usbf850_storage_cbwchk	Valid caller: Non-task | Task

[Outline]

This is a function that performs CBW data command analysis processing.

[C language format]

```
int usbf850_storage_cbwchk (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function analyzes the CBW data that has been read and activates a task that performs processing of the corresponding data direction.

Remark Refer to **3.7.2 (3) CBW data processing** for details of CBW reception processing.

[Return value]

Status upon CBW check

DEV_ERROR: CBWCB length is illegal.

DEV_OK: Normal termination

usbf850_cbw_error	Valid caller: Non-task I Task

[Outline]

This is a function that performs CBW data error processing.

[C language format]

```
void usbf850_cbw_error (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

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

[Return value]

None

usbf850_no_data	Valid caller: Non-task I Task

[Outline]

This is a task that performs SCSI NO DATA command processing.

[C language format]

```
void usbf850_no_data (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task performs SCSI NO DATA command processing. This task calls the *scsi_command_to_ata* function and passes the CBW processing status (GOOD, FAIL, or PHASE) to the CSW response processing function, according to the execution result.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

None

usbf850_data_in	Valid caller: Non-task I Task

[Outline]

This is a task that performs SCSI DATA IN command processing.

[C language format]

```
void usbf850_data_in (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task performs SCSI DATA IN command processing. This task calls the *scsi_command_to_ata* function and passes the CBW processing status (GOOD, FAIL, or PHASE) to the CSW response processing function, according to the execution result.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

None

usbf850_data_out	Valid caller: Non-task I Task

[Outline]

This is a task that performs SCSI DATA OUT command processing.

[C language format]

```
void usbf850_data_out (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task performs SCSI DATA OUT command processing. This task calls the *scsi_command_to_ata* function and passes the CBW processing status (GOOD, FAIL, or PHASE) to the CSW response processing function, according to the execution result.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

None

usbf850_csw_ret	Valid caller: Non-task I Task

[Outline]

This is a function that performs CSW response processing.

[C language format]

```
long usbf850_csw_ret (BYTE status)
```

[Parameter]

I/O	Parameter	Description
I	BYTE status	Transmitted status (GOOD, FAIL, PHASE)

[Operation]

This function performs CSW response processing. This function sends the CBW processing status (GOOD, FAIL, or PHASE) passed via an argument to the host.

[Return value]

Status upon transmission

DEV_OK: Normal termination

usbf850_dma_init Valid caller: Non-task | Task

[Outline]

This is a function that initializes DMA.

[C language format]

```
void usbf850_dma_init (char ep)
```

[Parameter]

I/O	Parameter	Description
I	char ep	No. of endpoint that is used for DMA

[Operation]

This function performs DMA initialization processing for the endpoint specified by the argument.

[Return value]

None

usbf850_dma_start Valid caller: Non-task | Task

[Outline]

This is a function that activates DMA.

[C language format]

```
void usbf850_dma_start (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Pointer of buffer in which transmit/receive data is stored
I	long len	Data length
I	char ep	No. of endpoint that is used for DMA

[Operation]

If *ep* is 1, data whose size is specified by *len* is transferred from the buffer specified by *data* to the UF0BI1 register, via DMA.

If *ep* is 2, data whose size is specified by *len* is transferred from the buffer specified by *data* to the UF0BO1 register, via DMA.

[Return value]

None

storageDev_Init	Valid caller: Non-task | Task

[Outline]

This is a function that performs storage device initialization processing.

[C language format]

```
void storageDev_Init (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function initializes the storage device. Since the virtual device that just secures the storage area in the memory is used in the sample program, the program simply clears the secured memory area to 0.

[Return value]

None

scsi_command_to_ata	Valid caller: Non-task I Task

[Outline]

This is a function that performs SCSI command processing.

[C language format]

```
long scsi_command_to_ata
( BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag )
```

[Parameter]

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

[Operation]

This function calls the command processing function from the SCSI command reported by CBW.

[Return value]

Status upon command processing

DEV_ERR_NODATA:	Error in transfer direction for NO DATA command
DEV_ERR_READ:	Error in transfer direction for READ command
DEV_ERR_WRITE:	Error in transfer direction for WRITE command
DEV_ERROR:	Execution of each command is other than the above three statuses, or request is illegal
DEV_OK:	Normal termination

ata_test_unit_ready	Valid caller: Non-task I Task

[Outline]

This is a function that performs TEST UNIT READY command processing.

[C language format]

```
long ata_test_unit_ready (long TransFlag)
```

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Operation]

This function performs TEST UNIT READY command processing. Since the virtual device is used in the sample program, the program performs no processing, just returns OK, and terminates normally.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command

DEV_OK: Normal termination

ata_seek	Valid caller: Non-task I Task

[Outline]

This is a function that performs SEEK command processing.

[C language format]

```
long ata_seek (long TransFlag)
```

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Operation]

This function performs SEEK command processing. Since the virtual device is used in the sample program, the program performs no processing, just returns OK, and terminates normally.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command

DEV_OK: Normal termination

ata_start_stop_unit	Valid caller: Non-task I Task

[Outline]

This is a function that performs START STOP UNIT command processing.

[C language format]

```
long ata_start_stop_unit (long TransFlag)
```

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Operation]

This function performs START STOP UNIT command processing. Since the virtual device is used in the sample program, the program performs no processing, just returns OK, and terminates normally.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command

DEV_OK: Normal termination

ata_synchronize_cache	Valid caller: Non-task I Task

[Outline]

This is a function that performs SYNCHRONIZE CACHE command processing.

[C language format]

```
long ata_synchronize_cache (long TransFlag)
```

[Parameter]

I/O	Parameter	Description
I	long TransFlag	Data transfer direction

[Operation]

This function performs SYNCHRONIZE CACHE command processing. Since the virtual device is used in the sample program, the program performs no processing, just returns OK, and terminates normally.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command

DEV_OK: Normal termination

ata_request_sense Valid caller: Non-task | Task

[Outline]

This is a function that performs REQUEST SENSE command processing.

[C language format]

```
long ata_request_sense
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs REQUEST SENSE command processing. Since the virtual device is used in the sample program, the program performs no processing, just returns OK, and terminates normally if the transmit data size specified by the command is 0. If the transmit data size specified by the command is not 0, that size of REQUEST SENSE data is prepared and transmitted. However, the amount exceeding the specified size is not transmitted.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command

DEV_ERR_READ: Error in transfer direction for READ command

DEV_OK: Normal termination

ata_inquiry	Valid caller: Non-task I Task

[Outline]

This is a function that performs INQUIRY command processing.

[C language format]

```
long ata_inquiry (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs INQUIRY command processing. In the sample program, this function prepares and transmits INQUIRY data whose size is specified by the command. If an attempt is made to transmit the data whose size exceeds the specified INQUIRY data size, the amount exceeding the specified size is not transmitted.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ: Error in transfer direction for READ command
DEV_ERROR: Request is illegal, or scsi_to_usb execution is terminated abnormally
DEV_OK: Normal termination

ata_mode_select	Valid caller: Non-task I Task

[Outline]

This is a function that performs MODE SELECT (6) command processing.

[C language format]

```
long ata_mode_select
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs MODE SELECT (6) command processing. In the sample program, this function loads the specified amount of data to the MODE SELECT table. If the size of the loaded data exceeds the data length of the MODE SELECT table, the amount exceeding the specified size is not loaded.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_WRITE: Error in transfer direction for WRITE command
DEV_ERROR: CDB contents are illegal
DEV_OK: Normal termination

ata_mode_select10	Valid caller: Non-task I Task

[Outline]

This is a function that performs MODE SELECT (10) command processing.

[C language format]

```
long ata_mode_select10
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs MODE SELECT (10) command processing. In the sample program, this function loads the specified amount of data to the MODE SELECT (10) table. If the size of the loaded data exceeds the data length of the MODE SELECT (10) table, the amount exceeding the specified size is not loaded.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_WRITE:	Error in transfer direction for WRITE command
DEV_ERROR:	CDB contents are illegal
DEV_OK:	Normal termination

ata_mode_sense	Valid caller: Non-task I Task

[Outline]

This is a function that performs MODE SENSE (6) command processing.

[C language format]

```
long ata_mode_sense
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs MODE SENSE (6) command processing. In the sample program, this function prepares and transmits the specified amount of MODE SENSE data. If the size of the loaded data exceeds the specified MODE SENSE size, the amount exceeding the specified size is not loaded.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ:	Error in transfer direction for READ command
DEV_ERROR:	scsi_to_usb execution is terminated abnormally
DEV_OK:	Normal termination

ata_mode_sense10	Valid caller: Non-task I Task

[Outline]

This is a function that performs MODE SENSE (10) command processing.

[C language format]

```
long ata_mode_sense10
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs MODE SENSE (10) command processing. In the sample program, this function prepares and transmits the specified amount of MODE SENSE (10) data. If the size of the loaded data exceeds the specified MODE SENSE (10) size, the amount exceeding the specified size is not loaded.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ:	Error in transfer direction for READ command
DEV_ERROR:	scsi_to_usb execution is terminated abnormally
DEV_OK:	Normal termination

ata_read_format_capacities Valid caller: Non-task | Task

[Outline]

This is a function that performs READ FORMAT CAPACITIES command processing.

[C language format]

```
long ata_read_format_capacities
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs READ FORMAT CAPACITIES command processing. In the sample program, this function prepares and transmits the specified amount of READ FORMAT CAPACITIES data. If an attempt is made to transmit data whose size exceeds the specified READ FORMAT CAPACITIES data size, the amount exceeding the specified size is not transmitted.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ: Error in transfer direction for READ command
DEV_ERROR: scsi_to_usb execution is terminated abnormally
DEV_OK: Normal termination

ata_read_capacity	Valid caller: Non-task I Task

[Outline]

This is a function that performs READ CAPACITY command processing.

[C language format]

```
long ata_read_capacity
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs READ CAPACITY command processing. In the sample program, this function prepares and transmits the specified amount of READ CAPACITY data. If an attempt is made to transmit data whose size exceeds the specified READ CAPACITY data size, the amount exceeding the specified size is not transmitted.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ:	Error in transfer direction for READ command
DEV_ERROR:	scsi_to_usb execution is terminated abnormally
DEV_OK:	Normal termination

ata_read6 Valid caller: Non-task | Task

[Outline]

This is a function that performs READ (6) command processing.

[C language format]

```
long ata_read6 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs READ (6) command processing. This function reads the specified amount of data starting from the specified address in the storage device. In the sample program, data in the virtual device is read.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ: Error in transfer direction for READ command

DEV_ERROR: CDB contents are illegal, or scsi_to_usb execution is terminated abnormally

DEV_OK: Normal termination

ata_read10	Valid caller: Non-task I Task

[Outline]

This is a function that performs READ (10) command processing function.

[C language format]

```
long ata_read10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs READ (10) command processing. This function reads the specified amount of data starting from the specified address in the storage device. In the sample program, data in the virtual device is read.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_READ:	Error in transfer direction for READ command
DEV_ERROR:	CDB contents are illegal, or scsi_to_usb execution is terminated abnormally
DEV_OK:	Normal termination

ata_write6 Valid caller: Non-task | Task

[Outline]

This is a function that performs WRITE (6) command processing.

[C language format]

```
long ata_write6 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs WRITE (6) command processing. This function writes the specified amount of receive data starting from the specified address, to the storage device. In the sample program, data is written to the virtual device.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_WRITE: Error in transfer direction for WRITE command

DEV_ERROR: CDB contents are illegal

DEV_OK: Normal termination

ata_write10	Valid caller: Non-task I Task

[Outline]

This is a function that performs WRITE (10) command processing.

[C language format]

```
long ata_write10 (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs WRITE (10) command processing. This function writes the specified amount of receive data starting from the specified address, to the storage device. In the sample program, data is written to the virtual device.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_WRITE: Error in transfer direction for WRITE command
DEV_ERROR: CDB contents are illegal
DEV_OK: Normal termination

ata_verify Valid caller: Non-task | Task

[Outline]

This is a function that performs VERIFY command processing.

[C language format]

```
long ata_verify (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs VERIFY command processing. This function checks the specified amount of data starting from the specified address in the storage device. Since the virtual device is used in the sample program, the program does not check the data.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_ERR_NODATA: Error in transfer direction for NO DATA command
DEV_ERROR: CDB contents are illegal
DEV_OK: Normal termination

ata_write_verify	Valid caller: Non-task I Task

[Outline]

This is a function that performs WRITE VERIFY command processing.

[C language format]

```
long ata_write_verify
(BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs WRITE VERIFY command processing. This function writes the specified amount of data starting from the specified address to the storage device and checks whether or not the written data is correct. Since the virtual device is used in the sample program, the program simply writes data to the specified memory but does not check the written data.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_OK: Normal termination

ata_write_buff	Valid caller: Non-task I Task

[Outline]

This is a function that performs WRITE BUFF command processing.

[C language format]

```
long ata_write_buff (BYTE *ScsiCommandBuf, BYTE *pbData, long lDataSize, long TransFlag)
```

[Parameter]

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

[Operation]

This function performs WRITE BUFF command processing. This function writes data to the memory (data buffer). Since the virtual device is used in the sample program, the program performs no processing and terminates normally.

Remark Refer to **3.7.2 (4) SCSI command processing** for details of SCSI commands.

[Return value]

Status upon command processing

DEV_OK: Normal termination

scsi_to_usb	Valid caller: Non-task I Task

[Outline]

This is a function used to transfer data from the virtual device to the USB device.

[C language format]

```
long scsi_to_usb (BYTE *pbData, long TransFlag)
```

[Parameter]

I/O	Parameter	Description
I	BYTE *pbData	Start address of transmit data
I	long TransFlag	Data transfer direction

[Operation]

This function performs data transmission from the virtual device to the USB device.

[Return value]

Status upon command processing

DEV_ERR_READ: Error in transfer direction for READ command

DEV_OK: Normal termination

USBF850REG_SET Valid caller: Non-task | Task

[Outline]

This is a function that sets the V850E/ME2 peripheral I/O registers (1-byte units: 8 bits).

[C language format]

```
USBF850REG_SET (offset, val)
```

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Data for setting

[Operation]

This function sets data specified by *val* to the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-byte (8-bit) units.

[Return value]

None

USBF850REG_READ	Valid caller: Non-task I Task

[Outline]

This is a function that reads the V850E/ME2 peripheral I/O registers (1-byte units: 8 bits).

[C language format]

```
USBF850REG_READ (offset)
```

[Parameter]

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

[Operation]

This function reads the value in the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-byte (8-bit) units.

[Return value]

None

USBF850REG_SET_W Valid caller: Non-task | Task

[Outline]

This is a function that sets the V850E/ME2 peripheral I/O registers (1-word units: 16 bits).

[C language format]

```
USBF850REG_SET_W (offset, val)
```

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Data for setting

[Operation]

This function sets data specified by *val* to the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-word (16-bit) units.

[Return value]

None

USBF850REG_READ_W	Valid caller: Non-task I Task

[Outline]

This is a function that reads the V850E/ME2 peripheral I/O registers (1-word units: 16 bits).

[C language format]

