V850E2/MN4

USB MSC (Mass Storage Class) Driver

R01AN0011EJ0102
Rev.1.02
Jan 23, 2012

Summary

This application note describes the sample MSC (Mass Storage Class) driver for the USB function controller that is incorporated in the V850E2/MN4 microcontroller.

The application note consists primarily of the following parts:

- Sampler driver specifications
- Environment for developing application programs that make use of the sample driver
- Reference information that is useful for using the sample driver

Target Device

RTE-V850E2/MN4-EB-S incorporating the V850E2/MN4 (μPD70F3512)

Contents

1. Introduction

1.1 Note

The sample program introduced in this application note is provided only for reference purposes. Renesas does not guarantee normal operation of the sample program under any circumstances.
When using the sample program, make extensive evaluations of the driver on a user's set.

1.2 Intended Audiences

This application note is intended for the users who have basic understanding of the capabilities of the V850E2/MN4 microcontroller and who are to develop application systems utilizing that microcontroller.

1.3 Objective

The objective of this application note is to help the users acquire an understanding of the specifications for the sample program for utilizing the USB function controller incorporated in the V850E2/MN4 microcontroller.

1.4 Organization

This application note is divided into the following topics:

- Overview of the USB standards
- Specifications for the sample driver
- Development environment (CubeSuite, Multi (Note1), or IAR Embedded Workbench (Note2))
- Application of the sample driver

(Note 1) Multi is a registered trademark of Green Hills SoftwareTM, Inc.
(Note 2) IAR Embedded Workbench is a registered trademark of IAR Systems AB.

1.5 How to Read this Document

The readers of this document are assumed to have general knowledge about electronics, logic circuits, and microcontrollers.

— If you want to know the hardware capabilities and electrical characteristics of the V850E2/MN4 microcontroller
  →Refer to the separately available V850E2/MN4 Microcontroller User's Manual [Hardware].

— If you want to know the instruction set of the V850E2/MN4 microcontroller
  →Refer to the separately available V850E2M User's Manual [Architecture].

2. Overview

This application note describes the sample MSC (Mass Storage Class) driver for the USB function controller incorporated in the V850E2/MN4 microcontroller. It is composed of the following topics:

- Specifications for the sample driver
- Environment for developing application programs that are to use the sample driver
- Reference information useful for making use of the sample driver

In this section, an overview of the sample driver and the description of the applicable microcontrollers are introduced.

2.1 Overview

2.1.1 Features of the USB Function Controller

The USB function controller of the V850E2/MN4 microcontroller, which is the control target of this sample driver, has the features listed below.

- Conforms to the USB (Universal Serial Bus Specification) 2.0.
- Operates as a full-speed (12 Mbps) device.
- Endpoints are configured as summarized in the table below.

Table 2.1 V850E2/MN4 Microcontroller's Endpoint Configuration

Endpoint Name	FIFO Size (Bytes)	Transfer Type	Remarks
Endpoint0 Read	64	Control transfer (IN)	—
Endpoint0 Write	64	Control transfer (OUT)	—
Endpoint1	64 × 2	Bulk transfer 1 (IN)	2-buffer configuration
Endpoint2	64 × 2	Bulk transfer 1 (OUT)	2-buffer configuration
Endpoint3	64 × 2	Bulk transfer 2 (IN)	2-buffer configuration
Endpoint4	64 × 2	Bulk transfer 2 (OUT)	2-buffer configuration
Endpoint7	64	Interrupt transfer (IN)	—
Endpoint8	64	Interrupt transfer (IN)	—

- Automatically responds to USB standard requests (except part of requests)
- Bus-powered or self-powered mode selectable
- Internal or external clock selectable (Note 2)

  Internal clock: External 9.6 MHz × 20 (internally) ÷ 4 (48 MHz)
  or External 7.2 MHz × 20 (internally) ÷ 3 (48 MHz)
  External clock: Input to the USBCLK pin (fUSB = 48 MHz)

  (Note 2) The internal clock is selected for the sample driver.

2.1.2 Features of the Sample Driver

The MSC (Mass Storage Class) sample driver for the V850E2/MN4 microcontroller has the features listed below. For details about the features and operations of the sample driver, see section 4, Sample Driver Specifications.

- Operates as a self-powered device.
- Recognized as a bulk-only device of the mass storage class when connected to the host.
- Can be formatted for arbitrary file systems by the host.
- Allows file and folder data to be written into internal RAM.
- Allows the file and folder data to be read out of internal RAM.
- Occupies memory areas of the following sizes (excluding that of the vector table):

  ROM: Approx. 9.0 Kbytes

  RAM: Approx. 25.5 Kbytes (Note 3)

  (Note3) 24 Kbytes of the RAM area (approx. 25.5 Kbytes) is used as the data storage area.
  For this reason, the data in the storage area is initialized when device power is turned off or when the Reset SW is pressed.

2.1.3 Sample Driver Configuration

The sample driver is available in three versions, i.e., the CubeSuite version, the Multi version, and the IAR Embedded Workbench version. Use the correct version of the sample driver according to your development environment.
Each version of the sample driver is made up of the files that are described below.

(1) CubeSuite Version

The CubeSuite version of the sample driver comprises files that are summarized below.

Table 2.2 CubeSuite Version Sample Driver File Configuration

Folder	File	Outline
src	main.c	Main routine
	scsi_cmd.c	SCSI command processing
	usbf850.c	USB initialization, endpoint control, bulk transfer, and control transfer
	usbf850_storage.c	MSC-specific processing
	cstart.asm	Bootstrap
include	main.h	main.c function prototype declaration
	scsi.h	SCSI related macro definitions
	usbf850.h	usbf850.c function prototype declarations
	usbf850_desc.h	Descriptor definitions
	usbf850_errno.h	Error code definitions
	usbf850_storage.h	usbf850_storage.c function prototype declarations
	usbf850_types.h	User type declarations
	reg_v850e2mn4.h	USB function register definitions

Remarks: The sample driver package comes also with a set of project-related files for the CubeSuite (Renesas Electronics' integrated development tool suit). For further information, see section 5.2.1, Setting up the Host Environment.

(2) Multi Version

The Multi version of the sample driver comprises files that are summarized below.

Table 2.3 Multi Version Sample Driver File Configuration

Folder	File	Outline
src	main.c	Main routine
	scsi_cmd.c	SCSI command processing
	usbf850.c	USB initialization, endpoint control, bulk transfer, and control transfer
	usbf850_storage.c	MSC-specific processing
	initial.s	Bootstrap
	vector.s	Interrupt vector table declarations
include	main.h	main.c function prototype declarations
	scsi.h	SCSI-related macro definitions
	usbf850.h	usbf850.c function prototype declarations
	usbf850_desc.h	Descriptor definitions
	usbf850_errno.h	Error code definitions
	usbf850_storage.h	usbf850_storage.c function prototype declarations
	usbf850_types.h	User type declarations
	reg_v850e2mn4.h	USB function register definitions
	df3512_800.h	V850E2/MN4 register definitions

Remarks: The sample driver package comes also with a set of project-related files for the Multi (Green Hills SoftwareTM, Inc. integrated development tool suit). For further information, see section 5.4.1, Setting up the Host Environment.

(3) IAR Embedded Workbench Version

The IAR Embedded Workbench version of the sample driver comprises files that are summarized below.

Table 2.4 IAR Embedded Workbench Version Sample Driver File Configuration

Folder	File	Outline
src	main.c	Main routine
	scsi_cmd.c	SCSI command processing
	usbf850.c	USB initialization, endpoint control, bulk transfer, and control transfer
	usbf850_storage.c	MSC-specific processing
include	main.h	main.c function prototype declarations
	scsi.h	SCSI-related macro definitions
	usbf850.h	usbf850.c function prototype declarations
	usbf850_desc.h	Descriptor definitions
	usbf850_errno.h	Error code definitions
	usbf850_storage.h	usbf850_storage.c function prototype declarations
	usbf850_types.h	User type declarations
	reg_v850e2mn4.h	USB function register definitions

Remarks: The sample driver package comes also with a set of project-related files for the IAR Embedded Workbench. For further information, see section 5.6.1, Setting up the Host Environment.

2.2 V850E2/MN4 Microcontroller

For details on the V850E2/MN4 microcontroller that is to be controlled by the sample driver, refer to the user's manual [hardware] of the individual products.

2.2.1 Applicable Products

The sample driver is applicable to the products that are listed below.

Table 2.5 List of Supported V850E2/MN4 Microcontroller Products

Model Name	Part Number	Internal Memory		Internal USB Function	Interrupt		UM
		Flash Memory	RAM		Internal Note4	External Note 4	
V850E2/MN4	μPD70F3510	1 Mbytes	64 Kbytes + 64 Kbytes	Host and Function	180	29	V850E2/MN4 User's Manual [Hardware] (R01UH0011EJ)
	μPD70F3512	1 Mbytes	64 Kbytes + 64 Kbytes	Host and Function	190	29	
	μPD70F3514	1 Mbytes	64 Kbytes × 2 + 64 Kbytes	Host and Function	196	29	
	μPD70F3515	2 Mbytes	64 Kbytes × 2 + 64 Kbytes	Host and Function	196	29	

(Note 4) Includes nonmaskable interrupts

2.2.2 Features

The major features of the V850E2/MN4 are listed below.

- Internal memory

  RAM: Single core, 64 Kbytes; Dual core, 64 Kbytes × 2
  Flash memory: 1 Mbyte

- Flash cache memory

  Single core: 16 Kbytes (4-way associative)
  Dual core: 16 Kbytes (4-way associative) × 2

- External bus interface

  Equipped with 2 systems of memory controllers.
  Primary memory controller (SRAM/SDRAM connectable)
  Secondary memory controller (SRAM/SDRAM connectable)

- Serial interfaces

  Asynchronous serial interface UART: 6 channels
  Clock synchronous serial interface CSI: 6 channels
  Asynchronous serial interface UART (FIFO): 4 channels
  Clock synchronous serial interface CSI (FIFO): 4 channels
  I2C: 6 channels
  CAN: 2 channels (μPD70F3512, μPD70F3514, and μPD70F3515)
  USB function controller: 1 channel
  USB host controller: 1 channel
  Ethernet controller : 1 channel (μPD70F3512, μPD70F3514, and μPD70F3515)

- DMA controllers

  DMA controller: 16 channels
  DTS: 128 channels maximum

3. USB Overview

This section provides a brief description of the USB standard to which the sample driver conforms.

USB (Universal Serial Bus) is a standard for interfacing various peripheral devices with a host computer with a common connector. It provides an interface that is more flexible and easier to use than conventional interfaces. For example, it supports the hot-plug feature and allows a maximum of 127 devices to be connected together through the use of additional connection nodes called hubs. The ratio of the PCs having the USB interface installed to the entire PCs that are presently available is reaching almost 100%. It can safely be said that the USB interface has become the standard interface for connecting the PC and peripheral devices.

The USB standard is formulated and managed by the organization called the USB Implementers Forum (USB-IF). For details on the USB standard, visit the USB-IF's official web site (www.usb.org).

3.1 Transfer Modes

The USB standard defines four types of transfer modes (control, bulk, interrupt, and isochronous). The major features of the transfer modes are summarized in table 3.1.

Table 3.1 USB Transfer Modes

Item \ Transfer Mode		Control Transfer	Bulk Transfer	Interrupt Transfer	Isochronous Transfer
Feature		Transfer mode that is used to exchange information necessary for controlling peripheral devices.	Transfer mode that is used to handle a large amount of data nonperiodically.	Transfer mode that is used to transfer data periodically and has a narrow band width.	Transfer mode used in applications that are required of high realtime performance.
Allowable packet size	High speed (480 Mbps)	64 bytes	512 bytes	1 to 1024 bytes	1 to 1024 bytes
	Full speed (12 Mbps)	8, 16, 32, or 64 bytes	8, 16, 32, or 64 bytes	1 to 64 bytes	1 to 1023 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 item of information used by the host device to identify a specific communication counterpart. An endpoint is specified by a number from 0 to 15 and the direction (IN or OUT). An endpoint need be provided for each data communication channel that is to be used by a peripheral device and cannot be shared by two or more communication channels (Note 5). For example, a device that has the capabilities to write and read to and from an SD card and to print out data need be provided with an endpoint for writing to an SD card, an endpoint for reading from an SD card, and an endpoint for sending data to a printer. Endpoint 0 is used for control transfer which must always be performed by every device.

In data communication, the host device specifies the destination within the USB device using the USB device address which identifies the device and an endpoint (number and direction).

A buffer memory is provided within every peripheral device as a physical circuit for endpoints. It also serves as a FIFO that absorbs the difference in communication speed between the USB and the communication counterpart (e.g., memory).

(Note 5) There is a method of switching channels exclusively using a mechanism called the alternate setting.

3.3 Classes

Peripheral devices (function devices) connected via the USB have various classes defined according to their functionality. Typical classes include the mass storage class (MSC), communications device class (CDC), printer class, and human interface device class (HID). For each class, standard specifications are defined in the form of protocols. A common host driver can be used provided that it conforms to those standard specifications.

3.3.1 Mass Storage Class (MSC)

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

There are two types of communication protocols for the MSC, i.e., the bulk-only transport protocol and CBI (control/bulk/interrupt) transport protocol. With the bulk-only transport protocol, data is transferred only in bulk transfer mode. With the CBI transport protocol, control and interrupt transfer modes are used in addition to the bulk transfer mode. The CBI transport protocol is available only for full-speed floppy disk drives.

The sample driver uses the bulk-only transport protocol for the mass storage class (MSC). For the specifications for the USB mass storage class (MSC), refer to the MSC specification entitled “Universal Serial Bus Mass Storage Class Bulk-Only Transport Revision 1.0.”

(1) Data transfer

With the bulk-only transport protocol, all transfers (commands, status, and data) are carried out in bulk transfer mode.

The host sends commands to devices using bulk OUT transfers.

When a command that involves data transfers is sent, data input/output operations are performed using bulk IN/bulk OUT transfers.

The device sends the status (command execution result) to the host using a bulk IN transfer.

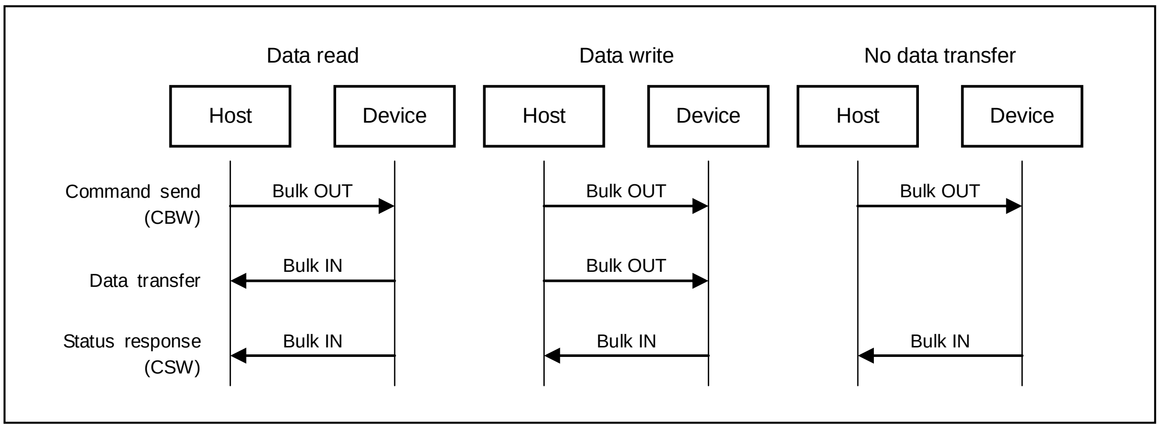


Figure 3.1 Data Transfer Flow

(2) CBW format

The structure of a packet for sending a command is defined as a Command Block Wrapper (CBW).

Table 3.2 CBW Format

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

dCBWSignature:	Signature. Fixed at 0x43425355 (little endian).
dCBWTag:	A tag containing an arbitrary number defined by the host. Used to associate the status with the corresponding command.
dCBWDataTransferLength:	Length of data to be transferred in the data phase. 0 if there is no data to transfer.
bmCBWFlags:	Direction of transfer (bit 7). 0 = Bulk OUT, 1 = Bulk IN. Bits 0 to 6 must always be set to 0.
bCBWLUN:	Drive number of one of the two or more drives connected to a single USB device
bCBWCBLength:	Length of the command packet
CBWCB:	Command packet data

(3) CSW format

The structure of the status packet is defined as a Command Status Wrapper (CSW).

Table 3.3 CSW Format

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

dCSWSignature:	Signature. Fixed at 0x53425355 (little endian).
dCSWTag:	The host confirms a phase match when this tag matches with the dCBWTag that is transferred with the command.
dCSWDataResidue:	Remaining data. This field is loaded with the amount of remaining data when the amount of data returned by the device is found smaller than the amount of data requested by the host due to, for example, an error occurring during data transfer. A nonzero value in this field indicates that the length of response data from the device is shorter than the expected length of data even if the status (bCSWStatus) indicates a success.
dCSWStatus:	Status indicating the result of CBW processing

Table 3.4 CBW Processing Status Parameter Values

dCSWStatus	Description
0x00	Success
0x01	Failure
0x02	Phase error
0x03 to 0xFF	Reserved

3.3.2 Subclasses

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

(1) Subclass types

Table 3.5 shows a list of subclass codes that are specified for the USB mass storage class.

Table 3.5 USB Mass Storage Subclass Codes

Subclass Code	Specification
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
0x09-0xFE	Reserved
0xFF	Specific to device vender

(2) SCSI commands

The SCSI transfer command set (0x06) must be specified as a subclass when USB memory or a USB card reader is to be connected. SCSI (Small Computer System Interface) is an interface specification for connecting a computer with peripheral devices in a bus topology configuration.

Data transfer and function configuration are carried out by specifying SCSI commands in the CBWCB (command packet data) of the CBW. See section 4.1.4, SCSI Command Handling, for the SCSI commands supported by the sample driver.

3.4 Requests

According to the USB specification, communication is initiated by the host device issuing a command called a request to all function devices. The request contains data such as the direction and type of processing and the address of the target function device. Each function device decodes the request, determines whether the request is directed to itself, and responds to the request only when it is directed to the device.

3.4.1 Types

There are three types of requests, namely, the standard requests, class requests, and vendor requests.
See section 4.1.2, Requests Handling, for the requests that the sample driver support.

(1) Standard requests

Standard requests are used in common by all USB compatible devices. A request is a standard request when both bits 6 and 5 of the bmRequestType field of the request are set to 0. Refer to the USB specification (Universal Serial Bus Specification Rev. 2.0) for the processing that is to be performed for the standard requests.

Table 3.6 List of Standard Requests

Request Name	Target Descriptor	Outline
GET_STATUS	Device	Read power (self or bus) and remote wakeup settings.
	Endpoint	Read Halt status.
CLEAR_FEATURE	Device	Clear remote wakeup.
	Endpoint	Cancel Halt (DATA PID = 0).
SET_FEATURE	Device	Set up remote wakeup or test mode.
	Endpoint	Set Halt
GET_DESCRIPTOR	Device, configuration, string	Read target descriptor
SET_DESCRIPTOR	Device, configuration, string	Set target descriptor (optional)
GET_CONFIGURATION	Device	Read current configuration value.
SET_CONFIGURATION	Device	Set configuration value.
GET_INTERFACE	Interface	Read alternate value out of the current settings of the target interface.
SET_INTERFACE	Interface	Set alternate value of the target interface.
SET_ADDRESS	Device	Set USB address.
SYNCH_FRAME	Endpoint	Read frame-synchronous data.

(2) Class requests

The class requests are unique to the class. A request is a class request when bit 6 of the bmRequestType field is set to 0 and bit 5 to 1.

The bulk-only transport protocol of the mass storage class (MSC) needs to handle the following requests:

- GET_MAX_LUN (bRequest = 0xFE)
  Request used to get the number of logical units (logical unit number) of the mass storage devices.
- MASS_STORAGE_RESET (bRequest = 0xFF)
  Request used to reset the interface associated with the mass storage devices.

(3) Vendor requests

The vendor requests are defined uniquely by the individual vendors. A vendor who is to use a vendor request needs to provide a host driver that handles that request. A request is a vendor request when bit 6 of the bmRequestType field is set to 1 and bit 5 to 0.

3.4.2 Format

A USB request is 8 bytes long and consists of the fields that are listed in the table below.

Table 3.7 USB Request Format

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

3.5 Descriptors

In the USB specification, a set of information that is specific to a function device and is encoded in a predetermined format is called a descriptor. Each function device sends its descriptor in response to a request from the host device.

3.5.1 Types

The following five types of descriptors are defined:

- Device descriptor
  This descriptor is present in all types of devices. It contains basic information such as the version of the supported USB specification, device class, protocol, maximum packet length available for transfer to Endpoint0, vendor ID, and product ID.
  The descriptor must be sent in response to a GET_DESCRIPTOR_Device request.

- Configuration descriptor
  Every device has one or more configuration descriptors. It contains such information as device attributes (power supplying method) and power consumption. The descriptor must be sent in response to a GET_DESCRIPTOR_Configuration request.

- Interface descriptor
  This descriptor is necessary for each interface. It contains an interface ID, interface class, and the number of endpoints that are supported. The descriptor must be sent in response to a GET_DESCRIPTOR_Configuration request.

- Endpoint descriptor
  This descriptor is necessary for each endpoint that is specified in the interface descriptor. It defines the transfer type (direction of transfer), maximum packet length available for transfer to the endpoint, and transfer interval. Endpoint0, however, does not have this descriptor.
  The descriptor must be sent in response to a GET_DESCRIPTOR_Configuration request.

- String descriptor
  This descriptor contains an arbitrary string. The descriptor must be sent in response to a GET_DESCRIPTOR_String request.

3.5.2 Formats

The size and field structure of descriptors varies depending on the descriptor type as summarized in the tables below. The data in each field is arranged in little endian format.

Table 3.8 Device Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Size of the descriptor
bDescriptorType	1	Type of the descriptor
bcdUSB	2	Release number of the USB specification
bDeviceClass	1	Class code
bDeviceSubClass	1	Subclass code
bDeviceProtocol	1	Protocol code
bMaxPacketSize0	1	Maximum packet size of Endpoint0
idVendor	2	Vendor ID
idProduct	2	Product ID
bcdDevice	2	Device release number
iManufacturer	1	Index of the string descriptor describing the manufacturer
iProduct	1	Index of the string descriptor describing the product
iSerialNumber	1	Index of the string descriptor describing the device’s serial number
bNumConfigurations	1	Number of configurations

Remarks	Vendor ID:	Identification number that the vendor who is to develop a USB device acquires from USB-IF
	Product ID:	Identification number that the vendor assigns to each of its products after acquiring a vendor ID.

Table 3.9 Configuration Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Size of the descriptor
bDescriptorType	1	Type of the descriptor
wTotalLength	2	Total number of bytes of the configuration, interface, and endpoint descriptors
bNumInterfaces	1	Number of interfaces supported by this configuration
bConfigurationValue	1	Identification number of this configuration
iConfiguration	1	Index of the string descriptor describing this configuration
bmAttributes	1	Characteristics of this configuration
bMaxPower	1	Maximum consumption current of this configuration (in 2 µA units)

Table 3.10 Interface Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Size of the descriptor
bDescriptorType	1	Type of the descriptor
bInterfaceNumber	1	Identification number of this interface
bAlternateSetting	1	Presence or absence of alternate setting for this interface
bNumEndpoints	1	Number of endpoints used by this interface
bInterfaceClass	1	Class code
bInterfaceSubClass	1	Subclass code
bInterfaceProtocol	1	Protocol code
iInterface	1	Index of the string descriptor describing this interface

Table 3.11 Endpoint Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Size of the descriptor
bDescriptorType	1	Type of the descriptor
bEndpointAddress	1	Transfer direction of this endpoint Address of this endpoint
bmAttributes	1	Transfer type of this endpoint
wMaxPacketSize	2	Maximum packet size available for transfer at this endpoint
bInterval	1	Interval for polling this endpoint

Table 3.12 String Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Size of the descriptor
bDescriptorType	1	Type of the descriptor
bString	Arbitrary	Arbitrary data string

4. Sample Driver Specifications

This section contains a detailed description of the features and operations of the USB mass storage class (MSC) sample driver for the V850E2/MN4 microcontroller. It also describes the specifications for the functions of the sample driver.

4.1 Overview

4.1.1 Features

The sample driver has the following processing implemented:

(1) Main routine

The main routine performs initialization and waits for interrupts. It performs suspend/resume processing when a suspend/resume interrupt occurs. For details, see section 4.2.7, Suspend/Resume Processing.

(2) Initialization

The initialization routine manipulates and sets up various registers to make the USB function controller ready for use. The register settings are broadly divided into those for the V850E2/MN4's CPU registers and those for the registers of the USB function controller. For details, see section 4.2.1, CPU Initialization Processing, and section 4.2.2, USB Function Controller Initialization Processing.

(3) Interrupt processing

The INTUSFA0I1 interrupt handler monitors the state of the endpoint for control transfer (Endpoint0) and the endpoint for bulk OUT transfer (reception) (Endpoint2) and performs appropriate processing according to the request and data that are received. The INTUSFA0I2 interrupt handler performs the processing that is required when a resume interrupt occurs. For details, see section 4.2.3, USBF Interrupt Processing (INTUSFA0I1), and section 4.2.4, USBF Resume Interrupt Processing (INTUSFA0I2).

(4) SCSI command processing

This routine analyzes the CBW data that is received and determines whether it is a SCSI command. If a SCSI command is received, the routine performs the required processing according to the received SCSI command. For details, see section 4.1.4, SCSI Command Handling.

4.1.2 Request Handling

Table 4.1 lists the USB requests that are defined for the hardware (V850E2/MN4) and firmware (sample driver).

Table 4.1 USB Request Processing

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

Remarks HW: Hardware (V850E2/MN4)
FW: Firmware (sample driver)
0xXX: Undefined

(1) Standard requests

The sample driver performs the following response processing for requests that the V850E2/MN4 does not automatically respond:

(a) GET_DESCRIPTOR_string

This request is used by the host to get the string descriptor of a function device.

Upon receipt of this request, the sample driver performs the processing of sending the requested string descriptor (control read transfer).

(b) SET_DESCRIPTOR

This request is used by the host to set the descriptor of a function device.

Upon receipt of this request, the sample driver returns a STALL response.

(2) Class requests

The sample driver performs the following response processing for class requests of the bulk-only transport protocol for the USB mass storage class (MSC):

(a) GET_MAX_LUN

This request is used to get the number of logical units (logical unit number) of mass storage device.

The host specifies the number of the logical unit in the bCBWLUN field of the CBW when sending it.

The sample driver returns 0 (number of logical units = 1) when it receives a GET_MAX_LUN request.

Table 4.2 GET_MAX_LUN Request Format

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

(b) MASS_STORAGE_RESET

This request is used to reset the interface that is associated with a mass storage device.

When the sample driver receives a MASS_STORAGE_RESET request, it resets the interface of the USB function controller that the sample driver is using.

Table 4.3 MASS_STORAGE_RESET Request Format

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

(3) Undefined requests

The sample driver returns a STALL response when it receives an undefined request.

4.1.3 Descriptor Settings

The descriptor settings that the sample driver makes are summarized in the tables below. The settings of the individual descriptors are defined in the header file named "usbf850_desc.h."

(1) Device descriptor

This descriptor is sent in response to a GET_DESCRIPTOR_device request.

Since the hardware automatically responds to the GET_DESCRIPTOR_device request, the settings are stored in the USFA0DDn registers (n = 0 to 17) when the USB function controller is initialized.

Table 4.4 Device Descriptor Settings

Field	Size (Bytes)	Value	Description
bLength	1	0x12	Size of the descriptor: 18 bytes
bDescriptorType	1	0x01	Type of the descriptor: Device
bcdUSB	2	0x0200	USB specification release number: USB 2.0
bDeviceClass	1	0x00	Class code: None
bDeviceSubClass	1	0x00	Subclass code: None
bDeviceProtocol	1	0x00	Protocol code: No unique protocol used
bMaxPacketSize0	1	0x40	Maximum packet size of Endpoint0: 64
idVendor	2	0x045B	Vendor ID: Renesas Electronics
idProduct	2	0x0200	Product ID: V850E2/MN4
bcdDevice	2	0x0001	Device release number: First version
iManufacturer	1	0x01	Index of string descriptor describing the manufacturer: 1
iProduct	1	0x00	Index of string descriptor describing the product: 0
iSerialNumber	1	0x00	Index of string descriptor describing the serial number of the device: 0
bNumConfigurations	1	0x01	Number of configurations: 1

(2) Configuration descriptor

This descriptor is sent in response to a GET_DESCRIPTOR_configuration request.

Since the hardware automatically responds to the GET_DESCRIPTOR_configuration request, the settings are stored in the USFA0CIEn registers (n = 0 to 255) when the USB function controller is initialized.

