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.

RENESAS

Application Note

V850ES/JG3-H, V850ES/JH3-H
V850ES/JG3-U, V850ES/JH3-U

32-bit Single-Chip Microcontroller

USB MSC (Mass Storage Class) Driver

V850ES/JG3-H	V850ES/JH3-H
μPD70F3760	μPD70F3765
μPD70F3761	μPD70F3766
μPD70F3762	μPD70F3767
μPD70F3770	μPD70F3771

V850ES/JG3-U	V850ES/JH3-U
μPD70F3763	μPD70F3768
μPD70F3764	μPD70F3769

Document No. U19659EJ1V0AN00 (1st edition)
Date Published March 2009 N

[MEMO]

PREFACE

Readers This application note is intended for users who understand the features of the V850ES/Jx3-H or V850ES/Jx3-U, and are going to develop application systems using this product.

Purpose This application note is intended to give users an understanding of the specifications of the sample driver provided for using the USB function controller incorporated in the V850ES/Jx3-H and V850ES/Jx3-U.

Organization This application note is broadly divided into the following five sections:

- An overview of the USB function controller incorporated in the V850ES/Jx3-H and V850ES/Jx3-U
- An overview of the USB standard
- The specifications for the sample driver
- Development environment
- How to use the sample driver

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 learn about the hardware features and electrical specifications of the V850ES/Jx3-H and V850ES/Jx3-U
  → See the separately provided **V850ES/JG3-H, V850ES/JH3-H Hardware User's Manual** and **V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual**.

- To learn about the instructions of the V850ES/Jx3-H and V850ES/Jx3-U
  → See the separately provided **V850ES Architecture User's Manual**.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Binary or decimal ... XXXX
	Hexadecimal ... 0xXXXX

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} = $1,024^3$
T (tera): 2^{40} = $1,024^4$
P (peta): 2^{50} = $1,024^5$
E (exa): 2^{60} = $1,024^6$

Related Documents

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

• Documents Related to the V850ES/Jx3-H and V850ES/Jx3-U

Document Name	Document No.
V850ES Architecture User's Manual	U15943E
V850ES/JG3-H, V850ES/JH3-H Hardware User's Manual	U19181E
V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual	U19287E
V850ES/JG3-H, V850ES/JH3-H, V850ES/JG3-U, V850ES/JH3-U USB MSC (Mass Storage Class) Driver Application Note	This manual

• Documents Related to Development Tools (User's Manuals)

Document Name		Document No.
QB-V850ESJX3H In-Circuit Emulator		U19170E
QB-V850MINI On-Chip Debug Emulator		U17638E
QB-MINI2 On-Chip Debug Emulator with Flash Programming Function		U18371E
CA850 Ver. 3.20 C Compiler Package	Operation	U18512E
	C Language	U18513E
	Assembly Language	U18514E
	Link Directives	U18515E
PM+ Ver. 6.30 Project Manager		U18416E
ID850QB Ver. 3.40 Integrated Debugger	Operation	U18604E
SM850 Ver. 2.50 System Simulator	Operation	U16218E
SM850 Ver. 2.00 or Later System Simulator	External Component User Open Interface	U14873E
SM+ System Simulator	Operation	U17199E
	User Open Interface	U17198E
	Basics	U13430E
	Installation	U17419E
	Technical	U13431E
RX850 Ver. 3.20 Real-Time OS	Task Debugger	U17420E
	Basics	U18165E
	In-Structure	U18164E
RX850 Pro Ver. 3.21 Real-Time OS	Task Debugger	U17422E
AZ850 Ver. 3.30 System Performance Analyzer		U17423E
PG-FP5 Flash Memory Programmer		U18865E

CONTENTS

CHAPTER 1 OVERVIEW 8
1.1 Overview 8
1.1.1 Features of the USB function controller 8
1.1.2 Features of the sample driver 9
1.1.3 Files included in the sample driver 9
1.2 Overview of the V850ES/Jx3-H and V850ES/Jx3-U 10
1.2.1 Applicable products 10
1.2.2 Features 11

CHAPTER 2 OVERVIEW OF USB 12
2.1 Transfer Format 12
2.2 Endpoints 13
2.3 Class 13
2.3.1 Mass storage class (MSC) 13
2.3.2 Subclass 16
2.4 Requests 17
2.4.1 Types 17
2.4.2 Format 18
2.5 Descriptor 19
2.5.1 Types 19
2.5.2 Format 20

CHAPTER 3 SAMPLE DRIVER SPECIFICATIONS 22
3.1 Overview 22
3.1.1 Features 22
3.1.2 Supported requests 23
3.1.3 Descriptor settings 25
3.1.4 Supported SCSI commands 28
3.2 Operation of Each Section 40
3.2.1 CPU Initialization 41
3.2.2 USBF initialization processing 43
3.2.3 USBF interrupt servicing (INTUSBF0) 47
3.2.4 USBF resume interrupt servicing (INTUSBF1) 49
3.2.5 CBW data reception processing 50
3.2.6 SCSI command processing 52
3.2.7 Suspend/resume processing 54
3.3 Function Specifications 56
3.3.1 Functions 56
3.3.2 Correlation of the functions 58
3.3.3 Function features 62
3.4 Data Structures 98

CHAPTER 4 DEVELOPMENT ENVIRONMENT 99
4.1 Used Products 99

4.1.1 System components ... 99
4.1.2 Program development ... 100
4.1.3 Debugging ... 100
4.2 Setting Up the Environment ... 101
4.2.1 Preparing the host environment ... 101
4.2.2 Setting up the target environment ... 109
4.3 On-Chip Debugging ... 111
4.3.1 Generating a load module ... 111
4.3.2 Loading and executing the load module ... 112
4.4 Checking the Operation ... 115

CHAPTER 5 USING THE SAMPLE DRIVER ... 120
5.1 Overview ... 120
5.2 Customizing the Sample Driver ... 121
5.2.1 Application section ... 121
5.2.2 Setting up the registers ... 122
5.2.3 Descriptor information ... 122
5.2.4 Changing SCSI command processing ... 123
5.2.5 Changing the RAM disk capacity ... 124
5.2.6 Specifying the vendor and product names ... 125
5.3 Using Functions ... 127

APPENDIX A STARTER KIT ... 128
A.1 Overview ... 128
A.2 TK-850/JG3H ... 128
A.2.1 Features ... 128
A.2.2 Specifications ... 129
A.3 TK-850/JH3U-SP ... 129
A.3.1 Features ... 129
A.3.2 Specifications ... 129

CHAPTER 1 OVERVIEW

This application note describes the MSC (mass storage class) sample driver created for the USB function controller incorporated in the V850ES/Jx3-H and V850ES/Jx3-U microcontrollers.

This application note provides the following information:

- The specifications for the sample driver
- Information about the environment used to develop an application program by using the sample driver
- The reference information provided for using the sample driver

This chapter provides an overview of the sample driver and describes the microcontrollers for which the sample driver can be used.

1.1 Overview

1.1.1 Features of the USB function controller

The USB function controller (USBF) that is incorporated in the V850ES/Jx3-H and V850ES/Jx3-U and is to be controlled by the sample driver has the following features:

- Conforms to the Universal Serial Bus Rev. 2.0 Specification.
- Operates as a full-speed (12 Mbps) device.
- Includes the following endpoints:

Table 1-1. Configuration of the Endpoints of the V850ES/Jx3-H and V850ES/Jx3-U

Endpoint Name	FIFO Size (Bytes)	Transfer Type	Remark
Endpoint 0 Read	64	Control transfer (IN)	–
Endpoint 0 Write	64	Control transfer (OUT)	–
Endpoint 1	64 × 2	Bulk transfer 1 (IN)	Dual-buffer configuration
Endpoint 2	64 × 2	Bulk transfer 1 (OUT)	Dual-buffer configuration
Endpoint 3	64 × 2	Bulk transfer 2 (IN)	Dual-buffer configuration
Endpoint 4	64 × 2	Bulk transfer 2 (OUT)	Dual-buffer configuration
Endpoint 7	8	Interrupt transfer (IN)	–

- Automatically responds to standard USB requests (except some requests).
- Can operate as a bus-powered device or self-powered device[Note 1]
- The internal or external clock can be selected[Note 2]
  Internal clock: 6 MHz external clock multiplied by 8 (48 MHz)
  External clock: Input to the UCLK pin (f_{USB} = 48 MHz))

Notes 1. The sample driver selects bus power.
2. The sample driver selects the internal clock.

1.1.2 Features of the sample driver

The MSC sample driver for the V850ES/Jx3-H and V850ES/Jx3-U has the features below. For details about the features and operations, see **CHAPTER 3 SAMPLE DRIVER SPECIFICATIONS**.

- Operates as a bus-powered device.
- Recognized as a mass storage class bulk-only device when connected to a host.
- Can be formatted to any file system format from a host.
- Data such as files and folders can be written to internal RAM.
- Files and folders written to internal RAM can be read.
- Exclusively uses the following amounts of memory (excluding the vector table):
  ROM: About 7.9 KB
  RAM: About 25.9 KB[Note]

Note 24 KB in RAM is used as a data storage area. Therefore, data stored in the area is initialized when the device is turned off or the reset switch is pressed.

1.1.3 Files included in the sample driver

The sample driver includes the following files:

Table 1-2. Files Included in the Sample Driver

Folder	File	Overview
src	`main.c`	Main routine
	`scsi_cmd.c`	SCSI command processing
	`usbf850.c`	USB initialization, endpoint control, bulk transfer, control transfer
	`usbf850_storage.c`	MSC-specific processing
include	`errno.h`	Error code definitions
	`main.h`	`main.c` function prototype declarations
	`RegDef.h`	Register definitions
	`scsi.h`	SCSI macro definitions
	`Types.h`	User-defined type declarations
	`usbf850.h`	`usbf850.c` function prototype declarations
	`usbf850_storage.h`	`usbf850_storage.c` function prototype declarations
	`usbstrg_desc.h`	Descriptor definitions
	`usbf850_sfr.h`	Macro definitions for accessing the USBF registers

Remark In addition, the project-related files generated when creating a development environment by using the USB-to-serial conversion host driver or PM+ (an integrated development tool made by NEC Electronics) are also included. For details, see **4.2.1 Preparing the host environment**.

1.2 Overview of the V850ES/Jx3-H and V850ES/Jx3-U

This section describes the V850ES/Jx3-H and V850ES/Jx3-U, which are controlled by using the sample driver.

The V850ES/Jx3-H and V850ES/Jx3-U are products in the low-power series of V850 single-chip microcontrollers for real-time control, made by NEC Electronics. They use a V850 CPU core and have peripherals such as ROM, RAM, timers, counters, a serial interface, an A/D converter, a D/A converter, a DMA controller, a CAN controller, and a USB function controller. For details, see the **V850ES/JG3-H, V850ES/JH3-H Hardware User's Manual** and **V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual**.

1.2.1 Applicable products

The sample driver can be used for the following products:

Table 1-3. V850ES/Jx3-H and V850ES/Jx3-U Products

Generic Name	Part Number	Internal Memory		Incorporated USB Function	Interrupt	
		Flash Memory	RAM Note 1		Internal Note 2	External Note 2
V850ES/JG3-H	μPD70F3760	256 KB	40 KB	Function controller	69	17
	μPD70F3761	384 KB	48 KB	Function controller		
	μPD70F3762	512 KB	56 KB	Function controller		
	μPD70F3770	256 KB	40 KB	Function controller	73	
V850ES/JH3-H	μPD70F3765	256 KB	40 KB	Function controller	69	20
	μPD70F3766	384 KB	48 KB	Function controller		
	μPD70F3767	512 KB	56 KB	Function controller		
	μPD70F3771	256 KB	40 KB	Function controller	73	
V850ES/JG3-U	μPD70F3763	384 KB	48 KB	Function controller Host controller	72	15
	μPD70F3764	512 KB	56 KB	Function controller Host controller		
V850ES/JH3-U	μPD70F3768	384 KB	48 KB	Function controller Host controller		20
	μPD70F3769	512 KB	56 KB	Function controller Host controller		

Notes 1. Includes a data-dedicated 8 KB RAM area.

2. Includes one non-maskable interrupt source.

Caution In this application note, all target microcontrollers are collectively indicated as the V850ES/Jx3-H, unless distinguishing between them is necessary.

1.2.2 Features

The main features of the V850ES/Jx3-H and V850ES/Jx3-U are as follows.

- Memory space: 64 MB of linear address space (for programs and data)
  External expansion: Up to 16 MB (including 1 MB used as internal ROM/RAM space)
- Internal memory: RAM: 40/48/56 KB
  Flash memory: 256/384/512 KB
- External bus interface: Multiplexed bus output (V850ES/JG3-H, V850ES/JG3-U)
  Separate bus/multiplexed bus output selectable (V850ES/JH3-H, V850ES/JH3-U)
  8/16-bit data bus sizing
  Wait function
  - Programmable wait
  - External wait

  Idle state
  Bus hold
- Serial interface: Asynchronous serial interface C (UARTC): 5 shared channels
  Three-wire variable-length serial interface F (CSIF): 2 dedicated channels and 3 shared channels
  I^2C bus interface (I^2C): 3 shared channels
  CAN interface: 1 shared channel (μPD70F3770 and μPD70F3771)
  USB function interface: 1 dedicated channel
  USB host interface: 1 dedicated channel (V850ES/Jx3-U)
- DMA controller: 4 channels
- Clock generator: Main clock or subclock operation:
  Seven-level CPU clock (f_{XX}, $f_{XX}/2$, $f_{XX}/4$, $f_{XX}/8$, $f_{XX}/16$, $f_{XX}/32$, f_{XT})
  Clock-through mode/PLL mode selectable

CHAPTER 2 OVERVIEW OF USB

This chapter provides an overview of the USB standard, which the sample driver conforms to.

USB (Universal Serial Bus) is an interface standard for connecting various peripherals to a host by using the same type of connector. The USB interface is more flexible and easier to use than older interfaces in that it can connect up to 127 devices by adding a branching point known as a hub, and supports the hot-plug feature, which enables devices to be recognized by Plug & Play. The USB interface is provided in most current computers and has become the standard for connecting peripherals to a computer.

The USB standard is formulated and managed by the USB Implementers Forum (USB-IF). For details about the USB standard, see the official USB-IF website (www.usb.org).

2.1 Transfer Format

Four types of transfer formats (control, bulk, interrupt, and isochronous) are defined in the USB standard. Table 2-1 shows the features of each transfer format.

Table 2-1. USB Transfer Format

Item \ Transfer Format		Control Transfer	Bulk Transfer	Interrupt Transfer	Isochronous Transfer
Feature		Transfer format used to exchange information required for controlling peripheral devices	Transfer format used to periodically handle large amounts of data	Periodic data transfer format that has a low band width	Transfer format used for a real-time transfer
Specifiable packet size	High speed 480 Mbps	64 bytes	512 bytes	1 to 1,024 bytes	1 to 1,024 bytes
	Full speed 12 Mbps	8, 16, 32, or 64 bytes	8, 16, 32, or 64 bytes	1 to 64 bytes	1 to 1,023 bytes
	Low speed 1.5 Mbps	8 bytes	–	1 to 8 bytes	–
Transfer priority		3	3	2	1

2.2 Endpoints

An endpoint is an information unit that is used by the host to specify a communicating device and is specified using a number from 0 to 15 and a direction (IN or OUT). An endpoint must be provided for every data communication path that is used for a peripheral device and cannot be shared by multiple communication paths[Note]. For example, a device that can write to and read from an SD card and print out documents must have a separate endpoint for each purpose. Endpoint 0 is used to control transfers for any type of device.

During data communication, the host uses a USB device address, which specifies the device, and an endpoint (a number and direction) to specify the communication destination in the device.

Peripheral devices have buffer memory that is a physical circuit to be used for the endpoint and functions as a FIFO that absorbs the difference in speed of the USB and communication destination (such as memory).

Note An endpoint can be exclusively switched by using the alternative setting.

2.3 Class

Various classes, such as the mass storage class (MSC), communication device class (CDC), printer class, and human interface device class (HID), are defined according to the functions of the peripheral devices connected via USB (the function devices). A common host driver can be used if the connected devices conform to the standard specifications of the relevant device class, which is defined by a protocol. A separate driver is not necessary for each device, enabling users to connect any device and vendors to save labor hours for developing application programs.

2.3.1 Mass storage class (MSC)

The mass storage class (MSC) is an interface class used to recognize and control memory devices such as flash memories, hard disk drives, and optical disk storage devices that are connected via USB.

Communication using the MSC is performed using the bulk-only transfer protocol or CBI (control/bulk/interrupt) transfer protocol. The bulk-only transfer protocol uses only bulk transfer to transfer data. The CBI transfer protocol uses control and interrupt transfers in addition to bulk transfer and is used only for full-speed floppy disk drives.

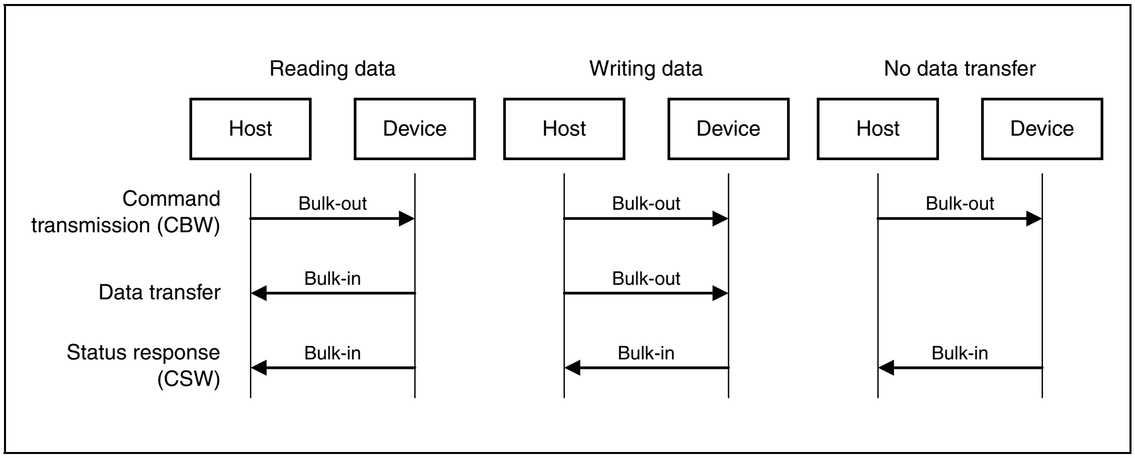
The sample driver uses the mass storage class (MSC) bulk-only transfer protocol.

For details about the USB mass storage class (MSC) specifications, see the MSC standard specification **Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0.**

(1) Data transfer

The bulk-only transfer protocol transfers commands, statuses, and data by using bulk transfer.
The host uses bulk-out transfer to transmit commands to a device.
If a command that involves data transfer is transmitted, data is input or output using bulk-in or bulk-out transfer.
The device uses bulk-in transfer to transmit the status (command execution result) to the host.

Figure 2-1. Data Transfer Flowchart



(2) CBW format

The packet structure when a command is transmitted is defined as a command block wrapper (CBW).

Bytes \ Bit	7	6	5	4	3	2	1	0
0 to 3	dCBWSignature							
4 to 7	dCBWTag							
8 to 11	dCBWDataTransferLength							
12	bmCBWFlags							
13	Reserved				bCBWLUN			
14	Reserved			bCBWCBLength				
15 to 30	CBWCB							

dCBWSignature: Signature. Fixed to 0x43425355 (little endian)

dCBWTag: Tag whose number is defined by the host and that matches a command to a status

dCBWDataTransferLength: Length of the data transferred during the data phase. This is 0 if there is no data.

bmCBWFlags: Transfer direction (bit 7). 0 = bulk-out, 1 = bulk-in. Bits 0 to 6 are fixed to 0.

bCBWLUN: If multiple drives are connected to one USB device, the numbers of those drives are specified.

bCBWCBLength: Command packet length

CBWCB: Command packet data

(3) CSW format

The packet structure when a status is transmitted is defined as a command status wrapper (CSW).

Bytes \ Bit	7	6	5	4	3	2	1	0
0 to 3	dCSWSignature							
4 to 7	dCSWTag							
8 to 11	dCSWDataResidue							
12	bCSWStatus							

dCSWSignature: Signature. Fixed to 0x53425355 (little endian)

dCSWTag: By matching this to dCBWTag when transferring a command, the host checks for a match in the phase.

dCSWDataResidue: Remaining data. If the data returned by the host is shorter than the data requested by the host, due to causes such as when an error occurred during data transfer, the remaining amount of data is set up here. Therefore, even if the status (bCSWStatus) indicates that the CBW processing was successful, if a value other than 0 is specified here, it means that the data returned from the device was short.

dCSWStatus: CBW processing result status

dCSWStatus	Description
0x00	Successful
0x01	Failed
0x02	Phase error
0x03 to 0xFF	Reserved

2.3.2 Subclass

For the mass storage class (MSC), specify the format in which commands are transmitted from the host to the target device as the subclass.

(1) Subclass types

Table 2-2 shows the subclass codes that are specified for the USB mass storage class.

Table 2-2. Subclass Codes for the USB Mass Storage Class

Subclass Code	Standard
0x00	SCSI command set not reported (normally not used)
0x01	Reduced block commands (RBC), T10 Project 1240-D
0x02	MMC-5 (ATAPI)
0x03	SFF-8070i
0x04	USB floppy interface (UFI)
0x05	QIC-157 (IDE QIC tape drive)
0x06	SCSI transparent command set
0x07	Lockable mass storage
0x08	IEEE1667
0x09 to 0xFE	Reserved
0xFF	Specific to device vendor

(2) SCSI command

To connect a USB memory or USB card reader, specify the SCSI transfer command set (0x06) as the subclass. SCSI (Small Computer System Interface) is an interface standard for connecting peripherals to a computer by using bus lines.

Transfer data and set up functions by specifying a SCSI command by using CBWCB (command packet data) of the CBW. For the SCSI commands supported by the sample driver, see **3.1.4 Supported SCSI commands**.

2.4 Requests

For the USB standard, communication starts with the host issuing a command, known as a request, to all function devices. A request includes data such as the direction and type of processing and address of the function device. Each function device decodes the request, judges whether the request is addressed to it, and responds only if the request is addressed to it.

2.4.1 Types

There are three types of requests: standard requests, class requests, and vendor requests.

For details about requests that the sample driver supports, see **3.1.2 Supported requests**.

(1) Standard requests

Standard requests are used for all USB-compatible devices. A request is a standard request if the values of bits 6 and 5 in the bmRequestType field are both 0. For details about the processing of standard requests, see the **Universal Serial Bus Specification Rev. 2.0**.

Table 2-3. Standard Requests

Request Name	Target Descriptor	Overview
GET_STATUS	Device	Reads the settings of the power supply (self or bus) and remote wakeup.
	Endpoint	Reads the halt status.
CLEAR_FEATURE	Device	Clears remote wakeup.
	Endpoint	Cancels the halt status (DATA PID = 0).
SET_FEATURE	Device	Specifies remote wakeup or test mode.
	Endpoint	Specifies the halt status.
GET_DESCRIPTOR	Device	Reads the target descriptor.
	Configuration	
	String	
SET_DESCRIPTOR	Device	Changes the target descriptor (optional).
	Configuration	
	String	
GET_CONFIGURATION	Device	Reads the currently specified configuration values.
SET_CONFIGURATION	Device	Specifies the configuration values.
GET_INTERFACE	Interface	Reads the alternatively specified value among the currently specified values of the target interface.
SET_INTERFACE	Interface	Specifies the alternatively specified value of the target interface.
SET_ADDRESS	Device	Specifies the USB address.
SYNCH_FRAME	Endpoint	Reads frame-synchronous data.

(2) Class requests

Class requests are unique to classes. A request is a class request if the values of bits 6 and 5 in the bmRequestType field are 0 and 1, respectively.

The mass storage class (MSC) bulk-only transfer protocol must support the following requests:

- GET_MAX_LUN (bRequest = 0xFE)
  This request is used to acquire the logical unit number of the MSC device.
- MASS_STORAGE_RESET (bRequest = 0xFF)
  This request is used to reset the interfaces related to the MSC device.

(3) Vendor requests

Vendor requests are requests that are uniquely defined by each vendor. To make vendor requests available for use, the vendor must provide a host driver that supports the requests. A request is a vendor request if bits 6 and 5 in the bmRequestType field are 1 and 0, respectively.

2.4.2 Format

USB requests have an 8-byte length and consist of the following fields:

Table 2-4. USB Request Format

Offset	Field		Description
0	bmRequestType		Request attribute
		Bit 7	Data transfer direction
		Bits 6 and 5	Request type
		Bits 4 to 0	Target descriptor
1	bRequest		Request code
2	wValue	Lower	Any value used by the request
3		Higher	
4	wIndex	Lower	Index or offset used by the request
5		Higher	
6	wLength	Lower	Number of bytes transferred at the data stage (the data length)
7		Higher	

2.5 Descriptor

For the USB standard, a descriptor is information that is specific to a function device and is encoded in a specified format. A function device transmits a descriptor in response to a request transmitted from the host.

