To our customers,

# Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

# Notice

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

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

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

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

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

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

Application Note

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

## 32-bit Single-Chip Microcontrollers

## Updating USB Function Firmware

V850ES/JG3-H
μPD70F3760
μPD70F3761
μPD70F3762
μPD70F3770

V850ES/JH3-H
μPD70F3765
μPD70F3766
μPD70F3767
μPD70F3771

V850ES/JG3-U
μPD70F3763
μPD70F3764

V850ES/JH3-U
μPD70F3768
μPD70F3769

Document No. U19684EJ1V0AN00 (1st edition)
Date Published March 2009 N

[MEMO]

## NOTES FOR CMOS DEVICES

### ① VOLTAGE APPLICATION WAVEFORM AT INPUT PIN

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

### ② HANDLING OF UNUSED INPUT PINS

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

### ③ PRECAUTION AGAINST ESD

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

### ④ STATUS BEFORE INITIALIZATION

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

### ⑤ POWER ON/OFF SEQUENCE

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

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

### ⑥ INPUT OF SIGNAL DURING POWER OFF STATE

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

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

Caution: This product uses SuperFlash® technology licensed from Silicon Storage Technology, Inc.

- **The information in this document is current as of February, 2009. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

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

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

# PREFACE

**Caution**

**The sample programs used in this application note are simply for reference. NEC Electronics does not guarantee the operation of these programs.**
**Be sure to sufficiently evaluate the sample programs in your set before using them.**

**Readers**

This application note is intended for users who understand the features of the V850ES/JG3-H, V850ES/JH3-H, V850ES/JG3-U or V850ES/JH3-U, and are going to develop application systems using this product.

**Purpose**

This application note is intended to give users an understanding of the specifications of the sample driver provided for using the USB function controller incorporated in the V850ES/JG3-H, V850ES/JH3-H, V850ES/JG3-U, or V850ES/JH3-U.

**Organization**

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

- Overview of USB function firmware update
- Program organization
- How to use the application
- How to apply the sample program

**How to Read This Document**

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

To learn about the hardware features (particularly the roles of registers and how they should be set up) and electrical specifications of the V850ES/JG3-H, V850ES/JH3-H, V850ES/JG3-U, and V850ES/JH3-U microcontrollers:
→See the **V850ES/JG3-H, V850ES/JH3-H Hardware User's Manual** and the **V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual**.

To learn about the instruction set in detail:
→See the **V850ES Architecture User's Manual**.

**Conventions**

| | |
|---|---|
| Data significance: | Higher digits on the left and lower digits on the right |
| Active low representation: | $\overline{\text{xxx}}$ (overscore over pin or signal name) |
| Memory map address: | Higher addresses on the top and lower addresses on the bottom |
| **Note**: | Footnote for item marked with **Note** in the text |
| **Caution**: | Information requiring particular attention |
| **Remark**: | Supplementary information |
| Numeric representation: | Binary/Decimal... XXXX<br>Hexadecimal ... XXXXH or 0xXXXX |
| Prefix indicating power of 2 (address space, memory capacity): | K (kilo): $2^{10}$ = 1,024<br>M (mega): $2^{20}$ = $1{,}024^2$<br>G (giga): $2^{30}$ = $1{,}024^3$ |
| Data type: | Word ... 32 bits<br>Halfword ... 16 bits<br>Byte ... 8 bits |

**Related Documents**

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

Documents related to V850ES/JG3-H, V850ES/JH3-H, V850ES/JG3-U, and V850ES/JH3-U

| Document Name | Document No. |
|---|---|
| V850ES Architecture User's Manual | U15943E |
| V850ES/JG3-H, V850ES/JH3-H Hardware User's Manual | U19181E |
| V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual | U19287E |
| V850 Microcontrollers Flash Memory Self Programming Library Type 04 Ver. 1.20 User's Manual | U17819E |

Documents related to development tools (user's manuals)

| Document Name | | Document No. |
|---|---|---|
| QB-V850ESJX3H In-Circuit Emulator | | U19170E |
| QB-V850MINI On-Chip Debug Emulator | | U17638E |
| QB-MINI2 On-Chip Debug Emulator with Programming Function | | U18371E |
| CA850 Ver. 3.20 C Compiler Package | Operation | U18512E |
| | C Language | U18513E |
| | Assembly Language | U18514E |
| | Link Directives | U18515E |
| PM+ Ver. 6.30 Project Manager | | U18416E |
| ID850QB Ver. 3.40 Integrated Debugger | Operation | U18604E |
| SM850 Ver. 2.50 System Simulator | Operation | U16218E |
| SM850 Ver. 2.00 or Later System Simulator | External Part User Open Interface Specifications | U14873E |
| SM+ System Simulator | Operation | U18601E |
| | User Open Interface | U18212E |
| RX850 Ver. 3.20 Real-Time OS | Basics | U13430E |
| | Installation | U17419E |
| | Technical | U13431E |
| | Task Debugger | U17420E |
| RX850 Pro Ver. 3.21 Real-Time OS | Basics | U18165E |
| | In-Structure | U18164E |
| | Task Debugger | U17422E |
| AZ850 Ver. 3.30 System Performance Analyzer | | U17423E |
| PG-FP4 Flash Memory Programmer | | U15260E |
| PG-FP5 Flash Memory Programmer | | U18865E |

**Remarks 1.** The starter kit (TK-850/JH3U-SP) is a product of Tessera Technology Inc. Contact Tessera Technology Inc. for details.

**2.** The USB standard was formulated and is managed by the USB Implementers Forum (USB-IF).
To see the **Universal Serial Bus Class Definitions for Communication Devices**, visit the USB-IF website (www.usb.org).

## CONTENTS

# CHAPTER 1 OVERVIEW

## 1.1 Purpose

The purpose of this application note is give readers an understanding of how to overwrite data in the on-chip flash memory with user-specified values by using a flash-memory self-programming library (referred to hereafter as the *self-programming library*), as well as how to execute processing using the USB function controller communications device class (*CDC* hereafter).

This processing is illustrated using a sample program for updating the USB function firmware.

Note that the TK-850/JH3U-SP evaluation board that comes with an LCD panel is used as the evaluation environment. The TK-850/JH3U-SP is a product of Tessera Technology, Inc. The self-programming library used is Type 04 V1.20 from NEC Electronics.

## 1.2 Overview of Updating the USB Function Firmware

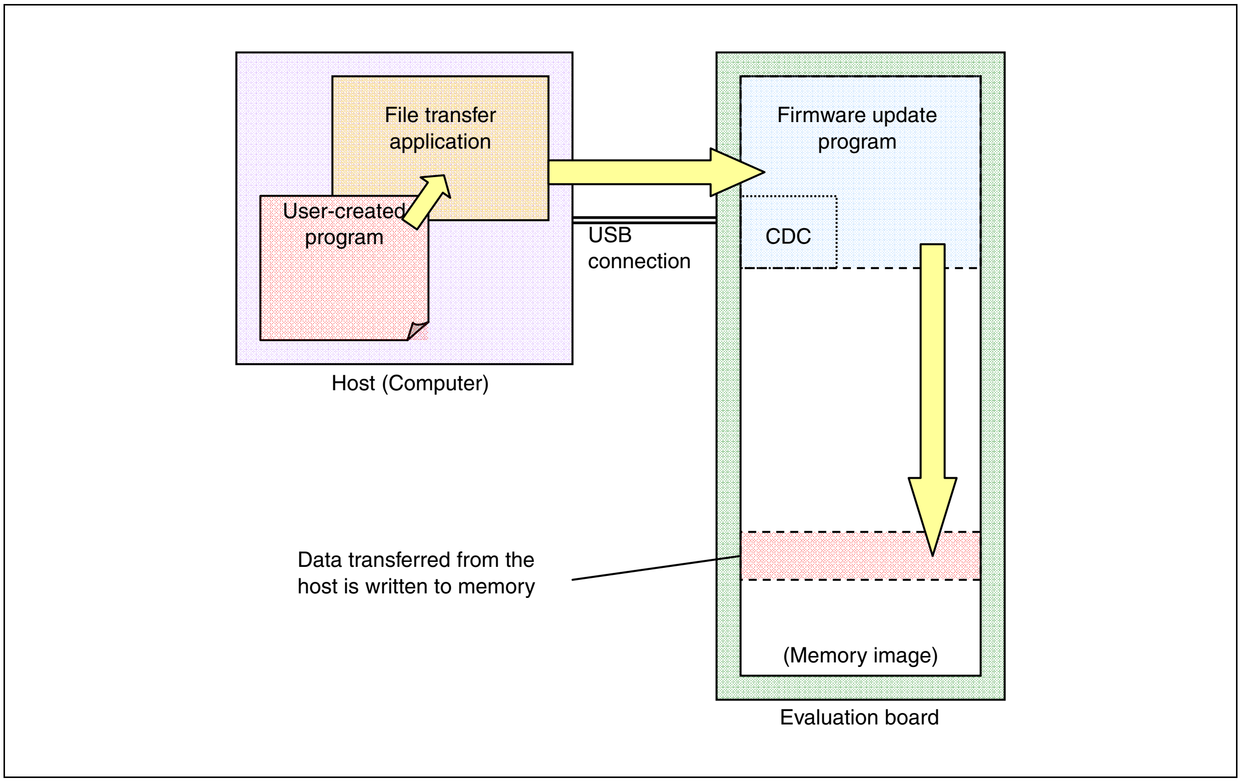
The sample program used to update the USB function firmware uses the file transfer application on the host (computer) to transfer the specified files to the evaluation board by means of USB serial communication. These files are then written to the boot area for the user-created program or to a memory location using the self-programming library.

The sample program used to update the USB function firmware includes the following:

- Firmware update program
  This program is written to the memory on the evaluation board and overwrites the USB function firmware via USB serial communication.
- File transfer application
  The file transfer application runs on the host and transfers the specified files to the evaluation board using serial communication.
- Sample user-created program
  This is a group of HEX files used to confirm that the programs are running correctly.
  Touch panel program: Items can be manipulated by touching the LCD screen.
  Photo frame program: Two images are switched repeatedly at set intervals.

The flow of data when updating the USB function firmware is shown below.

**Figure 1-1. Flow of Data When Updating USB Function Firmware**



Usually, the user-created program runs when the evaluation board is started up. However, the firmware update program will run when the evaluation board is started up under certain conditions or when the evaluation board is reset.

### 1.2.1 Features

The sample program for updating the USB function firmware has the following features:

- The firmware update program uses four blocks (16 KB) of internal flash memory.
- The user-created program (HEX files) can be overwritten in Motorola S-record format or Intel extended HEX format.
- Data can be written to any area in the memory by specifying memory addresses.
- All types of interrupts can be used in the user-created program.

The internal resources used by the firmware update program are shown in Table 1-1.

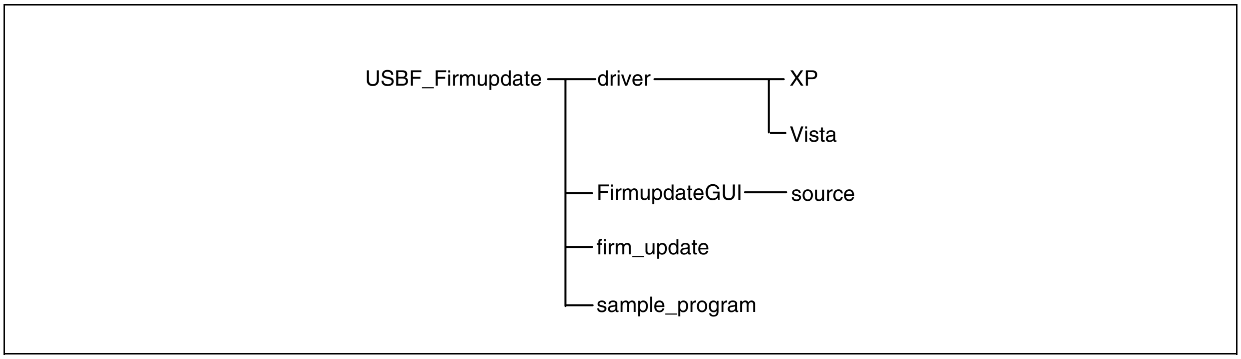
**Table 1-1. Internal Resources Used by the Firmware Update Program**

| Resource Name | Section Name | Size (Bytes) |
|---|---|---|
| ROM (CONST) | `.const` | 24 |
| ROM (TEXT) | `SelfLib_Rom.text`<br>`.text` | 5,444 |
| ROM | `apstart` | 52 |
| RAM (FLASHTEXT) | `SelfLib_ToRamUsrInt.text (8)`<br>`SelfLib_ToRamUsr.text (8)`<br>`SelfLib_RomOrRam.text (974)`<br>`SelfLib_ToRam.text (480)`<br>`flash.text (466)` | 1,936 |
| RAM (DATA) | `.data (12)`<br>`.sdata (200)`<br>`.sbss (5,280)`<br>`.bss (2,048)`<br>`SelfLib_RAM.bss (32)` | 7,572 |

### 1.2.2 Folder organization

The folders in the sample program for updating the USB function firmware are organized as shown in Figure 1-2 below.

**Figure 1-2. Organization of Folders in the Sample Program for Updating the USB Function Firmware**



The contents of these folders are described below.

**(1) `driver\XP`**

This folder stores the CDC driver for Windows XP$^{\text{TM}}$.

`JG3H_CDC_XP.inf`: CDC driver for Windows XP

**(2) `driver\VISTA`**

This folder stores the CDC driver for Windows Vista$^{\text{TM}}$.

`JG3H_CDC_VISTA.inf`: CDC driver for Windows Vista

**(3) `FimupdateGUI`**

This folder stores the file transfer application.

`UsbfUpdate.exe`: Executable file for the file transfer application

`UsbfUpdate.ini`: Configuration file for the file transfer application

**(4) `FirmupdateGUI\source`**

This folder stores the source program for the file transfer application. For details about this application, see **CHAPTER 4 FILE TRANSFER APPLICATION**.

**(5) `firm_update`**

This folder stores the firmware update program. For details about this program, see **CHAPTER 3 FIRMWARE UPDATE PROGRAM**.

**(6) `sample_program`**

This folder stores the sample user-created program.

`photo_sample.hex`: Photo frame program

`touch_sample.hex`: Touch panel program

# CHAPTER 2 EXECUTING THE SAMPLE PROGRAM FOR UPDATING THE USB FUNCTION FIRMWARE

This chapter describes how to execute the sample program for updating the USB function firmware.

The sample program for updating the USB function firmware is used to confirm that the user-created program has updated the firmware information in the memory on the evaluation board, and is executed using a touch panel program and then a photo frame program.

## 2.1 Operating Environment

The hardware environment is as follows:

| | |
|---|---|
| • Evaluation board | TK-850/JH3U-SP (product of Tessera Technology Inc.) |
| • Evaluation board CPU | μPD70F3769 (V850ES/JH3-U) |
| • In-circuit emulator | QB-V850MINI (MINICUBE®) |
| • USB cable | For executing serial communication between the evaluation board and host |
| • Host | Computer running Windows XP |

The software environment is as follows:

| | | |
|---|---|---|
| • Integrated development environment | PM+ V6.31 | |
| • Compiler | CA850 W3.30 | |
| • Debugger | ID850QB V3.50 | |
| • Sample program for updating USB function firmware, which includes the following: | | |
| | Firmware update program | |
| | File transfer application | |
| | Sample user-created program: | Touch panel program |
| | | Photo frame program |

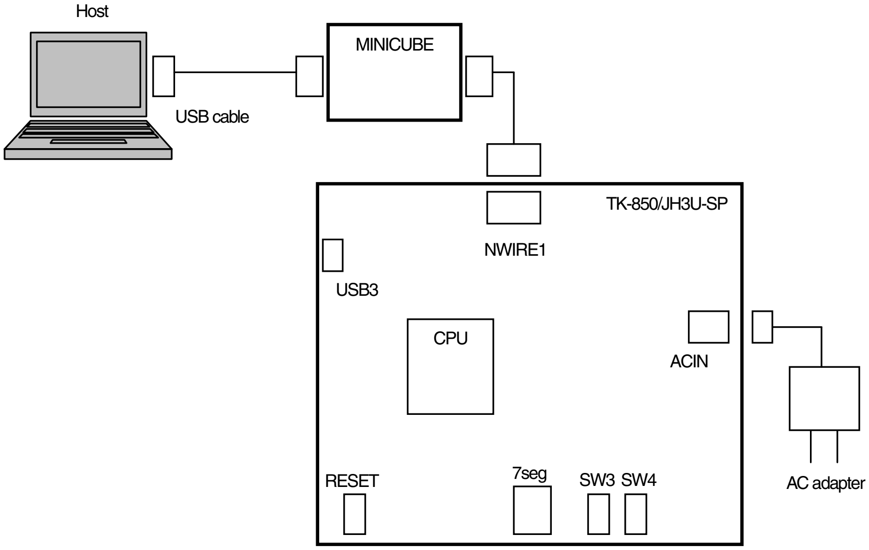
## 2.2 Executing the Sample Program

The operating environment in which the sample program for updating the USB function firmware is executed and the execution procedure are shown below.