Table 4.5 Configuration Descriptor Settings

Field	Size (Bytes)	Value	Description
bLength	1	0x09	Size of the descriptor: 9 bytes
bDescriptorType	1	0x02	Type of the descriptor: Configuration
wTotalLength	2	0x0020	Total number of bytes of the configuration, interface, and endpoint descriptors: 32 bytes
bNumInterfaces	1	0x01	Number of interfaces supported by this configuration: 1
bConfigurationValue	1	0x01	Identification number of this configuration: 1
iConfiguration	1	0x00	Index of the string descriptor describing this configuration: 0
bmAttributes	1	0x80	Characteristics of this configuration: Bus powered, no remote wakeup
bMaxPower	1	0x1B	Maximum consumption current of this configuration: 54 mA

(3) Interface descriptor

This descriptor is sent in response to a GET_DESCRIPTOR_configuration request.

Since the hardware automatically responds to the GET_DESCRIPTOR_configuration request, the settings are stored in the USFA0CIEn registers (n = 0 to 255) when the USB function controller is initialized.

Table 4.6 Interface Descriptor Settings

Field	Size (Bytes)	Value	Description
bLength	1	0x09	Size of the descriptor: 9 bytes
bDescriptorType	1	0x04	Type of the descriptor: Interface
bInterfaceNumber	1	0x00	Identification number of this interface: 0
bAlternateSetting	1	0x00	Presence or absence of alternate setting for this interface: Absence
bNumEndpoints	1	0x02	Number of endpoints used by 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 of the string descriptor describing this interface: 0

(4) Endpoint descriptor

This descriptor is sent in response to a GET_DESCRIPTOR_configuration request.

Since the hardware automatically responds to the GET_DESCRIPTOR_configuration request, the settings are stored in the USFA0CIEn registers (n = 0 to 255) when the USB function controller is initialized.

Since the sample driver uses two endpoints, two endpoint descriptors are set up.

Table 4.7 Endpoint1 (Bulk IN) Endpoint Descriptor Settings

Field	Size (Bytes)	Value	Description
bLength	1	0x07	Size of the descriptor: 7 bytes
bDescriptorType	1	0x05	Type of the descriptor: 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 available for transfer to this endpoint: 64 bytes
bInterval	1	0x00	Interval for polling this endpoint: 0 ms

Table 4.8 Endpoint2 (Bulk OUT) Endpoint Descriptor Settings

Field	Size (Bytes)	Value	Description
bLength	1	0x07	Size of the descriptor: 7 bytes
bDescriptorType	1	0x05	Type of the descriptor: 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 available for transfer to this endpoint: 64 bytes
bInterval	1	0x00	Interval for polling this endpoint: 0 ms

(5) String descriptor

This descriptor is sent in response to a GET_DESCRIPTOR_string request.

When the sample driver receives a GET_DESCRIPTOR_string request, it fetches the string descriptor settings from the header file named "usbf850_desc.h" and stores them in the USFA0E0W registers of the USB function controller.

Table 4.9 String Descriptor Settings

(a) String 0

Field	Size (Bytes)	Value	Description
bLength	1	0x04	Size of the descriptor: 4 bytes
bDescriptorType	1	0x03	Type of the descriptor: String
bString	2	0x09, 0x04	Language code: English (U.S.)

(b) String 1

Field	Size (Bytes)	Value	Description
bLength Note 6	1	0x16	Size of the descriptor: 24 bytes
bDescriptorType	1	0x03	Type of the descriptor: String
bString Note 7	22	–	Serial number: V850E2/MN4: 020008065010

(Note 6) The value varies with the size of the bString field.

(Note 7) The size and value are not fixed because this area can be set up arbitrarily by the vendor.

4.1.4 SCSI Command Handling

The sample driver specifies the SCSI transfer command set (0x06) as a subclass.

The SCSI commands that are supported by the sample driver are listed in table 4.10. The sample driver returns a STALL response when it receives a command that is not listed in table 4.10.

Table 4.10 SCSI Commands Supported by the Sample Driver

Command Name	Code	Direction of Bulk Transfer	Outline
TEST_UNIT_READY	0x00	NO DATA	Checks the type and configuration of the device.
REQUEST_SENSE	0x03	IN	Gets the sense data.
READ6	0x08	IN	Reads data.
WRITE6	0x0A	OUT	Writes data.
SEEK	0x0B	NO DATA	Specifies a seek to given data position.
INQUIRY	0x12	IN	Get configuration information/attributes.
MODE_SELECT	0x15	OUT	Set parameters.
MODE_SENSE6	0x1A	IN	Reads parameter values.
START_STOP_UNIT	0x1B	NO DATA	Loads/unloads media or starts/stops the motor.
PREVENT	0x1E	NO DATA	Enables/disables media unloading.
READ_FORMAT_CAPACITIES	0x23	IN	Gets storage capacity information.
READ_CAPACITY	0x25	IN	Gets capacity information.
READ10	0x28	IN	Reads data.
WRITE10	0x2A	OUT	Writes data.
WRITE_VERIFY	0x2E	OUT	Writes data and verifies it.
VERIFY	0x2F	NO DATA	Executes verify processing.
SYNCHRONIZE_CACHE	0x35	NO DATA	Writes data left in cache.
WRITE_BUFF	0x3B	OUT	Writes data to buffer memory.
MODE_SELECT10	0x55	OUT	Sets parameters.
MODE_SENSE10	0x5A	IN	Gets parameter values.

(1) TEST_UNIT_READY command (0x00)

This command notifies the initiator (host device) of the state of the logical unit. The sample driver initializes the sense data and terminates normally.

Table 4.11 TEST_UNIT_READY Command Format

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

(2) REQUEST_SENSE command (0x03)

This command sends the sense data to the host. The sample driver sends the sense data listed in table 4.14 to the host.

Table 4.12 REQUEST_SENSE Command Format

Byte \ 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 4.13 REQUEST_SENSE Data Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	VALID	Response code						
1	Reserved							
2	Filemark	EOM	ILI	Reserved	Sense key			
3–6	Information							
7	Additional sense data length (n – 7 bytes)							
8–11	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-n	Additional sense data (data length variable)							

Table 4.14 Sense Data

Sense Key	ASC	ASCQ	Outline
0x00	0x00	0x00	NO SENSE
0x05	0x00	0x00	ILLEGAL REQUEST
0x05	0x20	0x00	INVALID COMMAND OPERATION CODE
0x05	0x24	0x00	INVALID FIELD IN COMMAND PACKET

(3) READ6 command (0x08)

This command transfers data from the specified range of logical data blocks to the host.

Table 4.15 READ6 Command Format

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

(4) WRITE6 command (0x0A)

This command writes the received data into the specified block on the storage device.

Table 4.16 WRITE6 Command Format

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

(5) SEEK command (0x0B)

This command performs a seek to the specified position on the recording medium. The sample driver initializes the sense data and terminates normally.

Table 4.17 SEEK Command Format

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

(6) INQUIRY command (0x12)

This command notifies the host of the configuration information and attributes of the device. The sample driver sends the INQUIRY_TABLE values to the host.

Table 4.18 SEEK Command Format

Byte \ 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 4.19 INQUIRY Data Format

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

```
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 sets the data format of the device and other parameters. The sample driver loads MODE_SELECT_TABLE with values.

Table 4.20 MODE_SELECT Command Format

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

Table 4.21 MODE_SELECT Data Format

Byte \ 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							
5-7	Number of blocks							
8	Reserved							
9-11	Block length							
12	PS	1	Page code					
13	Page length (n – 13 bytes)							
14-n	Mode parameter (data length variable)							

```
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 sends the values of the mode select parameters and other attributes of the device to the host. The sample driver sends the MODE_SENSE_TABLE values to the host.

Table 4.22 MODE_SENSE6 Command Format

Byte \ 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 4.23 MODE_SENSE6 Data Format

Byte \ 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							
5-7	Number of blocks							
8	Reserved							
9-11	Block length							
12	PS	Reserved	Page code					
13	Page length (n – 13 bytes)							
14-n	Mode parameter (data length variable)							

```
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 accesses to the device. The sample driver initializes the sense data and terminates normally.

Table 4.24 START_STOP_UNIT Command Format

Byte \ 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 medium unloading. The sample driver does nothing and terminates normally.

Table 4.25 PREVENT Command Format

Byte \ 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 notifies the host of the capacity (number of blocks and block length) of the device. The sample driver sends the READ_FORMAT_CAPACITY_TABLE values to the host.

Table 4.26 READ_FORMAT_CAPACITIES Command Format

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

Table 4.27 READ_FORMAT_CAPACITIES Data Format

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

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

Figure 4.4 READ_FORMAT_CAPACITY_TABLE

(12) READ_CAPACITY command (0x25)

This command notifies the host of the size of the data on the device. The sample driver sends the READ_CAPACITY_TABLE values to the host.

Table 4.28 READ_CAPACITY Command Format

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

Table 4.29 READ_CAPACITY Data Format

Byte \ Bit	7	6	5	4	3	2	1	0
0-3	Logical block address (LBA)							
4-7	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(Byte)          */
};
```

Figure 4.5 READ_CAPACITY_TABLE

(13) READ10 command (0x28)

This command transfers data from the specified range of logical data blocks to the host.

Table 4.30 READ10 Command Format

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

(14) WRITE10 command (0x2A)

This command writes the received data into the specified block on the device.

Table 4.31 WRITE10 Command Format

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

(15) WRITE_VERIFY command (0x2E)

This command writes the received data into the specified block on the device. After the data is written, its validity is verified. The sample driver carries out only the write operation.

Table 4.32 WRITE_VERIFY Command Format

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

(16) VERIFY command (0x2F)

This command checks the validity of the data on the device. The sample driver does nothing and terminates processing.

Table 4.33 VERIFY Command Format

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

(17) SYNCHRONIZE_CACHE command (0x35)

This command synchronizes the data in the specified range of blocks in cache memory with that on the medium. The sample driver initializes the sense data and terminates normally.

Table 4.34 SYNCHRONIZE_CACHE Command Format

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

(18) WRITE_BUFF command (0x3B)

This command writes data into memory (data buffer). The sample driver reads and discards data and terminates normally.

Table 4.35 WRITE_BUFF Command Format

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

(19) MODE_SENSE10 command (0x5A)

This command notifies the host of the values of the mode select parameters and attributes of the device. The sample driver sends the MODE_SENSE10_TABLE values to the host.

Table 4.36 MODE_SENSE10 Command Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	Operation code (0x5A)							
1	Reserved			LLBAA	DBD	Reserved		
2	PC		Page code					
3-6	Reserved							
7-8	Additional data length							
9	Reserved						Flag	Link

Table 4.37 MODE_SENSE10 Data Format

Byte \ 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							
5-7	Number of blocks (0x0000C0)							
8	Reserved							
9-11	Block length (0x000200)							
12	PS	Reserved	Page code					
13	Page length (n – 13 bytes)							
14-n	Mode parameter (data length variable)							

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

Figure 4.6 MODE_SENSE10_TABLE

(20) MODE_SELECT10 command (0x55)

This command sets the data format of the device and other parameters. The sample driver loads MODE_SELECT10_TABLE with values.

