V850E2/ML4 Microcontrollers

Example of USB Multifunction Operation

R01AN1037EJ0100
Rev.1.00
Mar. 23, 2012

Summary

This application note describes the usage of the USB function controller (USBF) incorporated in the V850E2/ML4 microcontrollers by using an example of operating the sample program created for the USB mass storage class (MSC) and communication device class (CDC).

The description and software presented in this application note are used to explain an example of using the USB function module. Note, however, that the described operation is not guaranteed.

Operation Verified Devices

V850E2/ML4 (μPD70F4022)

Contents

1. INTRODUCTION

1.1 Caution

The sample programs used in this application note are provided for reference purposes only and their operation is not guaranteed by Renesas Electronics Corporation.
Before using these sample programs, carefully evaluate them on the user's set.

1.2 Readers

This application note is intended for users who understand the features of the V850E2/ML4 microcontrollers, and are planning to develop application systems using a V850E2/ML4 microcontroller.

1.3 Purpose

This application note is intended to give users an understanding of the specifications of the sample program for using the USBF incorporated in the V850E2/ML4 microcontrollers.

1.4 Organization

This application note is broadly divided into the following four sections:

- An overview of the USB standard
- Overview of USB multifunction
- The specifications for the sample program
- Development environment (CubeSuite+)

1.5 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 V850E2/ML4 microcontrollers
  → See the separately provided *V850E2/ML4 microcontrollers Hardware User's Manual.*
– To learn about the instructions of the V850E2/ML4 microcontrollers
  → See the separately provided *V850E2M Architecture User's Manual.*
– To understand the specifications of the MSC driver and CDC driver
  → See the application note of each driver.

2. OVERVIEW

This application note describes the usage of the USB function controller (USBF) incorporated in the V850E2/ML4 microcontrollers by using an example of operating the sample program created for the MSC and CDC commands.
This sample program covers processing of control transfers, bulk transfers, interrupt transfers, and USB multifunctions (MSC and CDC class commands).

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

2.1 Overview

2.1.1 Features of the USBF

The USBF that is incorporated in the V850E2/ML4 microcontrollers and is to be controlled by the sample program 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 2-1. Configuration of the Endpoints of the V850E2/ML4 Microcontrollers

Endpoint Name	FIFO Size (Bytes)	Transfer Type	Remarks
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	64	Interrupt transfer (IN)	–
Endpoint 8	64	Interrupt transfer (IN)	–

- Automatically responds to standard USB requests (except some requests).
- The internal or external clock can be selected[Note]
  - Internal clock: 9.6 MHz external clock internally multiplied by 20, divided by 4 (48 MHz)
    7.2 MHz external clock internally multiplied by 20, divided by 3 (48 MHz)
  - External clock: Input to the USBCLK pin (f_{USB} = 48 MHz)

Note The sample program selects the internal clock.

2.1.2 Features of the sample program

The multifunction sample program for the V850E2/ML4 microcontrollers has the features below. For details about the features and operations, see **4. SAMPLE PROGRAM SPECIFICATIONS**.

- Allows the host to recognize connected devices with MSC and CDC functions.
- Operates as a self-powered device.
- 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.
- Compliant with Abstract Control Model of USB Communication Device Class Ver. 1.1
- Operation as a virtual COM device
- Exclusively uses the following amounts of memory (excluding the vector table):
  ROM: About 10.0 KB
  RAM: About 26.0 KB[Note]

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

2.1.3 Files included in the sample program

The sample program is created assuming use with CubeSuite+. The files included in the sample program are shown below.

Table 2-2. Files Included in the Sample Program

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_communication.c	CDC-specific processing
	usbf850_storage.c	MSC-specific processing
	cstart.asm	Bootstrap
include	main.h	main.c function prototype declaration
	scsi.h	SCSI macro definitions
	usbf850.h	usbf850.c function prototype declaration
	usbstrg_desc.h	Descriptor definition
	usbf850_errno.h	Error code definition
	usbf850_storage.h	usbf850_storage.c function prototype declarations
	usbf850_communication.h	usbf850_communication.c function prototype declaration
	usbf850_types.h	User-defined type declaration
	reg_v850e2ML4.h	USBF register definitions
inf file	*XXX*_CDC.inf	INF file for CDC (Windows® environment) The name of the respective microcontrollers fits in *XXX*:: ML4

Remark In addition, the project-related files for CubeSuite+ (an integrated development tool made by Renesas Electronics) are also included. For details, see **5.2.1 Preparing the host environment**.

2.2 V850E2/ML4 Microcontrollers

For details about the V850E2/ML4 microcontrollers that are to be controlled by the sample program, see the hardware user's manual of the applicable product.

2.2.1 Applicable products

The sample program can be used for the following products:

Table 2-3. V850E2/ML4 Microcontrollers

Generic Name	Part Number	Internal Memory		Incorporated USB Function	Interrupt		Refer to:
		Flash Memory	RAM		Internal[Note]	External[Note]	
V850E2/ML4	μPD70F4021	768 KB	Internal RAM 64 KB + shared memory 64 KB	Host/function	122	29	V850E2/ML4 Hardware User's Manual (R01UH0262EJ)
	μPD70F4022	1 MB	Internal RAM 64 KB + shared memory 64 KB	Host/function	122	29	

Note Includes non-maskable interrupts.

2.2.2 Features

The main features of the V850E2/ML4 are as follows.

- Internal memory: RAM: 64 KB
  Flash memory: 768 KB (μPD70F4021), 1 MB (μPD70F4022)
- Flash memory cache: Single core: 16 KB (4-way set associative)
- External bus interface: SRAM or SDRAM connectable
- Serial interface: Asynchronous serial interface UART: 2 channels
  Clocked serial interface CSI: 2 channels
  Asynchronous serial interface UART (FIFO): 2 channels
  Clocked serial interface CSI (FIFO): 2 channels
  I^2C: 2 channels
  CAN controller: 1 channel
  USB function controller: 1 channel
  USB host controller: 1 channel
  Ethernet controller: 1 channel
- DMA controller: DMA controller: 8 channels
  DTS: Up to 128 channels

3. OVERVIEW OF USB

This chapter provides an overview of the USB standard, which the sample program 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).

3.1 Transfer Format

Four types of transfer formats (control, bulk, interrupt, and isochronous) are defined in the USB standard. Table 3-1 shows the features of each transfer format.

Table 3-1. 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

3.2 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.

3.3 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 the USB (the function devices). A common host driver can be used if the connected devices conform to the standard specifications of the relevant class, which is defined by a protocol.

This sample program uses the MSC driver and CDC driver for multifunction processing. For details about each driver, see the application note.

3.3.1 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 the USB.

Communication using the MSC is performed using the bulk-only transfer protocol or CBI (control, bulk, or 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 program 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.

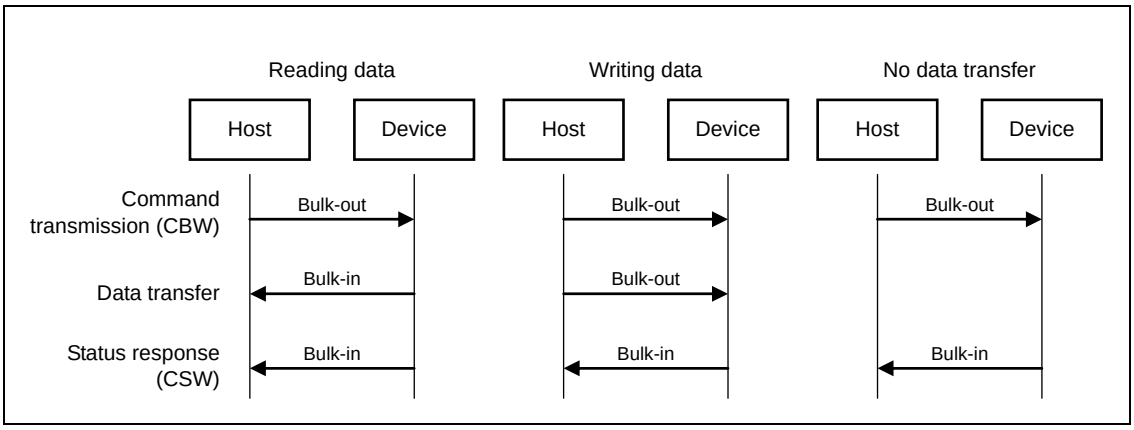


Figure 3-1. Data Transfer Flowchart

(2) CBW format

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

Table 3-2. CBW Format

Bit / Bytes	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).

Table 3-3. CSW Format

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

Table 3-4. Parameters Indicating CBW Processing Results

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

3.3.2 Subclass (MSC)

For MSC devices, specify the format in which commands are transmitted from the host to the target device as the subclass.

(1) Subclass types

Table 3-5 shows the subclass codes that are specified for the USB mass storage class.

Table 3-5. 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 program, see **4.1.5 Supported SCSI commands**.

3.3.3 Communication device class (CDC)

The communication device class (CDC) is a class for communication devices to be connected to the host computer, which includes modems, fax machines, and network cards. As computers no longer offer the RS-232C interface, CDC is commonly used for devices that provide USB serial conversion capability for UART communication with a computer. While various models of CDC are defined according to the target device, the sample program uses the Abstract Control Model.

3.4 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.

3.4.1 Types

There are three types of requests: standard requests, class requests, and vendor requests.
For details about requests that the sample program supports, see **4.1.3 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 3-6. 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, configuration, string	Reads the target descriptor.
SET_DESCRIPTOR	Device, configuration, string	Changes the target descriptor (optional).
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 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.

The sample program implements response processing for class requests corresponding to the Abstract Control Model of the CDC. The requests for which a response can be sent are listed below.

- Send Encapsulated Command
  This request is used to issue commands in the CDC interface's control protocol format.
- Get Encapsulated Response
  This request is used to request a response in the CDC interface's control protocol format.
- Set Line Coding
  This request is used to specify the serial communication format.
- Get Line Coding
  This request is used to acquire the current communication format of the device.
- Set Control Line State
  This request is used for RS-232/V.24 format control signals.

(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.

3.4.2 Format

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

Table 3-7. USB Request Format

Offset	Field		Description
0	bmRequestType		Request attribute
		Bit 7	Data transfer direction
		Bits 6, 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	

3.5 Descriptors

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.

3.5.1 Types

The following six 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 association descriptor
  This descriptor can associate multiple interface descriptors with a single function. This descriptor includes interface information that configures the function, such as the first interface ID number, the number of interface channels, and the class.
  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.

3.5.2 Format

The size and fields of each descriptor type vary as described below. The data sequence of each field is in little endian format.

Table 3-8. 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 3-9. 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 association, 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 3-10. Interface Association Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bFirstInterface	1	Number of the first interface channel incorporated in the USBF
bInterfaceCount	1	Number of interfaces incorporated in the USBF
bFunctionClass	1	Code of the class supported by the USBF
bFunctionSubClass	1	Code of the subclass supported by the USBF
bFunctionProtocol	1	Code of the protocol supported by the USBF
iFunction	1	Index to the string descriptor specifying the source code for this interface

Table 3-11. 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 3-12. 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 3-13. String Descriptor Format

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

4. SAMPLE PROGRAM SPECIFICATIONS

This chapter provides details about the features and processing of multifunctions when using the V850E2/ML4 microcontrollers, and the specifications of the functions of these programs.
For details about the sample program for each class, see the application note.

4.1 Overview

4.1.1 Overview of USB multifunction driver

The USB multifunction driver enables multiple function controllers to be implemented in one device.
USB multifunction processing executed by this sample program can handle the mass storage class (MSC) and communication device class (CDC) protocols. As a result, the host recognizes the connected device as a device that has two function controllers. In addition, the function controller for the CDC has two interface channels: control and data. This enables the device to use three interface channels in total, including one interface channel for the MSC controller.

Table 4-1. USB Multifunction Organization

Function	Interface	Class
Function 0	Interface 0	MSC (Bulk-Only)
Function 1	Interface 1	CDC (Control)
	Interface 2	CDC (Data)

The interface descriptors are used to report the ID of each interface to the host. The interface association descriptor (IAD) is used to report that interfaces 1 and 2 use the same function (CDC) to the host.
For a device that has an IAD, the classes shown in the table below must be specified by using the device descriptor.

Table 4-2. Device Descriptor Classes

Class	SubClass	Protocol	Description
0xEF	0x02	0x01	Device with IAD

Table 4-3. Interface Descriptor Classes

Interface	Class	SubClass	Protocol	Description
Interface 0	0x08	0x06	0x50	MSC bulk-only
Interface 1	0x02	0x02	0x00	CDC control
Interface 2	0x02	0x0A	0x00	CDC data

4.1.2 Features

The sample program 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 **4.2.7 Suspend/resume processing**. The CDC protocol is used to read data stored at an endpoint for bulk-out transfer (reception) and write the data to an endpoint for bulk-in transfer (transmission).

(2) Initialization

The USBF is set up for use by manipulating various registers. This setup includes specifying settings for the CPU registers of the V850E2M/ML4 and specifying settings for the registers of the USBF. For details, see **4.2.1 CPU initialization** and **4.2.2 USBF initialization**.

(3) Interrupt servicing

The INTUSFA0I1 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 INTUSFA0I2 interrupt handler is used to perform processing when a resume interrupt occurs. For details, see **4.2.3 USBF interrupt servicing (INTUSFA0I1)** and **4.2.4 USBF resume interrupt servicing (INTUSFA0I2)**.

(4) SCSI command processing (MSC)

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 **4.1.5 Supported SCSI commands**.

4.1.3 Supported requests

Table 4-4 shows the USB requests defined by the hardware (the V850E2/ML4) and firmware (the sample program).

Table 4-4. USB Request Processing

Request Name	Codes								Processing
	0	1	2	3	4	5	6	7	
Standard requests									
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 (MSC)									
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
Class requests (CDC)									
SEND_ENCAPSULATED_COMMAND	0x21	0x00	0x00	0x00	0xXX	0xXX	0xXX	0xXX	Firmware response
GET_ENCAPSULATED_RESPONSE	0xA1	0x01	0x00	0x00	0xXX	0xXX	0xXX	0xXX	Firmware response
SET_LINE_CODING	0x21	0x20	0x00	0x00	0xXX	0xXX	0xXX	0xXX	Firmware response
GET_LINE_CODING	0xA1	0x21	0x00	0x00	0xXX	0xXX	0xXX	0xXX	Firmware response
SET_CONTROL_LINE_STATE	0xA1	0x22	0x00	0x00	0xXX	0xXX	0xXX	0xXX	Firmware response
Other requests	Other than the above								Firmware STALL response

Remark Hardware: V850E2/ML4
Firmware: Sample program
0xXX: Undefined value

(1) Standard requests

The sample program responds to requests not automatically responded to by the V850E2/ML4 microcontrollers by using the following requests.

(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 program 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 program returns a STALL response.

(2) Class requests (MSC)

The sample program responds to class requests of the USB 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 program returns 0 (the number of logical units = 1).

Table 4-5. 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 program resets the interface of the USB function controller it uses when it receives a MASS_STORAGE_RESET request.

Table 4-6. Format of the MASS_STORAGE_RESET Request

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

(3) Class requests (CDC)

The sample program responds to class requests of the USB CDC by using the following requests.

(a) SendEncapsulatedCommand

This request is used to issue commands in the CDC interface's control protocol format.
Upon receiving this request, the sample program loads the data corresponding to the request and executes transmission processing.

(b) GetEncapsulatedResponse

This request is used to request a response in the CDC interface's control protocol format.
The sample program does not currently support this request.

(c) SetLineCoding

This request is used to specify the serial communication format.
Upon receiving this request, the sample program loads the data corresponding to the request, sets the communication rate, etc., and transmits a NULL packet through control read transfer.

(d) GetLineCoding

This request is used to acquire the current communication format of the device.
Upon receiving this request, the sample program reads the communication rate and other settings, and executes the transmission processing through control read transfer.

(e) SetControlLineState

This request is for RS-232/V.24 format control signals.
Upon receiving this request, the sample program transmits a NULL packet through control read transfer.

(4) Undefined requests

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

4.1.4 Descriptor settings

The settings of each descriptor specified by the sample program are shown below. These settings are included in header file `usbf850_desc.h`.

(1) Device descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_device request.
The settings are stored in the USFA0DDn registers (where n = 0 to 17) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_device request.

Table 4-7. 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	0xEF	Class code that has interface association descriptor
`bDeviceSubClass`	1	0x02	
`bDeviceProtocol`	1	0x01	
`bMaxPacketSize0`	1	0x40	Maximum packet size of endpoint 0: 64
`idVendor`[Note]	2	0x045B	Vendor ID: Renesas Electronics
`idProduct`[Note]	2	0x0218	Product ID: V850E2/ML4
`bcdDevice`	2	0x0001	Device release number: 1st version
`iManufacturer`	1	0x01	Index to the string descriptor representing the manufacturer: 1
`iProduct`	1	0x02	Index to the string descriptor representing the product: 0
`iSerialNumber`	1	0x03	Index to the string descriptor representing the device production number: 3
`bNumConfigurations`	1	0x01	Number of configurations: 1

Note: Set the vendor ID and product ID appropriately for the actual user system.

(2) Configuration descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the USFA0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Table 4-8. Configuration Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x02	Descriptor type: Configuration
wTotalLength	2	0x004F	Total number of bytes of the configuration, interface, and endpoint descriptors: 79 bytes
bNumInterfaces	1	0x03	Number of interfaces in this configuration: 3
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	0xC0	Features of this configuration: Self-powered, no remote wakeup
bMaxPower	1	0x1B	Maximum current consumed in this configuration: 54 mA

(3) Interface association descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the USFA0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

In this sample program, the USBF for the MSC has one interface channel (interface 0), and the USBF for the CDC has two interface channels: one for control (interface 1) and one for data (interface 2). Use this descriptor to indicate that the USBF for the CDC consists of two interface channels.

Table 4-9. Interface Association Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x08	Descriptor size: 8 bytes
bDescriptorType	1	0x0b	Descriptor type: Interface association
bFirstInterface	1	0x01	Number of first interface that configures this function: 1
bInterfaceCount	1	0x02	Number of interfaces in this function: 2
bFunctionClass	1	0x02	Class code in this function: CDC
bFunctionSubClass	1	0x00	Subclass code in this function: None
bFunctionProtocol	1	0x00	Protocol code in this function: None
iFunction	1	0x00	Index to the string descriptor specifying the source code for this function: 0

(4) Interface descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the USFA0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

This sample program uses three interfaces: interface 0 for MSC, interfaces 1 and 2 for CDC, each of which is used for control and data, respectively.

Table 4-10. Interface Descriptor Settings (MSC)

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

Table 4-11. Interface Descriptor Settings (CDC Control)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x04	Descriptor type: Interface
bInterfaceNumber	1	0x01	Identification number of this interface: 1
bAlternateSetting	1	0x00	Whether the alternative settings are specified for this interface: No
bNumEndpoints	1	0x01	Number of endpoints of this interface: 1
bInterfaceClass	1	0x02	Class code: Communication interface class
bInterfaceSubClass	1	0x02	Subclass code: Abstract Control Model
bInterfaceProtocol	1	0x00	Protocol code: No unique protocol is used
iInterface	1	0x00	Index to the string descriptor specifying the source code for this interface: 0

Table 4-12. Interface Descriptor Settings (CDC Data)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x04	Descriptor type: Interface
bInterfaceNumber	1	0x02	Identification number of this interface: 2
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	0x0A	Class code: Communication interface class
bInterfaceSubClass	1	0x00	Subclass code: Abstract Control Model
bInterfaceProtocol	1	0x00	Protocol code: No unique protocol is used
iInterface	1	0x00	Index to the string descriptor specifying the source code for this interface: 0

(5) Endpoint descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the USFA0CIEn registers (where n = 0 to 255) when the USBF is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

There are five types of endpoint descriptors, because the sample program uses three endpoints.