2.5.1 Types

The following five types of descriptors are defined:

- Device descriptor
  This descriptor exists in every device and includes basic information such as the supported USB specification version, device class, protocol, maximum packet length that can be used when transferring data to endpoint 0, vendor ID, and product ID.
  This descriptor is transmitted in response to a GET_DESCRIPTOR_Device request.
- Configuration descriptor
  At least one configuration descriptor exists in every device and includes information such as the device attribute (power supply method) and power consumption.
  This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.
- Interface descriptor
  This descriptor is required for each interface and includes information such as the interface identification number, interface class, and supported number of endpoints.
  This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.
- Endpoint descriptor
  This descriptor is required for each endpoint specified for an interface descriptor and defines the transfer type (direction), maximum packet length that can be used for a transfer, and transfer interval. However, endpoint 0 does not have this descriptor.
  This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.
- String descriptor
  This descriptor includes any character string.
  This descriptor is transmitted in response to a GET_DESCRIPTOR_String request.

2.5.2 Format

The size and fields of each descriptor type vary as described below.

Remark The data sequence of each field is in little endian format.

Table 2-5. Device Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bcdUSB	2	USB specification release number
bDeviceClass	1	Class code
bDeviceSubClass	1	Subclass code
bDeviceProtocol	1	Protocol code
bMaxPacketSize0	1	Maximum packet size of endpoint 0
idVendor	2	Vendor ID
idProduct	2	Product ID
bcdDevice	2	Device release number
iManufacturer	1	Index to the string descriptor representing the manufacturer
iProduct	1	Index to the string descriptor representing the product
iSerialNumber	1	Index to the string descriptor representing the device production number
bNumConfigurations	1	Number of configurations

Remark Vendor ID: The identification number each company that develops a USB device acquires from USB-IF
Product ID: The identification number each company assigns to a product after acquiring the vendor ID

Table 2-6. Configuration Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
wTotalLength	2	Total number of bytes of the configuration, interface, and endpoint descriptors
bNumInterfaces	1	Number of interfaces in this configuration
bConfigurationValue	1	Identification number of this configuration
iConfiguration	1	Index to the string descriptor specifying the source code for this configuration
bmAttributes	1	Features of this configuration
bMaxPower	1	Maximum current consumed in this configuration (in 2 μA units)

Table 2-7. Interface Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bInterfaceNumber	1	Identification number of this interface
bAlternateSetting	1	Whether the alternative settings are specified for this interface
bNumEndpoints	1	Number of endpoints of this interface
bInterfaceClass	1	Class code
bInterfaceSubClass	1	Subclass code
bInterfaceProtocol	1	Protocol code
iInterface	1	Index to the string descriptor specifying the source code for this interface

Table 2-8. Endpoint Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bEndpointAddress	1	Transfer direction of this endpoint Address of this endpoint
bmAttributes	1	Transfer type of this endpoint
wMaxPacketSize	2	Maximum packet size of this transfer
bInterval	1	Polling interval of this endpoint

Table 2-9. String Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bString	Any	Any data string

CHAPTER 3 SAMPLE DRIVER SPECIFICATIONS

This chapter provides details about the features and processing of the USB mass storage class (MSC) sample driver for the V850ES/Jx3-H and V850ES/Jx3-U and the specifications of the functions provided in the V850ES/Jx3-H and V850ES/Jx3-U.

3.1 Overview

3.1.1 Features

The sample driver can perform the following processing:

(1) Main routine

The system waits for an interrupt after initialization. If a suspend or resume interrupt occurs, suspend or resume processing is performed. For details, see **3.2.7 Suspend/resume processing**.

(2) Initialization

The USBF is set up for use by manipulating various registers. This setup includes specifying settings for the CPU registers of the V850ES/Jx3-H or V850ES/Jx3-U and specifying settings for the registers of the USBF. For details, see **3.2.1 CPU initialization** and **3.2.2 USBF initialization**.

(3) Interrupt servicing

The INTUSBF0 interrupt handler is used to monitor the statuses of the endpoint for control transfer (endpoint 0) and the endpoint for bulk-out transfer (reception) (endpoint 2) and processes the received requests and data. The INTUSBF1 interrupt handler is used to perform processing when a resume interrupt occurs. For details, see **3.2.3 USBF interrupt servicing (INTUSBF0)** and **3.2.4 USBF resume interrupt servicing (INTUSBF1)**.

(4) SCSI command processing

The received CBW data is analyzed to determine whether it is a SCSI command. If a SCSI command is received, processing corresponding to the command is executed. For details, see **3.1.4 Supported SCSI commands**.

3.1.2 Supported requests

Table 3-1 shows the USB requests defined by the hardware (the V850ES/Jx3-H or V850ES/Jx3-U) and firmware (the sample driver).

Table 3-1. USB Request Processing

Request Name	Codes								Processing
	0	1	2	3	4	5	6	7	
Standard request									
GET_INTERFACE	0x81	0x0A	0x00	0x00	0xXX	0xXX	0x01	0x00	Automatic hardware response
GET_CONFIGURATION	0x80	0x08	0x00	0x00	0x00	0x00	0x01	0x00	Automatic hardware response
GET_DESCRIPTOR Device	0x80	0x06	0x00	0x01	0x00	0x00	0xXX	0xXX	Automatic hardware response
GET_DESCRIPTOR Configuration	0x80	0x06	0x00	0x02	0x00	0x00	0xXX	0xXX	Automatic hardware response
GET_DESCRIPTOR String	0x80	0x06	0x00	0x03	0x00	0x00	0xXX	0xXX	Firmware response
GET_STATUS Device	0x80	0x00	0x00	0x00	0x00	0x00	0x02	0x00	Automatic hardware response
GET_STATUS Interface	0x81	0x00	0x00	0x00	0xXX	0xXX	0x02	0x00	Automatic hardware STALL response
GET_STATUS Endpoint n	0x82	0x00	0x00	0x00	0xXX	0xXX	0x02	0x00	Automatic hardware response
CLEAR_FEATURE Device	0x00	0x01	0x01	0x00	0x00	0x00	0x00	0x00	Automatic hardware response
CLEAR_FEATURE Interface	0x01	0x01	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware STALL response
CLEAR_FEATURE Endpoint n	0x02	0x01	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_DESCRIPTOR	0x00	0x07	0xXX	0xXX	0xXX	0xXX	0xXX	0xXX	Firmware STALL response
SET_FEATURE Device	0x00	0x03	0x01	0x00	0x00	0x00	0x00	0x00	Automatic hardware response
SET_FEATURE Interface	0x02	0x03	0xXX	0xXX	0xXX	0xXX	0x00	0x00	Automatic hardware STALL response
SET_FEATURE Endpoint n	0x02	0x03	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_INTERFACE	0x01	0x0B	0xXX	0xXX	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_CONFIGURATION	0x00	0x09	0xXX	0xXX	0x00	0x00	0x00	0x00	Automatic hardware response
SET_ADDRESS	0x00	0x05	0xXX	0xXX	0x00	0x00	0x00	0x00	Automatic hardware response
Class requests									
MASS_STORAGE_RESET	0x21	0xFE	0x00	0x00	0xXX	0xXX	0x00	0x00	Firmware response
GET_MAX_LUN	0xA1	0xFF	0x00	0x00	0xXX	0xXX	0x01	0x00	Firmware response
Other requests	Other than the above								Firmware STALL response

Remark Hardware: V850ES/Jx3-H or V850ES/Jx3-U
Firmware: Sample driver
0xXX: Undefined value

(1) Standard requests

The sample driver returns the following responses for requests to which the hardware (the V850ES/Jx3-H or V850ES/Jx3-U) does not automatically respond.

(a) GET_DESCRIPTOR_string

The host issues this request to acquire the string descriptor of the function device.

If this request is received, the sample driver transmits the requested string descriptor to the host through a control read transfer.

(b) SET_DESCRIPTOR

The host issues this request to specify the descriptor of the function device.

If this request is received, the sample driver returns a STALL response.

(2) Class requests

The sample driver responds to class requests of the USB mass storage class (MSC) bulk-only transfer protocol by using the following requests:

(a) GET_MAX_LUN

This request is used to acquire the number of logical units of the MSC device.

The host specifies the LUN in the bCBWLUN field when it transmits the CBW.

When a GET_MAX_LUN request is received, the sample driver returns 0 (the number of logical units = 1).

Table 3-2. Format of the GET_MAX_LUN Request

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0xA1	0xFE	0x0000	0x0000	0x0001	1 byte

(b) MASS_STORAGE_RESET

This request is used to reset the interfaces related to the MSC device.

The sample driver resets the interface of the USB function controller it uses when it receives a MASS_STORAGE_RESET request.

Table 3-3. Format of the GET_MAX_LUN Request

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x21	0xFF	0x0000	0x0000	0x0000	None

(3) Undefined requests

If an undefined request is received, the sample driver returns a STALL response.

3.1.3 Descriptor settings

The settings of each descriptor specified by the sample driver are shown below. These settings are included in header file `usbstrg_desc.h`.

(1) Device descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_device request.

The settings are stored in the UF0DDn registers (where n = 0 to 17) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_device request.

Table 3-4. Device Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x12	Descriptor size: 18 bytes
bDescriptorType	1	0x01	Descriptor type: device
bcdUSB	2	0x0200	USB specification release number: USB 2.0
bDeviceClass	1	0x00	Class code: none
bDeviceSubClass	1	0x00	Subclass code: none
bDeviceProtocol	1	0x00	Protocol code: No unique protocol is used.
bMaxPacketSize0	1	0x40	Maximum packet size of endpoint 0: 64
idVendor	2	0x0409	Vendor ID: NEC
idProduct	2	0x01D2	Product ID: V850ES/JG3-H
bcdDevice	2	0x0001	Device release number: 1st version
iManufacturer	1	0x01	Index to the string descriptor representing the manufacturer: 1
iProduct	1	0x00	Index to the string descriptor representing the product: 0
iSerialNumber	1	0x00	Index to the string descriptor representing the device production number: 0
bNumConfigurations	1	0x01	Number of configurations: 1

(2) Configuration descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Table 3-5. Configuration Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x02	Descriptor type: configuration
wTotalLength	2	0x0020	Total number of bytes of the configuration, interface, and endpoint descriptors: 32 bytes
bNumInterfaces	1	0x01	Number of interfaces in this configuration: 1
bConfigurationValue	1	0x01	Identification number of this configuration: 1
iConfiguration	1	0x00	Index to the string descriptor specifying the source code for this configuration: 0
bmAttributes	1	0x80	Features of this configuration: bus-powered, no remote wakeup
bMaxPower	1	0x1B	Maximum current consumed in this configuration: 54 mA

(3) Interface descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Table 3-6. Interface Descriptor Settings for the Interface

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x04	Descriptor type: interface
bInterfaceNumber	1	0x00	Identification number of this interface: 0
bAlternateSetting	1	0x00	Whether the alternative settings are specified for this interface: no
bNumEndpoints	1	0x02	Number of endpoints of this interface: 2
bInterfaceClass	1	0x08	Class code: mass storage class
bInterfaceSubClass	1	0x06	Subclass code: SCSI transparent command set
bInterfaceProtocol	1	0x50	Protocol code: bulk-only transfer
iInterface	1	0x00	Index to the string descriptor specifying the source code for this interface: 0

(4) Endpoint descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Two descriptor types are specified because the sample driver uses two endpoints.

Table 3-7. Endpoint Descriptor Settings for Endpoint 1 (Bulk-in)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: endpoint
bEndpointAddress	1	0x02	Transfer direction of this endpoint: OUT Address of this endpoint: 2
bmAttributes	1	0x02	Transfer type of this endpoint: bulk
wMaxPacketSize	2	0x0040	Maximum packet size of this transfer: 64 bytes
bInterval	1	0x00	Polling interval of this endpoint: 0 ms

Table 3-8. Endpoint Descriptor Settings for Endpoint 2 (Bulk-Out)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: endpoint
bEndpointAddress	1	0x81	Transfer direction of this endpoint: IN Address of this endpoint: 1
bmAttributes	1	0x02	Transfer type of this endpoint: bulk
wMaxPacketSize	2	0x0040	Maximum packet size of this transfer: 64 bytes
bInterval	1	0x00	Polling interval of this endpoint: 0 ms

(5) String descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_string request.

If a GET_DESCRIPTOR_string request is received, the sample driver extracts the settings of this descriptor from `usbstrg_desc.h` and stores them into the UF0E0W register of the USBF.

Table 3-9. String Descriptor Settings

(a) String 0

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x04	Descriptor size: 4 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString	2	0x09, 0x04	Language code: English (U.S.)

(b) String 1

Field	Size (Bytes)	Specified Value	Description
bLength[Note 1]	1	0x2A	Descriptor size: 42 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString[Note 2]	40	–	Vendor: NEC Electronics Corporation

Notes 1. The specified value depends on the size of the bString field.

2. The vendor can freely set up the size and specified value of this field.

3.1.4 Supported SCSI commands

For the sample driver, the SCSI transfer command set (0x06) is specified as the subclass.

Table 3-10 shows the SCSI commands supported by the sample driver. The sample driver returns a STALL response if it receives a command that is not shown in Table 3-10.

Table 3-10. SCSI Commands Supported by the Sample Driver

Command Name	Code	Bulk Transfer Direction	Description
TEST_UNIT_READY	0x00	NO DATA	Checks the type and configuration of the device.
REQUEST_SENSE	0x03	IN	Acquires sense data.
READ6	0x08	IN	Reads data.
WRITE6	0x0A	OUT	Writes data.
SEEK	0x0B	NO DATA	Seeks the data position.
INQUIRY	0x12	IN	Acquires configuration information and attributes.
MODE_SELECT	0x15	OUT	Specifies various parameters.
MODE_SENSE6	0x1A	IN	Reads the values of various parameters.
START_STOP_UNIT	0x1B	NO DATA	Loads or unloads media and starts and stops motors.
PREVENT	0x1E	NO DATA	Enables or disables media removal.
READ_FORMAT_CAPACITIES	0x23	IN	Acquires memory capacity information.
READ_CAPACITY	0x25	IN	Acquires capacity information.
READ10	0x28	IN	Reads data.
WRITE10	0x2A	OUT	Writes data.
WRITE_VERIFY	0x2E	OUT	Writes and verifies data to be valid.
VERIFY	0x2F	NO DATA	Verifies data to be valid.
SYNCHRONIZE_CACHE	0x35	NO DATA	Writes the data left in the cache.
WRITE_BUFF	0x3B	OUT	Writes data to buffer memory.
MODE_SELECT10	0x55	OUT	Specifies various parameters.
MODE_SENSE10	0x5A	IN	Reads the values of various parameters.

(1) TEST_UNIT_READY command (0x00)

This command reports the logical unit status to the initiator (host). For the sample driver, this command initializes the sense data and ends normally.

Table 3-11. TEST_UNIT_READY Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x00)							
1	Logical unit number (LUN)			Reserved				
2 to 4	Reserved							
5	Reserved						Flag	Link

(2) REQUEST_SENSE command (0x03)

This command transmits sense data to the host. For the sample driver, this command transmits the sense data shown in Table 3-14 to the host.

Table 3-12. REQUEST_SENSE Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x03)							
1	Logical unit number (LUN)			Reserved				
2	Page code							
3	Reserved							
4	Additional data length							
5	Reserved						Flag	Link

Table 3-13. REQUEST_SENSE Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	VALID	Response code						
1	Reserved							
2	Filemark	EOM	ILI	Reserved	Sense key			
3 to 6	Information							
7	Additional sense data length (n – 7 bytes)							
8 to 11	Command-specific information							
12	ASC (additional sense code)							
13	ASCQ (additional sense code qualifier)							
14	FRU (field replaceable unit) code							
15	SKSV	Sense-key-specific information						
16	Sense-key-specific information							
17	Sense-key-specific information							
18 to n	Additional sense data (variable data length)							

Table 3-14. Sense Data

Sense Key	ASC	ASCQ	Description
0x00	0x00	0x00	No sense
0x05	0x00	0x00	Invalid request
0x05	0x20	0x00	Invalid command operation code
0x05	0x24	0x00	Invalid field in command packet

(3) READ6 command (0x08)

This command transfers the data of the logic data blocks in the specified range to the host.

Table 3-15. READ6 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x08)							
1	Logical unit number (LUN)			Logical block address (LBA)				
2 and 3	Logical block address (LBA)							
4	Transfer data length							
5	Reserved						Flag	Link

(4) WRITE6 command (0x0A)

This command writes the received data to a specified block in the storage device.

Table 3-16. WRITE6 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x0A)							
1	Logical unit number (LUN)			Logical block address (LBA)				
2 and 3	Logical block address (LBA)							
4	Transfer data length							
5	Reserved						Flag	Link

(5) SEEK command (0x0B)

This command seeks the specified position in the recording medium. For the sample driver, this command initializes the sense data and ends normally.

Table 3-17. SEEK Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x0B)							
1	Logical unit number (LUN)			Logical block address (LBA)				
2 and 3	Logical block address (LBA)							
4	Reserved							
5	Reserved						Flag	Link

(6) INQUIRY command (0x12)

This command reports configuration information and attributes of the device to the host. For the sample driver, this command transmits the INQUIRY_TABLE values to the host.

Table 3-18. INQUIRY Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x12)							
1	Logical unit number (LUN)			Reserved			CMDDT	EVPD
2	Page code							
3	Reserved							
4	Additional data length							
5	Reserved						Flag	Link

Table 3-19. INQUIRY Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Identifier				Device type			
1	RMB	Device type modifier						
2	ISO version		ECMA version			ANSI version		
3	AENC	TmIOP	Response data format					
4	Additional data length (n – 4 bytes)							
5 and 6	Reserved							
7	RelAdr	WBus32	WBus16	Sync	Linked	Reserved	CmdQue	SftRe
8 to 15	Vendor ID (ASCII code)							
16 to 31	Product ID (ASCII code)							
32 to 35	Product version (ASCII code)							
36 to 55	Vendor-specific information							
56 to 95	Reserved							
96 to n	Additional vendor-specific information (variable data length)							

List 3-1. INQUIRY_TABLE

```
UINT8  INQUIRY_TABLE[INQUIRY_LENGTH]={
       0x00,                /*Qualifier, device type code*/
       0x80,                /*RMB, device type modification child*/
       0x02,                /*ISO Version, ECMA Version, ANSI Version*/
       0x02,                /*AENC, TrmIOP, response data form*/
       0x1F,                /*addition data length*/
       0x00,0x00,0x00,      /*reserved*/
       'N','E','C',' ','c','o','r','p',                /*vender ID*/
       'S','t','o','r','a','g','e','F','n','c','D','r','i','v','e','r',
                                                       /*product ID*/
       '0','.','0','1'      /*Product Revision*/
};
```

(7) MODE_SELECT command (0x15)

This command specifies and changes various parameters such as for the device data format. For the sample driver, this command writes values to MODE_SELECT_TABLE.

Table 3-20. MODE_SELECT Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x15)							
1	Logical unit number (LUN)			PF	Reserved			SP
2 and 3	Reserved							
4	Additional data length							
5	Reserved						Flag	Link

Table 3-21. MODE_SELECT Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Mode parameter length							
1	Media type							
2	Device-specific parameter							
3	Block descriptor length							
4	Density code							
5 to 7	Number of blocks							
8	Reserved							
9 to 11	Block length							
12	PS	1	Page code					
13	Page length (n – 13 bytes)							
14 to n	Mode parameter (variable data length)							

List 3-2. MODE_SELECT_TABLE

```
UINT8   MODE_SELECT_TABLE[MODE_SELECT_LENGTH]={
    0x17,              /*length of the mode parameter*/
    0x00,              /*medium type*/
    0x00,              /*device peculiar parameter*/
    0x08,              /*length of the block descriptor*/
    0x00,              /*density code*/
    0x00,0x00,0xC0,    /*number of the blocks*/
    0x00,              /*Reserved*/
    0x00,0x02,0x00,    /*length of the block*/
    0x01,              /*PS, page code*/
    0x0A,              /*length of the page*/
    0x08,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00    /*mode parameter*/
};
```

(8) MODE_SENSE6 command (0x1A)

This command transmits mode selection parameter values and attributes of the device to the host. For the sample driver, this command transmits the MODE_SENSE_TABLE values to the host.

Table 3-22. MODE_SENSE6 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x14)							
1	Logical unit number (LUN)			Reserved	DBD	Reserved		
2	PC		Page code					
3	Reserved							
4	Additional data length							
5	Reserved						Flag	Link

Table 3-23. MODE_SENSE6 Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Mode parameter length							
1	Media type							
2	Device-specific parameter							
3	Block descriptor length							
4	Density code							
5 to 7	Number of blocks							
8	Reserved							
9 to 11	Block length							
12	PS	Reserved	Page code					
13	Page length (n – 13 bytes)							
14 to n	Mode parameter (variable data length)							

List 3-3. MODE_SENSE_TABLE

```
UINT8   MODE_SENSE_TABLE[MODE_SENSE_LENGTH]={
    0x17,               /*length of the mode parameter*/
    0x00,               /*medium type*/
    0x00,               /*device peculiar parameter*/
    0x08,               /*length of the block descriptor*/
    0x00,               /*density code*/
    0x00,0x00,0xC0,     /*number of the blocks*/
    0x00,               /*Reserved*/
    0x00,0x02,0x00,     /*length of the block*/
    0x81,               /*PS, page code*/
    0x0A,               /*length of the page*/
    0x08,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00   /*mode parameter*/
};
```

(9) START_STOP_UNIT command (0x1B)

This command enables or disables accessing a device. For the sample driver, this command initializes the sense data and ends normally.

Table 3-24. START_STOP_UNIT Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x1B)							
1	Logical unit number (LUN)			Reserved				IMMED
2	Reserved							
3	Reserved							
4	Reserved						Load/Eject	Start
5	Reserved						Flag	Link

(10) PREVENT command (0x1E)

This command enables or disables media removal. For the sample driver, this command ends normally without performing any processing.

Table 3-25. PREVENT Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x1E)							
1	Reserved							
2	Reserved							
3	Reserved							
4	Reserved						Persistent	Prevent
5	Reserved						Flag	Link

(11) READ_FORMAT_CAPACITIES command (0x23)

This command reports the device capacity (the number of blocks and block length) to the host. For the sample driver, this command transmits the READ_FORMAT_CAPACITY_TABLE values to the host.

Table 3-26. READ_FORMAT_CAPACITIES Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x23)							
1	Logical unit number (LUN)			Reserved				
2 to 6	Reserved							
7 to 8	Transfer data length							
9	Reserved						Flag	Link

Table 3-27. READ_FORMAT_CAPACITIES Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0 to 2	Reserved							
3	Capacity list length (bytes)							
5 to 7	Number of blocks							
8	Reserved						Descriptor code	
9 to 11	Block length							
12 to 15	Number of blocks							
16	Reserved							
17 to 19	Block length							

List 3-4. READ_FORMAT_CAPACITY_TABLE

```
UINT8  READ_FORMAT_CAPACITY_TABLE[READ_FORM_CAPA_LENGTH]={
       0x00,0x00,0x00,              /* Reserved            */
       0x08,                        /* Capacity list length      */
       0x00,0x00,0x00,0x30,         /* Number of blocks    */
       0x01,                        /* Descriptor Code     */
       0x00,0x02,0x00,              /* Block length        */
       0x00,0x00,0x00,0x30,         /* Number of blocks    */
       0x00,                        /* Reserved            */
       0x00,0x02,0x00               /* Block length        */
};
```

(12) READ_CAPACITY command (0x25)

This command reports the data capacity of the device to the host. For the sample driver, this command transmits the values of READ_CAPACITY_TABLE to the host.

Table 3-28. READ_CAPACITY Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x25)							
1	Logical unit number (LUN)			Reserved				RA
2 to 8	Reserved							
9	Reserved						Flag	Link

Table 3-29. READ_CAPACITY Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0 to 3	Logical block address (LBA)							
4 to 7	Block length (bytes)							

List 3-5. READ_CAPACITY_TABLE

```
UINT8  READ_CAPACITY_TABLE[8]={     /*big endian*/
    0x00,0x00,0x00,0x2F,            /*number of the outline reason blocks - 1*/
    0x00,0x00,0x02,0x00             /*size of the data block (bytes)*/
};
```

(13) READ10 command (0x28)

This command transfers the data of the logic data blocks in the specified range to the host.

Table 3-30. READ10 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x28)							
1	Logical unit number (LUN)			OPD	FUA	Reserved		RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(14) WRITE10 command (0x2A)

This command writes the received data to the specified block in the device.

Table 3-31. WRITE10 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x2A)							
1	Logical unit number (LUN)			OPD	FUA	EBP	TSR	RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(15) WRITE_VERIFY command (0x2E)

This command writes the received data to the specified block in the device. Next, the command checks the validity of the data. For the sample driver, this command only writes the received data.

Table 3-32. WRITE_VERIFY Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x2E)							
1	Logical unit number (LUN)			OPD	FUA	EBP	BYTCHK	RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(16) VERIFY command (0x2F)

This command checks the validity of the data in the device. For the sample driver, this command ends normally without performing any processing.