Table 4.38 MODE_SELECT10 Command Format

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

Table 4.39 MODE_SELECT10 Data Format

Byte \ 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							
5-7	Number of blocks							
8	Reserved							
9-11	Block length							
12	PS	1	Page code					
13	Page length (n – 13 bytes)							
14-n	Mode parameter (data length variable)							

```
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 Operations

When the sample driver is started, it performs the sequence of processes that are illustrated in the figure below. This section describes the individual processes.

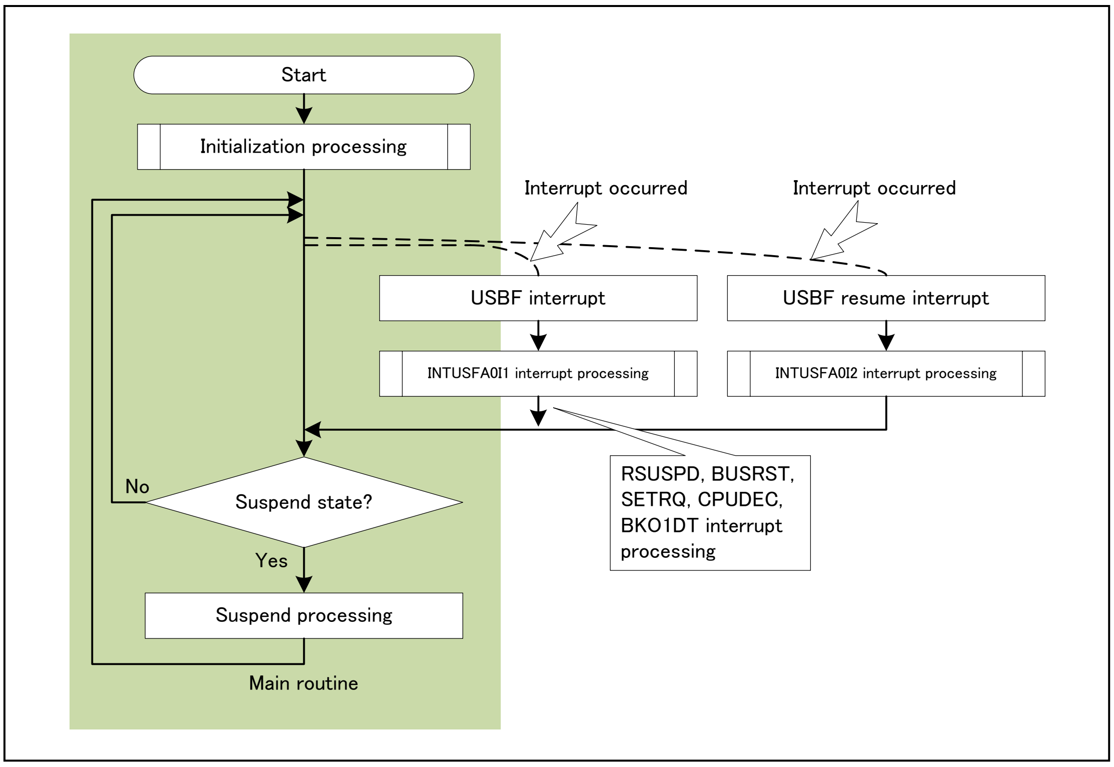

Start
Initialization processing
Interrupt occurred
Interrupt occurred
USBF interrupt
USBF resume interrupt
INTUSFA0I1 interrupt processing
INTUSFA0I2 interrupt processing
RSUSPD, BUSRST, SETRQ, CPUDEC, BKO1DT interrupt processing
No
Suspend state?
Yes
Suspend processing
Main routine

Figure 4.8 Sample Driver Processing Flow

4.2.1 CPU Initialization Processing

The CPU initialization processing routine sets up the parameters that are necessary for using the USB function controller.

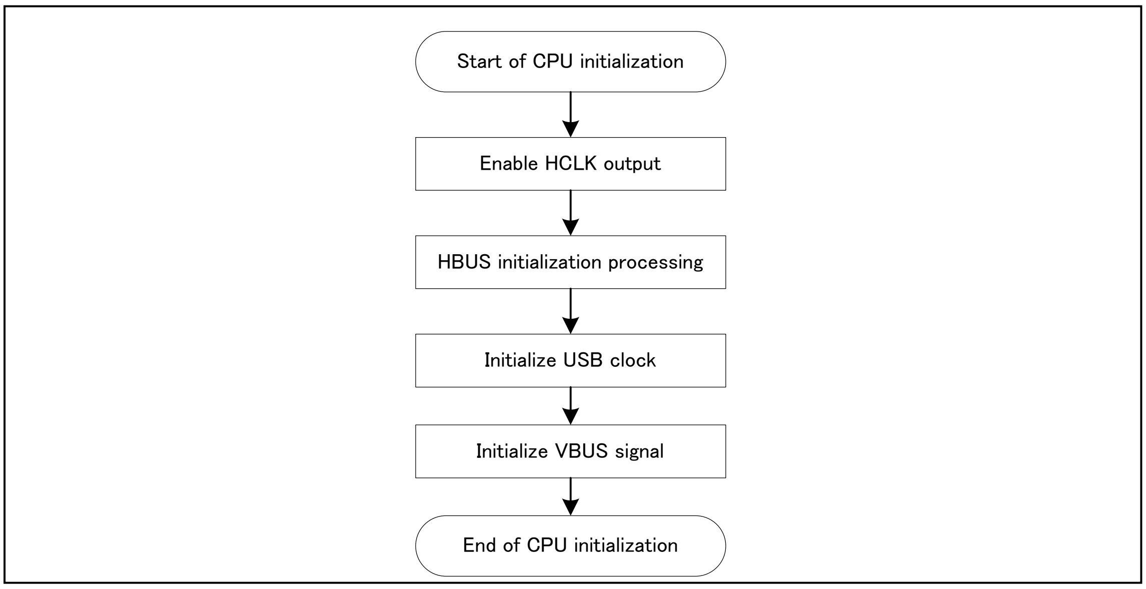

Start of CPU initialization
Enable HCLK output
HBUS initialization processing
Initialize USB clock
Initialize VBUS signal
End of CPU initialization

Figure 4.9 CPU Initialization Processing Flow

(1) Enabling HCLK output

This process makes settings to enable the HCLK output so that the USBF connected to the H bus becomes enabled. Since the SFRCTL2 register used for this setup is a specific write register, a specific write sequence is followed for the setup.

(2) H bus initialization

This process initializes the H-bus. The routine initializes the H bus according to the specified directions. See the V850E2/MN4 Microcontroller User’s Manual [Hardware].

(3) Initializing USB clock

This process sets up the multiplexed pin P13 to which UCLK is connected. This sample driver uses UCLK as the USB clock input to the USB.

(4) Initializing VBUS signal

This process initializes the VBUS signal.

4.2.2 USB Function Controller Initialization Processing

The USB function controller initialization processing routine sets up the parameters necessary for starting the use of the USB function controller.

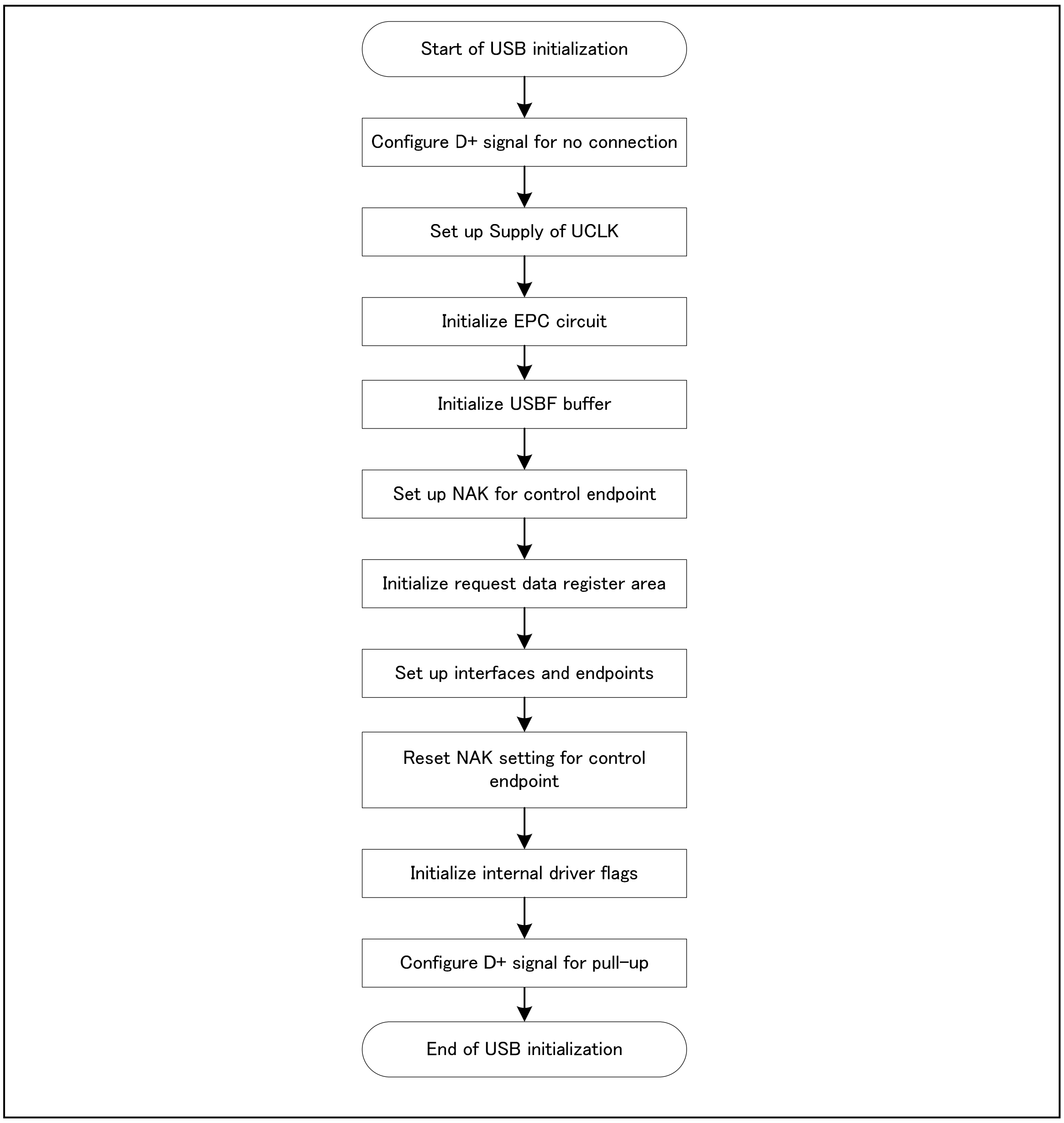

Start of USB initialization
Configure D+ signal for no connection
Set up Supply of UCLK
Initialize EPC circuit
Initialize USBF buffer
Set up NAK for control endpoint
Initialize request data register area
Set up interfaces and endpoints
Reset NAK setting for control endpoint
Initialize internal driver flags
Configure D+ signal for pull-up
End of USB initialization

Figure 4.10 USB Function Controller Initialization Processing Flow

(1) Configuring the D+ signal as pull down

Loads the CPU's P4.10 with "0." This sets the D+ signal low, disabling the host side to detect any device connection.

(2) Setting up for the supply of UCLK

Loads the SFRCTL3 register with "0x48" to enable the clock to be supplied to the USB function.

(3) Initializing the EPC circuit

Loads the USFA0EPCCTL register with “0x00000000” to cancel the EPC reset signal.

(4) Initializing the USB function buffer

Loads the USFBC register with “0x00000003” to enable the USBF buffer and floating provisions.

(5) Setting up NAK for control endpoint

Sets the EP0NKA bit of the USFA0E0NA register to 1. This setting causes the hardware to respond with NAK against all requests including automatically responded requests.

This bit is used by the software until the registration of data to be used in automatically responded requests is completed, so that the hardware will not return unintended data in response to an automatically responded request.

(6) Initializing the request data register area

Loads relevant registers with descriptor data that is to be used to automatically respond to GET_DESCRIPTOR requests.

The following registers are accessed during this processing:

(a) The USFA0DSTL register is loaded with “0x01.” This setting disables the remote wakeup function and the USB function controller operates as a self-powered device.
(b) The USFA0EnSL registers (n = 0 to 2) are loaded with “0x00.” These settings indicate that the Endpoint n are running normally.
(c) The USFA0DSCL register is loaded with the total length (in bytes) of the data in the necessary descriptors. This setting determines the range of the USFA0CIEn registers (n = 0 to 255) to be used.
(d) The USFA0DDn registers (n = 0 to 7) are loaded with the data for the device descriptor.
(e) The USFA0CIEn registers (n = 0 to 255) are loaded with the data for the configuration, interface, and endpoint descriptors.
(f) The USFA0MODC register is loaded with “0x00.” This setting enables GET_DESCRIPTOR_configuration requests to be automatically responded.

(7) Setting up the interfaces and endpoints

Loads relevant registers with the number of interfaces to support, alternate setting status, and the relationship between the interfaces and endpoints.

The following registers are accessed during this processing:

(a) The USFA0AIFN register is loaded with “0x00.” This setting enables only interface 0.
(b) The USFA0AAS register is loaded with “0x00.” This setting disables the alternate setting.
(c) The USFA0E1IM register is loaded with “0x20.” This setting causes Endpoint1 to be linked to Interface0.
(d) The USFA0E2IM register is loaded with “0x20.” This setting causes Endopoint2 to be linked to Interface0.

(8) Resetting NAK setting for control endpoint

Sets the EP0NKA bit of the USFA0E0NA register to 0. This setting enables the resumption of responses to all requests including automatically responded requests.

(9) Setting up the interrupt mask register

Sets the mask bits associated with the interrupt sources of the USB function controller.
The following registers are accessed during this processing:

(a) The USFA0ICn registers (n = 0 to 4) are loaded with “0x00. This setting causes all interrupt sources to be cleared.
(b) The USFA0FIC0 register is loaded with “0xF7” and the USFA0FIC1 register with “0x0F.” These settings cause all FIFOs available for data transfer to be cleared.
(c) The USFA0IM0 register is loaded with “0x1B.” This setting masks all interrupt sources defined in the USFA0IS0 register, except those for the BUSRST, RSUSPD, and SETRQ interrupts.
(d) The USFA0IM1 register is loaded with “0x7E.” This setting masks all interrupt sources defined in the USFA0IS1 register, except that for the CPUDEC interrupt.
(e) The USFA0IM2 register is loaded with “0xF1.” This setting masks all interrupt sources defined in the USFA0IS2 register.
(f) The USFA0IM3 register is loaded with “0xFE.” This setting masks all interrupt sources defined in the USFA0IS3 register, except that for the BKO1DT interrupt.
(g) The USFA0IM4 register is loaded with “0x20.” This setting masks all interrupt sources defined in the USFA0IS4 register.
(i) The USFA0EPCINTE register is loaded with “0x0003” to enable the interrupts for which the EPC_INT0BEN and EPC_INT1BEN bits are set.
(j) The ICUSFA0I1 is loaded with “0” and the ICUSFA0I2 with “0” to enable INTUSFA0I1 and INTUSFA0I2, respectively.

(10) Initializing the internal driver flags

Initializes the flags (usbf850_busrst_flg, usbf850_rsuspd_flg, and usbf850_rdata_flg) that are to be used within the driver.

(11) Setting up the D+ signal as pull-up

Loads the CPU’s P4 register with “0x0400.” This setting causes a “1” to be output from P4_10, which generates a high-level output from the D+ signal pin, notifying the host that a device has been connected. The sample driver assumes the wiring configuration shown in figure 4.11.

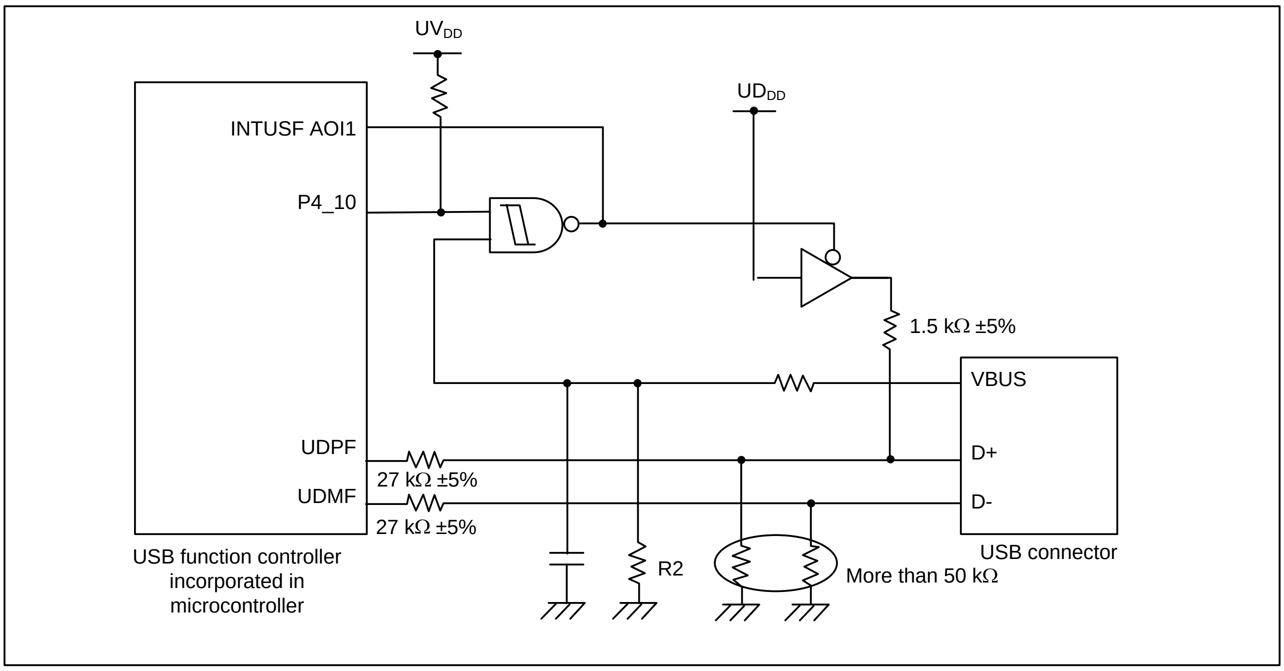


Figure 4.11 USB Function Controller Configuration Example

4.2.3 USBF Interrupt Processing (INTUSFA0I1)

The INTUSFA0I1 interrupt handler monitors the state of the endpoint (Endpoint0) for control transfer and the endpoint (Endpoint2) for bulk OUT transfer (reception) and takes the actions according to the received requests and data.

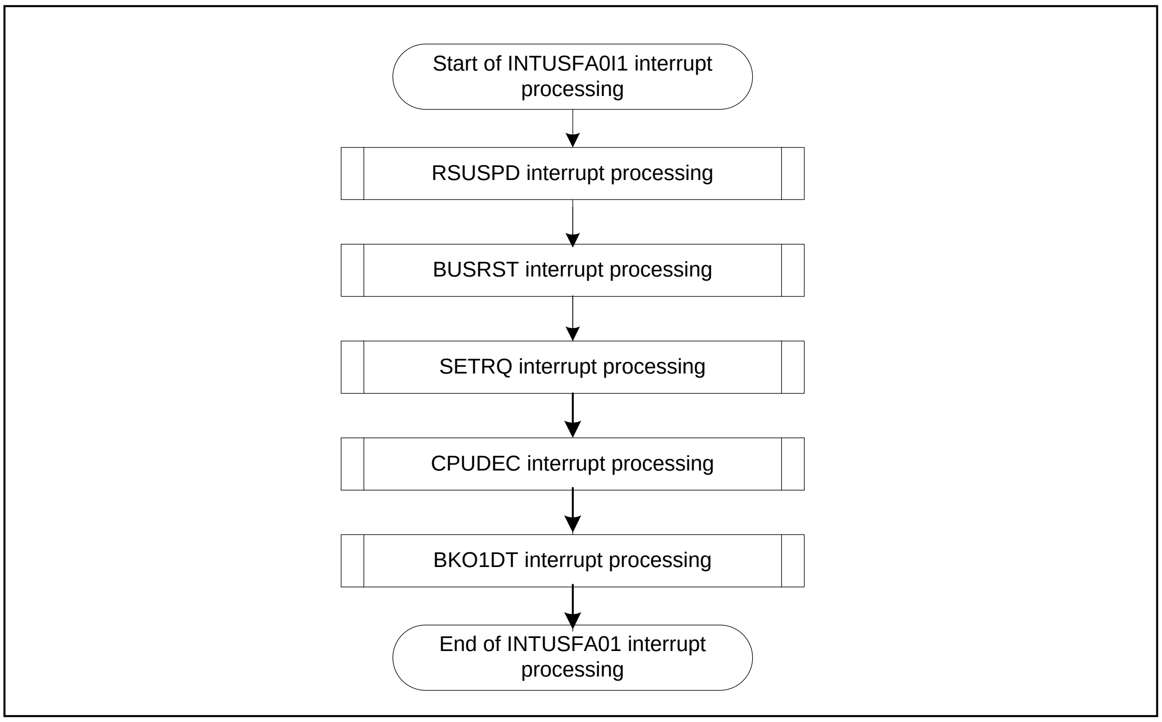

Start of INTUSFA0I1 interrupt processing
RSUSPD interrupt processing
BUSRST interrupt processing
SETRQ interrupt processing
CPUDEC interrupt processing
BKO1DT interrupt processing
End of INTUSFA01 interrupt processing

Figure 4.12 INTUSFA0I1 Interrupt Handler Processing Flow

(1) RSUSPD interrupt processing

The interrupt handler recognizes the occurrence of an RSUSPD interrupt when the RSUSPD bit of the USFA0IS0 register is set to 1.
The interrupt handler takes the following actions if an RSUSPD interrupt has occurred:

- Clears the interrupt source (sets the RSUSPDC bit of the USFA0IC0 register to 0.)
- Determines the suspend/resume state.

(2) Suspend-time processing

The interrupt handler determines that the endpoint is in the suspend state if the RSUM bit of the USFA0EPS1 register is set to 1.
The interrupt handler does not perform the subsequent processing and terminates the INTUSFA0I1 interrupt processing if the resume/suspend flag (rs_flag) is already set to "SUSPEND (0x00)" in the Suspend state.
If the resume/suspend flag (rs_flag) is not set to "SUSPEND," the interrupt handler sets that flag to "SUSPEND" and clears all of the USB interrupt sources. This causes the subsequent INTUSFA0I1 interrupt processing to be skipped.

(3) BUSRST interrupt processing

The interrupt handler recognizes the occurrence of a BUSRST interrupt when the BUSRST bit of the USFA0IS0 register is set to 1.

The interrupt handler takes the following actions if a BUSRST interrupt has occurred:

- Clears the interrupt source (sets the BUSRST bit of the USFA0IC0 register to 0).
- Sets the BUS Reset interrupt flag (usbf_busrst_flg) to 1.
- Clears FIFO for the bulk endpoints.

(4) SETRQ interrupt processing

The interrupt handler recognizes the occurrence of this interrupt when the SETRQ bit of the USFA0IS0 register is set to 1.

The interrupt handler takes the following actions if a SETRQ interrupt has occurred:

- Clears the interrupt source (sets the SETRQ bit of the USFA0IC0 register to 0).
- Performs automatically responded request (SET_XXXX) processing.

(5) Automatically responded request (SET_XXXX) processing

The interrupt handler recognizes that a SET_CONFIGURATION request has been received and automatically processed when the SETCON bit of the UF0SET register is set to 1.

The interrupt handler sets the BUS Reset interrupt flag (usbf_busrst_flg) to 0 when automatic processing is performed.

(Note) Examine the value of the UF0CNF register to confirm more exactly that the Configured state has been entered.

(6) CPUDEC interrupt processing

The interrupt handler recognizes the occurrence of this interrupt when the CPUDEC bit of the USFA0IS1 register is set to 1.

The interrupt handler takes the following actions if a CPUDEC interrupt has occurred:

- Clears the port interrupt source (sets the PORT bit of the USFA0IC1 register to 0).
- Reads the receive data from the FIFO and constructs request data.
- Performs request processing.

(7) Request processing

The interrupt handler checks to determine if the request data is not to be automatically responded by the hardware (standard, class, or vendor) and processes the request according to its request type.

Endpoint0 is an endpoint dedicated to control transfer. During the enumeration processing performed at plug-in time, almost all standard device requests are automatically processed by the hardware. Here, the standard requests that are not to be automatically processed by hardware and the class and vendor requests are processed.

(8) BKO1DT interrupt processing

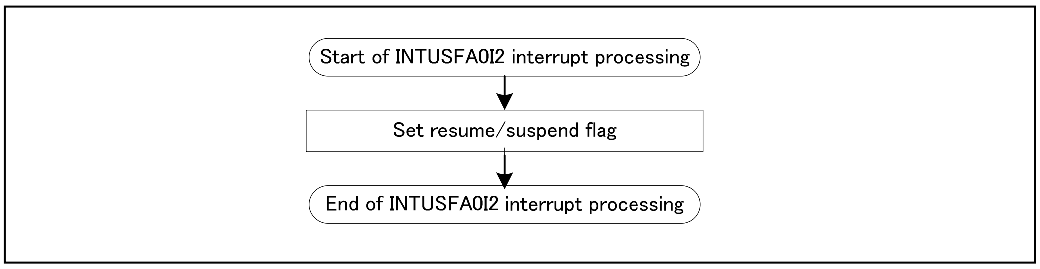
The interrupt handler recognizes the occurrence of this interrupt when the BKODT bit of the USFA0IS3 register is set to 1.

The interrupt handler takes the following actions if a BKODT interrupt has occurred:

- Clears the BKODT interrupt source (sets the BKO1DT bit of the USFA0IC3 register to 0).
- Calls the CBW data receive processing function (usbf850_rx_cbw) to receive the CBW data.

4.2.4 USBF Resume Interrupt Processing (INTUSFA0I2)

The INTUSFA0I2 interrupt handler performs processing when a resume interrupt occurs.
During resume interrupt processing, the resume/suspend flag (rs_flag) is set to "RESUME (0x01)."
The processing to be performed when rs_flag is set to "RESUME" is accomplished by the main routine.

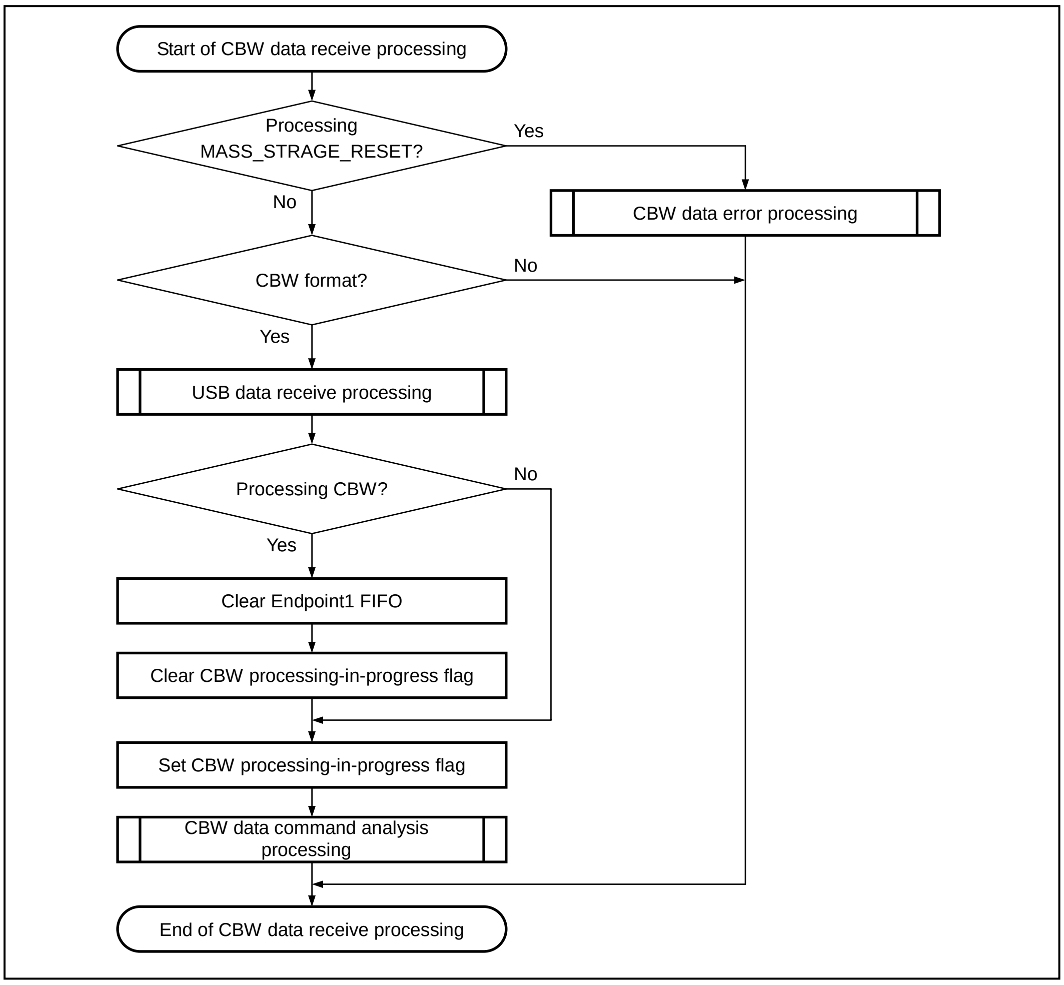


Start of INTUSFA0I2 interrupt processing
Set resume/suspend flag
End of INTUSFA0I2 interrupt processing

Figure 4.13 INTUSFA0I2 Interrupt Handler Processing Flow

4.2.5 CBW Data Receive Processing

The CBW data receive processing routine reads data from the FIFO for the bulk OUT endpoint (Endpoint2) and calls the command analysis routine for the CSW data.



Start of CBW data receive processing
Processing MASS_STRAGE_RESET?
Yes
No
CBW data error processing
CBW format?
No
Yes
USB data receive processing
Processing CBW?
No
Yes
Clear Endpoint1 FIFO
Clear CBW processing-in-progress flag
Set CBW processing-in-progress flag
CBW data command analysis processing
End of CBW data receive processing

Figure 4.14 CBW Data Receive Processing Flow

(1) Determining whether the MASS_STORAGE_RESET processing is in progress

The routine recognizes that MASS_STORAGE_RESET processing is in progress if the MASS_STRAGE_RESET processing flag (mass_storage_reset) is set to 1.

If the processing is in progress, the interrupt handler calls the CBW data error processing function (usbf850_cbw_error) and terminates the CBW data receive processing.

(2) Determining the CBW format

The routine gets, from the UF0 bulk out 1 length register (USFA0BO1L), the size (data length) of the data that is stored in the bulk OUT endpoint (Endpoint2). The data is judged to match the CBW format if the data length is 31 bytes.

The interrupt handler terminates CBW data receive processing if the data does not conform to the CBW format.

For CBW format data, the interrupt handler calls the USB data receive processing function (usbf850_data_receive) and continues processing.

(3) Determining whether the CBW processing is in progress

The routine recognizes that CBW processing is in progress if the CBW processing flag (cbw_in_cbw) is set to "USB_CBW_PROCESS (0x01)."

If the processing is in progress, the interrupt handler clears the FIFO for Endpoint1 and sets the CBW processing-in-progress flag (cbw_in_cbw) to "USB_CBW_END (0x00)."

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

The routine sets the CBW processing-in-progress flag (cbw_in_cbw) to "USB_CBW_PROCESS (0x01)."

(5) CBW command processing

The routine calls the CBW command analysis function (usbf850_storage_cbwchk) to process the SCSI command that is received.

4.2.6 SCSI Command Processing

When CBW data is received via the USB, the CBW command analysis function (usbf850_storage_cbwchk) is called to process the received SCSI command.

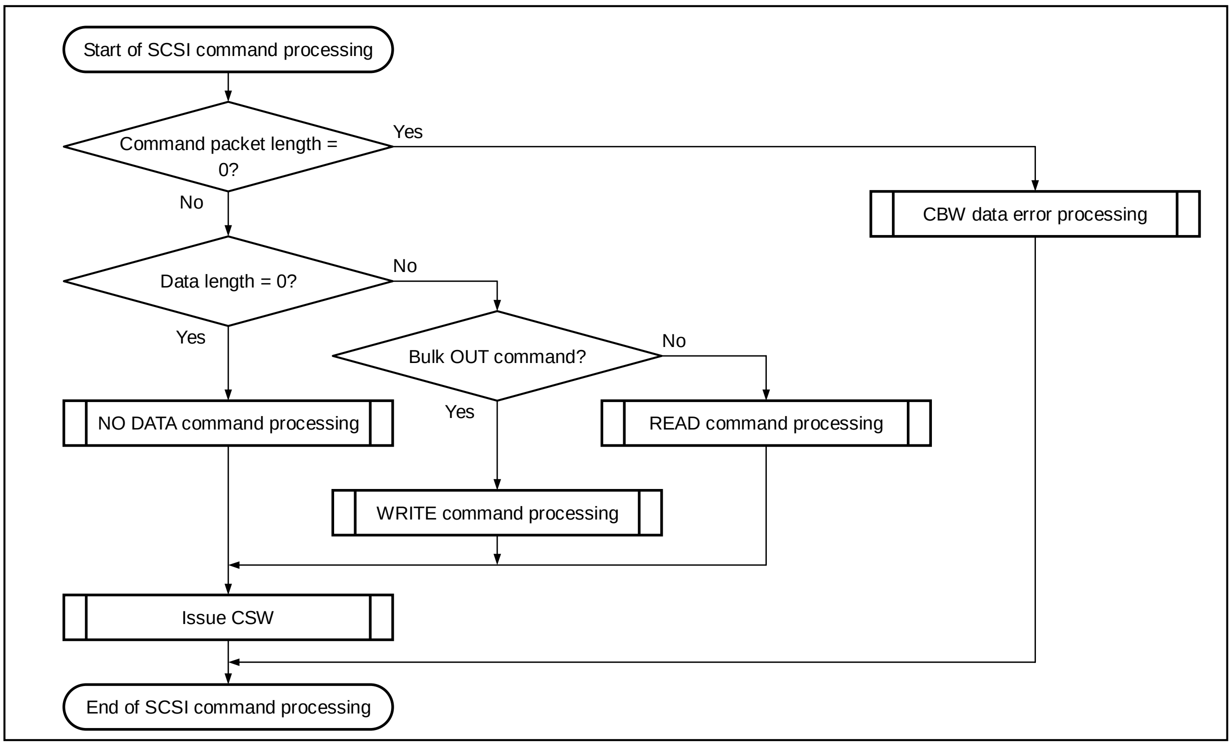

Start of SCSI command processing
Command packet length = 0?
Yes
No
CBW data error processing
Data length = 0?
No
Yes
Bulk OUT command?
No
Yes
NO DATA command processing
READ command processing
WRITE command processing
Issue CSW
End of SCSI command processing

Figure 4.15 SCSI Command Processing Flow

(1) Checking for a SCSI command

The command analysis function recognizes that the CBW data is not an SCSI command if the command packet length (bCBWCBLength) is 0x00.

If no SCSI command is identified, the command analysis function calls the CBW data error processing function (usbf850_cbw_error) and terminates the SCSI command processing.

(2) Checking for a NO DATA command

The command analysis function recognizes that the CBW data is a NO DATA command if the length of the data to be transferred in the data phase (dCBWDataTransferLength) is set to 0x00000000.

For a NO DATA command, the command analysis function calls the NO DATA command processing function (usbf850_no_data) to perform the processing associated with the received command.

Upon completion of the command processing, the command analysis function calls the CSW response processing function (usbf850_csw_ret) to send CSW.

(3) Checking the direction of data transfer

If bit 7 of the transfer direction (bmCBWFlags) is set to 0, the command analysis function identifies a WRITE command and calls the DATA OUT command processing function (usbf850_data_out) to perform the processing associated with the received command.

If bit 7 of bmCBWFlags is set to 1, the command analysis function identifies a READ command and calls the DATA IN command processing function (usbf850_data_in) to perform the processing associated with the received command.

Upon completion of the command processing, the command analysis function calls the CSW response processing function (usbf850_csw_ret) to send CSW.

4.2.7 Suspend/Resume Processing

The suspend/resume processing is executed within the main routine according to the processing flow shown below.

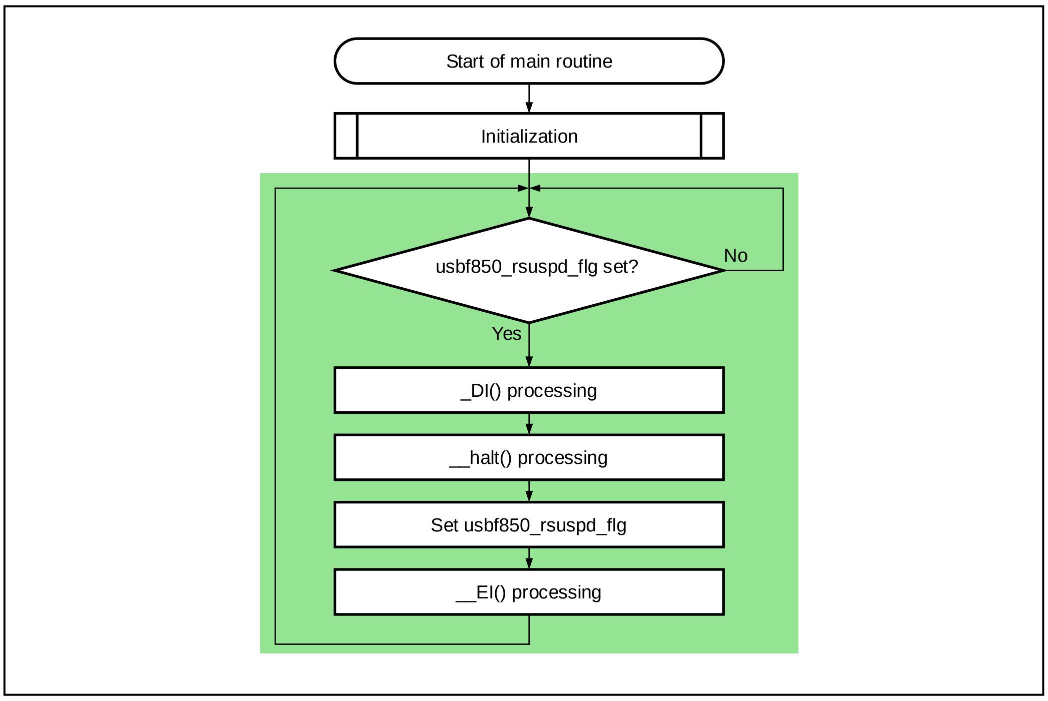


Figure 4.16 Suspend/Resume Processing Flow

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

The main routine monitors the resume/suspend flag (usbf850_rsuspd_flg) that is set up by the sample driver. The value of the flag being "SUSPEND (0x00)" indicates that the USB bus in the suspend state.

(2) Disabling CPU interrupts.

The main routine disables CPU interrupts if the resume/suspend flag (usbf850_rsuspd_flg) is set to "SUSPEND (0x00)."

(3) CPU HALT processing

The processor is stopped and placed in the HALT state. The restoration of the processor from the HALT state for processing resumption is triggered by a maskable interrupt, NMI, or reset. In this sample program, processing is resumed by an INTUSFA0I2 resume interrupt.

(4) Updating the resume/suspend flag (usbf850_rsuspd_flg)

The main routine sets the resume/suspend flag (usbf850_rsuspd_flg) to “RESUME(0x01).”

(5) Enabling CPU interrupts

The main routine enables CPU interrupts. This completes the resume processing.

4.3 Function Specifications

This section describes the functions that are implemented in the sample driver.

4.3.1 List of Functions

Table 4.40 shows a list of functions that are implemented in the source files for the sample driver.

Table 4.40 Sample Driver Functions (1/2)

Source File	Function Name	Description
main.c	main	Main routine
	cpu_init	Initializes the CPU.
	SetProtectReg	Processes access to a write-protected register.
usbf850.c	usbf850_init	Initializes the USB function controller.
	usbf850_intusbf0	Monitors Endpoint0 and controls responses to requests.
	usbf850_intusbf1	Processes resume interrupts.
	usbf850_data_send	Sends USB data.
	usbf850_data_receive	Receives USB data.
	usbf850_rdata_length	Gets USB receive data length.
	usbf850_send_EP0	Sends at Endpoint0.
	usbf850_receive_EP0	Receives at Endpoint0.
	usbf850_send_null	Sends Null packets to Bulk/ Interrupt In Endpoint.
	usbf850_sendnullEP0	Sends out NULL packet for Endpoint0.
	usbf850_sendstallEP0	Returns STALL for Endpoint0.
	usbf850_ep_status	Notifies FIFO state of Bulk/ Interrupt In Endpoint.
	usbf850_fifo_clear	Clears FIFOs for endpoints other than Endpoint0.
	usbf850_standardreq	Processes a standard request.
	usbf850_getdesc	Processes a GET_DESCRIPTOR request.
usbf850_storage.c	usbf850_classreq	Processes an MSC class request.
	usbf850_blkonly_mass_storage_reset	Processes a Mass Storage Reset request.
	usbf850_max_lun	Processes a Get Max Len request
	usbf850_rx_cbw	Receives CBW data.
	usbf850_storage_cbwchk	Analyzes a CBW data command.
	usbf850_cbw_error	Processes CBW data errors.
	usbf850_no_data	Processes a SCSI NO DATA command.
	usbf850_data_in	Processes a SCSI WRITE command.
	usbf850_data_out	Processes a SCSI READ command.
	usbf850_csw_ret	Processes a CSW response.
	usbf850_bulkin_stall	Controls bulk IN STALL responses.
	usbf850_bulkout_stall	Controls bulk OUT STALL responses.

Table 4.41 Sample Driver Functions (2/2)

Source File	Function Name	Description
scsi_cmd.c	scsi_command_to_ata	Executes a SCSI command.
	ata_test_unit_ready	Processes the TEST UNIT READY command.
	ata_seek	Processes the SEEK command.
	ata_start_stop_unit	Processes the START STOP UNIT command.
	ata_synchronize_cache	Processes the SYNCHRONIZE CACHE command.
	ata_request_sense	Processes the REQUEST SENSE command.
	ata_inquiry	Processes the INQUIRY command.
	ata_mode_select	Processes the MODE SELECT(6) command.
	ata_mode_select10	Processes the MODE SELECT(10) command.
	ata_mode_sense	Processes the MODE SENSE(6) command.
	ata_mode_sense10	Processes the MODE SENSE(10) command.
	ata_read_format_capacities	Processes the READ FORMAT CAPACITIES command.
	ata_read_capacity	Processes the READ CAPACITY command.
	ata_read6	Processes the READ(6) command.
	ata_read10	Processes the READ(10) command.
	ata_write6	Processes the WRITE(6) command.
	ata_write10	Processes the WRITE(10) command.
	ata_verify	Processes the VERIFY command.
	ata_write_verify	Processes the WRITE VERIFY command.
	ata_write_buff	Processes the WRITE BUFFER command.
	scsi_to_usb	Performs USB data transmission processing (SCSI commands).

4.3.2 Correlation among the Sample Driver Functions

There are some sample driver functions that call another function during their execution. This function call relationships are shown below.

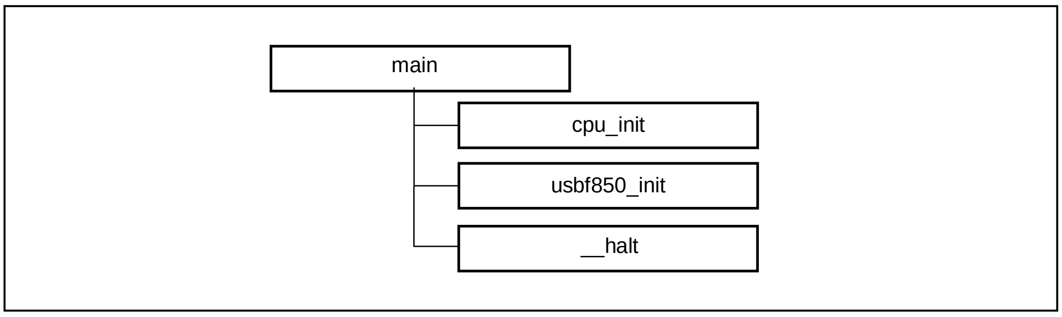


Figure 4.17 Function Calls within main Processing

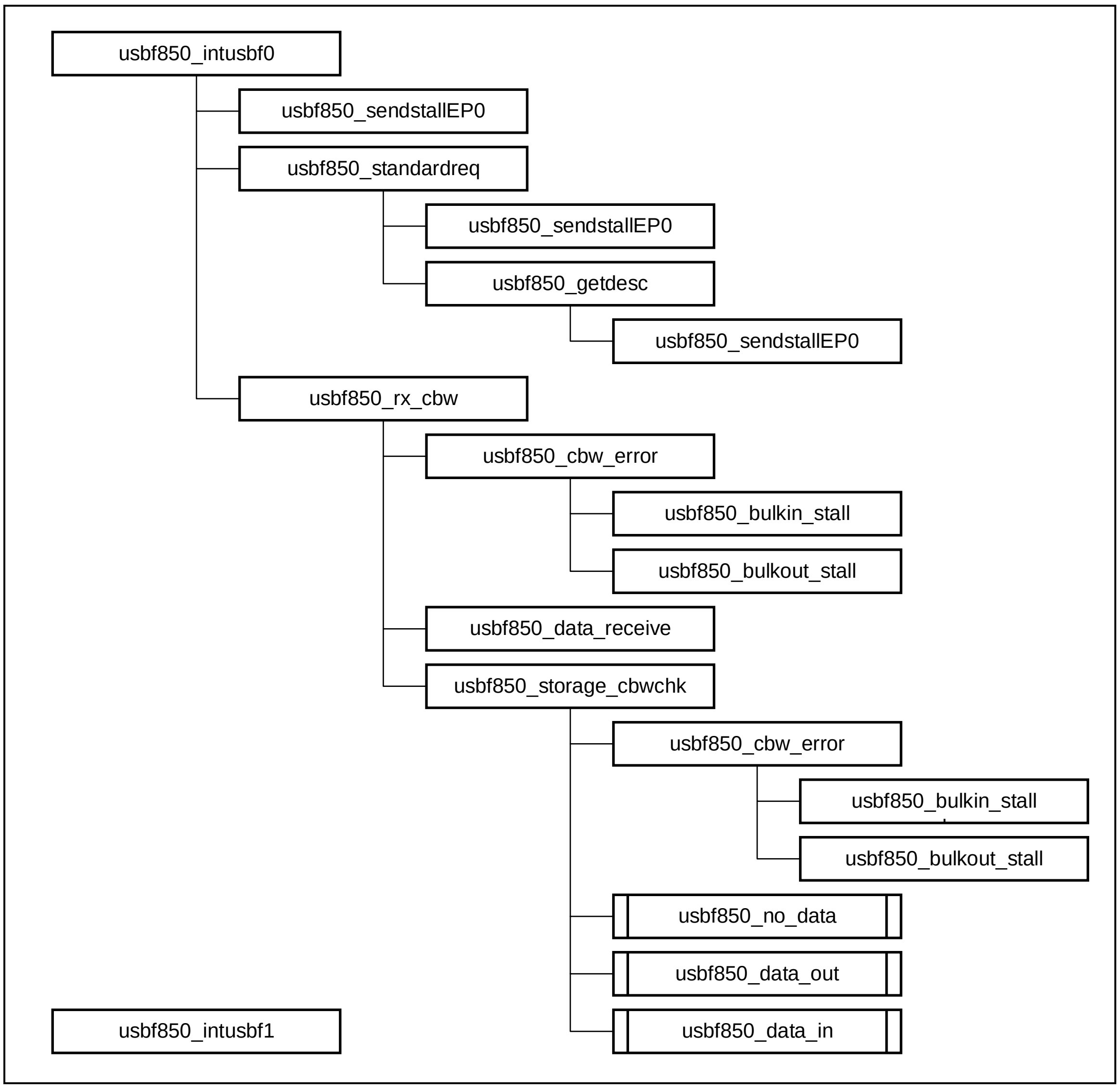


Figure 4.18 Function Calls within USB Interrupt Processing

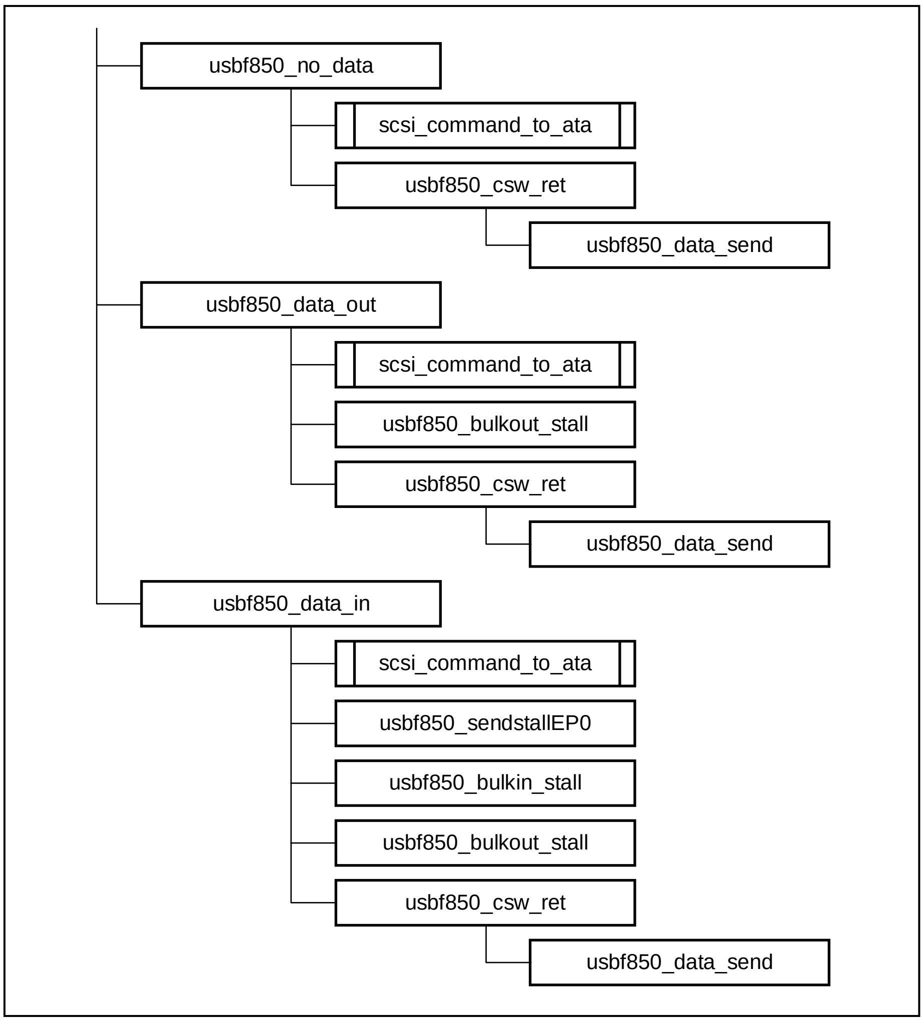


Figure 4.19 Function Calls within CBW/CSW Processing

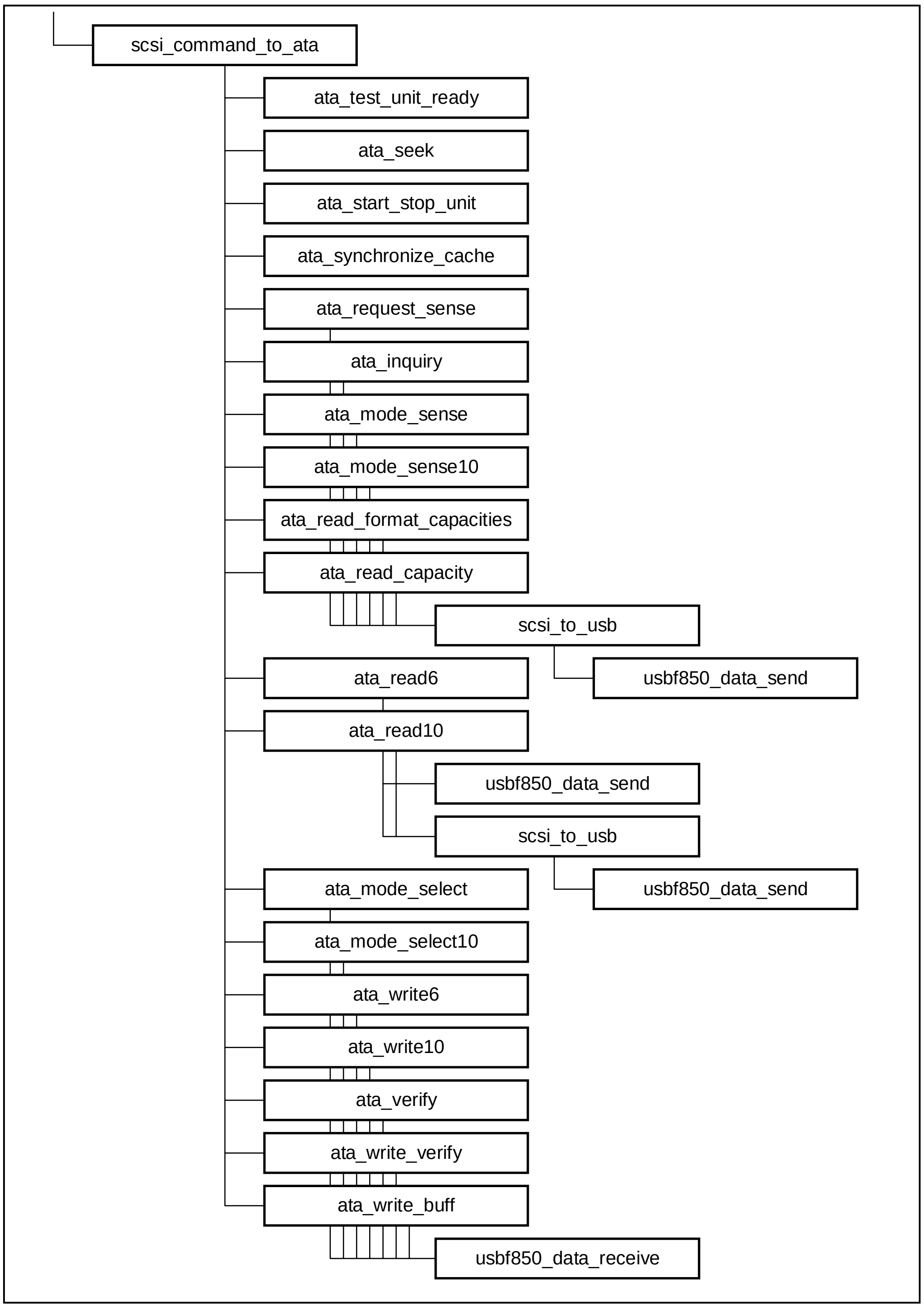


Figure 4.20 Function Calls within SCSI Command Processing

4.3.3 Function Descriptions

This section contains a description of the functions that are implemented in the sample driver.

(1) Functional description format

The functional descriptions are given in the format shown below.

Function Name

[Synopsis]

Gives a synopsis of the function.

[C language format]

Shows the format in C language

[Parameters]

Describes the parameters (arguments).

Parameter	Description
Parameter type, name	Parameter outline

[Return Value]

Describes the return value.

Symbol	Description
Type of return value, name	Return value outline

[Function]

Explains the function.

(2) Main routine functions

main

[Synopsis]

Perform main processing.

[C language format]

void main(void)

[Parameters]

None

[Return Value]

None

[Function]

This function is called first when the sample driver is started.
The function calls the USB initialization function (usbf850_init), then monitors the resume/suspend flag (usbf850_rsuspd_flg). It performs suspend processing when the usbf850_rsuspd_flg is set to "SUSPEND (0x00)."

cpu_init

[Synopsis]

Initialize CPU.

[C language format]

void cpu_init(void)

[Parameters]

None

[Return Value]

None

[Function]

This function is called during initialization processing.

It initializes the H bus and sets up the USB clock and other parameters that are necessary to use the USB function controller.

SetProtectReg

[Synopsis]

Access write-protected register.

[C language format]

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

[Parameters]

Parameter	Description
volatile UINT32 *dest_reg	Protected register address
UINT32 wr_dt	Write value
volatile UINT8 *prot_reg	Protect command register address

[Return Value]

None

[Function]

This function writes a value into the given write-protected register.

(3) USB function controller processing functions

usbf850_init

[Synopsis]

Initialize USB function controller.

[C language format]

void usbf850_init(void)

[Parameters]

None

[Return Value]

None

[Function]

This function is called during initialization processing.
It allocates and sets up the data area, and sets interrupt request masks and other parameter items that are necessary to use the USB function controller.

usbf850_intusbf0

[Synopsis]

INTUSFA0I1 interrupt handler processing.

[C language format]

void usbf850_intusbf0(void)

[Parameters]

None

[Return Value]

None

[Function]

This function is called as a USB interrupt handler (INTUSFA0I1).
It monitors the endpoint for control transfer (Endpoint0) and the endpoint for bulk OUT transfer (reception) (Endpoint2), and takes the required actions according to the received request or command.
For Endpoint0, the function checks for RSUSPD, BUSRST, SETRQ, and CPUDEC interrupts. When a CPUDEC interrupt occurs, the function decodes the request data and calls the pertinent function for response processing.
For Endpoint2, the function checks for BKO1DT interrupts. When a BKO1DT interrupt occurs, the function calls the CBW data receive function (usbf850_rx_cbw) and takes the required actions according to the received command.

usbf850_intusbf1

[Synopsis]

Perform INTUSFA0I2 interrupt handler processing.

[C language format]

void usbf850_intusbf1(void)

[Parameters]

None

[Return Value]

None

[Function]

This function is called as a USB resume interrupt (INTUSFA0I2) handler.
It sets the resume/suspend flag (usbf850_rsuspd_flg) to "RESUME (0x01)."

usbf850_data_send

[Synopsis]

Send USB data.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *data	Pointer to transmit data buffer
INT32 len	Transmit data length
INT8 ep	Endpoint number of the endpoint to be used for data transmission

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function transfers data from the transmit data buffer to the FIFO for the specified endpoint, one byte at a time.

usbf850_data_receive

[Synopsis]

Receive USB data.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *data	Pointer to receive data buffer
INT32 len	Receive data length
INT8 ep	Endpoint number of the endpoint to be used for data reception

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function reads data from the FIFO for the specified endpoint into the receive data buffer, one byte at a time.

usbf850_rdata_length

[Synopsis]

Get USB receive data length.

[C language format]

void usbf850_rdata_length(INT32 *len , INT8 ep)

[Parameters]

Parameter	Description
INT32* len	Pointer to the address storing the receive data length
INT8 ep	Endpoint number of the data receiving endpoint

[Return Value]

None

[Function]

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

usbf850_send_EP0

[Synopsis]

Send USB data for Endpoint0.

[C language format]

INT32 usbf850_send_EP0(UINT8* data, INT32 len)

[Parameters]

Parameter	Description
UINT8* data	Pointer to transmit data buffer
INT32 len	Transmit data size

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function transfers data from the transmit data buffer to the transmit FIFO for Endpoint0, one byte at a time.

usbf850_receive_EP0

[Synopsis]

Receive USB data for Endpoint0.

[C language format]

INT32 usbf850_receive_EP0(UINT8* data, INT32 len)

[Parameters]

Parameter	Description
UINT8* data	Pointer to receive data buffer
INT32 len	Receive data size

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function reads data from the receive FIFO for Endpoint0 into the receive data buffer, one byte at a time.

usbf850_send_null

[Synopsis]

Send Null packet for Bulk/Interrupt In Endpoint.

[C language format]

INT32 usbf850_send_null(INT8 ep)

[Parameters]

Parameter	Description
INT8 ep	Endpoint number of the data transmitting endpoint

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function sends a Null packet from the USB function controller by clearing the FIFO for the specified Endpoint (for transmission) and setting the bit that specifies the end of data to 1.

usbf850_sendnullEP0

[Synopsis]

Send NULL packet for Endpoint0.

[C language format]

void usbf850_sendnullEP0(void)

[Parameters]

None

[Return Value]

None

[Function]

This function sends a Null packet from the USB function controller by clearing the FIFO for Endpoint0 and setting the bit that specifies the end of data to 1.

usbf850_sendstallEP0

[Synopsis]

Send STALL response for Endpoint0.

[C language format]

void usbf850_sendstallEP0(void)

[Parameters]

None

[Return Value]

None

[Function]

This function causes the USB function controller to return a STALL response by setting the bit that indicates the use of a STALL handshake to 1.

usbf850_ep_status

[Synopsis]

Notify state of FIFO for Bulk/ Interrupt In Endpoint.

[C language format]

INT32 usbf850_ep_status(INT8 ep)

[Parameters]

Parameter	Description
INT8 ep	Endpoint number of the data transmitting endpoint

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_RESET	Bus Reset processing in progress
DEV_ERROR	Abnormal termination

[Function]

This function notifies the state of the FIFO for the specified Endpoint (for transmission).

usbf850_fifo_clear

[Synopsis]

Clear FIFO for Bulk/ Interrupt Endpoint.

[C language format]

void usbf850_fifo_clear(INT8 in_ep, INT8 out_ep)

[Parameters]

Parameter	Description
INT8 in_ep	Data transmitting Endpoint
INT8 out_ep	Data receiving Endpoint

[Return Value]

None

[Function]

This function clears the FIFO for the specified Endpoint (Bulk/Interrupt) and the data receive flag (usbf850_rdata_flg).

usbf850_standardreq

[Synopsis]

Process standard request not automatically responded by USB function controller.

[C language format]

void usbf850_standardreq(USB_SETUP *req_data)

[Parameters]

Parameter	Description
USB_SETUP *req_data	Pointer to area storing the request data

[Return Value]

None

[Function]

This function is called by the Endpoint0 monitoring routine.
It calls the GET_DESCRIPTOR request processing function (usbf850_getdesc) if the decoded request is GET_DESCRIPTOR. For the other requests, the function calls the STALL response processing function for Endpoint0 (usbf850_sendstallEP0).

usbf850_getdesc

[Synopsis]

Process GET_DESCRIPTOR request.

[C language format]

void usbf850_getdesc(USB_SETUP *req_data)

[Parameters]

Parameter	Description
USB_SETUP *req_data	Pointer to area storing the request data

[Return Value]

None

[Function]

This function is called to process standard requests that are not automatically responded by the USB function controller.
If the decoded request asks for a string descriptor, the function calls the USB data transmit processing function (usbf850_data_send) to send a string descriptor from Endpoint0. If a descriptor other than the string descriptor is requested, the function calls the STALL response processing function for Endpoint0 (usbf850_sendstallEP0).

(4) USB Mass storage class processing functions

usbf850_classreq

[Synopsis]

Process MSC class request.

[C language format]

void usbf850_classreq(USB_SETUP *req_data)

[Parameters]

Parameter	Description
USB_SETUP *req_data	Pointer to area storing the request data

[Return Value]

None

[Function]

This function is called for the CPUDEC interrupt source during INTUSFA0I1 interrupt processing. If the decoded request is the one that is specific to the communication device class, the function calls the corresponding request processing function. In the other cases, the function sends a STALL to Endpoint0.

usbf850_blkonly_mass_storage_reset

[Synopsis]

Perform Mass Storage Reset processing.

[C language format]

void usbf850_blkonly_mass_storage_reset(USB_SETUP *req_data)

[Parameters]

Parameter	Description
USB_SETUP *req_data	Pointer to area storing the request data

[Return Value]

None

[Function]

This function clears the FIFOs for Endpoint1 and Endpoint2 and sets up a STALL response. Subsequently, the function sends a NULL packet from Endpoint0.

usbf850_max_lun

[Synopsis]

Perform Get Max Lun processing.

[C language format]

void usbf850_max_lun(USB_SETUP *req_data)

[Parameters]

Parameter	Description
USB_SETUP *req_data	Pointer to area storing the request data

[Return Value]

None

[Function]

This function sends the number of logical units (Logical Unit Number) of the mass storage device.

usbf850_rx_cbw

[Synopsis]

Receive CBW data.

[C language format]

void usbf850_rx_cbw(void)

[Parameters]

None

[Return Value]

None

[Function]

This function reads CBW data from the FIFO for the bulk IN endpoint (Endpoint2) and calls the CBW data command analysis function (usbf850_storage_cbwchk).

usbf850_storage_cbwchk

[Synopsis]

Analyze and process CBW data command.

[C language format]

INT32 usbf850_storage_cbwchk(void)

[Parameters]

None

[Return Value]

The status established during CBW checking is returned.

Symbol	Description
DEV_OK	Normal termination
DEV_ERROR	Abnormal termination

[Function]

This function analyzes the CBW data, identifies the command type (NO DATA, DATA IN (WRITE), or DATA OUT (READ)), and processes the command.

usbf850_cbw_error

[Synopsis]

Perform CBW data error processing.

[C language format]

void usbf850_cbw_error(void)

[Parameters]

None

[Return Value]

None

[Function]

This command reports a STALL response for the bulk IN endpoint (Endpoint1) and bulk OUT endpoint (Endpoint2).

usbf850_no_data

[Synopsis]

Process SCSI NO DATA command.

[C language format]

void usbf850_no_data(void)

[Parameters]

None

[Return Value]

None

[Function]

This function performs NO DATA command processing and returns the processing results in the CSW format.

usbf850_data_in

[Synopsis]

Performs SCSI DATA IN command.

[C language format]

void usbf850_data_in(void)

[Parameters]

None

[Return Value]

None

[Function]

This function performs DATA IN (WRITE) command processing and returns the processing results in the CSW format.

usbf850_data_out

[Synopsis]

Process SCSI DATA OUT command.

[C language format]

void usbf850_data_out(void)

[Parameters]

None

[Return Value]

None

[Function]

This function performs DATA OUT (READ) command processing and returns the processing results in the CSW format.

usbf850_csw_ret

[Synopsis]

Process CSW response.

[C language format]

INT32 usbf850_csw_ret(UINT8 status)

[Parameters]

Parameter	Description
UINT8 status	Results of command processing

[Return Value]

Results of CSW transmission processing

Symbol	Description
DEV_OK	Normal termination

[Function]

This function creates CSW format data from the processing results and sends it via USB.

usbf850_bulkin_stall

[C language format]

void usbf850_bulkin_stall(void)

[Parameters]

None

[Return Value]

None

[Function]

This function clears the FIFO for Endpoint1 and returns a STALL response.

usbf850_bulkout_stall

[C language format]

void usbf850_bulkout_stall(void)

[Parameters]

None

[Return Value]

None

[Function]

This function clears the FIFO for Endpoint2 and returns a STALL response.

(5) SCSI command processing functions

scsi_command_to_ata

[Synopsis]

Process SCSI command execution.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

The processing result of the SCSI command is returned.

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function identifies a SCSI command and performs the corresponding command processing. If no pertinent command is found, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_test_unit_ready

[Synopsis]

Process TEST UNIT READY command.

[C language format]

INT32 ata_test_unit_ready(INT32 TransFlag)

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in NO DATA command

[Function]

This function clears the sense data (sense key = 0x00). If the transfer direction is not NO DATA, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_seek

[Synopsis]

Process SEEK command.

[C language format]

INT32 ata_seek(INT32 TransFlag)

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command

[Function]

This function clears the sense data (sense key = 0x00). If the transfer direction is not NO DATA, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_start_stop_unit

[Synopsis]

Process START STOP UNIT command.

[C language format]

INT32 ata_start_stop_unit(INT32 TransFlag)

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return Value]

Processing result

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command

[Function]

This function clears the sense data (sense key = 0x00). If the transfer direction is not NO DATA, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_synchronize_cache

[Synopsis]

Process SYNCHRONIZE CACHE command.

[C language format]

INT32 ata_synchronize_cache(INT32 TransFlag)

[Parameters]

Parameter	Description
INT32 TransFlag	Direction of data transfer

[Return Value]

Processing result

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command

[Function]

This function clears the sense data (sense key = 0x00). If the transfer direction is not NO DATA, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_request_sense

[Synopsis]

Process REQUEST SENSE command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command
DEV_ERR_READ	Transfer direction error in a READ command

[Function]

This function sends sense data.

If the specified data size is set to 0 and the transfer direction is not NO DATA, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_inquiry

[Synopsis]

Process INQUIRY command

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends INQUIRY data. If the CMDDT and EVPD bits of command byte 1 are both set to 1, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_mode_select

[Synopsis]

Process MODE SELECT(6) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and updates the MODE SELECT data table with the receive data.
If an illegal transfer direction or data size is found, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_mode_select10

[Synopsis]

Process MODE SELECT(10) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and updates the MODE SELECT(10) data table with the receive data.

If an illegal transfer direction or data size is found, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_mode_sense

[Synopsis]

Process MODE SENSE(6) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the MODE SENSE data.

ata_mode_sense10

[Synopsis]

Process MODE SENSE(10) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the MODE SENSE(10) data.

ata_read_format_capacities

[Synopsis]

Process READ FORMAT CAPACITIES command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the FORMAT CPACITY data.

ata_read_capacity

[Synopsis]

Process READ CAPACITY command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the CPACITY data.

ata_read6

[Synopsis]

Process READ(6) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the data that is read from the data area.
The read start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the Flag or Link bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_read10

[Synopsis]

Process READ(10) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and sends the data that is read from the data area.
The read start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the Flag or Link bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_write6

[Synopsis]

Process WRITE(6) command

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and writes the receive data into the data area.
The write start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the Flag or Link bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_write10

[Synopsis]

Process WRITE(10) command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and writes the receive data into the data area.
The write start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the Flag or Link bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_verify

[Synopsis]

Process VERIFY command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_NODATA	Transfer direction error in a NO DATA command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function writes the receive data into the data area.
The write start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the BYTCHK bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_write_verify

[Synopsis]

Process WRITE VERIFY command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

This function clears the sense data (sense key = 0x00) and writes the receive data into the data area.
The write start address is calculated from the LBA (Local Block Address) and block size in the SCSI command.
If the transfer direction or the Flag or Link bit of the SCSI command is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

ata_write_buff

[Synopsis]

Process WRITE BUFF command.

[C language format]

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

[Parameters]

Parameter	Description
UINT8 *ScsiCommandBuf	Pointer to buffer storing the SCSI command
UINT8 *pbData	Pointer to buffer storing command data
INT32 lDataSize	Data size
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_WRITE	Transfer direction error in a WRITE command
DEV_ERROR	Status other than the above or illegal request

[Function]

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

scsi_to_usb

[Synopsis]

Send USB data (SCSI command).

[C language format]

INT32 scsi_to_usb(UINT8 *pbData, INT32 TransFlag)

[Parameters]

Parameter	Description
UINT8 *pbData	Pointer to buffer storing command data
INT32 TransFlag	Direction of data transfer

[Return Value]

Symbol	Description
DEV_OK	Normal termination
DEV_ERR_READ	Transfer direction error in a READ command

[Function]

This function calls the USB data send function (usbf850_data_send) to send data from the bulk OUT endpoint (Endpoint1).

If the transfer direction is illegal, the function sets the sense key to ILLEGAL REQUEST and updates the sense data.

4.4 Data Structures

This section describes the data structures that are used by the sample driver.

(1) USB device request structure

The USB device request structure is defined in the file "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 4.21 USB Device Request Structure

(2) CBW data structure

The CBW data structure is defined in the file "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;                   /* Specifies data direction */
                                            /* (OUT/IN/NO DATA). */
      UINT8   bCBWLUN;                      /* Number of target device */
      UINT8   bCBWCBLength;                 /* Number of significant CBWCB bytes */
      UINT8   CBWCB[16];                    /* CBWCB (command) */
} CBW_INFO,*PCBW_INFO;
```