The endpoint address can be switched by enabling or disabling the `define` definitions shown below, which are in the header file `usbf850.h`. (The default setting is *enabled*.)

```
#define          USE_EP_BKI1
#define          USE_EP_BKO1
```

Table 4-13. Combination of Endpoints

	MSC		CDC	
	Bulk In	Bulk Out	Bulk In	Bulk Out
When `define` is enabled	EP1	EP2	EP3	EP4
When `define` is disabled	EP3	EP4	EP1	EP2

Table 4-14. 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	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

Table 4-15. 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	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 4-16. Endpoint Descriptor Settings for Endpoint 3 (Bulk-In)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: Endpoint
bEndpointAddress	1	0x83	Transfer direction of this endpoint: IN Address of this endpoint: 3
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 4-17. Endpoint Descriptor Settings for Endpoint 4 (Bulk-Out)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: Endpoint
bEndpointAddress	1	0x04	Transfer direction of this endpoint: OUT Address of this endpoint: 4
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 4-18. Endpoint Descriptor Settings for Endpoint 7 (Interrupt-In)

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: Endpoint
bEndpointAddress	1	0x87	Transfer direction of this endpoint: IN Address of this endpoint: 7
bmAttributes	1	0x03	Transfer type of this endpoint: Interrupt
wMaxPacketSize	2	0x0040	Maximum packet size of this transfer: 64 bytes
bInterval	1	0x0A	Polling interval of this endpoint: 10 ms

(6) String descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_string request.

If a GET_DESCRIPTOR_string request is received, the sample program extracts the settings of this descriptor from `usbf850_desc.h` and stores them into the USFA0E0W register of the USBF.

Table 4-19. 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	0x40	Descriptor size: 64 bytes
`bDescriptorType`	1	0x03	Descriptor type: String
`bString`Note 2	62	–	Vendor: Renesas 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.

(c) String 2

Field	Size (Bytes)	Specified Value	Description
`bLength`Note 1	1	0x10	Descriptor size: 16 bytes
`bDescriptorType`	1	0x03	Descriptor type: String
`bString`Note 2	14	–	Product type: MultDrv (Multifunction driver)

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.

(d) String 3

Field	Size (Bytes)	Specified Value	Description
`bLength`Note 1	1	0x1A	Descriptor size: 26 bytes
`bDescriptorType`	1	0x03	Descriptor type: String
`bString`Note 2	24	–	Serial number: 0216EF020110 (V850E2/ML4)

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.

4.1.5 Supported SCSI commands

For the MSC sample program, the SCSI transfer command set (0x06) is specified as the subclass. Table 4-20 shows the SCSI commands supported by the sample program. The sample program returns a STALL response if it receives a command that is not shown in Table 4-20.

Table 4-20. SCSI Commands Supported by the Sample Program

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 program, this command initializes the sense data and ends normally.

Table 4-21. 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 program, this command transmits the sense data shown in Table 4-22 to the host.

Table 4-22. 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 4-23. 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 4-24. 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 4-25. 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 4-26. 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 program, this command initializes the sense data and ends normally.

Table 4-27. 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 program, this command transmits the INQUIRY_TABLE values to the host.

Table 4-28. 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 4-29. 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)							

```
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*/
       'R','e','n','e','s','a','s',' ',               /*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*/
};
```

Figure 4-1. INQUIRY_TABLE

(7) MODE_SELECT command (0x15)

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

Table 4-30. 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 4-31. 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)							

```
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*/
};
```

Figure 4-2. MODE_SELECT_TABLE

(8) MODE_SENSE6 command (0x1A)

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

Table 4-32. 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 4-33. 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)							

```
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*/
};
```

Figure 4-3. MODE_SENSE_TABLE

(9) START_STOP_UNIT command (0x1B)

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

Table 4-34. 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 program, this command ends normally without performing any processing.

Table 4-35. 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 program, this command transmits the READ_FORMAT_CAPACITY_TABLE values to the host.

Table 4-36. 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 4-37. 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							

```
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            */
};
```

Figure 4-4. READ_FORMAT_CAPACITY_TABLE

(12) READ_CAPACITY command (0x25)

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

Table 4-38. 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 4-39. 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)							

```
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)*/
};
```

Figure 4-5. READ_CAPACITY_TABLE

(13) READ10 command (0x28)

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

Table 4-40. 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 4-41. 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 program, this command only writes the received data.

Table 4-42. 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 program, this command ends normally without performing any processing.

Table 4-43. 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 program, this command initializes the sense data and ends normally.

Table 4-44. 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 program, this command reads and then discards data, and then ends normally.

Table 4-45. 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 program, this command transmits the values of MODE_SENSE10_TABLE to the host.

Table 4-46. 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 4-47. 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)							

```
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 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*/
};
```

Figure 4-6. MODE_SENSE10_TABLE

(20) MODE_SELECT10 command (0x55)

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

Table 4-48. 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 4-49. 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)							

```
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*/
};
```

Figure 4-7. MODE_SELECT10_TABLE

4.2 Operation of Each Section

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

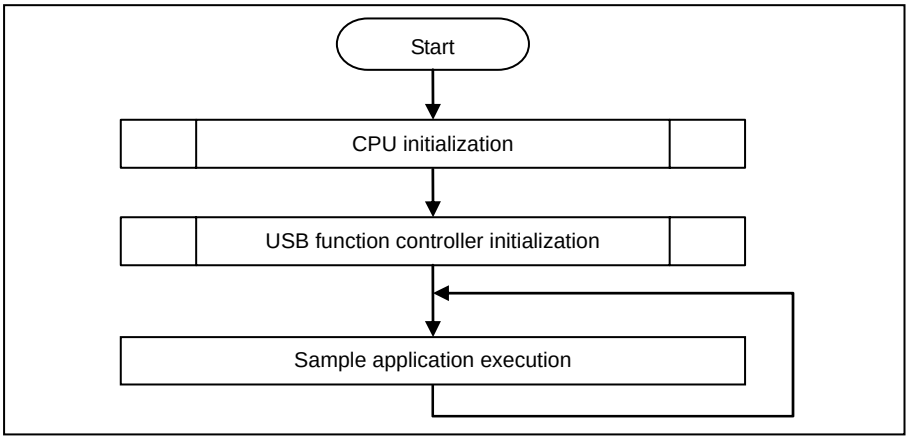


Figure 4-8. Sample Program Processing Flowchart

4.2.1 CPU initialization

The settings necessary to use the USBF are specified.

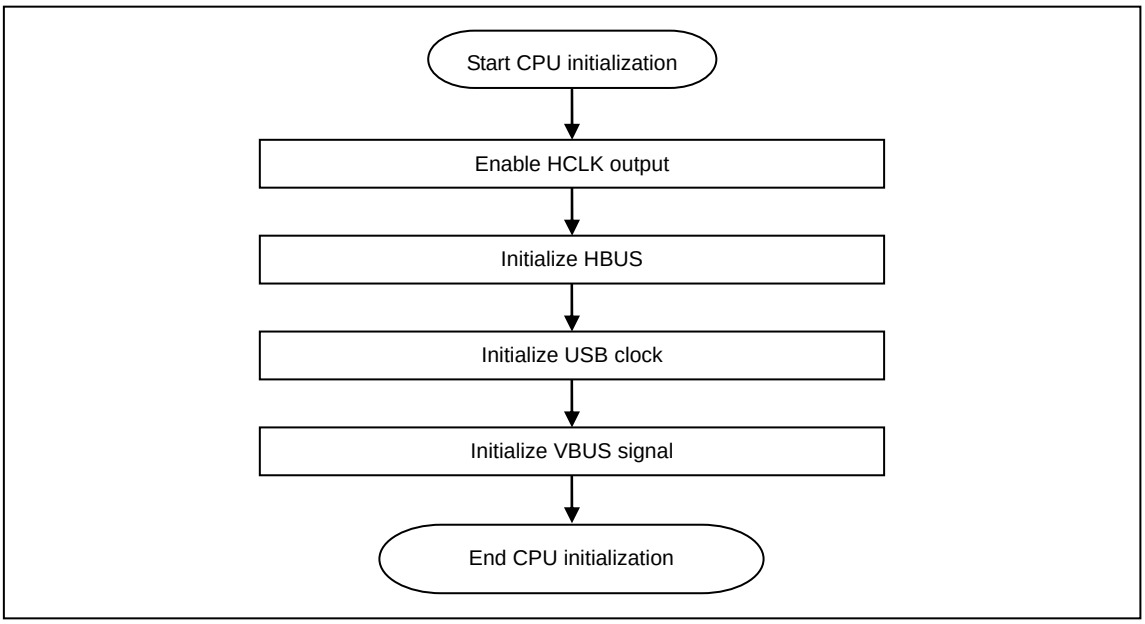


Figure 4-9. CPU Initialization Flowchart

(1) Enabling HCLK output

Outputting HCLK is enabled so as to enable the USBF connected to the H bus. The SFRCTL2 register used for specifying the settings must be written by using a special writing sequence.

(2) Initializing the H bus

The H bus is initialized according to the specified conditions. For details, see the V850E2/ML4 microcontroller hardware user's manual.

(3) Initializing the USB clock

The setting of alternate function pin P2_11, which is connected to UCLK, is specified. In this sample program, UCLK is used to input the USB clock.

(4) Initializing the VBUS signal

The VBUS signal is initialized.

4.2.2 USBF initialization

The settings necessary to use the USBF are specified.

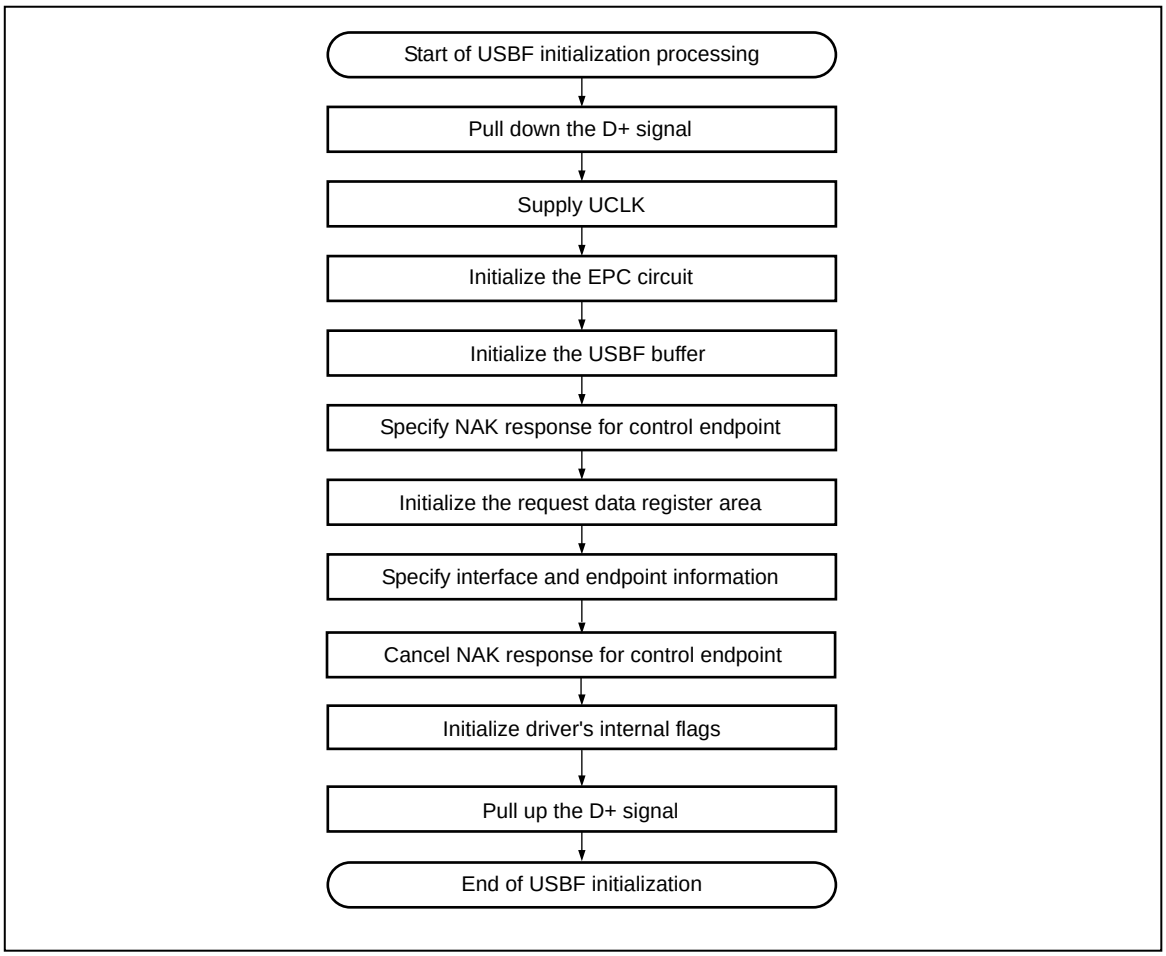


Figure 4-10. USBF Initialization Flowchart

(1) Pulling down the D+ signal

0 is written to the P2_4 of the CPU. This sets the D+ signal to low level output, meaning that device connection is not reported to the host.

(2) Specifying UCLK supply

Set the SFRCTL3 register to “0x48” to enable clock supply to the USB function controller.

(3) Initializing the EPC circuit

Set the USFA0EPCCTL register to “0x00000000” to cancel the EPC reset.

(4) Initializing the USB function buffer

Set the USFBC register to “0x00000003” to enable the USBF buffer and enable the floating countermeasure.

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

1 is written to the EP0NKA bit of the USFA0E0NA 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.

(6) 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:

(a) 0x01 is written to the USFA0DSTL register to disable remote wakeup and operate the USBF as a self-powered device.
(b) 0x00 is written to the USFA0EnSL registers (where n = 0 to 2) to indicate that endpoint n operates normally.
(c) The total data length (number of bytes) of the required descriptor is written to the USFA0DSCL register to determine the range of the USFA0CIEn registers (where n = 0 to 255).
(d) The device descriptor data is written to the USFA0DDn registers (where n = 0 to 17).
(e) The data of the configuration, interface, and endpoint descriptors is written to the USFA0CIEn registers (where n = 0 to 255).
(f) 0x00 is written to the USFA0MODC register to enable automatic responses to GET_DESCRIPTOR_configuration requests.

(7) 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:

(a) 0x81 is written to the USFA0AIFN register to enable interfaces 0, 1, and 2.
(b) 0x00 is written to the USFA0AAS register to disable the alternative setting.
(c) 0x20 is written to the USFA0E1IM register to link endpoint 1 to interface 0.
(d) 0x20 is written to the USFA0E2IM register to link endpoint 2 to interface 0.
(e) 0x60 is written to the USFA0E3IM register to link endpoint 3 to interface 2.
(f) 0x60 is written to the USFA0E4IM register to link endpoint 4 to interface 2.
(g) 0x40 is written to the USFA0E7IM register to link endpoint 7 to interface 1.

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

0 is written to the EP0NKA bit of the USFA0E0NA register to restart responses corresponding to each request, including requests that are automatically responded to.

(9) Setting up the interrupt mask registers

Masking is specified for each USBF interrupt source.
The following registers are accessed:

(a) 0x00 is written to the USFA0ICn register (n = 0 to 4) to clear all interrupt sources.
(b) 0xF7 is written to the USFA0FIC0 register and 0x0F is written to the USFA0FIC1 register to clear all transfer FIFOs.
(c) 0x1B is written to the USFA0IM0 register to mask interrupt sources indicated by the USFA0IS0 register other than those of the BUSRST, RSUSPD, and SETRQ interrupts.
(d) 0x7E is written to the USFA0IM1 register to mask interrupt sources indicated by the USFA0IS1 register other than those of the CPUDEC interrupt.
(e) 0xF1 is written to the USFA0IM2 register to mask all interrupt sources indicated by the USFA0IS2 register.
(f) 0xEE is written to the USFA0IM3 register to mask interrupt sources indicated by the USFA0IS3 register other than those of the BKO1DT and BKO2DT interrupt.
(g) 0x20 is written to the USFA0IM4 register to mask all interrupt sources indicated by the USFA0IS4 register.
(h) 0x0003 is written to the USFA0EPCINTE register to enable interrupts when the EPC_INT0BEN and EPC_INT1BEN bits are set.
(i) 0 is written to ICUSFA0I1 and to ICUSFA0I2 to enable INTUSFA0I1 and INTUSFA0I2.

(10) Initialization of driver's internal flags

The flags (`usbf850_busrst_flg`, `usbf850_rsuspd_flg`, and `usbf850_rdata_flg`) used in the driver are initialized.

(11) Pulling up the D+ signal

0x0010 is written to the P2 register of the CPU to output 1 from P2_4. This outputs a high level signal from the D+ pin to report to the host that a device has been connected. For the sample program, the connections shown in Figure 4-11 are assumed.

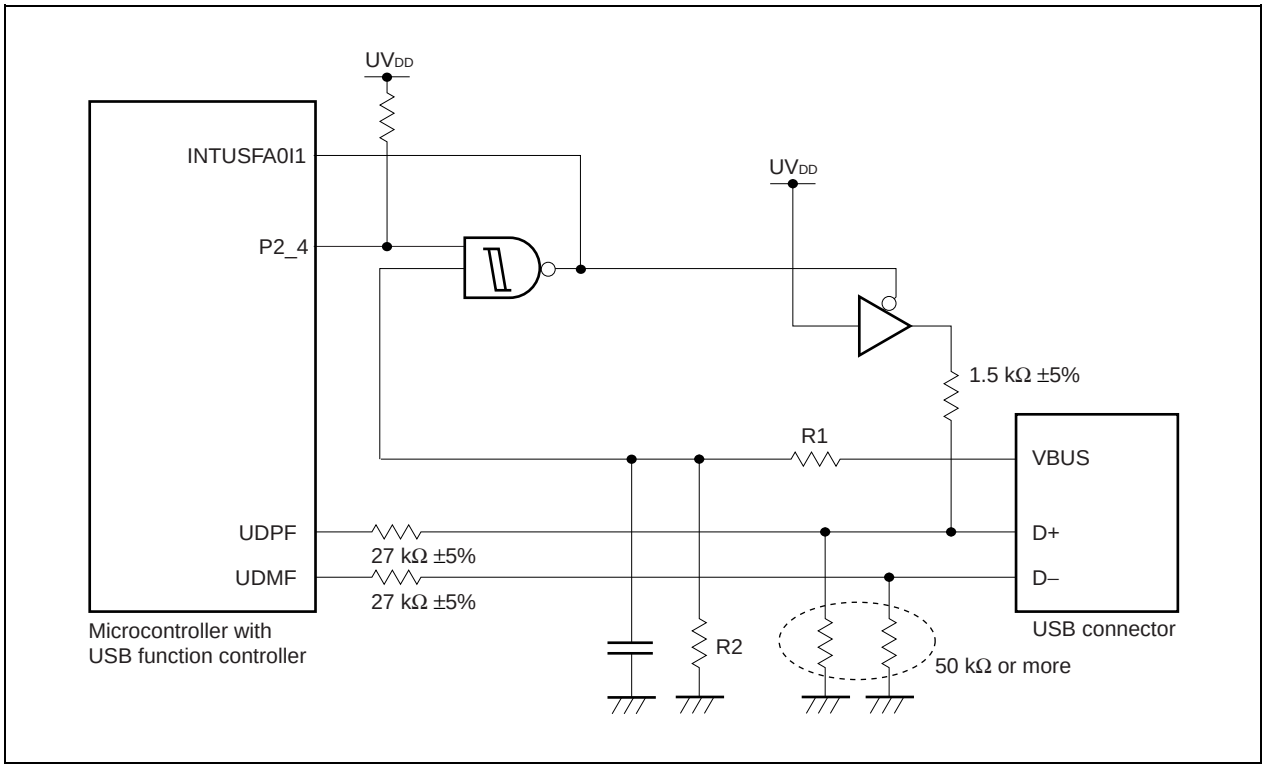


Figure 4-11. USBF Connection Example

4.2.3 USBF interrupt servicing (INTUSFA0I1)

The INTUSFA0I1 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, endpoint 4) and to perform processing corresponding to received requests and data.