Table 3-33. VERIFY Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x2F)							
1	Logical unit number (LUN)			OPD	Reserved		BYTCHK	RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(17) SYNCHRONIZE_CACHE command (0x35)

This command matches the values of cache memory and a medium for blocks in the specified range. For the sample driver, this command initializes the sense data and ends normally.

Table 3-34. SYNCHRONIZE_CACHE Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x35)							
1	Logical unit number (LUN)			Reserved			IMMED	RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(18) WRITE_BUFF command (0x3B)

This command writes data to memory (the data buffer). For the sample driver, this command reads and then discards data, and then ends normally.

Table 3-35. WRITE_BUFF Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x3B)							
1	Logical unit number (LUN)			OPD	FUA	EBP	Reserved	RA
2 to 5	Logical block address (LBA)							
6	Reserved							
7 and 8	Transfer data length							
9	Reserved						Flag	Link

(19) MODE_SENSE10 command (0x5A)

This command reports mode selection parameter values and attributes of the device to the host. For the sample driver, this command transmits the values of MODE_SENSE10_TABLE to the host.

Table 3-36. MODE_SENSE10 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x5A)							
1	Reserved			LLBAA	DBD	Reserved		
2	PC		Page code					
3 to 6	Reserved							
7 and 8	Added data length							
9	Reserved						Flag	Link

Table 3-37. MODE_SENSE10 Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Mode parameter length							
1	Media type							
2	Device-specific parameter							
3	Block descriptor length							
4	Density code							
5 to 7	Number of blocks (0x0000C0)							
8	Reserved							
9 to 11	Block length (0x000200)							
12	PS	Reserved	Page code					
13	Page length (n – 13 bytes)							
14 to n	Mode parameter (variable data length)							

List 3-6. MODE_SENSE10_TABLE

```
UINT8   MODE_SENSE10_TABLE[MODE_SENSE10_LENGTH]={
    0x00,0x1A,          /*length of the mode parameter*/
    0x00,               /*medium type*/
    0x00,               /*device peculiar parameter*/
    0x00,0x00,          /*Reserved*/
    0x00,0x08,          /*length of the block descriptor*/
    0x00,               /*density code*/
    0x00,0x00,0xC0,     /*number of the blocks*/
    0x00,               /*Reserved*/
    0x00,0x02,0x00,     /*length of the block*/
    0x81,               /*PS, page code*/
    0x0A,               /*length of the page*/
    0x08,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00   /*mode parameter*/
};
```

(20) MODE_SELECT10 command (0x55)

This command specifies and changes various parameters such as for the device data format. For the sample driver, this command writes values to MODE_SELECT10_TABLE.

Table 3-38. MODE_SELECT10 Command Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x55)							
1	Logical unit number (LUN)			PF	Reserved			SP
2 to 6	Reserved							
7 and 8	Additional data length							
9	Reserved						Flag	Link

Table 3-39. MODE_SELECT10 Data Format

Bytes \ Bit	7	6	5	4	3	2	1	0
0	Mode parameter length							
1	Media type							
2	Device-specific parameter							
3	Block descriptor length							
4	Density code							
5 to 7	Number of blocks							
8	Reserved							
9 to 11	Block length							
12	PS	1	Page code					
13	Page length (n – 13 bytes)							
14 to n	Mode parameter (variable data length)							

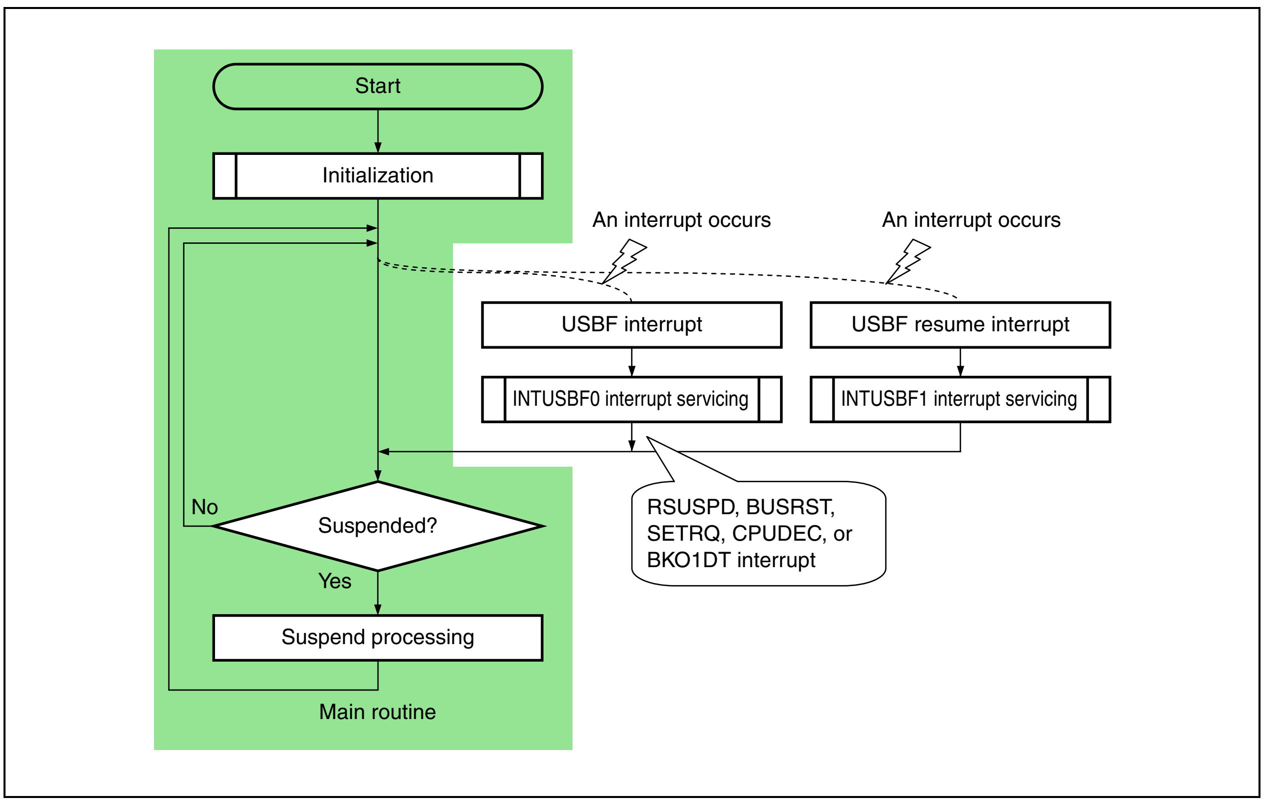
List 3-7. MODE_SELECT10_TABLE

```
UINT8   MODE_SELECT10_TABLE[MODE_SELECT10_LENGTH]={
     0x00,0x1A,          /*length of the mode parameter*/
     0x00,               /*medium type*/
     0x00,               /*device peculiar parameter*/
     0x00,0x00,          /*Reserved*/
     0x00,0x08,          /*length of the block descriptor*/
     0x00,               /*density code*/
     0x00,0x00,0xC0,     /*number of the blocks*/
     0x00,               /*Reserved*/
     0x00,0x02,0x00,     /*length of the block*/
     0x01,               /*PS, page code*/
     0x0A,               /*length of the page*/
     0x08,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00    /*mode parameter*/
};
```

3.2 Operation of Each Section

The processing sequence below is performed when the sample driver is executed. This section describes each processing.

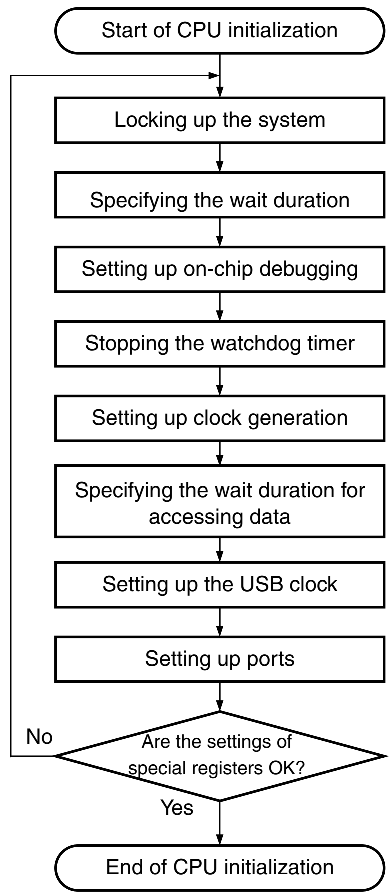
Figure 3-1. Sample Driver Processing Flowchart



3.2.1 CPU Initialization

The settings necessary to use the USBF are specified.

Figure 3-2. CPU Initialization Flowchart



(1) Locking up the system

The system is locked up until the CPU clock frequency stabilizes.
Whether the LOCK bit of the LOCKR register is 0 is monitored.

(2) Specifying the wait duration

The number of clock cycles the system waits for a bus access to the internal peripheral I/O register is specified. 0x12 is written to the VSWC register. For the V850ES/Jx3-H and V850ES/Jx3-U, the number of clock cycles the system waits varies depending on the operation frequency. However, for the sample driver, operation at 33.3 MHz to 48 MHz is assumed.

(3) Setting up on-chip debugging

The CPU operation mode is switched. This setup is necessary for the V850ES/JG3-H and V850ES/JG3-U (but not for the V850ES/JH3-H and V850ES/JH3-U).
1 is written to the OCDM0 bit of the OCDM register to enable the V850ES/JG3-H or V850ES/JG3-U to operate in on-chip debugging mode.

(4) Stopping the watchdog timer

The operation mode of the watchdog timer is switched.

0x00 is written to the WDTM2 register to stop the watchdog timer.

(5) Setting up clock generation

The operation of the internal CPU clock is set up. The following four registers are accessed:

<1> 0x0B is written to the CKC register to multiply the frequency of the clock signal generated by the internal oscillator by 8 by using the PLL.

<2> 0x03 is written to the PLLCTL register to specify PLL mode and start the PLL.

<3> 0x00 is written to the PCC register to specify f_{XX} as the internal clock frequency. For the sample driver, this is specified assuming operation using the main clock.

<4> 0x03 is written to the OSTS register to set the oscillation stabilization time to $2^{13}/f_X$ (1.365 ms).

(6) Specifying the wait duration for accessing data

The number of clock cycles the system waits when peripherals that operate at different speeds are accessed for data is specified for each peripheral.

0x1171 is written to the DWC0 register to access all three peripherals while the system waits for one clock cycle.

(7) Setting up the USB clock

The operation of the USBF is set up. The following three registers are accessed:

<1> 0x02 is written to the UCKSEL register to supply the internal clock signal to the USBF.

<2> 0x00 is written to the UFCKMSK register to enable the USBF.

<3> 0x00 is written to the UHCKMSK register to enable the data-dedicated RAM (8 KB) to use for the USBF.

(8) Setting up ports

The operation of the USBF is set up. The following registers are accessed:

<1> 0xFFFF is written to the PMCDL register to select the AD15 to AD0 pins.

<2> 0x3F is written to the PMC6 register, 0x02 is written to the PFC6 register, and 0x3F is written to the PFCE6 register to select the CS3, CS2, CS0, ASTB, RD, and WAIT pins.

<3> 0x03 is written to the PMCCT register to select the WR1 and WR0 pins.

<4> 0x02 is written to the PMCCM register to select the CLKOUT pin.

<5> 0 is written to bit 2 of the PMC0, PM0, and P0 registers to stop UCLK input.

(9) Checking for errors when setting up special registers

Protection errors are checked for.

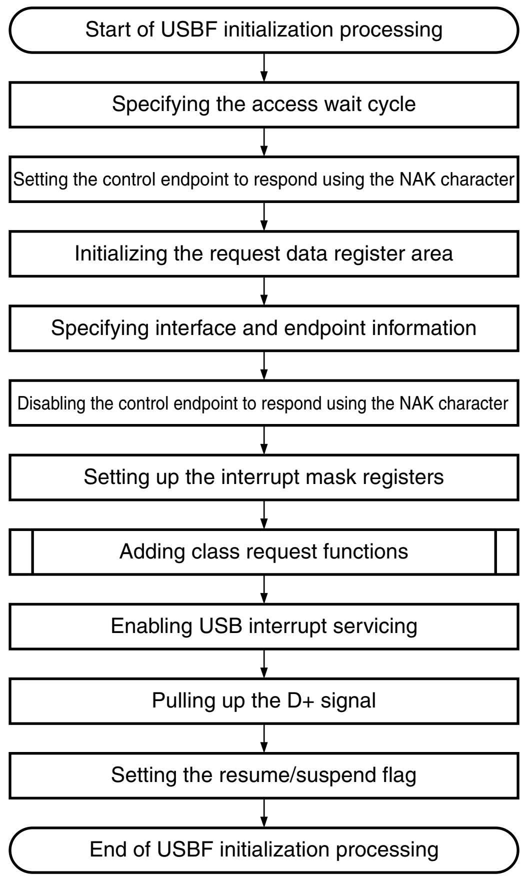
Whether the PRERR bit of the SYS register is 0 is monitored.

A protection error occurs if the OCDM, CKC, or PCC register is manipulated without following the specified procedure.

3.2.2 USBF initialization processing

The settings necessary to use the USBF are specified.

Figure 3-3. USBF Initialization Processing Flowchart



(1) Specifying the access wait cycle

0x06 is written to the CPUBCTL register to insert a wait cycle at the end of the CPU bus cycle and for every data set or bulk data set when accessing the bulk register for reading data.

(2) Setting the control endpoint to respond using the NAK character

The NAK response operations for all requests are switched.

1 is written to the EP0NKA bit of the UF0E0NA register so that the hardware responds to all requests, including requests that are automatically responded to, with a NAK.

The EP0NKA bit is used by software until the data used by requests that are automatically responded to has been added to prevent the hardware from returning unintended data for such requests.

(3) Initializing the request data register area

The descriptor data transmitted in response to a GET_DESCRIPTOR request is added to various registers.

The following registers are accessed:

<1> 0x00 is written to the UF0DSTL register to disable remote wakeup and operate the USBF as a bus-powered device.

<2> 0x00 is written to the UF0EnSL registers (where n = 0 to 2) to indicate that endpoint n operates normally.

<3> The total data length (number of bytes) of the required descriptor is written to the UF0DSCL register to determine the range of the UF0CIEn registers (where n = 0 to 255).

<4> The device descriptor data is written to the UF0DDn registers (where n = 0 to 17).

<5> The data of the configuration, interface, and endpoint descriptors is written to the UF0CIEn registers (where n = 0 to 255).

<6> 0x00 is written to the UF0MODC register to enable automatic responses to GET_DESCRIPTOR_configuration requests.

(4) Specifying interface and endpoint information

Information such as the number of supported interfaces, whether the alternative setting is used, and the relationship between the interfaces and endpoints is specified for various registers.

The following registers are accessed:

<1> 0x00 is written to the UF0AIFN register to enable only one interface.

<2> 0x00 is written to the F0AAS register to disable the alternative setting.

<3> 0x20 is written to the UF0E1IM register to link endpoint 1 to interface 0.

<4> 0x20 is written to the UF0E2IM register to link endpoint 2 to interface 0.

(5) Disabling the control endpoint to respond using the NAK character

The NAK response operations for all requests are switched.

0x00 is written to the UF0E0NA register to restart responses corresponding to each request, including requests that are automatically responded to.

(6) Setting up the interrupt mask registers

Masking is specified for each USBF interrupt source.

The following registers are accessed:

<1> 1 is written to all valid bits of the UF0ICn registers (where n = 0 to 4) to clear all interrupt sources.

<2> 1 is written to all valid bits of the UF0FICn registers (where n = 0 and 1) to clear all transfer FIFOs.

<3> 0x1F is written to the UF0IM0 register to mask interrupt sources indicated by the UF0IS0 register other than those of the RSUSPDM and BUSRSTM interrupts.

<4> 0x7E is written to the UF0IM1 register to mask interrupt sources indicated by the UF0IS1 register other than those of the CPUDEC interrupt.

<5> 0xF3 is written to the UF0IM2 register to mask all interrupt sources indicated by the UF0IS2 register.

<6> 0xFE is written to the UF0IM3 register to mask interrupt sources indicated by the UF0IS3 register other than those of the BKO1DT interrupt.

<7> 0x20 is written to the UF0IM4 register to mask all interrupt sources indicated by the UF0IS4 register.

(7) Adding class request functions

The function for adding class request functions (`usbf850_setfunction_storage`) is called to add all class request function addresses.

(8) Enabling USB interrupt servicing

0x0007 is written to the BRGINTE register, 0 is written to the UFMK0 bit of the UFIC0 register, and 1 is written to the UFMK1 bit of the UFIC1 register to enable INTUSB0 interrupt servicing and disable INTUSB1 interrupt servicing.

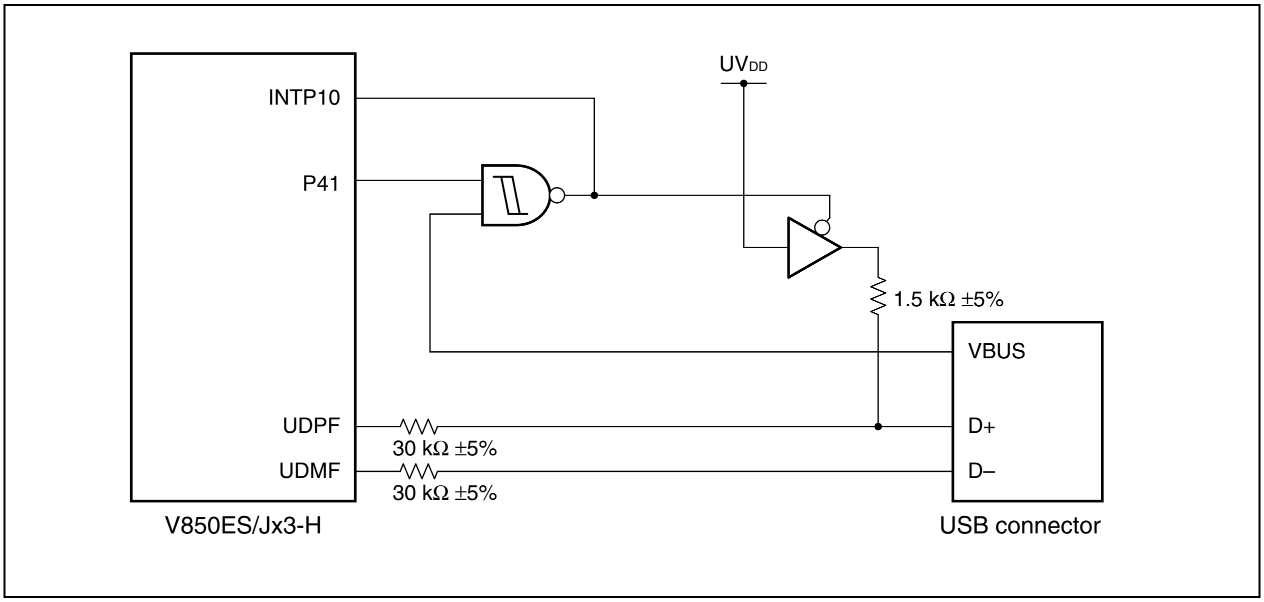
(9) Pulling up the D+ signal

A high level signal is output from the D+ pin to report to the host that a device has been connected. For the sample driver, the connections shown in Figure 3-4 are assumed and the following registers are accessed:

<1> 0xFC is written to the PM4 register of the CPU to set P41 to output mode.

<2> 0x02 is written to the P4 register of the CPU to output 1 from P41.

Figure 3-4. USBF Connection Example



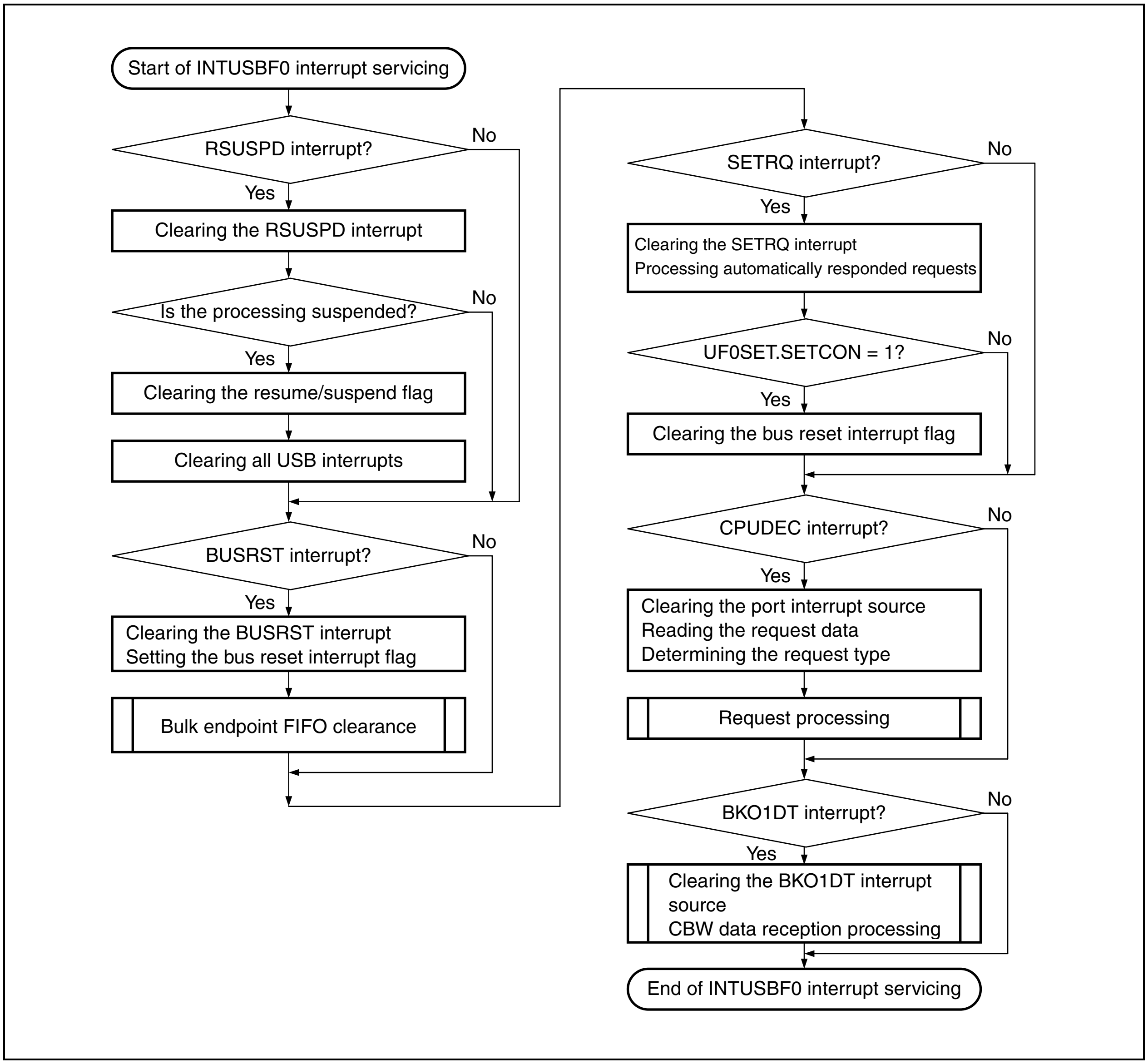
(10) Setting the resume/suspend flag

The resume/suspend flag (rs_flag) is set to “RESUME (0x01)”.

3.2.3 USBF interrupt servicing (INTUSBF0)

The INTUSBF0 interrupt handler is used to monitor the statuses of the endpoint for control transfer (endpoint 0) and the endpoint for bulk-out transfer (reception) (endpoint 2) and to perform processing corresponding to received requests and data.

Figure 3-5. INTUSBF0 Interrupt Handler Processing Flowchart



(1) RSUSPD interrupt servicing

If the RSUSPD bit of the UF0IS0 register is 1, an RSUSPD interrupt is judged to have occurred.

If an RSUSPD interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the RSUSPDC bit of the UF0IC0 register.)
- Whether the processing is suspended or has resumed is determined.

(2) Processing during suspension

If the RSUM bit of the UF0EPS1 register is 1, the processing is judged to have been suspended.

If the resume/suspend flag (rs_flag) is already set to SUSPEND (0x00) when processing is suspended, the subsequent processing is not performed and INTUSBF0 interrupt servicing ends.

If the resume/suspend flag (rs_flag) is not set to SUSPEND, it is set to SUSPEND to clear all USB interrupt sources. This omits the subsequent INTUSBF0 interrupt servicing.

If the processing is suspended, all USB interrupt sources are cleared. This omits all subsequent INTUSB0B interrupt servicing.

(3) BUSRST interrupt servicing

If the BUSRST bit of the UF0IS0 register is 1, a BUSRST interrupt is judged to have occurred.

If a BUSRST interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the BUSRST bit of the UF0IC0 register.)
- The bus reset interrupt flag (usbf_busrst_flg) is set to 1.
- The bulk endpoint FIFOs are cleared.