Figure 4.22 CBW Data Structure

(3) CSW data structure

The CSW data structure is defined in the file "usbf850_storage.h."

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

Figure 4.23 CSW Data Structure

(4) SCSI SENSE DATA Structure

The SCSI SENSE DATA structure is defined in the file "scsi_cmd.c."

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

Figure 4.24 SCSI SENSE DATA Structure

5. Development Environment

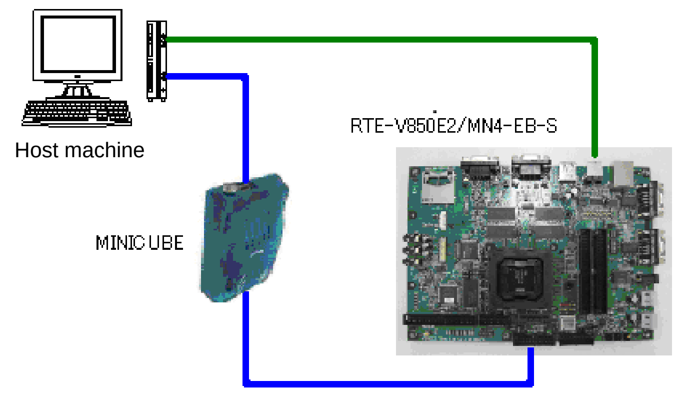
This section gives an example of constructing an environment for developing application programs using the USB mass storage class sample driver for the V850E2/MN4 and the procedures for debugging them in that environment.

5.1 Development Environment

This section introduces a sample development configuration of hardware and software tool products.

5.1.1 System Configuration

The system configuration in which the sample driver is to be used is shown in figure 5.1.



Remarks: See section 7, Outline of the Starter Kit, for the physical appearance and port configuration of the RTE-V850E2/MN4-EB-S.

Figure 5.1 System Configuration of the Development Environment

5.1.2 Program Development

The hardware and software that are summarized below are required to develop a system using the sample driver.

Table 5.1 Example of Program Development Environment Configuration

Component Products		Product Example	Remarks
Hardware	Host machine	—	PC/AT™ compatible (OS: Windows XP or Windows Vista®)
Software	Integrated development tool	CubeSuite	V1.40
		Multi	V5.1.7D
		IAR Embedded Workbench	V3.71
	Compiler	CX850	V1.00
		CCV850	V5.1.7D
		ICCV850	V3.71.2

5.1.3 Debugging

The hardware and software that are summarized below are required to debug a system using the sample driver.

Table 5.2 Example of Debugging Environment Configuration

Component Products		Product Example	Remarks
Hardware	Host machine	—	PC/AT compatible (OS: Windows XP or Windows Vista®))
	Target	RTE-V850E2/MN4-EB-S	Manufactured by MIDAS LAB
	USB cable	—	Connection between B receptacle to A receptacle
Software	Integrated development tool/debugger	CubeSuite	V1.40
		Multi	V5.1.7D
		IAR Embedded Workbench	V3.71
File	Device file	DF703512	For V850E2/MN4 (separately available for CubeSuite, Multi, and IAR Embedded Workbench)
	Host driver for debugging port	—	(Note 8)
	Project-related file	—	(Note 9)

(Note 8) Contact Renesas for product and ordering information.

(Note 9) The sample driver package comes with sample files that are built with CubeSuite, Multi, and IAR Embedded Workbench.