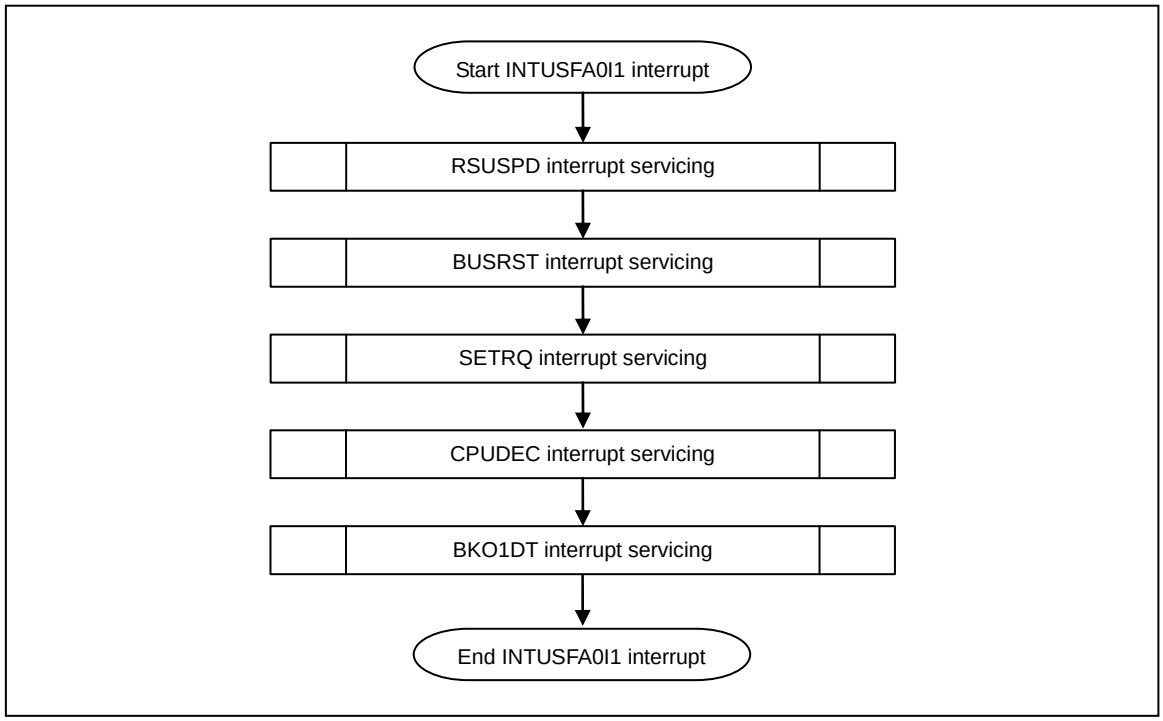


Figure 4-12. INTUSFA0I1 Interrupt Handler Processing Flowchart

(1) RSUSPD interrupt servicing

If the RSUSPD bit of the USFA0IS0 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 USFA0IC0 register.)
- Whether the processing is suspended or has resumed is determined.

(2) Suspend processing

If the RSUM bit of the USFA0EPS1 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 INTUSFA0I1 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 INTUSFA0I1 interrupt servicing.

(3) BUSRST interrupt servicing

If the BUSRST bit of the USFA0IS0 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 USFA0IC0 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.

Remark 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 USFA0IS1 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 USFA0IC1 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) BKOnDT interrupt servicing

If the BKOnDT bit of the USFA0IS3 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 USFA0IC3 register.)
- The CBW data reception function (`usbf850_rx_cbw`) is called to receive CBW data.

If a BKO2DT interrupt occurred, the following processing is performed:

- The BKO2DT interrupt source is cleared. (0 is written to the BKO2DT bit of the USFA0IC3 register.)
- The flag indicating that CDC data has been received (`usbf850_cdc_rdata_flg`) is updated.

4.2.4 USBF resume interrupt servicing (INTUSFA0I2)

The INTUSFA0I2 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 `rs_flag` is set to RESUME, the processing is performed in the main routine.

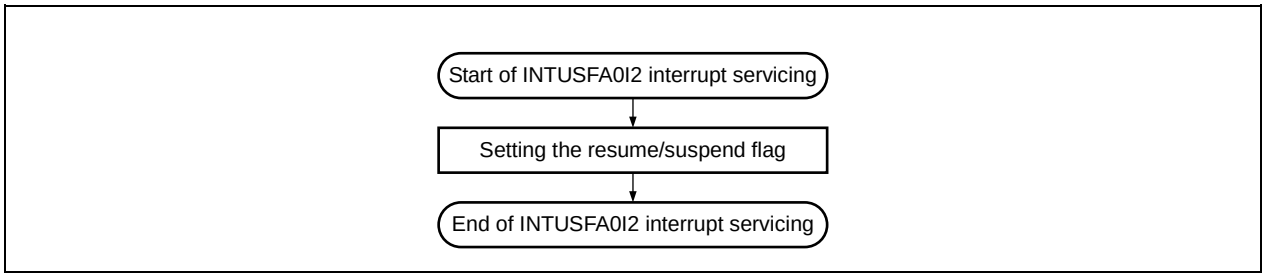


Figure 4-13. INTUSFA0I2 Interrupt Handler Processing Flowchart

4.2.5 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.

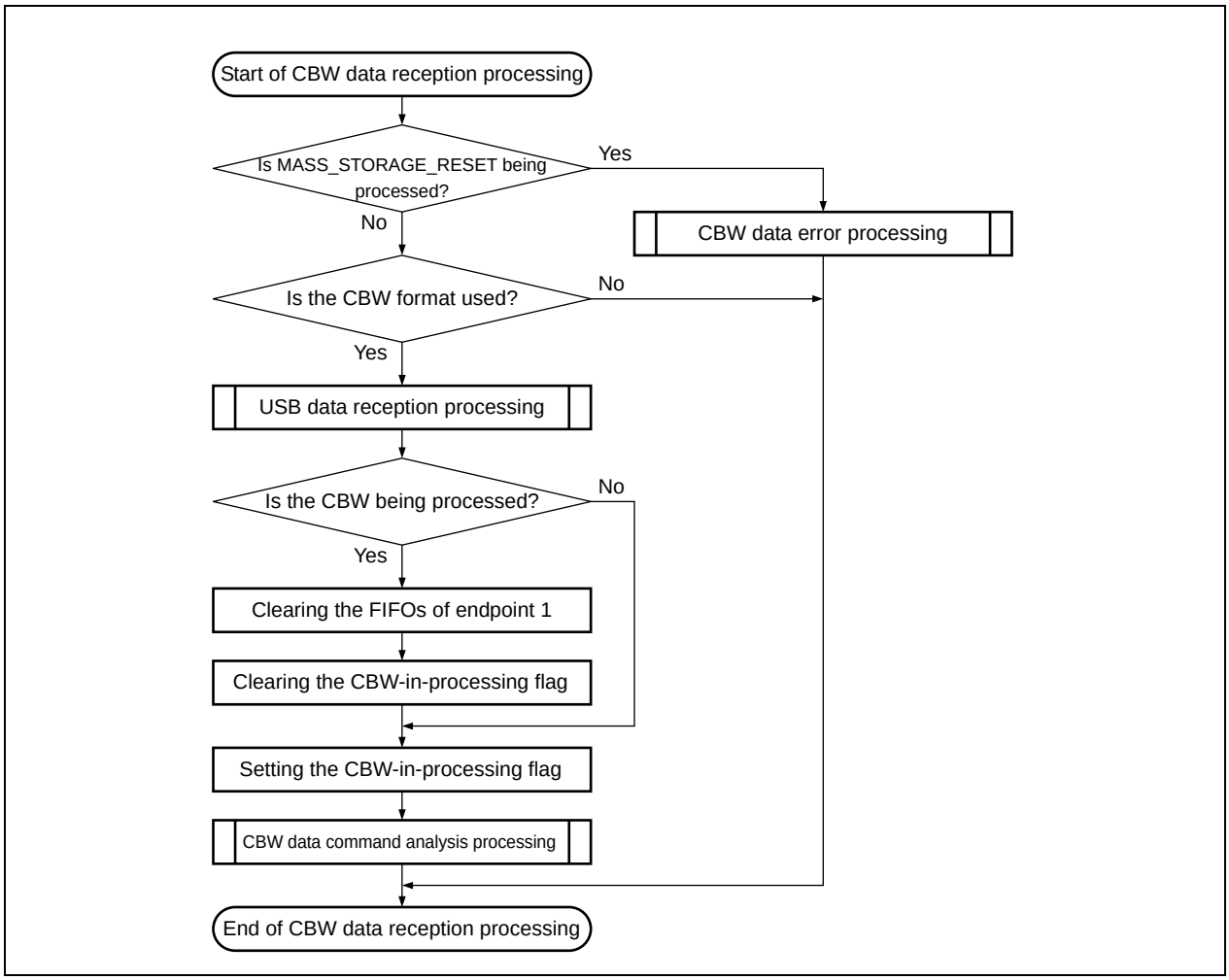


Figure 4-14. CBW Data Reception Processing Flowchart

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

If the MASS_STORAGE_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 (USFA0BO1L). 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_multiple_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.

4.2.6 SCSI command processing

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

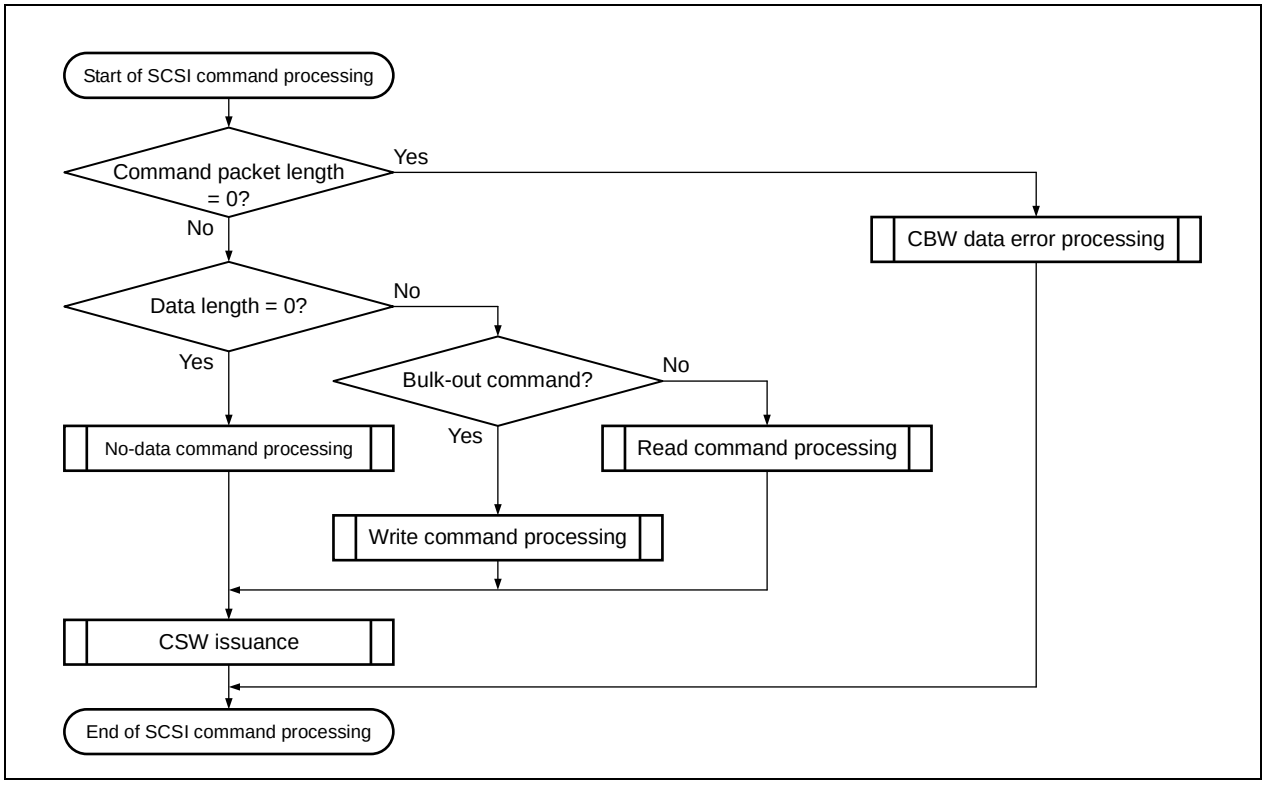


Figure 4-15. SCSI Command Processing Flowchart

(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_multiple_data_send`) 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_multiple_data_receive`) 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.

4.2.7 Suspend/resume processing

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

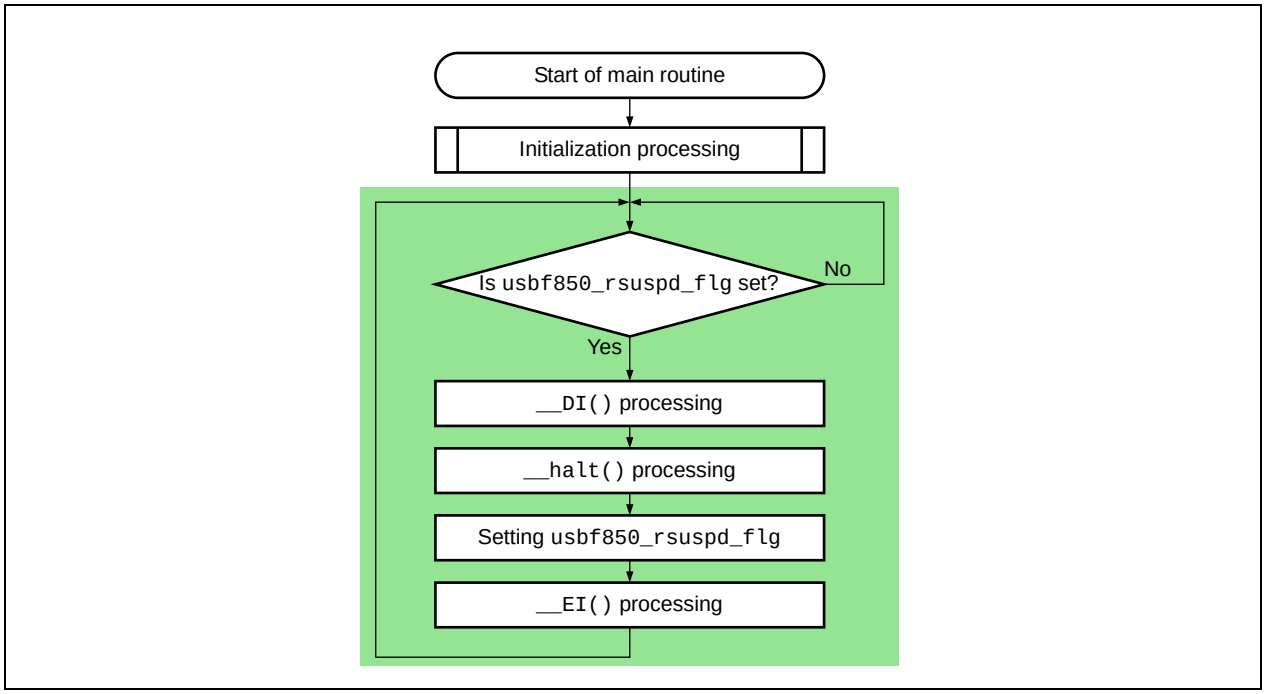


Figure 4-16. Suspend/Resume Processing Flowchart

(1) Monitoring the resume/suspend flag (`usbf850_rsuspd_flg`)

The resume/suspend flag (`usbf850_rsuspd_flg`) that is set by the sample program is monitored. If this flag is set to SUSPEND (0x00), the USB bus is suspended.

(2) Disabling CPU interrupts

The occurrence of a CPU interrupt when the resume/suspend flag (`usbf850_rsuspd_flg`) is set to "SUSPEND (0x00)" is disabled.

(3) CPU HALT processing

The processor is stopped and enters HALT mode. The processor exits HALT mode and resumes processing when a maskable interrupt, NMI, or reset occurs. In this sample program, the processor resumes processing when the resume interrupt INTUSFA0I2 occurs.

(4) Setting the resume/suspend flag (`rs_flag`)

The resume/suspend flag (`usbf850_rsuspd_flg`) is set to RESUME (0x01).

(5) Enabling CPU interrupts

The occurrence of CPU interrupts is enabled. Resume processing then finishes.

4.3 Function Specifications

This section describes the functions implemented in the sample program.

4.3.1 Functions

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

Table 4-50. Functions in the Sample Program (1/2)

Source File	Function Name	Description
main.c	main	Main routine
	cpu_init	Initialization the CPU
	SetProtectReg	Allows access to write-protected registers
usbf850.c	usbf850_init	Initializes the USBF.
	usbf850_intusbf0	Monitors endpoint 0 and controls response to request.
	usbf850_intusbf1	Resume interrupt servicing
	usbf850_multiple_data_send	Transmits USB data. (MSC)
	usbf850_data_send	Transmits USB data. (CDC)
	usbf850_multiple_data_receive	Receives USB data. (MSC)
	usbf850_data_receive	Receives USB data. (CDC)
	usbf850_rdata_length	Acquires the USB reception data length.
	usbf850_send_EP0	Transmits USB data for endpoint 0.
	usbf850_receive_EP0	Receives USB data for endpoint 0.
	usbf850_send_null	Transmits a NULL packet to Bulk/Interrupt In endpoint.
	usbf850_sendnullEP0	Transmits a NULL packet for endpoint 0.
	usbf850_sendstallEP0	Performs a STALL response for endpoint 0.
	usbf850_ep_status	Notifies the FIFO status for Bulk/Interrupt In endpoint.
	usbf850_fifo_clear	Clears the FIFO for endpoints other than endpoint 0.
	usbf850_standardreq	Processes standard requests.
	usbf850_getdesc	Processes GET_DESCRIPTOR requests.
usbf850_storage.c	usbf850_classreq	Processes MSC class requests.
	usbf850_blkonly_mass_storage_reset	Processes MASS_STORAGE_RESET requests.
	usbf850_max_lun	Processes GET_MAX_LUN requests.
	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.
	usbf850_bulkin_stall	Controls the STALL response for bulk-in transfer.
	usbf850_bulkout_stall	Controls the STALL response for bulk-out transfer.

Table 4-51. Functions in the Sample Program (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).
usbf850_communication.c	usbf850_cdc_classreq	Processes CDC class requests.
	usbf850_send_encapsulated_command	Processes Send Encapsulated Command requests.
	usbf850_get_encapsulated_response	Processes Get Encapsulated Command requests.
	usbf850_set_line_coding	Processes Set Line Coding requests.
	usbf850_get_line_coding	Processes Get Line Coding requests.
	usbf850_set_control_line_state	Processes Set Control Line State requests
	usbf850_buff_init	Clears the FIFO of the endpoint for CDC data transfer.
	usbf850_get_bufinit_flg	Notifies the execution status of FIFO initialization.
	usbf850_send_buf	Transmits CDC data.
	usbf850_recv_buf	Registers CDC's class request processing function.