```
USBF850REG_READ_W (offset)
```

[Parameter]

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

[Operation]

This function reads the value in the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-word (16-bit) units.

[Return value]

None

CHAPTER 4 USB COMMUNICATION CLASS DRIVER

4.1 General

4.1.1 Overview

The USB communication class driver is a sample program for the USB function controller that is incorporated in the V850E/ME2. It conforms to Universal Serial Bus Specification Revision 1.1 and operates on the embedded real-time control operating system RX850 Pro (conforms to the μITRON 3.0 specifications).

This sample program uses the control endpoint (endpoint 0), IN and OUT of the bulk endpoint (endpoints 3 and 4), and IN of the interrupt endpoint (endpoint 7), and is connected to the Windows XP standard communication class host driver to operate as a virtual COM port. The communication class is defined as the class.

This sample program uses the emulation board SolutionGear MINI (SG-703111-1) as the hardware execution environment. When using the SolutionGear MINI and sample program as is, create the execution object by following the procedure described in **4.6 Load Module** and confirm its operation by following the procedure described in **4.2 Execution of Load Module**.

When using another target board instead of SolutionGear MINI, change the board referring to **4.3 System Configuration**, **4.4 RX850 Pro-Dependent Processing Modules**, and **4.5 Section Map File**, in accordance with the board specifications.

When changing both SolutionGear MINI and sample program, change them referring to **4.3 System Configuration**, **4.4 RX850 Pro-Dependent Processing Modules**, **4.5 Section Map File**, **4.6 Load Module**, and **4.7 USB Driver Functions**.

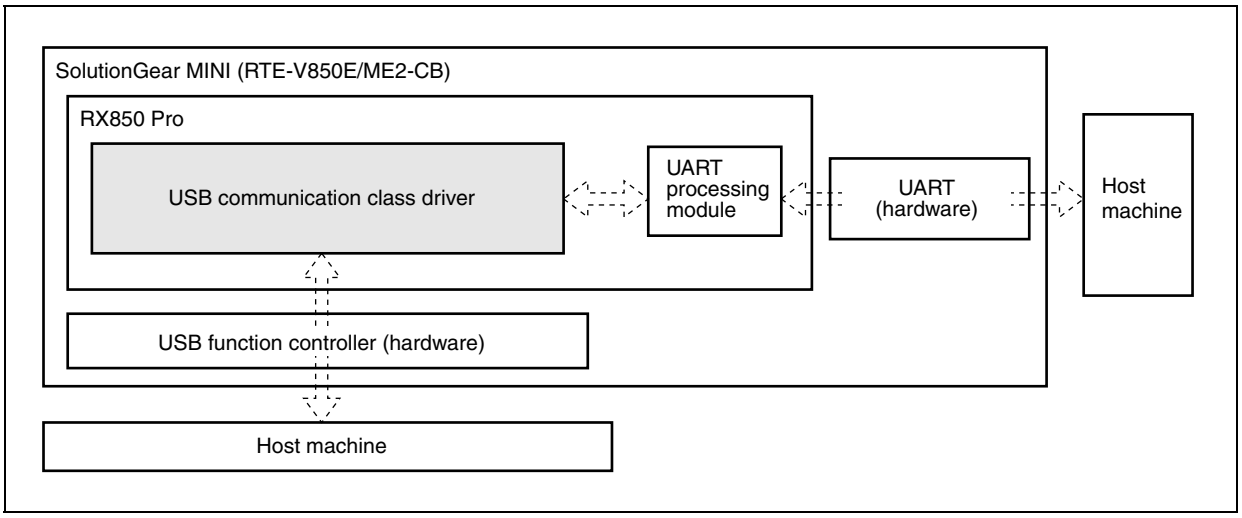
The positioning of the USB communication class driver is shown below.

Caution Since the Windows XP communication class host driver is not supported officially, users have to create an *inf* file for calling the driver module. For the *inf* file, refer to the description under Device Classes in USB Developers FAQ on http://www.lvr.com/usbfaq.htm.

Remarks 1. Refer to Universal Serial Bus Class Definitions for Communication Devices Version 1.1 for details of the USB communication class.

2. The descriptions in **4.2.1 Execution procedure of load module** assume the user environment described in **4.1.3 Execution environment**.

Figure 4-1. Positioning of USB Communication Class Driver



4.1.2 Development environment

This section assumes the following hardware and software environments are used for system development using the sample program.

- Hardware environment
  Host machine: PC/AT-compatible machines (OS: Windows XP)
- Software environment
  Real-time OS: RX850 Pro Version 3.15
  USB communication class driver: Sample program set described in this section
  C compiler package: MULTI2000
  (CCV850 Version 3.5 (made by Green Hills Software, Inc.))

Caution If the directory configuration of the user environment differs from that handled in the build file of the sample program, adjust the build file to the user environment.

Remark Refer to the help of MULTI (made by Green Hills Software, Inc.) for the description of the build file.

4.1.3 Execution environment

This section assumes the following hardware and software environments are used for load module execution using the sample program.

- Hardware environment

  Host machine: PC/AT-compatible machines (OS: Windows XP)
  IE control machine: PC/AT-compatible machines (OS: Windows XP)
  Target board: SolutionGear MINI (SG-703111-1)
  In-circuit emulator (IE): N-wire IE (RTE-2000-TP) (made by Midas Lab Inc.)
  JTAG probe
  USB cable
  RS-232C cable: RS-232C cable included with SolutionGear MINI
- Software environment

  Software for IE: PARTNER Setup Program Version 1.242

Remarks 1. Refer to **APPENDIX A SG-703111-1 BOARD** and the **SG-703111-1 User's Manual** for details of how to set up the execution environment.

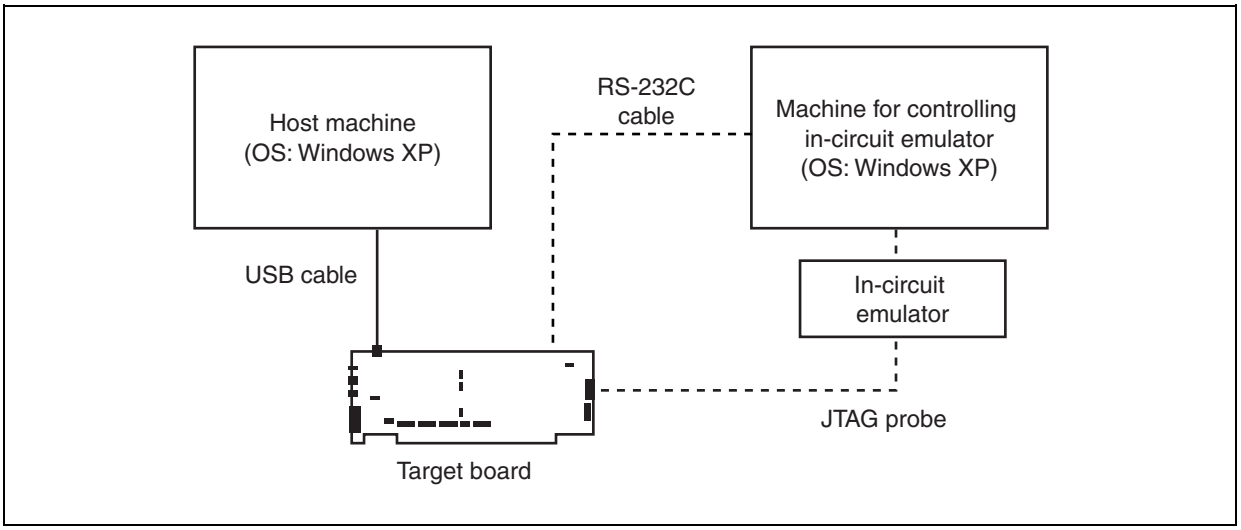
2. Refer to the **RTE-2000-TP Hardware User's Manual** for details of how to set up the in-circuit emulator (RTE-2000-TP).

3. Refer to the **PARTNER User's Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for details of PARTNER.

4. The RS-232C cable is used to connect the serial connector JSIO2 on the target board and the host machine communicating with UART, using the conversion connector. Refer to the **SG-703111-1 User's Manual** for details of the relevant connector.

5. In Figure 4-2, the IE control machine is used as the host machine communicating with UART.

Figure 4-2. Execution Environment



4.2 Execution of Load Module

4.2.1 Execution procedure of load module

The following shows the procedure for executing the load module under the environment described in **4.1.3 Execution environment**, taking the load module using the sample program as an example.

(1) Preparation of machine for controlling in-circuit emulator (IE)

Turn on the power and start up the IE control machine and the in-circuit emulator.

(2) Preparation of host machine

Turn on the power and start up the host machine (the IE control machine can be used as the host machine, but it is strongly recommended to provide an independent machine for development).

(3) Reset SG-703111-1 board

Press the RESET button of the SG-703111-1 board to reset the SG-703111-1 board.

(4) Startup of software for IE

Start up software for IE.

Select the [Start] button → “All Programs” → “PARTNER” → “RPTSETUP (NB85ET)” in Windows.

Click the [Open] button and specify a project file; the [Run] button is then selectable. Click the [Run] button to start up PARTNER. Make the board settings after startup. It is useful to create at this time the setting file loaded at startup. Refer to **APPENDIX A SG-703111-1 BOARD**, **PARTNER User’s Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for setup files for the sample described in this section.

Cautions 1. Be sure to apply power to the target board before starting up the in-circuit emulator.

2. If you want to load the setting file for resetting the target board after the in-circuit emulator is started up, load the setting file (init.mcr in the example below) by inputting a command to the command window, as shown below.

[Command input example]

```
><init.mcr<Enter>
```

(5) Loading the load module

Load the load module to the board using the in-circuit emulator function.

The load module (usb_communication.out in the example below) can be loaded by selecting [Load] in the [File] menu on the toolbar, or input the L command (loading file) in the command window.

[Command input example]

```
>l usb_communication.out<Enter>
```

(6) Execution

The code loaded to the board is executed by pressing the F5 key or the [Go] button.

Remark The same operation is performed by selecting [Go] in the [Run] menu on the toolbar.

(7) USB connection

Connect the USB cable.

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

Cautions 1. The USB cable can be connected before/after starting up the target board.

2. When the device is detected by the host machine, the software installation screen appears. Install the Windows XP standard USB communication class host driver in this sample program. Since the Windows XP communication class host driver is not supported officially, users have to create an *inf* file for calling the driver module. For the *inf* file, refer to the description under Device Classes in USB Developers FAQ on http://www.lvr.com/usbfaq.htm.

(8) Startup of Device Manager

Open the Properties window from My computer and select the Hardware tab. Select the Device Manager to start up the Device Manager.

Remark The Device Manager can also be started up from [Manage] menu of My computer or the Control Panel.

(9) Confirmation of USB device connection

Make sure that “COM-USB *xxx* (COM3)” is displayed under Port (COM and LPT) in the Device Manager screen. (“xxx” is the connection name set by the user.)

Caution An item other than COM3 (such as COM4) may be displayed depending on the usage state of the COM port.

(10) Connection of RS-232C cable

Connect UARTB0 that is incorporated in the V850E/ME2 and the IE control machine using the RS-232C cable.

Cautions 1. The RS-232C cable can be connected before/after starting up the target board.

2. The RS-232C cable is used to connect the serial connector JSIO2 on the target board and the IE control machine. Refer to the SG-703111-1 User’s Manual for details of the relevant connector, using the conversion connector.

(11) Confirmation of operation

Start up the terminal software, such as Hyper Terminal, in the host machine and IE control machine. The following description uses an example that UARTB0 incorporated in the V850E/ME2 is connected to COM1.

How to set the host machine is described below. Read UARTB0 as COM1 for setting. Set the same values to the host machine and IE control machine for the transfer rate, etc.; otherwise the system does not operate normally.

Select ”All Programs” → “Accessories” → “Communications” → “Hyper Terminal”.

When the Hyper Terminal is started up, input the name for the new connection.

The [Setting Connection] screen is then displayed. Select COM3 as the connection method on the USB side, and COM1 on the UARTB0 side.

When the connection method is determined, the [Setting Port] screen is displayed. Select the transfer rate, data length, parity, and stop bit.

When setting is complete, data transmission /reception can be checked.
To change the setting value of UART transfer rate, etc., disconnect from the line, select [Modem Configuration] in the Properties menu of Hyper Terminal, and then change the setting on the setup screen. At this time, change the setting in the same manner in the communicating device.

(12) Exiting program

Terminate the program under execution.
Click the forcible break button on the PARTNER screen, or select “Forcible Break” in the [Run] menu on the toolbar to stop program execution.

(13) Shutting down in-circuit emulator

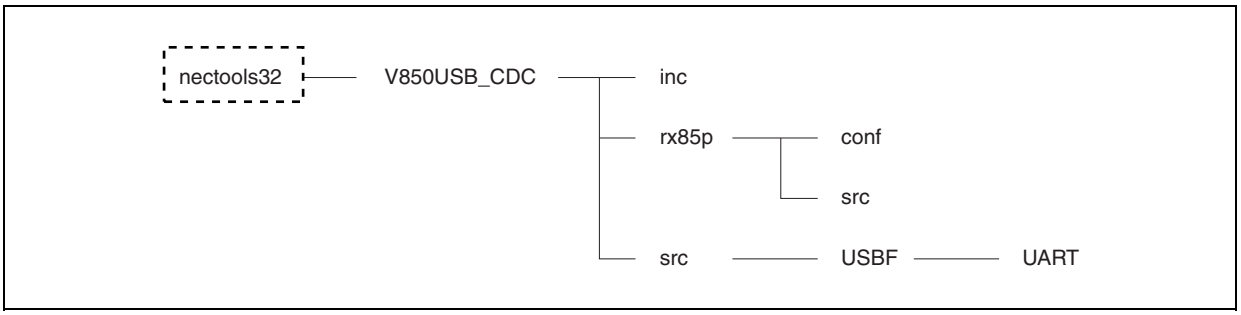
Shut down the in-circuit emulator and reset the target board by following the procedure described in **(1)**.
Select [Exit] in the [File] menu on the toolbar to terminate PARTNER.
After terminating PARTNER, reset the target board by following the procedure described in **(1)**.

4.2.2 Directory configuration

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

Caution It is recommended to place the directory of the USB communication class driver files directly under the directory where the RX850 Pro is installed (\nectools32).

Figure 4-3. Sample Program Directory Configuration



The outline of each directory is shown below.

(1) nectools32

A directory created when the RX850 Pro is installed. Place the directory (directory name: V850USB_CDC) of the driver directly under this directory.

(2) nectools32\V850USB_CDC

A directory for the USB communication class driver.

- usb_communication.bld: Build file of USB communication class driver
- common.lx: Section map file

(3) nectools32\V850USB_CDC\inc

A directory in which header files for the USB communication class driver are stored.

- errno.h: Header file for return value
- types.h: Header file for data type
- sys.h: Header file for system information

Caution sys.h (header file for system information) is usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create this file because the command is automatically executed when build is executed.

(4) nectools32\V850USB_CDC\rx85p

A directory in which files for the RX850 Pro are stored.

(5) nectools32\V850USB_CDC\rx85p\conf

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

- sit.850: System information table
- svc.850: System call table
- sysi.tbl: System information table
- sysc.tbl: System call table

Cautions 1. Files in this directory are usually created by command input using the configurator when build is executed. If a build file in the sample program is used, however, users are not required to create these files because the command is automatically executed when build is executed.

2. sit.850 and sysi.tbl, svc.850 and sysc.tbl differ only in their file extension.

(6) nectools32\V850USB_CDC\rx85p\src

A directory in which files for RX850 Pro are stored.

- boot.850: Assembler file for boot processing
- entry.850: Assembler file for entry processing
- init.c: Source file for hardware initialization module
- init.h: Header file for hardware initialization module
- sys.cf: CF definition file
- varfunc.c: Source file for software initialization module

(7) nectools32\V850USB_CDC\src

A directory in which files of the USB communication class driver board-dependent module are stored.

- port.c: Source file for port setting
- port.h: Header file for port setting

(8) nectools32\V850USB_CDC\src\USBF

A directory in which files of the USB communication class driver USB processing module are stored.

- usbf850.c: Source file for USB device
- usbf850.h: Header file for USB device
- usbf850desc.h: USB descriptor definition file
- usbf850_communication.c: Source file for USB-UART interface
- usbf850_communication.h: Header file for USB-UART interface

(9) nectools32\V850USB_CDC\src\USBF\UART

A directory in which files of the USB communication class driver UART processing module are stored.

- uart_ctrl.c: Source file for UART processing module
- uart_ctrl.h: Header file for UART processing module

Caution Refer to 4.8 UART Processing Module for details of the UART processing module. The UART processing module contained in the sample program only supports the minimum functions required in this sample program. Therefore, the operation of this UART processing module is not guaranteed when used as a general UART driver.

4.3 System Configuration

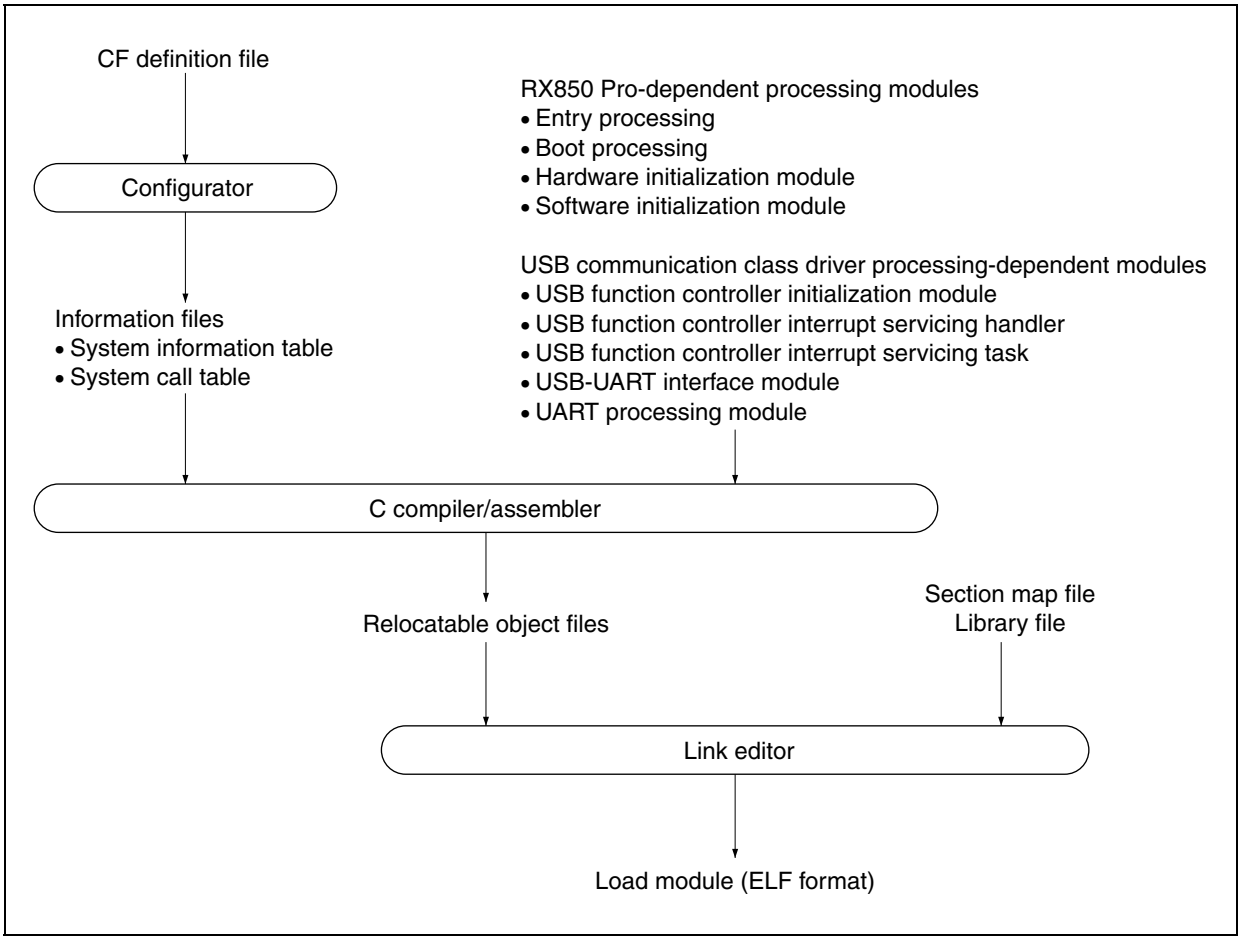
4.3.1 Overview

System configuration means creation of the load module using files that are installed in the user's development environment (the host machine) from the USB communication class driver supply medium.

The system configuration procedure of USB communication class drivers is shown below.

(1) Describing RX850 Pro-dependent processing module
(2) Describing board-dependent module
(3) Describing USB communication class driver processing-dependent module
(4) Describing section map file
(5) Creating load module

Figure 4-4. System Configuration Procedure



4.3.2 Describing RX850 Pro-dependent processing module

Some functions provided by the USB communication class driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
  - Boot processing
  - Hardware initialization module
  - Software initialization module

Remark Refer to **4.4 RX850 Pro-Dependent Processing Modules** for details of the RX850 Pro-dependent processing module.

4.3.3 Describing board-dependent module

The initialization processing, which is related to the processing dependent on the user's execution environment and application system, is provided as a board-dependent module in the USB communication class driver source program.

The board-dependent module is as follows.

- CPU board-dependent module
  The port input/output manipulation required for the USB communication class driver is provided as a CPU board-dependent module.

Caution Since port setting is handled in the same manner as setting of other registers, no dedicated function is provided.
Refer to the RX850 Pro standard header file *SFR.h* stored in \nectools32\inc850\common\ for the register definition. For detailed processing, refer to the source program for port setting (port.c) called from the boot processing module (boot.850) and software initialization module.

4.3.4 Describing USB communication class driver processing-dependent module

The driver functions, which are used to implement the USB communication class driver functions, are provided as the USB communication class driver processing-dependent module in this sample program.

The USB communication class driver processing-dependent modules are listed below.

- USB function controller initialization module
- USB function controller interrupt handlers
- USB function controller interrupt servicing tasks
- USB function controller general-purpose functions
- USB-UART interface module
- UART processing module

Remark Refer to **4.7 USB Communication Class Driver Functions** for details of the USB communication class driver processing-dependent module, and refer to **4.8 UART Processing Module** for details of the UART processing module.

4.3.5 Describing section map file

The section map file is used by the user to fix address assignment performed by the link editor.

The following five text areas are essential sections when using the RX850 Pro.

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

Remark Refer to **4.5 Section Map File** for details of the section map file.

4.3.6 Creating load module

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing modules, USB communication class driver processing-dependent module, and section map file, that have been coded.

Remark Refer to **4.6 Load Module** for details of how to create the load module.

4.4 RX850 Pro-Dependent Processing Modules

4.4.1 Overview

Some functions provided by the USB communication class driver use the functions of the real-time OS (RX850 Pro), and the processing modules described by the user are executed under RX850 Pro control.

Therefore, it is necessary to describe the RX850 Pro-dependent processing modules for normal RX850 Pro operation.

The RX850 Pro-dependent processing modules are listed below.

○ CF definition file
○ Entry processing
○ System initialization processing
  - Boot processing
  - Hardware initialization module
  - Software initialization module

4.4.2 CF definition file

An information file (CF definition file) that contains data provided to the RX850 Pro is required to configure the system in which the RX850 Pro is used.

The following information is required for using the USB communication class driver function.

○ Real-time OS information
  - RX Series information
○ SIT information
  - System information
  - System maximum value information
  - System memory information
  - Task information
  - Interrupt handler information
  - Initialization handler information
○ SCT information
  - Task management/task-associated synchronization system call information
  - Interrupt servicing management system call information
  - Time management system call information

Caution This sample program implements each functions using four tasks, six interrupt handlers, and seven system calls. Therefore, the CF definition file, the maximum number of tasks to be created must be set to four as the system's maximum value information and the maximum number of interrupt handlers to be created must be set to six for the USB communication class driver and use of *sta_tsk*, *ext_tsk*, *slp_tsk*, and *wup_tsk* system calls must be defined as task management/task-associated synchronization system call information, use of the *loc_cpu* and *unl_cpu* system calls as interrupt servicing management system call information, and use of the *dly_tsk* system call as time management function system call information. Of the six interrupt handlers, three are used by the UART interrupt.

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

(1) Procedure for creating information files

A procedure for creating information files (system information table, system call table, and system information header file) is shown below.

The information file can be created from the Windows command prompt.

Caution If a build file in the sample program is used, users are not required to create information files in this procedure because they are automatically executed when build is executed.

<1> Change current directory

Move the current directory to the directory in which the CF definition file is stored using the cd command of Windows.

A command 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> Creating information files

Create the information file from the CF definition file that has been created in the specific description format, using the configurator cf850pro.exe.

A command input example when creating three information files (system information table: sit.850, system call table: svc.850, and system information header file: sys.h) from an input file (CF definition file name: sys.cf) is shown below.

[Command input example]