5.2 Setting up a CubeSuite Environment

This section explains the preparatory steps that are required to develop or debug using CubeSuite which is introduced in section 5.1, Development Environment. See section 5.4, Setting up a Multi Environment, when using Multi for program development and debugging. See section 5.6, Setting up IAR Embedded Workbench Environment, when using IAR Embedded Workbench for program development and debugging.

5.2.1 Setting up the Host Environment

You create a dedicated workspace on the host machine.

(1) Installing the CubeSuite integrated development tools

Install CubeSuite. Refer to the CubeSuite user’s manual for details.

(2) Expanding driver and other files

Store a set of distribution sample driver files in an arbitrary directory without modifying their folder structure.
Store the host driver for the debugging port in an arbitrary directory.

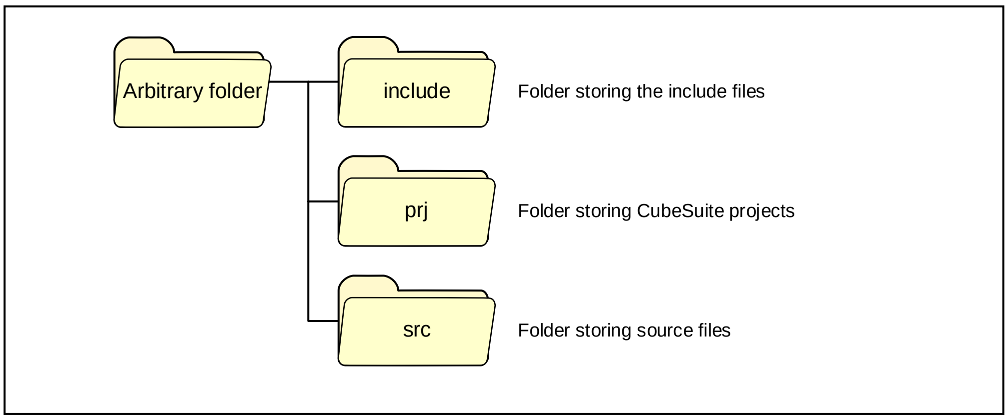

Arbitrary folder
include
Folder storing the include files
prj
Folder storing CubeSuite projects
src
Folder storing source files

Figure 5.2 Folder Configuration for the Sample Driver (CubeSuite Version)

(3) Installing device files

Copy the V850E2/MN4 device files for CubeSuite in the folder where CubeSuite is installed.

Example: D:\Renesas Electronics CubeSuite\CubeSuite\Device_Custom

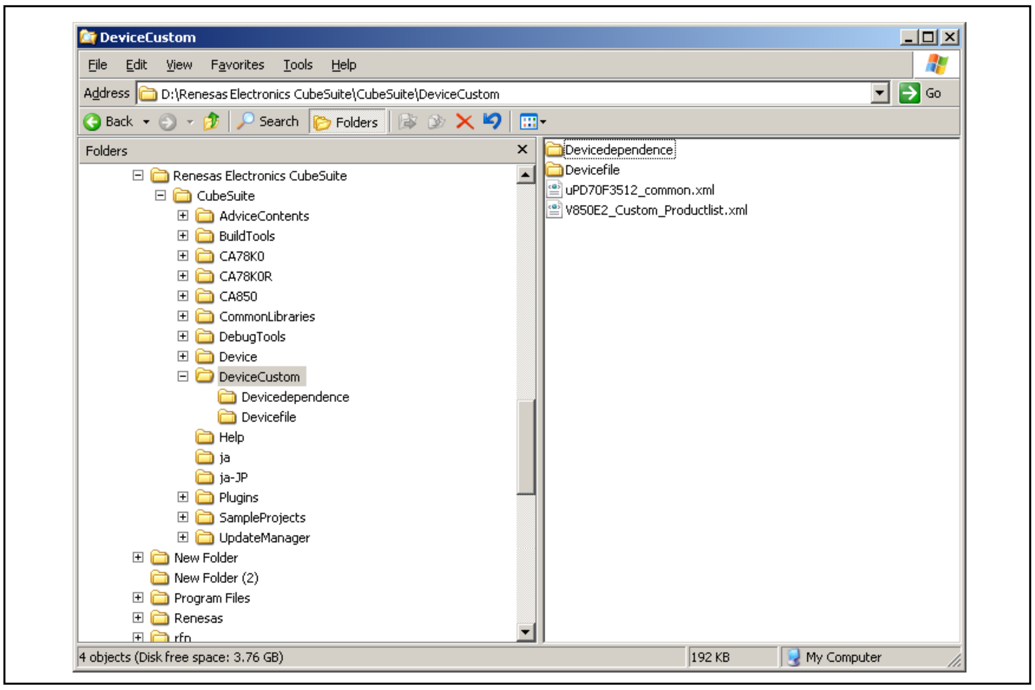


Figure 5.3 Example of Destination Folder for Storing the Device Files

(4) Setting up a workspace

Follow the procedure given below when using the project-related files that come with the sample driver package.

<1> Start CubeSuite and choose “Open File” from the “File” menu.

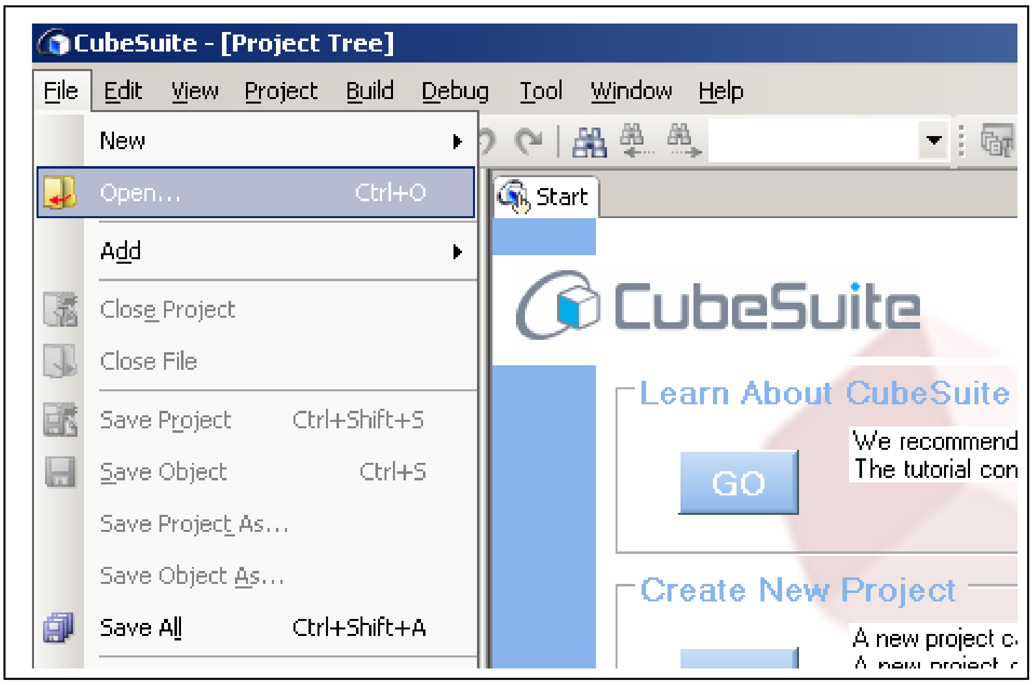
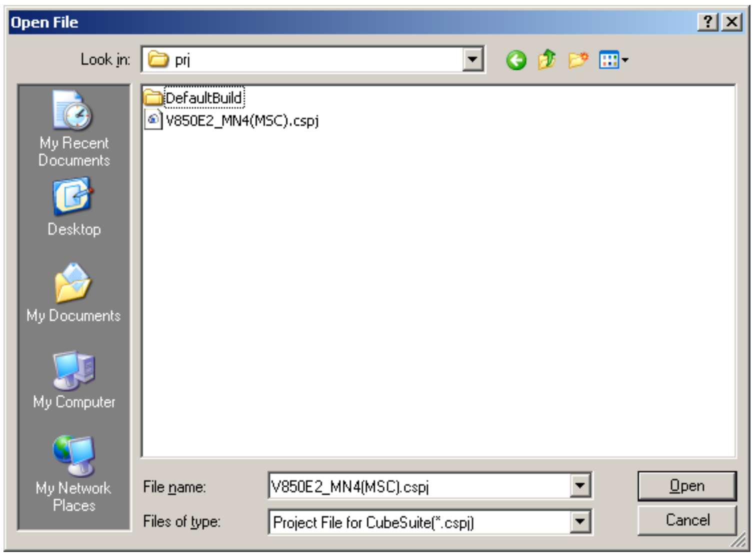


Figure 5.4 Choosing a CubeSuite Menu Item

<2> The “Open File” dialog box will appear. Select the project file for CubeSuite which is located in the “prj” folder in the directory in which the sample driver is installed.



File name: V850E2_MN4(MSC).cspj

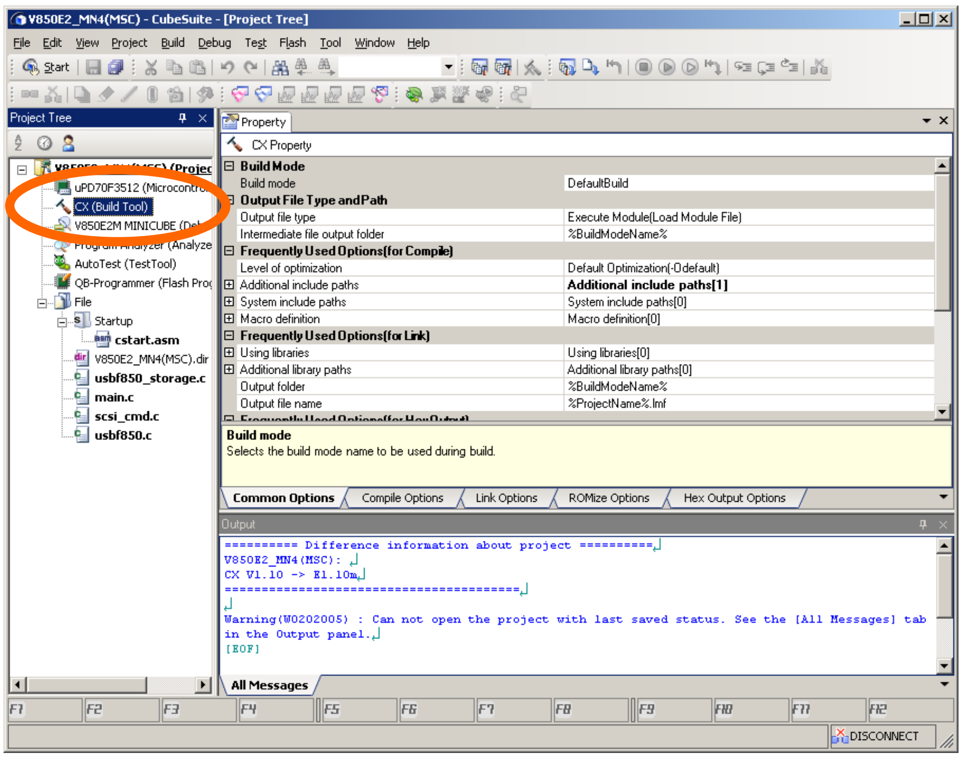
Files of type: Project File for CubeSuite(*.cspj)

Figure 5.5 Selecting the CubeSuite Project File

(5) Setting up the build tool

Follow the procedure given below to select the version of CX850 which is to be used as the build tool and to designate V850E2M MINICUBE as the debugging tool.

<1> Select "CX (build tool)" from the "Project Tree" for CubeSuite to display its properties.



CX Property

Build Mode	
Build mode	DefaultBuild
Output File Type and Path	
Output file type	Execute Module(Load Module File)
Intermediate file output folder	%BuildModeName%
Frequently Used Options(for Compile)	
Level of optimization	Default Optimization(-Odefault)
Additional include paths	**Additional include paths[1]**
System include paths	System include paths[0]
Macro definition	Macro definition[0]
Frequently Used Options(for Link)	
Using libraries	Using libraries[0]
Additional library paths	Additional library paths[0]
Output folder	%BuildModeName%
Output file name	%ProjectName%.lmf

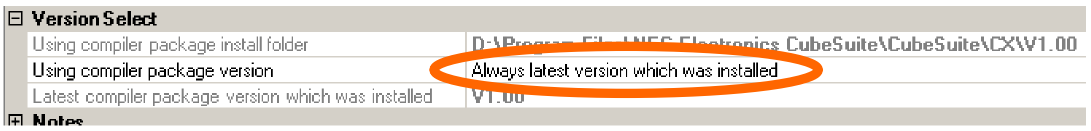
Build mode
Selects the build mode name to be used during build.

```
========== Difference information about project ==========
V850E2_MN4(MSC):
CX V1.10 -> E1.10m
==========================================

Warning(W0202005) : Can not open the project with last saved status. See the [All Messages] tab
in the Output panel.
[EOF]
```

Figure 5.6 Selecting the Build Tool

<2> Select the "Version Select" properties item and sets the "Using compiler package version" entry to "Always latest version which was installed."



Version Select	
Using compiler package install folder	D:\Program Files\NEC Electronics CubeSuite\CubeSuite\CX\V1.00
Using compiler package version	Always latest version which was installed
Latest compiler package version which was installed	V1.00
Notes	

Figure 5.7 Setting up the Compiler Package

<3> Select ”V850E2M MINICUBE (Debug Tool)” from the Project Tree and select ”Using Debug Tool”→”V850E2M MINICUBE” from the right-click menu.

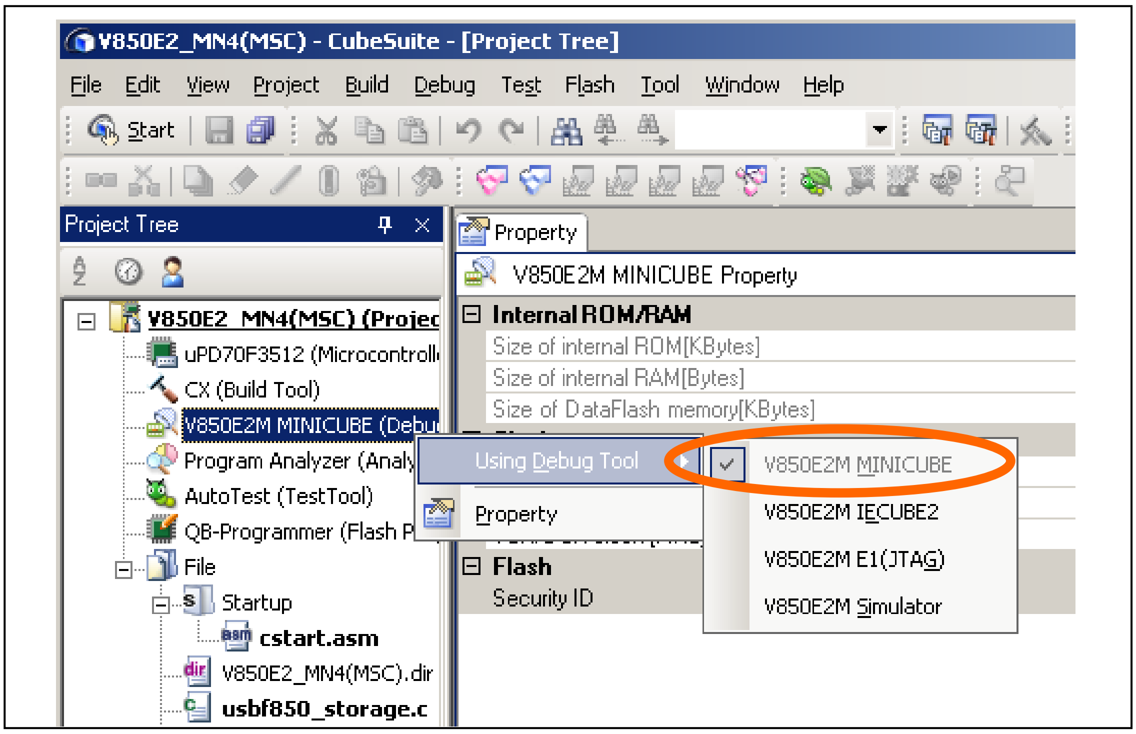


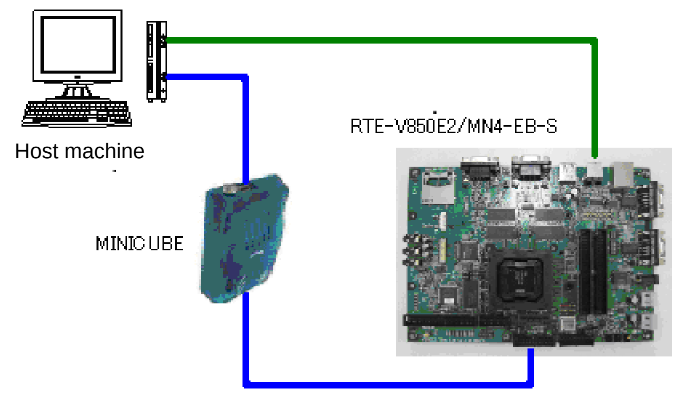
Figure 5.8 Selecting the Debugging Tool

5.2.2 Setting up the Target Environment

You connect the target device to be used for debugging to the host machine. The procedure is common to CubeSuite, Multi, and IAR Embedded Workbench.

(1) Connecting to the debugging port

Connect between the RTE-V850E2/MN4-EB-S and the host machine. Connect the RTE-V850E2/MN4-EB-S and the host machine via the MINICUBE for debugging. In addition, connect between the USB B type receptacle of the RTE-V850E2/MN4-EB-S and the USB receptacle of the host machine for the MSC.



Remarks: See section 7, Outline of the Starter Kit, for the physical appearance and port configuration of the RTE-V850E2/MN4-EB-S.

Figure 5.9 Connecting the RTE-V850E2/MN4-EB-S

(2) Installing the host driver

It is necessary to install a driver to connect any device to the host machine using a USB B receptacle.

The driver to be used for connection with a USB B receptacle is the mass storage class host driver which comes standard with Windows. See section 5.8, Operation Check, for details.

5.3 Debugging in the CubeSuite Environment

This section explains the procedure to debug an application program that is developed in the workspace introduced in section 5.2, Setting up the CubeSuite Environment.

5.3.1 Generating a Load Module

To write a program into the target device, it is necessary to compile its source file that is coded in C or assembly language into a load module.
In CubeSuite, a load module is generated by choosing “Build Project” from the “Build” menu.

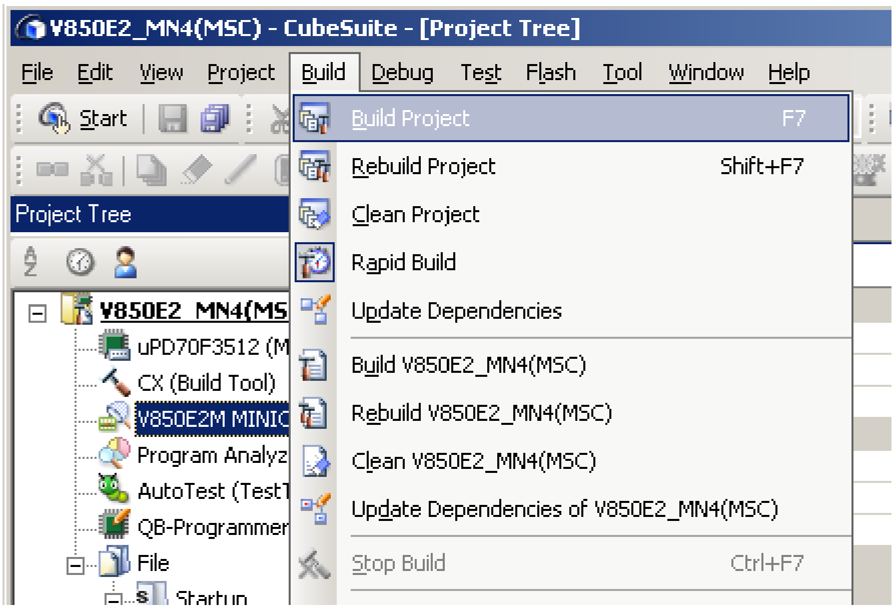


Figure 5.10 Selecting a Build Project

5.3.2 Loading and Executing

You write (load) the generated load module into the target for execution.

(1) Writing a load module

Shown below is the procedure to write a load module into the RTE-V850E2/MN4-EB-S via CubeSuite.

<1> Choose “Download” from the “Debug” menu and start the debugger.

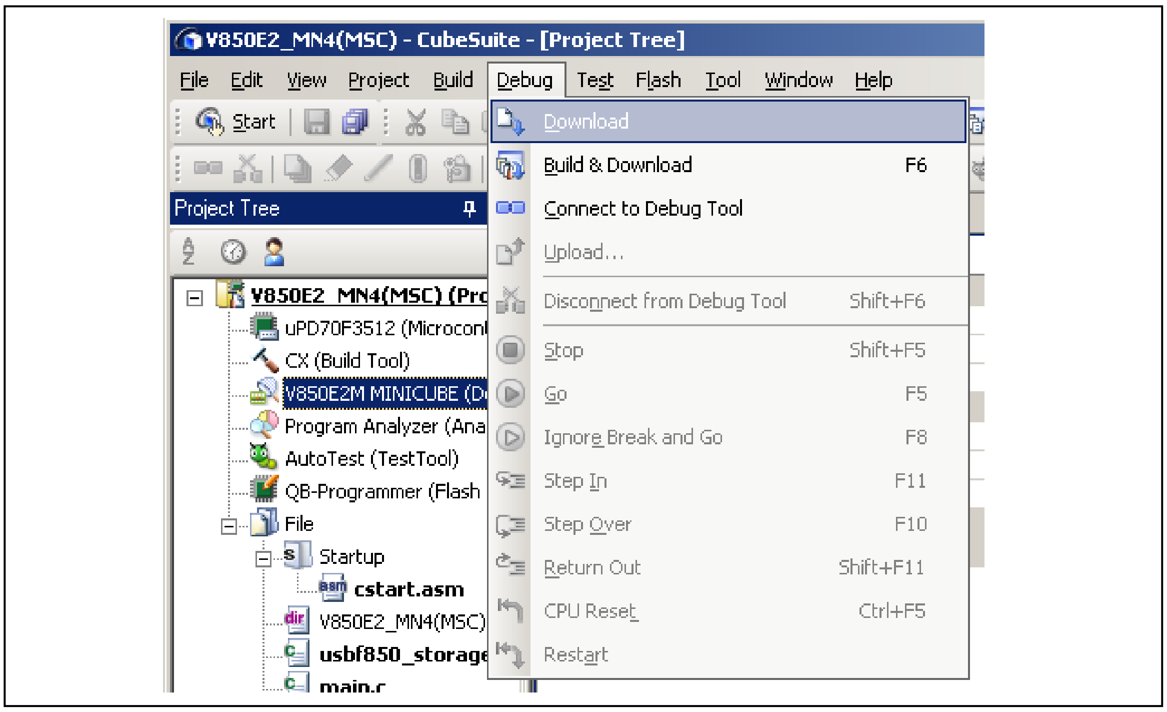
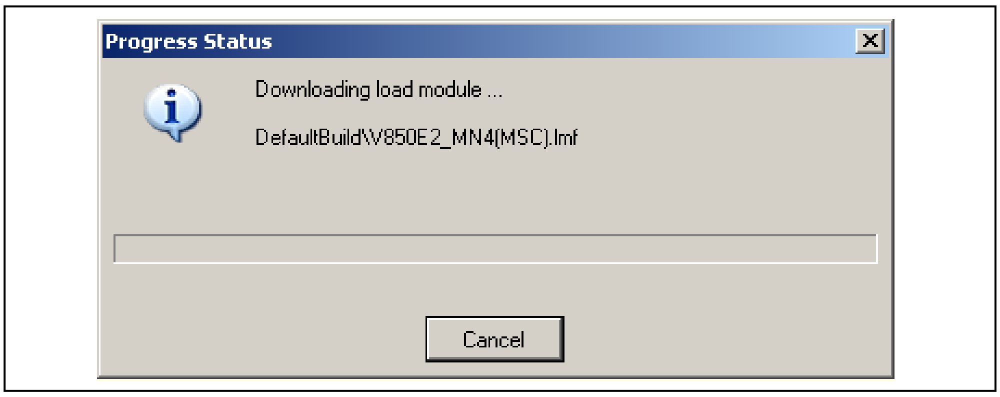


Figure 5.11 Choosing Download into Debugging Tool

<2> The downloading of the load module is started via the debugging tool.


Progress Status

Downloading load module ...

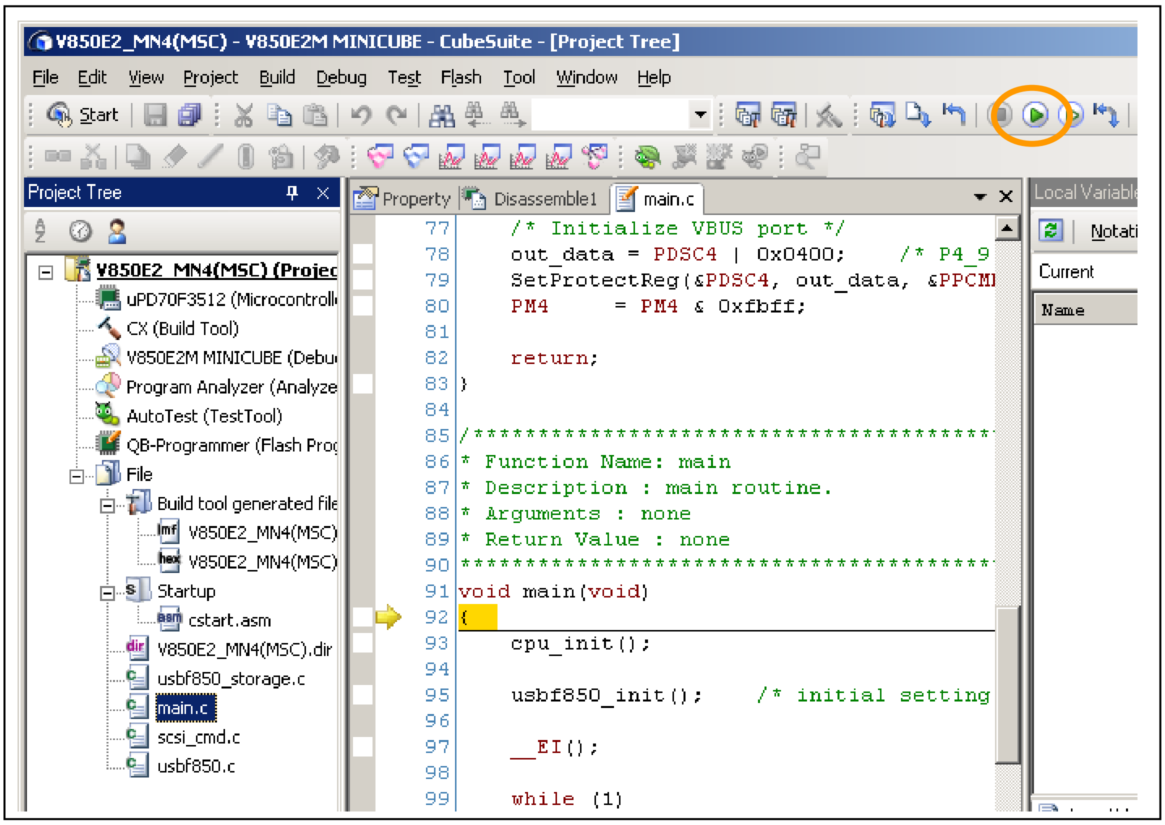
DefaultBuild\V850E2_MN4(MSC).lmf

Cancel

Figure 5.12 Executing the Download

(2) Running the program

Press the CubeSuite’s button or choose “Go” from the “Debug” menu.



```
    /* Initialize VBUS port */
    out_data = PDSC4 | 0x0400;     /* P4_9
    SetProtectReg(&PDSC4, out_data, &PPCMI
    PM4      = PM4 & 0xfbff;

    return;
}

/****************************************
* Function Name: main
* Description : main routine.
* Arguments : none
* Return Value : none
****************************************
void main(void)
{
    cpu_init();

    usbf850_init();     /* initial setting

    __EI();

    while (1)
```

Figure 5.13 Running the Program

5.4 Setting up a Multi Environment

This section explains the preparatory steps that are required to develop or debug using Multi which is introduced in section 5.1, Development Environment.

5.4.1 Setting up the Host Environment

You create a dedicated workspace on the host machine.

(1) Installing the Multi integrated development tools

Install Multi. Refer to the GHS user's manual for details.

(2) Expanding driver and other files

Store a set of distribution sample driver files in an arbitrary directory without modifying their folder structure.