4.3.2 Correlation of the functions

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

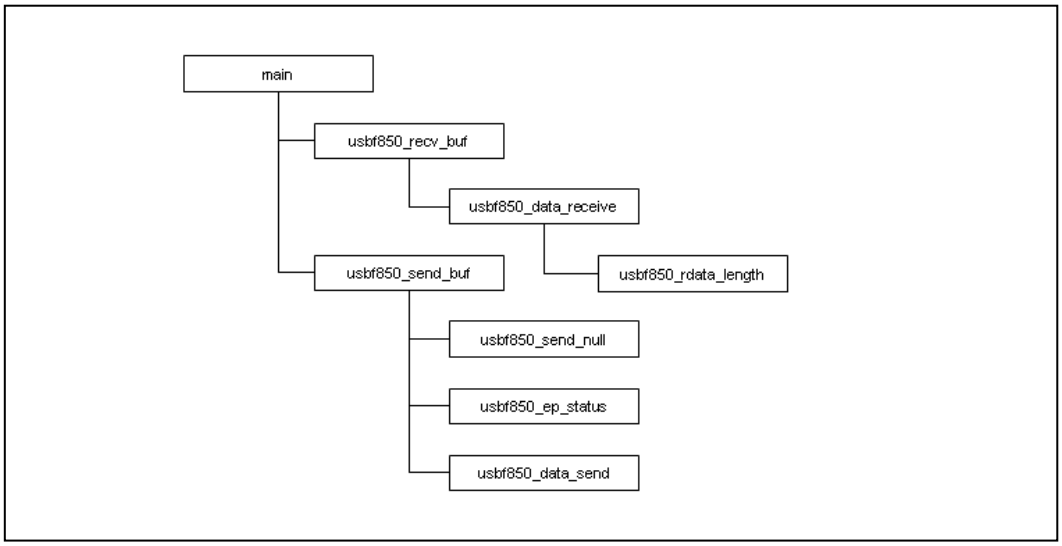


Figure 4-17. Calling Functions During `main` Processing

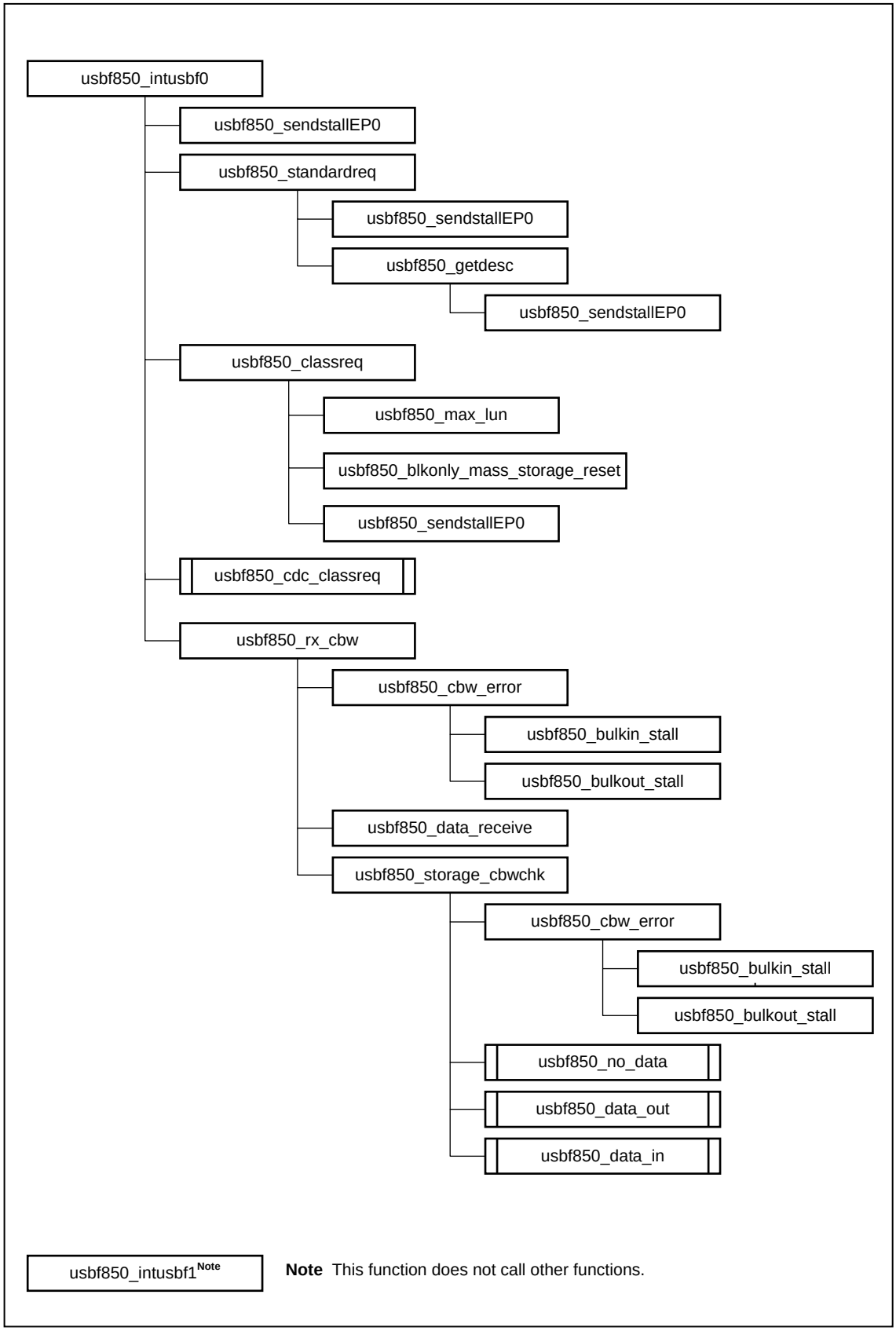


Figure 4-18. Calling Functions During USB Interrupt Servicing

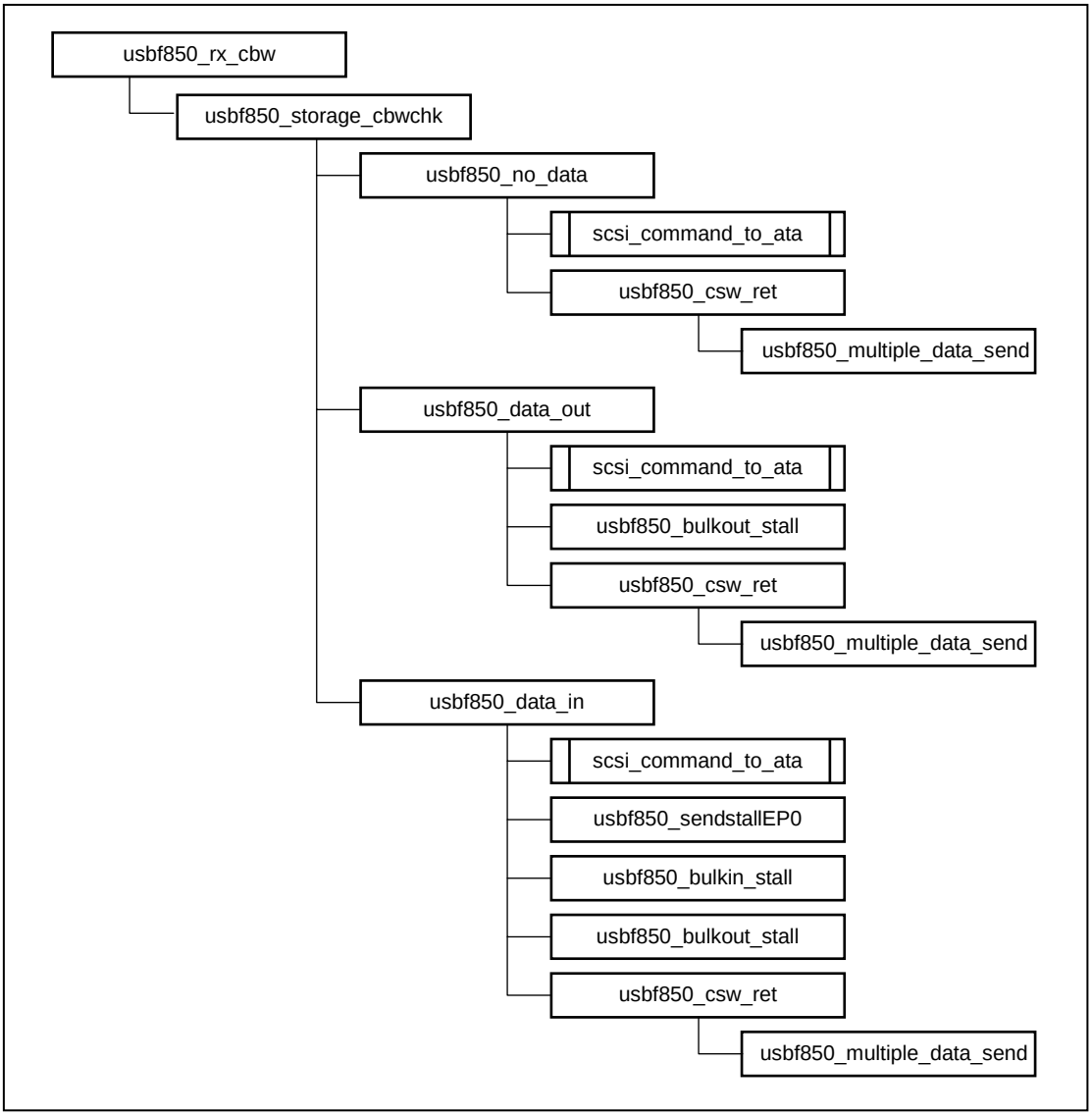


Figure 4-19. Calling Functions During CBW or CSW Processing

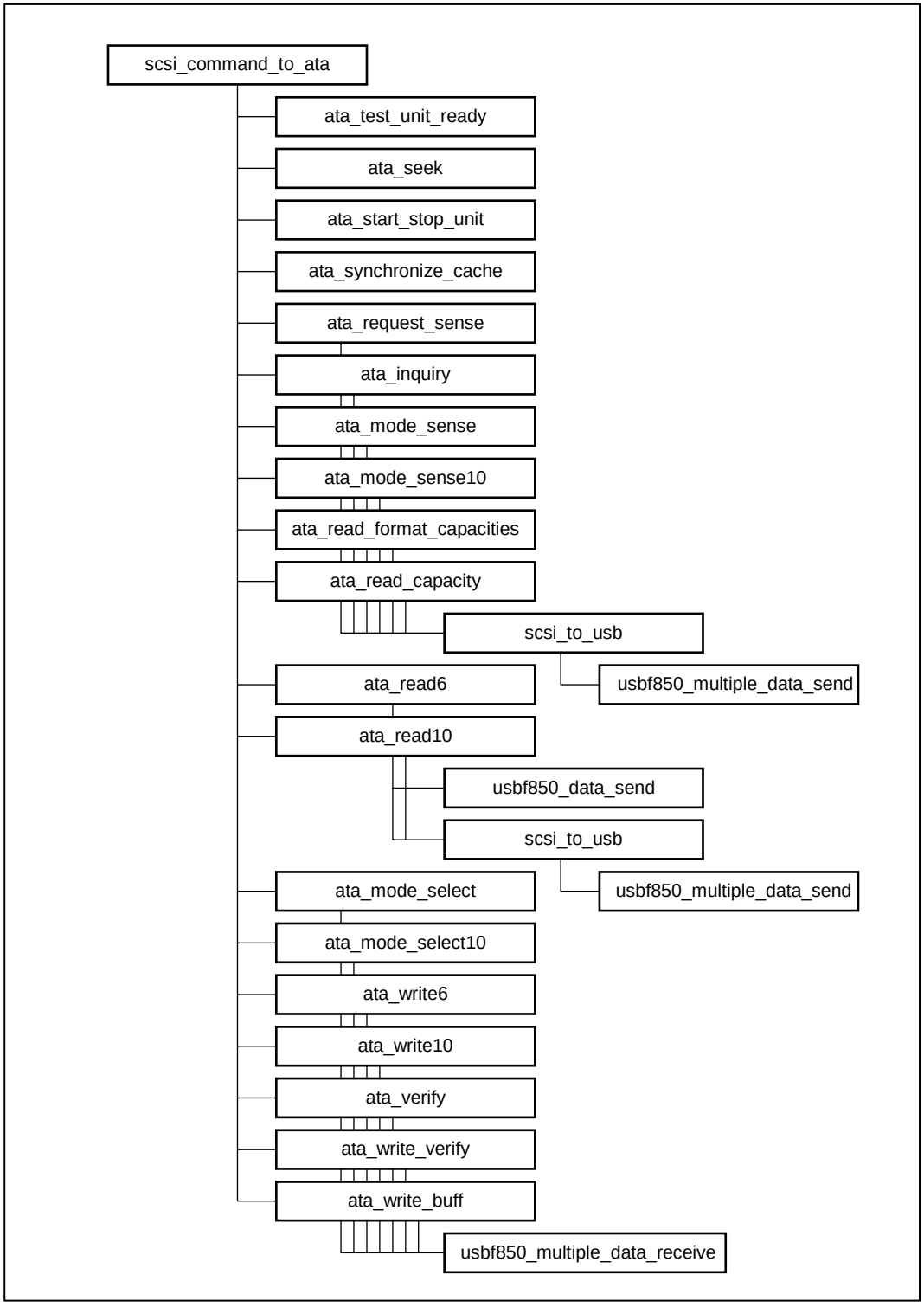


Figure 4-20. Calling Functions During SCSI Command Processing

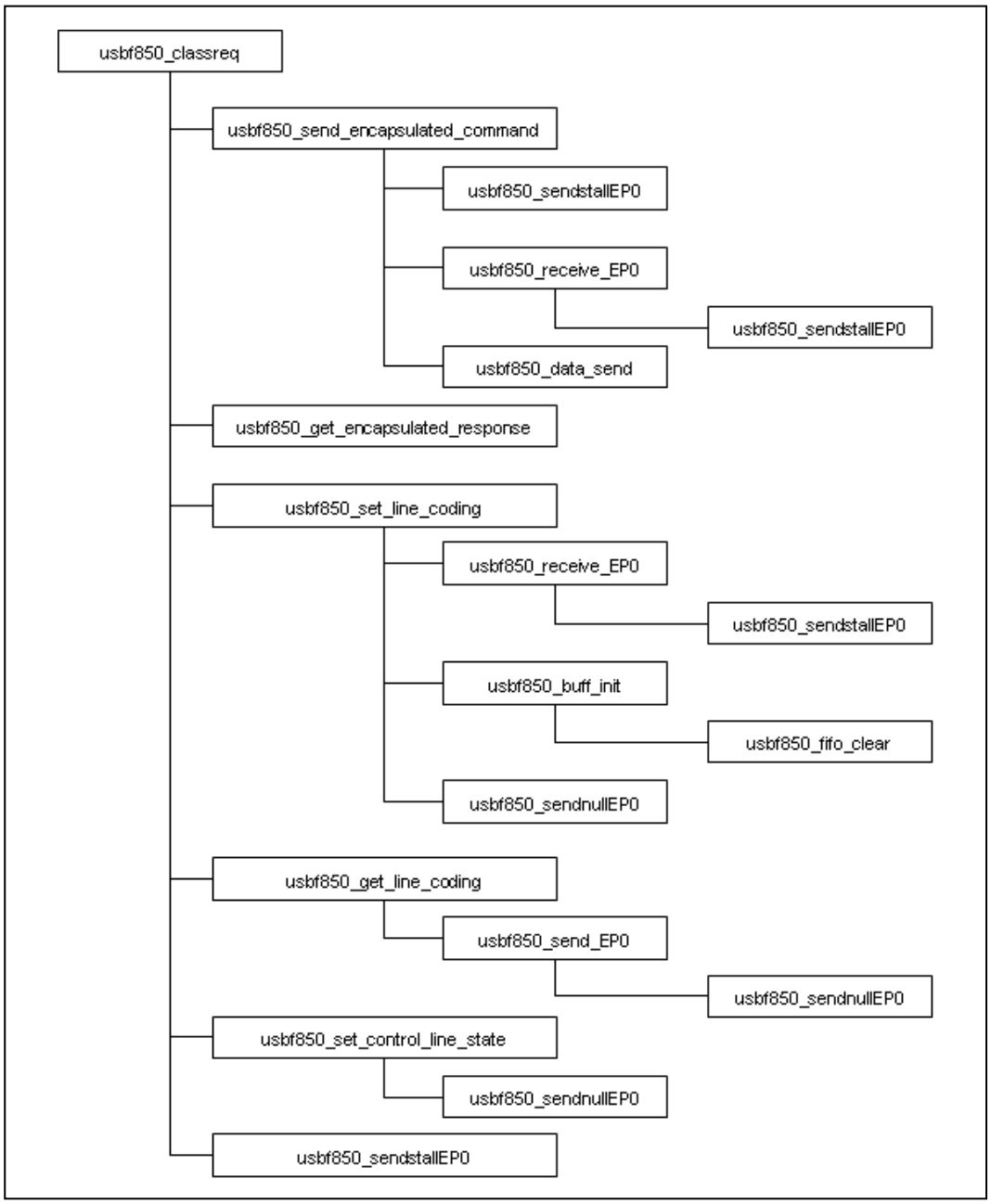


Figure 4-21. Calling Functions During Processing for USB Communication Class (1/2)

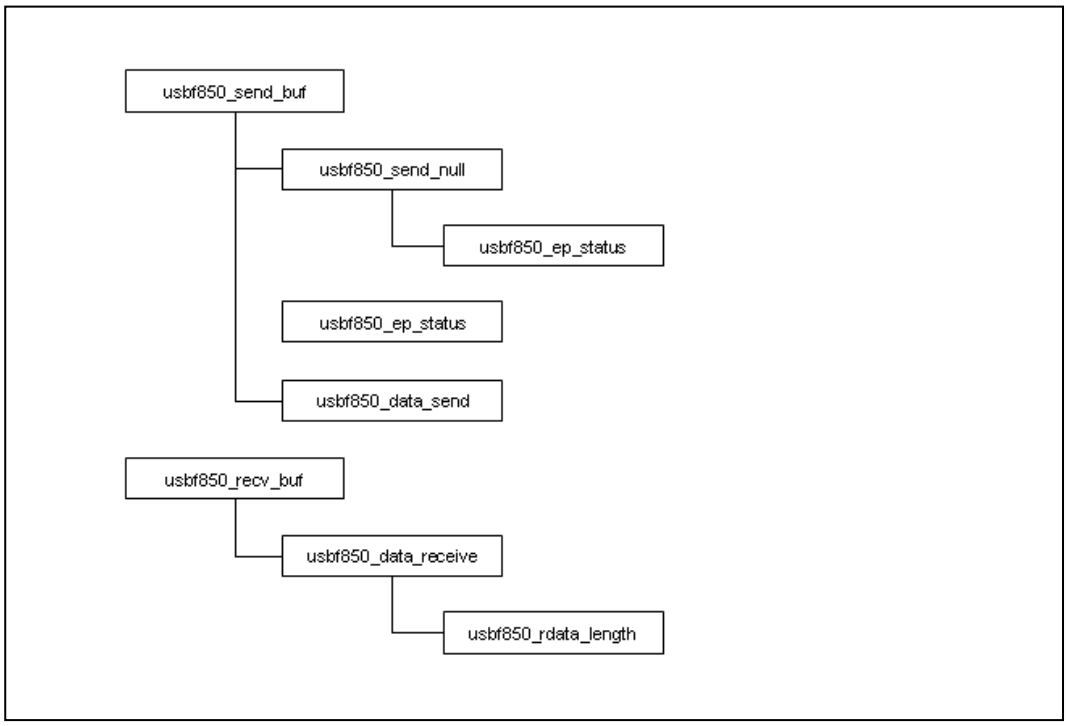


Figure 4-22. Calling Functions During Processing for USB Communication Class (2/2)

4.3.3 Function features

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

(1) Function description format

The functions are described in the following format.