### 2.2.1 Running the firmware update program

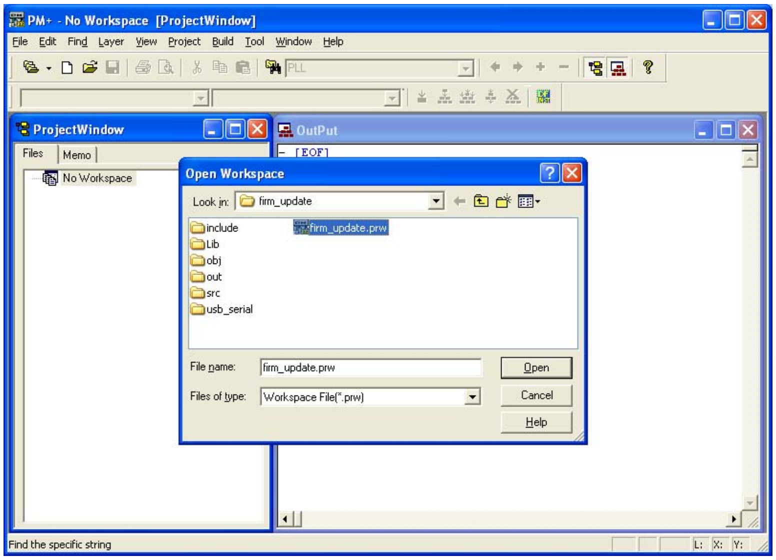
(1) Connect MINICUBE to the evaluation board as shown in Figure 2-1 below.

**Figure 2-1. Connecting MINICUBE to the Evaluation Board**



(2) Start PM+. On the **File** menu, click **Open Workspace**, and then select the workspace file `firm_update.prw`.

**Figure 2-2. Specifying the Workspace File**



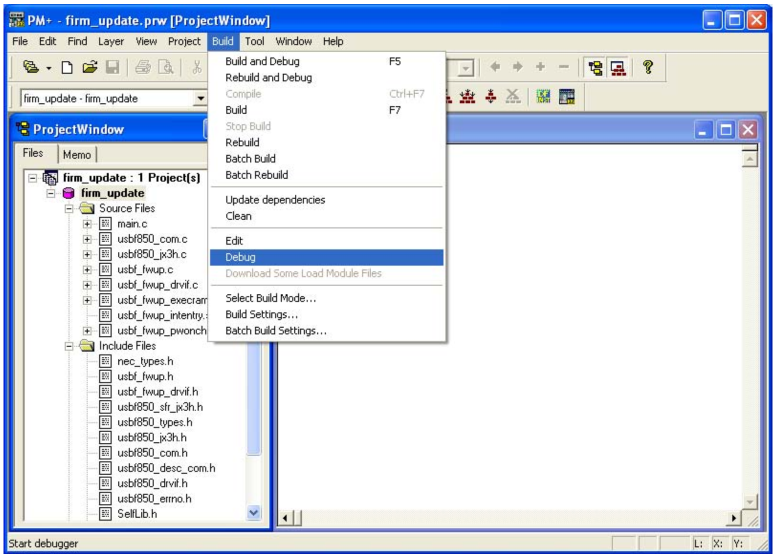
Look in: firm_update

File name: firm_update.prw

Files of type: Workspace File(*.prw)

(3) On the **Build** menu, click **Debug**. The firmware update program is written to the evaluation board.

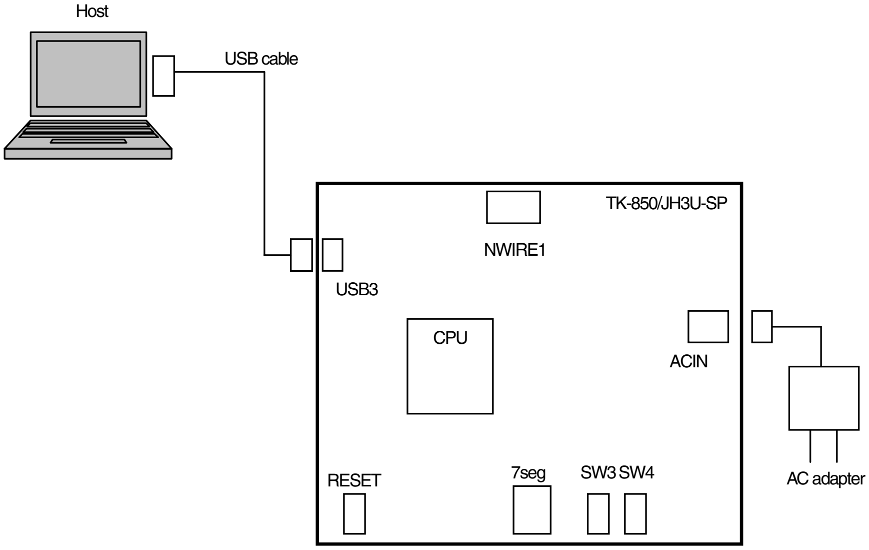
**Figure 2-3. Writing the Firmware Update Program to the Evaluation Board**



### 2.2.2 Updating the firmware information

(1) To update the firmware information, disconnect MINICUBE, and then connect the host to the evaluation board using the USB cable, as shown in Figure 2-4 below.

**Figure 2-4. Connecting the Host to the Evaluation Board**



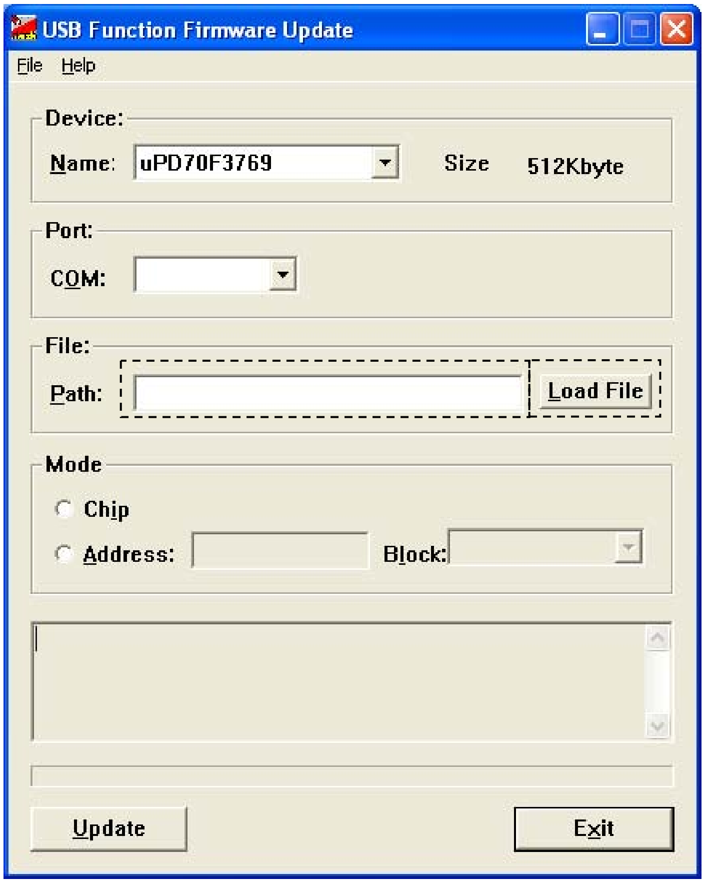
(2) Press the RESET button while holding down the SW3 and SW4 switches. When the mode changes to update mode, the host is ready to transfer data.

**Caution The CDC driver must be installed the first time the mode changes to update mode after connecting the host to the TK-850/JH3U-SP evaluation board. For details, see 2.2.3 Installing the CDC driver.**

(3) Load the HEX files of the sample user-created program to be transferred to the evaluation board into the host by specifying `touch_sample.hex` from the touch panel program. Start the file transfer application on the host (see **Figure 2-5**).
Click the **Load File** button, and then select the HEX file to be transferred. The file can be specified by typing the file path directly into the **Path** textbox, or by dragging the file path and dropping it into the **Path** textbox.
Under **Mode**, select **Chip**. In the **COM** drop-down list, select the USB port to which the host is connected. The USB port can be identified in the **Device Manager** window.

**Caution The COM number differs depending on the environment.**

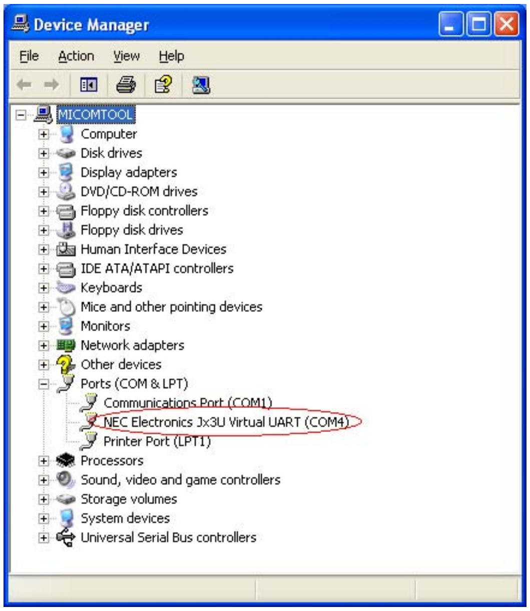
**Figure 2-5. Selecting the File to Be Transferred by the File Transfer Application**



USB Function Firmware Update

Device:
Name: uPD70F3769
Size 512Kbyte

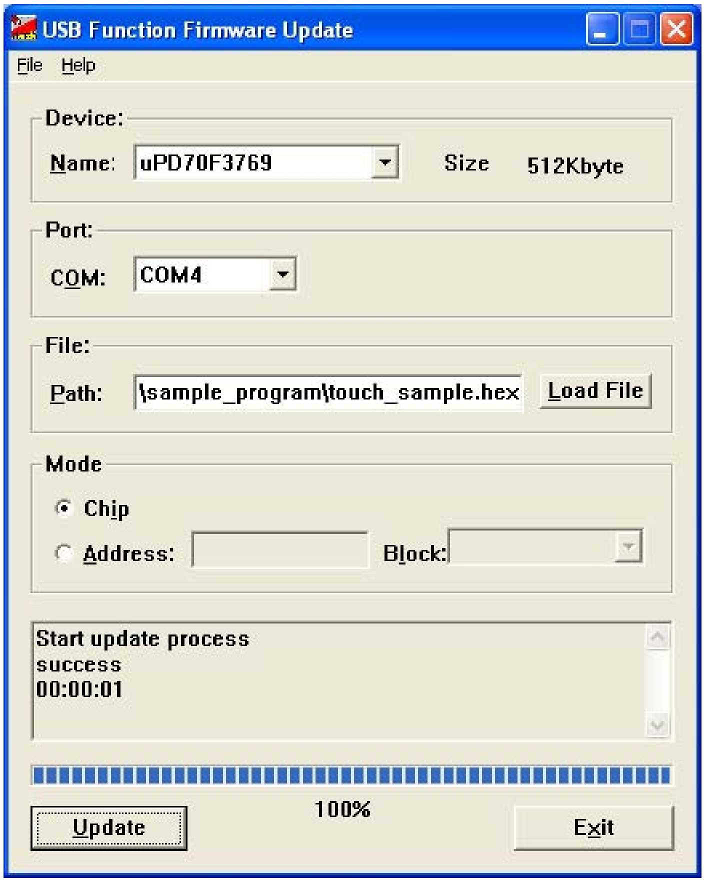
**Figure 2-6. Identifying the USB Port Using the Device Manager**



(4) Click the **Update** button in the **USB Function Firmware Update** window. A message indicating the start of transfer is displayed, the files are transferred, and the firmware information is updated.

(5) When the file transfer and firmware information update are complete, the file transfer application displays a message indicating the end of file transfer. This also means that the firmware information has been updated.

**Figure 2-7. End of Firmware Update 1**



Device:

Name: uPD70F3769 Size 512Kbyte

Port:

COM: COM4

File:

Path: \sample_program\touch_sample.hex

Mode

● Chip

○ Address: Block:

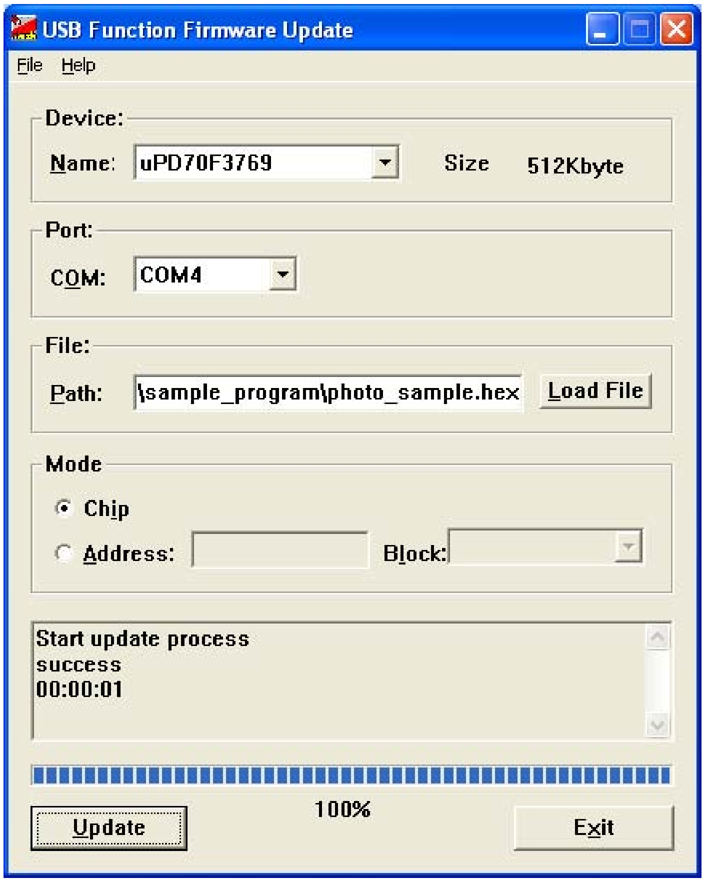
```
Start update process
success
00:00:01
```

100%

(6) Reset the evaluation board and start the user-created program that was written to the evaluation board in the previous steps.
Items on the LCD screen can now be manipulated by touching the screen directly.

(7) Update the user-created program. Load the photo frame program `photo_sample.hex` and execute the above procedure again from step (4).

Figure 2-8. End of Firmware Update 2



Device:
Name: uPD70F3769 Size 512Kbyte

Port:
COM: COM4

File:
Path: \sample_program\photo_sample.hex

Mode
● Chip
○ Address: Block:

```
Start update process
success
00:00:01
```

100%

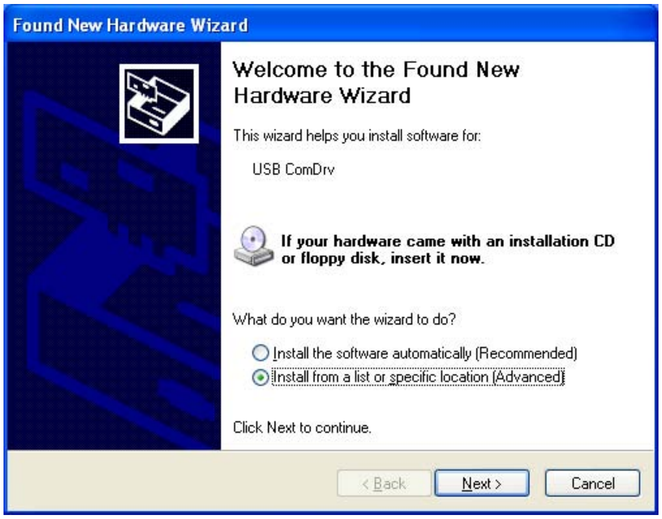
(8) Reset the evaluation board and start the user-created program that was written to the evaluation board in the previous steps.
The images on the LCD screen will switch at set intervals.

### 2.2.3 Installing the CDC driver

The CDC driver must be installed on the host the first time the mode changes to update mode after connecting the host to the TK-850/JH3U-SP evaluation board. The procedure for installing the CDC driver is shown below, using the Windows XP environment as an example.

(1) When the host detects new hardware, it opens the **Found New Hardware wizard** window. Select **Install from a list or specific location (Advanced)**, and then click **Next**.

**Figure 2-9. Found New Hardware Wizard**



Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

This wizard helps you install software for:

USB ComDrv

If your hardware came with an installation CD or floppy disk, insert it now.

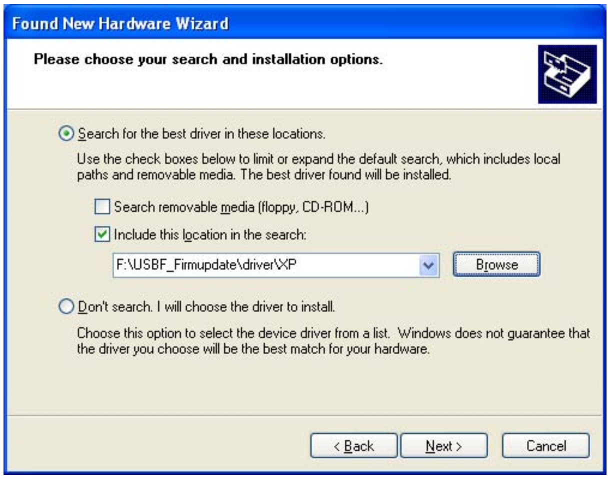
What do you want the wizard to do?

○ Install the software automatically (Recommended)
● Install from a list or specific location (Advanced)

Click Next to continue.

(2) Under **Search for the best driver in these locations**, select **Include this location in the search**.
Click **Browse**, select the folder that includes the file `JG3H_CDC_XP.inf`, and then click **Next**.

**Figure 2-10. Selecting the Driver Location**



Found New Hardware Wizard

Please choose your search and installation options.

● Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

□ Search removable media (floppy, CD-ROM...)
☑ Include this location in the search:

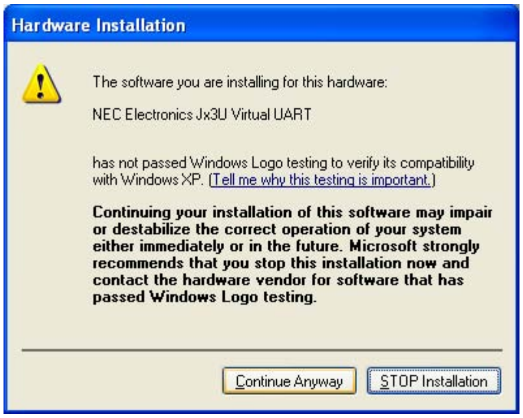
F:\USBF_Firmupdate\driver\XP

○ Don't search. I will choose the driver to install.

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

(3) A warning message appears. Click **Continue Anyway**.

**Figure 2-11. Warning Message**



Hardware Installation

The software you are installing for this hardware:

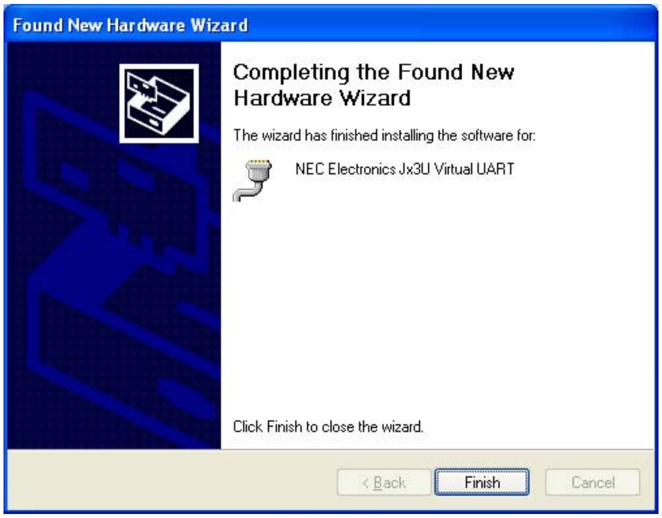
NEC Electronics Jx3U Virtual UART

has not passed Windows Logo testing to verify its compatibility with Windows XP. (Tell me why this testing is important.)

**Continuing your installation of this software may impair or destabilize the correct operation of your system either immediately or in the future. Microsoft strongly recommends that you stop this installation now and contact the hardware vendor for software that has passed Windows Logo testing.**

(4) The installation wizard ends with the following window. Click **Finish**.

**Figure 2-12. End of Installation**



Found New Hardware Wizard

Completing the Found New Hardware Wizard

The wizard has finished installing the software for:

NEC Electronics Jx3U Virtual UART

Click Finish to close the wizard.

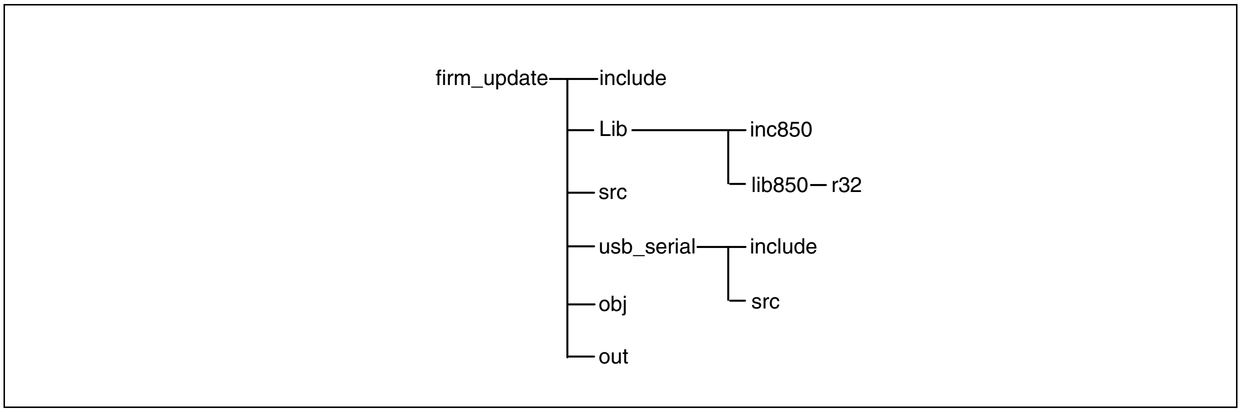
# CHAPTER 3 FIRMWARE UPDATE PROGRAM

This chapter describes the files used by the firmware update program.

## 3.1 Organization of Files and Folders

The files and folders that store the source code of the firmware update program are organized as follows.

**Figure 3-1. Organization of Firmware Update Program Folders**



### 3.1.1 `firm_update` folder

This folder stores the project files used by the firmware update program. The main project files in the `firm_update` folder are shown in Table 3-1 below.

**Table 3-1. Project Files Used by the Firmware Update Program**

| File Name | Description |
|---|---|
| `firm_update.prw` | PM+ workspace file |
| `firm_update.prj` | PM+ project file |
| `firm_update.pri` | PM+ project PRI file |
| `firm_update.cld` | PM+ project CLD file |
| `firm_update.mak` | Make file |
| `firm_update.dir` | Linker directive file |

### 3.1.2 `firm_update\include` folder

This folder stores the header files used by the firmware update program.

**Table 3-2. Header Files Used by the Firmware Update Program**

| File Name | Description |
|---|---|
| `usbf_fwup.h` | Header file used when executing self updating |
| `usbf_fwup_drvif.h` | Header file for the USB function control driver interface |
| `usbf_fwup_mem_def_usr.h` | Header file in which the firmware update memory allocation has been customized by the user |

### 3.1.3 `firm_update\lib` folder

This folder stores the self-programming library.

**Table 3-3. Self-Programming Library and Library Header Files**

| File Name | Description |
|---|---|
| `inc850\nec_types.h` | Header file defining types in a unified format |
| `inc850\SelfLib.h` | Header file for the self-programming library |
| `lib850\r32\libf.a` | Self-programming library |

### 3.1.4 `firm_update\src` folder

This folder stores the source files for the firmware update program.

**Table 3-4. Source Files for Firmware Update Program**

| File Name | Description |
|---|---|
| `crtE.s` | Startup file |
| `main.c` | Main routine source file |
| `usbf_fwup_intentry.s` | Interrupt entry source file in the flash environment<br>(For details about the flash environment, see **3.5 Interrupt Processing**.) |
| `usbf_fwup.c` | Source file used when executing self updating |
| `usbf_fwup_execram.c` | Source file used to write data to the flash memory |
| `usbf_fwup_pwonchk_usr.c` | Source file customized by the user<br>(Specify code for determining whether to execute the self-update program or the user-created program in this file.) |
| `usbf_fwup_drvif.c` | Source file for interfacing with the CDC driver |

### 3.1.5 `firm_update\usb_serial` folder

This folder stores the source files and header files used by the CDC program.

**Table 3-5. Source Files and Header Files Used by the CDC Program**

| File Name | Description |
|---|---|
| `include\usbf850_types.h` | Header file defining types in a unified format |
| `include\usbf850_error.h` | Header file defining end codes and error codes |
| `include\usbf850_jx3h.h` | Header file defining the macro for specifying USB register settings |
| `include\usbf850_sfr_jx3h.h` | Header file defining the macro for controlling USB function registers |
| `include\usbf850_desc_com.h` | Header file containing descriptor definitions |
| `include\usbf850_com.h` | Header file for executing processing specific to the CDC |
| `include\usbf850_devif.h` | Header file defining the interface with the CDC driver |
| `src\usbf850_jx3h.c` | Source file for initializing the USB registers, controlling the endpoints, and executing bulk and control transfers |
| `src\usbf850_com.c` | Source file for executing processing specific to the CDC |

### 3.1.6 `firm_update\obj` folder

This folder stores the object files used by the firmware update program.

### 3.1.7 `firm_update\out` folder

This folder stores the executable object file and HEX file used by the firmware update program.

| File Name | Description |
|---|---|
| `romp.out` | Executable object file |
| `firm_update.hex` | Executable object file in HEX format |

## 3.2 Memory Map

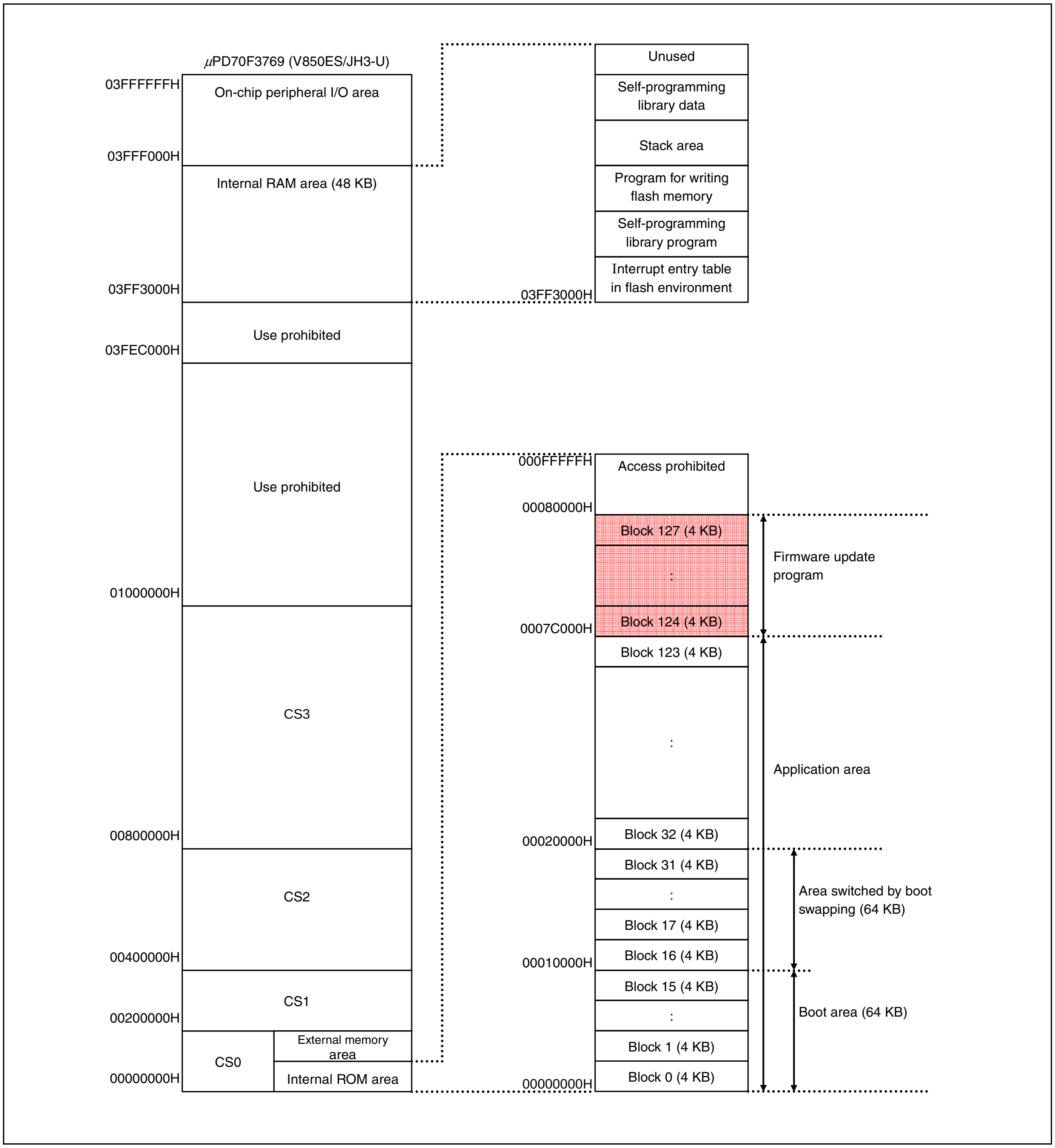
This section describes the memory allocation and the linker directive file.

### 3.2.1 Memory map

The memory map of the self-update program is shown below.

In the memory map below, *block* refers to the unit in which the internal flash memory is updated by the self-programming library.

**Figure 3-2. Memory Map**



### 3.2.2 Linker directive file (`flash_update.dir`)

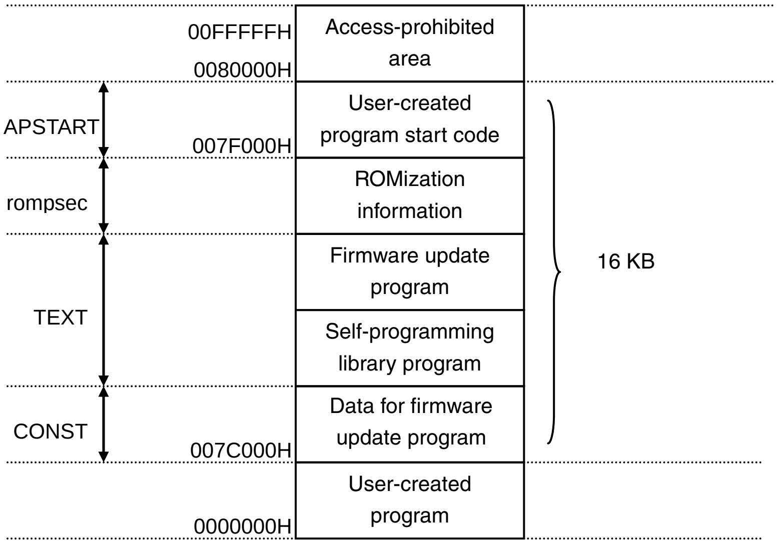
The linker directive file (`flash_update.dir`) is used to assign areas. The memory is mapped by defining segments.

Areas such as executable sections (`.text`: program data), nonexecutable sections (`.const`: constant data), and RAM areas are allocated to the memory of the μPD70F3769 (V850ES/JH3-U) based on the information in this file.

**(1) Assignment of ROM area**

Data used by the firmware update program is allocated to the 16 KB ROM area of addresses 0007C000H to 0007FFFFH. The user-created program must therefore be allocated within the 496 KB ROM area of addresses 00000000H to 0007BFFFH.

**Figure 3-3. Linker Directives for the Assigning ROM Area**



```
CONST    : !LOAD ?R V0x7c000 {
     .const           = $PROGBITS  ?A .const;
};

TEXT     : !LOAD ?RX {
     SelfLib_Rom.text = $PROGBITS  ?AX SelfLib_Rom.text;
     .text            = $PROGBITS  ?AX .text;
};

APSTART  : !LOAD ?RX V0x7f000 {
     apstart.text     = $PROGBITS  ?AX apstart.text;
};
```

The sections added by these directives are described below.

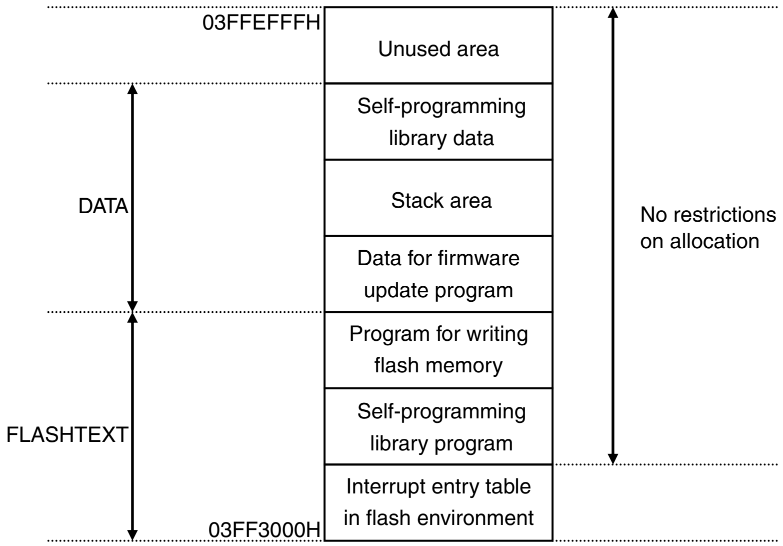
| Section | Description |
|---|---|
| `SelfLib_Rom.text` | Section used to initialize the self-programming library program |
| `.text` | Section to which the firmware update program is allocated |
| `apstart.text` | Area to which the code for jumping to the user-created program is written. This code is executed by the firmware update program. |

### (2) Assignment of the RAM area

The RAM area is allocated to addresses 3FF3000H to 3FFEFFFH.

The 8 bytes from address 3FF3000H constitute the interrupt entry table in the flash environment. Note that the interrupt entry table is allocated to the RAM area even though the firmware update program does not use interrupts in the flash environment. For details about interrupts, see **3.5 Interrupt Processing**.

**Figure 3-4. Linker Directives for Assigning the RAM Area**


03FFEFFFH
Unused area
Self-programming library data
Stack area
Data for firmware update program
Program for writing flash memory
Self-programming library program
Interrupt entry table in flash environment
03FF3000H
DATA
FLASHTEXT
No restrictions on allocation

```
FLASHTEXT: !LOAD ?RX V0x3ff3000 {
    SelfLib_ToRamUsrInt.text = $PROGBITS ?AX SelfLib_ToRamUsrInt.text;
    SelfLib_ToRamUsr.text    = $PROGBITS ?AX SelfLib_ToRamUsr.text;
    SelfLib_RomOrRam.text    = $PROGBITS ?AX SelfLib_RomOrRam.text;
    SelfLib_ToRam.text       = $PROGBITS ?AX SelfLib_ToRam.text;
    flash.text               = $PROGBITS ?AX flash.text;
};

DATA    : !LOAD ?RW {
    .data           = $PROGBITS    ?AW  .data;
    .sdata          = $PROGBITS    ?AWG .sdata;
    .sbss           = $NOBITS      ?AWG .sbss;
    .bss            = $NOBITS      ?AW  .bss;
    SelfLib_RAM.bss = $NOBITS      ?AW  SelfLib_RAM.bss;
};
```

The sections added by these directives are described below.

| Section | Description |
|---|---|
| `SelfLib_ToRamUsrInt.text` | Section used to execute interrupt processing in the self-programming library |
| `SelfLib_ToRamUsr.text` | Section where the user-created program is allocated |
| `SelfLib_RomOrRam.text` | Section used to interface with the self-programming library |
| `SelfLib_ToRam.text` | Section used to call the flash macro service in the self-programming library |
| `flash.text` | Work area on the RAM for the firmware update program |
| `SelfLib_RAM.text` | Work area for the self-programming library |

For details about the linker directives, see the **CA850 User's Manual**.