(4) SETRQ interrupt servicing

If the SETRQ bit of the UF0IS0 register is 1, an SETRQ interrupt is judged to have occurred.

If a SETRQ interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the SETRQ bit of the UF0IC0 register.)
- A request that is automatically responded to (SET_XXXX) is processed.

(5) Processing an automatically responded request (SET_XXXX)

If the SETCON bit of the UF0SET register is 1, a SET_CONFIGURATION request is received and automatic processing is judged to have been performed.

If automatic processing was performed, the bus reset interrupt flag (usbf_busrst_flg) is set to 0.

Caution To check whether a configured status has been entered, check the values of the UF0CNF register.

(6) CPUDEC interrupt servicing

If the CPUDEC bit of the UF0IS1 register is 1, a CPUDEC interrupt is judged to have occurred.

If a CPUDEC interrupt occurred, the following processing is performed:

- The port interrupt source is cleared. (0 is written to the PORT bit of the UF0IC1 register.)
- The received data is read from the FIFOs and request data is created.
- Request processing

(7) Request processing

Whether the request is one to which the hardware does not automatically respond (a standard, class, or vendor request) is determined and processing according to the type of request is executed.

Endpoint 0 is used for a control transfer. During the enumeration processing when a device is plugged in, almost all standard device requests are automatically processed by the hardware. Here, the standard, class, and vendor requests that are not automatically processed are processed.

(8) BKO1DT interrupt servicing

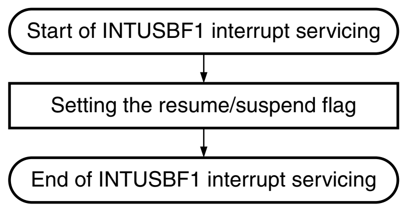
If the BKO1DT bit of the UF0IS3 register is set to 1, an interrupt is judged to have occurred.
If a BKO1DT interrupt occurred, the following processing is performed:

- The BKO1DT interrupt source is cleared. (0 is written to the BKO1DT bit of the UF0IC3 register.)
- The CBW data reception function (`usbf850_rx_cbw`) is called to receive CBW data.

3.2.4 USBF resume interrupt servicing (INTUSBF1)

The INTUSBF1 interrupt handler is used to perform processing when a resume interrupt occurs.
During this processing, the resume/suspend flag (rs_flag) is set to RESUME (0x01).
When the rs_flag is set to RESUME, the processing is performed in the main routine.

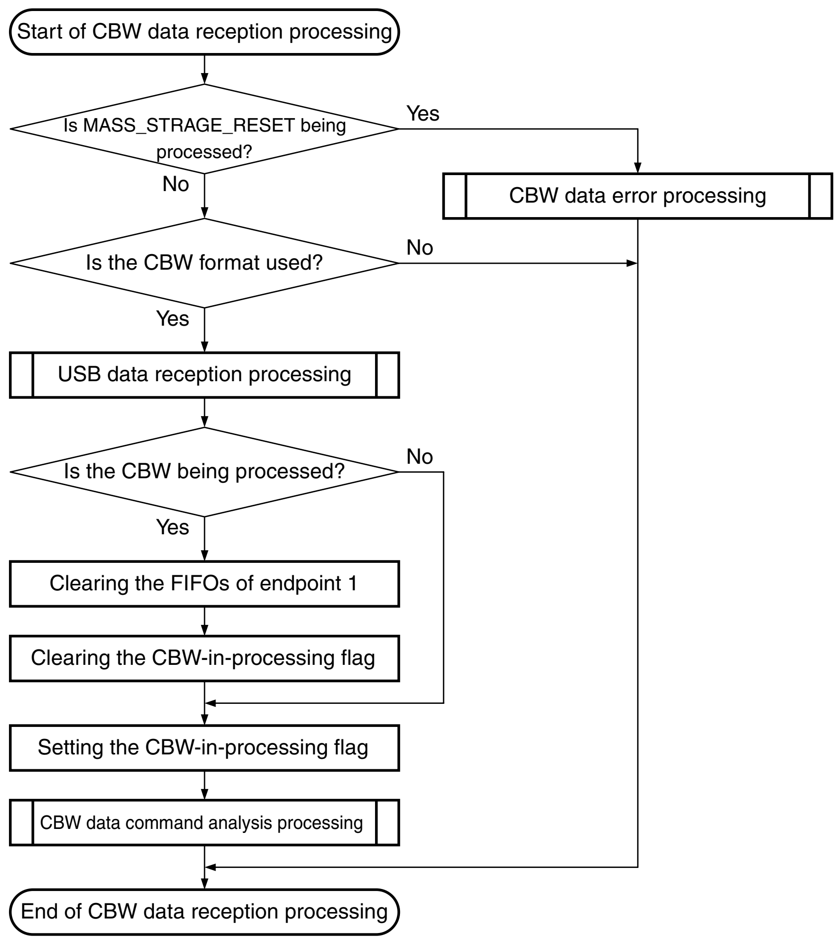
Figure 3-6. INTUSBF1 Interrupt Handler Processing Flowchart



3.2.5 CBW data reception processing

During CBW data reception processing, data is read from the FIFOs of the bulk-out endpoint (endpoint 2) and then CBW data command analysis processing is called.

Figure 3-7. CBW Data Reception Processing Flowchart



(1) Judging whether processing is under execution by using the MASS_STRAGE_RESET processing flag

If the MASS_STRAGE_RESET processing flag (mass_storage_reset) is set to 1, processing is judged to be under execution.

If processing is under execution, the CBW data error processing function (`usbf850_cbw_error`) is called to end CBW data reception processing.

(2) Judging the CBW format

The size (length) of the data stored at the bulk-out endpoint (endpoint 2) is acquired from the UF0 bulk-out 1 length register (UF0BO1L). If the data length is 31 bytes, the data is judged to match the CBW format.

If the data is not in the CBW format, CBW data reception processing ends.

If the data is in the CBW format, the USB data reception processing function (`usbf850_data_receive`) is called to continue processing.

(3) Judging whether processing is under execution by using the CBW-processing-in-progress flag

If the CBW-processing-in-progress flag (cbw_in_cbw) is set to USB_CBW_PROCESS (0x01), processing is judged to be under execution.

If processing is under execution, the FIFOs of endpoint 1 are cleared and the CBW-processing-in-progress flag (cbw_in_cbw) is set to USB_CBW_END (0x00).

(4) Setting the CBW-processing-in-progress flag

The CBW-processing-in-progress flag (cbw_in_cbw) is set to USB_CBW_PROCESS (0x01).

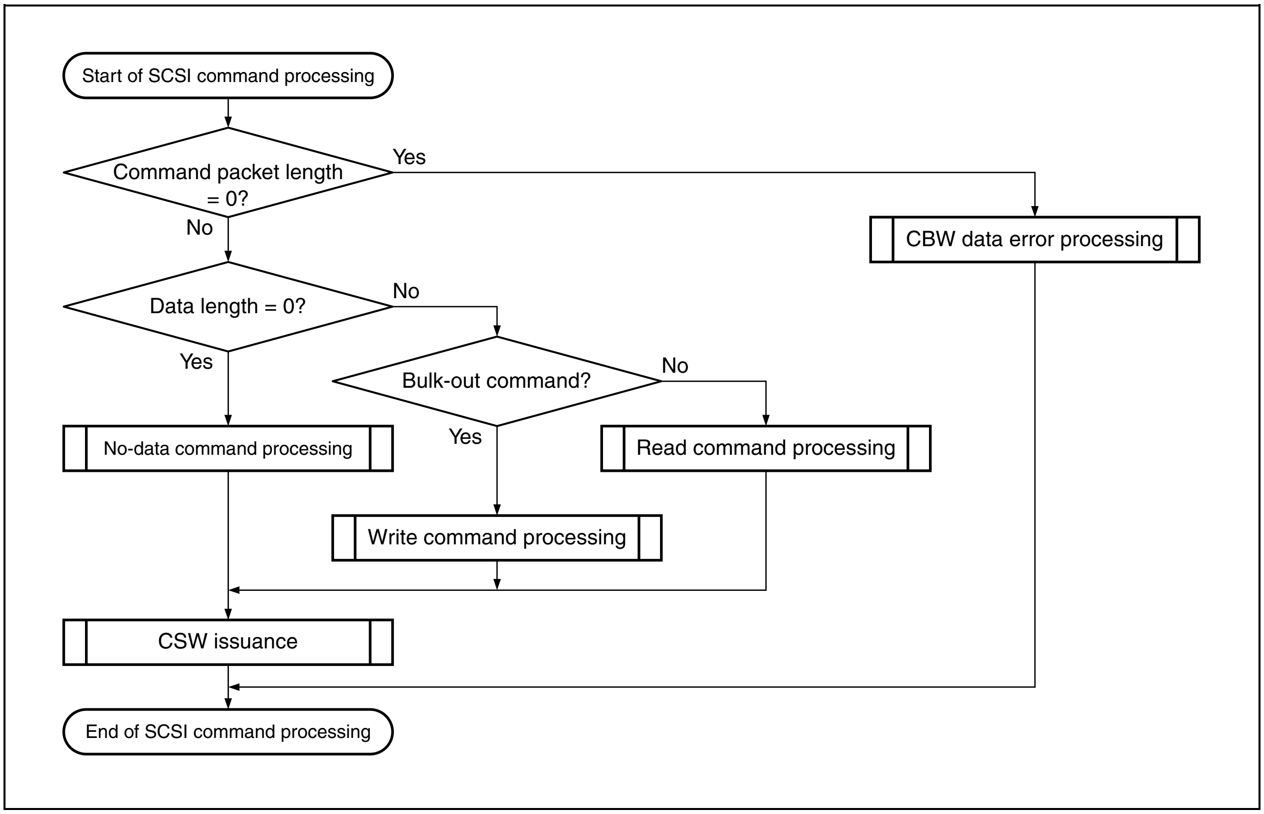
(5) CBW command analysis processing

The CBW data command analysis processing function (`usbf850_storage_cbwchk`) is called to perform processing for the received SCSI command.

3.2.6 SCSI command processing

If CBW data is received via USB, the CBW data command analysis processing function (`usbf850_storage_cbwchk`) is called to perform processing for the received SCSI command.

Figure 3-8. SCSI Command Processing Flowchart



(1) Judging SCSI commands

If the command packet length (bCBWCBLength) is 0x00, the received command is judged not to be a SCSI command.

If the received command is not a SCSI command, the CBW data error processing function (`usbf850_cbw_error`) is called to finish SCSI command processing.

(2) Judging NO DATA commands

If the length of data to transmit in the data phase (dCBWDataTransferLength) is 0x00000000, the received command is judged to be a NO DATA command.

If the received command is a NO DATA command, the NO DATA command processing function (`usbf850_no_data`) is called to execute the processing corresponding to the received command.

When command processing ends, the CSW response processing function (usbf850_csw_ret) is called to transmit the CSW.

(3) Judging the data transfer direction

If bit 7 of the transfer direction (bmCBWFlags) is 0, the received command is judged to be a write command, the data-out command processing function (`usbf850_data_out`) is called, and then processing corresponding to the received command is executed.

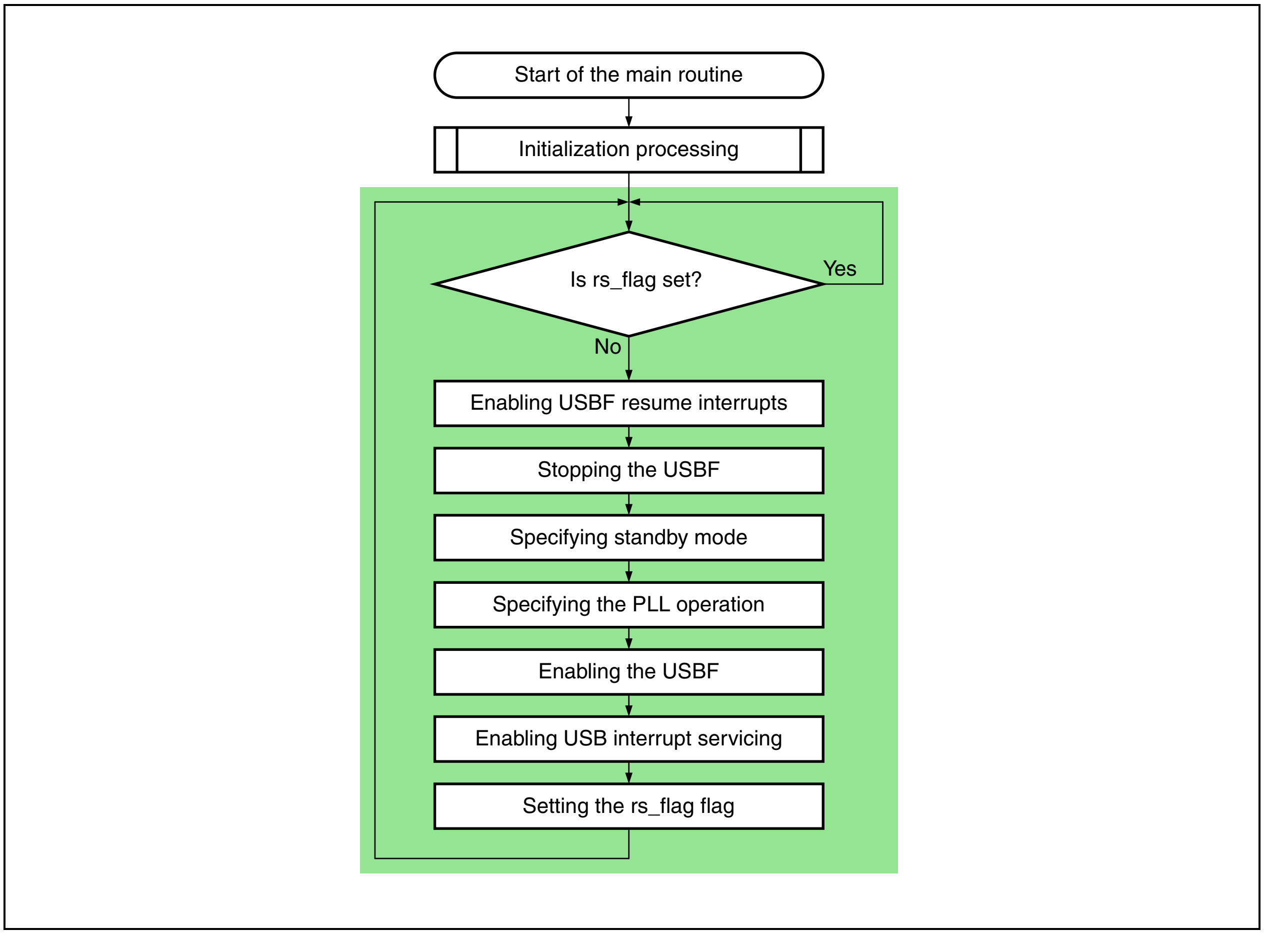
If bit 7 of bmCBWFlags is 1, the received command is judged to be a read command, the data-in command processing function (`usbf850_data_in`) is called, and then processing corresponding to the received command is executed.

When command processing ends, the CSW response processing function (`usbf850_csw_ret`) is called, and then the CSW is transmitted.

3.2.7 Suspend/resume processing

In the main routine, suspend/resume processing is performed according to the following sequence.

Figure 3-9. Suspend/Resume Processing Flowchart



(1) Monitoring the resume/suspend flag (rs_flag)

The resume/suspend flag (rs_flag) that is set by the sample driver is monitored. If this flag is set to SUSPEND (0x00), the USB bus is suspended.

(2) Enabling USBF resume interrupts

Only the USBF resume interrupt (INTUSB1) of the USBF is enabled.
The following registers are accessed:

(a) 0 is written to all valid bits of the UF0ICn registers (n = 0 to 4) to clear all interrupt sources.

(b) 0 is written to the UFIFn bits (n = 0, 1) of the UFICn registers to clear the USBF interrupt request signal.

(c) 0x0002 is written to the BRGINTE register to mask the interrupt sources indicated by the BRGINTT register, except the EPCINT1B interrupt source.

(d) 1 is written to the UFMK0 bit of the UFIC0 register and 0 is written to the UFMK1 bit of the UFIC1 register to disable INTUSB0 interrupt servicing and enable INTUSB1 interrupt servicing.

(3) Stopping the USBF

0x03 is written to the UFCKMSK register to disable the USBF.

(4) Specifying standby mode

The device is made to enter standby mode. The following registers are accessed:

(a) 0x03 is written to the PSMR register so that operation in software standby mode is in STOP mode.

(b) 0x02 is written to the PSC register according to the procedure for writing to special registers to specify standby mode.

(5) Specifying the PLL operation

0x03 is written to the PLLCTL register to specify PLL mode and start the PLL.

(6) Enabling the USBF

Operation of the USBF is specified. The following three registers are accessed:

(a) 0x02 is written to the UCKSEL register to supply the internal clock signal to the USBF.

(b) 0x00 is written to the UFCKMSK register to enable the USBF.

(c) 0x00 is written to the UHCKMSK register to enable the RAM dedicated to the data to use for the USBF (8 KB).

(7) Enabling USB interrupt servicing

The USBF interrupt (INTUSB0) of the USBF is enabled.
The following three registers are accessed:

(a) 0x00 is written to the UF0ICn registers (n = 0 to 4) to clear all interrupt sources.

(b) 0x0007 is written to the BRGINTE register to mask the interrupt sources indicated by the BRGINTT register, except the EPCINT0B to EPCINT2B interrupt sources.

(c) 0 is written to the UFMK0 bit of the UFIC0 register, and 1 is written to the UFMK1 bit of the UFIC1 register to disable INTUSB1 interrupt servicing and enable INTUSB0 interrupt servicing.

(8) Setting the resume/suspend flag (rs_flag)

The resume/suspend flag (rs_flag) is set to RESUME (0x01).

3.3 Function Specifications

This section describes the functions implemented in the sample driver.

3.3.1 Functions

The functions of each source file included in the sample driver are described below.

Table 3-40. Functions in the Sample Driver (1/2)

Source File	Function Name	Description
main.c	main	Main routine
	init	Initialization routine
	cpu_init	Initializes the CPU.
	romp_init	Initializes ROMization data.
usbf850.c	usbf850_init	Initializes the USBF.
	usbf850_intusbf0	Executes the INTUSBF0 interrupt handler.
	usbf850_intusbf1	Executes the INTUSBF1 interrupt handler.
	usbf850_data_send	Transmits USB data.
	usbf850_data_receive	Receives USB data.
	usbf850_standardreq	Processes standard requests.
	usbf850_getdesc	Processes GET_DESCRIPTOR requests.
	usbf850_sendnullEP0	Transmits a NULL packet for endpoint 0.
	usbf850_sendstallEP0	Performs a STALL response for endpoint 0.
	usbf850_bulkin1_stall	Controls the STALL response for bulk-in transfer.
	usbf850_bulkout1_stall	Controls the STALL response for bulk-out transfer.
	usbf850_sstall_ctrl	Controls the STALL responses for control transfer.
usbf850_storage.c	usbf850_blkonly_mass_storage_reset	Processes MASS_STORAGE_RESET requests.
	usbf850_max_lun	Processes GET_MAX_LUN requests.
	usbf850_setfunction_storage	Adds class request functions.
	usbf850_rx_cbw	Receives CBW data.
	usbf850_storage_cbwchk	Analyzes the CBW data commands.
	usbf850_cbw_error	Processes errors in CBW data.
	usbf850_no_data	Executes SCSI NO DATA commands.
	usbf850_data_in	Executes SCSI write commands.
	usbf850_data_out	Executes SCSI read commands.
	usbf850_csw_ret	Executes CSW responses.

Table 3-40. Functions in the Sample Driver (2/2)

Source File	Function Name	Description
scsi_cmd.c	scsi_command_to_ata	Executes SCSI commands.
	ata_test_unit_ready	Executes the TEST_UNIT_READY command.
	ata_seek	Executes the SEEK command.
	ata_start_stop_unit	Executes the START_STOP_UNIT command.
	ata_synchronize_cache	Executes the SYNCHRONIZE_CACHE command.
	ata_request_sense	Executes the REQUEST_SENSE command.
	ata_inquiry	Executes the INQUIRY command.
	ata_mode_select	Executes the MODE_SELECT6 command.
	ata_mode_select10	Executes the MODE_SELECT10 command.
	ata_mode_sense	Executes the MODE_SENSE6 command.
	ata_mode_sense10	Executes the MODE_SENSE10 command.
	ata_read_format_capacities	Executes the READ_FORMAT_CAPACITIES command.
	ata_read_capacity	Executes the READ_CAPACITY command.
	ata_read6	Executes the READ6 command.
	ata_read10	Executes the READ10 command.
	ata_write6	Executes the WRITE6 command.
	ata_write10	Executes the WRITE10 command.
	ata_verify	Executes the VERIFY command.
	ata_write_verify	Executes the WRITE_VERIFY command.
	ata_write_buff	Executes the WRITE_BUFFER command.
	scsi_to_usb	Transmits USB data (SCSI command).

3.3.2 Correlation of the functions

Some functions call other functions during the processing. The following figures show the correlation of the functions.