Function name

Overview

An overview of the function is provided.

C coding 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 coding format

```
void main(void)
```

Parameters

None

Return values

None

Description

This function is called first when the sample program is executed.
The resume/suspend flag (`usbf850_rsuspd_flg`) is monitored after the USB initialization function (`usbf850_init`) is called. Suspend processing is performed when `usbf850_rsuspd_flg` is set to SUSPEND (0x00).

cpu_init

Overview

Initializes the CPU.

C coding format

```
void cpu_init(void)
```

Parameters

None

Return values

None

Description

This function is called during initialization.

This function specifies the settings necessary to use the USBF, such as initializing the H bus and USB clock.

SetProtectReg

Overview

Allows access to write-protected registers.

C coding format

```
void SetProtectReg(volatile UINT32 *dest_reg, UINT32 wr_dt, volatile UINT8 *prot_reg)
```

Parameters

Parameter	Description
`volatile UINT32 *dest_reg`	Address of the write-protected register
`UINT32 wr_dt`	Value to write
`volatile UINT8 *prot_reg`	Address of the protection command register

Return values

None

Description

This function is used to write data to write-protected registers.

(3) Functions for the USBF

usbf850_init

Overview

Initializes the USBF.

C coding format

```
void usbf850_init(void)
```

Parameters

None

Return values

None

Description

This function is called during initialization.

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 INTUSFA0I1 interrupt handler.

C coding format

```
void usbf850_intusbf0(void)
```

Parameters

None

Return values

None

Description

This function is called as the USB interrupt (INTUSFA0I1) handler.

The statuses of the endpoint for control transfer (endpoint 0) and the endpoint for bulk-out transfer (reception) (endpoint 2, endpoint 3) 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.

The BKO2DT interrupt is monitored at endpoint 3. If a BKO2DT interrupt occurs, a flag indicating that CDC data has been received (`usbf850_cdc_rdata_flg`) is updated.

usbf850_intusbf1

Overview

Executes the INTUSFA0I2 interrupt handler.

C coding format

```
void usbf850_intusbf1(void)
```

Parameters

None

Return values

None

Description

This function is called as the USB resume interrupt (INTUSFA0I2) handler.
The resume/suspend flag (`usbf850_rsuspd_flg`) is set to RESUME (0x01).

usbf850_multiple_data_send

Overview

Transmits USB data (MSC).

C coding format

```
INT32 usbf850_multiple_data_send(UINT8 *data, INT32 len, INT8 ep)
```

Parameters

Parameter	Description
UINT8 *data	Transmission data buffer pointer
INT32 len	Transmission data length
INT8 ep	Data transmission endpoint number

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

Description

This function can be used to transmit multiple packets.

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

usbf850_data_send

Overview

Transmits USB data (CDC).

C coding 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 (<Max packet size)
INT8 ep	Data transmission endpoint number

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

Description

This function is used to transmit a single packet.

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

usbf850_multiple_data_receive

Overview

Receives USB data (MSC).

C coding format

```
INT32 usbf850_multiple_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 is used to receive multiple packets, but can only be used for the MSC.

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

usbf850_data_receive

Overview

Receives USB data. (CDC)

C coding 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 (< Max Packet Size)
INT8 ep	Data reception endpoint number

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

Description

This function is used to receive a single packet.

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

usbf850_rdata_length

Overview

Acquires the USB reception data length.

C coding format

```
void usbf850_rdata_length(INT32 *len , INT8 ep)
```

Parameters

Parameter	Description
INT32* len	Reception data length storage address pointer
INT8 ep	Data reception endpoint number

Return values

None

Description

This function reads the reception data length of the specified endpoint.

usbf850_send_EP0

Overview

Transmits USB data for endpoint 0.

C coding format

```
INT32 usbf850_send_EP0(UINT8* data, INT32 len)
```

Parameters

Parameter	Description
UINT* data	Transmission data buffer pointer
INT32 len	Transmission data size

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 transmission FIFO for endpoint 0 byte by byte.

usbf850_receive_EP0

Overview

Receives USB data for endpoint 0.

C coding format

```
INT32 usbf850_receive_EP0(UINT8* data, INT32 len)
```

Parameters

Parameter	Description
UINT* data	Reception data buffer pointer
INT32 len	Reception data size

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

Description

This function reads data from the reception FIFO for endpoint 0 byte by byte and stores the data into the reception data buffer.

usbf850_send_null

Overview

Transmits a NULL packet to Bulk/Interrupt In Endpoint.

C coding format

```
INT32 usbf850_send_null(INT8 ep)
```

Parameters

Parameter	Description
INT8 ep	Data transmission endpoint number

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

Description

This function clears the FIFO of the specified endpoint (for transmission), sets the bit that indicates the end of the data to 1, and transmits a NULL packet from the USBF.

usbf850_sendnullEP0

Overview

Transmits a NULL packet for endpoint 0.

C coding format

```
void usbf850_sendnullEP0(void)
```

Parameters

None

Return values

None

Description

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

usbf850_sendstallEP0

Overview

Performs a STALL response for endpoint 0.

C coding format

```
void usbf850_sendstallEP0(void)
```

Parameters

None

Return values

None

Description

This function sets the bit indicating the use of STALL handshaking to make the USBF perform a STALL response.

usbf850_ep_status

Overview

Notifies the FIFO status for Bulk/Interrupt In endpoint.

C coding format

```
INT32 usbf850_ep_status(INT8 ep)
```

Parameters

Parameter	Description
INT8 ep	Data transmission endpoint number

Return values

Symbol	Description
DEV_OK	Normal completion
DEV_RESET	Bus reset being processed
DEV_ERROR	Abnormal termination

Description

This function notifies the FIFO status of the specified endpoint (for transmission).

usbf850_fifo_clear

Overview

Clears the FIFO for Bulk/Interrupt endpoint.

C coding format

```
void usbf850_fifo_clear(INT8 in_ep, INT8 out_ep)
```

Parameters

Parameter	Description
INT8 in_ep	Data transmission endpoint
INT8 out_ep	Data reception endpoint

Return values

None

Description

This function clears the FIFO of the specified endpoint (Bulk/Interrupt) and clears the data reception flag (usbf850_rdata_flg).

usbf850_standardreq

Overview

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

C coding 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 coding 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`).

(4) Functions for USB mass storage class processing

usbf850_classreq

Overview

Processes MSC class requests.

C coding format

```
void usbf850_classreq(USB_SETUP *req_data)
```

Parameters

Parameter	Description
`USB_SETUP *req_data`	Request data storage pointer address

Return values

None

Description

This function is called by the CPUDEC interrupt source of the INTUSFA0I1 interrupt servicing. If the decoded request is a CDC-specific request, the respective request processing function is called. In all other cases, STALL is transmitted to endpoint 0.

usbf850_blkonly_mass_storage_reset

Overview

Processes MASS_STORAGE_RESET requests.

C coding 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.

After that, this function transmits a NULL packet from endpoint 0.

usbf850_max_lun

Overview

Processes GET_MAX_LUN requests.

C coding 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_rx_cbw

Overview

Receives CBW data.

C coding 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 coding 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, determines the command type (NO DATA, DATA IN (write command), or DATA OUT (read command)), and executes each command processing.

usbf850_cbw_error

Overview

Processes errors in CBW data.

C coding 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 coding 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 the CSW format.

usbf850_data_in

Overview

Executes SCSI data-in commands.

C coding 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 the CSW format.

usbf850_data_out

Overview

Executes SCSI data-out commands.

C coding 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 the CSW format.

usbf850_csw_ret

Overview

Processes CSW responses.

C coding 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 the USB.

usbf850_bulkin_stall

C coding format

```
void usbf850_bulkin_stall(void)
```

Parameters

None

Return values

None

Description

This function clears the FIFO of endpoint 1 and then sets up the endpoint to issue a STALL response.

usbf850_bulkout_stall

C coding format

```
void usbf850_bulkout_stall(void)
```

Parameters

None

Return values

None

Description

This function clears the FIFO of endpoint 2 and then sets up the endpoint to issue a STALL response.

(5) SCSI command processing functions

scsi_command_to_ata

Overview

Executes SCSI commands.

C coding 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.

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.
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

This function determines the SCSI command type and executes the command.

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 coding 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 coding 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 coding format

```
INT32 ata_start_stop_unit(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_synchronize_cache

Overview

Executes the SYNCHRONIZE_CACHE command.

C coding format

```
INT32 ata_synchronize_cache(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_request_sense

Overview

Executes the REQUEST_SENSE command.

C coding 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

Processing result

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 coding 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 coding 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 coding 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_sence

Overview

Executes the MODE_SENSE(6) command.

C coding 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_sence10

Overview

Executes the MODE_SENSE(10) command.

C coding 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 coding 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 coding 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 coding 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 coding 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 WRITE(6) command.

C coding 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 coding 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 coding 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 coding 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 coding 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 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 reads and discards the received data.

scsi_to_usb

Overview

Transmits USB data (SCSI command).

C coding 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_ERROR_READ	A transfer direction error occurred for a read command.

Description

This function calls the USB data transmission processing function (usbf850_multiple_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.

(6) Functions for USB communication class processing

usbf850_cdc_classreq

Overview

Processes CDC class requests.

C coding format

```
void usbf850_cdc_classreq(USB_SETUP *req_data)
```

Parameters

Parameter	Description
`USB_SETUP *req_data`	Request data storage pointer address

Return values

None

Description

This function is called by the CPUDEC interrupt source of the INTUSFA0I1 interrupt servicing. If the decoded request is a CDC-specific request, the respective request processing function is called. In all other cases, STALL is transmitted to endpoint 0.

usbf850_send_encapsulated_command

Overview

Processes Send Encapsulated Command requests.

C coding format

```
void usbf850_send_encapsulated_command(void)
```

Parameters

None

Return values

None

Description

This function calls the data reception processing function (`usbf850_data_receive`) to load the data received at endpoint 0, and then calls the data transmission processing function (`usbf850_data_send`) to transmit the data from endpoint 2 by using bulk-in transfer.

usbf850_set_line_coding

Overview

Processes Set Line Coding requests.

C coding format

```
void usbf850_set_line_coding(void)
```

Parameters

None

Return values

None

Description

This function calls the data reception processing function (`usbf850_data_receive`) to load the data received at endpoint 0, and writes it to the UART_MODE_INFO structure. Moreover, after setting the UART mode, including the transfer rate and data length based on that value, this function calls the NULL packet transmission processing function for endpoint 0 (`usbf850_sendnullEP0`).

usbf850_get_line_coding

Overview

Processes Get Line Coding requests.

C coding format

```
void usbf850_get_line_coding(void)
```

Parameters

None

Return values

None

Description

This function calls the data transmission processing function (`usbf850_data_send`) to transmit the value of the UART_MODE_INFO structure from endpoint 0.

usbf850_set_control_line_state

Overview

Processes Set Control Line State requests.

C coding format

```
void usbf850_set_control_line_state(void)
```

Parameters

None

Return values

None

Description

This function calls the NULL packet transmission processing function for endpoint 0 (`usbf850_sendnullEP0`).

4.4 Data Structures

The sample program 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;
```

Figure 4-23. USB Device Request Structure

(2) CBW data structure

This structure is defined in usbf850_storage.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;
```

Figure 4-24. CBW Data Structure

(3) CSW data structure

This structure is defined in usbf850_storage.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;
```