For details about the self-programming library, see **V850 Microcontrollers Flash Memory Self-Programming Library Type 04 Ver. 1.20 User's Manual**.

## 3.3 Boot Processing

Boot processing is executed by the boot program before the `main` function (`main ()` in C) is executed after the V850 microcontroller is reset.

After the V850 microcontroller is reset, the following initialization processing is executed:

- The reset handler that operates when a reset occurs is set up.
- The startup routine registers are set up.
- The stack area is allocated and the stack pointer is set up.
- The area for storing the arguments of the `main` function is allocated.
- The tp, gp, and ep registers are set up, as well as the mask values for the mask registers.
- Peripheral I/O registers are initialized that is required before the `main` function is executed.
- The sbss, bss, sebss, tibss.byte, tibss.word, and sibss areas are initialized.
- The program branches to the `main` function.

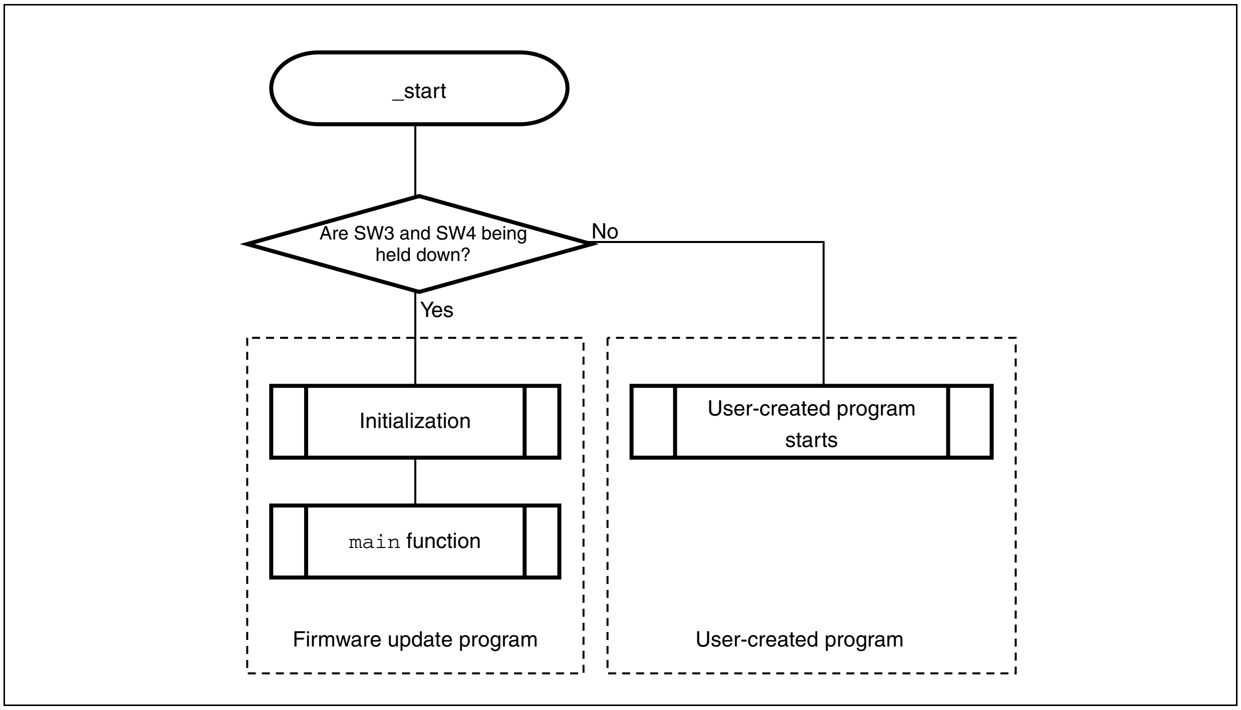
The boot processing to be executed is defined in the startup file (`crtE.s`).

For details about this processing, see the **CA850 User's Manual**.

With the firmware update program, there is also an option to branch to the user-created program and initialize the V850 microcontroller during boot processing.

An overview of the boot processing is shown in Figure 3-5 below.

**Figure 3-5. Overview of the Boot Processing in the Firmware Update Program**



### 3.3.1 Startup file (`crtE.s`)

The startup file of the firmware update program is described below.

**Figure 3-6. Startup File (1/4)**

```
#----------------------------------------------------------------------------
#         special symbols
#----------------------------------------------------------------------------
          .extern   __tp_TEXT, 4
          .extern   __gp_DATA, 4
          .extern   __ep_DATA, 4
          .extern   __ssbss, 4
          .extern   __esbss, 4
          .extern   __sbss, 4
          .extern   __ebss, 4

#----------------------------------------------------------------------------
#          C program main function
#----------------------------------------------------------------------------
          .extern   _main
          .extern   _usbf_fwup_pwonchk_usr

#----------------------------------------------------------------------------
#         for argv
#----------------------------------------------------------------------------
          .data
          .size     __argc, 4
          .align    4
__argc:
          .word     0
          .size     __argv, 4
__argv:
          .word     #.L16
.L16:
          .byte     0
          .byte     0
          .byte     0
          .byte     0

#----------------------------------------------------------------------------
#         dummy data declaration for creating sbss section
#----------------------------------------------------------------------------
          .sbss
          .lcomm    __sbss_dummy, 0, 0

#----------------------------------------------------------------------------
#         system stack
#----------------------------------------------------------------------------
          .set      STACKSIZE, 0x800
          .bss
          .lcomm    __stack, STACKSIZE, 4

#----------------------------------------------------------------------------
#         RESET handler
#----------------------------------------------------------------------------
          .section  "RESET", text
          jr        __start
```

Allocates 2,048 bytes for the stack area.

The program branches to the _start reset vector (0000H) after a reset.

**Figure 3-6. Startup File (2/4)**

```
#-----------------------------------------------------------------------------
#         application start routine
#-----------------------------------------------------------------------------
                    .section "apstart.text", text
                    .align    4
                    .globl    __apstart
__apstart:
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    nop
                    halt

#-----------------------------------------------------------------------------
#         start up
#                   pointers: tp - text pointer
#                             gp - global pointer
#                             sp - stack pointer
#                             ep - element pointer
#                   mask reg: r20 - 0xff
#                             r21 - 0xffff
#         exit status is set to r10
#-----------------------------------------------------------------------------
                    .text
                    .align    4
                    .globl    __start
                    .globl    __exit
                    .globl    __startend

__start:
                    mov       #__tp_TEXT, tp              -- set tp register
                    mov       #__gp_DATA, gp              -- set gp register offset
                    add       tp, gp                      -- set gp register
                    mov       #__stack+STACKSIZE, sp      -- set sp register
                    mov       #__ep_DATA, ep              -- set ep register
```

This is the area where the code for branching to the start of the user-created program is written.
The self-update program updates this area when the user-created program is written.

Sets up the tp, gp, ep, and sp registers.

**Figure 3-6. Startup File (3/4)**

```
                   .option nowarning
                   mov      0xff,  r20          -- set mask register
                   mov      0xffff, r21         -- set mask register
                   .option warning

                   mov      #__ssbss, r13       -- clear sbss section
                   mov      #__esbss, r12
                   cmp      r12, r13
                   jnl      .L11
.L12:
                   st.w     r0, [r13]
                   add      4, r13
                   cmp      r12, r13
                   jl       .L12
.L11:

                   mov      #__sbss, r13        -- clear bss section
                   mov      #__ebss, r12
                   cmp      r12, r13
                   jnl      .L14
.L15:
                   st.w     r0, [r13]
                   add      4, r13
                   cmp      r12, r13
                   jl       .L15
.L14:
      #----------------------------------------
      # Which program is executed is examined.
      # Firm update program or User program.
      jarl     _usbf_fwup_pwonchk_usr, lp
      cmp      0, r10
      jnz      __apstart
      #----------------------------------------

                   jarl     ___Init_jh3u, lp

                   .extern __S_romp, 4
                   mov      #__S_romp, r6
                   mov      -1, r7
                   jarl     __rcopy, lp

                   ld.w     $__argc, r6         -- set argc
                   movea    $__argv, gp, r7     -- set argv
                   jarl     _main, lp           -- call main function
__exit:
                   halt     -- end of program
__startend:
```

Sets up the mask registers.

Initializes the RAM.

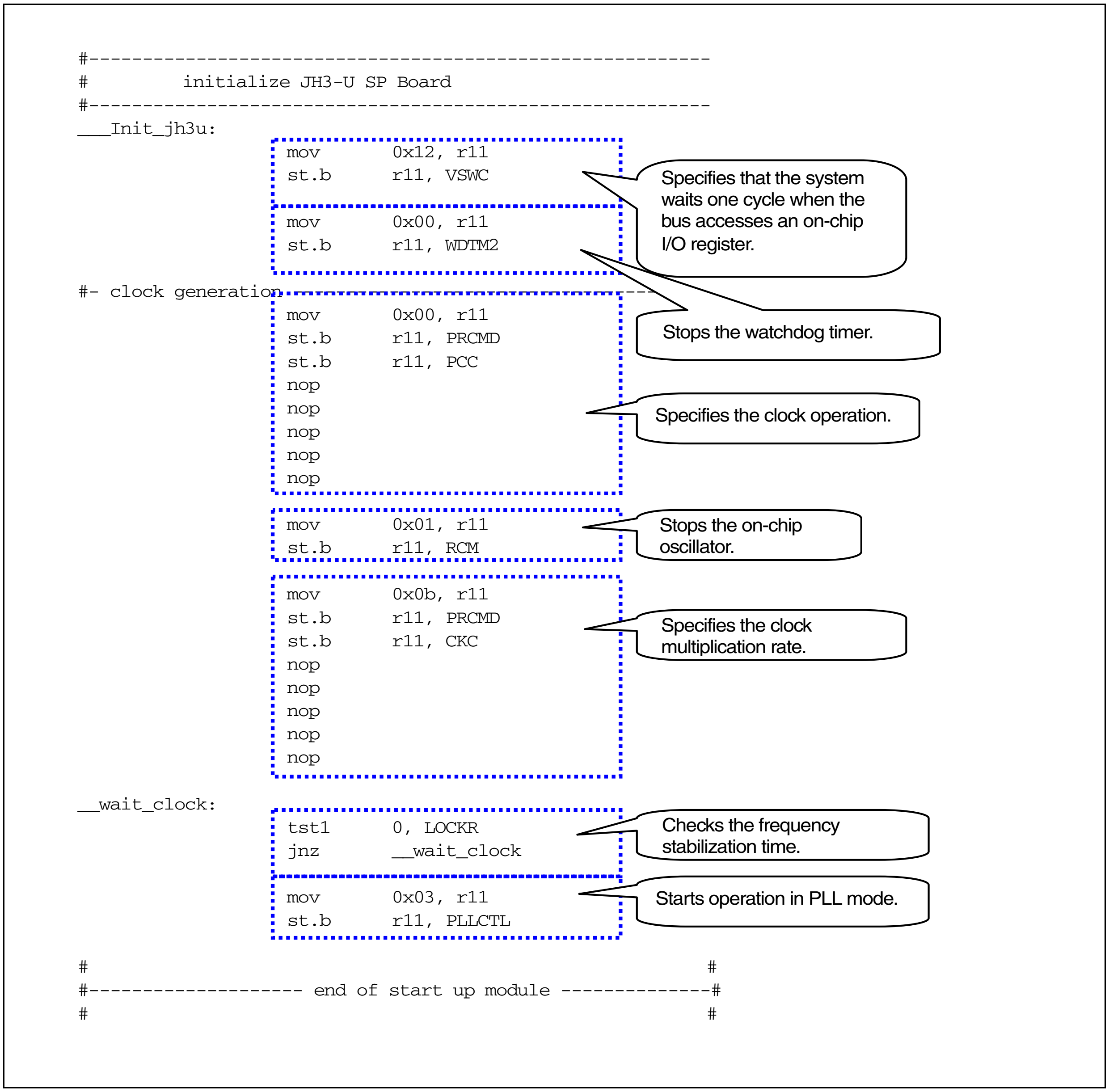
The status of the switches is referenced by the `usr_startchk` function, which judges whether to branch to the user-created program or the firmware update program.

Shifts to V850 microcontroller initialization.

Transfers data to the RAM.

The program branches to the `main` function.

**Figure 3-6. Startup File (4/4)**



```
#----------------------------------------------------------
#         initialize JH3-U SP Board
#----------------------------------------------------------
___Init_jh3u:
                  mov       0x12, r11
                  st.b      r11, VSWC

                  mov       0x00, r11
                  st.b      r11, WDTM2

#- clock generation -------------------------------------
                  mov       0x00, r11
                  st.b      r11, PRCMD
                  st.b      r11, PCC
                  nop
                  nop
                  nop
                  nop
                  nop

                  mov       0x01, r11
                  st.b      r11, RCM

                  mov       0x0b, r11
                  st.b      r11, PRCMD
                  st.b      r11, CKC
                  nop
                  nop
                  nop
                  nop
                  nop

__wait_clock:
                  tst1      0, LOCKR
                  jnz       __wait_clock

                  mov       0x03, r11
                  st.b      r11, PLLCTL

#                                                         #
#-------------------- end of start up module --------------#
#                                                         #
```

Specifies that the system waits one cycle when the bus accesses an on-chip I/O register.

Stops the watchdog timer.

Specifies the clock operation.

Stops the on-chip oscillator.

Specifies the clock multiplication rate.

Checks the frequency stabilization time.

Starts operation in PLL mode.

The CPU clock and the peripheral functions to be used are specified during initialization.
For details about using the evaluation board, see the **TK-850/JH3U-SP User's Manual**.
For details about using the CPU, see the **V850ES/JG3-U, V850ES/JH3-U Hardware User's Manual**.

### 3.3.2 Checking where to branch to when the power is turned on

When the power is turned on, the `usr_startchk` function is called from the startup file and judges whether to branch to the firmware update program or to the user-created program, according to the status of the SW3 and SW4 switches of the TK-850/JG3H. If both switches are being held down, the user-created program is executed. In other cases, the firmware update program is executed.

**Figure 3-7. Checking Where to Branch to When the Power Is Turned On**

```
#pragma ioreg

#define SW_PUSHED   0x00      /* pushed switch SW3 and SW4 */
#define SW_STATUS   0x03      /* switch status SW3 and SW4 */

s32 usbf_fwup_pwonchk_usr(void);

s32 usbf_fwup_pwonchk_usr(void){
          int       ret = -1;
          unsigned char  sts;

          sts = P9H;
          if ((sts & SW_STATUS) == SW_PUSHED) {
                    ret = 0;
          }

          return ret;
}
```

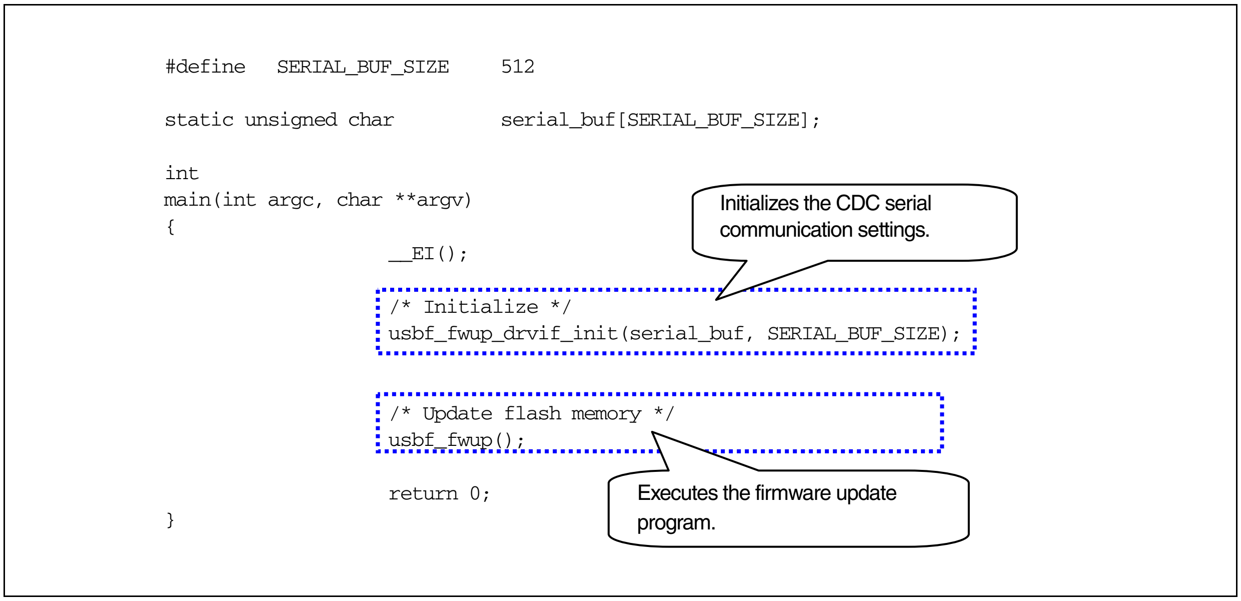
Judges the status of the SW3 and SW4 switches.