```
C:>cf850pro -i sit.850 -c svc.850 -d sys.h sys.cf<Enter>
```

The information files are created from the CF definition file.

Caution A sample file (CF definition file) used for creating the information files is provided in the sample program.

Remark Refer to the **RX850 Pro Installation User's Manual** for details of the option to activate the configurator cf850pro.exe and execution method.

4.4.3 Entry processing

This processing assigns a branch instruction to an interrupt handler to the handler address where control is forcibly passed by the processor when a maskable interrupt occurs.

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

Remark Refer to sample program *entry.850* for details of how to code the entry processing.

4.4.4 System initialization processing

The system initialization processing includes initialization processing (boot processing and hardware initialization module) of hardware required for operating the RX850 Pro normally, and software initialization processing (nucleus initialization module and Initialization handler).

The system initialization processing is performed first when the system is activated.

Caution Among the four types of system initialization processing, users are not required to describe the nucleus initialization module because it is a function provided by the RX850 Pro.

The processing performed by the nucleus initialization module is shown below.

○ Securement of system memory defined by CF definition file
- System pool 0
- User pool 0

○ Generation and activation of management object defined by CF definition file
- Generation and activation of task
- Registration of interrupt handler

○ Activation of initial task
○ Generation and activation of idle task
○ Calling software initialization module
○ Passing control to scheduler

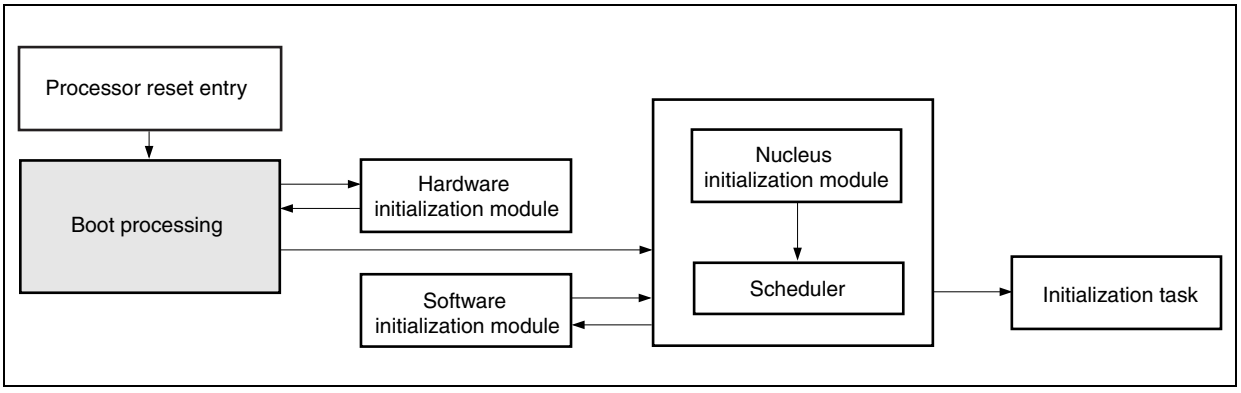
The idle task is a processing routine that is activated by the scheduler when a processing module (task) executed by the RX850 Pro is no longer in the run or ready state, that is, no processing module targeted to the scheduling by the RX850 Pro exist in the system. The idle task issues the HALT instruction.

(1) Boot processing

Boot processing is the function assigned to the processor reset entry, so it is executed first in the system initialization processing.

The positioning of boot processing is shown below.

Figure 4-5. Positioning of Boot Processing



The processing executed by boot processing is shown below.

Remark Refer to sample program *boot.850* for details of how to code boot processing.

- Setting tp, gp, and ep registers
  Values of the text pointer tp, global pointer gp, and stack pointer ep, which are required for execution of each processing module (including boot processing), are undefined when a system is activated. Boot processing first performs initial setting of these registers.

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

- Calling hardware initialization module
  Functions (hardware initialization module) are called to initialize the hardware on the target system.
  This step is not required if initialization of internal units is performed by other module.

  Caution In this chapter, this step is not required because initialization of internal units is performed by the software initialization module. Refer to the RX850 Pro Installation User's Manual for details.

- Passing control to nucleus initialization module
  The nucleus initialization module secures the system memory (system pool 0, user pool 0) and creation/initialization of management objects, based on information described in the system information table. Therefore, start address_sit of the system information table must be set to the r10 register before passing control to the nucleus initialization module.

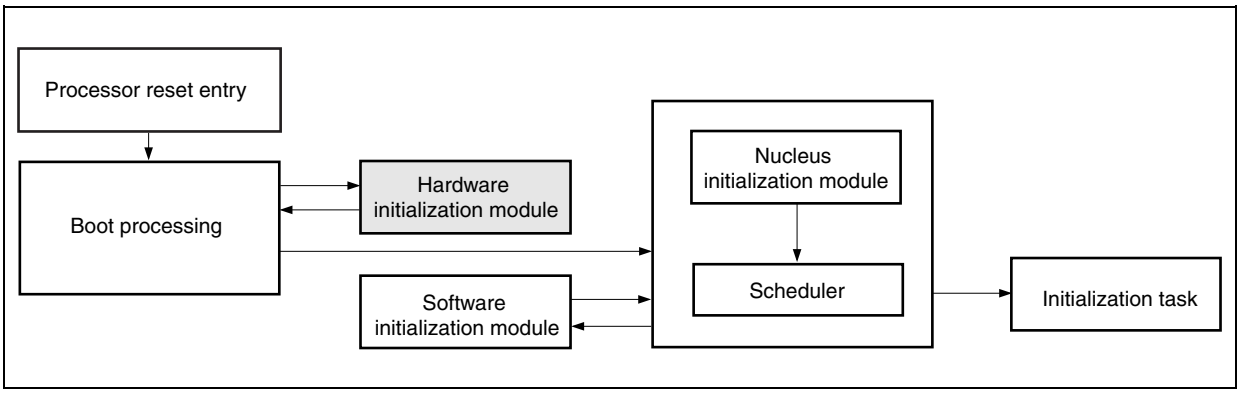
  Caution The system information table is a table in which the CF definition file created in a specific description format is converted to the assembly language format, using the utility tool (configurator cf850pro.exe) provided by the RX850 Pro.

(2) Hardware initialization module

The hardware initialization module is a function to initialize the hardware on the target system, and is called from boot processing.

The positioning of the hardware initialization module is shown below.

Figure 4-6. Positioning of Hardware Initialization Module



The processing executed by the hardware initialization module is shown below.

Cautions 1. Users are not required to disable the maskable interrupts because they are masked at initialization by default.

2. Hardware initialization is performed by the software initialization module in the sample program. Refer to the RX850 Pro Installation User's Manual for details of the hardware initialization module.

- Returning control to boot processing

  Control can be returned from the hardware initialization module to boot processing by issuing the "return( );" instruction, because the return address to the lp register is set when the hardware initialization module is called from boot processing.

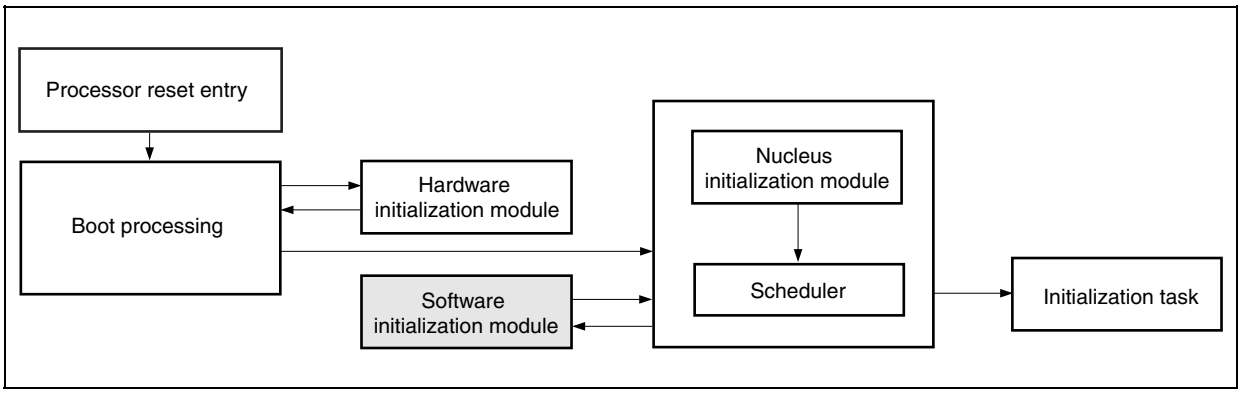
  If the hardware initialization module is described with the assembly language, this processing is implemented by issuing the "jmp [lp]" instruction.

(3) Software initialization module

The initialization handler is a function provided to enhance operability of the user software environment, and is called from the nucleus initialization module.

The positioning of the software initialization module is shown below.

Figure 4-7. Positioning of Software Initialization Module



The processing executed by the software initialization module is shown below.

Remark Refer to sample program *varfunc.c* for how to code the software initialization module.

- Initialization of internal unit (real-time pulse unit (RPU))

  The RX850 Pro implements the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) using the timer interrupt that occurs in a constant cycle. Therefore, the real-time pulse unit must be initialized before the RX850 Pro starts processing.

  The compare register CMD0 included in the real-time pulse unit must be set so that timer interrupts occur in a base clock cycle defined in system information in the CF definition file.

- Enabling timer interrupt acknowledgment

  Acknowledgment of timer interrupts is enabled. In addition, this enables the use of the timer operation functions (delay task wake-up, cyclic handler activation, timeout, etc.) provided by the RX850 Pro when processing by the nucleus initialization module ends.

- Passing control to nucleus initialization module

  Control can be returned from the initialization handler to the nucleus initialization module by issuing the “return( );” instruction, because the return address lp register is set when the initialization handler is called from the nucleus initialization module.

  If the initialization handler is described with the assembly language, this processing is implemented by issuing the “jmp [lp]” instruction.

4.4.5 Time management function

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

The RX850 Pro calls system clock processing when a clock interrupt occurs, and performs processing related to the time such as updating the system clock, task delay wake-up, and activation of the cyclic handler.

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

After the system clock is set to “0H” by system initialization processing, it is updated by system clock processing in base clock cycle units (specified at configuration).

Caution The system clock managed by the RX850 Pro is configured as 48 bits wide. Therefore, overflowed numeric values (numeric values that cannot be expressed by 48 bits) are ignored by the RX850 Pro. Refer to the RX850 Pro Basics User’s Manual for details of the time management function of the RX850 Pro.

4.5 Section Map File

4.5.1 Overview

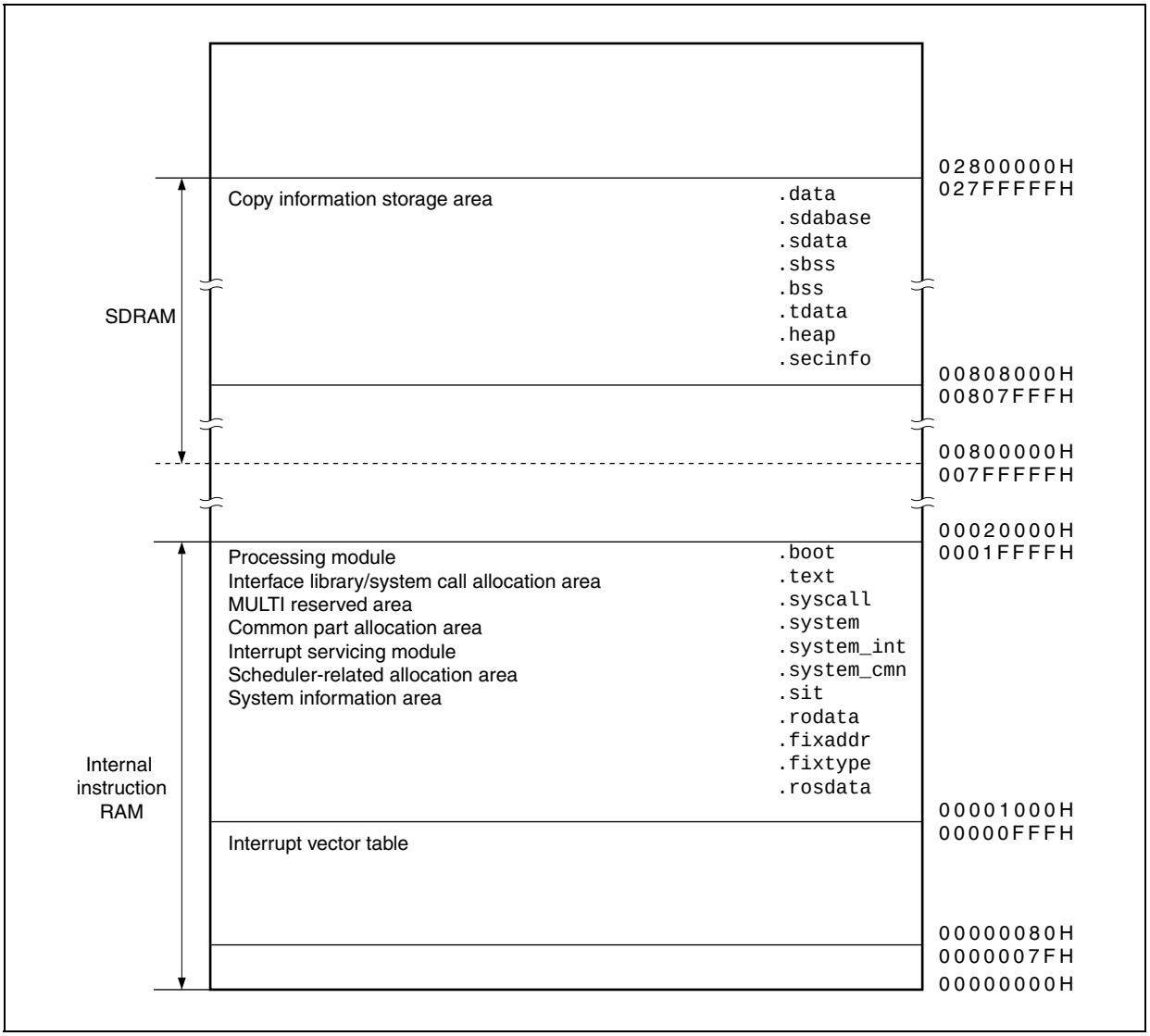
The section map file is used by the user to fix address assignment performed by the link editor.

Required assignments for addresses other than the user processing program (such as .data and .bss sections) are described in **4.5.2 Address assignment by RX850 Pro** and **4.5.3 Other address assignment**.

Address assignment performed in sample program *common.lx* is shown below.

Remark Refer to sample program *common.lx* for how to code the section map file.

Figure 4-8. Address Assignment Example



4.5.2 Address assignment by RX850 Pro

The RX850 Pro consists of five text areas: common part allocation area, interrupt servicing-related allocation area, scheduler-related allocation area, system information area, and interface library/system call allocation area. Using these areas, memory areas for which a large space is required can be assigned to the external RAM, and memory areas for which a high-speed access is required (interrupt servicing module, scheduling processing module) can be assigned to the internal instruction RAM (00000000H to 0001FFFFH).

Caution All five text areas are allocated to the internal instruction RAM in the sample program.

- Common part allocation area (.system section)
  Processing of the RX850 Pro (such as task management function, task-associated synchronization function) is assigned to this area.

- Interrupt servicing-related allocation area (.system_int section)
  Among the interrupt servicing management functions provided by the RX850 Pro, interrupt preprocessing that is performed when control is passed to the interrupt handler and interrupt postprocessing that is performed when control is handed back to the processing module in which a maskable interrupt occurs are assigned to this area.
  By assigning the interrupt servicing module to the internal instruction RAM, therefore, response performance to the interrupt handler can be improved.

  Caution It is recommended to assign the interrupt servicing module to the internal instruction RAM.

- Scheduler-related allocation area (.system_cmn section)
  Among the scheduling function provided by the RX850 Pro, task wake-up processing and task scheduling processing are assigned to this area.
  By assigning the scheduling processing section to the internal instruction RAM, therefore, task wake-up processing and task scheduling processing are accelerated, as well as system call processing involving scheduling processing.

  Caution It is recommended to assign the scheduling module to the internal instruction RAM.

- System information area (.sit section)
  The system information table created by executing the configurator cf850.exe on the CF definition file is assigned to this area.
  The system information table includes various data required for executing the nucleus initialization module (securement of the system memory and creation/initialization of management objects).

- Interface library/system call allocation area (.text section)
  The instructions including system calls are assigned to this area.

- System memory
  Various management block required for implementing functions provided by the RX850 Pro (such as the task management block, semaphore management block), area in which the stack used by the interrupt handler or task is assigned (system pool 0), and area in which dynamic memory manipulation (such as acquisition/release of memory blocks) from the processing module is enabled (user pool 0), are assigned to this area.

Cautions 1. The ”system memory start address” must be specified when creating the CF definition file. Be sure to specify the address when defining the system memory in the section map file.
2. The user can specify any section name in the system memory.

4.5.3 Other address assignment

The other sections for which address assignment is required are described below.

- MULTI reserved area (.syscall section)
  This area is used as a work area by the debugger MULTI (made by Green Hills Software, Inc.).

Cautions 1. The .syscall section must be defined regardless of whether or not MULTI is used.
2. Be sure to specify 4-byte alignment when defining the .syscall section.

- Copy information storage area (.secinfo section)
  This area is used by the link editor to output information (start address, size) required for transferring program (data, text) of a section for which the ROM identifier is specified in the section map file from ROM to RAM.
  Specification of the ROM identifier is required when performing ROMization of a processing module. Therefore, definition of the .secinfo section is not required when ROMization is not performed.

Caution This section is empty in the sample program because ROM identifier specification is not performed.

4.6 Load Module

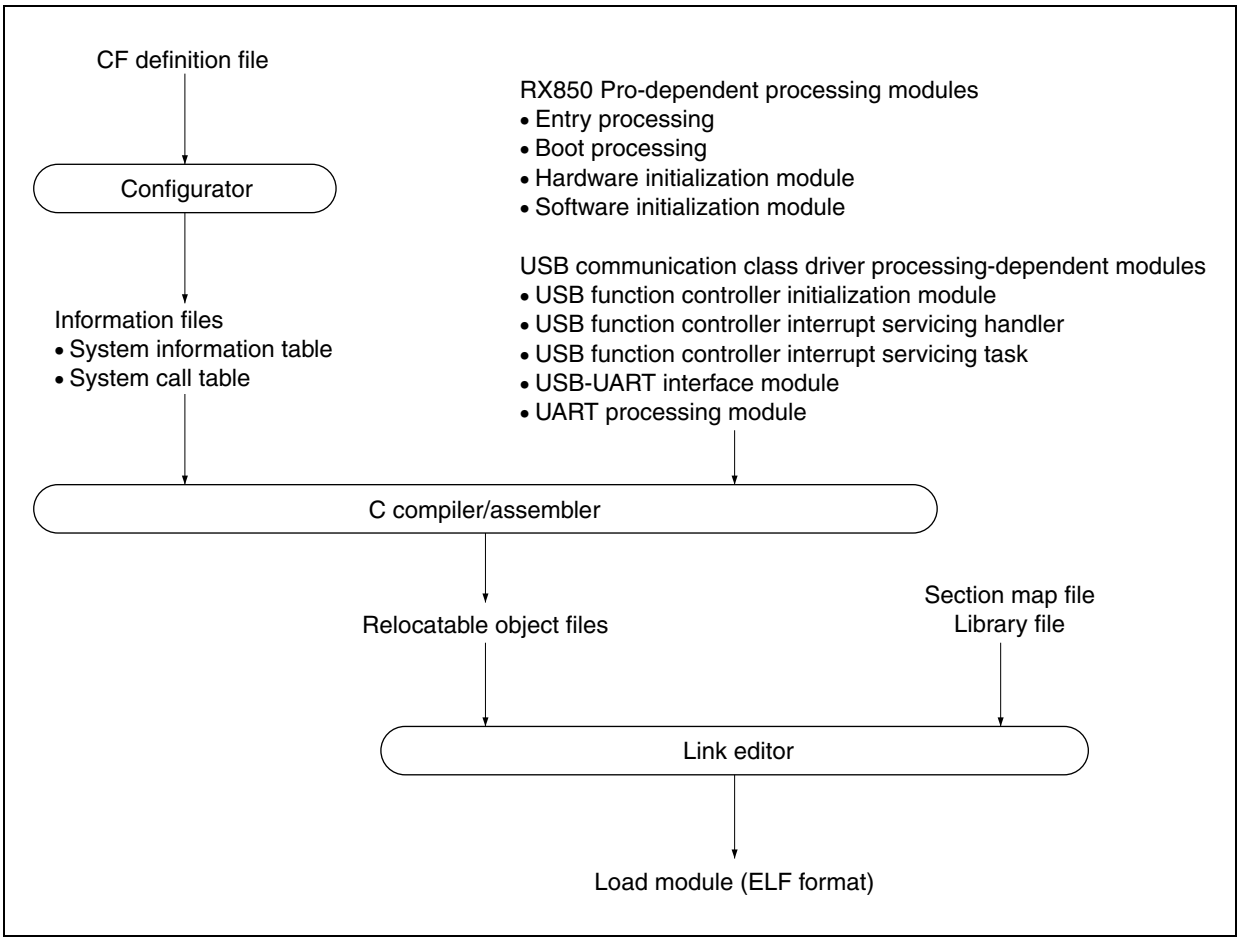
4.6.1 Overview

An ELF-format load module is created by executing the C compiler, assembler, or linker for the RX850 Pro-dependent processing module, USB communication class driver processing-dependent module, section map file, that have been coded.

The procedure for creating load modules is shown below.

Caution The load module corresponding to the sample program can be created by executing the .bld file in the sample program. However, definition of the .bld file must be adjusted to the user development environment.

Figure 4-9. Load Module Creation Procedure



4.6.2 Creating load module

An ELF-format load module can be created from the RX850 Pro-dependent processing module, USB communication class driver processing-dependent module, and section map file, that have been coded, using the following procedure.

(1) Creation of system information table and system call table

Original CF definition file formats are excluded from the link processing performed by the link editor when creating a load module.

Therefore, a file that can be assembled (system information table or system call table) must be created using the utility tool (configurator cf850.exe) provided by the RX850 Pro.

Remark Refer to **4.4.2 (1) Procedure for creating information file** for how to create the system information table and system call table.

(2) Creation of object file

A relocatable object file is created by executing the C compiler/assembler for the processing module (file described in the C language/assembly language) shown below.

○ RX850 Pro-dependent processing module
- System information table
- System call table
- Entry processing
- Boot processing
- Hardware initialization module
- Initialization handler

○ USB communication class driver processing-dependent module

○ UART processing module

(3) Creation of load module

An ELF-format load module is created by executing the link editor for relocatable object file created in **(2)**, library files, and section map file.

libansi.a	ANSI C library
libind.a	C library made by Green Hills Software, Inc. (routines independent of target CPU)
libarch.a	C library made by Green Hills Software, Inc. (routines dependent of target CPU)
libsys.a	C library made by Green Hills Software, Inc. (system call, initialization routines)

rxcore.o	Nucleus common part object
librxp.a	Nucleus library
libchp.a	Interface library

rxcore.o, *librxp.a*, and *libchp.a* are provided by the RX850 Pro, and *libansi.a*, *libind.a*, *libarch.a*, and *libsys.a* are provided by the CCV850 (made by Green Hills Software, Inc.).

4.7 USB Communication Class Driver Functions

4.7.1 Overview

Initialization processing performed by the USB function controller, as well as tasks and interrupt handlers to implement USB communication class driver processing, must be described in the USB communication class driver.

A list of USB communication class driver processing-dependent modules is shown below.

- USB function controller initialization processing
  This module is called from the RX850 Pro software initialization module and initializes the USB function controller.

- USB function controller interrupt handlers
  This is an interrupt servicing-dedicated routine that is called each time an interrupt by the USB function controller occurs, and is defined in the CF definition file.