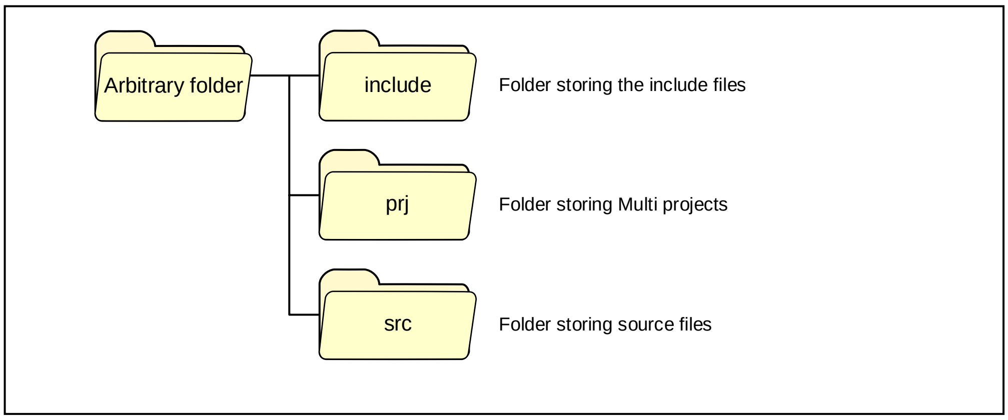

Arbitrary folder
include
Folder storing the include files
prj
Folder storing Multi projects
src
Folder storing source files

Figure 5.14 Folder Configuration for the Sample Driver (Multi Version)

(3) Installing device files

Copy the V850E2/MN4 device files for Multi in the folder where Multi is installed.

Example: C:\Green\V800.V517D\devicefile

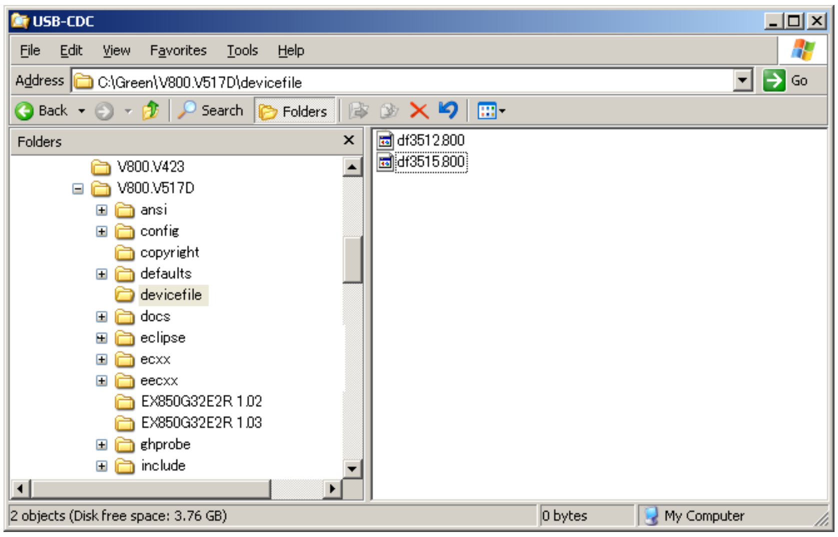


Figure 5.15 Example of Destination Folder for Storing the Device Files

(4) Starting Multi

Select and start Multi Project File in “V850E2_MN4(MSC)_GHS.gpj” which is included in the sample driver package from the Explorer.

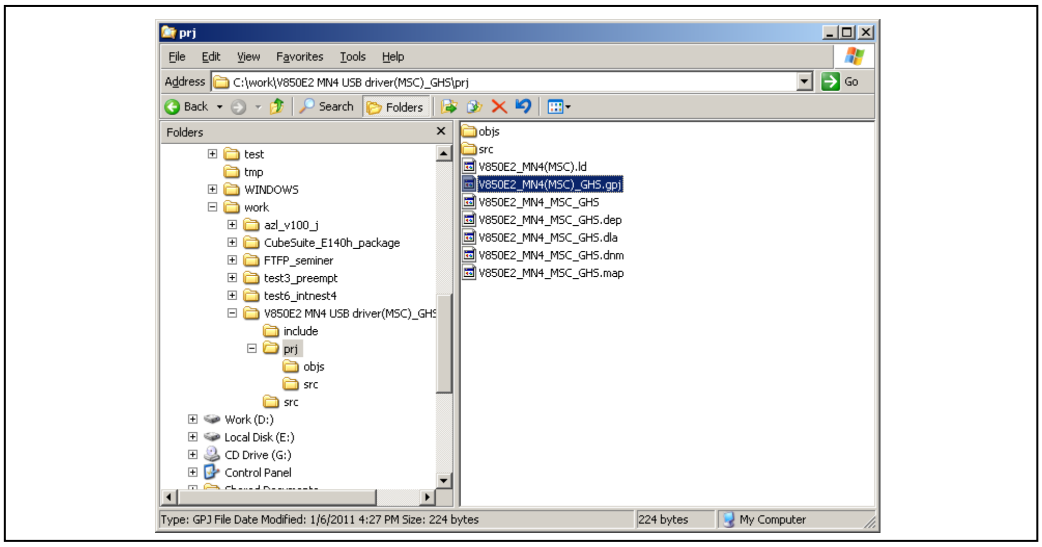


Figure 5.16 Selecting the Multi Project File

(5) Setting up the debugging tool

Given below is the procedure to use MINICUBE as the debugging tool.

<1> Choose “Connect” from the Multi’s “Connect” menu to open the Connection Chooser.

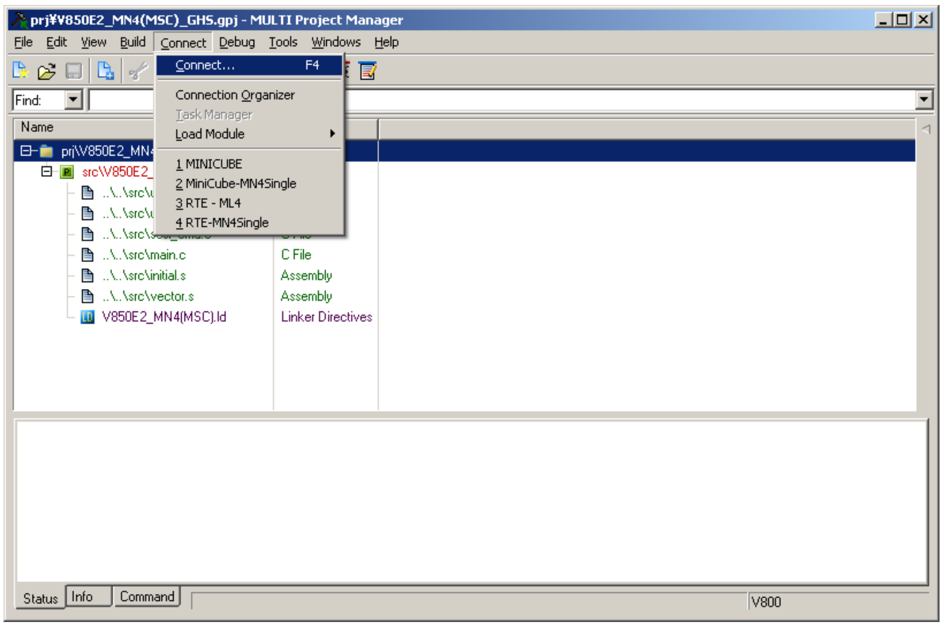
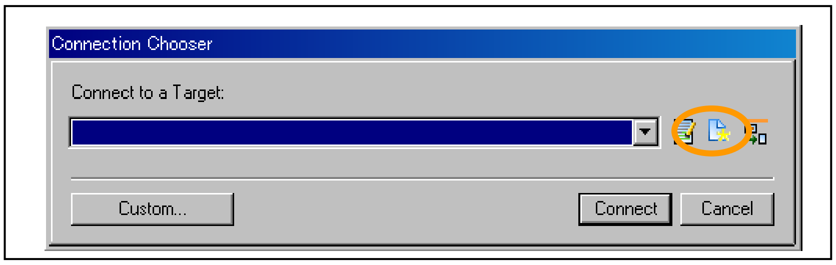


Figure 5.17 Starting the Connection Chooser

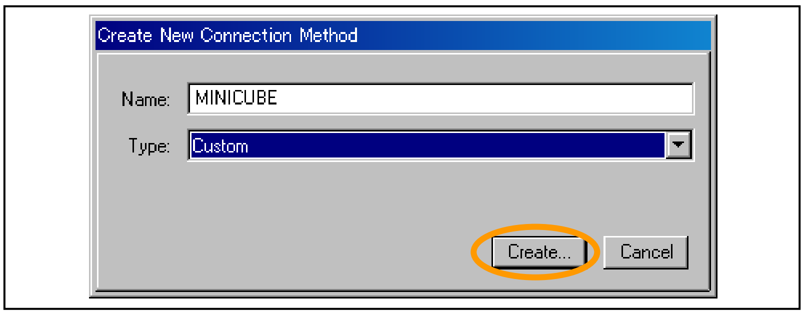
<2> Select the “Create New Connection Method” icon from the “Connection Chooser” dialog box.


Connection Chooser

Connect to a Target:

Figure 5.18 Selecting the Create New Connection Method

<3> In the Create New Connection Method dialog box, enter an arbitrary name in the Name textbox and select “Custom” in the Type combobox, then click the “Create…” button to create MINICUBE connection settings. In the example shown here, the name is set to “MINICUBE” in the Name textbox.


Create New Connection Method

Name: MINICUBE

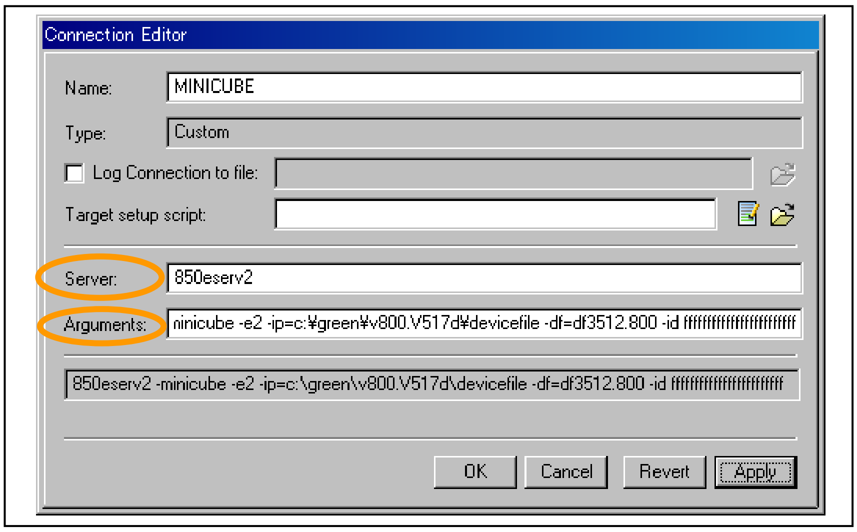
Type: Custom

Figure 5.19 Creating the Create New Connection Method

<4> Connection Editor will then start. Fill the “Server” and “Arguments” fields as shown below and click the OK button.

Server:	850eserv2
Arguments:	-minicube -e2 -ip=c:\green\v800.V517d\devicefile -df=df3512.800 -id ffffffffffffffffffffffff (Note 10)

(Note 10) 24 occurrences of "f”


Connection Editor

Name: MINICUBE

Type: Custom

Log Connection to file:

Target setup script:

Server: 850eserv2

Arguments: ninicube -e2 -ip=c:¥green¥v800.V517d¥devicefile -df=df3512.800 -id ffffffffffffffffffff

850eserv2 -minicube -e2 -ip=c:\green\v800.V517d\devicefile -df=df3512.800 -id ffffffffffffffffffff

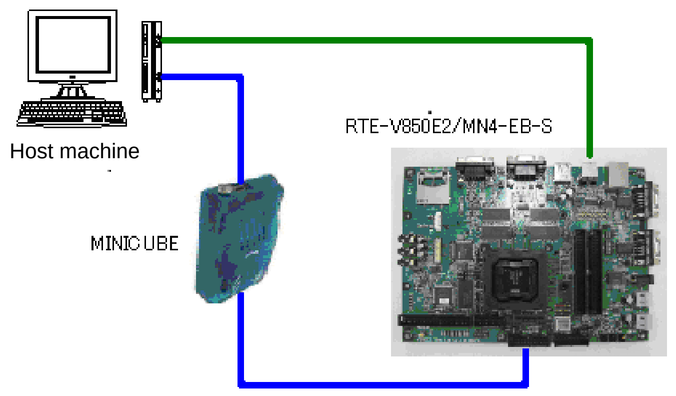
Figure 5.20 Configuring Connection Editor

5.4.2 Setting up the Target Environment

You connect the target device to be used for debugging to the host machine. The procedure is common to CubeSuite, Multi, and IAR Embedded Workbench.

(1) Connecting to the debugging port

Connect between the RTE-V850E2/MN4-EB-S and the host machine. Connect the RTE-V850E2/MN4-EB-S and the host machine via the MINICUBE for debugging. In addition, connect between the USB B type receptacle of the RTE-V850E2/MN4-EB-S and the USB receptacle of the host machine for the MSC.



Remarks: See section 7, Outline of the Starter Kit, for the physical appearance and port configuration of the RTE-V850E2/MN4-EB-S.

Figure 5.21 Connecting the RTE-V850E2/MN4-EB-S

(2) Installing the host driver

It is necessary to install a driver to connect any device to the host machine using a USB B receptacle.

The driver to be used for connection with a USB B receptacle is the mass storage class host driver which comes standard with Windows. See section 5.8, Operation Check, for details.

5.5 Debugging in the Multi Environment

This section explains the procedure to debug an application program that is developed in the workspace that is introduced in section 5.4, Setting up the Multi Environment.

5.5.1 Generating a Load Module

To write a program into the target device, it is necessary to compile its source file that is coded in C or assembly language into a load module.

In Multi, a load module is generated by choosing “Build Top Project V850E2_MN4(MSC)_GHS.gpj” from the “Build” menu.

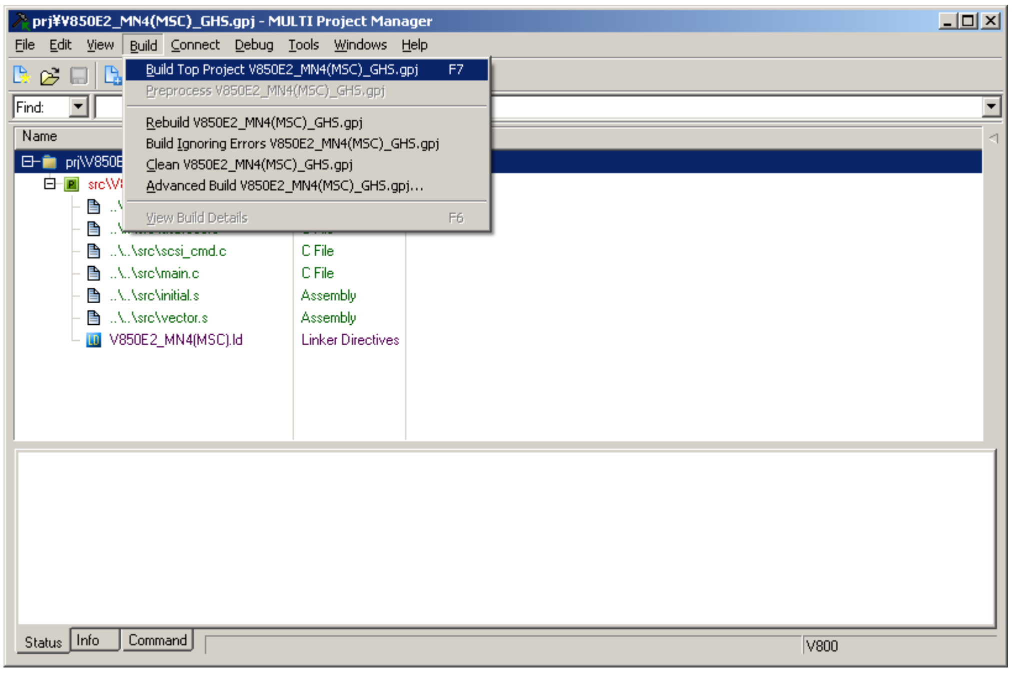

prj¥V850E2_MN4(MSC)_GHS.gpj - MULTI Project Manager
File Edit View Build Connect Debug Tools Windows Help
Build Top Project V850E2_MN4(MSC)_GHS.gpj F7
Preprocess V850E2_MN4(MSC)_GHS.gpj
Rebuild V850E2_MN4(MSC)_GHS.gpj
Build Ignoring Errors V850E2_MN4(MSC)_GHS.gpj
Clean V850E2_MN4(MSC)_GHS.gpj
Advanced Build V850E2_MN4(MSC)_GHS.gpj...
View Build Details F6
Find:
Name
..\..\src\scsi_cmd.c C File
..\..\src\main.c C File
..\..\src\initial.s Assembly
..\..\src\vector.s Assembly
V850E2_MN4(MSC).ld Linker Directives
Status Info Command
V800

Figure 5.22 Choosing Build

5.5.2 Loading and Executing

You program (load) the generated load module into the target for execution.

(1) Programming the load module

Shown below is the procedure to program a load module into the RTE-V850E2/MN4-EB-S via Multi.

<1> Choose “Connect” from the Multi’s “Connect” menu to open the Connection Chooser.

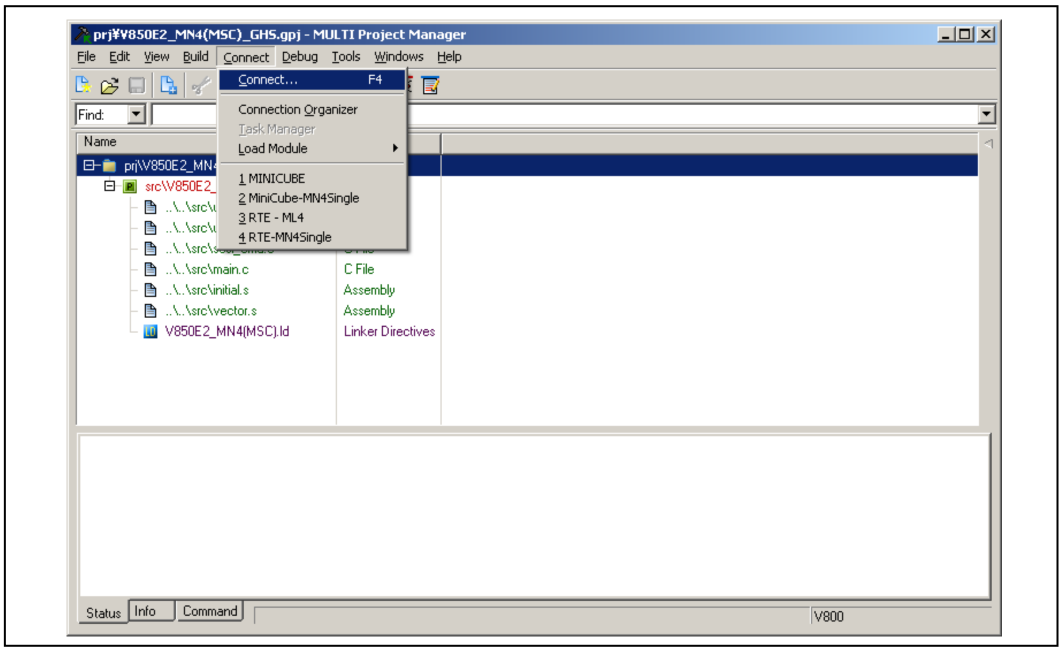


Figure 5.23 Starting the Connection Chooser

<2> From the Connection Chooser, select the MINICUBE connection settings you created according to the procedure explained in section 5.4.1, Setting up the Host Environment, and click the “Connect” button.

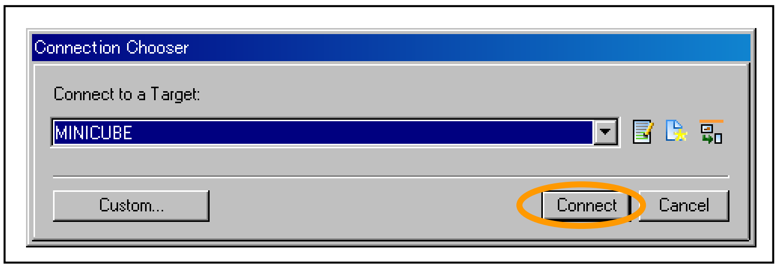
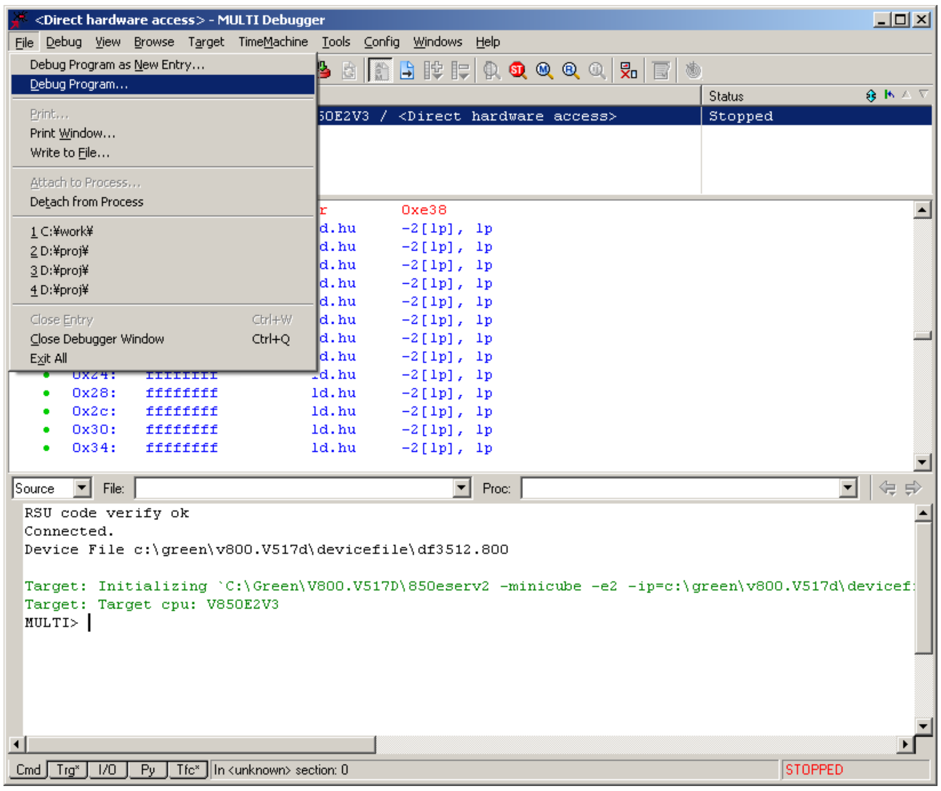


Figure 5.24 Selecting the MINICUBE Connection Settings

<3> MULTI Debugger will then start. Choose “Debug Program” from the “File” menu and download the load module.



```
50E2V3 / <Direct hardware access>

r        0xe38
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
d.hu     -2[lp], lp
0x24:  ffffffff     ld.hu     -2[lp], lp
0x28:  ffffffff     ld.hu     -2[lp], lp
0x2c:  ffffffff     ld.hu     -2[lp], lp
0x30:  ffffffff     ld.hu     -2[lp], lp
0x34:  ffffffff     ld.hu     -2[lp], lp
```

```
RSU code verify ok
Connected.
Device File c:\green\v800.V517d\devicefile\df3512.800

Target: Initializing `C:\Green\V800.V517D\850eserv2 -minicube -e2 -ip=c:\green\v800.V517d\devicef
Target: Target cpu: V850E2V3
MULTI>
```

Figure 5.25 Choosing a MULTI Debugger Menu

The load module is generated in the “prj” folder under the name of “V850E2_MN4_MSC_GHS.” Select it and click the “Open” button.

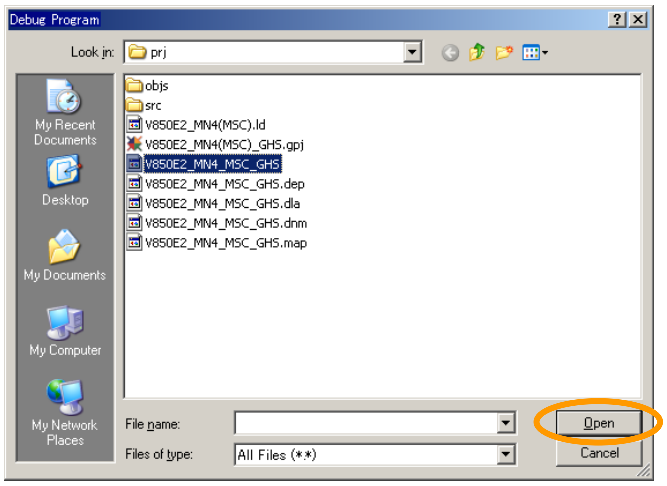
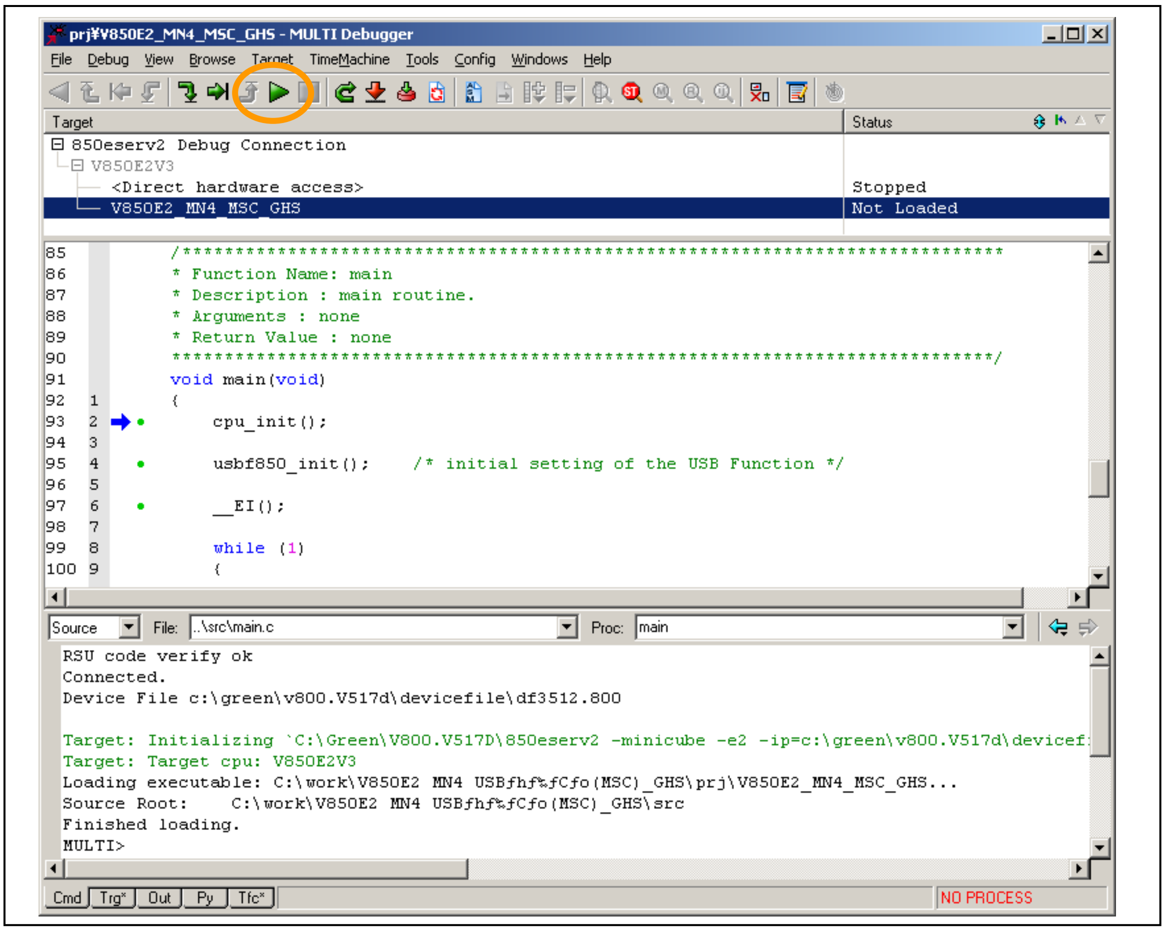


Figure 5.26 Selecting the Load Module

(2) Running the program

Press the MULTI Debugger's button or choose "Go on Selected Items" from the "Debug" menu.


prj¥V850E2_MN4_MSC_GHS - MULTI Debugger
File Debug View Browse Target TimeMachine Tools Config Windows Help
Target
Status
Source
File: ..\src\main.c
Proc: main
Cmd Trg* Out Py Tfc*
NO PROCESS

```
⊟ 850eserv2 Debug Connection
  ⊟ V850E2V3
     <Direct hardware access>                 Stopped
     V850E2_MN4_MSC_GHS                       Not Loaded
```

```
/******************************************************************************
 * Function Name: main
 * Description : main routine.
 * Arguments : none
 * Return Value : none
 ******************************************************************************/
void main(void)
{
    cpu_init();

    usbf850_init();    /* initial setting of the USB Function */

    __EI();

    while (1)
    {
```

```
RSU code verify ok
Connected.
Device File c:\green\v800.V517d\devicefile\df3512.800

Target: Initializing `C:\Green\V800.V517D\850eserv2 -minicube -e2 -ip=c:\green\v800.v517d\devicef
Target: Target cpu: V850E2V3
Loading executable: C:\work\V850E2 MN4 USBƒhƒ‰ƒCƒo(MSC)_GHS\prj\V850E2_MN4_MSC_GHS...
Source Root:    C:\work\V850E2 MN4 USBƒhƒ‰ƒCƒo(MSC)_GHS\src
Finished loading.
MULTI>
```

Figure 5.27 Running the Program

5.6 Setting up IAR Embedded Workbench Environment

This section explains the preparatory steps that are required to develop or debug using IAR Embedded Workbench which is introduced in section 5.1, Development.