## 3.4 Main Routine

At the end of boot processing, the program branches to the `main` function and executes the main routine.

In the main routine, the settings for CDC serial communication are initialized, and then the firmware update program is executed.

**Figure 3-8. Main Routine**



```
#define   SERIAL_BUF_SIZE     512

static unsigned char          serial_buf[SERIAL_BUF_SIZE];

int
main(int argc, char **argv)
{
                    __EI();
                    /* Initialize */
                    usbf_fwup_drvif_init(serial_buf, SERIAL_BUF_SIZE);

                    /* Update flash memory */
                    usbf_fwup();

                    return 0;
}
```

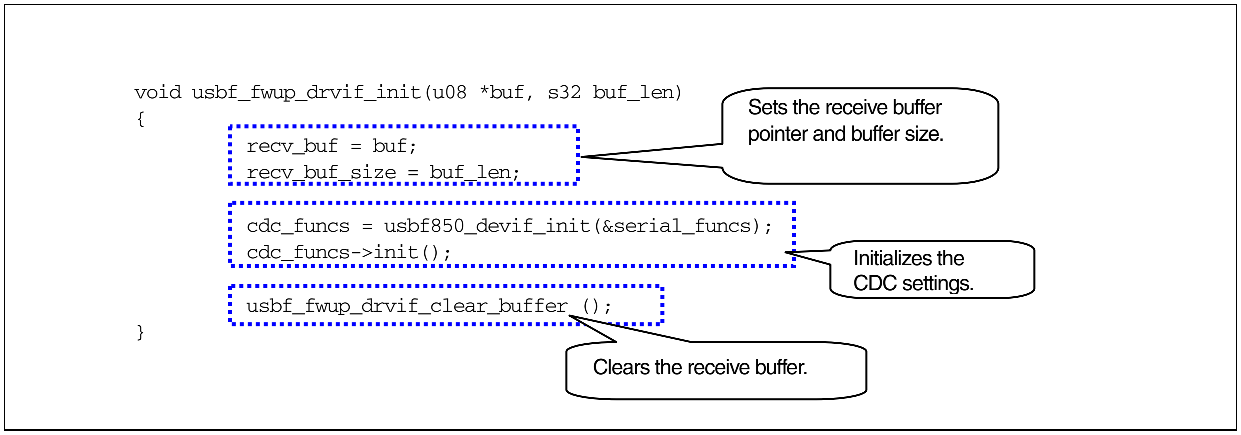
Initializes the CDC serial communication settings.

Executes the firmware update program.

### 3.4.1 Initializing the settings for USB communication (`usbf_fwup_drvif.c`)

The `usbf_fwup_drvif.c` file contains the function used to initialize the settings for USB serial communication. The structure in which the functions used to receive data are defined is passed to the `usbf850_devif_init` function. A pointer to the structure in which the functions used in the CDC processing are defined is received as the return value and the initialization function in that structure is called.

**Figure 3-9. Initialization of USB Communication Settings**



```
void usbf_fwup_drvif_init(u08 *buf, s32 buf_len)
{
        recv_buf = buf;
        recv_buf_size = buf_len;

        cdc_funcs = usbf850_devif_init(&serial_funcs);
        cdc_funcs->init();

        usbf_fwup_drvif_clear_buffer ();
}
```

Sets the receive buffer pointer and buffer size.

Initializes the CDC settings.

Clears the receive buffer.

`cdc_funcs` and `serial_funcs` are defined in the same source file.

For details about the `usbf_fwup_drvif_read` function, see **3.7.3 Monitoring EP1**.

**Figure 3-10. Definition of `cdc_funcs` and `serial_funcs`**

```
static const struct usb_cdc_funcs_st *cdc_funcs = (const struct usb_cdc_funcs_st *)0;

static const struct usb_serial_funcs_st serial_funcs = {
   usbf_fwup_drvif_read
};
```

Specifies the `usbf_fwup_drvif_read` function for reception processing.

The `usb_serial_func_st` and `usb_cdc_func_st` structures are defined in the `usbf850_drvif.h` file.

**Figure 3-11. `usb_serial_funcs_st` and `usb_cdc_funcs_st` Structures**

```
#ifndef __USBF850_DRVIF_H__
#define __USBF850_DRVIF_H__

struct usb_serial_funcs_st {
   void    (*read)(UINT8 len);
};
struct usb_cdc_funcs_st {
   void    (*init)(void);
   void    (*int0b)(void);
   void    (*int1b)(void);
   INT32   (*datasend)(UINT8* data, INT32 len, INT8 ep);
   INT32   (*datareceive)(UINT8* data, INT32 len, INT8 ep);
};

const struct usb_cdc_funcs_st *usbf850_devif_init(const struct usb_serial_funcs_st *funcs);

#endif/* __USBF850_DRVIF_H__ */
```

Structure for executing reception processing

Structure for executing CDC communication

The `usbf850_devif_init` function is defined in the `usbf850_jx3h.c` file.

**Figure 3-12. `usbf850_devif_init` Function**

```
const struct usb_cdc_funcs_st *usbf850_devif_init(const struct usb_serial_funcs_st *funcs)
{
   serial_funcs = funcs;

   return &cdc_funcs;
}
```

Returns the `cdc_funcs` pointer.

`serial_funcs` and `cdc_funcs` are defined in the same source file.

**Figure 3-13. Definition of `serial_funcs` and `cdc_funcs`**

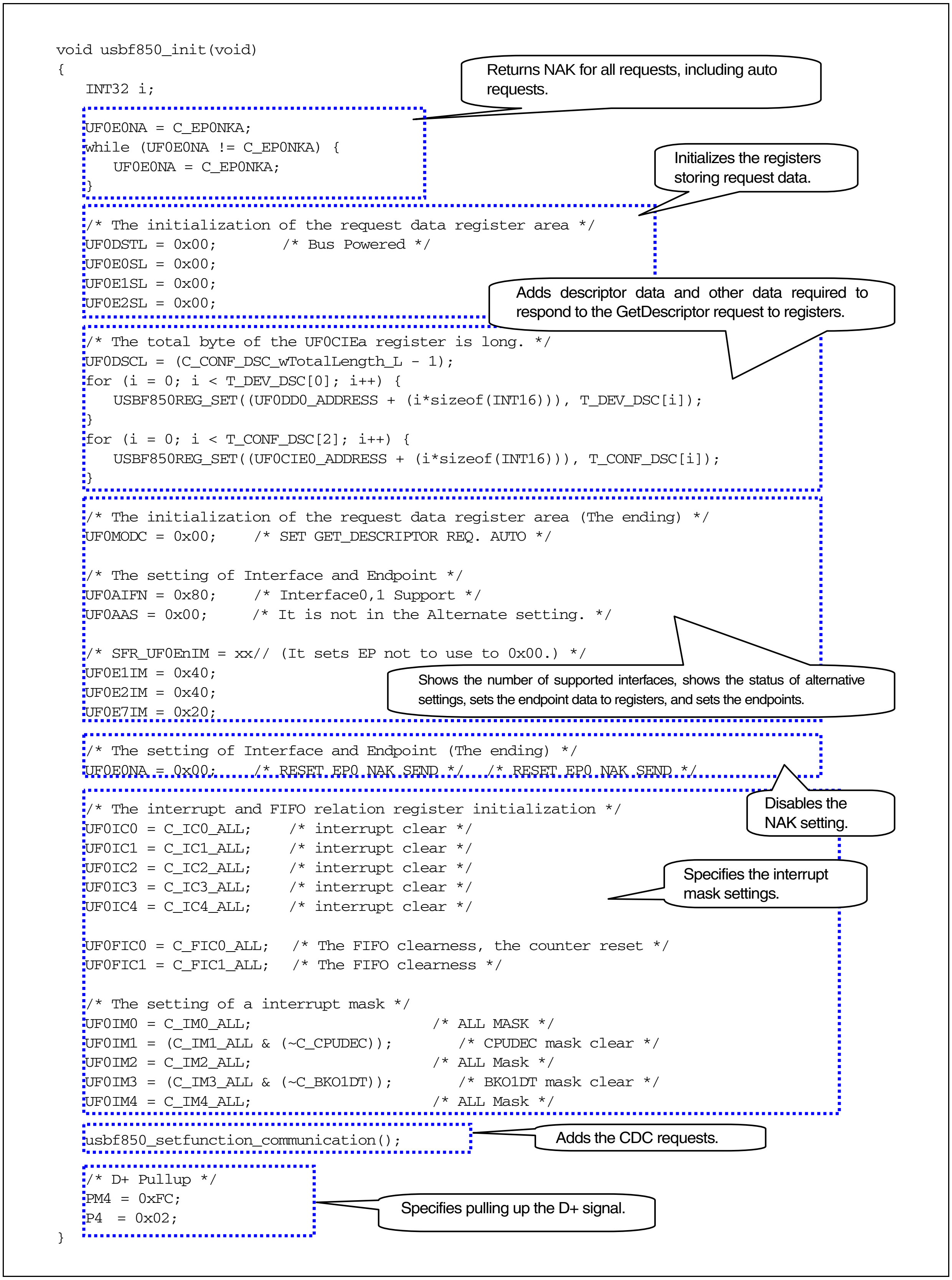
```
static const struct usb_serial_funcs_st *serial_funcs = (const struct usb_serial_funcs_st *)0;

static const struct usb_cdc_funcs_st cdc_funcs = {
   usbf850_init,
   usbf850_intusb0b,
   usbf850_intusb1b,
   usbf850_data_send,
   usbf850_data_receive
};
```

Structure in which the functions used for CDC communication are defined

According to the above definition, the `usbf850_init` function is called by the `cdc_funcs->init();` statement in the `usbf_fwup_drvif_init` function.

The `usbf850_init` function is shown below. For details about the `usbf850_intusb0b`, `usbf850_intusb1b`, `usbf850_data_send`, and `usbf850_data_receive` functions, see **3.7 CDC (Communications Device Class)**.

**Figure 3-14. `usbf850_init` Function**



```
void usbf850_init(void)
{
    INT32 i;
    UF0E0NA = C_EP0NKA;
    while (UF0E0NA != C_EP0NKA) {
       UF0E0NA = C_EP0NKA;
    }
    /* The initialization of the request data register area */
    UF0DSTL = 0x00;        /* Bus Powered */
    UF0E0SL = 0x00;
    UF0E1SL = 0x00;
    UF0E2SL = 0x00;
    /* The total byte of the UF0CIEa register is long. */
    UF0DSCL = (C_CONF_DSC_wTotalLength_L - 1);
    for (i = 0; i < T_DEV_DSC[0]; i++) {
       USBF850REG_SET((UF0DD0_ADDRESS + (i*sizeof(INT16))), T_DEV_DSC[i]);
    }
    for (i = 0; i < T_CONF_DSC[2]; i++) {
       USBF850REG_SET((UF0CIE0_ADDRESS + (i*sizeof(INT16))), T_CONF_DSC[i]);
    }
    /* The initialization of the request data register area (The ending) */
    UF0MODC = 0x00;     /* SET GET_DESCRIPTOR REQ. AUTO */

    /* The setting of Interface and Endpoint */
    UF0AIFN = 0x80;     /* Interface0,1 Support */
    UF0AAS = 0x00;      /* It is not in the Alternate setting. */

    /* SFR_UF0EnIM = xx// (It sets EP not to use to 0x00.) */
    UF0E1IM = 0x40;
    UF0E2IM = 0x40;
    UF0E7IM = 0x20;
    /* The setting of Interface and Endpoint (The ending) */
    UF0E0NA = 0x00;     /* RESET EP0 NAK SEND */   /* RESET EP0 NAK SEND */
    /* The interrupt and FIFO relation register initialization */
    UF0IC0 = C_IC0_ALL;     /* interrupt clear */
    UF0IC1 = C_IC1_ALL;     /* interrupt clear */
    UF0IC2 = C_IC2_ALL;     /* interrupt clear */
    UF0IC3 = C_IC3_ALL;     /* interrupt clear */
    UF0IC4 = C_IC4_ALL;     /* interrupt clear */

    UF0FIC0 = C_FIC0_ALL;   /* The FIFO clearness, the counter reset */
    UF0FIC1 = C_FIC1_ALL;   /* The FIFO clearness */

    /* The setting of a interrupt mask */
    UF0IM0 = C_IM0_ALL;                     /* ALL MASK */
    UF0IM1 = (C_IM1_ALL & (~C_CPUDEC));        /* CPUDEC mask clear */
    UF0IM2 = C_IM2_ALL;                     /* ALL Mask */
    UF0IM3 = (C_IM3_ALL & (~C_BKO1DT));        /* BKO1DT mask clear */
    UF0IM4 = C_IM4_ALL;                     /* ALL Mask */
    usbf850_setfunction_communication();
    /* D+ Pullup */
    PM4 = 0xFC;
    P4  = 0x02;
}
```

Returns NAK for all requests, including auto requests.

Initializes the registers storing request data.

Adds descriptor data and other data required to respond to the GetDescriptor request to registers.

Shows the number of supported interfaces, shows the status of alternative settings, sets the endpoint data to registers, and sets the endpoints.

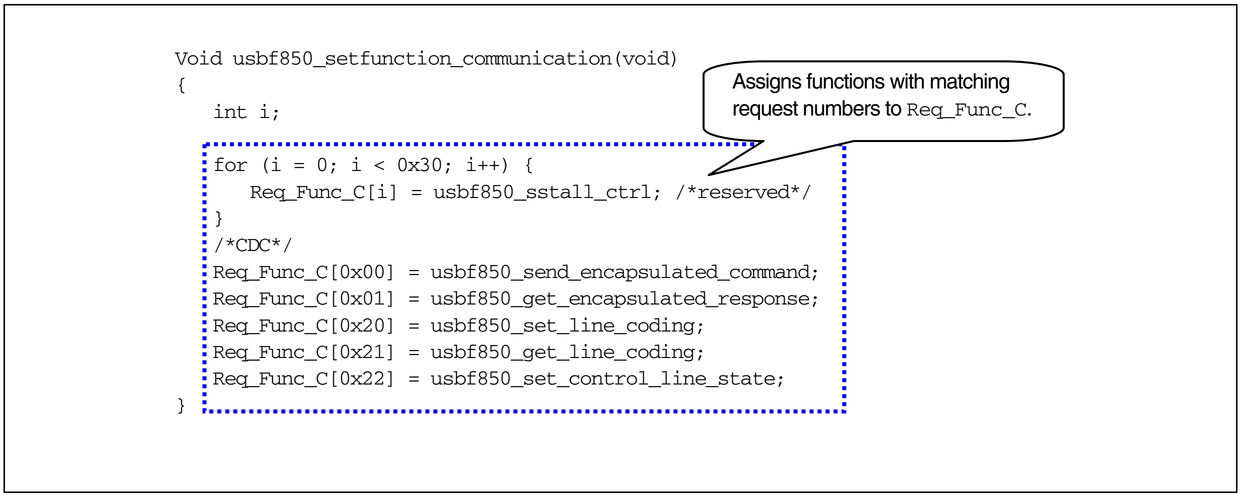
Disables the NAK setting.

Specifies the interrupt mask settings.

Adds the CDC requests.

Specifies pulling up the D+ signal.

**Figure 3-15. Adding CDC Requests**



```
Void usbf850_setfunction_communication(void)
{
   int i;

   for (i = 0; i < 0x30; i++) {
      Req_Func_C[i] = usbf850_sstall_ctrl; /*reserved*/
   }
   /*CDC*/
   Req_Func_C[0x00] = usbf850_send_encapsulated_command;
   Req_Func_C[0x01] = usbf850_get_encapsulated_response;
   Req_Func_C[0x20] = usbf850_set_line_coding;
   Req_Func_C[0x21] = usbf850_get_line_coding;
   Req_Func_C[0x22] = usbf850_set_control_line_state;
}
```

Assigns functions with matching request numbers to Req_Func_C.

For details about CDC requests, see **3.7 CDC (Communications Device Class)**.

## 3.5 Interrupt Processing

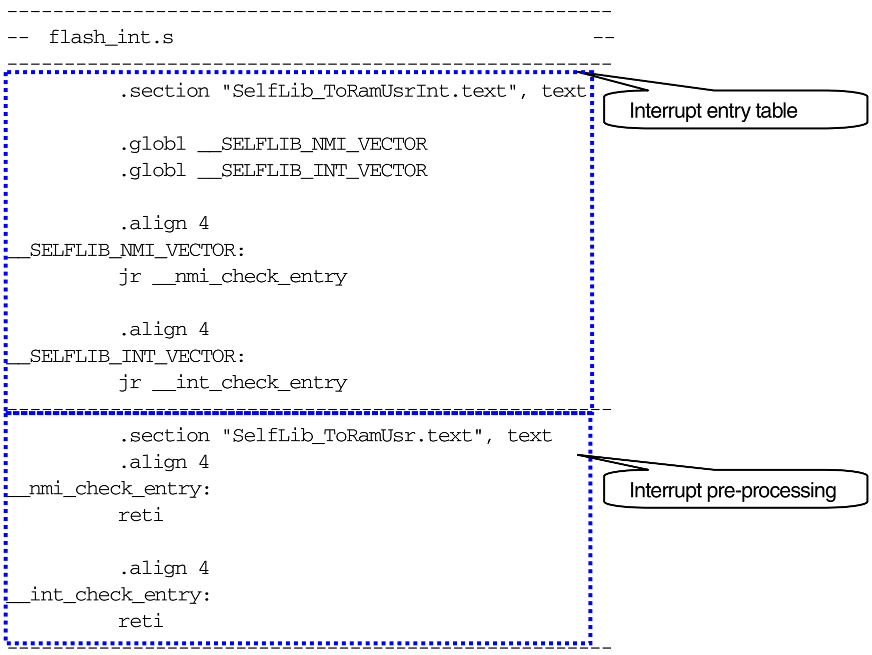
### 3.5.1 Interrupts in the flash environment (`usbf_fwup_intentry.s`)

*Flash environment* refers to a state in which the on-chip flash memory can be manipulated (written and erased). The flash environment can be entered and exited by calling the `FlashEnv` function from the self-programming library while the main routine is executing.