  Caution Interrupts other than required are masked in this sample program.
  The following three interrupts are used in this sample program.

  - **SETRQ interrupt reported by INTUSB0B signal**
    (Receives a SET_XXXX request to be handled automatically and indicates it is automatically handled)
  - **CPUDEC interrupt reported by INTUSB0B signal**
    (Indicates that there is a request that is decoded by FW in the UF0E0ST register)
  - **BKO2DT interrupt reported by INTUSB1B signal**
    (Indicates that data has been received normally by the UF0BO2 register)

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

- USB function controller general-purpose function
  This is a general-purpose function used by the USB communication class driver to perform the STALL response setting for each endpoint and transmission/reception processing.

Remark Refer to sample program *usbf850.c* for how to code the USB communication class driver processing-dependent module.

- USB-UART interface module
  This module performs processing of USB communication class requests specific to device class, and USB-UART data transmission/reception.

  Caution The following five requests specific to device class can be acknowledged in this sample program. Refer to Universal Serial Bus Class Definitions for communication Devices Version 1.1 for details of each request.

  - SEND ENCAPSULATED COMMAND request
  - GET ENCAPSULATED RESPONSE request
  - SET LINE CODING request
  - GET LINE CODING request
  - SET CONTROL LINE STATE request

  Remark Refer to sample program *usbf850_communication.c* for how to code the USB-UART interface module.

- USB suspend/resume processing
  Since the USB suspend/resume processing depends on the system, it is not supported in this sample program. If this processing is necessary in your system, add the processing making allowances for the following points.
  The suspend/resume state is reported to the USB function controller incorporated in the V850E/ME2 by an interrupt (INTUSB0B signal). Therefore, whether the current status is suspend or resume can be judged by checking the UF0IS0. RSUSPD bit in the interrupt handler (for the INTUSB0B signal); if this bit is 1, the UF0EPS1.RSUM bit is checked to judge the status.
  Processing can be added by adding the above code to judge the status to the interrupt handler (for the INTUSB0B signal) and wakes up a task to perform necessary processing from the code.

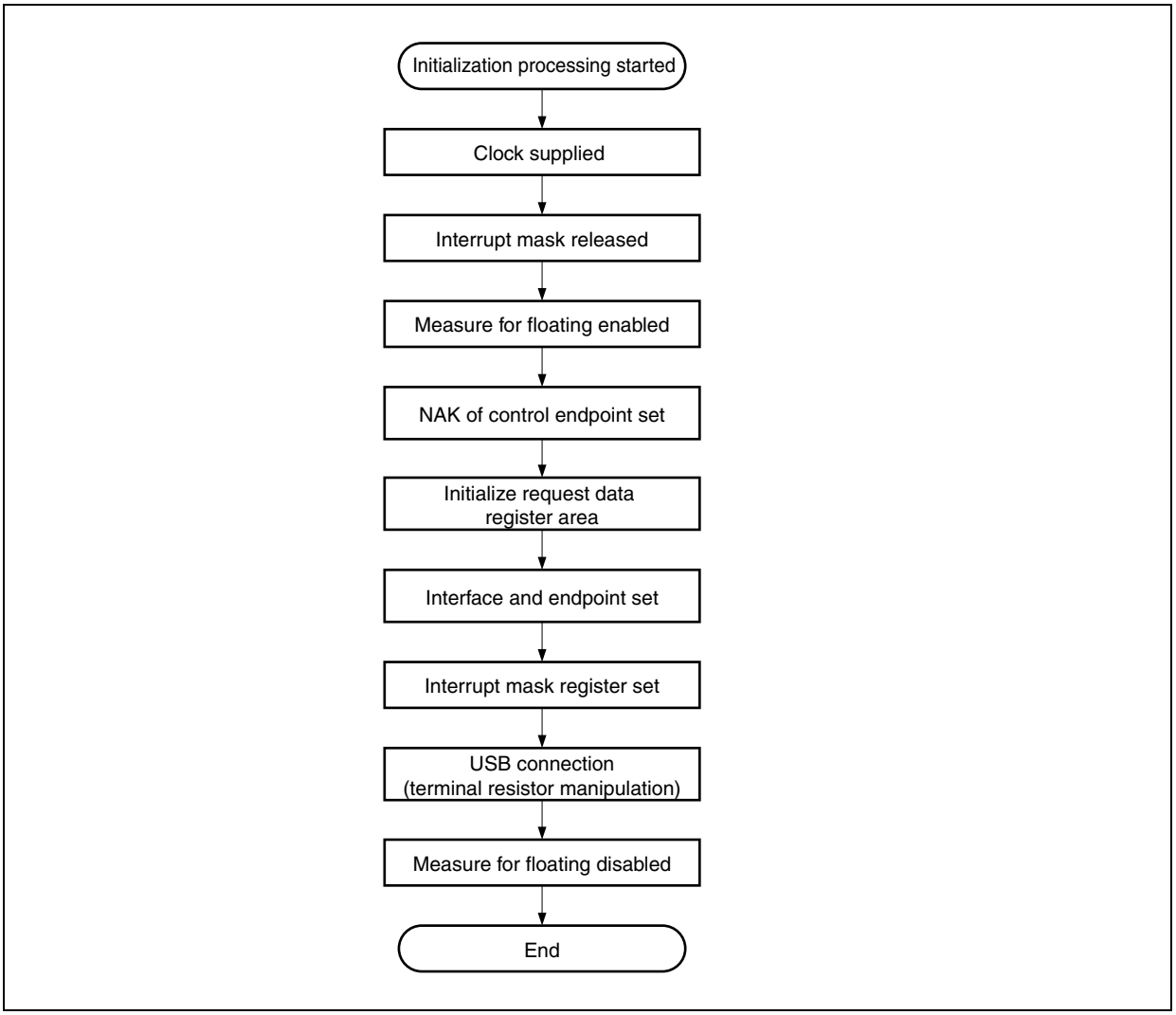
4.7.2 Processing flows

The processing flows of initialization processing and interrupt servicing in the sample program are shown below.

(1) Initialization processing

Initialization processing of the USB device is called and executed by the software initialization module.
The flow of USB device initialization processing (at power application) in the sample program is shown below.

Figure 4-10. Flowchart of Initialization Processing



The processing executed by the initialization processing is shown below.

Caution Initialization processing is required except for processing of ports. The pin assignment may differ if another target board is used. In such a case, read the descriptions in this manual making changes as necessary to match the specifications of the target board to be used.

- Clock supply
  Be sure to set the UCKC.UCKCNT bit to 1 before setting the USB function controller register. A clock to USB is supplied by setting this bit to 1.
  The P10 pin is used for inputting a clock, so set the P10 pin to input mode to enable clock input.

- Release of interrupt mask
  Masking of the USB-related interrupt signal is released using the interrupt control register.

- Enabling floating measure
  The UF0BC.UBFIOR bit is cleared to 0 to prevent mis-recognition due to a bus reset caused by an undefined value when the cable is disconnected.

- Setting of NAK for control endpoint
  A NAK response is sent to all the requests including automatic execution requests.
  This setting is made so that hardware does not return unexpected data in response to an automatic execution request until registration of data used for the automatic execution request is complete.

- Initialization of request data register area
  Descriptor data used to respond to a Get Descriptor request is registered in a register.
  Data such as device status, endpoint 0 status, device descriptor, configuration descriptor, interface descriptor, and endpoint descriptor are registered.

  Caution Registration of the descriptor for the class may be required depending on the class. The USB communication class is defined in this sample program, and only the USB standard descriptor is used.

- Setting of interface and endpoint
  Information such as the number of supported interfaces, the state of alternative settings, relationship between the interface and endpoints are set to a register.

- Release of NAK setting at control endpoint
  The NAK setting at control endpoint (endpoint 0) is released when registration of data for an automatic execution request is complete.

- Setting of interrupt mask register
  Masking for each interrupt source shown in the interrupt status register of the USB function controller.

- USB connection (terminal resistor manipulation)
  The D+ signal is pulled up.

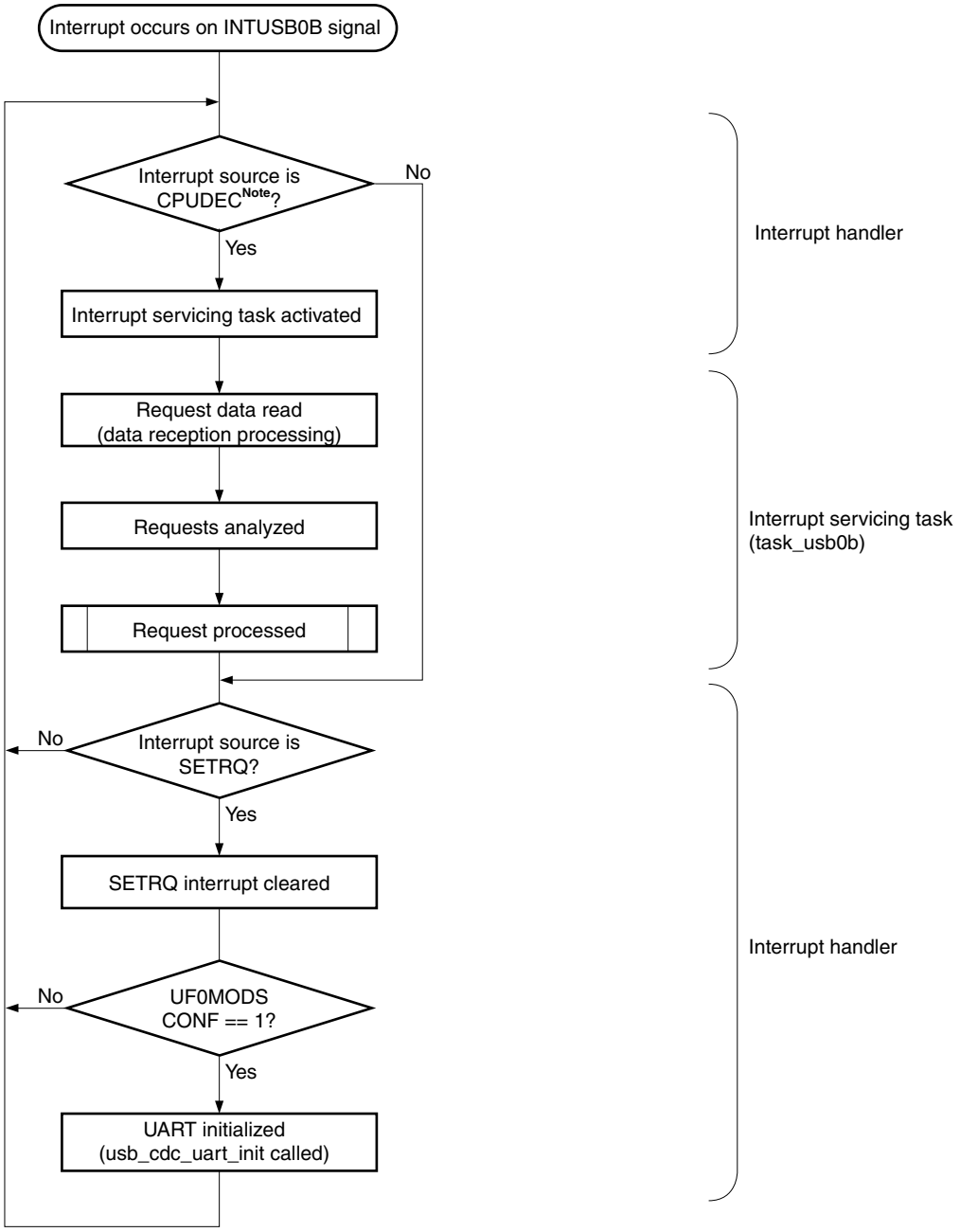
- Disabling floating measure
  The floating measure is disabled by setting the UF0BC.UBFIOR bit to 1.

(2) Interrupt servicing

The sample program operates by interrupt events after initialization. The device is in the idle state as long as no event occurs. However, interrupts are reported by UART, as well as by the USB function controller.
Figures 4-11 and 4-12 show the interrupt servicing flows in the sample program.

Caution The flowchart in Figure 4-11 illustrates the flow of interrupt servicing reported by the INTUSB0B signal of the USB function controller.
The flowchart in Figure 4-12 illustrates the flow of interrupt servicing reported by the INTUSB1B signal of the USB function controller.
Refer to 4.8 UART Processing Module for details of UART interrupt servicing.

Figure 4-11. Flowchart of Interrupt Servicing (1)



Note The CPUDEC and SETRQ interrupts can be acknowledged using the INTUSB0B signal of the USB communication class driver.

The processing of an interrupt by the INTUSB0B signal in the sample program is shown below.

[Processing in interrupt handler]

- Confirmation of interrupt source
  In this sample program, the analyzed interrupt status varies depending on the executed interrupt handler.
  The CPUDEC and SETRQ interrupts can be acknowledged using the INTUSB0B signal. When these interrupts occur, the interrupt handler is activated by the INTUSB0B signal. This interrupt handler reads the UF0IS1 register and judges if the interrupt source is CPUDEC interrupt or not. Furthermore, the interrupt handler reads the UF0IS0 register to confirm whether or not the CONF bit is set to 1.

  Caution In this sample program, the interrupt handlers to be used are registered in the CF definition file in advance.

- Activation of interrupt servicing task
  The *task_usb0b* task is activated if the interrupt source is CPUDEC.

  Caution In this sample program, the tasks to be activated are registered in the CF definition file in advance.

- Initialization of UART
  If the interrupt source is SETRQ, the interrupt handler reads the UF0IS0 register to confirm whether or not the CONF bit is set to 1. If the CONF bit 1, the interrupt handler calls the *usb_cdc_uart_init* function to initialize UART.

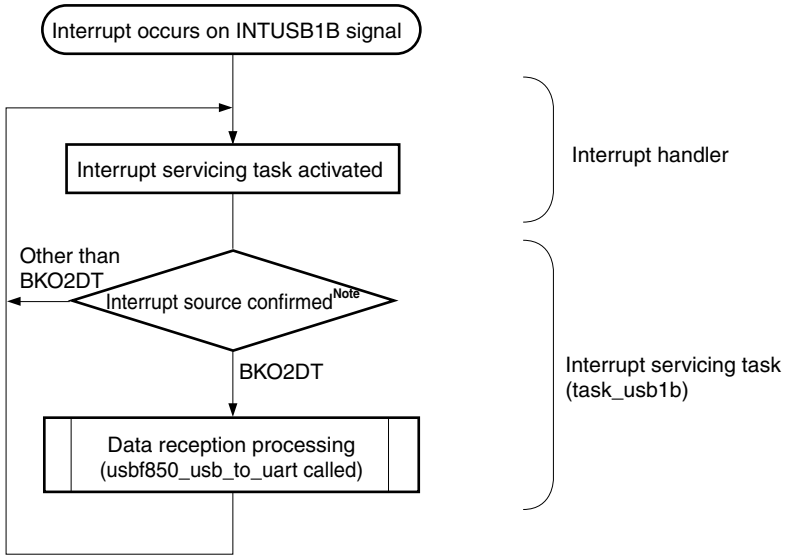
[Processing in task_usb0b task]

- Reading request data
  SETUP data is read from the UF0E0ST register.

- Analysis of request
  SETUP data that has been read is analyzed and the purpose of the request is confirmed.

- Processing of requests
  Processing of the analyzed request is performed.
  In the sample program, the standard device request Get Descriptor (String Descriptor) and device class-specific request are handled.

Figure 4-12. Flowchart of Interrupt Servicing (2)



Note Only the BKO2DT interrupt can be acknowledged using the INTUSB1B signal of the USB communication class driver.

The processing of an interrupt by the INTUSB1B signal in the sample program is shown below.

- Activation of interrupt servicing task
  The *task_usb1b* task is activated without confirming the interrupt source.

  Caution In this sample program, the tasks to be activated are registered in the CF definition file in advance.

- Confirmation of interrupt source
  Only the BKO2DT interrupt can be acknowledged using the INTUSB1B signal. When this interrupt occurs, the interrupt handler is activated by the INTUSB1B signal.
  The interrupt handler does not check the interrupt source, but the activated task confirms that the interrupt source is BKO2DT.

  Caution In this sample program, the interrupt handlers to be used are registered in the CF definition file in advance.

- Data reception processing
  If the interrupt source is BKO2DT, the function to transfer data to UART (usbf850_usb_to_uart) is called from the USB device.

  Remark Refer to **4.8 UART Processing Module** for details of UART processing.

4.7.3 USB communication class driver descriptor information

The USB standard descriptors defined in this sample program are shown below.
Descriptors described in (a) to (d) are the minimum required descriptors.

Remark Refer to **Universal Serial Bus Specification Revision 1.1** for details.

(a) Device descriptor

This descriptor holds general information of the device. One device descriptor must be prepared for each device. The information contained in this descriptor is used for identifying a unique in the device configuration. The USB communication class is defined in this sample program.

Table 4-1. Device Descriptor

Offset	Size (Byte)	Value	Description
0	1	12H	Length value of this descriptor (byte)
1	1	01H	Descriptor type (device)
2	2	10H/01H	USB version (USB 1.1)
4	1	02H	Class code (communication device class)
5	1	00H	Sub-class code
6	1	00H	Protocol code
7	1	40H	Maximum packet size at endpoint 0
8	2	09H/04H	Vendor ID (NEC Electronics)
10	2	FDH/FFH	Product ID
12	2	01H/00H	Device release number
14	1	01H	Index to string descriptor (Manufacturer)
15	1	02H	Index to string descriptor (Product)
16	1	03H	Index to string descriptor (Serial Number)
17	1	01H	Number of devices that can be configured

(b) Configuration descriptor

This descriptor holds information on concrete device configuration.

Table 4-2. Configuration Descriptor

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	02H	Descriptor type (configuration)
2	2	30H/00H	Total length value of descriptor returned together with configuration descriptor in response to the Get Descriptor request
4	1	02H	Number of interfaces supported in the configuration
5	1	01H	Configuration value
6	1	00H	Index to string descriptor (configuration)
7	1	C0H	Configuration of device (self-powered/remote wakeup function)
8	1	00H	Maximum power consumption of device

(c) Interface descriptor

This descriptor holds concrete interface information in the configuration.

The configuration provides two interfaces in this sample program.

This descriptor is always returned as a part of the configuration descriptor, and is not accessed directly by a Get Descriptor request or Set Descriptor request.

Table 4-3. Interface Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	04H	Descriptor type (interface)
2	1	00H	Interface value
3	1	00H	*Alternate* set value
4	1	01H	Endpoint number (excluding endpoint 0)
5	1	02H	Interface class (communication interface class)
6	1	02H	Interface sub-class (abstract control model)
7	1	00H	Interface protocol (No Class: USB specification)
8	1	00H	Index to string descriptor (interface)

Table 4-4. Interface Descriptor (2)

Offset	Size (Byte)	Value	Description
0	1	09H	Length value of this descriptor (byte)
1	1	04H	Descriptor type (interface)
2	1	01H	Interface value
3	1	00H	*Alternate* set value
4	1	02H	Endpoint number (excluding endpoint 0)
5	1	0aH	Interface class (data interface class)
6	1	00H	Interface sub-class (data class)
7	1	00H	Interface protocol (No Class: USB specification)
8	1	00H	Index to string descriptor (interface)

(d) Endpoint descriptor

This descriptor holds information required by the host for determining the bandwidth requirements for each endpoint.

This descriptor is always returned as a part of the configuration descriptor, and is not accessed directly by a Get Descriptor request or Set Descriptor request.

Table 4-5. Endpoint Descriptor (Interrupt IN)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (byte)
1	1	05H	Descriptor type (endpoint)
2	1	87H	Endpoint address value
3	1	03H	Endpoint transfer type
4	2	08H/00H	Maximum packet size at endpoint
6	1	0AH	Interval (ms): Valid only for isochronous and interrupt endpoints

Table 4-6. Endpoint Descriptor (Bulk IN)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (byte)
1	1	05H	Descriptor type (endpoint)
2	1	83H	Endpoint address value
3	1	02H	Endpoint transfer type
4	2	40H/00H	Maximum packet size at endpoint
6	1	00H	Interval (ms): Valid only for isochronous and interrupt endpoints

Table 4-7. Endpoint Descriptor (Bulk OUT)

Offset	Size (Byte)	Value	Description
0	1	07H	Length value of this descriptor (byte)
1	1	05H	Descriptor type (endpoint)
2	1	04H	Endpoint address value
3	1	02H	Endpoint transfer type
4	2	40H/00H	Maximum packet size at endpoint
6	1	00H	Interval (ms): Valid only for isochronous and interrupt endpoints

(e) String descriptor

This descriptor holds information on the manufacturer of the device in this sample program.

Table 4-8. String Descriptor (1)

Offset	Size (Byte)	Value	Description
0	1	04H	Length value of this descriptor (byte)
1	1	03H	Descriptor type (string)
2	2	09H/04H	Language type used by string descriptor (English/US)

Table 4-9. String Descriptor (2)

Offset	Size (Byte)	Value	Description
0	1	2AH	Length value of this descriptor (byte)
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: NEC Electronics Co.

Table 4-10. String Descriptor (3)

Offset	Size (Byte)	Value	Description
0	1	16H	Length value of this descriptor (byte)
1	1	06H	Descriptor type (string)
2	20	'C','o','m','m','u','n','i','D','r','v'	Product: CommuniDrv

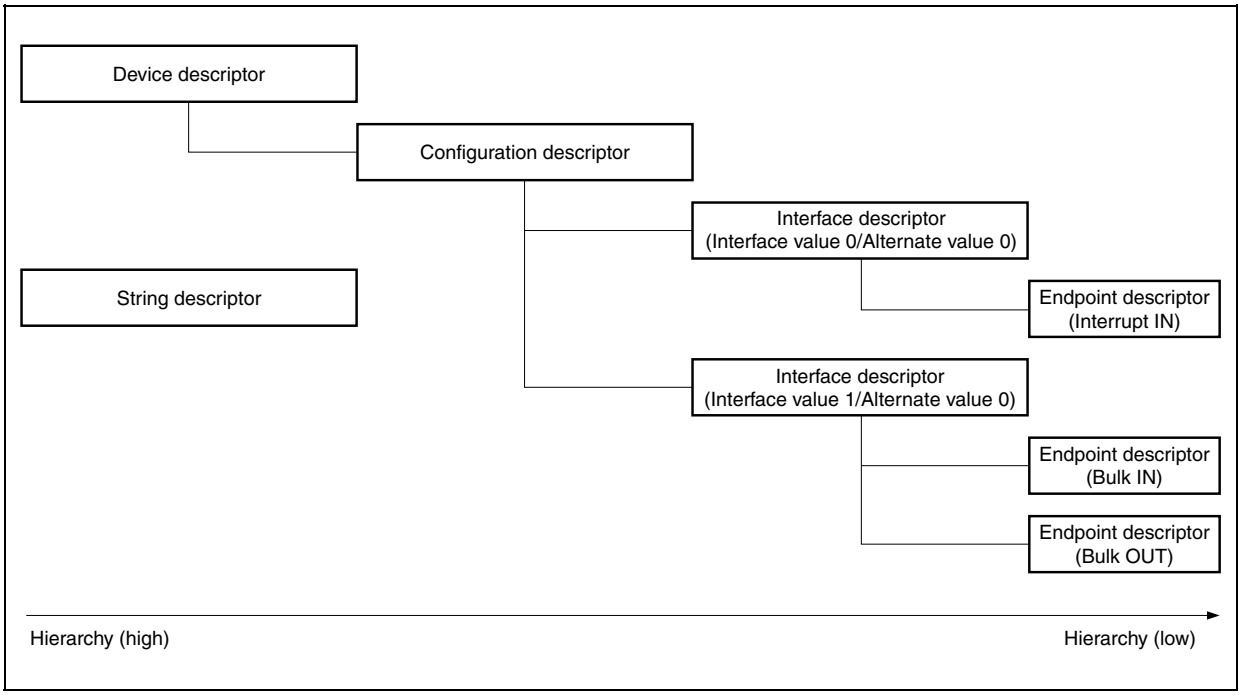
Table 4-11. String Descriptor (4)

Offset	Size (Byte)	Value	Description
0	1	16H	Length value of this descriptor (byte)
1	1	06H	Descriptor type (string)
2	20	'0','_','9','8','7','6','5','4','3','2'	Serial number: 0_98765432

- Descriptor configuration
  The descriptor configuration in this sample program is shown below. This configuration consists of the five descriptors described before.

Caution The device descriptor, configuration descriptor, and string descriptor are accessed by an independent Get Descriptor request. The interface descriptor and endpoint descriptor are accessed as part of the configuration descriptor.

Figure 4-13. Descriptor Configuration



4.7.4 Data macro

The data macros (data type, return value, etc.) used by the USB communication class driver are shown below.

(1) Data type

Data type macro for parameters specified when a USB communication class driver function is called is defined in the header file *types.h* in nectools32\USB_CDC\inc.

A list of the data types is shown below.

Table 4-12. List of Data Types

Macro	Type	Description
(*PFV) ()	void	Processing module activation address

(2) Return value

Macro of the return value from USB communication class driver function is defined in the header file *errno.h* in nectools32\USB_CDC\inc.

A list of the return values is shown below.

Table 4-13. List of Return Values

Macro	Type	Description
DEV_OK	0	Normal termination
DEV_ERROR	−1	Abnormal termination

4.7.5 Data structure

The data structure used by the USB communication class driver is shown below.

(1) USB device request structure

The USB device request structure is defined in USB header file *usbf850.h* in nectools32\V850USB_CDC\src \USBF. 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;
```