Figure 3-10. Functions Called During USB Interrupt Servicing

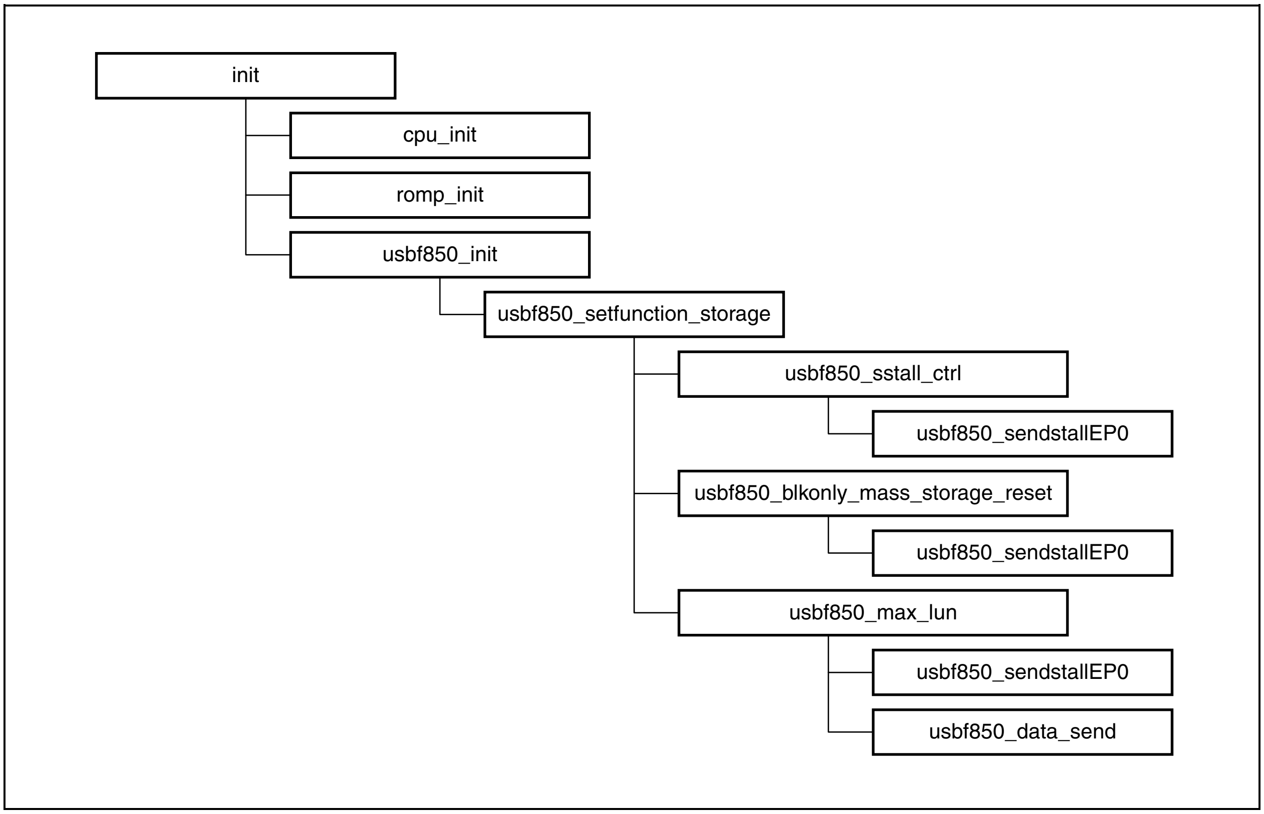


Figure 3-11. Calling Functions During USB Interrupt Servicing

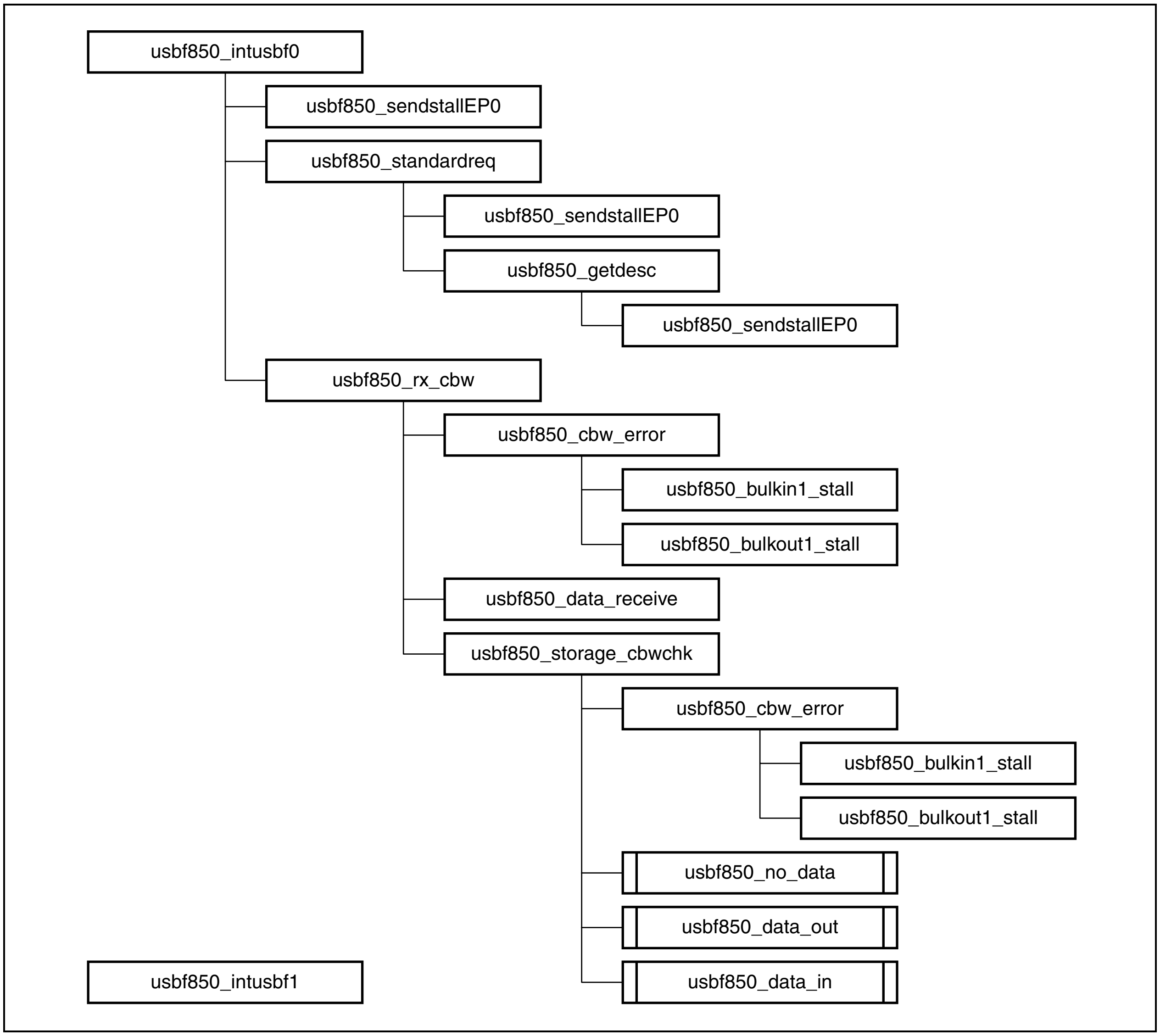


Figure 3-12. Calling Functions During CBW or CSW Processing

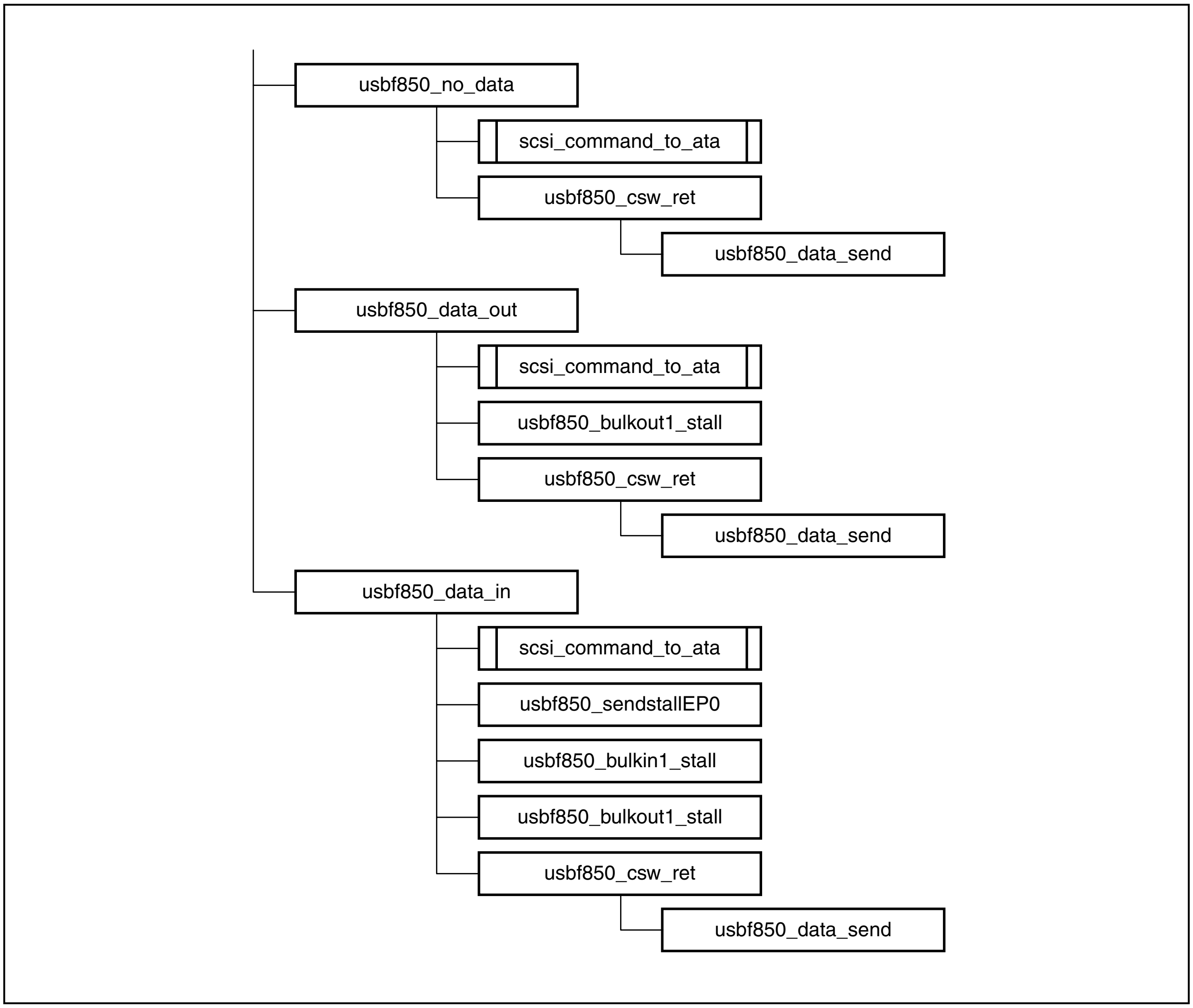
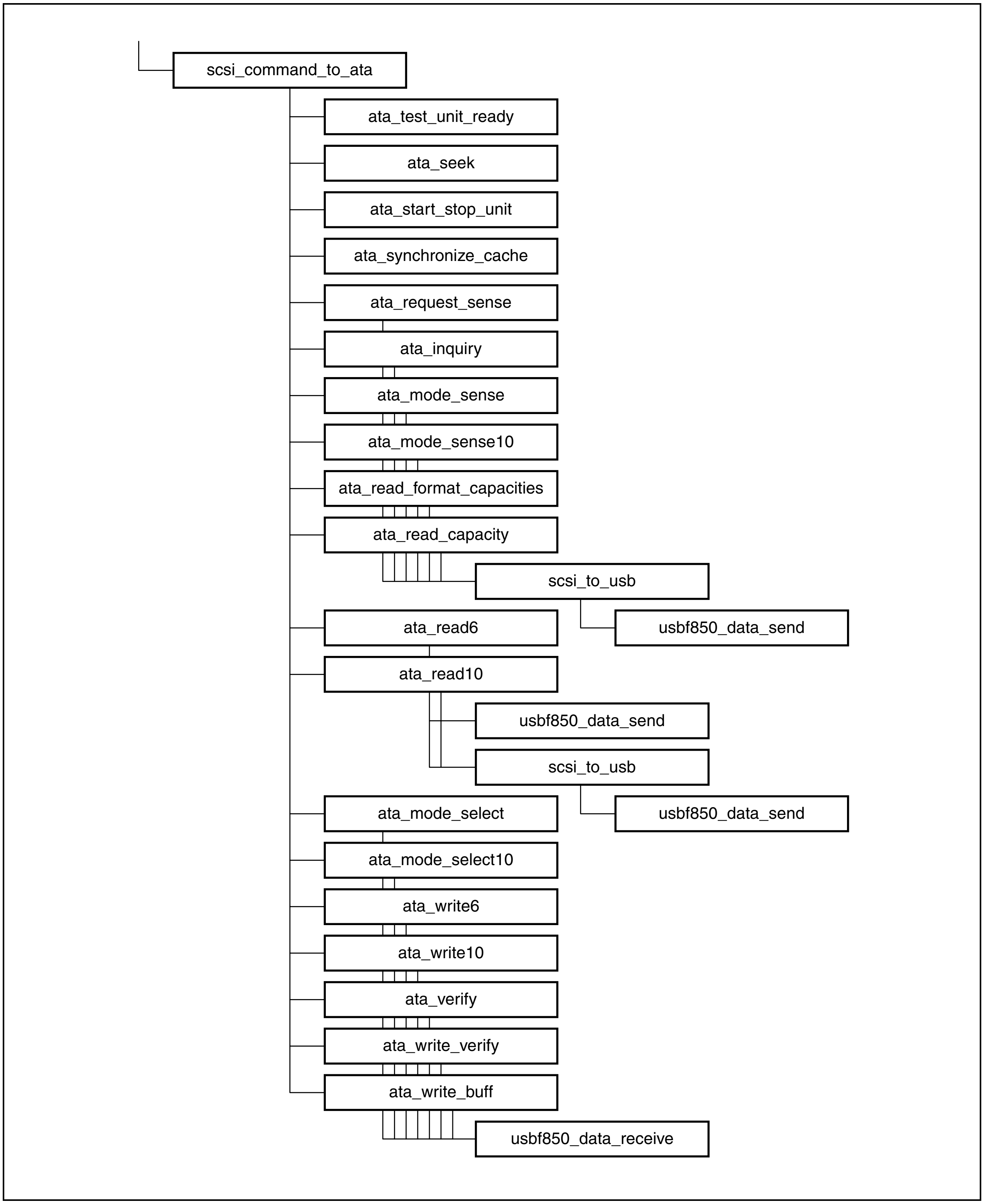


Figure 3-13. Calling Functions During SCSI Command Processing



3.3.3 Function features

This section describes the features of the functions implemented in the sample driver.

(1) Function description format

The functions are described in the following format.

Function name

[Overview]

An overview of the function is provided.

[C description format]

The format in which the function is written in C is provided.

[Parameters]

The parameters (arguments) of the function are described.

Parameter	Description
Parameter type and name	*Parameter summary*

[Return values]

The values returned by the function are described.

Symbol	Description
Return value type and name	*Return value summary*

[Description]

The feature of the function is described.

(2) Functions for the main routine

main

[Overview]

Main processing

[C description format]

```
void main(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called first when the sample driver is executed.
The resume/suspend flag (rs_flag) is monitored after the initialization function (`init`) is called. Suspend processing is performed when the rs_flag is set to SUSPEND (0x00).

init

[Overview]

Initialization processing

[C description format]

```
void init(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called in the main routine.
The CPU initialization processing function (`cpu_init`), ROMization package initialization processing function (`romp_init`), and then the USBF initialization processing function (`usbf850_init`) are called.

cpu_init

[Overview]

Initializes the CPU.

[C description format]

`void cpu_init(void)`

[Parameters]

None

[Return values]

None

[Description]

This function is called during initialization.

The settings that are necessary to use the USBF in the V850ES/Jx3-H or V850ES/Jx3-U, such as the number of wait cycles, clock frequency, and operation mode when accessing the bus, are specified.

romp_init

[Overview]

ROMization package initialization processing

[C description format]

`void romp_init(void)`

[Parameters]

None

[Return values]

None

[Description]

This function is called during initialization.

The copy function (`_rcopy`) is called and the information stored at the specified address is copied to the RAM area byte by byte.

(3) Functions for the USBF

usbf850_init

[Overview]

Initializes the USBF.

[C description format]

```
void usbf850_init(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during initialization processing.
This function specifies the settings required for using the USBF, such as allocating and specifying the data area, and masking interrupt requests.

usbf850_intusbf0

[Overview]

Executes the INTUSBF0 interrupt handler.

[C description format]

```
void usbf850_intusbf0(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called as the USB interrupt (INTUSBF0) handler.
The statuses of the endpoint for control transfer (endpoint 0) and the endpoint for bulk-out transfer (reception) (endpoint 2) are monitored and processing corresponding to the received requests and commands is performed.
The RSUSPD, BUSRST, SETRQ, and CPUDEC interrupts are monitored at endpoint 0. If a CPUDEC interrupt occurs, the request data is decoded and response processing is performed by calling the corresponding function.
The BKO1DT interrupt is monitored at endpoint 2. If a BKO1DT interrupt occurs, the CBW data reception function (`usbf850_rx_cbw`) is called and processing corresponding to the command is performed.

usbf850_intusbf1

[Overview]

Executes the INTUSBF1 interrupt handler.

[C description format]

```
void usbf850_intusbf1(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called as the USB resume interrupt (INTUSBF1) handler.

The resume/suspend flag (rs_flag) is set to RESUME (0x01).

usbf850_data_send

[Overview]

Transmits USB data.

[C description format]

```
INT32 usbf850_data_send(UINT8 *data, INT32 len, INT8 ep)
```

[Parameters]

Parameter	Description
UINT8 *data	Transmission data buffer pointer
INT32 len	Transmission data length
INT8 ep	Transmission data length

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function stores the data stored in the transmission data buffer into the FIFO for the specified endpoint, byte by byte.

usbf850_data_receive

[Overview]

Receives USB data.

[C description format]

```
INT32 usbf850_data_receive(UINT8 *data, INT32 len, INT8 ep)
```

[Parameters]

Parameter	Description
UINT8 *data	Reception data buffer pointer
INT32 len	Reception data length
INT8 ep	Data reception endpoint number

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function reads data from the FIFO for the specified endpoint byte by byte and stores the data into the reception data buffer.

usbf850_standardreq

[Overview]

Processes standard requests to which the USBF does not automatically respond.

[C description format]

```
void usbf850_standardreq(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called when endpoint 0 is monitored.
If a GET_DESCRIPTOR request is decoded, this function calls the GET_DESCRIPTOR request processing function (`usbf850_getdesc`). For other requests, this function calls the function for processing STALL responses for endpoint 0 (`usbf850_sendstallEP0`).

usbf850_getdesc

[Overview]

Processes GET_DESCRIPTOR requests.

[C description format]

```
void usbf850_getdesc(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during the processing of standard requests to which the USBF does not automatically respond.
If a decoded request requests a string descriptor, this function calls the USB data transmission function (`usbf850_data_send`) and transmits a string descriptor from endpoint 0. If a decoded request requests any other descriptor, this function calls the function for processing STALL responses for endpoint 0 (`usbf850_sendstallEP0`).

usbf850_sendnullEP0

[Overview]

Transmits a NULL packet for endpoint 0.

[C description format]

```
void usbf850_sendnullEP0(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function clears the FIFO for endpoint 0 and transmits a NULL packet from the USBF by setting the bit that indicates the end of data to 1.

usbf850_sendstallEP0

[Overview]

Performs a STALL response for endpoint 0.

[C description format]

```
void usbf850_sendstallEP0(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function makes the USBF perform a STALL response by setting the bit that indicates the use of STALL handshaking to 1.

usbf850_bulkin1_stall

[Overview]

Controls the STALL response for bulk-in transfer.

[C description format]

```
void usbf850_bulkin1_stall(void)
```

[Parameters]

None

[Return values]

None

[Description]

By setting the EP0NKA bit of the UF0E0NA register to 1, endpoint 0 of the USBF is set up to return a NAK response.
By clearing the FIFOs for endpoint 1 and setting the E1HALT bit of the UF0E1SL register to 1, endpoint 1 of the USBF issues a STALL response.
When the STALL response ends, the NAK response of endpoint 0 is disabled.

usbf850_bulkout1_stall

[Overview]

Controls the STALL response for bulk-out transfer.

[C description format]

```
void usbf850_bulkout1_stall(void)
```

[Parameters]

None

[Return values]

None

[Description]

By setting the EP0NKA bit of the UF0E0NA register to 1, endpoint 0 of the USBF is set up to return a NAK response.
By clearing the FIFOs for endpoint 2 and setting the E2HALT bit of the UF0E2SL register to 1, endpoint 2 of the USBF issues a STALL response.
When the STALL response ends, the NAK response of endpoint 0 is disabled.

usbf850_sstall_ctrl

[Overview]

Controls STALL responses.

[C description format]

```
void usbf850_sstall_ctrl(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function calls the STALL response processing function for endpoint 0 (usbf850_sendstallEP0).

(4) Functions for USB mass storage class processing

usbf850_blkonly_mass_storage_reset

[Overview]

Processes MASS_STORAGE_RESET requests.

[C description format]

```
void usbf850_blkonly_mass_storage_reset(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function clears the FIFOs of endpoints 1 and 2 and then sets up these endpoints to issue a STALL response. Next, endpoint 0 transmits a NULL packet.

usbf850_max_lun

[Overview]

Processes GET_MAX_LUN requests.

[C description format]

```
void usbf850_max_lun(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function transmits the number of logical units of the MSC device.

usbf850_setfunction_storage

[Overview]

Adds class request functions.

[C description format]

```
void usbf850_setfunction_storage(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function adds the address of each class request function.

usbf850_rx_cbw

[Overview]

Receives CBW data.

[C description format]

```
void usbf850_rx_cbw(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function reads CBW data from the FIFOs of the bulk-in endpoint (endpoint 2) and then calls the CBW data command analysis processing function (`usbf850_storage_cbwchk`).

usbf850_storage_cbwchk

[Overview]

Analyzes the CBW data commands.

[C description format]

```
INT32 usbf850_storage_cbwchk(void)
```

[Parameters]

None

[Return values]

The status when the CBW was checked is returned.

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function analyzes the CBW data, judges the command type (NO DATA, data-in (write), data-out (read)), and then executes each command.

usbf850_cbw_error

[Overview]

Processes errors in CBW data.

[C description format]

```
void usbf850_cbw_error(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function sets up the bulk-in endpoint (endpoint 1) and bulk-out endpoint (endpoint 2) to issue a STALL response.

usbf850_no_data

[Overview]

Executes SCSI NO DATA commands.

[C description format]

```
void usbf850_no_data(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function executes a NO DATA command and then transmits the result in CSW format.

usbf850_data_in

[Overview]

Executes SCSI write commands.

[C description format]

```
void usbf850_data_in(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function executes a data-in (write) command and then transmits the result in CSW format.

usbf850_data_out

[Overview]

Executes SCSI data-out commands.

[C description format]

```
void usbf850_data_out(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function executes a data-out (read) command and then transmits the result in CSW format.

usbf850_csw_ret

[Overview]

Processes CSW responses.

[C description format]

```
INT32 usbf850_csw_ret(UINT8 status)
```

[Parameters]

Parameter	Description
UINT8 status	Command processing result

[Return values]

CSW transmission processing result

Symbol	Description
DEV_OK	Normal completion

[Description]

This function creates CSW format data from the processing result and then transmits the data via USB.

(5) SCSI command processing functions

scsi_command_to_ata

[Overview]

Executes SCSI commands.

[C description format]

```
INT32 scsi_command_to_ata(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize,
INT32 TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

The results of executing SCSI commands are returned.

Parameter	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The execution result of a command is other than the above statuses or a request is invalid.

[Description]

The SCSI command type is judged and then processing for the command is executed.
If there are no corresponding commands, the sense data is updated assuming the sense keys to be invalid requests.

ata_test_unit_ready

[Overview]

Executes the TEST_UNIT_READY command.

[C description format]

```
INT32 ata_test_unit_ready(INT32 TransFlag)
```

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.

[Description]

This function clears the sense data (sense key = 0x00). If the bulk transfer direction of the above command is not NO DATA, the sense data is updated assuming the sense key to be an invalid request.

ata_seek

[Overview]

Executes the SEEK command.

[C description format]

```
INT32 ata_seek(INT32 TransFlag)
```

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.

[Description]

This function clears the sense data (sense key = 0x00). If the bulk transfer direction of the above command is not NO DATA, the sense data is updated assuming the sense key to be an invalid request.

ata_start_stop_unit

[Overview]

Executes the START_STOP_UNIT command.

[C description format]

```
INT32 ata_start_stop_unit(INT32 TransFlag)
```

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return values]

Processing result

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.

[Description]

This function clears the sense data (sense key = 0x00). If the bulk transfer direction of the above command is not NO DATA, the sense data is updated assuming the sense key to be an invalid request.

ata_synchronize_cache

[Overview]

Executes the SYNCHRONIZE_CACHE command.

[C description format]

```
INT32 ata_synchronize_cache(INT32 TransFlag)
```

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return values]

Processing result

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.

[Description]

This function clears the sense data (sense key = 0x00). If the bulk transfer direction of the above command is not NO DATA, the sense data is updated assuming the sense key to be an invalid request.

ata_request_sense

[Overview]

Executes the REQUEST_SENSE command.

[C description format]

```
INT32 ata_request_sense(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.
DEV_ERR_READ	A transfer direction error occurred for a read command.

[Description]

This function transmits the sense data.

If the data size is 0 and the transfer direction is not NO DATA, the sense data is updated assuming the sense key to be an invalid request.

ata_inquiry

[Overview]

Executes the INQUIRY command.

[C description format]

```
INT32 ata_inquiry(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits inquiry data. If the CMDDT and EVPD bits of command byte 1 are both “1”, the sense data is updated assuming the sense key to be an invalid request.

ata_mode_select

[Overview]

Processes the MODE_SELECT(6) command.

[C description format]

```
INT32 ata_mode_select(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then updates the MODE_SELECT data table by using the received data.

If the transfer direction or data size is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_mode_select10

[Overview]

Executes the MODE_SELECT(10) command.

[C description format]

```
INT32 ata_mode_select10(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then updates the MODE_SELECT(10) data table by using the received data.

If the transfer direction or data size is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_mode_sense

[Overview]

Executes the MODE_SENSE(6) command.

[C description format]

```
INT32 ata_mode_sense(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits MODE_SENSE data.

ata_mode_sense10

[Overview]

Executes the MODE_SENSE(10) command.

[C description format]

```
INT32 ata_mode_sense10(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits MODE_SENSE(10) data.

ata_read_format_capacities

[Overview]

Executes the READ_FORMAT_CAPACITIES command.

[C description format]

```
INT32 ata_read_format_capacities(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32
lDataSize, INT32 TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits READ_FORMAT_CAPACITIES data.

ata_read_capacity

[Overview]

Executes the READ_CAPACITY command.

[C description format]

```
INT32 ata_read_capacity(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits READ_CAPACITY data.

ata_read6

[Overview]

Executes the READ(6) command.

[C description format]

```
INT32 ata_read6(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits the data read from the data area.
The address from which to start reading data is calculated using the LBA (local block address) of the SCSI command and the block size.
If the transfer direction, SCSI command flag, or link bit is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_read10

[Overview]

Executes the READ(10) command.

[C description format]

```
INT32 ata_read10(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then transmits the data read from the data area.
The address from which to start reading data is calculated using the LBA (local block address) of the SCSI command and the block size.
If the transfer direction, SCSI command flag, or link bit is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_write6

[Overview]

Executes the WRITE6 command.

[C description format]

```
INT32 ata_write6(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then writes the received data to the data area.
The address from which to start writing the data is calculated using the LBA (local block address) of the SCSI command and the block size.
If the transfer direction, SCSI command flag, or link bit is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_write10

[Overview]

Executes the WRITE10 command.

[C description format]

```
INT32 ata_write10(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then writes the received data to the data area.
The address from which to start writing the data is calculated using the LBA (local block address) of the SCSI command and the block size.
If the transfer direction, SCSI command flag, or link bit is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_verify

[Overview]

Executes the VERIFY command.

[C description format]

```
INT32 ata_verify(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_NODATA	A transfer direction error occurred for a NO DATA command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function writes the received data to the data area.

The address from which to start writing the data is calculated using the LBA (local block address) of the SCSI command and the block size.

If the transfer direction or the BYTCHK bit of a SCSI command is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_write_verify

[Overview]

Executes the WRITE_VERIFY command.