The on-chip flash memory cannot be referenced in the flash environment, so the occurrence of non-maskable interrupts will cause the program to jump to the top of the internal RAM, and the occurrence of maskable interrupts, software exceptions, and exception traps will cause the program to jump to the 4-byte area at the top of the RAM. The interrupt entry table in the flash environment is described in the `usbf_fwup_intentry.s` file.

With the firmware update program, however, interrupts are not used in the flash environment, so the interrupt processing described in this file does not occur. Note that, even if the self-programming library does not execute interrupt processing, the processing must still be specified in the library as a dummy section. The code in this dummy section is shown in Figure 3-16 below.

**Figure 3-16. Interrupts in the Flash Environment**



```
-----------------------------------------------------
--  flash_int.s                                    --
-----------------------------------------------------
          .section "SelfLib_ToRamUsrInt.text", text

          .globl __SELFLIB_NMI_VECTOR
          .globl __SELFLIB_INT_VECTOR

          .align 4
__SELFLIB_NMI_VECTOR:
          jr __nmi_check_entry

          .align 4
__SELFLIB_INT_VECTOR:
          jr __int_check_entry
-----------------------------------------------------
          .section "SelfLib_ToRamUsr.text", text
          .align 4
__nmi_check_entry:
          reti

          .align 4
__int_check_entry:
          reti
-----------------------------------------------------
```

## 3.6 Writing to the On-Chip Flash Memory

The firmware update program updates the firmware and specified memory areas by overwriting the contents of the on-chip flash memory.

The firmware update program uses the self-programming library to write data to the on-chip flash memory.

There are four types of self-programming libraries, Type 01 to Type 04, which correspond with the type of flash memory used. This evaluation board requires the Type 04 self-programming library.

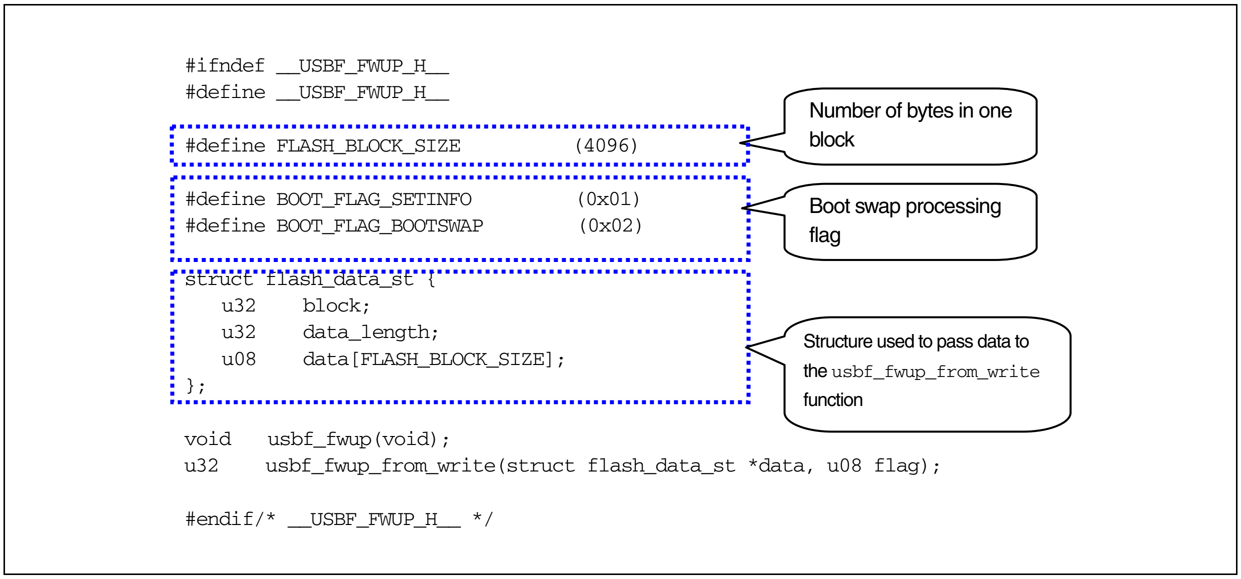
For details about the self-programming library, see the **V850 Microcontrollers Flash Memory Self-Programming Library Type 04 Ver. 1.20 User's Manual**.

### 3.6.1 Writing to the flash memory

The on-chip flash memory of the μPD70F3769 (V850ES/JH3-U) used by this evaluation board is made up of 128 blocks (blocks 0 to 127). The flash memory can be erased and written in block units.

The usbf_fwup_from_write function defined in the usbf_fwup.c file executes the processing to write to the specified block in the flash memory. The block to be written to and the data to be written are specified using the flash_data_st structure, which is declared in the usbf_fwup.h file.

**Figure 3-17. flash_data_st Structure**



```
#ifndef __USBF_FWUP_H__
#define __USBF_FWUP_H__

#define FLASH_BLOCK_SIZE          (4096)

#define BOOT_FLAG_SETINFO         (0x01)
#define BOOT_FLAG_BOOTSWAP        (0x02)

struct flash_data_st {
    u32    block;
    u32    data_length;
    u08    data[FLASH_BLOCK_SIZE];
};

void   usbf_fwup(void);
u32    usbf_fwup_from_write(struct flash_data_st *data, u08 flag);

#endif/* __USBF_FWUP_H__ */
```

Number of bytes in one block
Boot swap processing flag
Structure used to pass data to the usbf_fwup_from_write function

The block member in the flash_data_st structure specifies the number of the block to be written to and the data member specifies the data to be written. The data_length member specifies the number of bytes of data to be written.

The usbf_fwup_from_write function writes the data to the on-chip flash memory using the flash functions provided by the self-programming library.

**Figure 3-18. Writing to the On-Chip Flash Memory**

```
u32 usbf_fwup_from_write(struct flash_data_st *data, u08 flag)
{
    s32     *out;
    u32     ret;
    u32     info;
    u16     mask[6];

        mask[0] = IMR0;
        mask[1] = IMR1;
        mask[2] = IMR2;
        mask[3] = IMR3;
        mask[4] = IMR4;
        mask[5] = IMR5;
        IMR0 = 0xffff;
        IMR1 = 0xffff;
        IMR2 = 0xffff;
        IMR3 = 0xffff;
        IMR4 = 0xffff;
        IMR5 = 0xffff;

        /* FLMD0 to High */
        PM3.7 = 0;
        P3.7 = 1;

        /* Flash environment initialization */
        FlashEnv((u32)1);

        /* Status check of terminal FLMD */
        ret = FlashFLMDCheck();
        if (ret != SELFLIB_OK) {
                ret |= 0x00010000;
                goto end;
        }

        /* Get output address */
out = (s32 *)(data->block * FLASH_BLOCK_SIZE);

        /* Delete block */
        ret = FlashBlockErase(data->block, data->block);
        if (ret != SELFLIB_OK) {
                ret |= 0x00020000;
                goto end;
        }
        do {
                ret = FlashStatusCheck();
        } while (ret == SELFLIB_BUSY);
        if (ret != SELFLIB_OK) {
                ret |= 0x00030000;
                goto end;
        }

        /* Flash writing */
        info = (data->data_length + 7) / 8;
        info *= 2;
        ret = FlashWordWrite(out, data->data, info);
        if (ret != SELFLIB_OK) {
                ret |= 0x00040000;
                goto end;
        }
```

Saves the interrupt mask settings and masks all interrupts.

The program enters the flash environment.

Calculates the address of the block where data is to be written.

Erases the block where data is to be written.

Writes the specified data.

```
        /* Internal verify */
        ret = FlashBlockIVerify(data->block, data->block);
        if (ret != SELFLIB_OK) {
                ret |= 0x00050000;
                goto end;
        }
        do {
                ret = FlashStatusCheck();
        } while (ret == SELFLIB_BUSY);
        if (ret != SELFLIB_OK) {
                ret |= 0x00060000;
                goto end;
        }

        /* Specification boot swap */
        if (flag & BOOT_FLAG_SETINFO) {
                info = 0x1f00003e;
                ret = FlashGetInfo((u32)4);
                ret &= 0x00000001;
                info |= ret;
                ret = FlashSetInfo(info, (u32)0);
                if (ret != SELFLIB_OK) {
                        ret |= 0x00070000;
                        goto end;
                }
                do {
                        ret = FlashStatusCheck();
                } while (ret == SELFLIB_BUSY);
                if (ret != SELFLIB_OK) {
                        ret |= 0x00080000;
                        goto end;
                }
        }

        /* Execution boot swap */
        if (flag & BOOT_FLAG_BOOTSWAP) {
                ret = FlashBootSwap();
                if (ret != SELFLIB_OK) {
                        ret |= 0x000900
                        goto end;
                }
        }

        ret = 0;

end:
        /* Flash environment end */
        FlashEnv((u32)0);

        /* FLMD0 to Low */
        PM3.7 = 1;

        IMR0 = mask[0];
        IMR1 = mask[1];
        IMR2 = mask[2];
        IMR3 = mask[3];
        IMR4 = mask[4];
        IMR5 = mask[5];

        return ret;
}
```

Internally verifies the block where data was written.

Specifies the boot swap settings if the boot swap setting flag is set.

Executes boot swapping if the boot swap execution flag is set.

The program exits the flash environment.

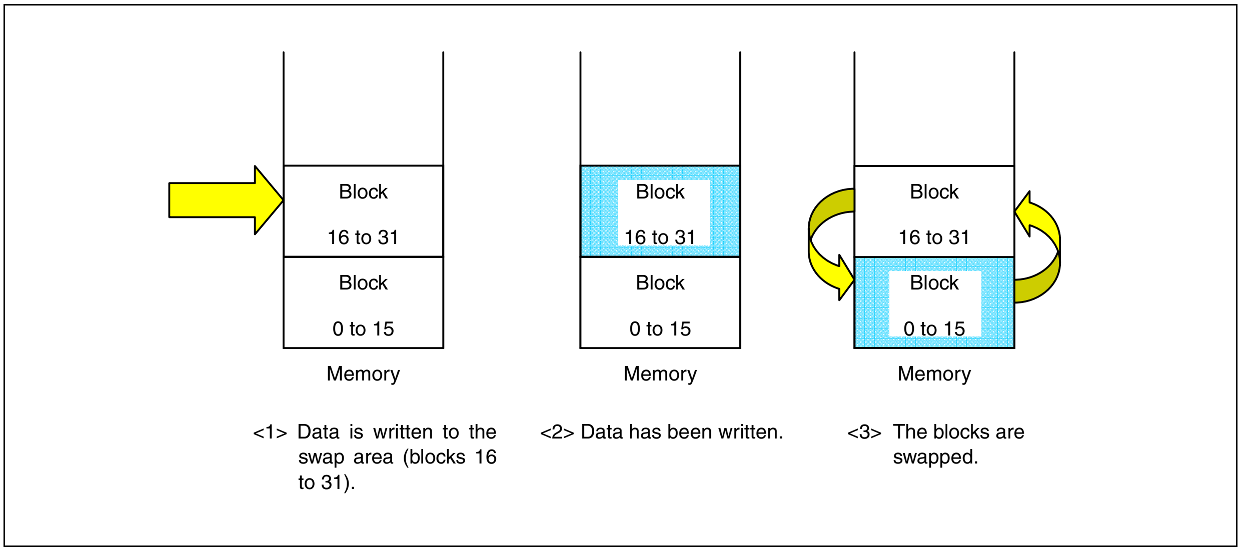
Restores the saved interrupt mask settings.

### 3.6.2 Boot swapping

The μPD70F3769 (V850ES/JH3-U) provides a boot swapping feature to protect the boot area and enable boot processing to be executed normally if the power supply is cut while the boot area is being overwritten during programming of the user-created program.

By using this feature, blocks 0 to 15 can be swapped with blocks 16 to 31 in the μPD70F3769 (V850ES/JH3-U).

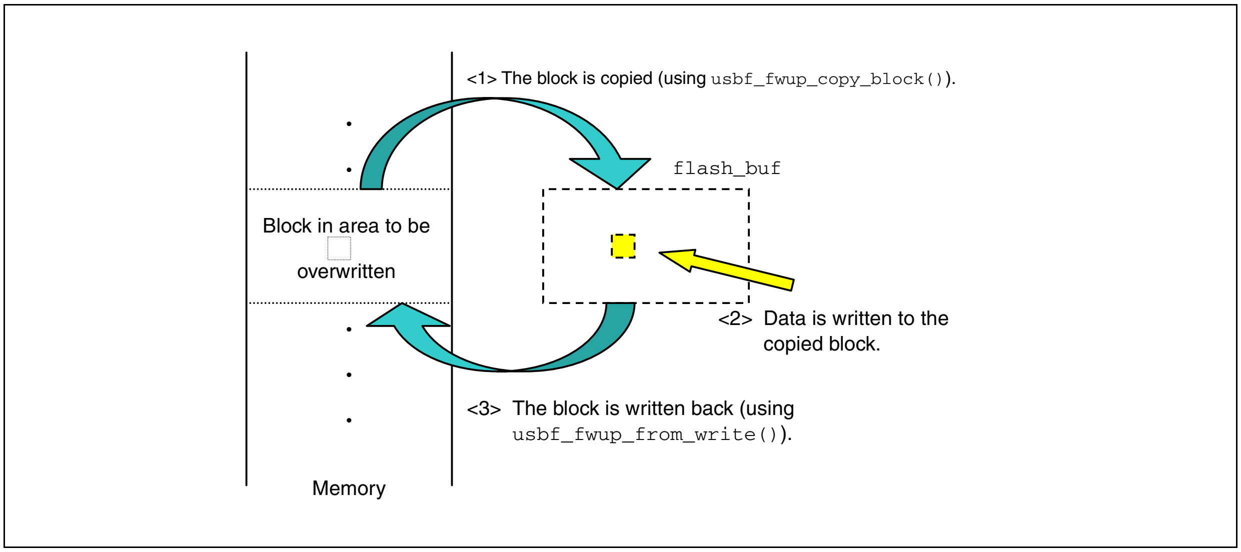
**Figure 3-19. Boot Swapping During Programming of the User-Created Program**


Block
16 to 31
Block
0 to 15
Memory
Block
16 to 31
Block
0 to 15
Memory
Block
16 to 31
Block
0 to 15
Memory
<1> Data is written to the swap area (blocks 16 to 31).
<2> Data has been written.
<3> The blocks are swapped.

### 3.6.3 Processing to update the firmware

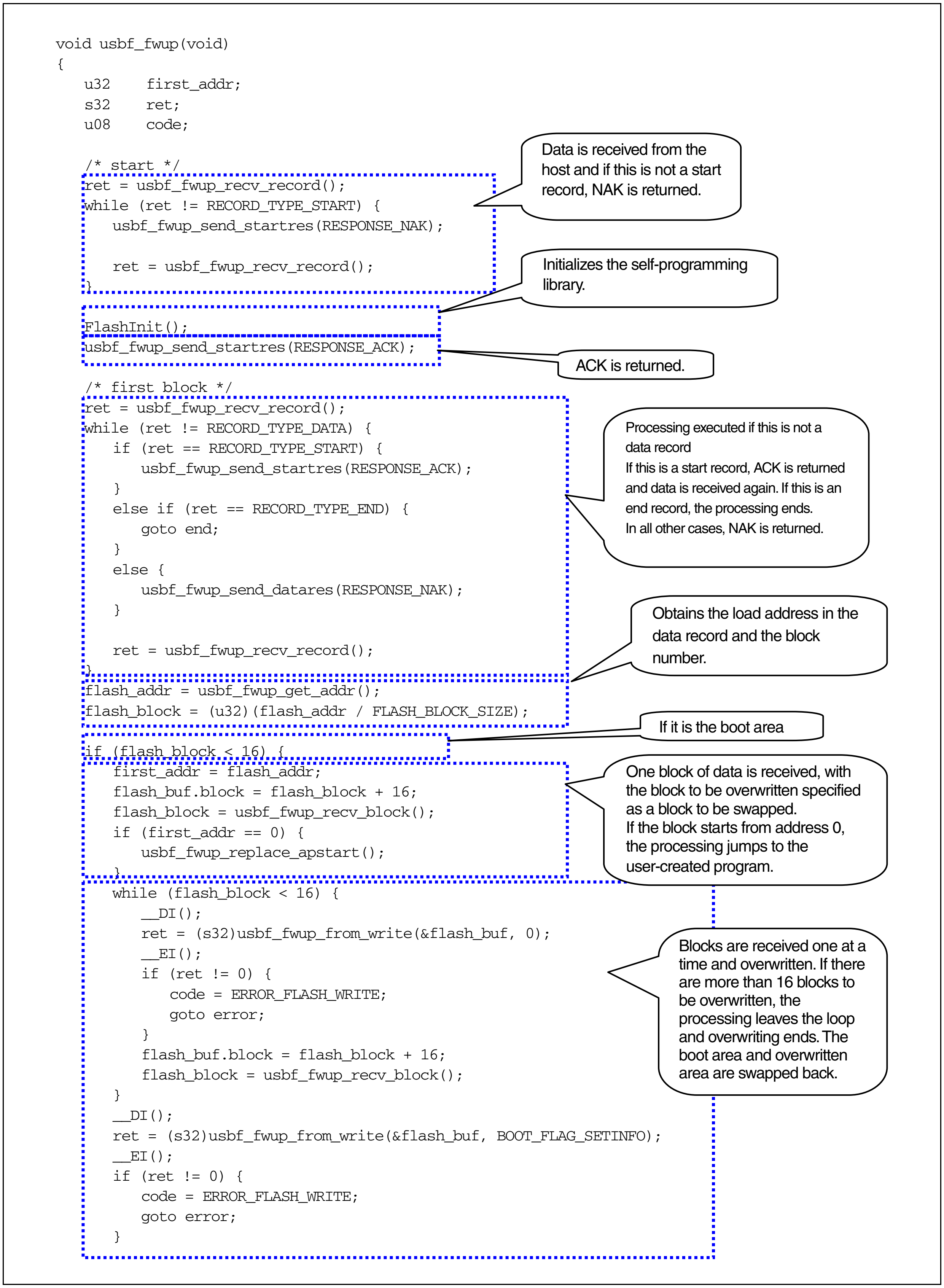
The on-chip flash memory is overwritten in block units. The firmware update program copies one of the blocks in the area to be overwritten to a buffer, overwrites the data in the block, and writes the block back to the on-chip flash memory. This means that the memory can be overwritten in 1-byte units.