Figure 4-25. CSW Data Structure

(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;
```

Figure 4-26. SCSI Sense Data Structure

5. DEVELOPMENT ENVIRONMENT

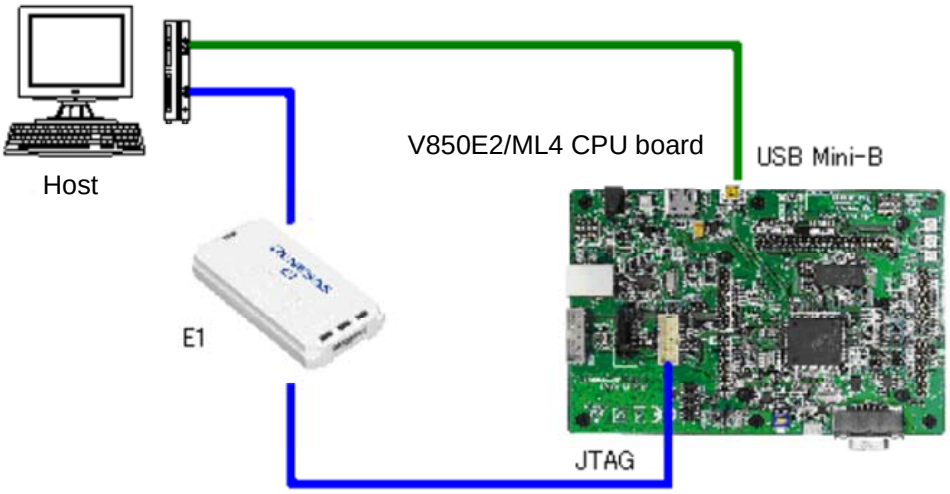
This chapter provides an example of creating an environment for developing an application program that uses the USB multifunction sample program for the V850E2/ML4 and the procedure for debugging the application.

5.1 Used Products

This section describes the hardware and software tool products used for development.

5.1.1 System components

Figure 5-1 shows the components used in a system that uses the sample program.



Remark For a diagram and details about the ports on the V850E2/ML4 CPU board, see **6. OVERVIEW OF V850E2/ML4 CPU BOARD**.

Figure 5-1. System Components Used in Development Environment

5.1.2 Program development

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

Table 5-1. Example of the Components Used in a Program Development Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT™-compatible computer using Windows XP, Windows Vista®, or Windows 7
Software	Integrated development tool	CubeSuite+	V1.00.01
	Compiler	CX	V1.20

5.1.3 Debugging

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

Table 5-2. Example of the Components Used in a Debugging Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT-compatible computer using Windows XP, Windows Vista, or Windows 7
	Target board	V850E2/ML4 CPU board	Made by Renesas Electronics
	Emulator	E1 emulator	Made by Renesas Electronics
	USB cable	–	miniB-to-A connector cable
Software	Integrated development tool/debugger	CubeSuite+	V1.00.01
Files	Device file	V850 device-specific information for CubeSuite+	V1.00.02
	Project files	–	**Note**

Note Files used when creating a system by using CubeSuite+ are included with the sample program.

5.2 Setting up the CubeSuite+ Environment

This section describes the preparations required for developing and debugging a system using CubeSuite+ by using the products described in **5.1 Used Products**.

5.2.1 Preparing the host environment

Create a dedicated workspace on the host for debugging.

(1) Installing the CubeSuite+ integrated development tool

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

(2) Installing the driver

Store the set of files provided with the sample program in any directory without changing the folder structure.

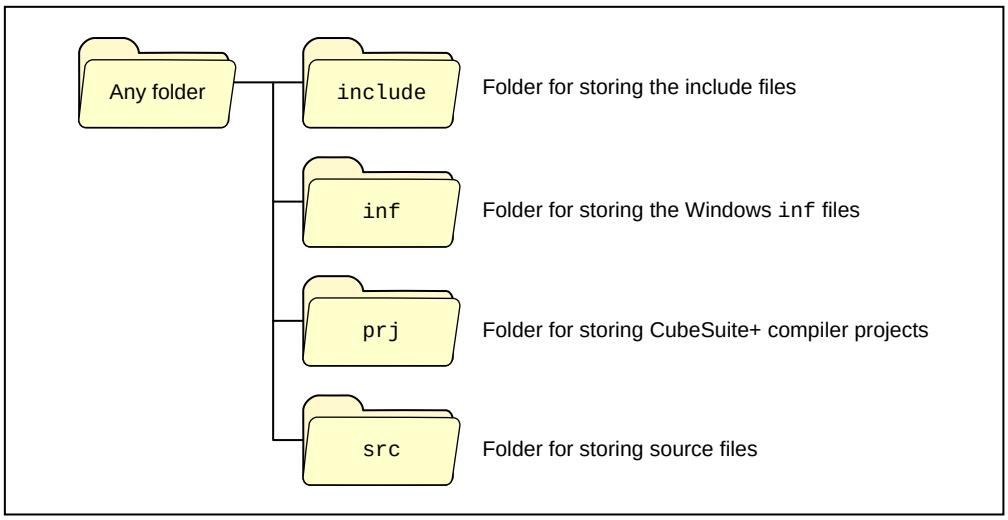


Figure 5-2. Folder Structure of Sample Program (CubeSuite+ Version)

(3) Setting up the workspace

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

<1> Start CubeSuite+ and then select **Open** on the **File** menu.

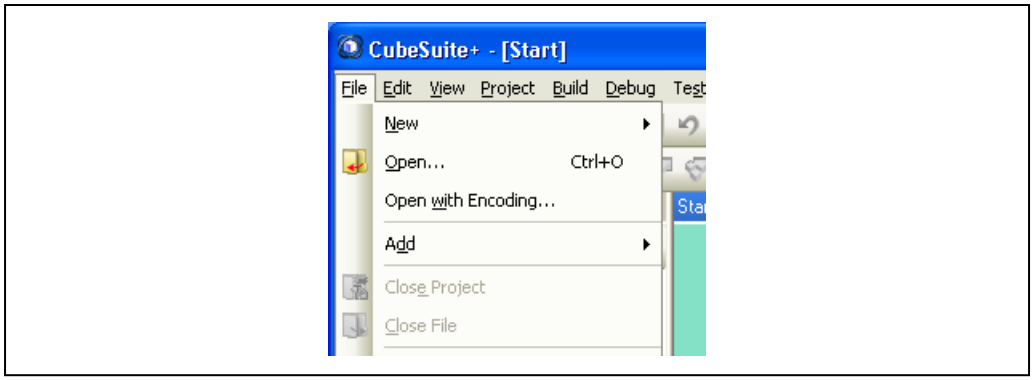


Figure 5-3. Selecting Open

<2> In the **Open File** dialog box, specify the CubeSuite+ project file in the `prj` folder, which is the sample program installation directory.

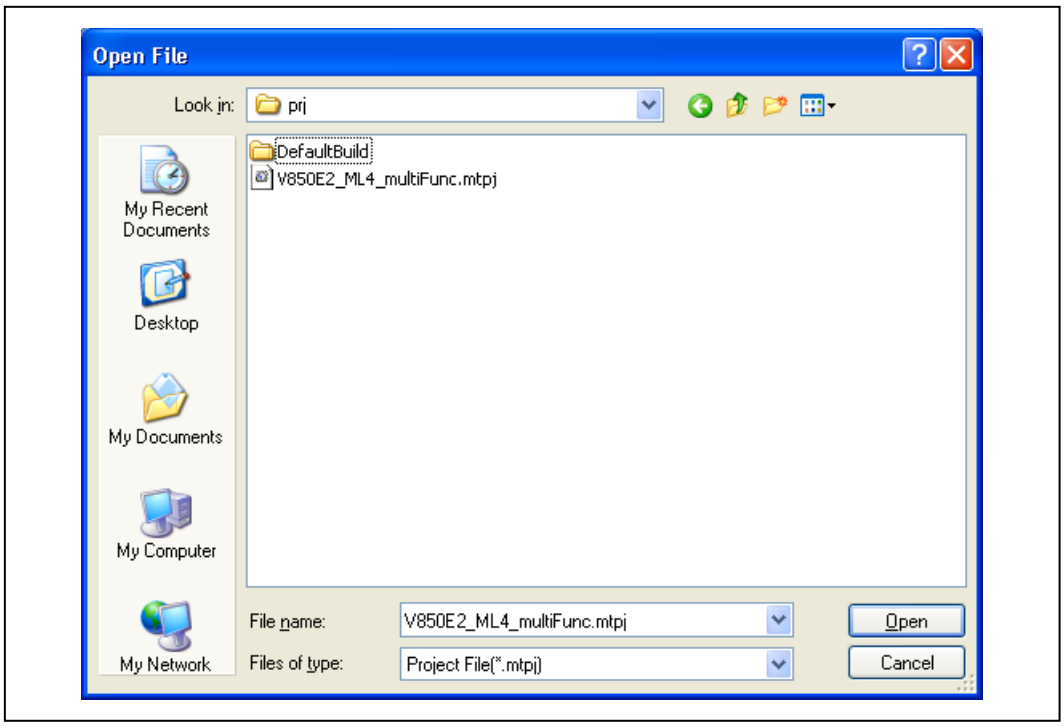


Figure 5-4. Specifying the CubeSuite+ Project File

(4) Setting up the building tool

The procedure to select the version of CX to be used as the build tool and use V850E2E1 as the debugging tool is described below.

<1> On the CubeSuite+ **Project Tree** window, select **CX (Build Tool)**.

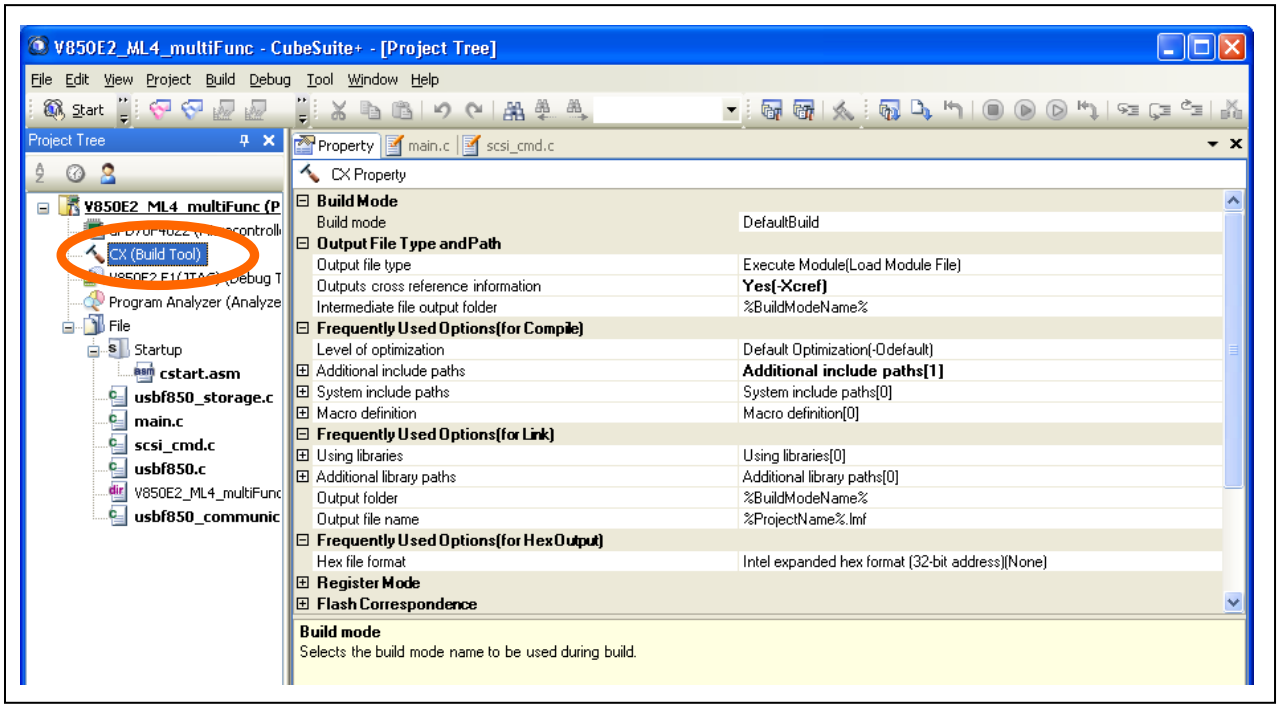


Figure 5-5. Selecting the Building Tool

<2> Select **Version Select** and specify **Always latest version which was installed** for **Using compiler package version**.

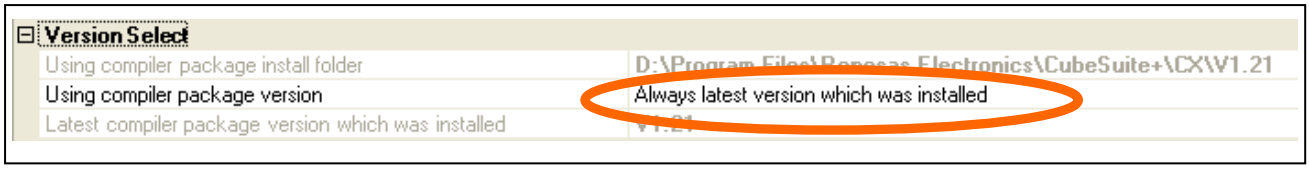


Figure 5-6. Specifying the Compiler Package

<3> In the **Project Tree** window, right-click **V850E2 E1(JTAG)(Debug Tool)**, point to **Using Debug Tool**, and then select **V850E2 E1(JTAG)**.

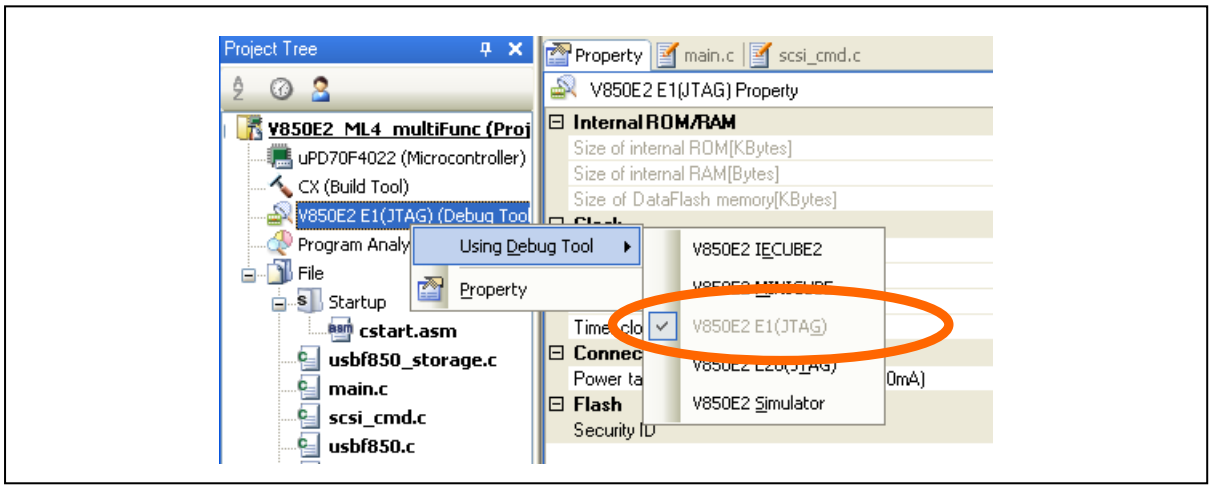


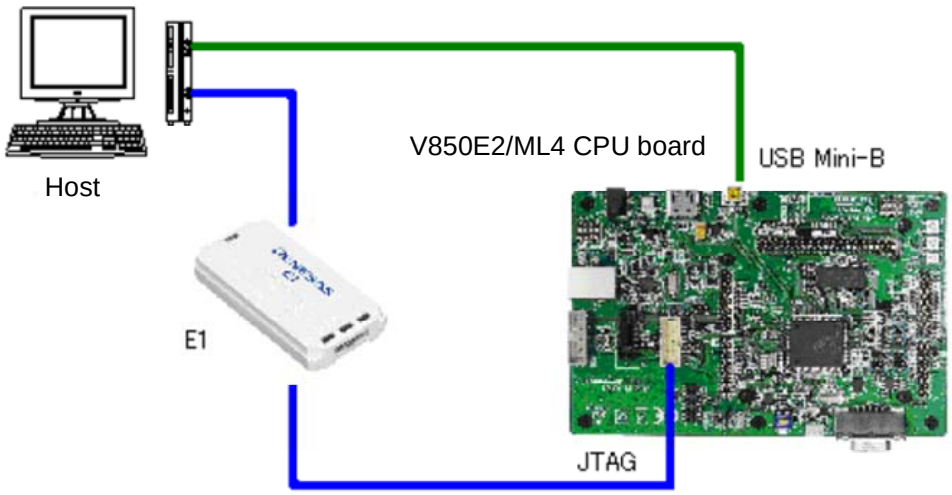
Figure 5-7. Selecting the Debugging Tool

5.2.2 Setting up the target environment

Connect the target device to use for debugging.

(1) Connecting to the debugging port

Connect the V850E2/ML4 CPU board to the host by using an E1 cable for debugging. Also connect the USB miniB type connector of the V850E2/ML4 CPU board to the USB connector of the host.



Remark For a diagram and details about the ports of the V850E2/ML4 CPU board, see **6. OVERVIEW OF V850E2/ML4 CPU BOARD**.

Figure 5-8. Connecting the V850E2/ML4 CPU Board

(2) Installing the host driver

Drivers must be installed to connect the V850E2/ML4 CPU board to the host by using the USB miniB type connector.