[C description format]

```
INT32 ata_write_verify(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00) and then writes the received data to the data area.
The address from which to start writing the data is calculated using the LBA (local block address) of the SCSI command and the block size.
If the transfer direction, SCSI command flag, or link bit is invalid, the sense data is updated assuming the sense key to be an invalid request.

ata_write_buff

[Overview]

Executes the WRITE_BUFF command.

[C description format]

```
INT32 ata_write_buff(UINT8 *ScsiCommandBuf, UINT8 *pbData, INT32 lDataSize, INT32
TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	SCSI command storage buffer pointer
UINT8 *pbData	Command data storage buffer pointer
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return value]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_WRITE	A transfer direction error occurred for a write command.
DEV_ERROR	The status of a command is other than the above or a request is invalid.

[Description]

This function clears the sense data (sense key = 0x00), and then reads and discards the received data.

scsi_to_usb

[Overview]

Transmits USB data (SCSI command).

[C description format]

```
INT32 scsi_to_usb(UINT8 *pbData, INT32 TransFlag)
```

[Parameters]

Parameter	Description
UINT8 *pbData	Command data storage buffer pointer
INT32 TransFlag	Direction of data transfer

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERR_READ	A transfer direction error occurred for a read command.

[Description]

This function calls the USB data transmission processing function (usbf850_data_send) to transmit data from the bulk-out endpoint (endpoint 1).

If the transfer direction is invalid, the sense data is updated assuming the sense key to be an invalid request.

3.4 Data Structures

The sample driver uses the following structures:

(1) USB device request structure

This structure is defined in `usbf850.h`.

```
typedef struct {
      UINT8  RequstType;   /*bmRequestType */
      UINT8  Request;      /*bRequest      */
      UINT16 Value;        /*wValue        */
      UINT16 Index;        /*wIndex        */
      UINT16 Length;       /*wLength       */
      UINT8* Data;         /*index to Data */
} USB_SETUP;
```

(2) CBW data structure

This structure is defined in `Types.h`.

```
typedef struct {    /* CBW(Command Block Wrapper) DATA */
      UINT8   dCBWSignature[4];            /* Signature */
      UINT8   dCBWTag[4];                  /* Tag */
      UINT8   dCBWDataTransferLength[4];   /* Transfer data length */
      UINT8   bmCBWFlags;                  /* Defines the transfer direction
(OUT, IN, or NO DATA) */
      UINT8   bCBWLUN;                     /* Target device number */
      UINT8   bCBWCBLength;                /* Number of valid bytes of CBWCB */
      UINT8   CBWCB[16];                   /* CBWCB (command) */
} CBW_INFO,*PCBW_INFO;
```

(3) CSW data structure

This structure is defined in `Types.h`.

```
typedef struct {    /* CSW(Command Status Wrapper) DATA */
      UINT8   dCSWSignature[4];            /* Signature */
      UINT8   dCSWTag[4];                  /* Tag */
      UINT8   dCSWDataResidue[4];          /* Difference between the specified
transfer data length and length of processed data */
      UINT8   bmCSWStatus;                 /* Processing result status*/
} CSW_INFO,*PCSW_INFO;
```

(4) SCSI sense data structure

This structure is defined in `scsi_cmd.c`.

```
typedef struct _SCSI_SENSE_DATA {
      UINT8  sense_key;
      UINT8  asc;
      UINT8  ascq;
} SCSI_SENSE_DATA, *PSCSI_SENSE_DATA;
```

CHAPTER 4 DEVELOPMENT ENVIRONMENT

This chapter provides an example of creating an environment for developing an application program that uses the USB communication device class sample driver for the V850ES/Jx3-H and the procedure for debugging the application.

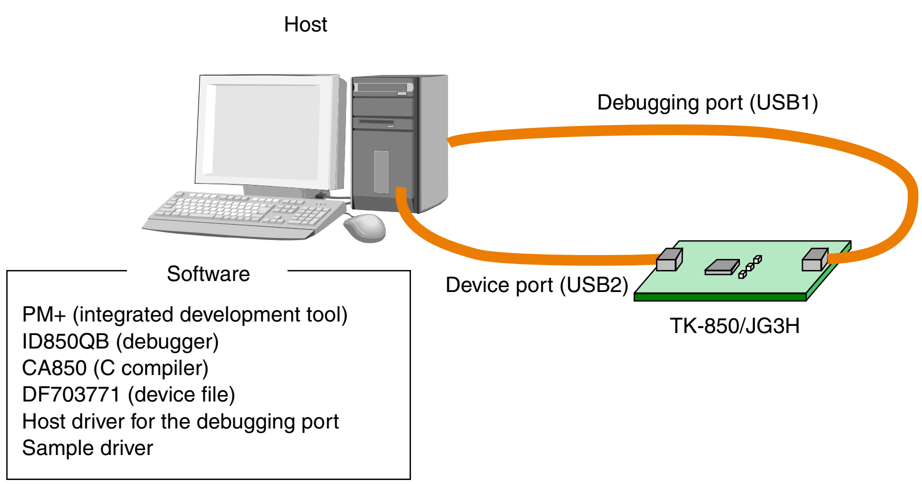
4.1 Used Products

This section describes the used hardware and software tool products.

4.1.1 System components

Figure 4-1 shows the components used in a system that uses the sample driver.

Figure 4-1. System Components Used in the Development Environment



Remark For a drawing and details about ports of the TK-850/JG3H, see **APPENDIX A STARTER KIT**.

4.1.2 Program development

The following hardware and software are necessary to develop a system that uses the sample driver:

Table 4-1. Example of the Components Used in a Program Development Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT™-compatible computer using Windows™ XP or Windows Vista™
Software	Integrated development tool	PM+	V6.31
	Compiler	CA850	W3.20
	Device file	DF703771	For the V850ES/Jx3-H and V850ES/Jx3-U
	Source files	Sample driver	
	Include files		

4.1.3 Debugging

The following hardware and software are necessary to debug a system that uses the sample driver:

Table 4-2. Example of the Components Used in a Debugging Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT-compatible computer using Windows XP or Windows Vista
	Target device	TK-850/JG3H or TK-850/JH3U-SP	Tessera Technology, Inc.
	2 USB cables	–	miniB-to-A connector cable
Software	Integrated development tool	PM+	V6.31
	Debugger	ID850QB	V3.50
	Flash memory programming tool	ID850QB	V3.50
Files	Device file	DF703771	For the V850ES/Jx3-H and V850ES/Jx3-U
	Host driver for the debugging port	Included with the TK-850/JG3H or TK-850/JH3U-SP	**Note 1**
	Source files	Sample driver	
	Include files		
	Project files		**Note 2**

Notes 1. For details about products and how to obtain them, contact NEC Electronics.

2. A file that is used when creating a system using PM+ is included with the sample driver.

4.2 Setting Up the Environment

This section describes the preparations required for developing and debugging a system by using the products described in **4.1 Used Products**.

4.2.1 Preparing the host environment

Create a dedicated workspace on the host for debugging.

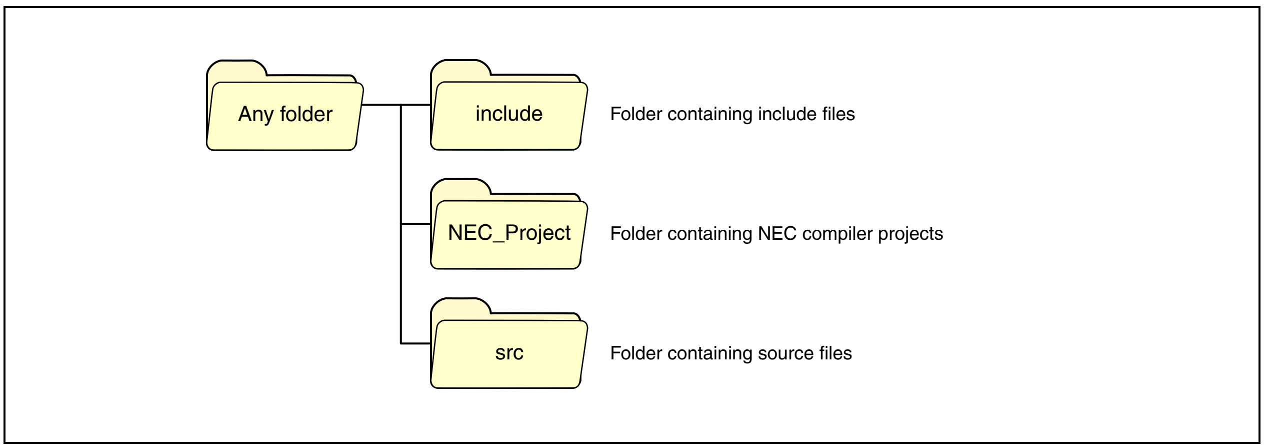
(1) Installing an integrated development tool

Install PM+. For details, see the **PM+ User's Manual**.

(2) Downloading drivers

Store the set of files provided with the sample driver in any directory without changing the folder structure.
Store the host driver for the debugging port in any directory.

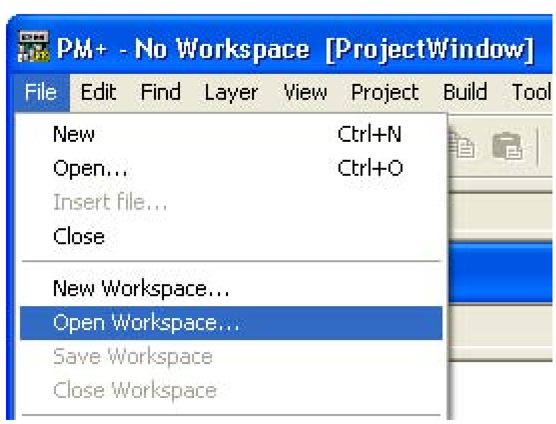
Figure 4-2. Folder Structure of the Sample Driver



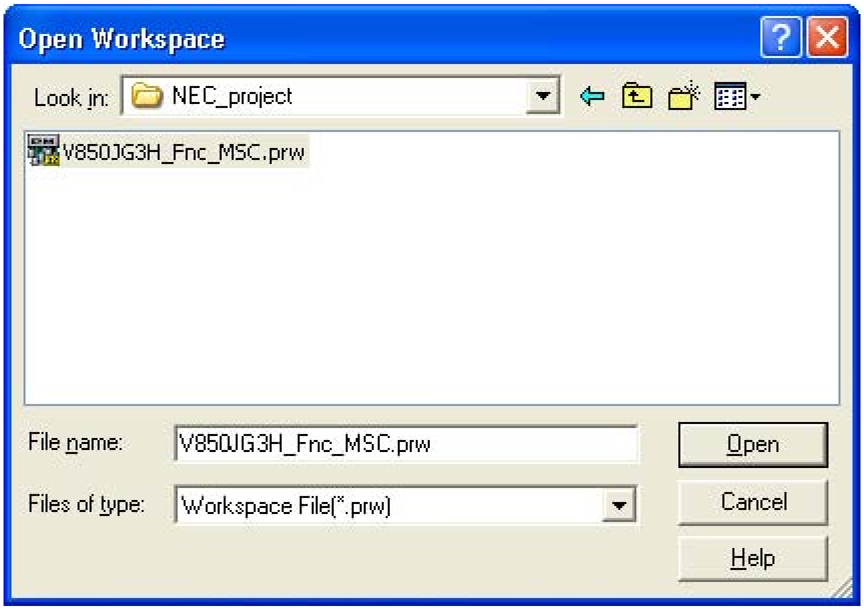
(3) Setting up the workspace

The procedure for using project files included with the sample driver is described below.

<1> Start PM+, and then select **Open Workspace** in the **File** menu.



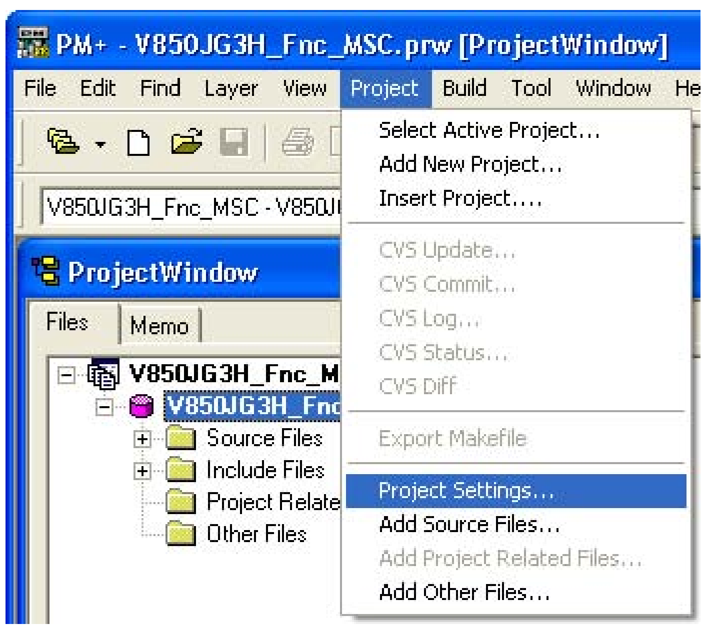
<2> In the **Open Workspace** dialog box, specify the workspace file in the NEC_project folder, which is the sample driver installation directory.



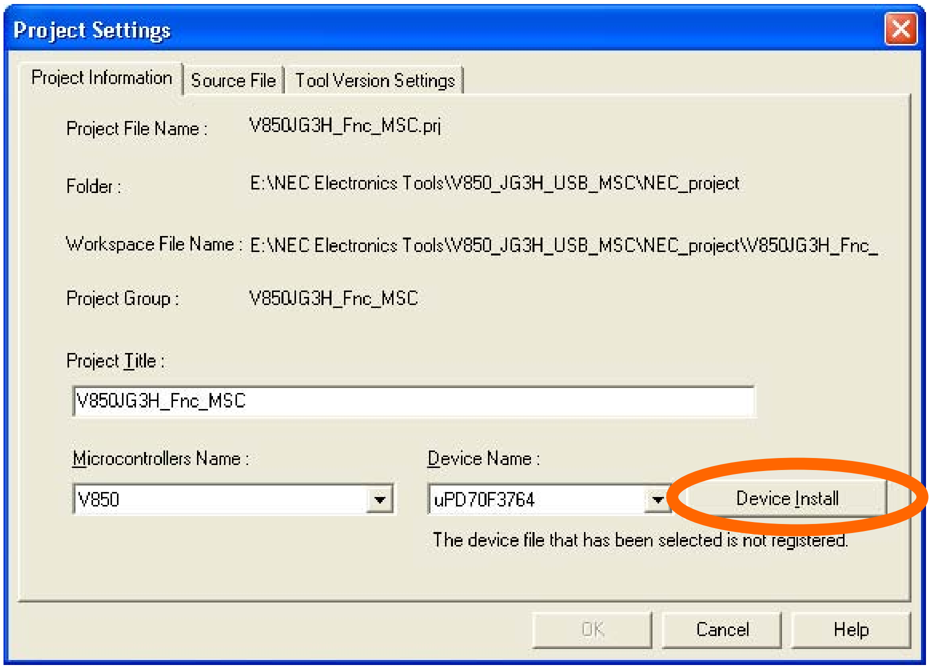
(4) Installing a device file

The procedure for using a device file for the V850ES/Jx3-H is described below.

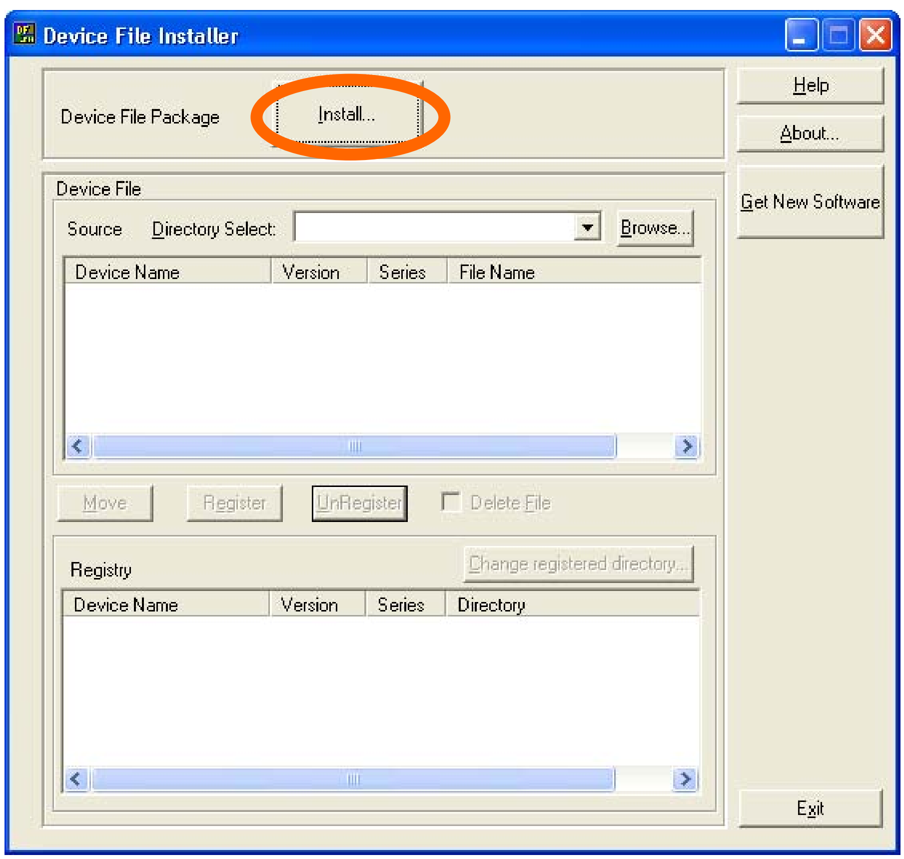
<1> Select **Project Settings** in the PM+ **Project** menu.



<2> In the **Project Settings** dialog box, click the **Device Install** button on the **Project Information** tab to start the Device File Installer.

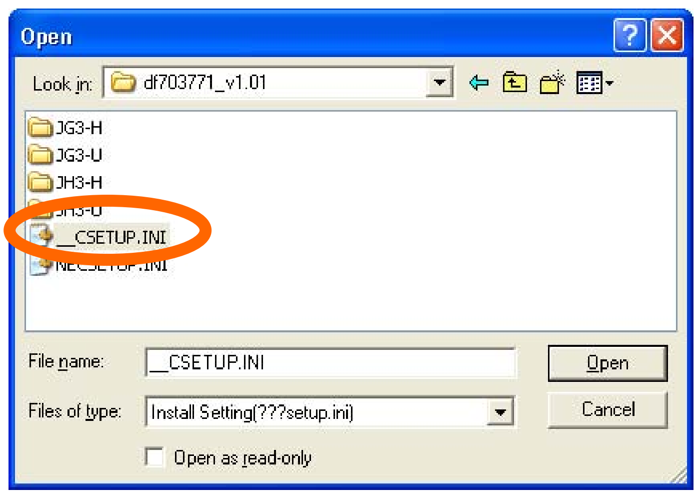


<3> In the **Device File Installer** dialog box, click the **Install** button to start the installation wizard.



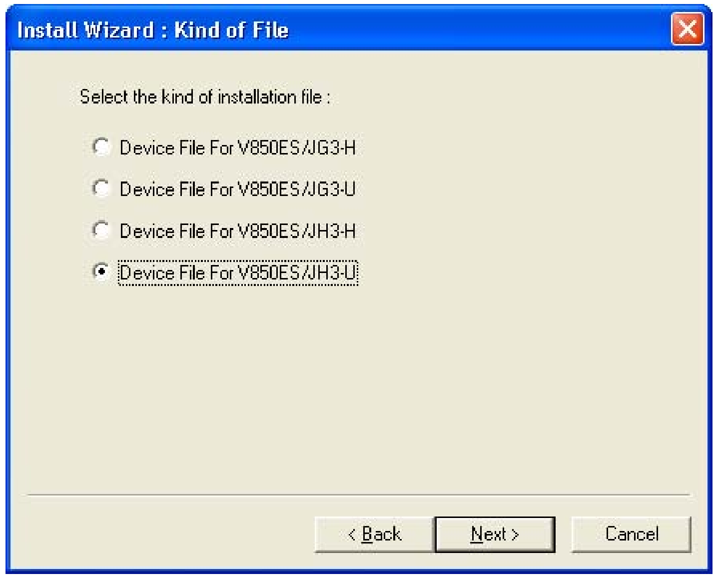
<4> In the **Install Information File** dialog box, click the **Browse** button.

<5> In the **Open** dialog box, open the directory in which the device file was stored, select *__CSETUP.INI*, and then click the **Open** button.

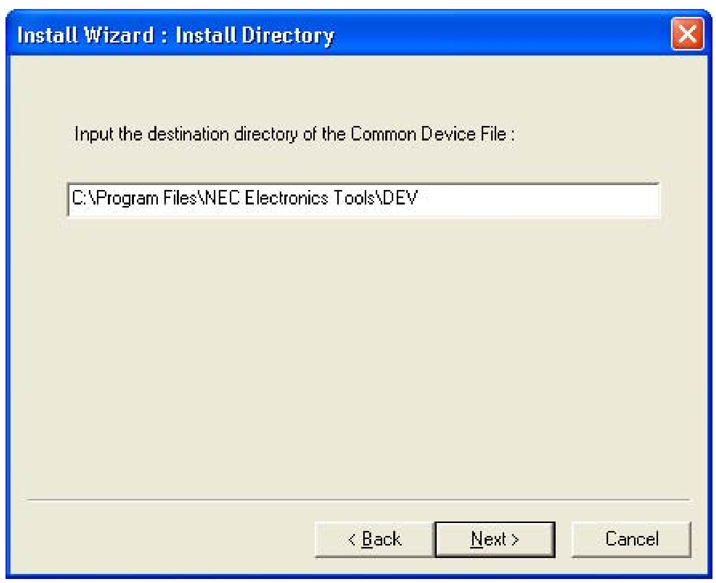


<6> In the **Install Information File** dialog box, click the **Next** button.
In the **NEC SOFTWARE LICENSE AGREEMENT** dialog box, read the license agreement, and then click the **Agree** button if you agree with the terms.

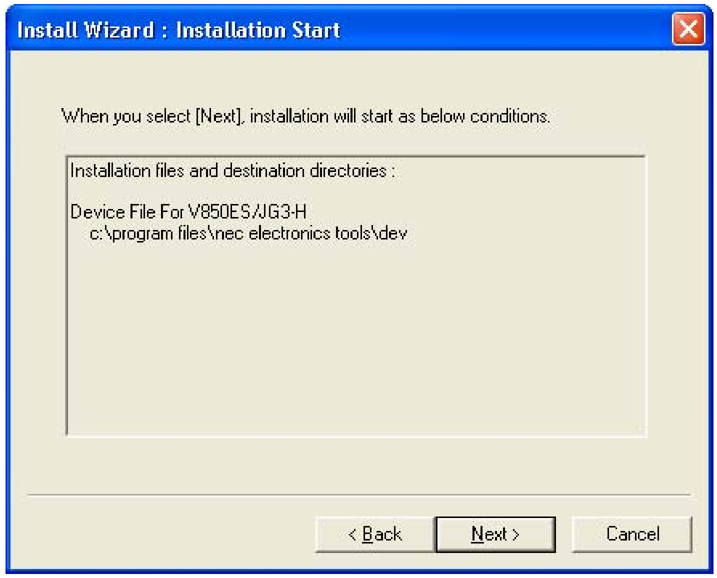
<7> In the **Kind of File** dialog box, select the device file to install, and then click the **Next** button.



<8> In the **Install Directory** dialog box, confirm that a path is displayed, and then click the **Next** button.



<9> In the **Installation Start** dialog box, click the **Next** button.



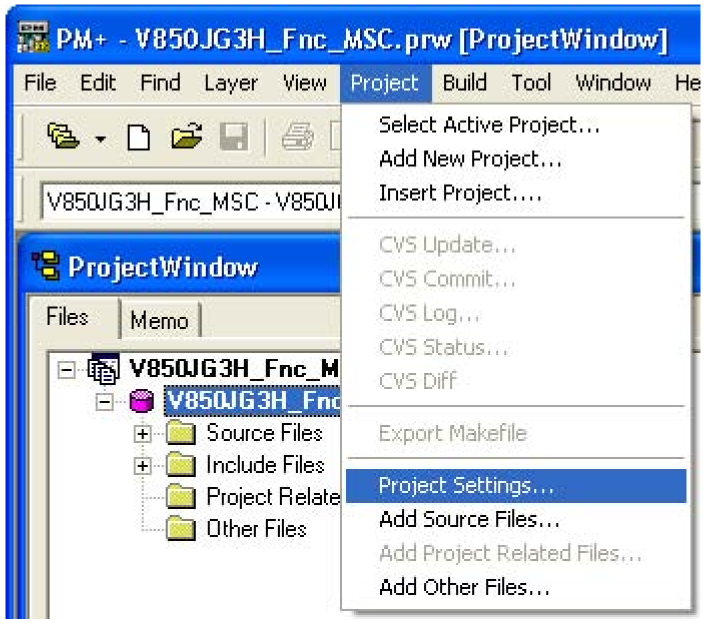
<10> The device file is installed to the project. This might take a while depending on the environment.