**Figure 3-20. Diagram of Overwriting Blocks**


<1> The block is copied (using usbf_fwup_copy_block()).
flash_buf
Block in area to be overwritten
<2> Data is written to the copied block.
<3> The block is written back (using usbf_fwup_from_write()).
Memory

Data is received from the host, responses are transmitted to the host, and data is overwritten using the `usbf_fwup` function in the `usbf_fwup.c` file.

### Figure 3-21. Processing to Update the Firmware (1/2)



```
void usbf_fwup(void)
{
    u32     first_addr;
    s32     ret;
    u08     code;

    /* start */
    ret = usbf_fwup_recv_record();
    while (ret != RECORD_TYPE_START) {
        usbf_fwup_send_startres(RESPONSE_NAK);

        ret = usbf_fwup_recv_record();
    }

    FlashInit();
    usbf_fwup_send_startres(RESPONSE_ACK);

    /* first block */
    ret = usbf_fwup_recv_record();
    while (ret != RECORD_TYPE_DATA) {
        if (ret == RECORD_TYPE_START) {
            usbf_fwup_send_startres(RESPONSE_ACK);
        }
        else if (ret == RECORD_TYPE_END) {
            goto end;
        }
        else {
            usbf_fwup_send_datares(RESPONSE_NAK);
        }

        ret = usbf_fwup_recv_record();
    }
    flash_addr = usbf_fwup_get_addr();
    flash_block = (u32)(flash_addr / FLASH_BLOCK_SIZE);

    if (flash_block < 16) {
        first_addr = flash_addr;
        flash_buf.block = flash_block + 16;
        flash_block = usbf_fwup_recv_block();
        if (first_addr == 0) {
            usbf_fwup_replace_apstart();
        }
        while (flash_block < 16) {
            __DI();
            ret = (s32)usbf_fwup_from_write(&flash_buf, 0);
            __EI();
            if (ret != 0) {
                code = ERROR_FLASH_WRITE;
                goto error;
            }
            flash_buf.block = flash_block + 16;
            flash_block = usbf_fwup_recv_block();
        }
        __DI();
        ret = (s32)usbf_fwup_from_write(&flash_buf, BOOT_FLAG_SETINFO);
        __EI();
        if (ret != 0) {
            code = ERROR_FLASH_WRITE;
            goto error;
        }
```

Data is received from the host and if this is not a start record, NAK is returned.

Initializes the self-programming library.

ACK is returned.

Processing executed if this is not a data record
If this is a start record, ACK is returned and data is received again. If this is an end record, the processing ends.
In all other cases, NAK is returned.

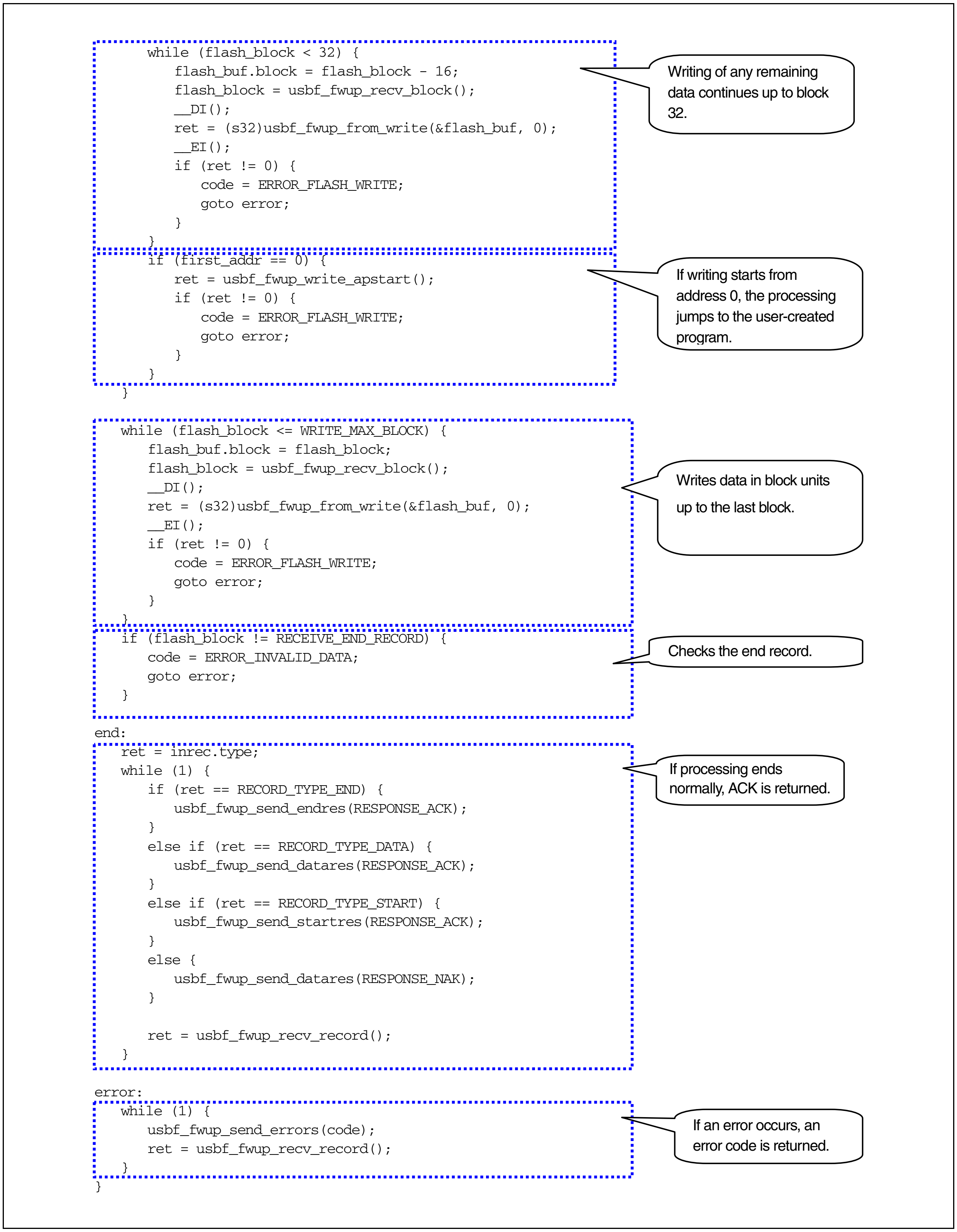
Obtains the load address in the data record and the block number.

If it is the boot area

One block of data is received, with the block to be overwritten specified as a block to be swapped.
If the block starts from address 0, the processing jumps to the user-created program.

Blocks are received one at a time and overwritten. If there are more than 16 blocks to be overwritten, the processing leaves the loop and overwriting ends. The boot area and overwritten area are swapped back.

### Figure 3-21. Processing to Update the Firmware (2/2)



```
        while (flash_block < 32) {
            flash_buf.block = flash_block - 16;
            flash_block = usbf_fwup_recv_block();
            __DI();
            ret = (s32)usbf_fwup_from_write(&flash_buf, 0);
            __EI();
            if (ret != 0) {
                code = ERROR_FLASH_WRITE;
                goto error;
            }
        }
        if (first_addr == 0) {
            ret = usbf_fwup_write_apstart();
            if (ret != 0) {
                code = ERROR_FLASH_WRITE;
                goto error;
            }
        }
    }

    while (flash_block <= WRITE_MAX_BLOCK) {
        flash_buf.block = flash_block;
        flash_block = usbf_fwup_recv_block();
        __DI();
        ret = (s32)usbf_fwup_from_write(&flash_buf, 0);
        __EI();
        if (ret != 0) {
            code = ERROR_FLASH_WRITE;
            goto error;
        }
    }
    if (flash_block != RECEIVE_END_RECORD) {
        code = ERROR_INVALID_DATA;
        goto error;
    }
end:
    ret = inrec.type;
    while (1) {
        if (ret == RECORD_TYPE_END) {
            usbf_fwup_send_endres(RESPONSE_ACK);
        }
        else if (ret == RECORD_TYPE_DATA) {
            usbf_fwup_send_datares(RESPONSE_ACK);
        }
        else if (ret == RECORD_TYPE_START) {
            usbf_fwup_send_startres(RESPONSE_ACK);
        }
        else {
            usbf_fwup_send_datares(RESPONSE_NAK);
        }

        ret = usbf_fwup_recv_record();
    }

error:
    while (1) {
        usbf_fwup_send_errors(code);
        ret = usbf_fwup_recv_record();
    }
}
```

Writing of any remaining data continues up to block 32.

If writing starts from address 0, the processing jumps to the user-created program.

Writes data in block units up to the last block.

Checks the end record.

If processing ends normally, ACK is returned.

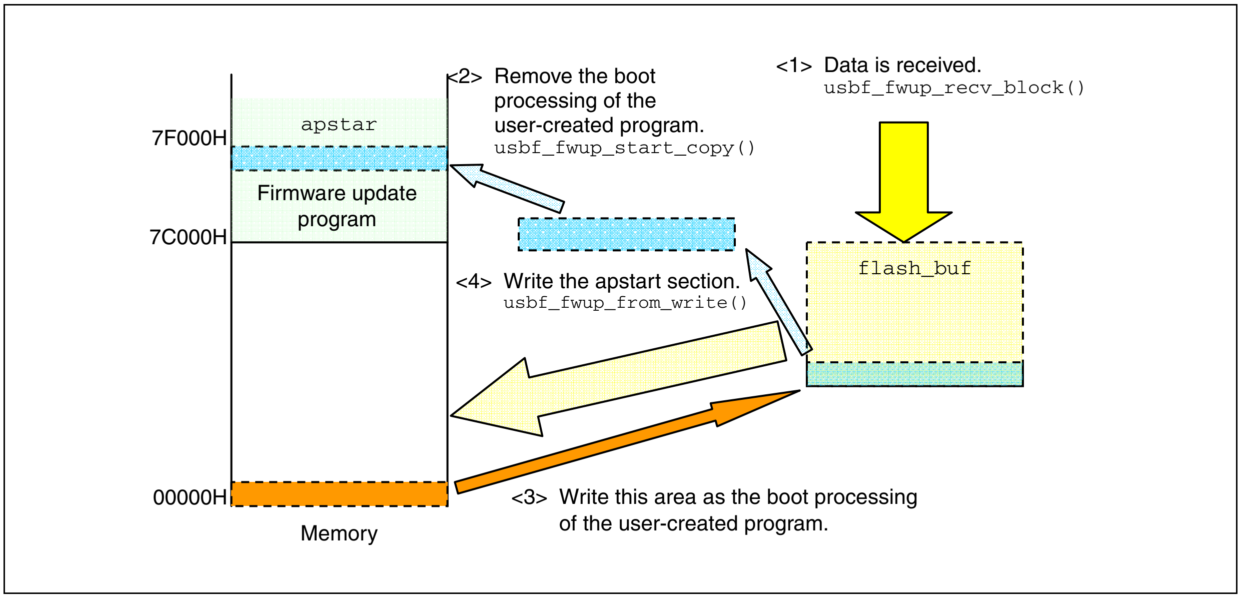
If an error occurs, an error code is returned.

### 3.6.4 Updating the user-created program

When writing the user-created program, change the `apstart` section as follows so that the user-created program runs when the system starts up.

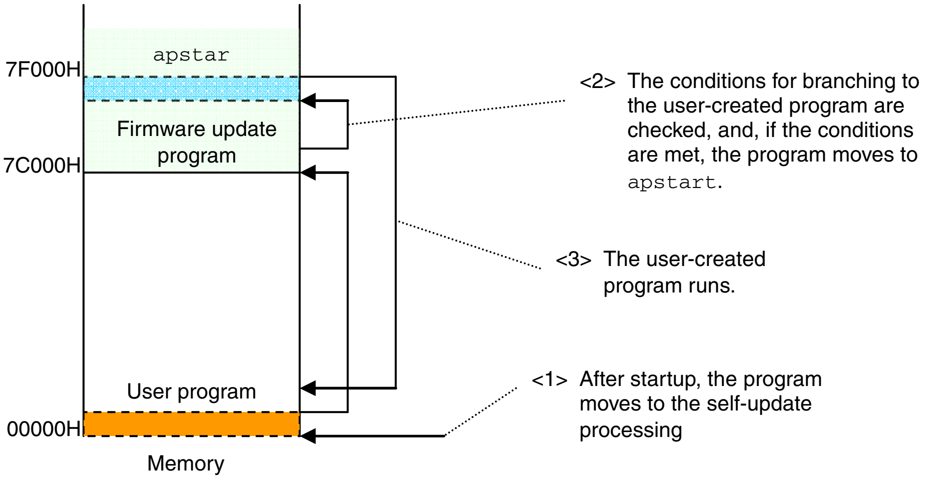
Remove the boot processing in the user-created program (the reset section) and write this as the branch destination of the firmware update program's boot processing. (The code to jump to the user-created program is in the `apstart` section.) By doing this, the boot processing of the user-created program is changed to the boot processing of the firmware update program. This means that the boot processing area can be preserved and the firmware update program can be manipulated again later.

**Figure 3-22. Overwriting the Boot Processing When Updating the User-Created Program**


apstar
7F000H
Firmware update program
7C000H
00000H
Memory
<2> Remove the boot processing of the user-created program.
usbf_fwup_start_copy()
<1> Data is received.
usbf_fwup_recv_block()
flash_buf
<4> Write the apstart section.
usbf_fwup_from_write()
<3> Write this area as the boot processing of the user-created program.

When the system starts up, the program moves to the boot processing of the firmware update program, checks the startup conditions in that processing (that is, the status of the SW3 and SW4 switches), moves to `apstart` as appropriate, and then moves to the start of the user-created program.

**Figure 3-23. Branching to the User-Created Program**

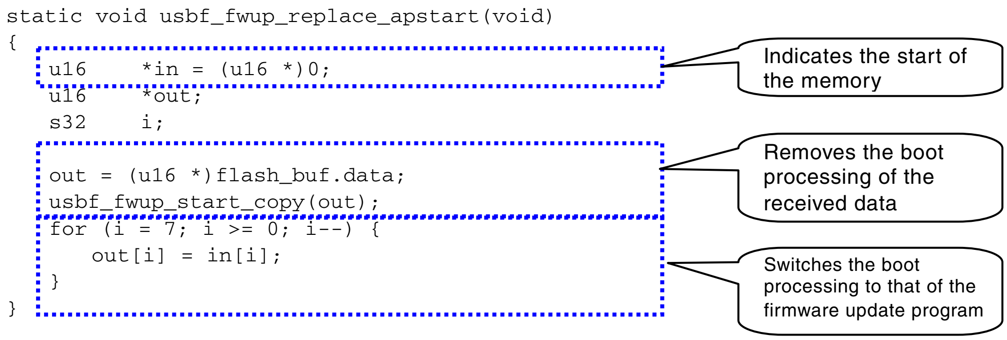


<2> The conditions for branching to the user-created program are checked, and, if the conditions are met, the program moves to `apstart`.

<3> The user-created program runs.

<1> After startup, the program moves to the self-update processing

**Figure 3-24. Switching the Boot Processing**



```
static void usbf_fwup_replace_apstart(void)
{
    u16     *in = (u16 *)0;
    u16     *out;
    s32     i;

    out = (u16 *)flash_buf.data;
    usbf_fwup_start_copy(out);
    for (i = 7; i >= 0; i--) {
        out[i] = in[i];
    }
}
```

Indicates the start of the memory
Removes the boot processing of the received data
Switches the boot processing to that of the firmware update program

Next, editing the processing for writing to `apstart` is described.