Use the standard Windows CDC and MSD host drivers. For details, see **5.4 Checking the Operation**.

5.3 CubeSuite+ Environment Debugging

This section describes the procedure for debugging an application program that was developed using the workspace described in **5.2 Setting up the CubeSuite+ Environment**

5.3.1 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.
On the CubeSuite+ **Build** menu, select **Build Project** to generate a load module.

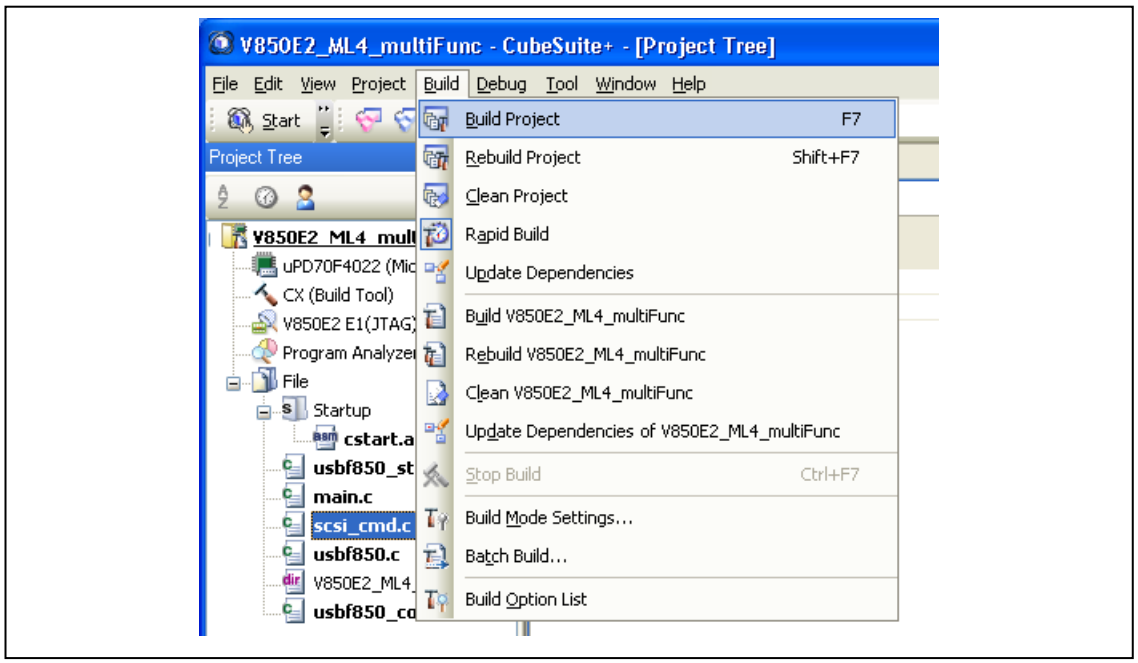


Figure 5-9. Selecting Build Project

5.3.2 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 V850E2/ML4 CPU board by using CubeSuite+ is described below.

<1> On the **Debug** menu, select **Download** to start the debugger.

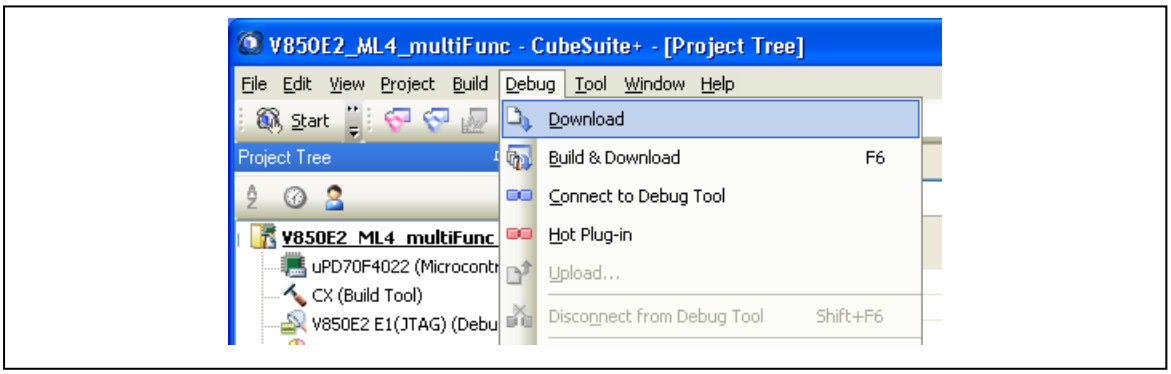


Figure 5-10. Starting the Debugger

<2> Downloading the load module is started by the debugging tool.

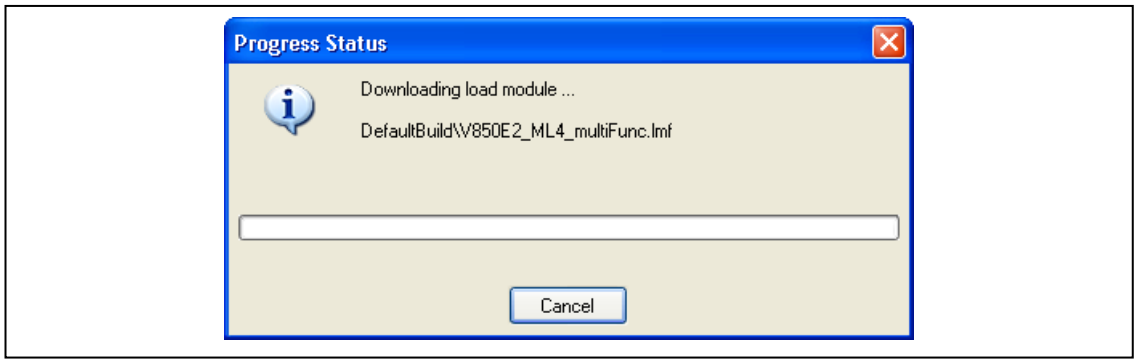


Figure 5-11. Downloading the Load Module File

(2) Executing the program

Click the button in the CubeSuite+ window or select **Run** on the **Debug** menu.

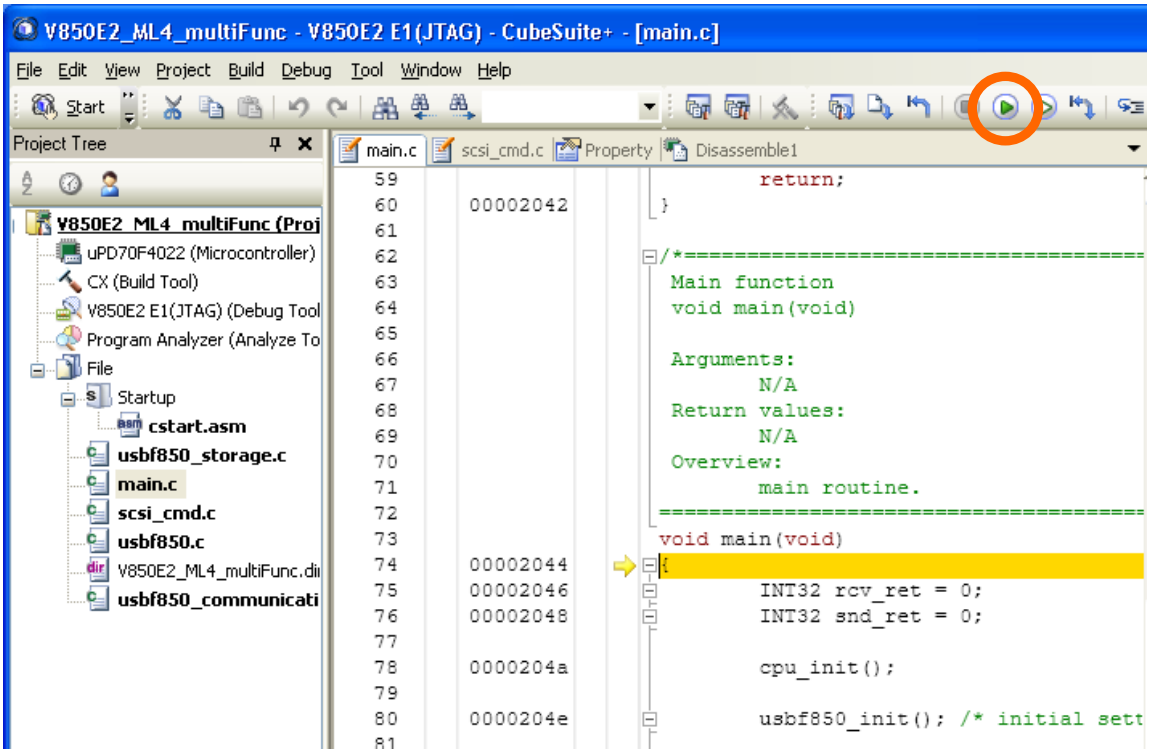


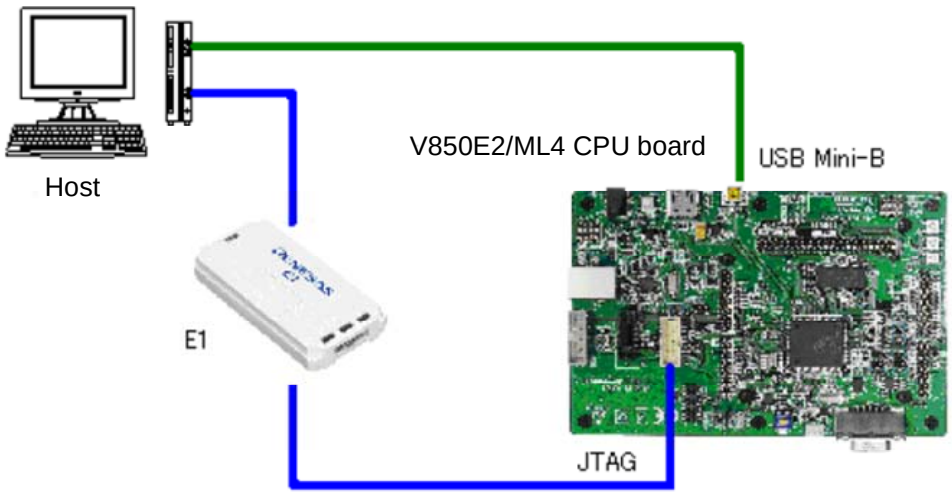
Figure 5-12. Executing the Program

5.3.3 Setting up the target environment

Connect the target device to use for debugging.

(1) Connecting to the debugging port

Connect the V850E2/ML4 CPU board to the host by using an E1 cable for debugging. Also connect the USB miniB type connector of the V850E2/ML4 CPU board to the USB connector of the host.



Remark For a diagram and details about the ports of the V850E2/ML4 CPU board, see **6. OVERVIEW OF V850E2/ML4 CPU BOARD**.

Figure 5-13. Connecting the V850E2/ML4 CPU Board

(2) Installing the host driver

Drivers must be installed to connect the V850E2/ML4 CPU board to the host by using the USB miniB type connector.

Use the standard Windows CDC and MSD host drivers. For details, see **5.4 Checking the Operation**.

5.4 Checking the Operation

This section describes the procedure for checking the results after executing the sample program in the CubeSuite+ environment.

(1) Connecting to the USB miniB connector

Connect the USB miniB type connector of the V850E2/ML4 CPU board to the USB port of the host by using a USB cable.

(2) Installing the host driver

<1> When the V850E2/ML4 CPU board connection is recognized by the host, the "Found New Hardware" message is displayed and the **Found New Hardware Wizard** starts.

<2> When the **Welcome to the Found New Hardware Wizard** window is displayed, select **No, not this time** and then click the **Next** button.

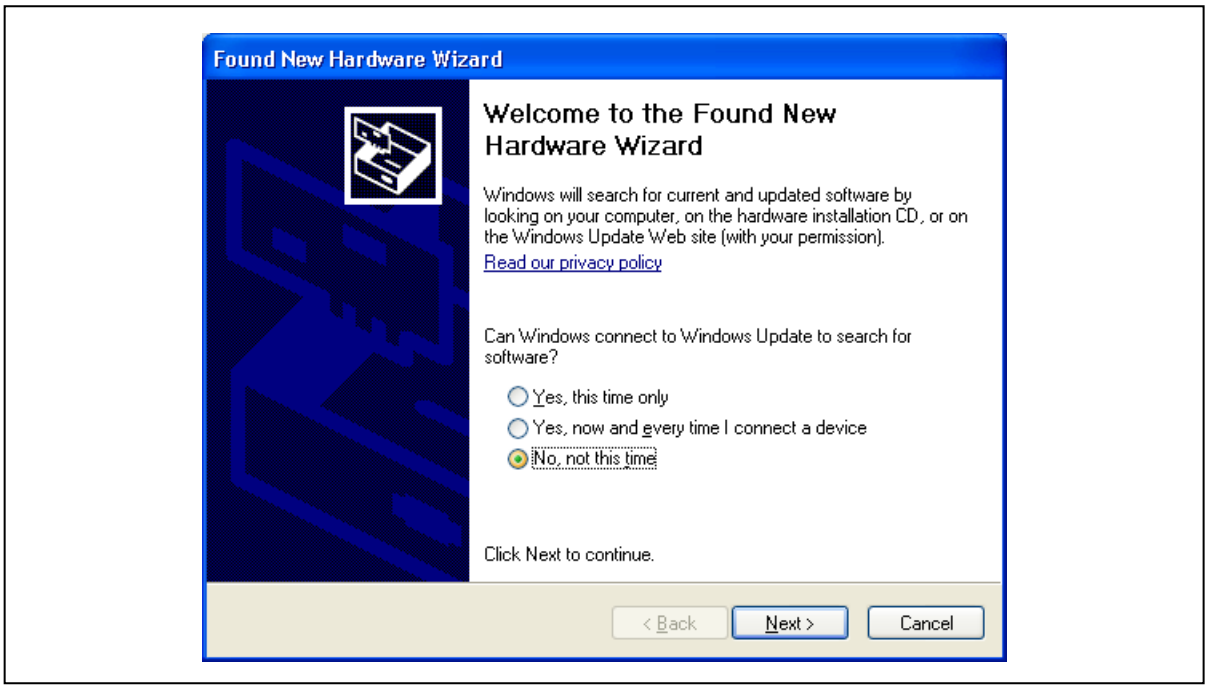


Figure 5-14. Found New Hardware Wizard (1)

<3> When the following window is displayed, select **Install from a list or specific location [Advanced]** and then click the **Next** button.

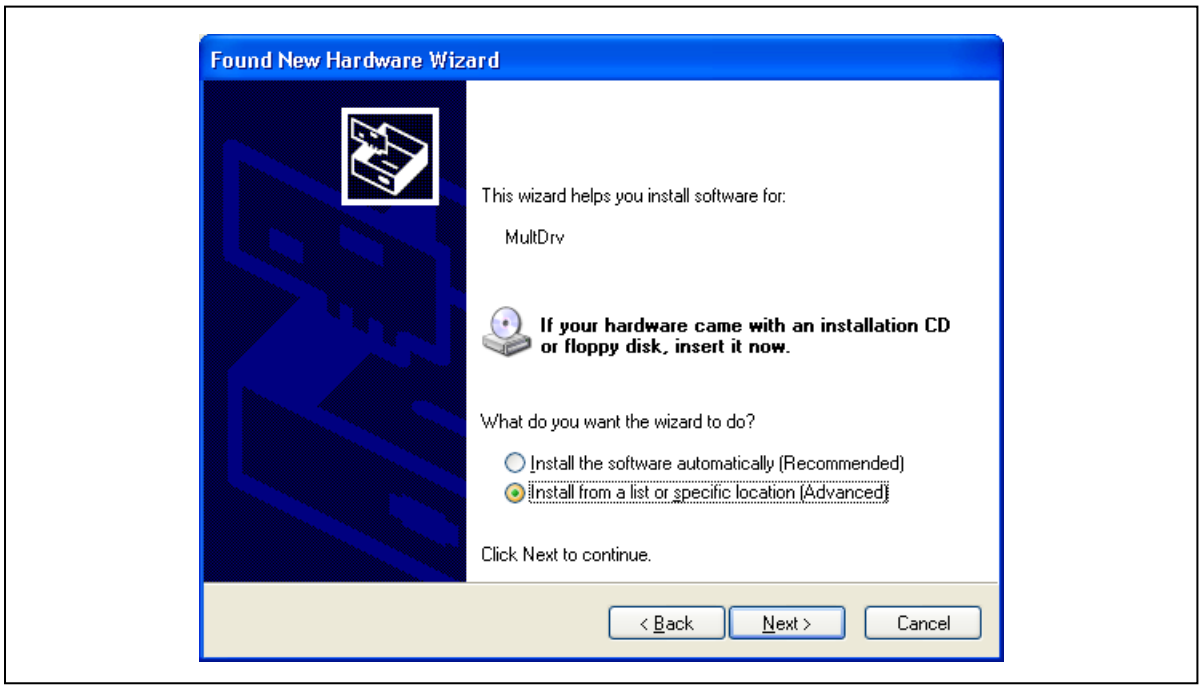


Figure 5-15. Found New Hardware Wizard (2)

<4> Select and install the `inf` file belonging to the sample program.

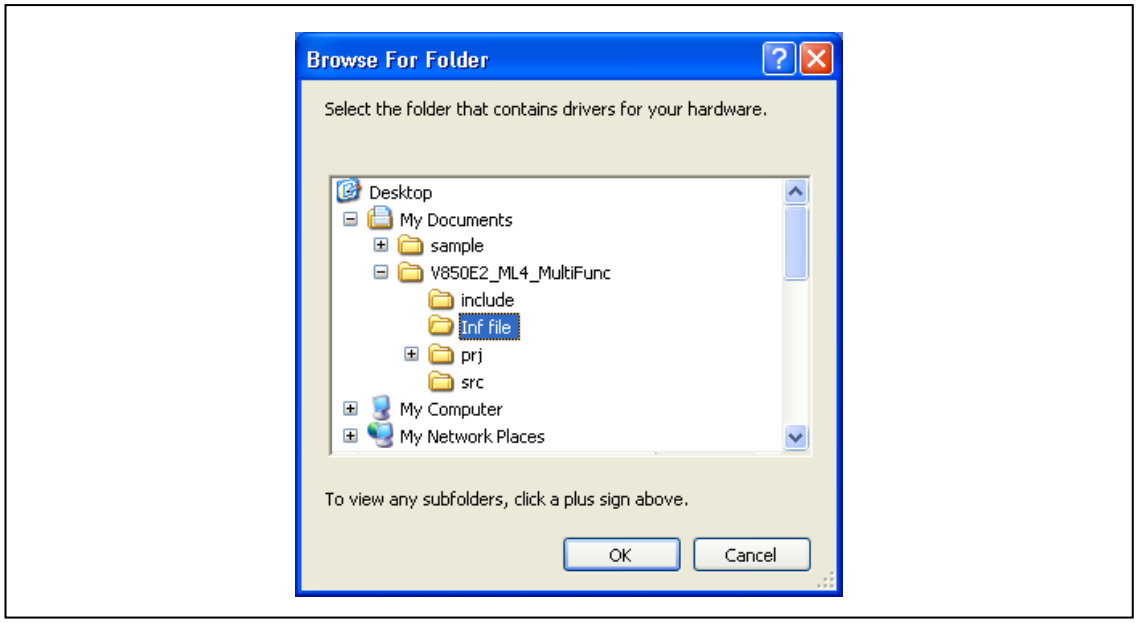


Figure 5-16. Found New Hardware Wizard (3)

(3) Checking the connection of USB devices

Open the **Device Manager** window. Make sure that **Renesas StorageFncDriver USB Device** is displayed in the **Disk drives** category, **Renesas Electronics V850E2/ML4 Virtual UART(COM*n*)** is displayed in the **Ports (COM & LPT)** category, and **USB Composite Device** and **USB Mass Storage Device** are displayed in the **Universal Serial Bus controllers** category.

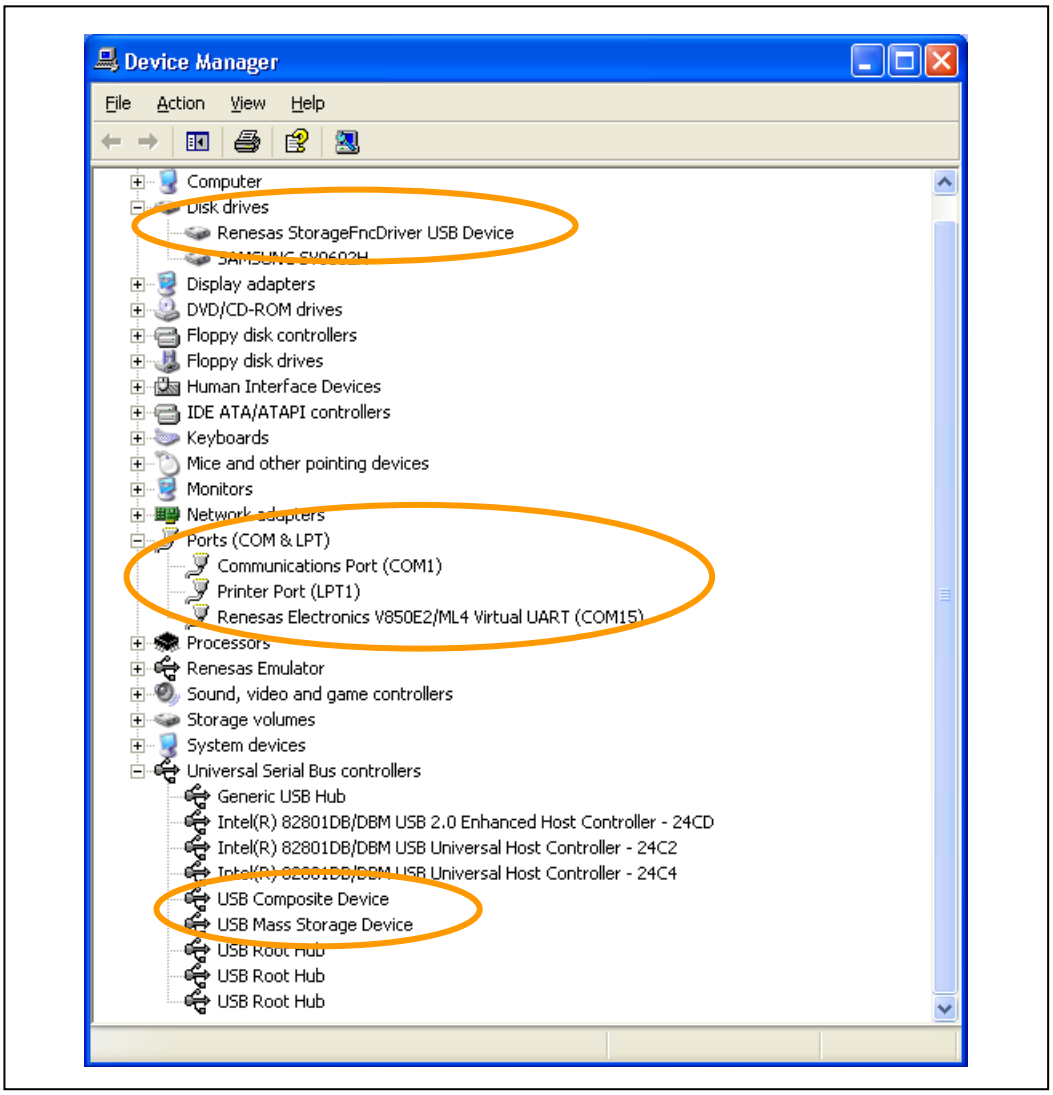


Figure 5-17. Checking Device Connection in the Device Manager

(4) Formatting removable disks

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

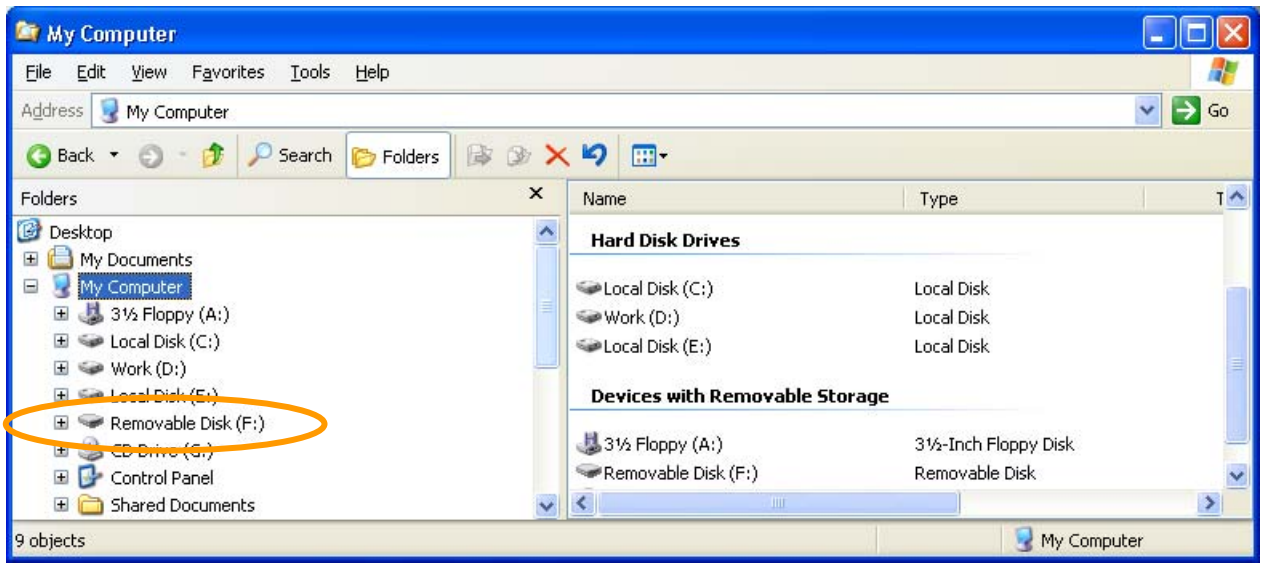


Figure 5-18. Checking Removable Disk

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

<1> Click **Removable Disk**. When the **Disk is not formatted** dialog box is displayed, click the **Yes** button.

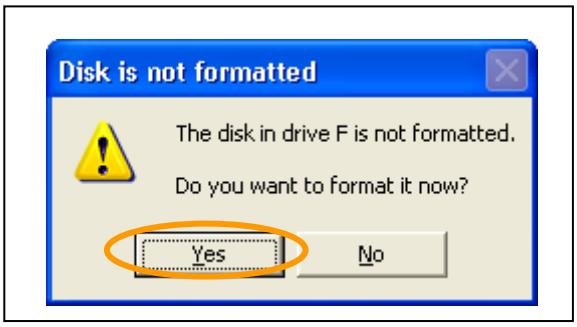


Figure 5-19. Format Confirmation Dialog Box

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

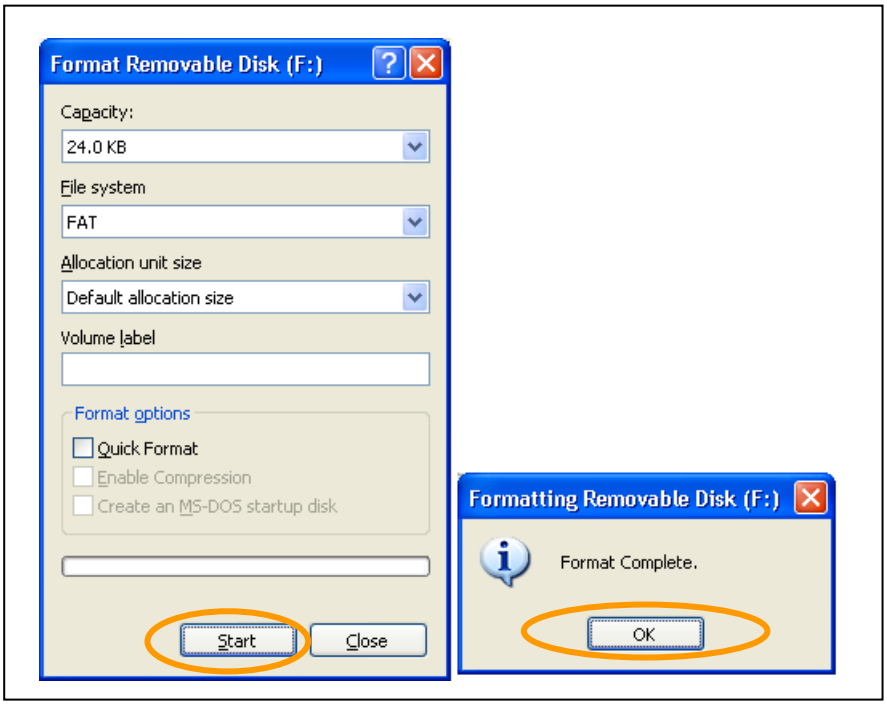


Figure 5-20. Format Menu and Completion Dialog Box

<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 `MSC Test` folder in the local disk.

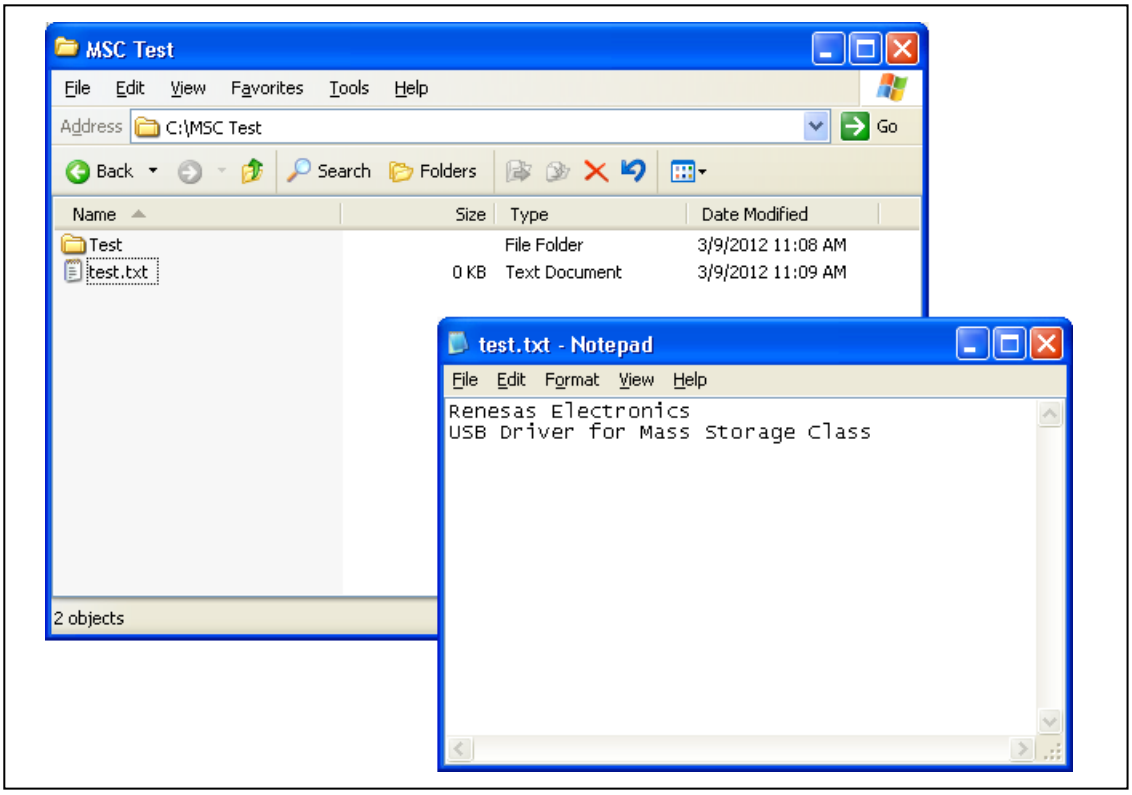


Figure 5-21. MSC Test Folder and Test Data File

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