5.6.1 Setting up the Host Environment

You create a dedicated workspace on the host machine.

(1) Installing the IAR Embedded Workbench integrated development tools

Install the IAR Embedded Workbench. Refer to the IAR Embedded Workbench user’s manual for details.

(2) Expanding driver and other files

Store a set of distribution sample driver files in an arbitrary directory without modifying their folder structure.

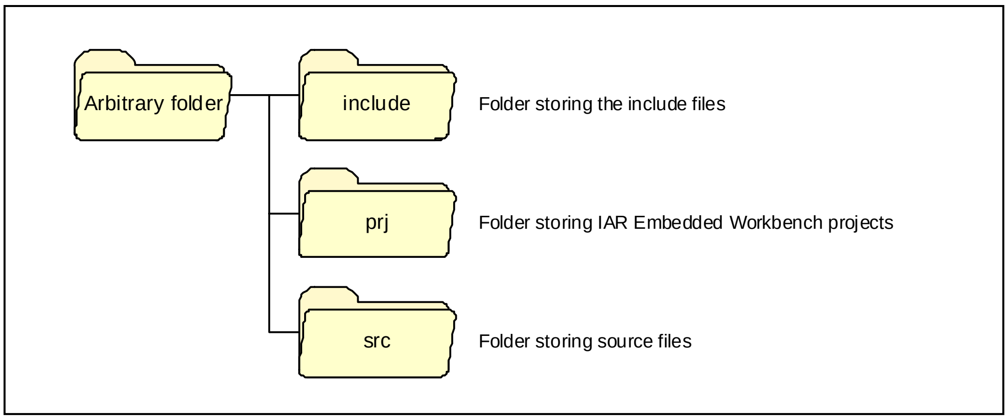
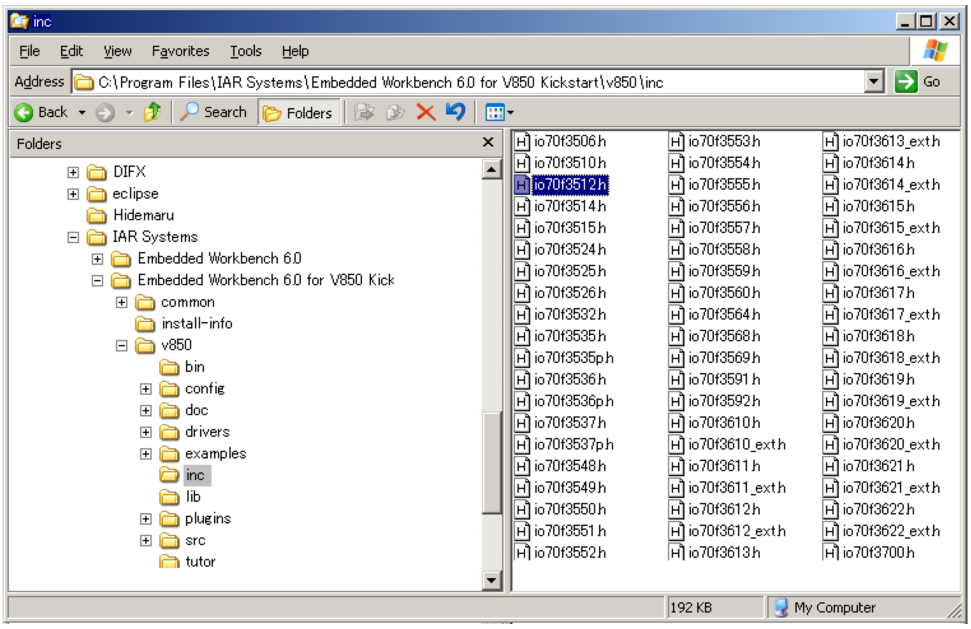

Arbitrary folder
include
Folder storing the include files
prj
Folder storing IAR Embedded Workbench projects
src
Folder storing source files

Figure 5.28 Folder Configuration for the Sample Driver (IAR Embedded Workbench Version)

(3) Installing device files

Copy the V850E2/MN4 device files for the IAR Embedded Workbench in the folder where the IAR Embedded Workbench is installed.

Example: C:\Program Files\IAR Systems\Embedded Workbench 6.0 for V850 kickstart\v850\inc


Address: C:\Program Files\IAR Systems\Embedded Workbench 6.0 for V850 Kickstart\v850\inc

```
Folders
  DIFX
  eclipse
  Hidemaru
  IAR Systems
    Embedded Workbench 6.0
    Embedded Workbench 6.0 for V850 Kick
      common
      install-info
      v850
        bin
        config
        doc
        drivers
        examples
        inc
        lib
        plugins
        src
        tutor
```

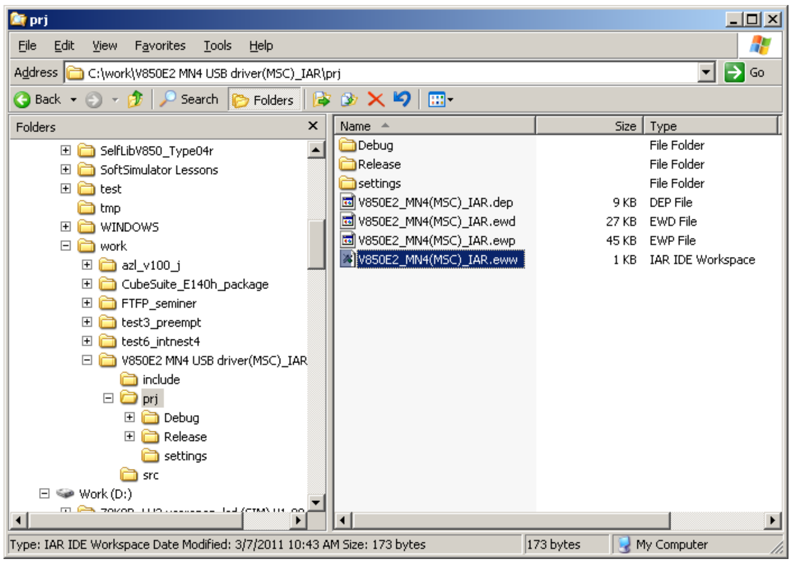
```
io70f3506.h    io70f3553.h       io70f3613_ext.h
io70f3510.h    io70f3554.h       io70f3614.h
io70f3512.h    io70f3555.h       io70f3614_ext.h
io70f3514.h    io70f3556.h       io70f3615.h
io70f3515.h    io70f3557.h       io70f3615_ext.h
io70f3524.h    io70f3558.h       io70f3616.h
io70f3525.h    io70f3559.h       io70f3616_ext.h
io70f3526.h    io70f3560.h       io70f3617.h
io70f3532.h    io70f3564.h       io70f3617_ext.h
io70f3535.h    io70f3568.h       io70f3618.h
io70f3535p.h   io70f3569.h       io70f3618_ext.h
io70f3536.h    io70f3591.h       io70f3619.h
io70f3536p.h   io70f3592.h       io70f3619_ext.h
io70f3537.h    io70f3610.h       io70f3620.h
io70f3537p.h   io70f3610_ext.h   io70f3620_ext.h
io70f3548.h    io70f3611.h       io70f3621.h
io70f3549.h    io70f3611_ext.h   io70f3621_ext.h
io70f3550.h    io70f3612.h       io70f3622.h
io70f3551.h    io70f3612_ext.h   io70f3622_ext.h
io70f3552.h    io70f3613.h       io70f3700.h
```

192 KB | My Computer

Figure 5.29 Example of Destination Folder for Storing the Device Files

(4) Starting IAR Embedded Workbench

Select an IAR IDE Workspace in “V850E2_MN4(MSC)_IAR.eww” which is included in the sample driver package from the Explorer. And start the IAR Embedded Workbench.


Address: C:\work\V850E2 MN4 USB driver(MSC)_IAR\prj

```
Folders
  SelfLibV850_Type04r
  SoftSimulator Lessons
  test
  tmp
  WINDOWS
  work
    azl_v100_j
    CubeSuite_E140h_package
    FTFP_seminer
    test3_preempt
    test6_intnest4
    V850E2 MN4 USB driver(MSC)_IAR
      include
      prj
        Debug
        Release
        settings
      src
Work (D:)
```

Name	Size	Type
Debug		File Folder
Release		File Folder
settings		File Folder
V850E2_MN4(MSC)_IAR.dep	9 KB	DEP File
V850E2_MN4(MSC)_IAR.ewd	27 KB	EWD File
V850E2_MN4(MSC)_IAR.ewp	45 KB	EWP File
V850E2_MN4(MSC)_IAR.eww	1 KB	IAR IDE Workspace

Type: IAR IDE Workspace Date Modified: 3/7/2011 10:43 AM Size: 173 bytes | 173 bytes | My Computer

Figure 5.30 Selecting IAR IDE Workspace

(5) Setting up the debugging tool

Given below is the procedure to use MINICUBE as the debugging tool.

<1> Select “Options” of the “V850E2_MN4(MSC)_IAR- Release (or Debug)” properties item, and open the Connection Chooser.

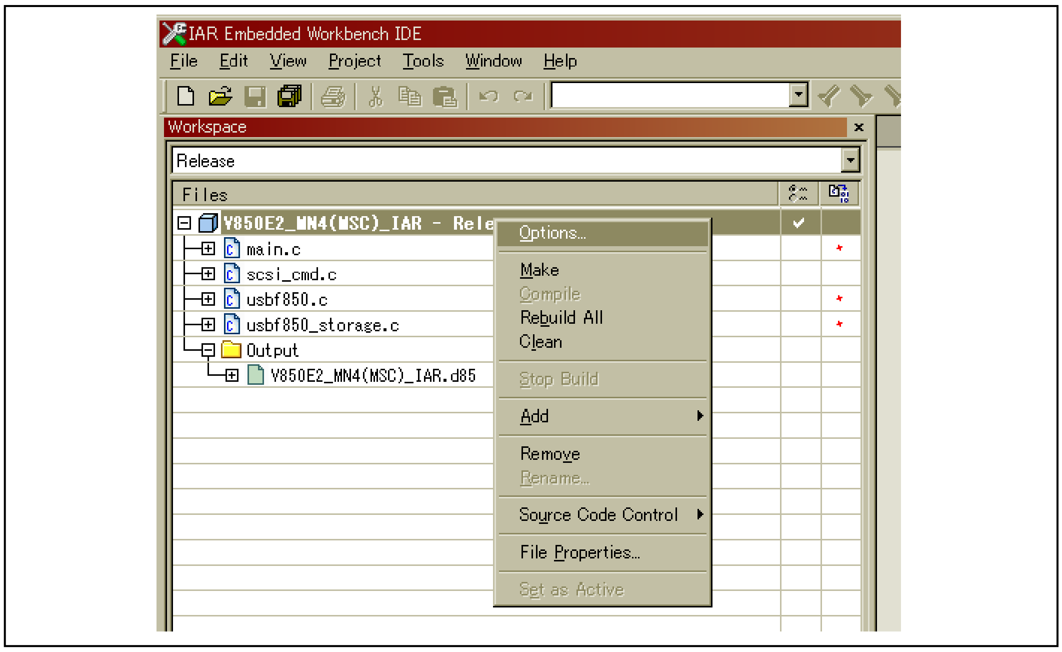


Figure 5.31 Selecting Options

<2> Select “Debugger” from the “Category” in the “Options for node “V850E2_MN4(MSC)_IAR”” dialog box.

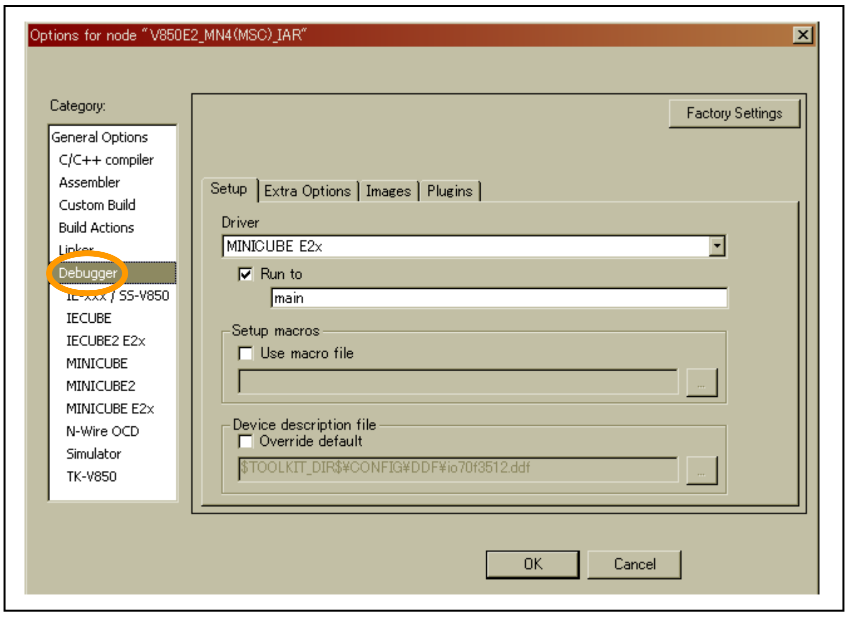
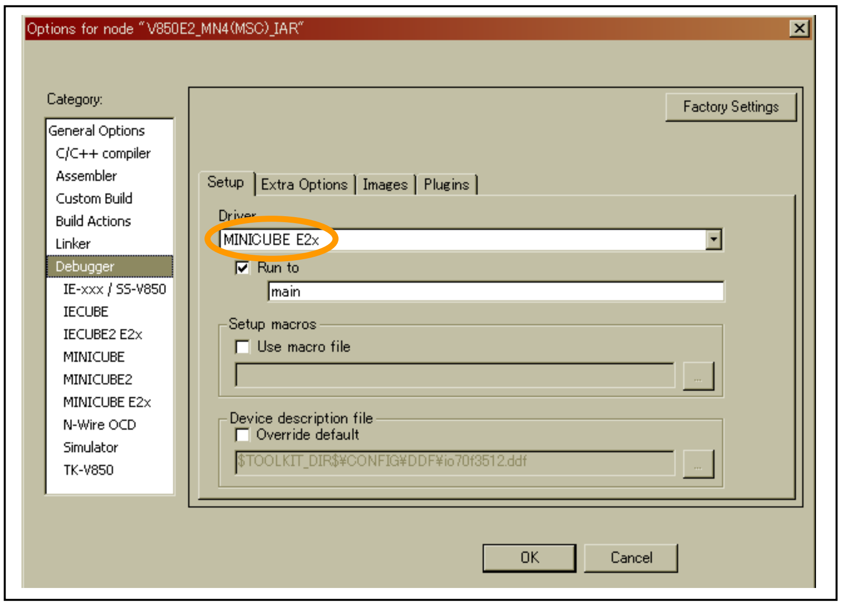


Figure 5.32 Selecting Debugger

<3> Select “MINICUBE E2x” from Driver in the “Setup” tab and press the “OK” button.



Driver: MINICUBE E2x

☑ Run to: main

Setup macros

☐ Use macro file

Device description file

☐ Override default

$TOOLKIT_DIR$¥CONFIG¥DDF¥io70f3512.ddf

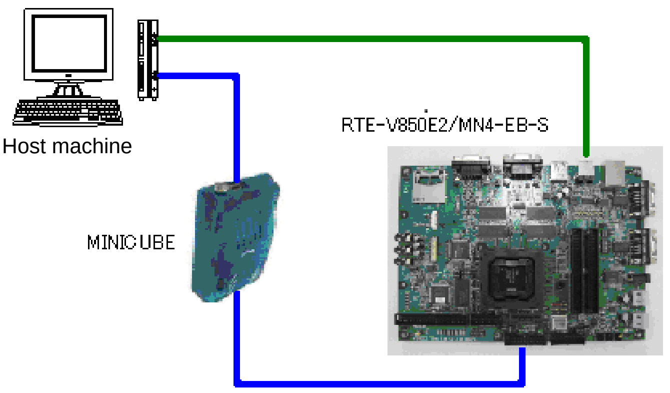
Figure 5.33 Selecting Debugger

5.6.2 Setting up the Target Environment

You connect the target device to be used for debugging to the host machine. The procedure is common to CubeSuite, Multi, and IAR Embedded Workbench.

(1) Connecting to the debugging port

Connect between the RTE-V850E2/MN4-EB-S and the host machine. Connect the RTE-V850E2/MN4-EB-S and the host machine via the MINICUBE for debugging. In addition, connect between the USB B type receptacle of the RTE-V850E2/MN4-EB-S and the USB receptacle of the host machine for the MSC.



Remarks: See section 7, Outline of the Starter Kit, for the physical appearance and port Configuration of the RTE-V850E2/MN4-EB-S.

Figure 5.34 Connecting the RTE-V850E2/MN4-EB-S

(2) Installing the host driver

It is necessary to install a driver to connect any device to the host machine using a USB B receptacle.
The driver to be used for connection with a USB B receptacle is the mass storage class host driver which comes standard with Windows. See section 5.8, Operation Check, for details.

5.7 Debugging in the IAR Embedded Workbench Environment

This section explains the procedure to debug an application program that is developed in the workspace that is introduced in section 5.6, Setting up IAR Embedded Workbench Environment.

5.7.1 Generating a Load Module

To write a program into the target device, it is necessary to compile its source file that is coded in C or assembly language into a load module.

In the IAR Embedded Workbench, a load module is generated by choosing “Rebuild All” from the “Project” menu.

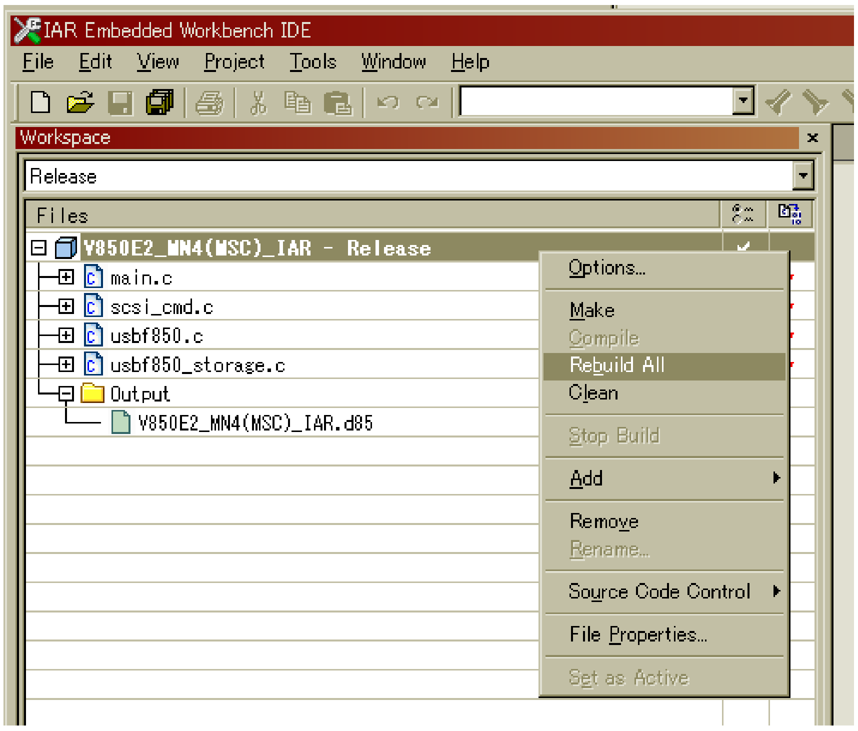


Figure 5.35 Choosing Rebuild All

5.7.2 Loading and Executing

You program (load) the generated load module into the target for execution.

(1) Programming the load module

Shown below is the procedure to program a load module into the RTE-V850E2/MN4-EB-S via IAR Embedded Workbench.

<1> Select "Download and Debug" from the "Project" menu in the IAR Embedded Workbench. And load the generated load module into the target.

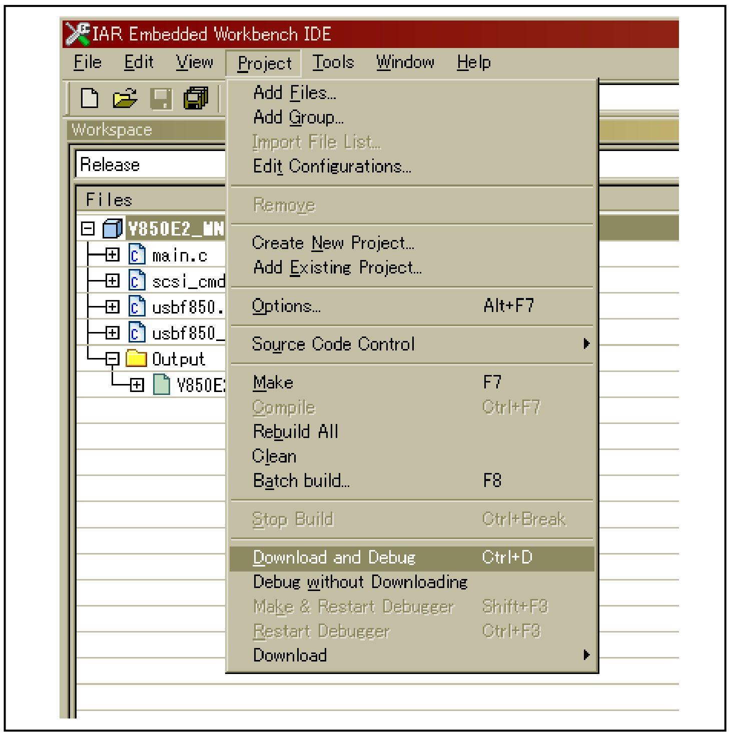


Figure 5.36 Starting Debugger

<2> Download of load module starts via the debugging tools.

(2) Running the program

Press the IAR Embedded Workbench button or choose “Go” from the “Debug” menu.

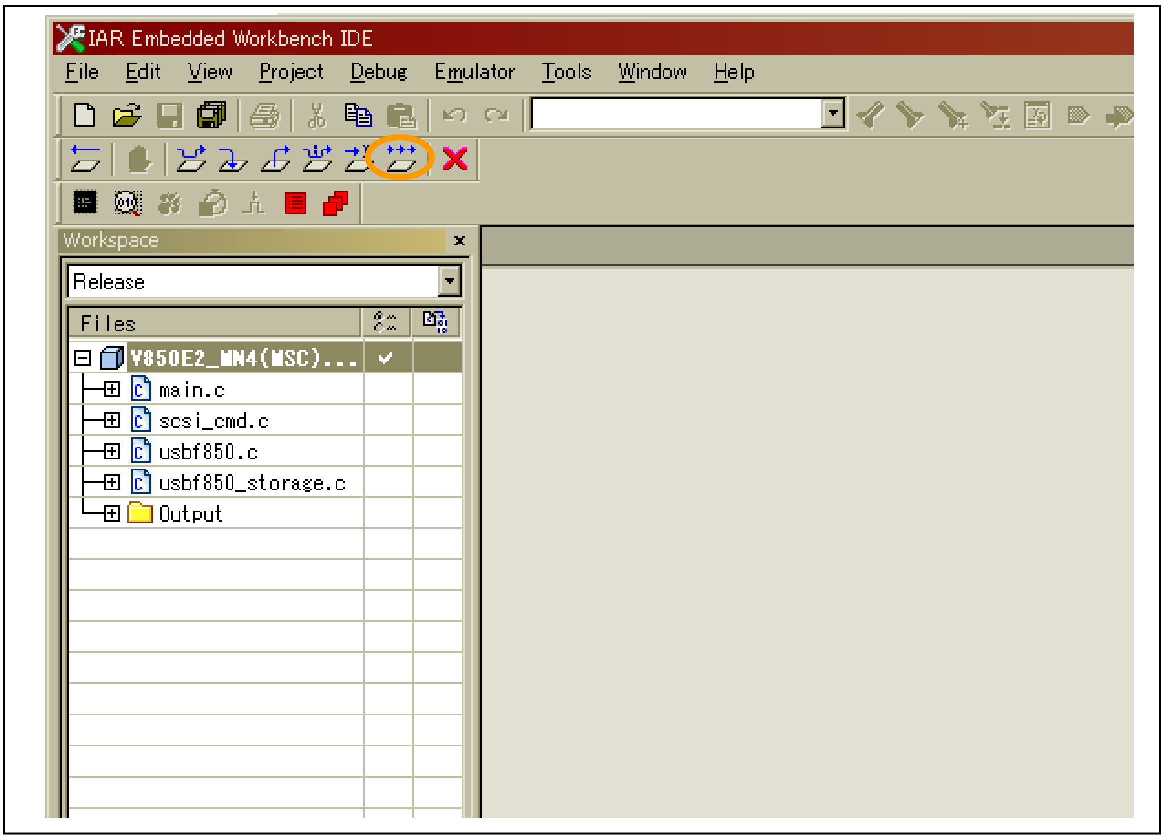


Figure 5.37 Running the Program

5.8 Operation Check

This section describes the procedures for verifying the results of executing the sample driver program in the CubeSuite, Multi, or IAR Embedded Workbench environment.

(1) Connecting to the USB B receptacle

Connect between the USB B receptacle of the RTE-V850E2/MN4-BE-S and a USB port of the host machine with a USB cable.

(2) Installing the host driver

The driver to be used for connection with the USB B receptacle is the host driver for the mass storage class which comes standard with Windows. The driver will automatically be installed when you connect to the host machine via the USB while the sample driver is running.

(3) Checking the connection status of the USB devices

Open the Windows Device Manager. Expand the "Universal Serial Bus controllers" tree and make sure that "USB Mass Storage Device" is shown. In addition, expand the "Disk drives" tree and make sure that "Renesas StorageFncDriver USB Device" is shown.

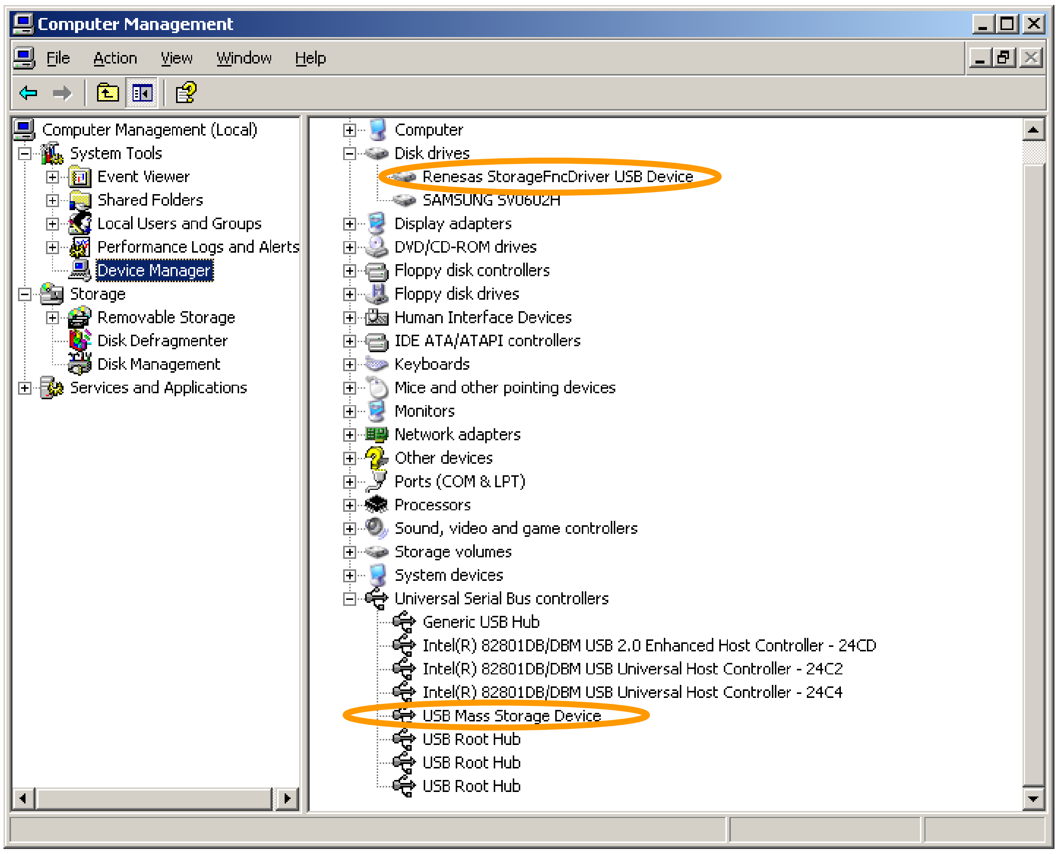


Figure 5.38 Checking with the Device Manager

(4) Formatting a removable disk

Open the Windows's "My Computer" and "Removable Disk" will appear.