<11> In the **Installation Finished** dialog box, click the **Finish** button.

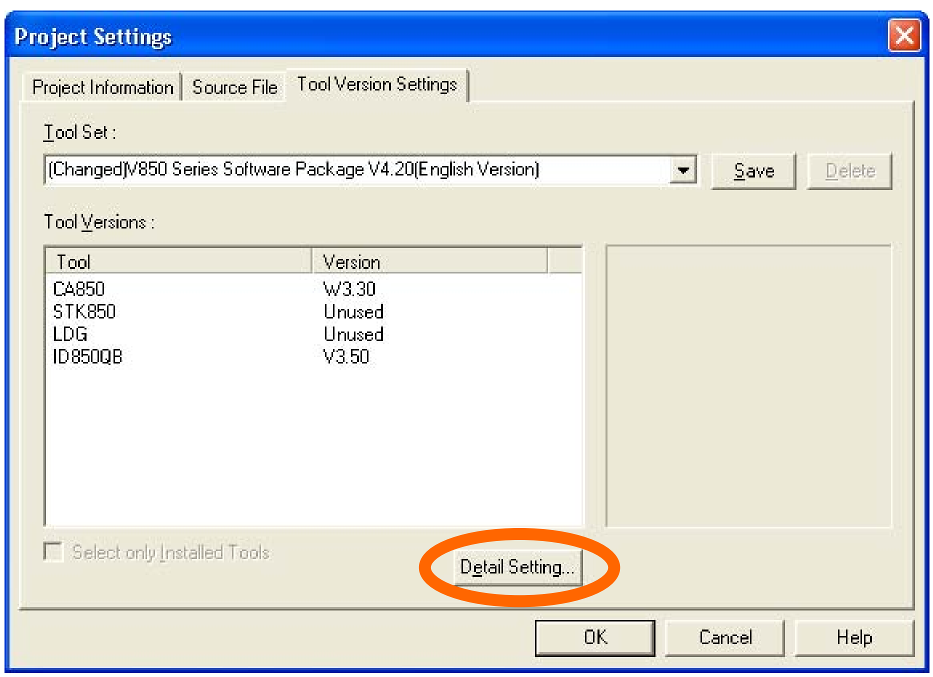
(5) Setting up the building tool

The procedure for using the CA850 as the building tool and the ID850QB as the debugging tool is described below.

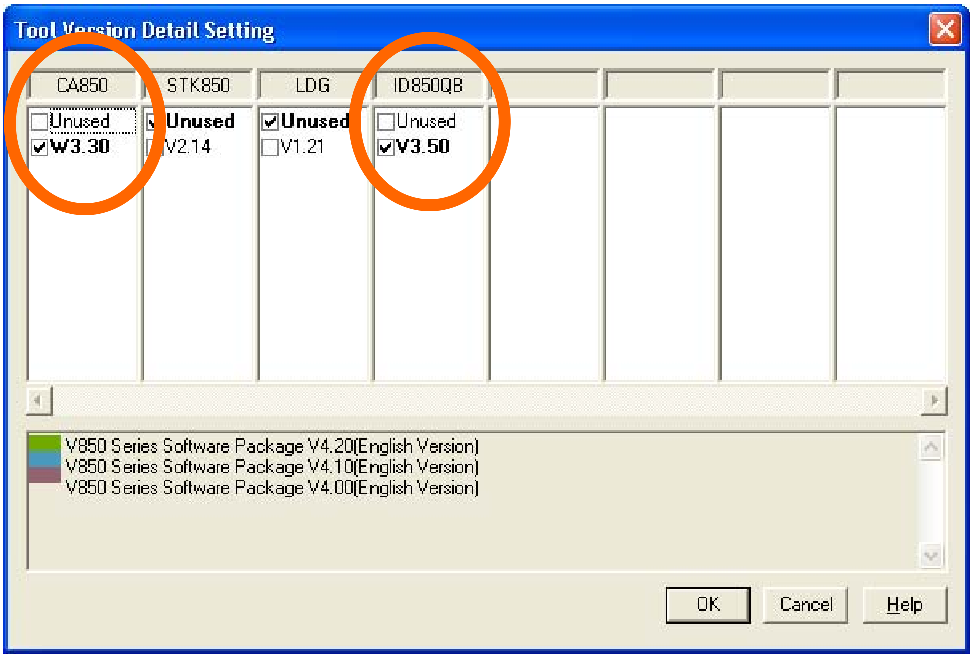
<1> Select **Project Settings** in the PM+ **Project** menu.



<2> In the **Project Settings** dialog box, click the **Detail Setting** button on the **Tool Version Settings** tab.



<3> In the **Tool Version Detail Setting** dialog box, select the compiler version to use in the **CA850** column and the debugger version to use in the **ID850QB** column.



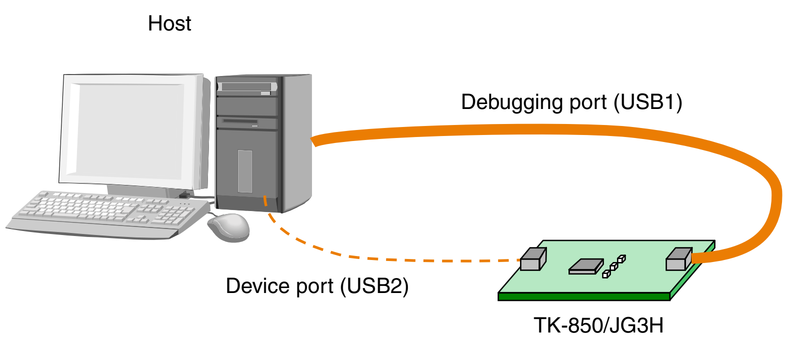
4.2.2 Setting up the target environment

Connect the target device to use for debugging.

(1) Connecting the debugging port

Connect the debugging port of the TK-850/JG3H (USB1) to a USB port of the host by using a USB cable.

Figure 4-3. Connecting the Debugging Port of the TK-850/JG3H



Remark The device port (USB2) is not connected in this figure.

(2) Installing the host driver

A driver must be installed to connect the TK-850/JG3H to the host by using the debugging port (USB1) or device port (USB2).

(a) Debugging port (USB1)

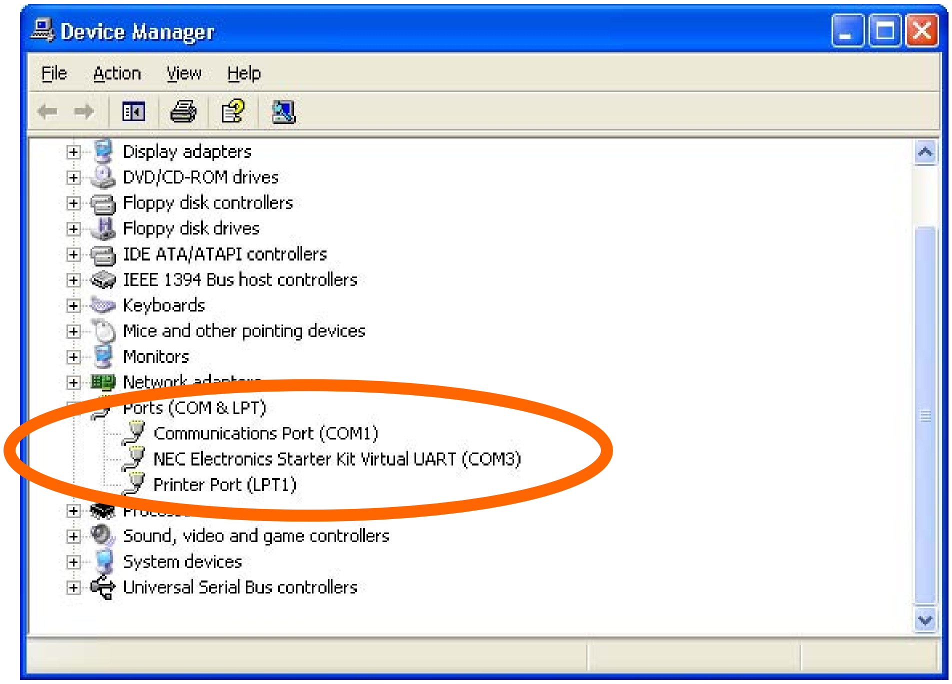
The USB port on the TK-850/JG3H (USB1) is a debugging port. A dedicated host driver is necessary to use this port. For details about how to install the driver, see the **TK-850/JG3H User's Manual**.

(b) Device port (USB2)

The mass storage class host driver, which is a standard Windows driver, is used for the device port. For details, see **4.4 Operation Check**.

(3) Checking the device assignment

Open the Windows **Device Manager** window. Confirm that **NEC Electronics Starter Kit Virtual UART (COM3)**[Note] is displayed in the **Ports (COM & LPT)** category.



Note The name of the displayed driver varies depending on the installed driver. For details, see the **TK-850/JG3H User's Manual**.

4.3 On-Chip Debugging

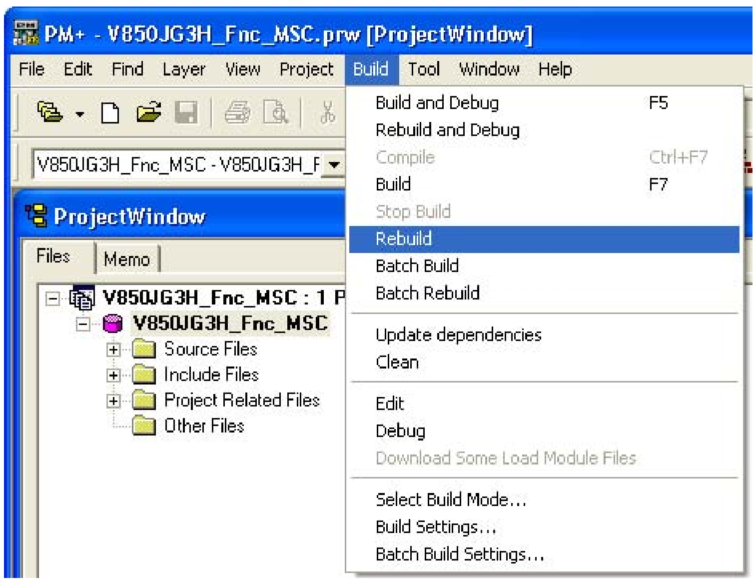
This section describes the procedure for debugging an application program that was developed using the workspace described in **4.2 Setting Up the Environment**.

For the V850ES/Jx3-H, a program can be written to its internal flash memory and the program operation can be checked by directly executing the program by using a debugger (on-chip debugging).

4.3.1 Generating a load module

To write a program to the target device, use a C compiler to generate a load module by converting a file written in C or assembly language.

For PM+, generate a load module by selecting **Rebuild** in the **Build** menu.



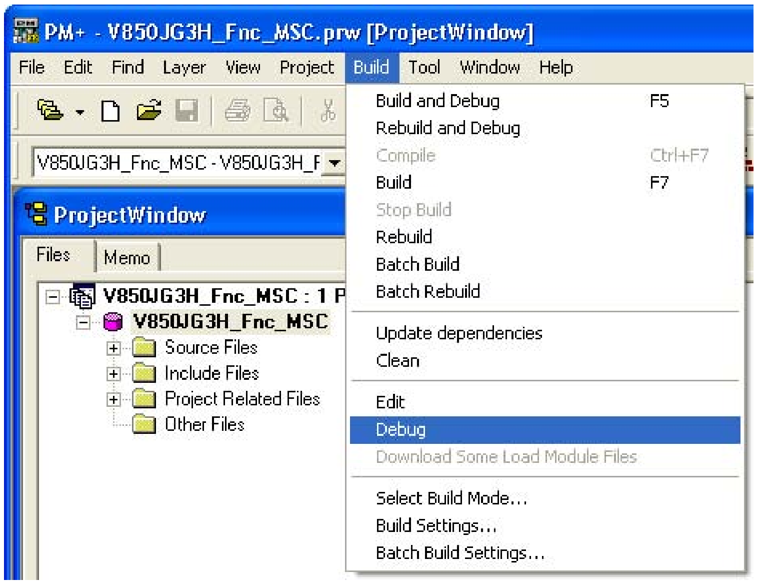
4.3.2 Loading and executing the load module

Execute the generated load module by writing (loading) it to the target.

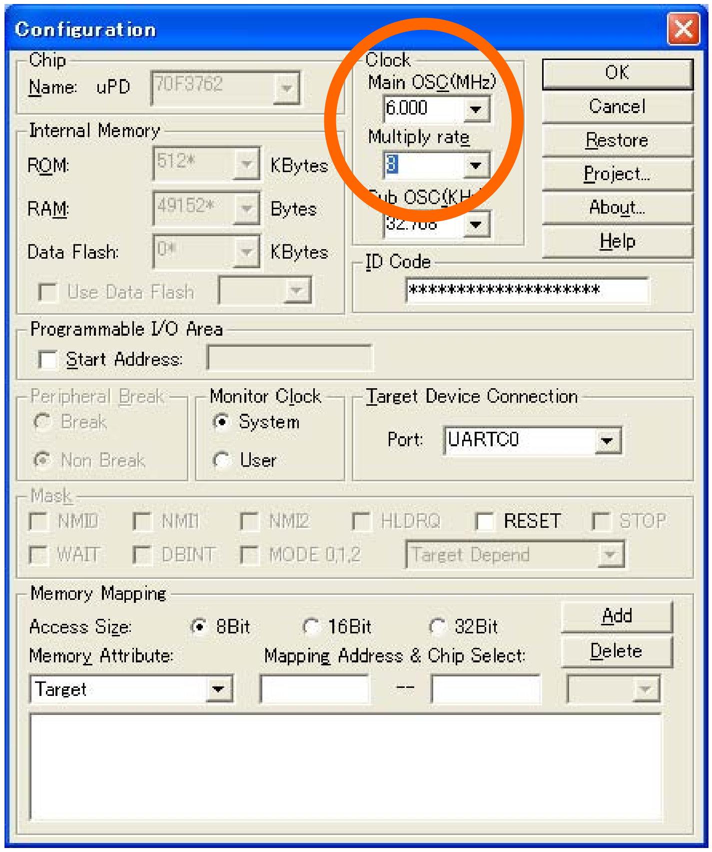
(1) Writing the load module

The procedure for writing the load module to the TK-850/JG3H by using PM+ is described below.

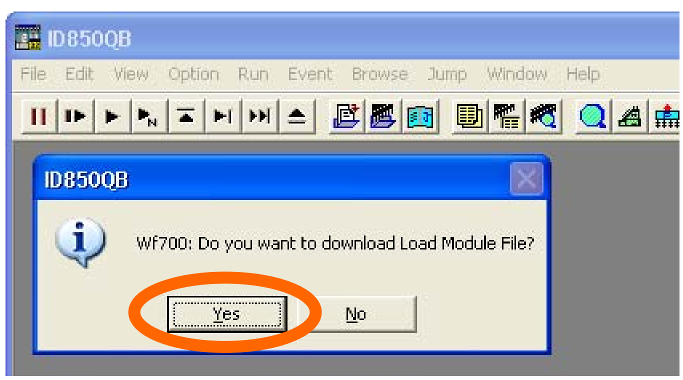
<1> Start the ID850QB by selecting **Debug** in the **Build** menu.



<2> In the **Configuration** dialog box, select “6.000” (MHz) for **Main OSC** and “8” for **Multiply rate**.

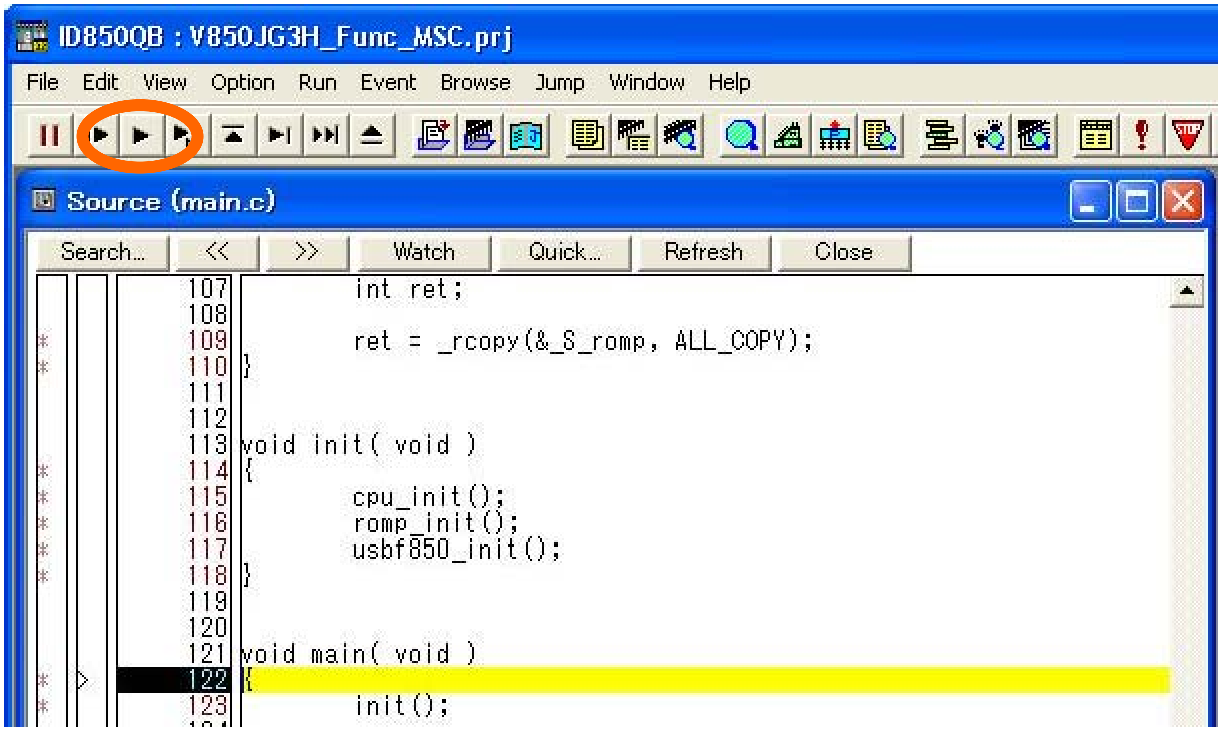


<3> If a project file included with the sample driver is used, the following dialog box is displayed. Click the **Yes** button to start writing the load module file.



(2) Executing the program

Click the ▶ button in the ID850QB window or select **Run Without Debugging** in the **Run** menu.



4.4 Checking the Operation

This section describes the procedure for checking the execution result after executing the sample driver program.

(1) Connecting the device port

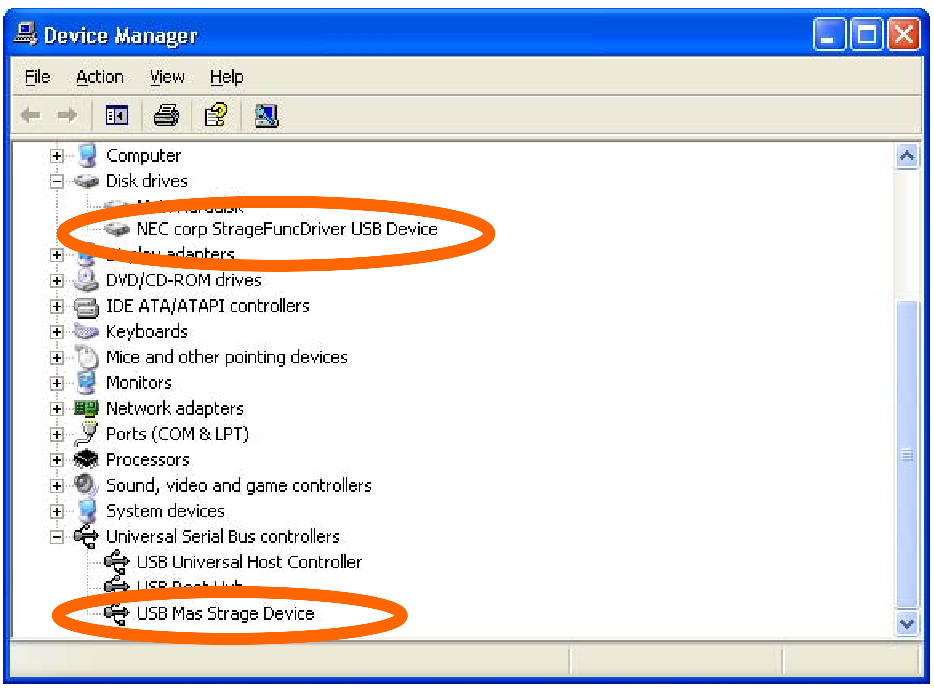
Connect the device port of the TK-850/JG3H (USB2) to the USB port of the host by using a USB cable.

(2) Installing the host driver

The mass storage class host driver, which is a standard Windows driver, is used for the device port. The driver is automatically installed if the TK-850/JG3H is connected to the host when the sample driver is running.

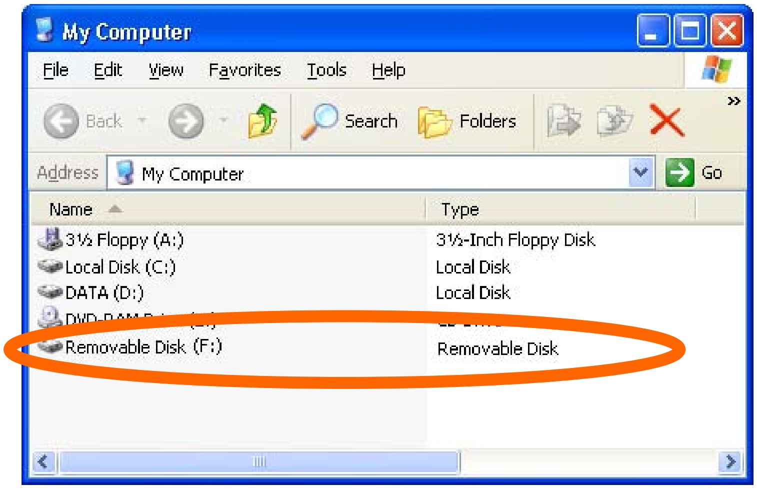
(3) Checking the connection of USB devices

Open the **Device Manager** window. In the **Universal Serial Bus controllers** category, make sure that **USB Mass Storage Device** is displayed. Also make sure that **NEC corp StorageFuncDriver USB Device** is displayed in the **Disk drives** category.



(4) Format of removable disks

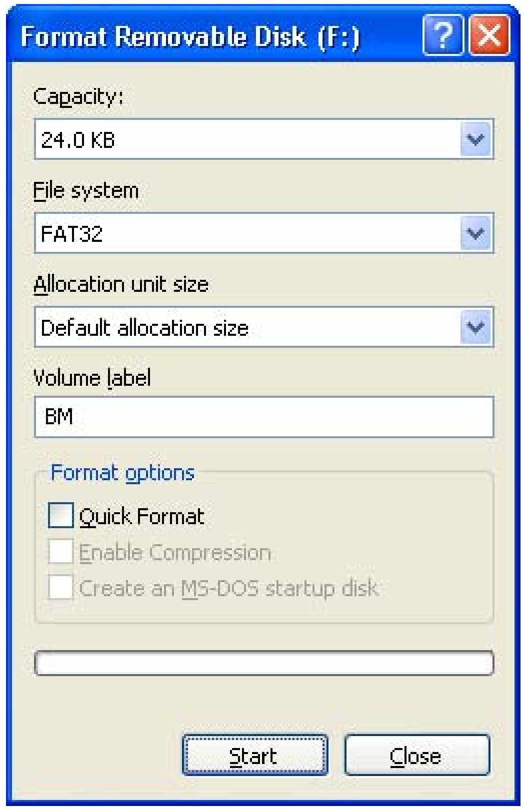
Open the **My Computer** window to display “Removable Disk”.



Remark ”(F:)” in the screenshot is a drive letter automatically assigned by the OS. The drive letter varies depending on the host setup.

<1> If "Removable Disk" is clicked in the **My Computer** window, the message “The disk in drive F is not formatted.” is displayed. Click the **Yes** button.

<2> In the **Format Removable Disk** dialog box, specify each setting, and then click the **Start** button.