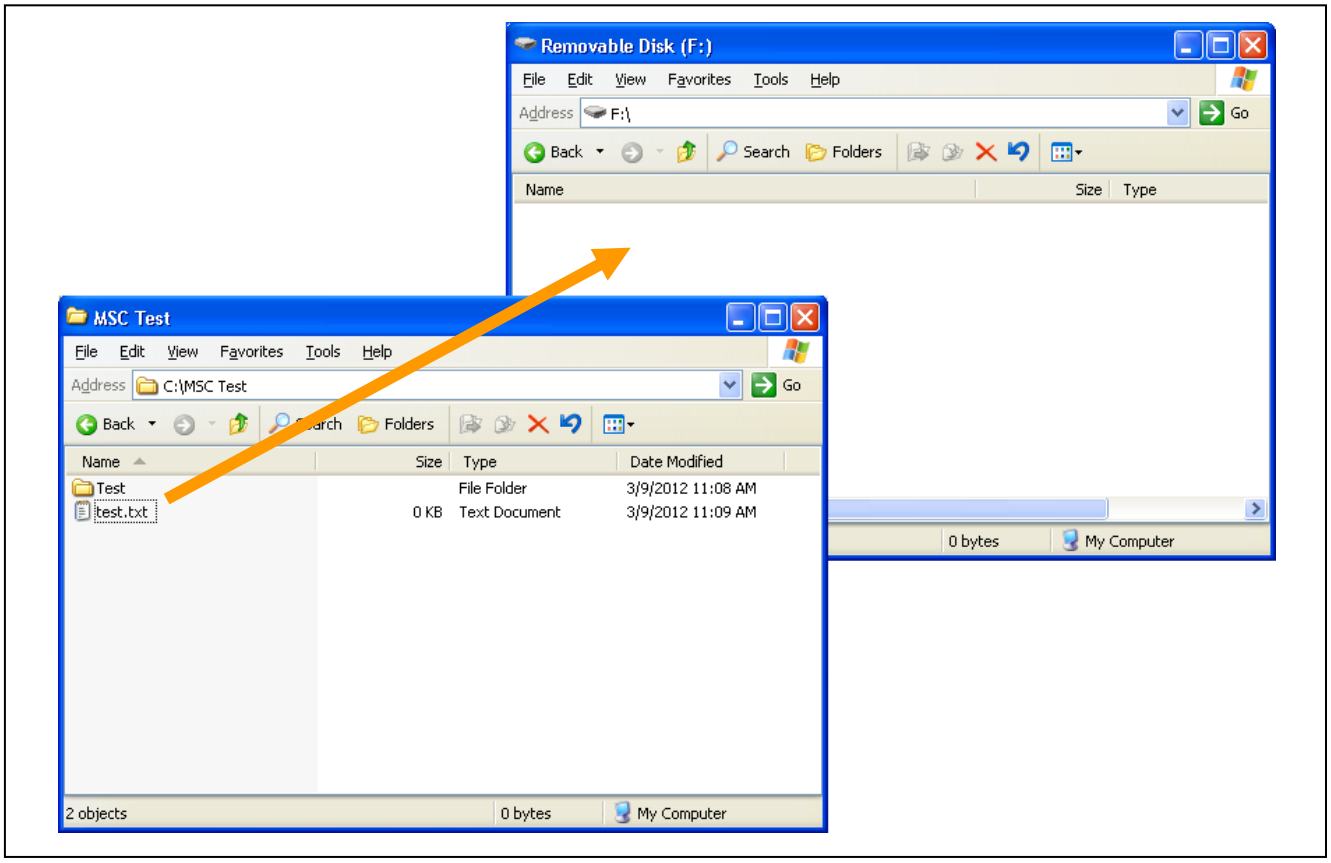


Figure 5-22. Copying the Test Data File

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

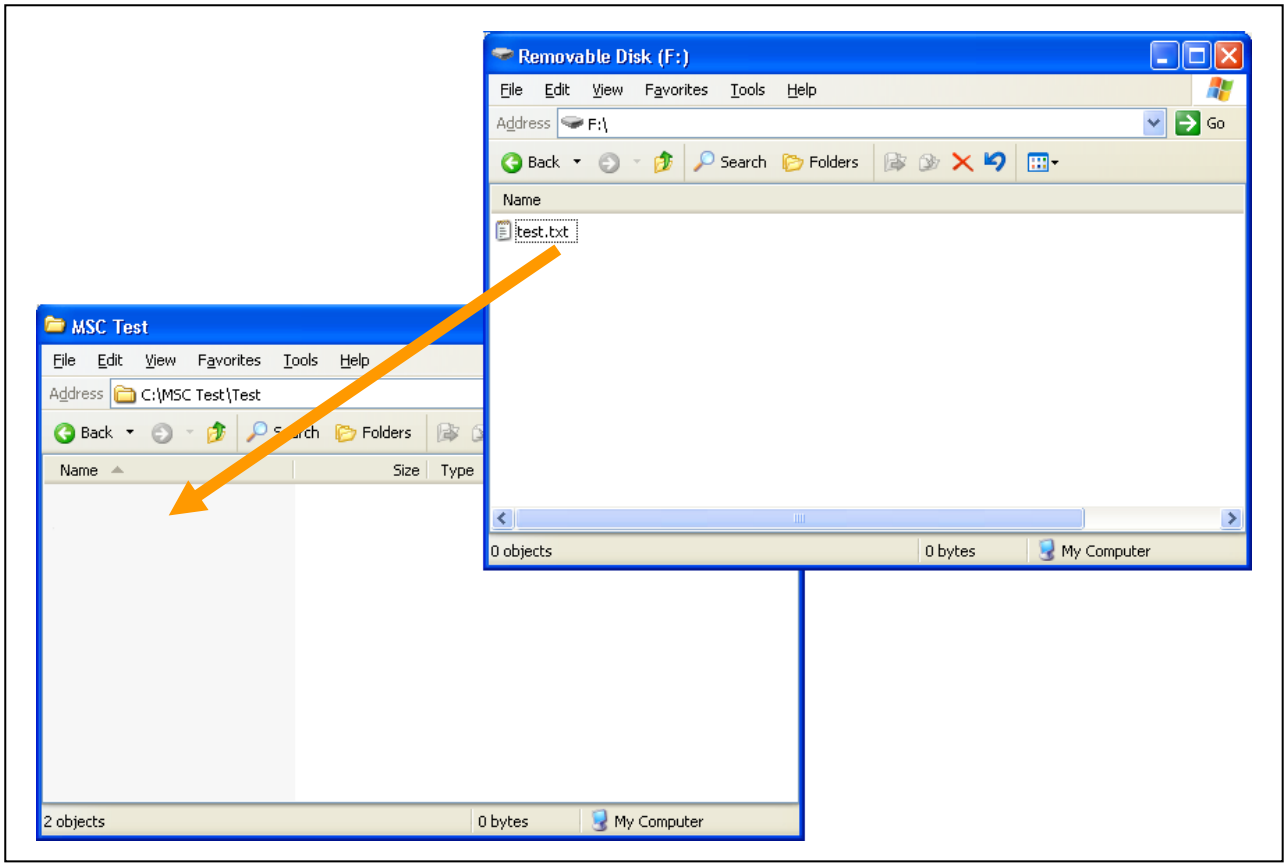


Figure 5-23. Recopying the Test Data File

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

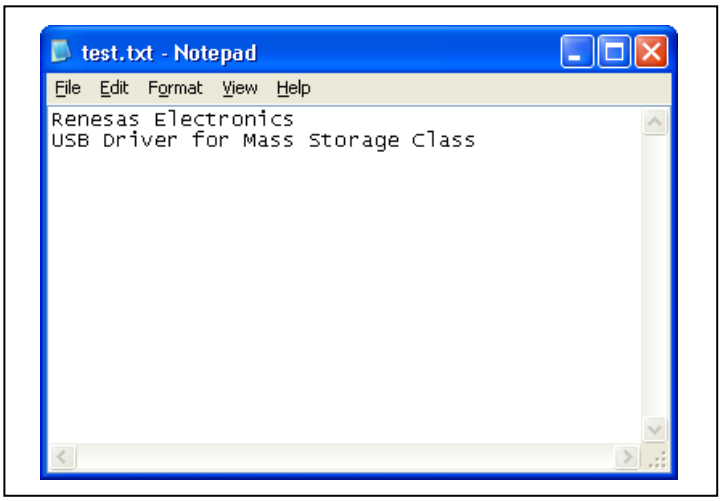


Figure 5-24. Checking the Test Data File

Remark 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.

(6) Checking the Operation of the COM Port

Start the terminal software on the host (Tera Term, etc.) and open the COM port recognized as **Renesas Electronics V850E2/ML4 Virtual UART(COM15)**. This is a program used to call back the received data. Check that the key input to the terminal has been called back and is displayed.

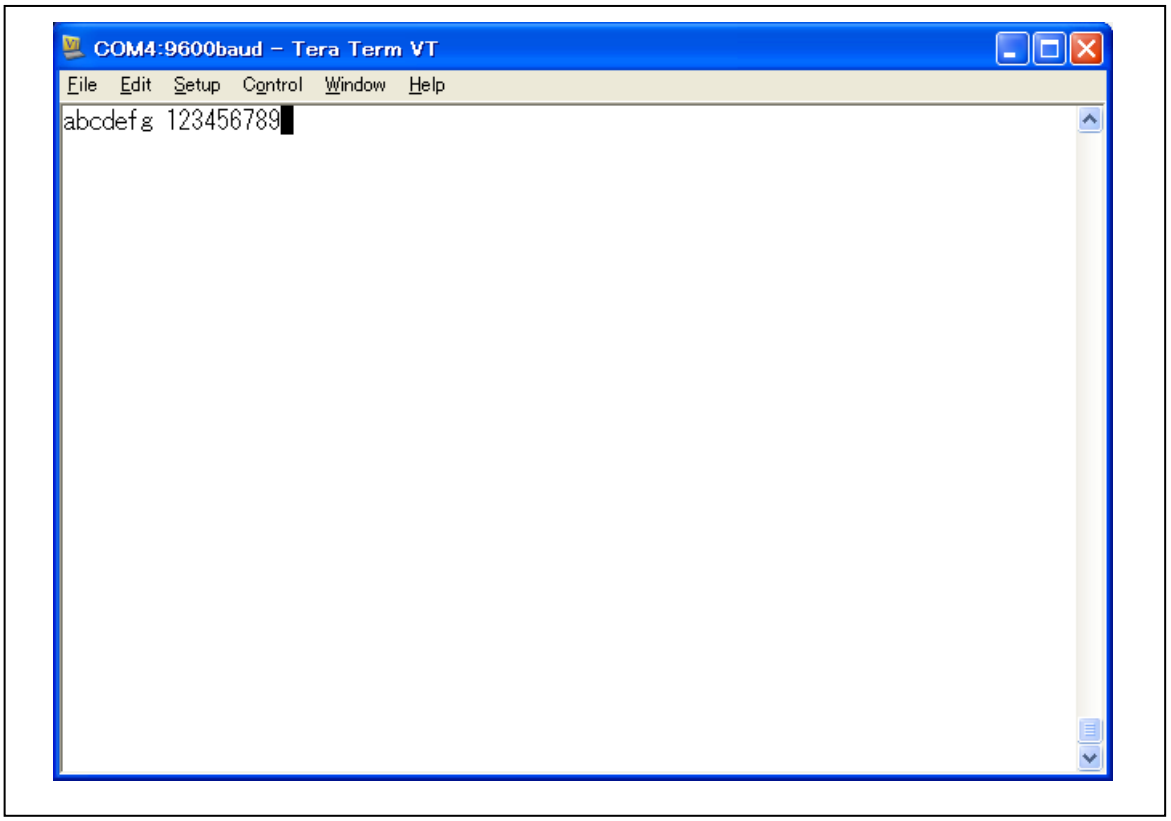


Figure 5-25. Checking the Operation of the COM Port

6. OVERVIEW OF V850E2/ML4 CPU BOARD

6.1 Overview

The V850E2/ML4 CPU board and other optional boards are provided to evaluate the functionality and performance of Renesas Electronics' V850E2/ML4 microcontrollers and to develop and evaluate application software for these microcontrollers.

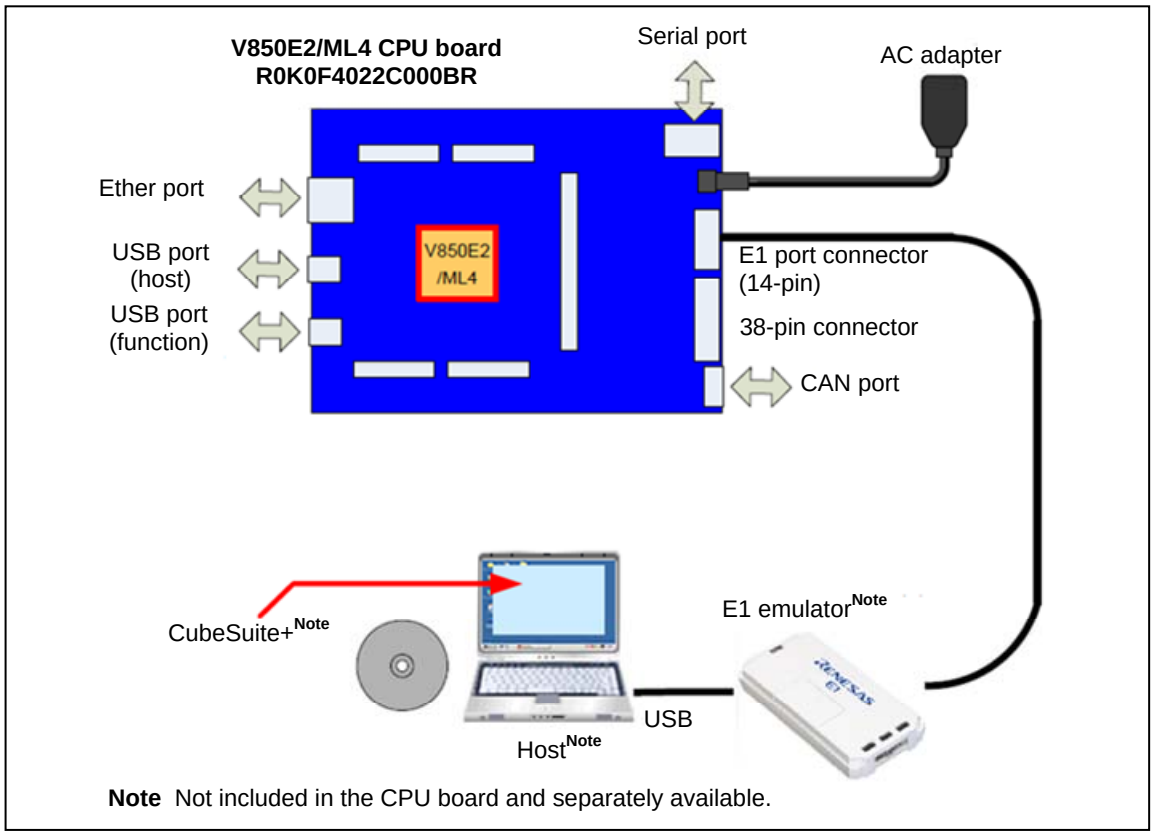


Figure 6-1. Connecting the V850E2/ML4 CPU Board (Illustration)

6.2 Features

The V850E2/ML4 CPU board (No.: R0K0F4022C000BR) has the following features:

- One 16 MB SDRAM (16-bit bus connection) and one 16 KB EEPROM provided as standard external memories.
- RS-232C, USB, Ethernet, and CAN connectors provided for interfacing with internal V850E2/ML4 peripherals.
- The USB connector is a standard series A receptacle. The basic pattern of the connector also enables the mounting of a Mini-B receptacle for evaluating the USB host module.
- The data bus, address bus and internal peripheral pins of the V850E2/ML4 are connected to an expansion connector, allowing users to connect a measuring instrument to evaluate the timing with peripheral devices and to develop additional boards that accord with the applications being developed.
- Can use Renesas Electronics' E1 on-chip emulator (14-pin connector).

6.3 Main Specifications

The main specifications of the V850E2/ML4 CPU board are as follows:

- CPU μPD70F4022 (V850E2/ML4)
- Operating frequency 200 MHz (clock multiplied by 20 by using PLL)
- Interface USB connector × 2 (USB host A type × 1, USB function miniB type × 1)
  UART connector
  CAN connector
  Ethernet connector
- Supported platform Host: PC/AT compatible computer with USB interface
  OS: Windows 7, Vista, XP
- Operating voltage 5.0 V
- Package dimensions W125 × D170 (mm)

Website and Support

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

Inquiries
http://www.renesas.com/contact/

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Mar. 23, 2012	—	First edition issued

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this manual, refer to the relevant sections of the manual. If the descriptions under General Precautions in the Handling of MPU/MCU Products and in the body of the manual differ from each other, the description in the body of the manual takes precedence.

1. Handling of Unused Pins
   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.
   — The input pins of CMOS products are generally in the high-impedance state. In operation with unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.
2. Processing at Power-on
   The state of the product is undefined at the moment when power is supplied.
   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.
3. Prohibition of Access to Reserved Addresses
   Access to reserved addresses is prohibited.
   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.
4. Clock Signals
   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.
   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.
5. Differences between Products
   Before changing from one product to another, i.e. to one with a different part number, confirm that the change will not lead to problems.
   — The characteristics of MPU/MCU in the same group but having different part numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to products of different part numbers, implement a system-evaluation test for each of the products.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
   - "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
   - "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
   - "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.1