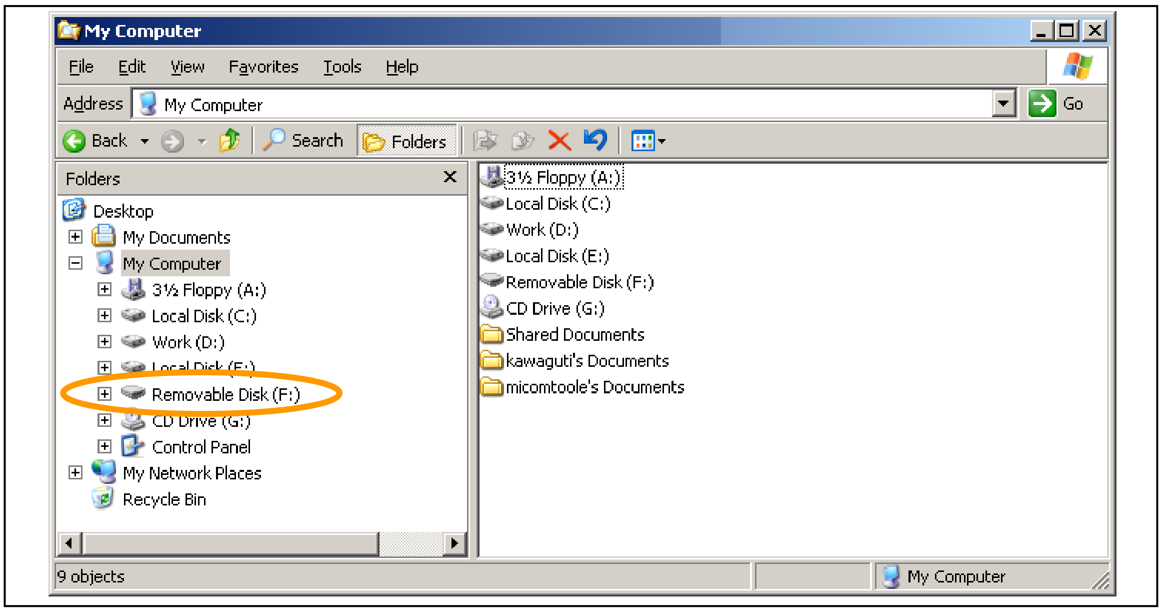
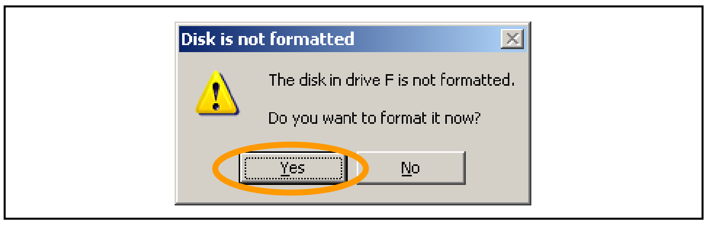


Figure 5.39 Checking the Removable Disk

Remarks: "(F:)" in this screen example is the drive letter that is automatically assigned by the OS. This drive letter varies with the host machine's configuration.

<1> Click "Removable Disk" under "My Computer," and the message "Disk is not formatted." will appear. Click the "Yes" button.


Disk is not formatted

The disk in drive F is not formatted.

Do you want to format it now?

Figure 5.40 Formatting Confirmation Dialog Box

<2> The “Format Removable Disk” dialog box will then appear. Select necessary items and click the “Start” button.

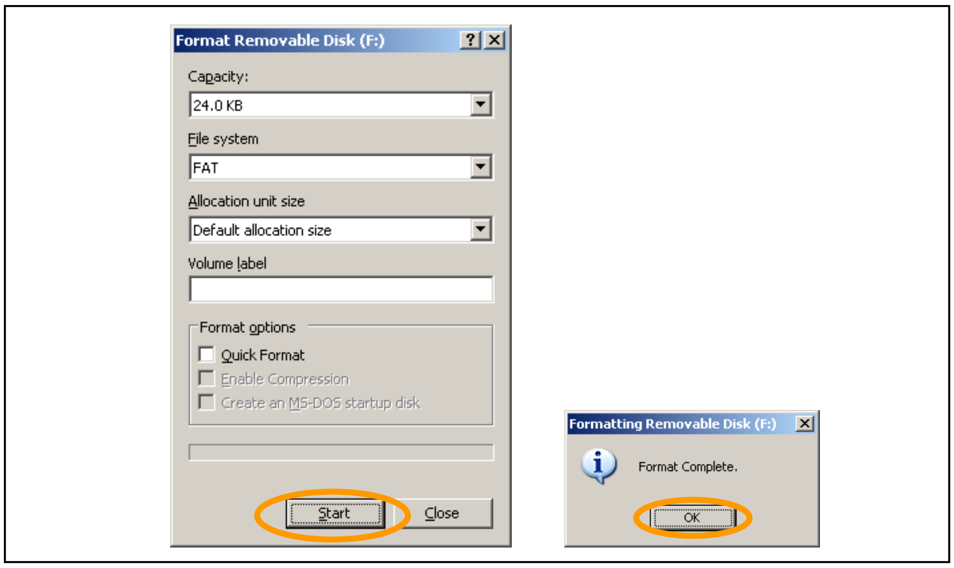


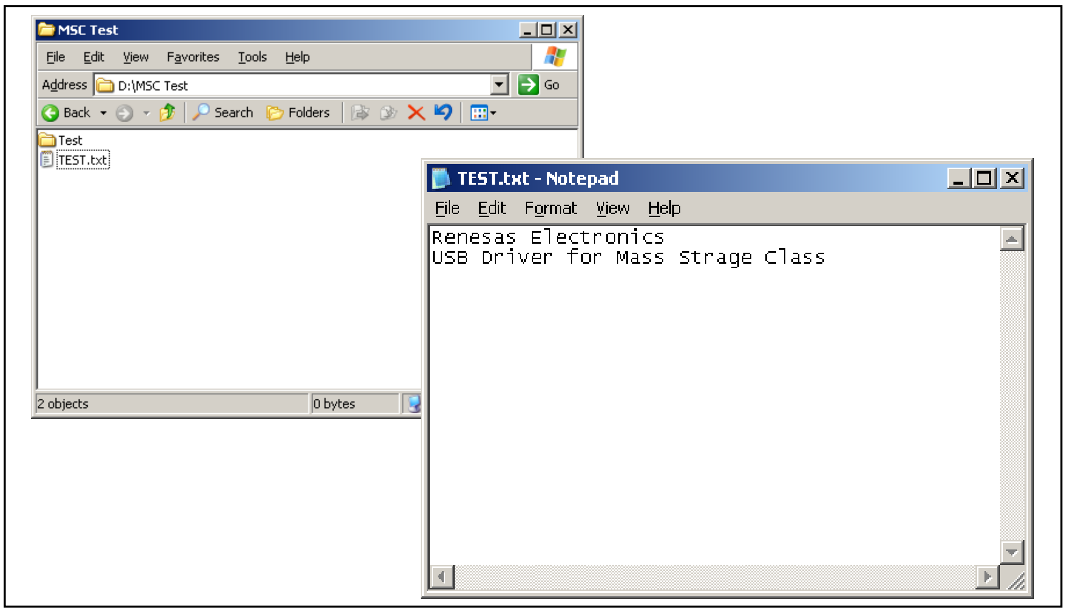
Figure 5.41 Format Menu and Format Complete Dialog Box

<3> A message will appear when formatting is completed. Click the “OK” button.

(5) Storing files and unloading

Check for normal write and read of a file to and from the removable disk.

<1> Prepare a file named TEST.txt and the Test folder on a local disk.



```
Renesas Electronics
USB Driver for Mass Strage Class
```

Figure 5.42 Test Folder and Test Data File

<2> Open the Removal Disk in the My Computer and copy the TEST.txt file from the local disk to the Removable Disk.

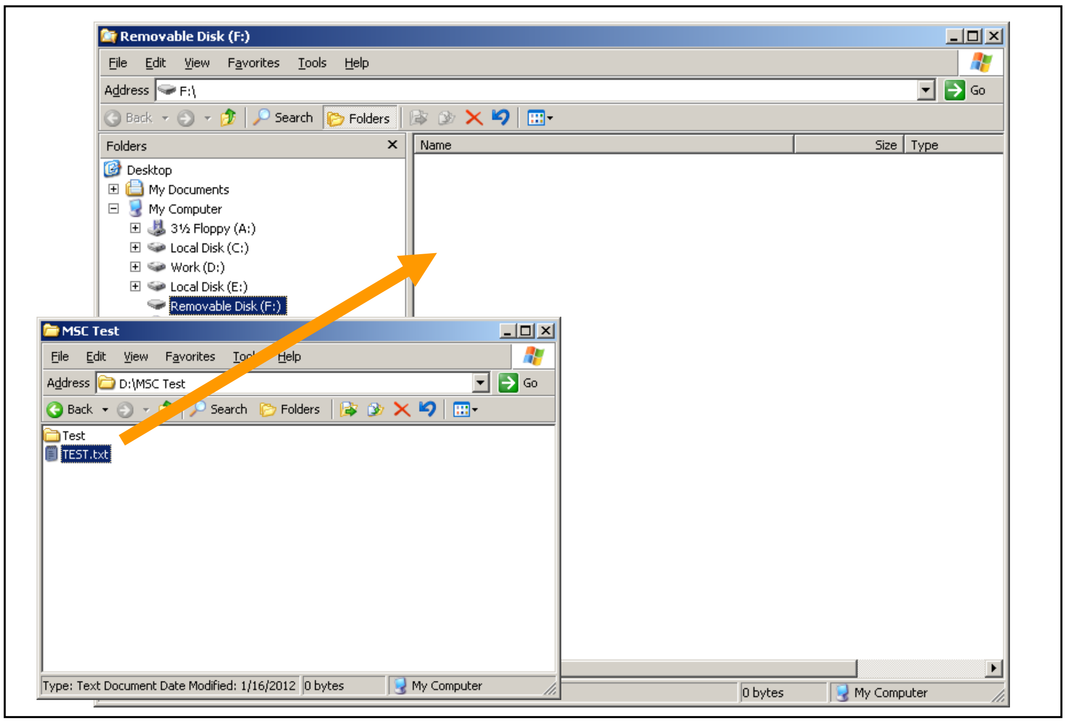


Figure 5.43 Copying the Test Data File

<3> Open the Test folder in the local disk and copy the TEST.txt file from the Removable Disk to the Test folder.

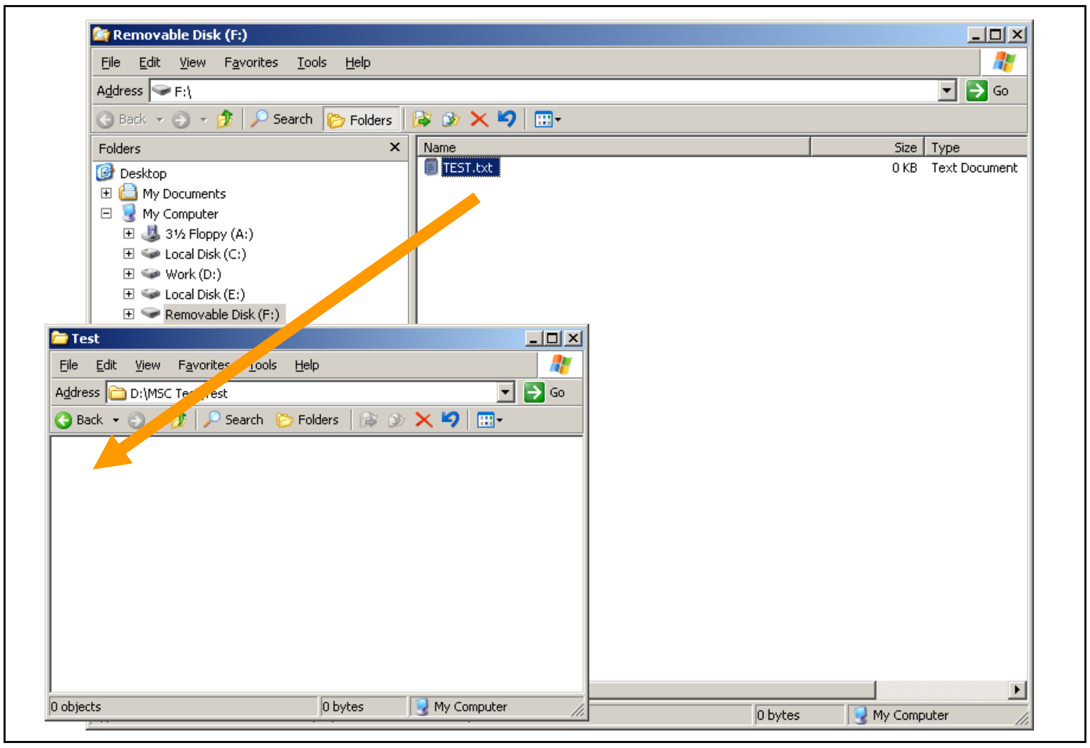
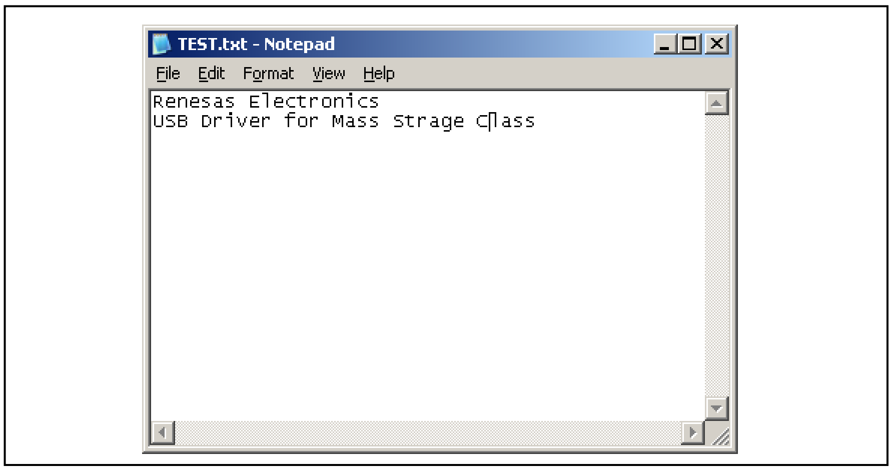


Figure 5.44 Copying back the Test Data File

<4> Open the TEST.txt file in the Test folder and make sure that its contents match those of the TEST.txt file on the local disk.



```
Renesas Electronics
USB Driver for Mass Strage Class
```

Figure 5.45 Checking the Test Data File

Remarks: An internal RAM area of 24 Kbytes is used as the data area. Consequently, the data you stored will be initialized when the device power is turned off or the Reset switch is pressed. Normal operation is not guaranteed if an attempt is made to write a file of 24 Kbytes or larger.

6. Sample Driver Application

This section contains the information you should be aware of when using the USB mass storage class (MSC) sample driver for the V850E2/MN4.

6.1 Overview

You create a driver that suits your system by customizing the sample driver.
The major sections you should rewrite as required are listed below.

- Sample application section in the file "main.c"
- Register settings in the file "usbf850.h"
- Contents of the descriptors in the file "usbf850_desc.h"
- SCSI command processing in the files "scsi_cmd.c" and "scsi.h"
- RAM disk size in the file "scsi.h"
- Vendor and product names in the file "scsi_cmd.c"

Note: See section 2.1.3, Sample Driver Configuration, for the file configuration of the sample driver.

6.2 Customization

This section describes the sections you should rewrite when using the sample driver.

6.2.1 Application Section

The main routine processing function (main) in the file "main.c" shows simple processing as an example of use of the sample driver. By coding the processing that is to be executed by the practical application in this section, you can make use of the existing initialization and interrupt processing without modification.

```
/*******************************************************************************
* Function Name: main
* Description : main routine.
* Arguments : none
* Return Value : none
*******************************************************************************/
void main(void)
{
    cpu_init();

    usbf850_init();    /* initial setting of the USB Function */

    __EI();

    while (1)
    {
        if (usbf850_rsuspd_flg == SUSPEND)
        {
            __DI();

            __halt();

            usbf850_rsuspd_flg = RESUME;

            __EI();
        }
    }
}
```

Figure 6.1 Coding the Main Routine

6.2.2 Register Settings

The registers that the sample driver uses (writes to) and their settings are defined in the file "usbf850.h." By rewriting these values in the file according to the actual application, you can configure the operation of the target device through the sample driver.

The GHS version of the sample driver comes with a separate file named "df3512_800.h" which contains the definitions of internal I/O registers of the V850E2/MN4.

The IAR Embedded Workbench version of the sample driver comes with a separate file named "io70f3512.h" which contains the definitions of internal I/O registers of the V850E2/MN4.

(1) File "usbf850.h"

Defines the settings of the USB function controller registers.

(2) File "df3512_800.h" (GHS version only)

Contains the definitions of the internal I/O registers of the V850E2/MN4.

(3) File "io70f3512.h" (IAR Embedded Workbench version only)

Contains the definitions of the internal I/O registers of the V850E2/MN4.

6.2.3 Contents of the Descriptors

The file "usbf850_desc.h" defines the data (see section 4.1.3, Descriptor Settings) that the sample driver registers in the USB function controller during initialization processing. By rewriting these values in the file according to the actual application, you can set up the attributes and other information of the target device through the sample driver.

You can register arbitrary information in string descriptors. For the sample driver, a serial number is defined; you should rewrite it accordingly.

```
  :
/* 0 : Language Code*/
DSTR(LangString, 2, (0x09,0x04));
/* 1 : Serial Number*/
USTR(SerialString, 12, ('0','2','0','0','0','8','0','6','5','0','1','0'));
   :
```

Figure 6.2 String Descriptor Settings in "usbf850_desc.h"

6.2.4 Making Changes to the SCSI Command Processing

The SCSI command processing is coded in the files "scsi_cmd.c" and "scsi.h." Make the changes shown below when adding a new SCSI command that is to be supported.

- Add the definition of the new processing function to the file "scsi_cmd.c."
- Add a case statement that calls the new function to the SCSI command execution function (scsi_command_to_ata) in the file "scsi_cmd.c".
- Add the declaration of the new function in the function declaration section of the file "scsi.h."

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

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

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

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

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

Figure 6.3 SCSI Command Execution Function (scsi_command_to_ata)

6.2.5 Changing the RAM Disk Size

The size of the RAM disk is defined in the file "scsi.h." The product of ALL_LOGICBLOCK (total number of blocks) and LOGICBLOCK_SIZE (block size) indicates the capacity of the RAM disk (this is set to 0x6000 (= 24 Kbytes) for the sample driver). Since disk space is also consumed by the FAT and other information, the size of disk space available for the PC is less than this set value.

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

#define MODE_SELECT_MIN_LEN      (4)    /*  4Byte */

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

Figure 6.4 Data Length Section of the "scsi.h" File

6.2.6 Vendor and Product Name Settings

You can modify the vendor and product names for any disk drive by editing the response values to the INQUIRE command defined in the file "scsi_cmd.c."

(1) INQUIRY_TABLE code

"INQUIRY_TABLE" in the file "scsi_cmd.c" contains the code that is shown in figure 6.5.

```
1 UINT8    INQUIRY_TABLE[INQUIRY_LENGTH] =
2 {
3    0x00,            /* Qualifier, device type code */
4    0x80,            /* RMB, device type modification child */
5    0x02,            /* ISO Version, ECMA Version, ANSI Version */
6    0x02,            /* AENC, TrmIOP, response data form */
7    0x1F,            /* addition data length */
8    0x00,0x00,0x00,    /* reserved */
9    'R','e','n','e','s','a','s',' ',                          /* vender ID */        <1>
10   'S','t','o','r','a','g','e','F','n','c','D','r','i','v','e','r',   /* product ID */   <2>
11   '0','.','0','1'                                          /* Product Revision */
12 };
```

Figure 6.5 "INQUIRY_TABLE" Code in the "scsi_cmd.c" File

Setting <1> on the 9th line defines the vendor name and setting <2> on the 10th line defines the product name. The vendor name may be a string of not longer than 8 bytes (eight 1-byte characters) and the product name a string of not longer than 16 bytes (sixteen 1-byte characters).

On data transmission, each character is converted to ASCII code. Consequently, any characters that cannot be decoded into ASCII code may not be displayed correctly.

(2) Displaying device names (list of devices)

The vendor and product names specified in "INQUIRY_TABLE" are displayed as the disk drive name for the Device Manager.

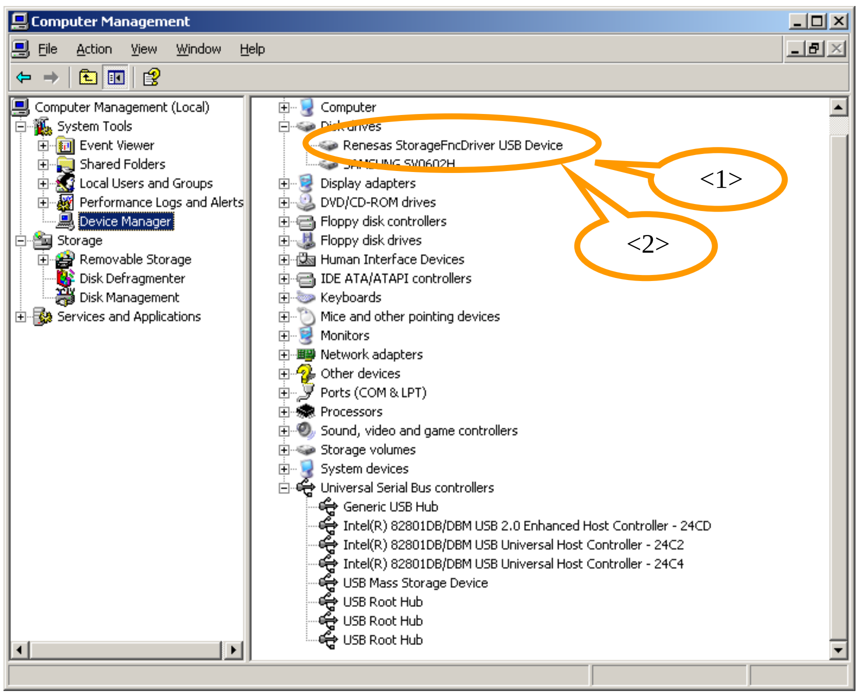


Figure 6.6 Device Manager Window View

6.3 Using Functions

Since processes that are frequently used or that have broad utility are implemented as defined functions, they simplify coding and contribute to reduction in code size. See section 4.3, Function Specifications, for details on the functions.

For example, CBW data receive processing in the file "usbf850_storage.c" is coded as shown in figure 6.7.

```
1  void usbf850_rx_cbw(void)
2  {
3     UINT8 * data = (UINT8 *)&CBW_TABLE;
4     INT8    len;
5
6     if (mass_storage_reset)
7     {
8        /*wait "Bulk-Only Mass Storage Reset" request*/
9        usbf850_cbw_error();
10       return;
11    }
12    len = USFA0BO1L;
13
14    if (len != 0x1F)
15    {
16       return; /*don't CBW*/
17    }
18
19    usbf850_data_receive(data, len, C_BKO1);
20
21    if (cbw_in_cbw)
22    {
23       /*CBW in CBW*/
24       USFA0FIC0 = (C_BKI1SC | C_BKI1CC); /*Clears EP1 buffers*/
25       cbw_in_cbw = USB_CBW_END;
26    }
27    cbw_in_cbw = USB_CBW_PROCESS;
28    usbf850_storage_cbwchk();
29
30    return;
31 }
```

Figure 6.7 CBW Data Receive Processing Function

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

The code on the 6th line monitors the flag (mass_storage_reset) that will be set by the sample driver. When this flag is set to "USB_MASS_RESET_WAIT (0x01)," it indicates that the sample driver is waiting for a mass storage reset request as the result of a command processing failure or another reason.

(2) Data receive processing

The code on the 19th line calls the function (usbf850_data_receive) that defines the processing of transferring data from an endpoint to its buffer. "C_BKO1" that represents the endpoint number is defined in the header file "usbf850.h."

7. Outline of the Starter Kit

This section gives a brief description of the RTE-V850E2/MN4-EB-S starter kit for the V850E2/MN4, manufactured by Midas lab Inc.

7.1 Outline

The RTE-V850E2/MN4-EB-S is a starter kit that allows you to experience the development of an application system using the V850E2/MN4. You can follow a sequence of development processes from program preparation, building, debugging, to operation check simply by installing required development tools and a USB driver on the host machine and connecting this kit via MINICUBE.

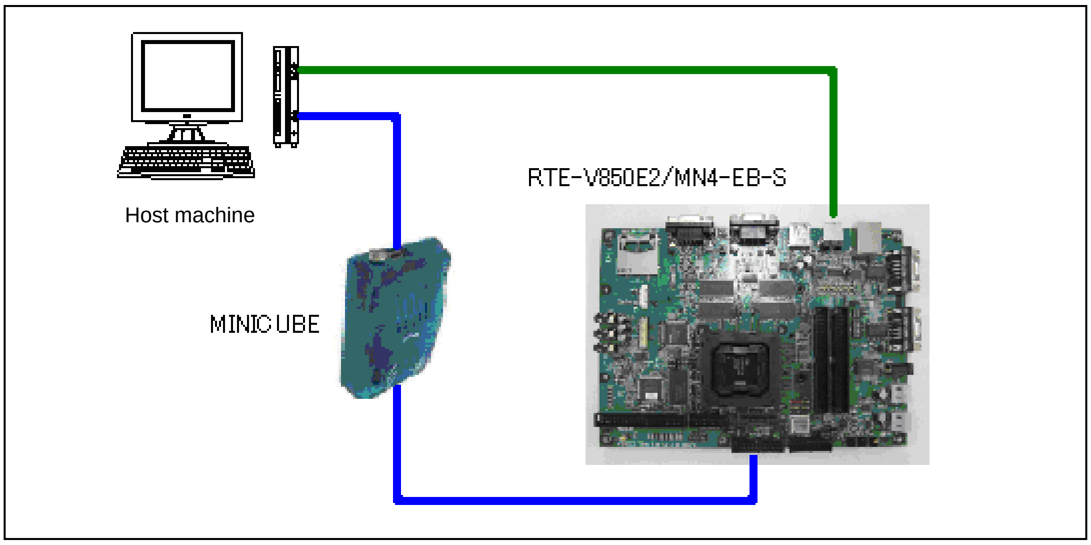


Figure 7.1 Outline of RTE-V850E2/MN4-EB-S Connection

7.2 Features of the Starter Kit

The RTE-V850E2/MN4-EB-S has the following features:

- 2 systems of memory controllers, DMA, timer array, UART, CSI , CAN, A/D converter, USB function controller, USB host controller, Ethernet controller, and other peripheral functions
- I/O ports for 7 input lines and 181 I/O lines
- Permits efficient development when combined with an integrated development environment (CubeSuite/Multi/IAR Embedded Workbench).

7.3 Major Specifications

The major specifications of the RTE-V850E2/MN4-EB-S are given below.

• CPU:	μPD70F3512 (V850E2/MN4)	
• Operating frequency:	200 MHz (PLL-driven x20 multiplier function)	
• Interface:	Two USB receptacles (USB host type A × 1, USB function type B × 1)	
	N-Wire connector	
	Two channels of UART	
	Two channels of CAN	
	Ethernet connector	
• Supported models:	Host machine:	PC/AT compatible with a USB interface
	OS:	Windows 2000 or Windows XP
• Operating voltage:	5.0 V	
• Dimensions:	W200 × D150 (mm)	

Website and Support

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

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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Jun 30, 2010	—	First edition issued.
1.01	Jan 14, 2011	All	Text format revised.
			Descriptions of GHS version added to Chapter 5, Development Environment.
1.02	Jan 23, 2012	2, 5, 7, 124, 129 to 136, 144, 149	Desriptions of IAR Embedded Workbench are added.

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.

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