(2) UART mode table structure

The UART mode table structure is defined in header file *types.h* in nectools32\V850USB_CDC\inc. The UART mode table structure UART_MODE_TBL is shown below.

```
typedef struct _UART_MODE_TBL{
         char  DTERate[4]; /*transfer rate(bps)*/
         char  STOPBIT;    /*length of the stop bit - 0:1bit (1:1.5bits) 2:2bits*/
         char  PARITYType; /*parity bit - 0:None 1:Odd 2:Even (3:Mark) 4:Space */
         char  DATABits;   /*data size (number of the bits:5,6,7,8,16) */
} UART_MODE_TBL , *PUART_MODE_TBL;
```

4.7.6 Description of functions

(1) Overview

A list of the processing modules described in this chapter is shown below.

Caution Functions starting with "usbf850" are used by the USB function controller incorporated in the V850E/ME2. Functions starting with "uartb0850" are used by UARTB0 incorporated in the V850E/ME2. Refer to 4.8.4 Description of functions for details of UARTB0 functions.

Table 4-14. List of Processing Modules in Sample Program (1/3)

Processing Module Name	Function Name	File Name	Remark
RX850 Pro-dependent processing module			
CF definition file	–	sys.cf	–
Entry processing	–	entry.850	Assembly language
Boot processing	boot	boot.850	Assembly language
Hardware initialization module	__InitSystemTimer	init.c	C language
Initialization handler	varfunc	varfunc.c	C language
Header file	–	init.h	–
Board-dependent processing module			
Port initialization	port850_reset	port.c	C language
Header file	–	port.h	–
Header file			
Data type declaration	–	types.h	–
Return value declaration	–	errno.h	–
Build file	–	usb_bus.bld	–
Section map file	–	common.lx	–

Table 4-14. List of Processing Modules in Sample Program (2/3)

Processing Module Name	Function Name	File Name	Remark
USB communication class driver processing module (USB processing module)			
Initialization function	usbf850_init	usbf850.c	C language
Interrupt handler (INTUSB0B signal)	usbf850_inthdr	usbf850.c	C language
Interrupt handler (INTUSB1B signal)	usbf850_inthdr1	usbf850.c	C language
Interrupt handler (INTUSB2B signal)	usbf850_inthdr2	usbf850.c	C language
Interrupt servicing task (INTUSB0B signal)	task_usb0b	usbf850.c	C language
Interrupt servicing task (INTUSB1B signal)	task_usb1b	usbf850.c	C language
Interrupt servicing task (INTUSB2B signal)	task_usb2b	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 calling function (loc_cpu)	usbf850_loc_cpu	usbf850.c	C language
System call calling 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	–

Table 4-14. List of Processing Modules in Sample Program (3/3)

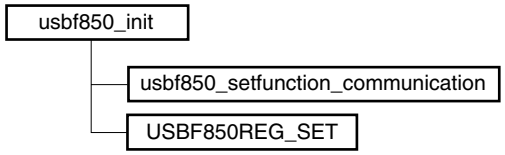
Processing Module Name	Function Name	File Name	Remark
USB communication class driver processing module (USB-UART interface module)			
USB-UART interface initialization function	usb_cdc_init	usbf850_communication.c	C language
SEND ENCAPSULATED COMMAND request processing function	usbf850_send_encapsulated_command	usbf850_communication.c	C language
GET ENCAPSULATED RESPONSE request processing function	usbf850_get_encapsulated_response	usbf850_communication.c	C language
SET LINE CODING request processing function	usbf850_set_line_coding	usbf850_communication.c	C language
GET LINE CODING request processing function	usbf850_get_line_coding	usbf850_communication.c	C language
SET CONTROL LINE STATE request processing function	usbf850_set_control_line_state	usbf850_communication.c	C language
USB-UART data transmission function	usbf850_usb_to_uart	usbf850_communication.c	C language
USB-UART data transmission function	usbf850_uart_to_usb	usbf850_communication.c	C language
Registration processing function of device class-specific request processing function for USB communication class	usbf850_setfunction_communication	usbf850_communication.c	C language
Header file for USB-UART interface function	–	usbf850_ communication.h	–
Function macro			
V850E/ME2 peripheral I/O register setting function (1-byte units: 8 bits)	USBF850REG_SET	usbf850.h	C language
V850E/ME2 peripheral I/O register read function (1-byte units: 8 bits)	USBF850REG_READ	usbf850.h	C language
V850E/ME2 peripheral I/O register setting function (1-word units: 16 bits)	USBF850REG_SET_W	usbf850.h	C language
V850E/ME2 peripheral I/O register read function (1-word units: 16 bits)	USBF850REG_READ_W	usbf850.h	C language

(2) Function tree

The calling relationship between the USB communication class driver processing-dependent modules (function tree) is illustrated below.