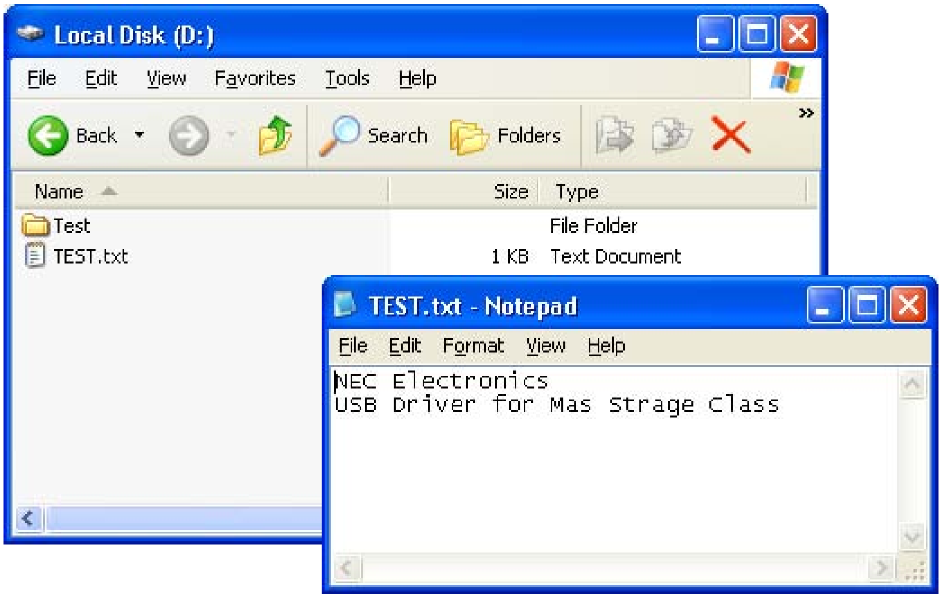


<3> When the disk has been formatted, a dialog box is displayed. Click the **OK** button.

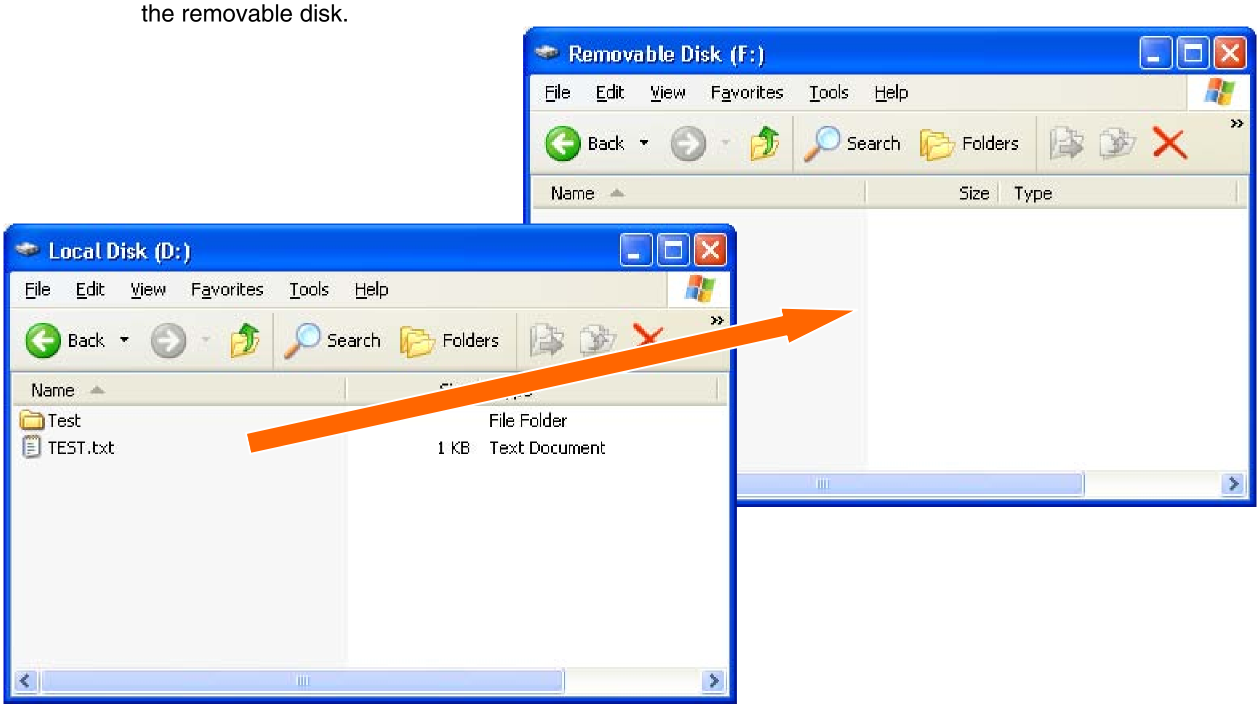
(5) Storing and extracting files

Confirm that files can be written to and read from the removable disk.

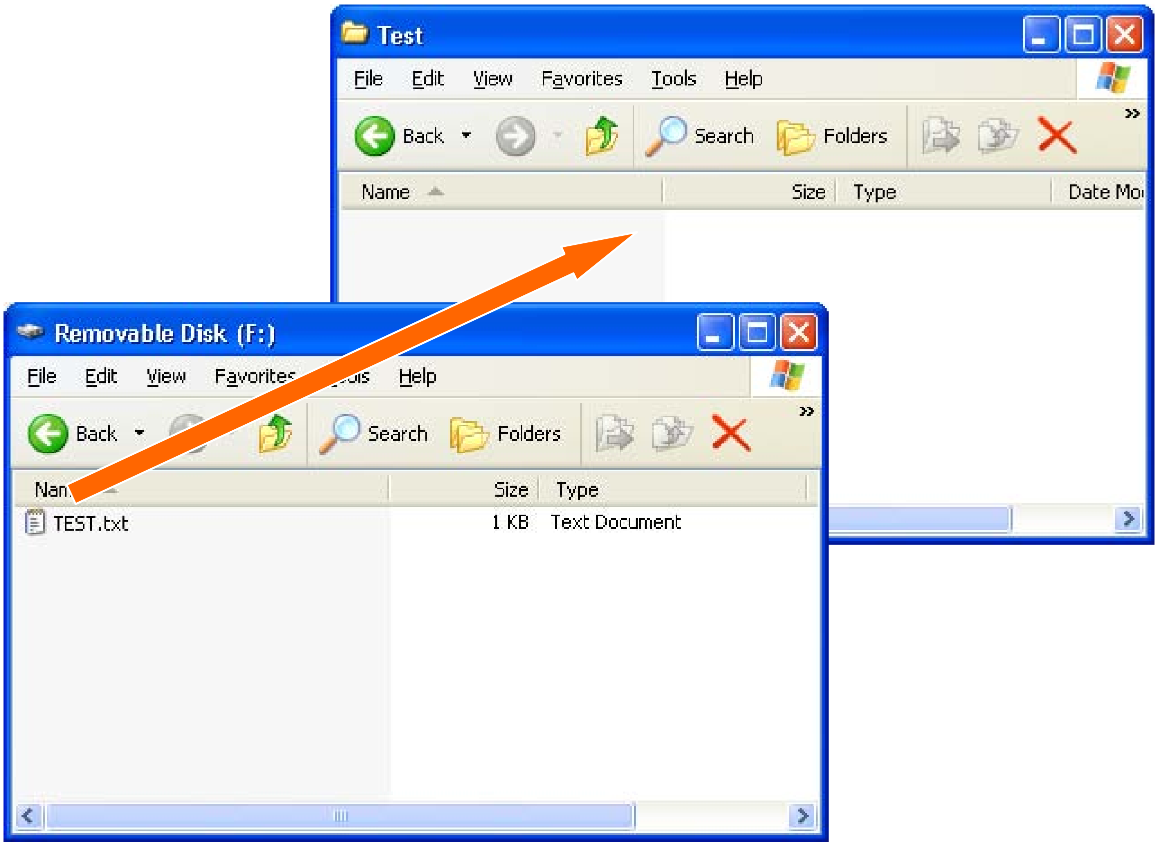
<1> Create a `TEST.txt` file and Test folder in the local disk.



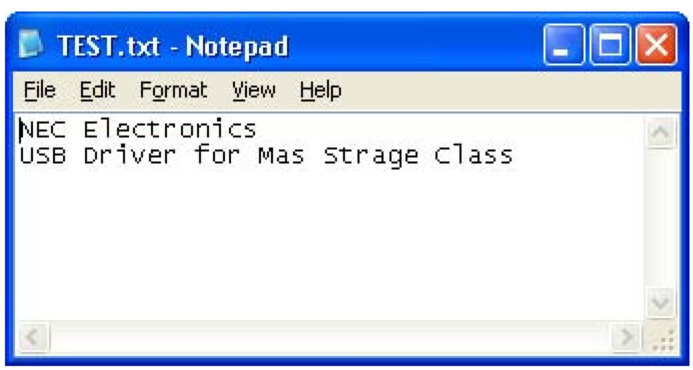
<2> Open the removable disk in the **My Computer** window, and then copy `TEST.txt` from the local disk to the removable disk.



<3> Open the Test folder in the local disk, and then copy `TEST.txt` from the removable disk to the Test folder.



<4> Open `TEST.txt` in the Test folder and confirm that the contents are the same as those in `TEST.txt` in the local disk.



Caution **24 KB of the internal RAM is used as the data area. Therefore, the saved data is initialized when the device is turned off or the reset switch is pressed.**
Operation is not guaranteed if a file that has a size of 24 KB or more is written.

CHAPTER 5 USING THE SAMPLE DRIVER

This chapter describes information that you should know when using the USB mass storage class (MSC) sample driver for the V850ES/Jx3-H.

5.1 Overview

The sample driver can be customized to make it suitable for your system.

For the most part, you will have to rewrite the following sections of the sample driver:

- The sample application section in `main.c`
- The values specified for the registers in `RegDef.h` and `usbf850_sfr.h`
- The descriptor information in `usbstrg_desc.h`
- SCSI command processing in `scsi_cmd.c` and `scsi.h`
- RAM disk capacity in `scsi.h`
- Vendor and product names in `scsi_cmd.c`

Remark For the list of files included in the sample driver, see **1.1.3 Files included in the sample driver**.

5.2 Customizing the Sample Driver

This section describes the sections to rewrite as required when using the sample driver.

5.2.1 Application section

The main routine function (`main`) in `main.c` includes a simple example of processing using the sample driver. The existing initialization processing and interrupt servicing can be used by including the processing to actually use for the application in this section.

List 5-1. Main Routine

```
void main( void )
{
     init();

     while(1)
     {
          if( rs_flag == SUSPEND )
          {
            __DI();                      /* maskable interrupt disable */

            UF0IC0 = C_IC0_ALL;          /* interrupt clear */
            UF0IC1 = C_IC1_ALL;          /* interrupt clear */
            UF0IC2 = C_IC2_ALL;          /* interrupt clear */
            UF0IC3 = C_IC3_ALL;          /* interrupt clear */
            UF0IC4 = C_IC4_ALL;          /* interrupt clear */

            UFIF0 = 0;                   /* INTUSB0 */
            UFIF1 = 0;                   /* INTUSB1 */
            BRGINTE = 0x0002;            /* INTUSB0, INTUSB1 */
            UFMK0 = 1;                   /* INTUSB0 */
            UFMK1 = 0;                   /* INTUSB1 */
            UFCKMSK = 0x03;              /* USB Disable */
    :
   Omitted
    :
            rs_flag = RESUME;

            __EI();                      /* maskable interrupt enable */
          }
     }
}
```

5.2.2 Setting up the registers

The registers the sample driver uses (writes to) and the values specified for them are defined in `RegDef.h` and `usbf850_sfr.h`. By rewriting the values in this file according to the actual use for the application, the operation of the target device can be specified by using the sample driver.

(1) `RegDef.h`

The CPU registers that are mainly used in initialization processing are defined in this file. (For details, see **3.2.1 CPU Initialization processing**.)

(2) `usbf850_sfr.h`

This file includes the definitions of the USBF registers, the register bits used in various types of processing, and the values specified for the bits. (For details, see **3.3.2 USBF initialization processing**.)

5.2.3 Descriptor information

The data the sample driver adds to the USBF during initialization processing (described in **3.1.3 Descriptor settings**) is defined in `usbstrg_desc.h`. Information such as the attributes of the target device can be specified by using the sample driver by rewriting the values in this file according to the use in an actual application.

Any information can be specified for the string descriptor. The sample driver defines manufacturer and product information, so rewrite the information as required.

List 5-2. Section in `usbstrg_desc.h` That Sets Up the String Descriptor

```
    :
/* 0 : Language Code*/
DSTR(LangString, 2, (0x09,0x04));
/* 1 : Manufacturer*/
USTR(ManString, 19, ('N','E','C',' ','E','l','e','c','t','r','o','n','i','c','s','
','C','o','.'));
    :
```

5.2.4 Changing SCSI command processing

SCSI command processing is included in `scsi_cmd.c` and `scsi.h`. Change the files as follows to add a supported SCSI command:

- Add the processing function to `scsi_cmd.c`.
- Add the case statement that calls the function added to the SCSI command execution processing function (`scsi_command_to_ata`) in `scsi_cmd.c`.
- Add the declaration of the function that was added to the function declaration section in `scsi.h`.

List 5-3. SCSI Command Execution Processing Function (`scsi_command_to_ata`)

```
INT32
scsi_command_to_ata(UINT8* ScsiCommandBuf, UINT8* pbData, INT32 lDataSize, INT32
TransFlag)
{
    long status;

    /*::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    ** It summons processing according to the contents of the command.
    **:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/

    switch (ScsiCommandBuf[0]) {
/*No data Access*/
    case TEST_UNIT_READY:        /*processing of TEST UNIT READY command*/
                status = ata_test_unit_ready(TransFlag);
                return status;

    case SEEK:                   /*processing of SEEK command*/
                status = ata_seek(TransFlag);
                return status;
        :
       Omitted
        :
    case PREVENT:                /* PREVENT/ALLOW MEDIUM REMOVAL command */
                u.clear_sense_data     = 0;
                return DEV_OK;

    default:                     /*processing of an un-supported command*/
                u.sense_data.sense_key = ILLEGAL_REQUEST;
                u.sense_data.asc       = 0x20;   // Invalid Command Operation Code
                u.sense_data.ascq      = 0x00;
                return DEV_ERROR;
    }
}
```

5.2.5 Changing the RAM disk capacity

The RAM disk capacity is written in `scsi.h`. The product of ALL_LOGICBLOCK (the total number of blocks) and LOGICBLOCK_SIZE (the block size) is the RAM disk capacity. (For the sample driver, the RAM disk capacity is set to 0x6000 (= 24 KB). However, the disk capacity that can be used in a computer is less than the specified value, because disk space is consumed by information such as FAT.)

List 5-4. Section in `scsi.h` That Specifies the Data Length

```
/*--------------------------------------------
 * data length of the table
 *--------------------------------------------*/
#define    INQUIRY_LENGTH          36       /*36Byte*/
#define    MODE_SENSE_LENGTH       24       /*24Byte*/
#define    MODE_SENSE10_LENGTH     28       /*28Byte*/
#define    MODE_SELECT_LENGTH      24       /*24Byte*/
#define    MODE_SELECT10_LENGTH    28       /*28Byte*/
#define    REQUEST_SENSE_LENGTH    18       /*18Byte*/
#define    READ_FORM_CAPA_LENGTH   20       /*20Byte*/

#define    MODE_SELECT_MIN_LEN     4        /*4Byte */

//#define  ALL_LOGICBLOCK     0x4      /*number of the outline reason blocks(192)*/
#define    ALL_LOGICBLOCK     0x30     /*number of the outline reason blocks(48)*/
//#define  ALL_LOGICBLOCK     0x7D0    /*number of the outline reason blocks(1000)*/
//#define  ALL_LOGICBLOCK     0xFA0    /*number of the outline reason blocks(4000)*/
#define    LOGICBLOCK_SIZE    0x200    /*1 logic block size(512Byte)*/
```

5.2.6 Specifying the vendor and product names

The names displayed as the vendor and product names of the disk drive can be changed by editing the INQUIRY command response values defined in `scsi_cmd.c`.

(1) INQUIRY_TABLE code

INQUIRY_TABLE in `scsi_cmd.c` is written as shown in List 5-5.

List 5-5. INQUIRY_TABLE Written in `scsi_cmd.c`

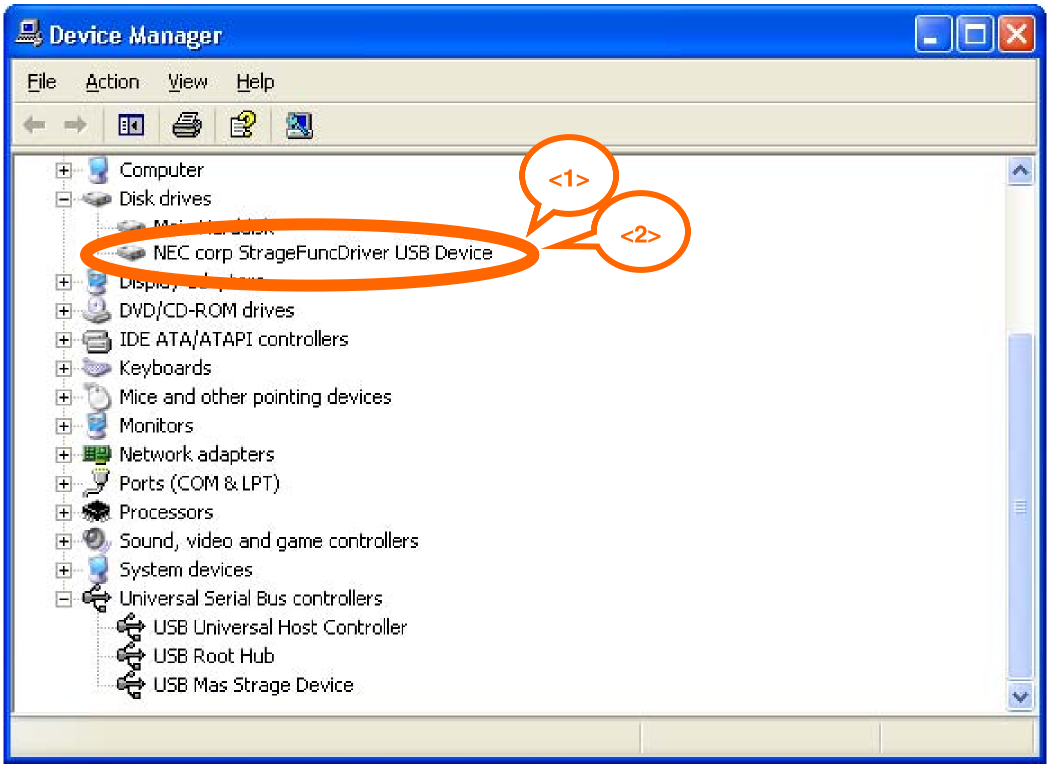
```
1  UINT8  INQUIRY_TABLE[INQUIRY_LENGTH]={
2      0x00,                              /*Qualifier, device type code*/
3      0x80,                              /*RMB, device type modification child*/
4      0x02,                              /*ISO Version, ECMA Version, ANSI
Version*/
5      0x02,                              /*AENC, TrmIOP, response data form*/
6      0x1F,                              /*addition data length*/
7      0x00,0x00,0x00,                    /*reserved*/
8      'N','E','C',' ','c','o','r','p',   /*vender ID*/                  <1>
9      'S','t','o','r','a','g','e','F','n','c','D','r','i','v','e','r',
                                          /*product ID*/                 <2>
10     '0','.','0','1'                    /*Product Revision*/
11 };
```

The vendor name is defined at <1>, on the eighth line, and the product name is defined at <2>, on the ninth line. An 8-byte character string can be used for the vendor name and a 16-byte character string for the product name.

When data is transmitted, all characters are transmitted as ASCII codes. Therefore, characters that cannot be decoded to ASCII code are not correctly displayed.

(2) Displaying device names (devices)

The vendor and product names specified in INQUIRY_TABLE are displayed as disk drive names in the **Device Manager** window.



5.3 Using Functions

The code for applications can be simplified and the code size can be reduced because frequently used and versatile types of processing are provided as defined functions. For details about each function, see **3.3 Function Specifications**.

For example, the CBW data reception processing section in `usbf850_storage.c` is as follows:

List 5-6. CBW Data Reception Processing Section

```
1  void
2  usbf850_rx_cbw(void)
3  {
4              UINT8* data = (UINT8 *)&CBW_TABLE;
5              INT8 len;
6
7              if (mass_storage_reset) {
8                 /*wait "Bulk-Only Mass Storage Reset" request*/
9                 usbf850_cbw_error();
10                return ;
11             }
12             len = UF0BO1L;
13
14 //          if (len != (sizeof(CBW_INFO))) {
15             if (len != 0x1F) {
16                return ; /*don't CBW*/
17             }
18
19             usbf850_data_receive(data, len, BKO1);
20
21             if (cbw_in_cbw) {
22                /*CBW in CBW*/
23                UF0FIC0 = (BKI1SC | BKI1CC); /*Clears EP1 buffers*/
24                cbw_in_cbw = USB_CBW_END;
25             }
26             cbw_in_cbw = USB_CBW_PROCESS;
27             usbf850_storage_cbwchk();
28             return ;
29 }
```

(1) Monitoring the mass storage reset flag (mass_storage_reset)

The flag that is set by the sample driver (mass_storage_reset) is monitored on the seventh line. If this flag is set to USB_MASS_RESET_WAIT (0x01), it indicates that the system is waiting for a mass storage reset request due to causes such as a failure in command processing.

(2) Data reception processing

The function that defines the processing that transfers the endpoint data to the buffer (`usbf850_data_receive`) is called on the 19th line. BKO1, which indicates the endpoint number, is defined in `usbf850.h`.

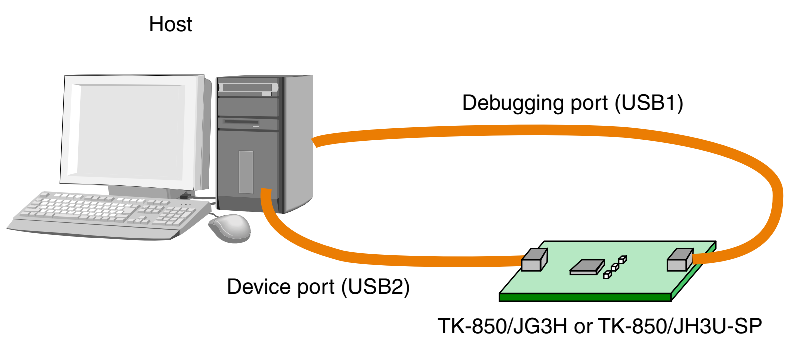
APPENDIX A STARTER KIT

This chapter describes the TK-850/JG3H starter kit for the V850ES/Jx3-H and the TK-850/JH3U-SP starter kit for the V850ES/Jx3-U, both made by Tessera Technology, Inc.

A.1 Overview

The TK-850/JG3H and TK-850/JH3U-SP are kits to develop applications that use the V850ES/Jx3-H and V850ES/Jx3-U, respectively. The entire development sequence from creating a program to building, debugging, and checking operation can be performed simply by installing development tools and USB drivers and then connecting either board to the host. These kits use a monitoring program that enables debugging without connecting an emulator (on-chip debugging).

Figure A-1. Connections of the TK-850/JG3H or TK-850/JH3U-SP



A.2 TK-850/JG3H

The TK-850/JG3H is a starter kit for the V850ES/Jx3-H.

A.2.1 Features

The TK-850/JG3H has the following features:

- A USB miniB connector for the internal USBF
- Up to 84 I/O ports
- As small as a business card
- Efficient development by using the board with the integrated development environment (PM+)

A.2.2 Specifications

The main specifications of the TK-850/JG3H are as follows:

• CPU	μPD70F3760 (V850ES/JG3-H)
• Operating frequency	48 MHz (subsystem clock: 32.768 kHz)
• Interface	USB connector (miniB) × 2 N-Wire connector (only the pad) MINICUBE®2 connector (SICA: only the pad) Peripheral board connector × 2 (only the pad)
• Supported platform	Host: DOS/V computer that has a USB interface OS: Microsoft Windows 2000, Microsoft Windows XP
• Operating voltage	5.0 V (internal operation at 3.3 V)
• Package dimensions	W89 × D52 (mm)

A.3 TK-850/JH3U-SP

The TK-850/JH3U-SP is a starter kit for the V850ES/Jx3-U.

A.3.1 Features

The TK-850/JH3U-SP has the following features:

- A USB miniB connector for the internal USBF
- A USB A type connector for the internal USBF
- Up to 96 I/O ports
- Peripheral devices for evaluation (such as Ethernet, IrDA, and SRAM)
- Efficient development by using the board with the integrated development environment (PM+)

A.3.2 Specifications

The main specifications of the TK-850/JG3H are as follows:

• CPU	μPD70F3769 (V850ES/JH3-U)
• Operating frequency	48 MHz (subsystem clock: 32.768 kHz)
• Interface	USB connector: A type, miniB type N-Wire connector (KEL), MINICUBE2 connector (SICA: only the pad), RS-232C connector (Dsub 9 pins), extension connector (100 pins) Ethernet RJ-45 connector, IrDA, audio I/O jack (φ3.5 mm, monaural)
• Other units	5-inch touch panel LCD (provided only in the LCD version) 7-segment LEDs (provided only in the non-LCD version)
• Supported platform	Host: DOS/V computer that has a USB interface OS: Microsoft Windows 2000, Microsoft Windows XP
• Operating voltage	5.0 V (USB or AC adapter)
• Package dimensions	W100 × D136 (mm)

For further information, please contact:

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

[America]

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

[Europe]

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

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

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

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

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

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

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

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

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

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

[Asia & Oceania]

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

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

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

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

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

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

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

G0706