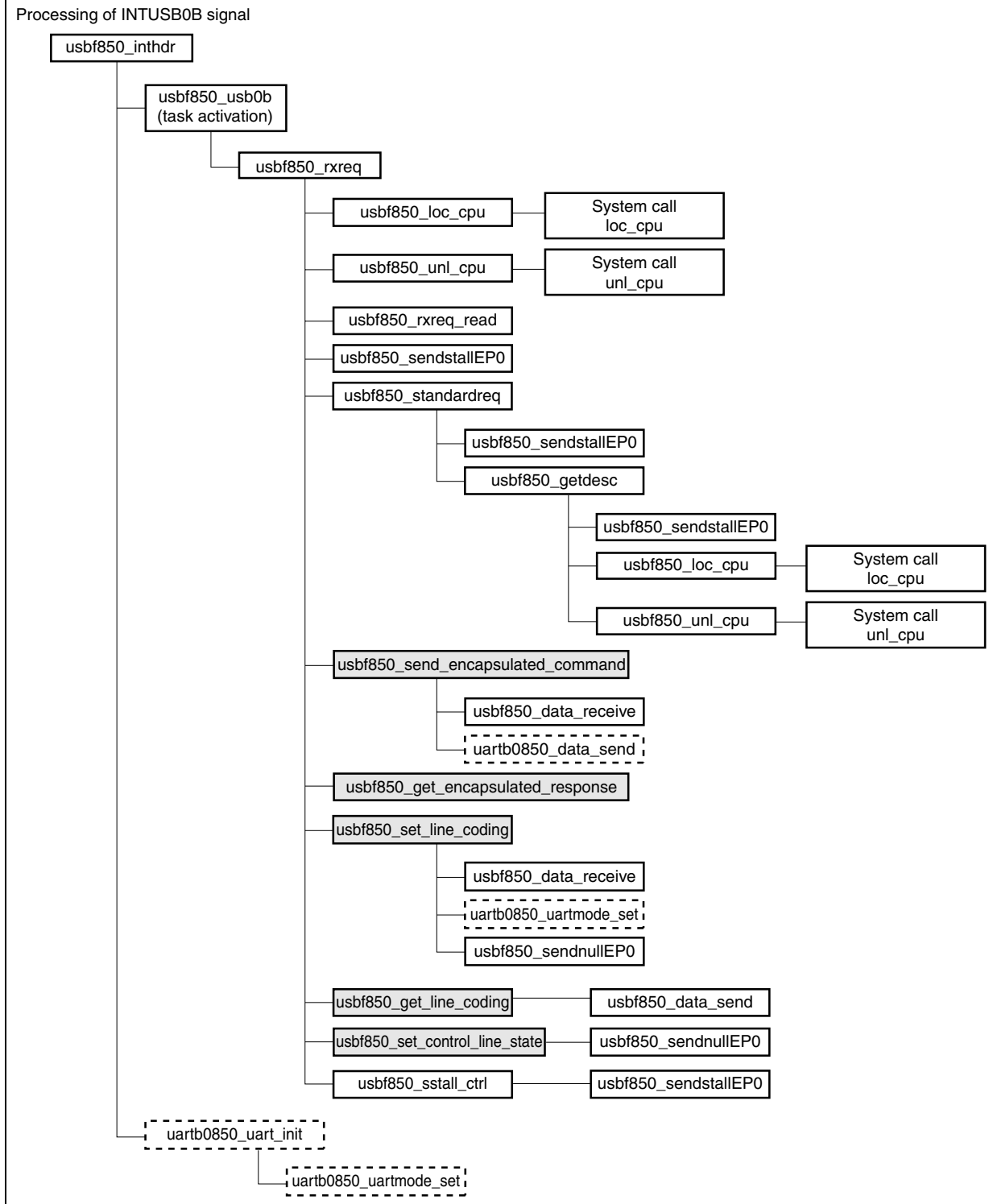
Remark Refer to **4.8.4 (2) Function tree** for details of the calling processing of the UART processing module.

Figure 4-14. Sample Program Function Tree (1/3)



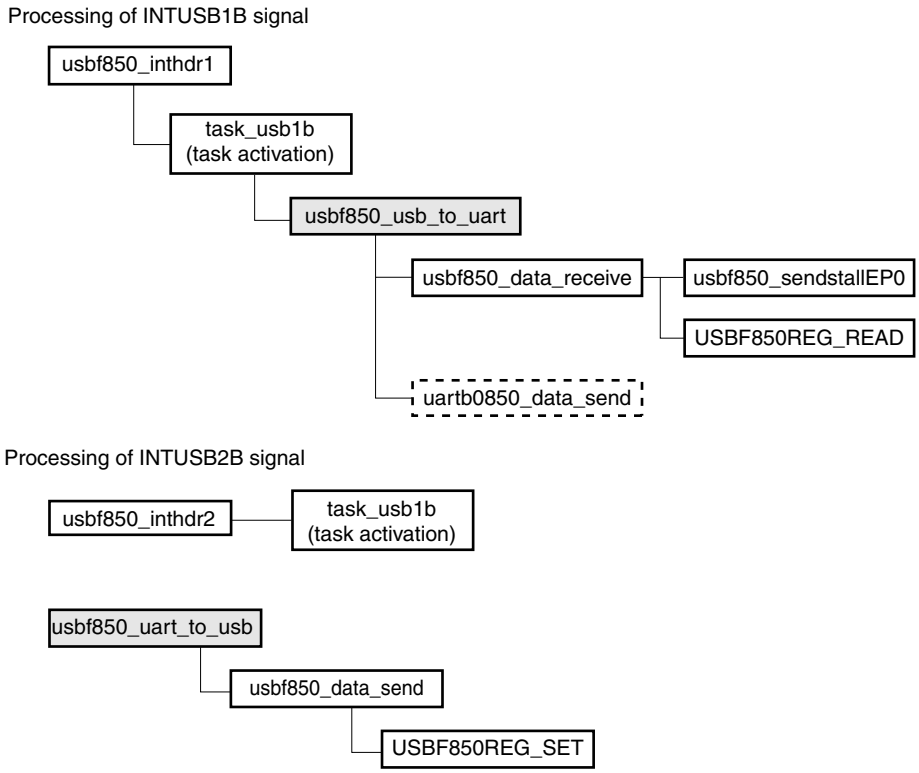
Caution *usbf850_init* is called from the initialization handler.

Figure 4-14. Sample Program Function Tree (2/3)



Remark The shaded portions in this figure indicate the USB-UART interface functions described in this section. The portions enclosed by the dashed lines indicate the UART processing module functions. Refer to **4.8 UART Processing Module** for details of the UART processing module function.

Figure 4-14. Sample Program Function Tree (3/3)



Remark The shaded portions in this figure indicate the USB-UART interface functions described in this section. The portions enclosed by the dashed lines indicate the UART processing module functions. Refer to **4.8 UART Processing Module** for details of the UART processing module function.

(3) Description of functions

The functions of the USB communication class driver processing-dependent module are explained in the following format.

Remark The functions of the UART processing module are explained in the same format as described in **4.8.4 (3) Description of functions**.

xxxx … <1>	Valid caller: – – – – … <2>

[Outline] … <3>

– –
– –

[C language format] … <4>

– – – – – – – –

[Parameter] … <5>

I/O	Parameter	Description

[Operation] … <6>

– –
– –

[Return value] … <7>

– –
– –

<1> Name

Indicates the function name.

<2> Valid caller

Indicates the type of the processing module from which a function can be called.

Task:	The function can be called only from a task.
Non-task:	The function can be called only from a non-task.
Non-task | Task:	The function can be called from a task or non-task.
–:	Interrupt handler or task, and is not used to call functions.

<3> Outline

Shows the outline of a function operation.

<4> C language format

Shows the description format when calling a function from the processing module described in the C language.

<5> Parameter

Shows the function parameter 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 parameter

<6> Operation

Describes detailed operation of the function.

<7> Return value

Indicates the return value from a function using the data macro or numeric value.

usbf850_init	Valid caller: Non-task I Task

[Outline]

This is a function that initializes the USB function controller incorporated in the V850E/ME2.

[C language format]

```
void usbf850_init (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called from the software initialization module and performs processing to initialize the USB function controller incorporated in the V850E/ME2.

Remark Refer to **4.7.2 (1) Initialization processing** for details of initialization processing.

[Return value]

None

usbf850_inthdr	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB0B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This is the interrupt handler activated by the INTUSB0B signal (USB function status 0).

In this sample program, the interrupt handler checks the interrupt source and activates the interrupt servicing task (task_usb0b) only when the source is the CPUDEC interrupt. If the source is the SETRQ interrupt, the interrupt handler calls the USB-UART interface initialization processing function (uartb0850_uart_init). This handler is defined in the CF definition file.

Remark Refer to **4.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

usbf850_inthdr1	Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB1B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr1 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB1B signal (USB function status 1).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb1b). This handler is defined in the CF definition file.

Remark Refer to **4.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

usbf850_inthdr2 Valid caller: –

[Outline]

This is an interrupt handler (for the INTUSB2B signal) used by the USB function controller incorporated in the V850E/ME2.

[C language format]

```
ID usbf850_inthdr2 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the INTUSB2B signal (USB function status 2).

In this sample program, the interrupt handler activates the interrupt servicing task (task_usb2b). This handler is defined in the CF definition file.

Caution This handler is not called because all the interrupts reported by the INTUSB2B signal are masked in this sample program.

[Return value]

Object ID number (task ID number)

task_usb0b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB0B signal.

[C language format]

```
void task_usb0b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB0B interrupt signal (USB function status 0 interrupt). In the sample program, this task calls the *usbf850_rxreq* function and performs processing of the USB standard device request and device class-specific request.

Caution In this sample program, the standard device request Get Descriptor (String Descriptor) that is not responded automatically by the USB function controller incorporated in the V850E/ME2 is handled.

Remark Refer to **4.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

None

task_usb1b Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB1B signal.

[C language format]

```
void task_usb1b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB1B interrupt signal (USB function status 1 interrupt). In the sample program, this task confirms the interrupt source and calls the function (usbf850_usb_to_uart) to transfer data from the USB device to UART, if the interrupt source is BKO2DT.

Remark Refer to **4.7.2 (2) Interrupt servicing** for details of interrupt servicing.

[Return value]

None

task_usb2b	Valid caller: –

[Outline]

This is a task that performs interrupt servicing by the INTUSB2B signal.

[C language format]

```
void task_usb2b (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the INTUSB2B interrupt signal (USB function status 2 interrupt). This processing is not provided in the sample program, so the program returns without processing.

Caution This function is not used in the sample program.

[Return value]

None

usbf850_data_send Valid caller: Non-task | Task

[Outline]

This is a data transmit function used by the USB function controller.

[C language format]

```
long usbf850_data_send (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of transmit data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function transmits from the endpoint specified by *ep* data whose size is specified by *len* starting from the address specified by *data*.

[Return value]

Status upon transmission

DEV_ERROR: Endpoint number is illegal

DEV_OK: Normal termination

usbf850_data_receive Valid caller: Non-task | Task

[Outline]

This is a data receive function used by the USB function controller.

[C language format]

```
long usbf850_data_receive (unsigned char* data, long len, char ep)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of the buffer for receive data
I	long len	Data size
I	char ep	Endpoint number

[Operation]

This function reads data whose size is specified by *len* from the buffer at the endpoint specified by *ep* and stores it to the address specified by specified *data*.

[Return value]

Status upon reception

DEV_ERROR: Receive data size is illegal, or endpoint number is illegal.

DEV_OK: Normal termination

usbf850_sendnullEP0 Valid caller: Non-task | Task

[Outline]

This is a function that transmits Null data from the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendnullEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function transmits Null data (whose data size is 0) from the control endpoint (endpoint 0).

[Return value]

None

usb850_sendstallEP0	Valid caller: Non-task I Task

[Outline]

This is a function that sends a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sendstallEP0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the control endpoint (endpoint 0).

[Return value]

None

usbf850_bulkin1_stall Valid caller: Non-task | Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 1).

[C language format]

```
void usbf850_bulkin1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 1).

Caution This function is not used in the sample program.

[Return value]

None

usbf850_bulkout1_stall	Valid caller: Non-task I Task

[Outline]

This is a function that sets a STALL response for the bulk endpoint (endpoint 2).

[C language format]

```
void usbf850_bulkout1_stall (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response for the bulk endpoint (endpoint 2).

Caution This function is not used in the sample program.

[Return value]

None

usbf850_loc_cpu	Valid caller: Task

[Outline]

This is a function that disables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_loc_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *loc_cpu* system call.

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

[Return value]

None

usbf850_unl_cpu	Valid caller: Task

[Outline]

This is a function that enables acknowledgment of maskable interrupts and dispatch processing.

[C language format]

```
void usbf850_unl_cpu (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls the *unl_cpu* system call.

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

[Return value]

None

usbf850_rxreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs USB request processing.

[C language format]

```
void usbf850_rxreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *task_usb0b* task that is activated by the INTUSB0B interrupt signal. This function calls SETUP data read processing, analyzes the read data, and calls USB request processing based on the analysis result.

[Return value]

None

usbf850_rxreq_read	Valid caller: Non-task I Task

[Outline]

This is a function that reads USB request data.

[C language format]

```
void usbf850_rxreq_read (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function reads SETUP data received subsequently to the Setup token at the control endpoint (endpoint 0). The SETUP data is distinguished from normal data and is stored in a dedicated register. It is always read in 8-byte units.

[Return value]

None

usbf850_standardreq	Valid caller: Non-task I Task

[Outline]

This is a function that performs the USB standard request.

[C language format]

```
void usbf850_standardreq (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called if the standard request is read from SETUP data and calls the *usbf850_getdesc* function when the request type is confirmed as the Get Descriptor request.

[Return value]

None

usbf850_getdesc	Valid caller: Non-task I Task

[Outline]

This is a function that performs the USB standard request Get Descriptor (String Descriptor) processing.

[C language format]

```
void usbf850_getdesc (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called by the *usbf850_standardreq* function and performs the USB standard request Get Descriptor (String Descriptor) processing. This function sets a STALL response for a request other than the Get Descriptor (String Descriptor) request.

[Return value]

None

usbf850_sstall_ctrl	Valid caller: Non-task | Task

[Outline]

This is a function that sets a STALL response for the control endpoint (endpoint 0).

[C language format]

```
void usbf850_sstall_ctrl (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets a STALL response at the control endpoint (endpoint 0).

With the *usbf850_setfunction_communication* function, when a class request processing function is prepared as the function pointer for an array, this function uses a request code as a subscript for array. If this function is registered to a location where is no relevant request, a STALL response can be set when an unsupported request code is sent.

[Return value]

None

usbf850_send_encapsulated_command	Valid caller: Non-task I Task

[Outline]

This is a function that handles the USB communication class-specific request (SEND ENCAPSULATED COMMAND).

[C language format]

```
void usbf850_send_encapsulated_command (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs SEND ENCAPSULATED COMMAND request processing. Command data sent from the host is passed to the CDC device. When this request is received, this function in the sample program receives data reported subsequently to the request from the host, transmits the data to UART, and sends a NULL response.

[Return value]

None

usbf850_get_encapsulated_response Valid caller: Non-task | Task

[Outline]

This is a function that handles the USB communication class-specific request (GET ENCAPSULATED RESPONSE).

[C language format]

```
void usbf850_get_encapsulated_response (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs GET ENCAPSULATED RESPONSE request processing. Command data sent from the host is passed to the CDC device. When this request is received, this function in the sample program performs no processing and ends normally.

[Return value]

None

usbf850_set_line_coding Valid caller: Non-task | Task

[Outline]

This is a function that handles the USB communication class-specific request (SET LINE CODING).

[C language format]

```
void usbf850_set_line_coding (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs SET LINE CODING request processing. This function sets the transfer rate, stop bit, parity bit, and data length. When this request is received, this function in the sample program receives data reported subsequently to the request from the host, and writes the data to the UART_MODE_INFO structure. In addition, this function sets the mode of UART based on the value written to the UART_MODE_ INFO structure and sends a NULL response.

[Return value]

None

usbf850_get_line_coding	Valid caller: Non-task I Task

[Outline]

This is a function that handles the USB communication class-specific request (GET LINE CODING).

[C language format]

```
void usbf850_get_line_coding (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs GET LINE CODING request processing. This function sends the current values set for the transfer rate, stop bit, parity bit, and data length to the host. When this request is received, this function in the sample program sends the current UART set values set to the UART_ MODE_INFO structure to the host.

[Return value]

None

usbf850_set_control_line_state	Valid caller: Non-task I Task

[Outline]

This is a function that handles the USB communication class-specific request (SET CONTROL LINE STATE).

[C language format]

```
void usbf850_set_control_line_state (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function performs SET CONTROL LINE STATE request processing. This function sets the RS-232/V.24 control signal (RTS/DTR). However, this setting is not possible because the RTS/DTR pin is not provided in the V850E/ME2. In this sample program, therefore, this function sends a NULL response and ends normally.

[Return value]

None

usbf850_usb_to_uart	Valid caller: Non-task I Task

[Outline]

This is a function that transfers data from USB to UART.

[C language format]

```
void usbf850_usb_to_uart (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function calls USB data reception processing function (usbf850_data_receive) and performs data reception processing. This function then calls the function to transmit data to UART (usbf850_cdc_data_send) and transmits data to UART.

[Return value]

None

usbf850_uart_to_usb Valid caller: Non-task | Task

[Outline]

This is a function to transfer data from UART to USB.

[C language format]

```
void usbf850_uart_to_usb (unsigned char* data, int len)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* data	Start address of transmit data
I	int	*len* data size

[Operation]

This function calls the function to transmit data to USB (usbf850_data_send) and transmit the UART receive data to USB.

[Return value]

None

usbf850_setfunction_communication Valid caller: Non-task | Task

[Outline]

This is a function that registers a USB communication class-specific request processing function to an array as the function pointer.

[C language format]

```
void usbf850_setfunction_communication (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function is called from USB initialization processing, and registers a USB communication class-specific request processing function as a function pointer to an array (array name: Req_Func_C).

This function registers the *usbf850_sstall_ctrl* function for an unsupported request code to send a STALL response when an unsupported request is sent.

[Return value]

None

USBF850REG_SET	Valid caller: Non-task I Task

[Outline]

This is a function that sets the V850E/ME2 peripheral I/O registers (1-byte units: 8 bits).

[C language format]

```
USBF850REG_SET (offset, val)
```

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Data for setting

[Operation]

This function sets data specified by *val* to the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-byte (8-bit) units.

[Return value]

None

USBF850REG_READ Valid caller: Non-task | Task

[Outline]

This is a function that reads the V850E/ME2 peripheral I/O registers (1-byte units: 8 bits).

[C language format]

```
USBF850REG_READ (offset)
```

[Parameter]

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

[Operation]

This function reads the value in the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-byte (8-bit) units.

[Return value]

None

USBF850REG_SET_W Valid caller: Non-task | Task

[Outline]

This is a function that sets the V850E/ME2 peripheral I/O registers (1-word units: 16 bits).

[C language format]

```
USBF850REG_SET_W (offset, val)
```

[Parameter]

I/O	Parameter	Description
I	offset	Peripheral I/O register address
I	val	Data for setting

[Operation]

This function sets data specified by *val* to the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-word (16-bit) units.

[Return value]

None

USBF850REG_READ_W Valid caller: Non-task | Task

[Outline]

This is a function that reads the V850E/ME2 peripheral I/O registers (1-word units: 16 bits).

[C language format]

```
USBF850REG_READ_W (offset)
```

[Parameter]

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

[Operation]

This function reads the value in the V850E/ME2 peripheral I/O registers (register address specified by *offset*). This macro is valid only for registers that can be accessed in 1-word (16-bit) units.

[Return value]

None

4.8 UART Processing Module

4.8.1 Overview

A simple driver that is used to operate UARTB0 incorporated in the V850E/ME2 is provided as the UART processing module in this sample program. This section explains the UART processing module.

A list of UART processing modules is shown below.

Caution The UART processing module incorporated in the V850E/ME2 is provided in the sample program, but it is only provided for performing minimum processing, so the operation as a general-purpose UART driver is not guaranteed.

Remark Refer to **4.7 USB Communication Class Driver Functions** for details of the USB communication class driver processing-dependent module.

- UART initialization processing
  This is called from the RX850 Pro software initialization module and performs UARTB0 initialization processing.

- UART interrupt handler
  This is an interrupt servicing-dedicated routine called each time a UART interrupt occurs, and is defined in the CF definition file.

  Caution Interrupts other than required are masked in this sample program.
  The following three interrupts are used in this sample program

  - UARTB0 reception error interrupt reported by the UBTIRE signal
  - UARTB0 reception end interrupt reported by the UBTIR0 signal
  - UARTB0 reception timeout interrupt reported by the UBTITO0 signal

- UART interrupt servicing task
  This task is called from the UART interrupt handler and performs data reception processing.

- UART general-purpose function
  The UART data transmission function and operation mode setting function are provided as a general-purpose function used by the UART processing module.

4.8.2 Processing flow

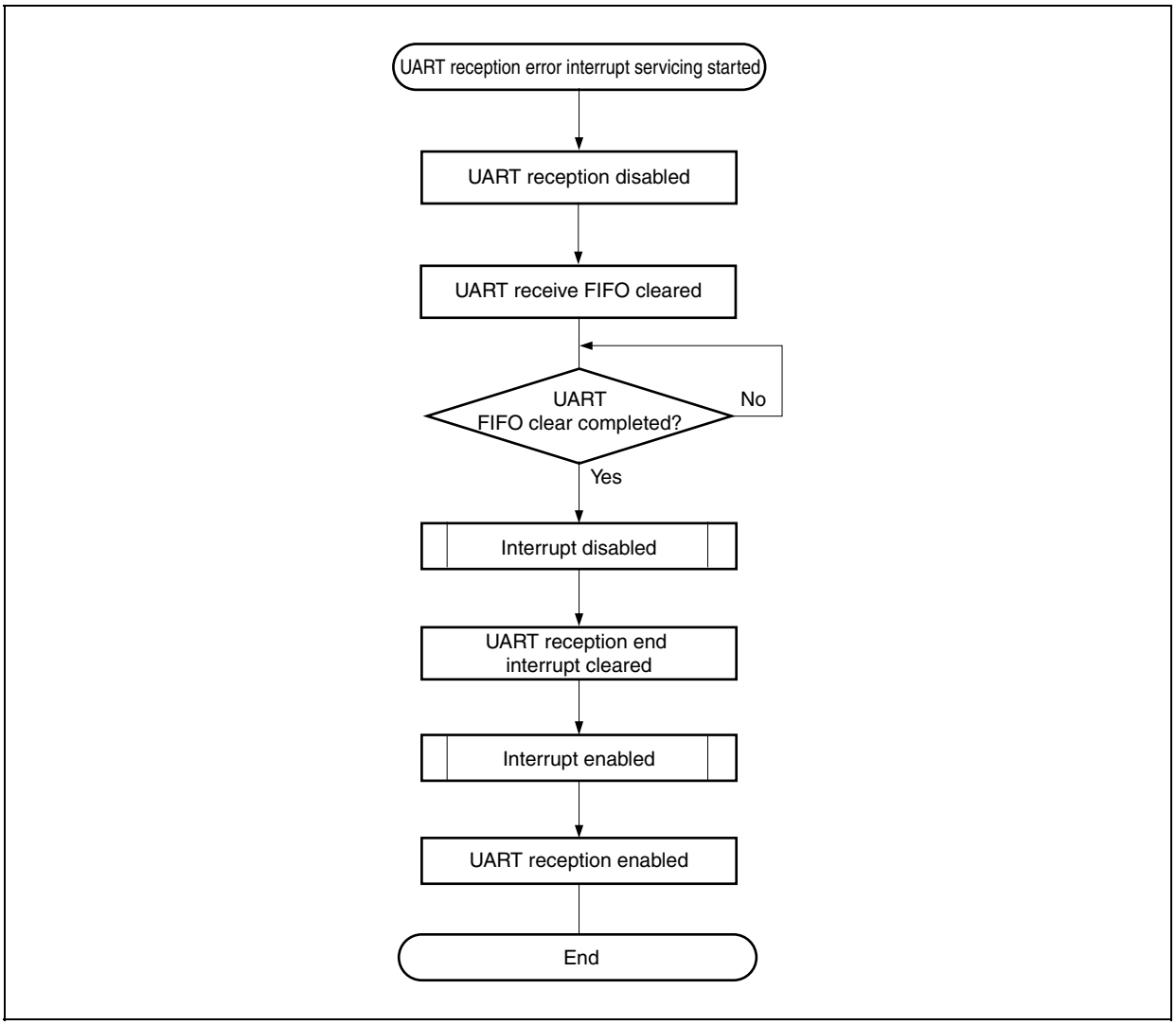
(1) Interrupt servicing

The sample program operates by interrupt events after initialization. It is in the idle state as long as no event occurs. However, interrupts are reported by the USB function controller, as well as by UART.

The flow of UART interrupt servicing in the sample program is shown below.

Remark Refer to **4.7.2 (2) Interrupt servicing** for details of interrupt servicing by the USB function controller.

Figure 4-15. Flowchart of Interrupt Servicing (1)



The processing of an interrupt by the UBTIRE0 signal in the sample program is shown below.

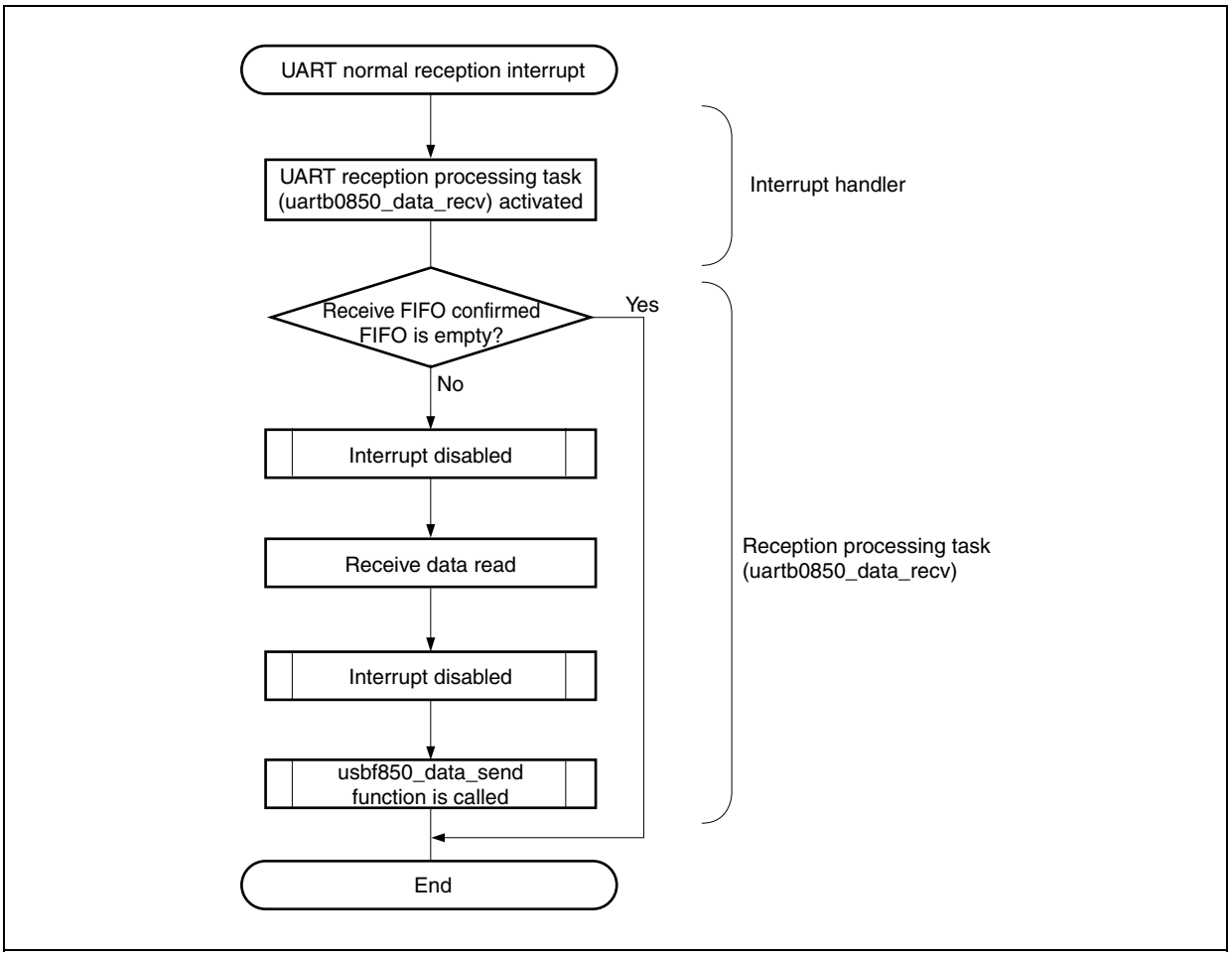
- Disabling UART reception
  UART receive operation is disabled using the UB0CTL0.UB0RXE bit.

- Clearing UART receive FIFO
  The receive FIFO is cleared by the UB0FIC0 register.

- Clearing UART reception end interrupt
  The UART reception end interrupt is cleared by the URIC0 register.

- Enabling UART reception
  UART receive operation is enabled using the UB0CTL0.UB0RXE bit.

Figure 4-16. Flowchart of Interrupt Servicing (2)



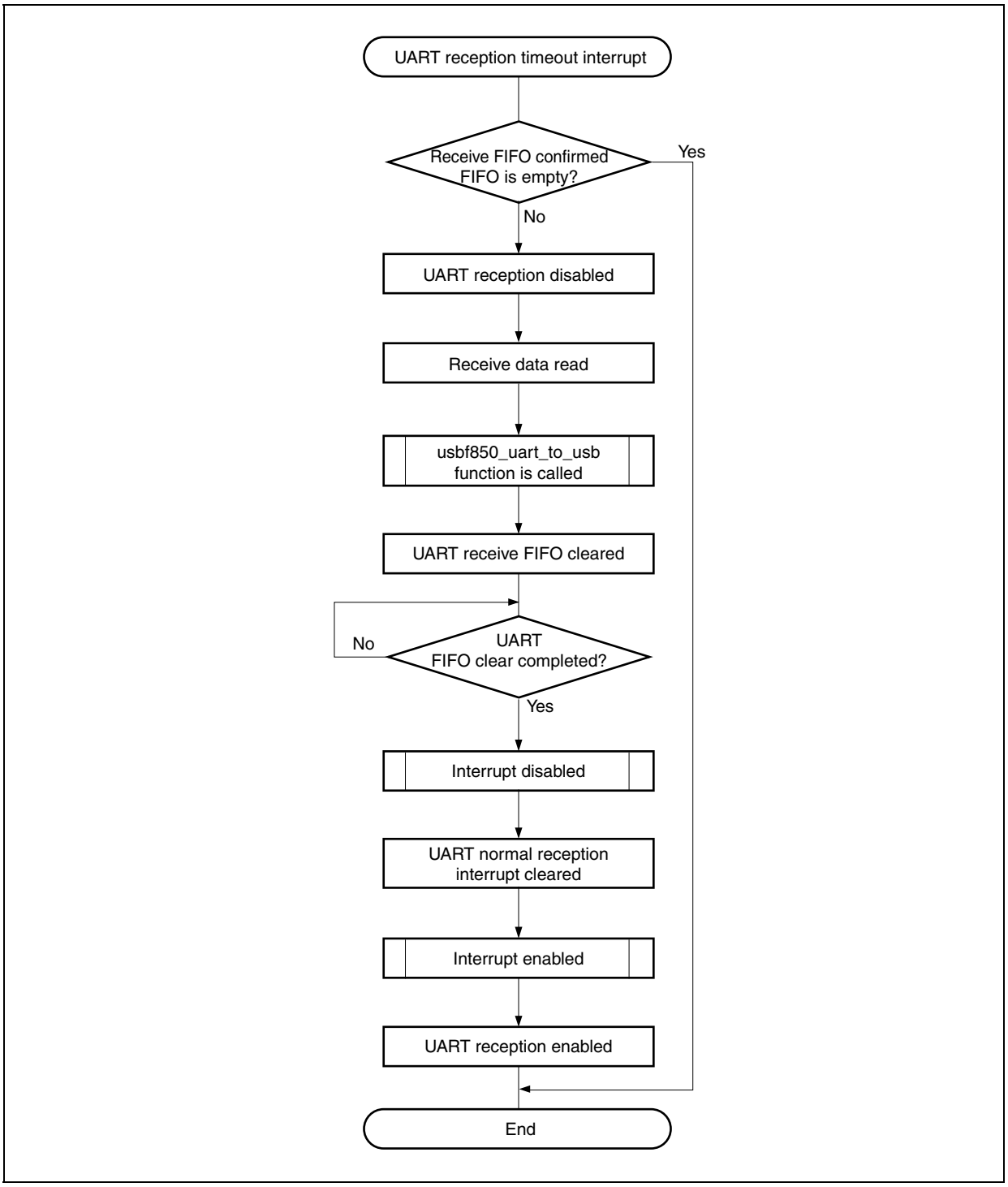
The processing of an interrupt by the UBTIR0 signal in the sample program is shown below.

- Activation of UART reception processing task
  The UART reception processing task (uartb0850_data_recv) is activated.

- Confirmation of UART receive FIFO
  Data storage status in the UART receive FIFO is confirmed; data is read if there is receive data, and the processing ends if there is no receive data.

- Calling *usbf850_data_send* function
  The receive data that has been read is sent to the host.

Figure 4-17. Flowchart of Interrupt Servicing (3)



The processing of an interrupt by the UBTITO0 signal in the sample program is shown below.

- Confirming UART receive FIFO
  Data storage status in the UART receive FIFO is confirmed; UART reception is disabled if there is receive data, and the processing ends if there is no receive data

- Disabling UART reception
  UART receive operation is disabled using the UB0CTL0.UB0RXE bit.

- Reading UART receive data
  Data is read from the UART receive FIFO.

- Calling usbf850_uart_to_usb function
  The receive data that has been read is sent to the host.

- Clearing UART receive FIFO
  The receive FIFO is cleared by the UB0FIC0 register.

- Clearing UART reception end interrupt
  The UART reception end interrupt is cleared by the URIC0 register.

- Enabling UART reception
  UART receive operation is enabled using the UB0CTL0.UB0RXE bit.

4.8.3 Operating mode

Some values valid for setting as UART operating mode are limited in the sample program.
Set values valid in the sample program are shown below.

Table 4-15. List of UART Set Values

Parameter	Set Value	Remark
Transfer rate	9,600 bps	–
	19,200 bps	Initial setting value
	38,400 bps	–
	57,600 bps	–
	115,200 bps	–
	Other than above	Set to 9,600 bps.
Parity bit	None	Initial setting value
	Odd number	–
	Even number	–
	Space	–
	Other than above	Set as a space.
Data length	7 bits	–
	8 bits	Initial setting value
	Other than above	Set to 8 bits.
Stop bit	1 bit	Initial setting value
	2 bits	
	Other than above	Set to 2 bits.

4.8.4 Description of functions

(1) Overview

A list of the UART processing modules in the sample program is shown below.

Caution Functions starting with "usbf850" are used by the USB function controller incorporated in the V850E/ME2. Functions starting with "uartb0850" are used by UARTB0 incorporated in the V850E/ME2. Refer to 4.7.6 Description of functions for details of the USB function controller functions.

Table 4-16. List of UART Processing Modules

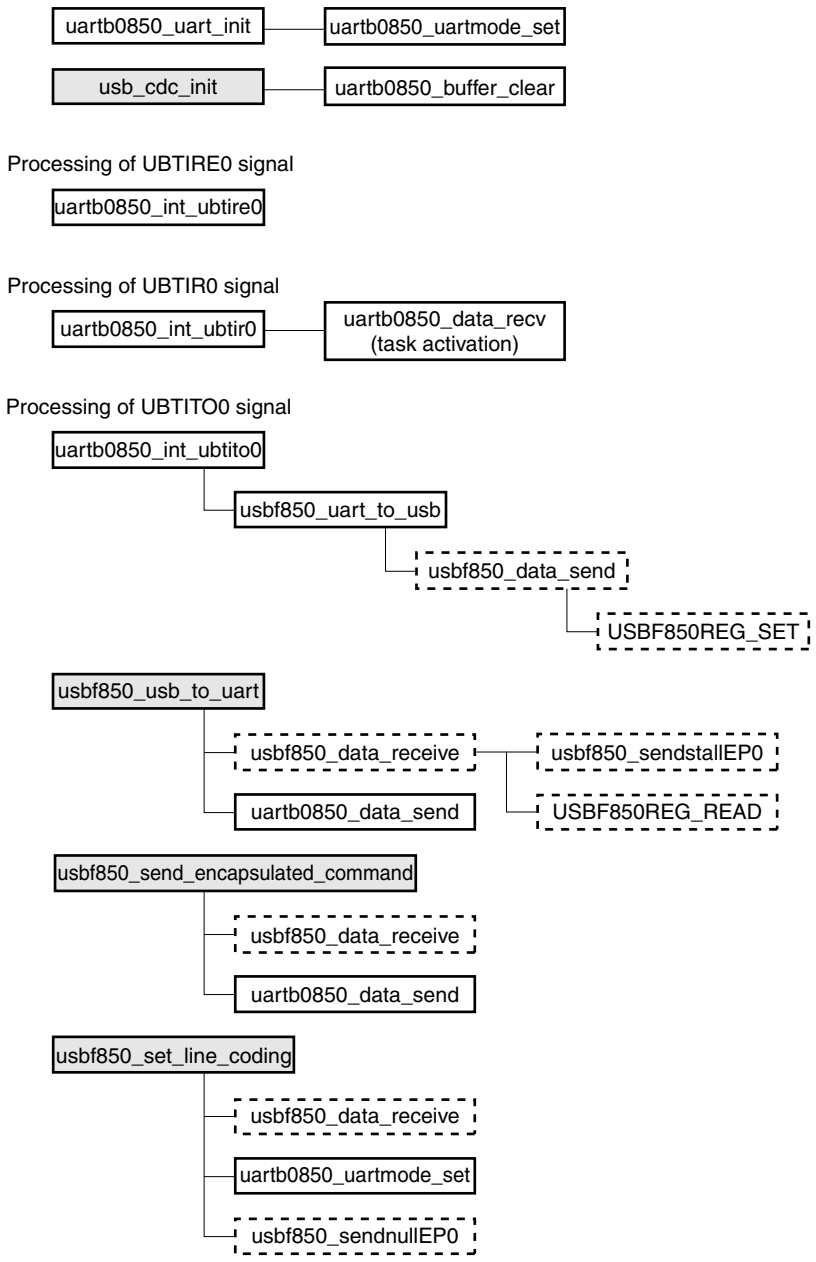
Processing Module Name	Function Name	File Name	Remark
UART processing module			
UART initialization function	uartb0850_enable	uart_ctrl.c	C language
UART setting initialization function	uartb0850_uart_init	uart_ctrl.c	C language
Interrupt servicing task (for UBTIR0 signal processing)	uartb0850_data_recv	uart_ctrl.c	C language
Data transmission function	uartb0850_data_send	uart_ctrl.c	C language
UART FIFO clear function	uartb0850_buffer_clear	uart_ctrl.c	C language
UART mode setting function	uartb0850_uartmode_set	uart_ctrl.c	C language
Interrupt handler (UBTIRE0 signal: UARTB0 reception error)	uartb0850_int_ubtire0	uart_ctrl.c	C language
Interrupt handler (UBTIR0 signal: UARTB0 reception end)	uartb0850_int_ubtir0	uart_ctrl.c	C language
Interrupt handler (UBTITO0 signal: UARTB0 reception timeout)	uartb0850_int_ubtito0	uart_ctrl.c	C language
UART header file	–	uart_ctrl.h	–

(2) Function tree

The calling relationship between the UART processing modules (function tree) in the sample program is illustrated below.

Remark Refer to **4.7.6 (2) Function tree** for the calling relationship in the USB communication class driver processing-dependent module.

Figure 4-18. Sample Program Function Tree of UART Processing Module



Caution ***uartb0850_enable*** **and** ***usb_cdc_init*** **are called from the initialization handler.**

Remark The shaded portions in this figure indicate the USB-UART interface functions described. The portions enclosed by the dashed lines indicate the communication class driver processing-dependent module functions. Refer to **4.7 USB Communication Class Driver Functions** for details of the USB communication class driver processing-dependent module functions and USB-UART interface functions.

(3) Description of functions

The functions of the UART processing module are explained in the same format as described in **4.7.6 (3) Description of functions**.

uartb0850_enable	Valid caller: Non-task I Task

[Outline]

This is a function that initializes UART.

[C language format]

```
void uartb0850_enable (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets ports and enables interrupts as UART initialization processing.

[Return value]

None

uartb0850_uart_init	Valid caller: Non-task | Task

[Outline]

This is a function that initializes the UART settings.

[C language format]

```
void uartb0850_uart_init (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets UART with the initial setting value shown below.

- Transfer rate: 19,200 bps
- Data bit: 8 bits
- Parity: None
- Stop bit: 1 bit

[Return value]

None

uartb0850_data_recv	Valid caller: –

[Outline]

This is a function that handles interrupt servicing by the UART UBTIR0 signal.

[C language format]

```
void uartb0850_data_recv (VP exinf)
```

[Parameter]

I/O	Parameter	Description
I	VP exinf	Extended information

This is the area for storing information specifically defined by the user for the target task, so the user can freely use this area.

Information set to *exinf* can be acquired dynamically by issuing the *ref_tsk* system call from the processing module (task or non-task).

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

[Operation]

This task is activated by the interrupt handler for the UBTIR0 interrupt signal. In the sample program, this task reads data if data exists in the receive FIFO and calls the USB data transmission function (usbf850_data_send). If there is no data in the receive FIFO, this task performs nothing and ends normally.

Remark Refer to **4.8.2 (1) interrupt servicing** for details of interrupt servicing.

[Return value]

None

uartb0850_data_send Valid caller: Non-task | Task

[Outline]

This is a function that transmits UART data.

[C language format]

```
void uartb0850_data_send (unsigned char* buffer, int size)
```

[Parameter]

I/O	Parameter	Description
I	unsigned char* buffer	Start address of transmit data
I	int size	Data size

[Operation]

This function transmits data whose size is specified by *size* starting from the address specified by *buffer* to UART.

[Return value]

None

uartb0850_buffer_clear Valid caller: Non-task | Task

[Outline]

This is a function that clears the UART transmit/receive buffer.

[C language format]

```
void uartb0850_buffer_clear (char* buffer, int size)
```

[Parameter]

I/O	Parameter	Description
I	char* buffer	Start address of the buffer
I	int size	Size of the buffer to be cleared

[Operation]

This function clears data whose size is specified by *size* starting from the address specified by *buffer* in the transmit/receive data buffer.

[Return value]

None

uartb0850_uartmode_set	Valid caller: Non-task I Task

[Outline]

This is a function that sets the UART operation mode.

[C language format]

```
void uartb0850_cdc_uartmode_set (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This function sets UART operating mode based on the current set values stored in the UART_MODE_INFO structure. The set values of the transfer rate, parity bit, data length, and stop bit. An initial setting value described in **Table 4-15 List of UART Set Values** is set to the UART_MODE_INFO structure. This value is changed by the host using the SET LINE CODING request. This function is called when this request is received and changes the UART operating mode.

Remarks 1. Refer to **4.7.5 (2) UART mode table structure** for details of the UART_MODE_INFO structure.

2. Refer to **4.8.3 Operating mode** for the UART set value valid in the sample program.

[Return value]

None

uartb0850_int_ubtire0	Valid caller: –

[Outline]

This is an interrupt handler (for the UBTIRE0 signal) used by UART incorporated in the V850E/ME2.

[C language format]

```
ID uartb0850_int_ubtire0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the UBTIRE0 signal (UARTB0 reception error interrupt) and performs error processing. This handler is defined in the CF definition file.

Remark Refer to **4.8.2 (1) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

uartb0850_int_ubtir0	Valid caller: –

[Outline]

This is an interrupt handler (for the UBTIR0 signal) used by UART incorporated in the V850E/ME2.

[C language format]

```
ID uartb0850_int_ubtir0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the UBTIR0 signal (UARTB0 reception end interrupt). In this sample program, it activates the interrupt servicing task (usbf850_cdc_data_recv). This handler is defined in the CF definition file.

Remark Refer to **4.8.2 (1) Interrupt servicing** for details of interrupt servicing.

[Return value]

Object ID number (task ID number)

uartb0850_int_ubtito0	Valid caller: –

[Outline]

This is an interrupt handler (for the UBTITO0 signal) used by UART incorporated in the V850E/ME2.

[C language format]

```
ID uartb0850_int_ubtito0 (void)
```

[Parameter]

I/O	Parameter	Description
–	–	–

[Operation]

This interrupt handler is activated by the UBTITO0 signal (UARTB0 reception timeout interrupt) and performs reception timeout processing. This handler is defined in the CF definition file.

Remark Refer to **4.8.2 (1) Interrupt servicing** for details of interrupt servicing.

[Return value]

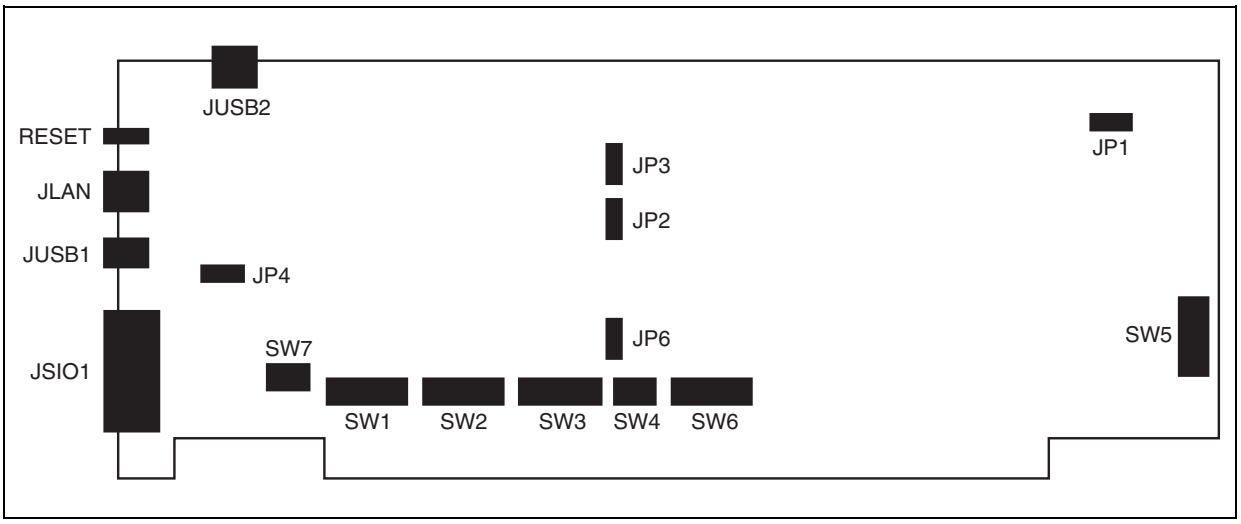
Object ID number (task ID number)

APPENDIX A SG-703111-1 BOARD

A.1 Overview

The sample programs of the USB function drivers operate on the SG-703111-1 board.
This section explains how to set up the SG-703111-1 board.

Figure A-1. Configuration of SG-703111-1



A.2 Setting of DIP Switches (SW1 to SW7)

There are seven DIP switches (SW1 to SW7) on the SG-703111-1 board.
An example of the DIP switch settings is shown below.

Table A-1. Setting of DIP Switches (SW1 to SW7) (1/2)

Switch Name	Setting	Remark
SW1	1: OFF	Settings related to the monitor ROM (setting not required).
	2: ON	
	3: ON	
	4: ON	
	5: OFF	
	6: OFF	
	7: OFF	
	8: OFF	
SW2	1: OFF	Activates the monitor (setting not required).
	2: OFF	Always use at OFF.
	3: OFF	Sets to use NMI.
	4: ON	Sets $\overline{CS0}$ to $\overline{CS2}$ to uncached.
	5: ON	Sets the SDRAM bus size to 32 bits.
	6: OFF	Always use at OFF.
	7: ON	Start the user program.
	8: ON	
SW3	1: ON	Normally use the factory settings.
	2: ON	
	3: ON	
	4: OFF	
	5: OFF	
	6: OFF	
	7: OFF	
	8: OFF	
SW4	1: ON	Normally use the factory settings.
	2: ON	
	3: OFF	
	4: ON	

Remark Refer to the **SG-703111-1 User's Manual** for details of the DIP switch settings (SW1 to SW7).

Table A-1. Setting of DIP Switches (SW1 to SW7) (2/2)

Switch Name	Setting	Remark
SW5	1: OFF	Set the bus size at activation to 16 bits.
	2: ON	
	3: OFF	Sets the input level of the SSEL0 pin of the V850E/ME2 to high.
	4: ON	Sets the input level of the SSEL1 pin of the V850E/ME2 to low.
	5: OFF	Sets the input level of the JIT0 pin of the V850E/ME2 to high.
	6: OFF	Sets the input level of the JIT1 pin of the V850E/ME2 to high.
	7: ON	Sets the input level of the PLLSEL pin of the V850E/ME2 to low.
	8: OFF	Always use at OFF.
SW6	1: OFF	Normally use the factory settings.
	2: OFF	
	3: OFF	
	4: OFF	
	5: ON	
	6: ON	
	7: OFF	
	8: OFF	
SW7	1: ON	Do not change the factory settings.
	2: ON	
	3: ON	
	4: OFF	

Remark Refer to the **SG-703111-1 User's Manual** for details of the DIP switch settings (SW1 to SW7).

A.3 Setting of Jumper Switches (JP1 to JP4, JP6)

There are five jumper switches (JP1 to JP4 and JP6) on the SG-703111-1 board.
An example of the jumper switch settings is shown below.

Table A-2. Setting of Jumper Switches (JP1 to JP4 and JP6)

Switch Name	Setting	Remark
JP1	1-2: Shorted	Supplies the power (AV_{DD}) to the A/D converter of the V850E/ME2 from the board.
JP2	1-2: Shorted	Factory setting
JP3	1-2: Open	Factory setting
JP4	1-2: Open	Factory setting
JP6	1-2: Shorted	Factory setting

Remark Refer to the **SG-703111-1 User's Manual** for details of the jumper switch settings (JP1 to JP4 and JP6).

A.4 File for Initializing Board at In-Circuit Emulator Startup

Since the execution file is written to the memory on the target board using the in-circuit emulator, it is required to initialize the target board (particularly the registers that control the memory) before program execution.

Prepare the automatic execution file (init. mcr) that automatically reads and executes the initialization program when the in-circuit emulator is started up. This file must be placed in the same directory as the one where the project file of the in-circuit emulator is stored.

An example of init.mcr for the V850E/ME2 is shown below.

Remark Refer to the **PARTNER User's Manual V800 Series Common Edition** and **NB85E-TP Part Edition** for details of the file format.

```
reset
_PC=0x00100000
POW 0xffff060,0x000f            * CSC0
POW 0xffff062,0x000f            * CSC1
POB 0xffff06E,0x33              * VSWC
POW 0xffff480,0x88b8            * BCT0
POW 0xffff482,0x8888            * BCT1
POW 0xffff484,0x1116            * DWC0
POW 0xffff486,0x1111            * DWC1
POW 0xffff488,0x0002            * BCC
POW 0xffff48A,0x0000            * ASC
POW 0xffff48e,0x6aa9            * LBS
POB 0xffff06E,0x37              * VSWC
POB 0xffff498,0x02              * BMC
POB 0xffff06E,0x33              * VSWC

POB 0xffff6c0,0x00              * OSTS

POW 0xffff4A4,0x20a5            *SCR1
POW 0xffff4A6,0x8203            *RFS1

POB 0xffff1fc,0xff              * PRCMD
POB 0xffff822,0x03              * CKC
POB 0xffff1fc,0xff              * PRCMD
POB 0xffff82c,0x01              * CKS

POW 0xffff056,0x01              * PFCDH *0x00
POW 0xffff040,0x0003            * PMCAL
POB 0xffff80a,0x00              * IRAMM

POB 0xffff04b,0x0f              * PFCCT
```

APPENDIX B FUNCTION INDEX

(1/4)

Function Name	Driver Name	Page
ata_inquiry	USB storage class driver	181
ata_mode_select	USB storage class driver	182
ata_mode_select10	USB storage class driver	183
ata_mode_sense	USB storage class driver	184
ata_mode_sense10	USB storage class driver	185
ata_read_capacity	USB storage class driver	187
ata_read_format_capacities	USB storage class driver	186
ata_read10	USB storage class driver	189
ata_read6	USB storage class driver	188
ata_request_sense	USB storage class driver	180
ata_seek	USB storage class driver	177
ata_start_stop_unit	USB storage class driver	178
ata_synchronize_cache	USB storage class driver	179
ata_test_unit_ready	USB storage class driver	176
ata_verify	USB storage class driver	192
ata_write_buff	USB storage class driver	194
ata_write_verify	USB storage class driver	193
ata_write10	USB storage class driver	191
ata_write6	USB storage class driver	190
scsi_command_to_ata	USB storage class driver	175
scsi_to_usb	USB storage class driver	195
storageDev_Init	USB storage class driver	174
task_usb0b	USB bus driver	63
	USB storage class driver	146
	USB communication class driver	251
task_usb1b	USB bus driver	64
	USB storage class driver	147
	USB communication class driver	252
task_usb2b	USB bus driver	65
	USB storage class driver	148
	USB communication class driver	253
uartb0850_buffer_clear	USB communication class driver	291
uartb0850_cdc_uartmode_set	USB communication class driver	292
uartb0850_data_recv	USB communication class driver	289
uartb0850_data_send	USB communication class driver	290

(2/4)

(3/4)

Function Name	Driver Name	Page
usbf850_inthdr2	USB bus driver	62
	USB storage class driver	145
	USB communication class driver	250
usbf850_loc_cpu	USB bus driver	72
	USB storage class driver	155
	USB communication class driver	260
usbf850_max_lun	USB storage class driver	163
usbf850_no_data	USB storage class driver	168
usbf850_rx_cbw	USB storage class driver	165
usbf850_rxreq	USB bus driver	74
	USB storage class driver	157
	USB communication class driver	262
usbf850_rxreq_read	USB bus driver	75
	USB storage class driver	158
	USB communication class driver	263
usbf850_send_encapsulated_command	USB communication class driver	267
usbf850_sendnullEP0	USB bus driver	68
	USB storage class driver	151
	USB communication class driver	256
usbf850_sendstallEP0	USB bus driver	69
	USB storage class driver	152
	USB communication class driver	257
usbf850_set_control_line_state	USB communication class driver	271
usbf850_set_line_coding	USB communication class driver	269
usbf850_setfunction_communication	USB communication class driver	274
usbf850_setfunction_storage	USB storage class driver	164
usbf850_sstall_ctrl	USB bus driver	78
	USB storage class driver	161
	USB communication class driver	266
usbf850_standardreq	USB bus driver	76
	USB storage class driver	159
	USB communication class driver	264
usbf850_storage_cbwchk	USB storage class driver	166
usbf850_uart_to_usb	USB communication class driver	273
usbf850_unl_cpu	USB bus driver	73
	USB storage class driver	156
	USB communication class driver	261
usbf850_usb_to_uart	USB communication class driver	272

(4/4)