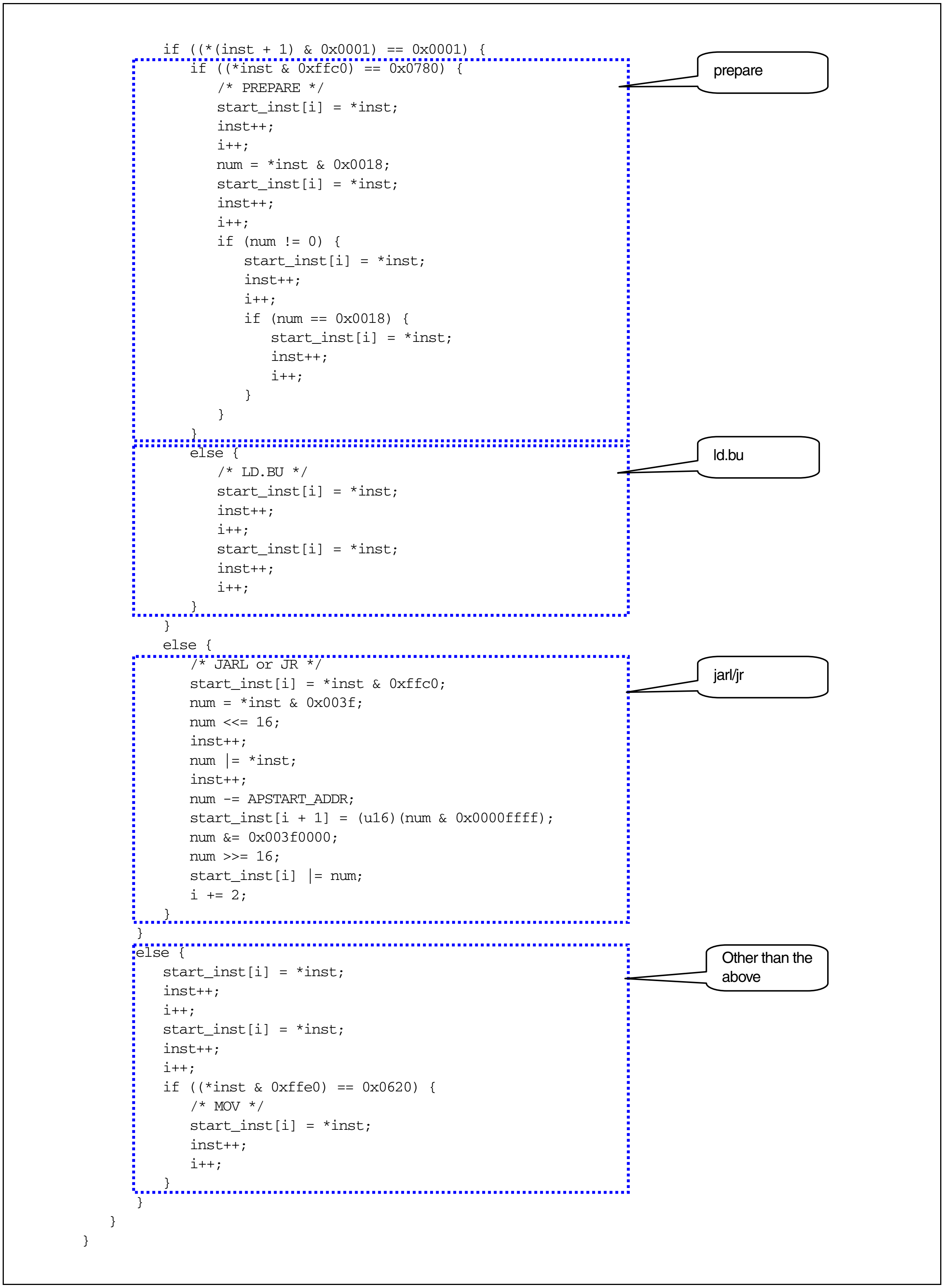
**Figure 3-25. Overwriting the Boot Processing (1/2)**

```
static void usbf_fwup_start_copy(u16 *inst)
{
   s32    num;
   s32    i;

   for (i = 23; i >= 0; i--) {
      start_inst[i] = 0xffff;
   }

   i = 0;
   while (i < 8) {
      if (*inst == 0xffff) {
         break;
      }
      if ((*inst & 0x0700) < 0x0600) {
         if ((*inst & 0x0780) == 0x0580) {
            /* Bcond */
            num = *inst & 0xf800;
            num >>= 4;
            num |= (*inst & 0x0070);
            num >>= 3;
            num -= APSTART_ADDR;
            if (num > 255 || num < -256) {
               usbf_fwup_add_jr(i, num);
               num = i * 2 + 16;
            }
            start_inst[i] = *inst & 0x078f;
            num <<= 3;
            start_inst[i] |= num & 0x00000070;
            num <<= 4;
            start_inst[i] |= num & 0x0000f800;
         }
         else {
            start_inst[i] = *inst;
         }
         inst++;
         i++;
      }
      else if ((*inst & 0x07c0) == 0x0780) {
```

Inserts the code to be written to `apstart` into `start_inst`. Initializes the elements of `start_inst` to 0xffff.

jcond/bcound

### Figure 3-25. Overwriting the Boot Processing (2/2)

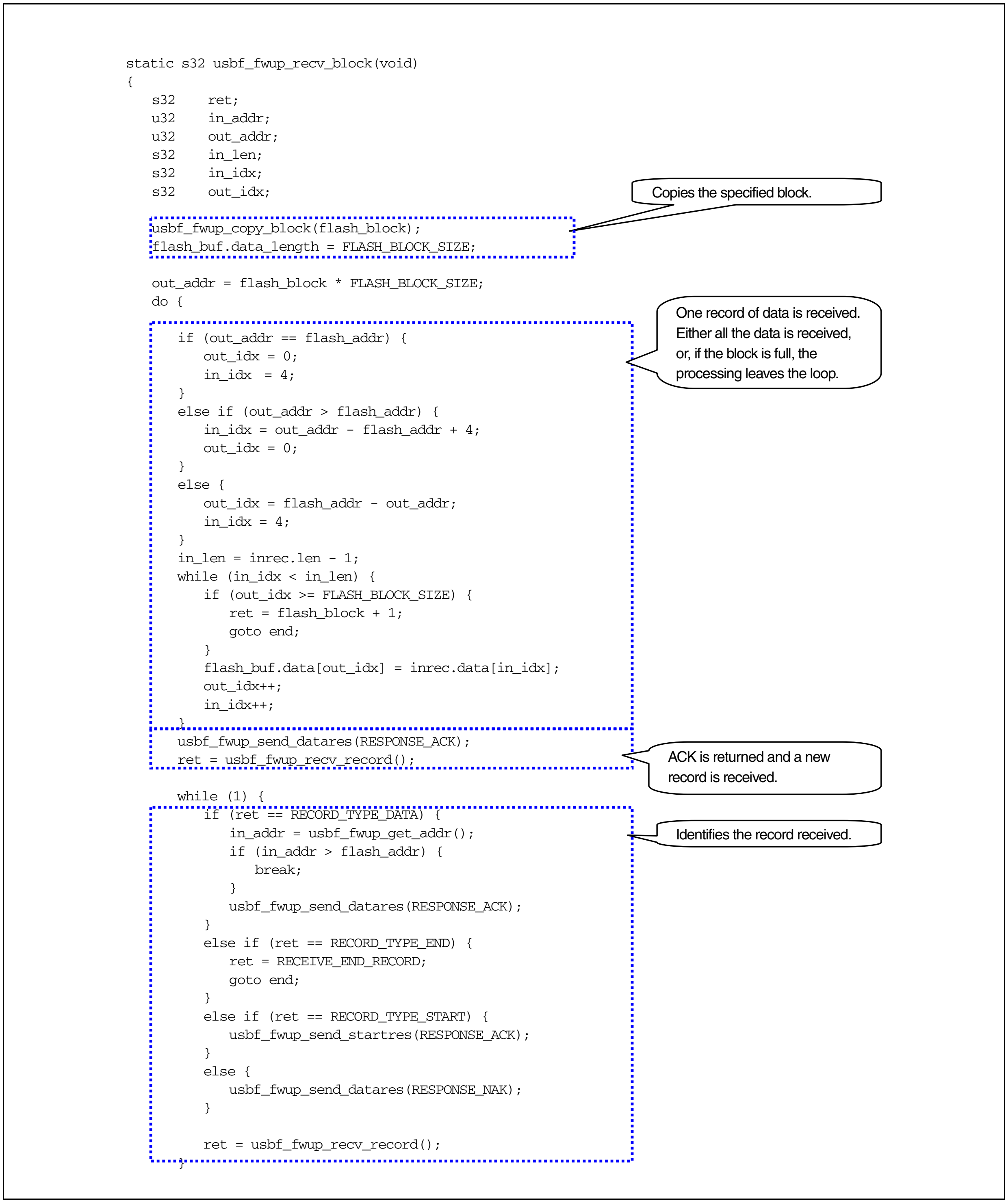


```
         if ((*(inst + 1) & 0x0001) == 0x0001) {
            if ((*inst & 0xffc0) == 0x0780) {
               /* PREPARE */
               start_inst[i] = *inst;
               inst++;
               i++;
               num = *inst & 0x0018;
               start_inst[i] = *inst;
               inst++;
               i++;
               if (num != 0) {
                  start_inst[i] = *inst;
                  inst++;
                  i++;
                  if (num == 0x0018) {
                     start_inst[i] = *inst;
                     inst++;
                     i++;
                  }
               }
            }
            else {
               /* LD.BU */
               start_inst[i] = *inst;
               inst++;
               i++;
               start_inst[i] = *inst;
               inst++;
               i++;
            }
         }
         else {
            /* JARL or JR */
            start_inst[i] = *inst & 0xffc0;
            num = *inst & 0x003f;
            num <<= 16;
            inst++;
            num |= *inst;
            inst++;
            num -= APSTART_ADDR;
            start_inst[i + 1] = (u16)(num & 0x0000ffff);
            num &= 0x003f0000;
            num >>= 16;
            start_inst[i] |= num;
            i += 2;
         }
      }
      else {
         start_inst[i] = *inst;
         inst++;
         i++;
         start_inst[i] = *inst;
         inst++;
         i++;
         if ((*inst & 0xffe0) == 0x0620) {
            /* MOV */
            start_inst[i] = *inst;
            inst++;
            i++;
         }
      }
   }
}
```

### 3.6.5 Receiving data

The firmware update program is used to initiate serial communication with the host and receive the new firmware data. For details about the communication interface specifications, see **7.1 Specifications of the Communication Interface for Updating the Firmware**.

**Figure 3-26. Receiving One Block of Data (1/2)**



```
static s32 usbf_fwup_recv_block(void)
{
   s32    ret;
   u32    in_addr;
   u32    out_addr;
   s32    in_len;
   s32    in_idx;
   s32    out_idx;

   usbf_fwup_copy_block(flash_block);
   flash_buf.data_length = FLASH_BLOCK_SIZE;

   out_addr = flash_block * FLASH_BLOCK_SIZE;
   do {
      if (out_addr == flash_addr) {
         out_idx = 0;
         in_idx  = 4;
      }
      else if (out_addr > flash_addr) {
         in_idx = out_addr - flash_addr + 4;
         out_idx = 0;
      }
      else {
         out_idx = flash_addr - out_addr;
         in_idx = 4;
      }
      in_len = inrec.len - 1;
      while (in_idx < in_len) {
         if (out_idx >= FLASH_BLOCK_SIZE) {
            ret = flash_block + 1;
            goto end;
         }
         flash_buf.data[out_idx] = inrec.data[in_idx];
         out_idx++;
         in_idx++;
      }
      usbf_fwup_send_datares(RESPONSE_ACK);
      ret = usbf_fwup_recv_record();

      while (1) {
         if (ret == RECORD_TYPE_DATA) {
            in_addr = usbf_fwup_get_addr();
            if (in_addr > flash_addr) {
               break;
            }
            usbf_fwup_send_datares(RESPONSE_ACK);
         }
         else if (ret == RECORD_TYPE_END) {
            ret = RECEIVE_END_RECORD;
            goto end;
         }
         else if (ret == RECORD_TYPE_START) {
            usbf_fwup_send_startres(RESPONSE_ACK);
         }
         else {
            usbf_fwup_send_datares(RESPONSE_NAK);
         }

         ret = usbf_fwup_recv_record();
      }
```

Copies the specified block.

One record of data is received. Either all the data is received, or, if the block is full, the processing leaves the loop.

ACK is returned and a new record is received.

Identifies the record received.

**Figure 3-26. Receiving One Block of Data (2/2)**

```
        flash_addr = in_addr;
        ret = flash_addr / FLASH_BLOCK_SIZE;

    } while (ret == flash_block);
end:
    return ret;
}
```

When the block changes, the processing leaves the loop.

The processing for receiving one record is shown below.

**Figure 3-27. Receiving One Record**

```
static s32 usbf_fwup_recv_record(void)
{
    s32    ret;
    s32    i;
    u16    chk;


    /* Read record */
    usbf_fwup_drvif_clear_buffer();
    ret = usbf_fwup_drvif_recv(&inrec.type, 1);
    ret = usbf_fwup_drvif_recv(&inrec.len, 1);
    if (inrec.len == 0) {
        ret = -1;
        goto end;
    }
    chk = inrec.len;
    if (inrec.len > 1) {
        ret = usbf_fwup_drvif_recv(inrec.data, inrec.len - 1);
        for (i = inrec.len - 2; i >= 0; i--) {
            chk += inrec.data[i];
        }
    }
    ret = usbf_fwup_drvif_recv(&inrec.sum, 1);

    /* Check sum */
    chk ^= 0xffff;
    chk &= 0x00ff;
    if (chk != inrec.sum) {
        ret = -1;
        goto end;
    }
    ret = inrec.type;

end:
    return ret;
}
```

Information on the record type and length is received.

The processing loops until the record reaches the specified length. The received data is accrued for checksum calculation.

Checksum

## 3.7 CDC (Communications Device Class)

This section describes the processing of the CDC (communications device class) used by the firmware update program.

For details about the USB communications device class (USB CDC), see the **Universal Serial Bus Class Definitions for Communication Devices**.

The CDC used by the firmware update program is an abstract control model and supports the following class requests.

**Remark** USB standards are formulated and managed by the USB Implementers Forum (USB-IF).
For details about the USB communications device class, see the **Universal Serial Bus Class Definitions for Communication Devices** on the official USB-IF website (www.usb.org).

**Table 3-6. Supported Class Requests**

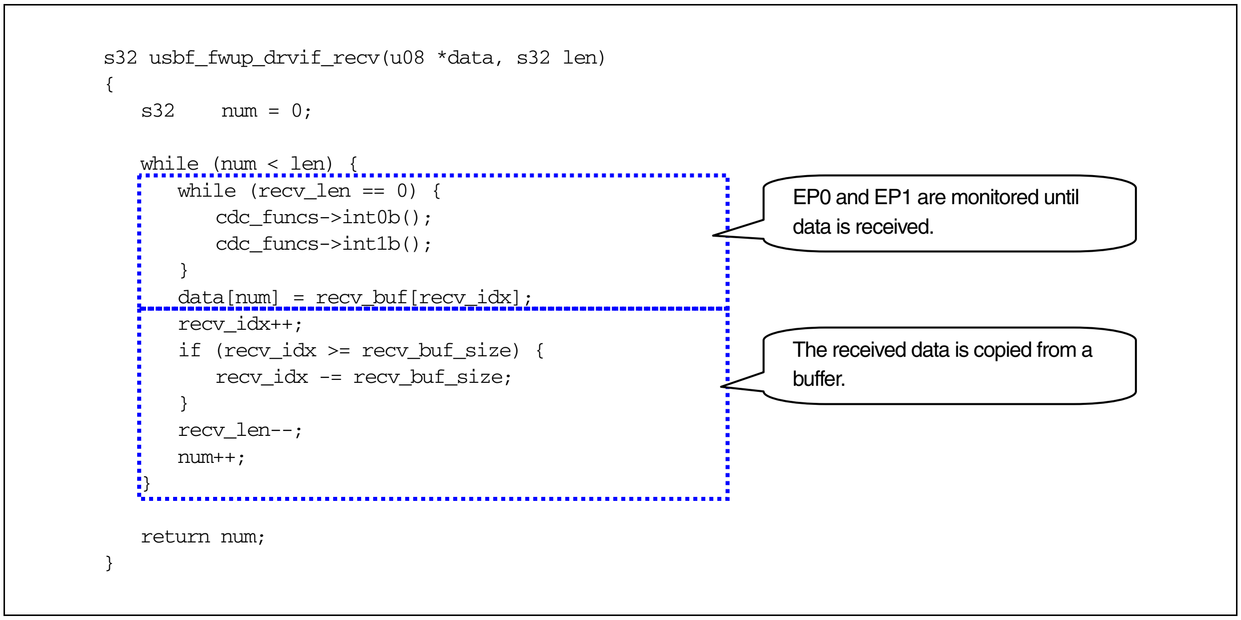
| Class Request | Description |
|---|---|
| `SendEncapsulatedCommand` | Request to issue a command in the format of the communications class interface control protocol |
| `GetEncapsulatedResponse` | Request to receive a response in the format of the communications class interface control protocol |
| `SetLineCoding` | Request to specify the serial communication format |
| `GetLineCoding` | Request to obtain the current communication format being used on the device side |
| `SetControlLineState` | Control signal transmitted in the RS-232/V.24 format |

### 3.7.1 Monitoring endpoints by polling

Endpoints are monitored by polling rather than by using interrupt vectors. The presence of data in the EP0 (endpoint for control transfers) and EP1 (endpoint for bulk-in transfers) FIFOs can be checked by monitoring the endpoint (EP) interrupt flags.

The processing for monitoring the endpoints when receiving data is shown below.

**Figure 3-28. Monitoring Endpoints When Receiving Data**



```
s32 usbf_fwup_drvif_recv(u08 *data, s32 len)
{
    s32     num = 0;

    while (num < len) {
        while (recv_len == 0) {
            cdc_funcs->int0b();
            cdc_funcs->int1b();
        }
        data[num] = recv_buf[recv_idx];
        recv_idx++;
        if (recv_idx >= recv_buf_size) {
            recv_idx -= recv_buf_size;
        }
        recv_len--;
        num++;
    }

    return num;
}
```

EP0 and EP1 are monitored until data is received.

The received data is copied from a buffer.

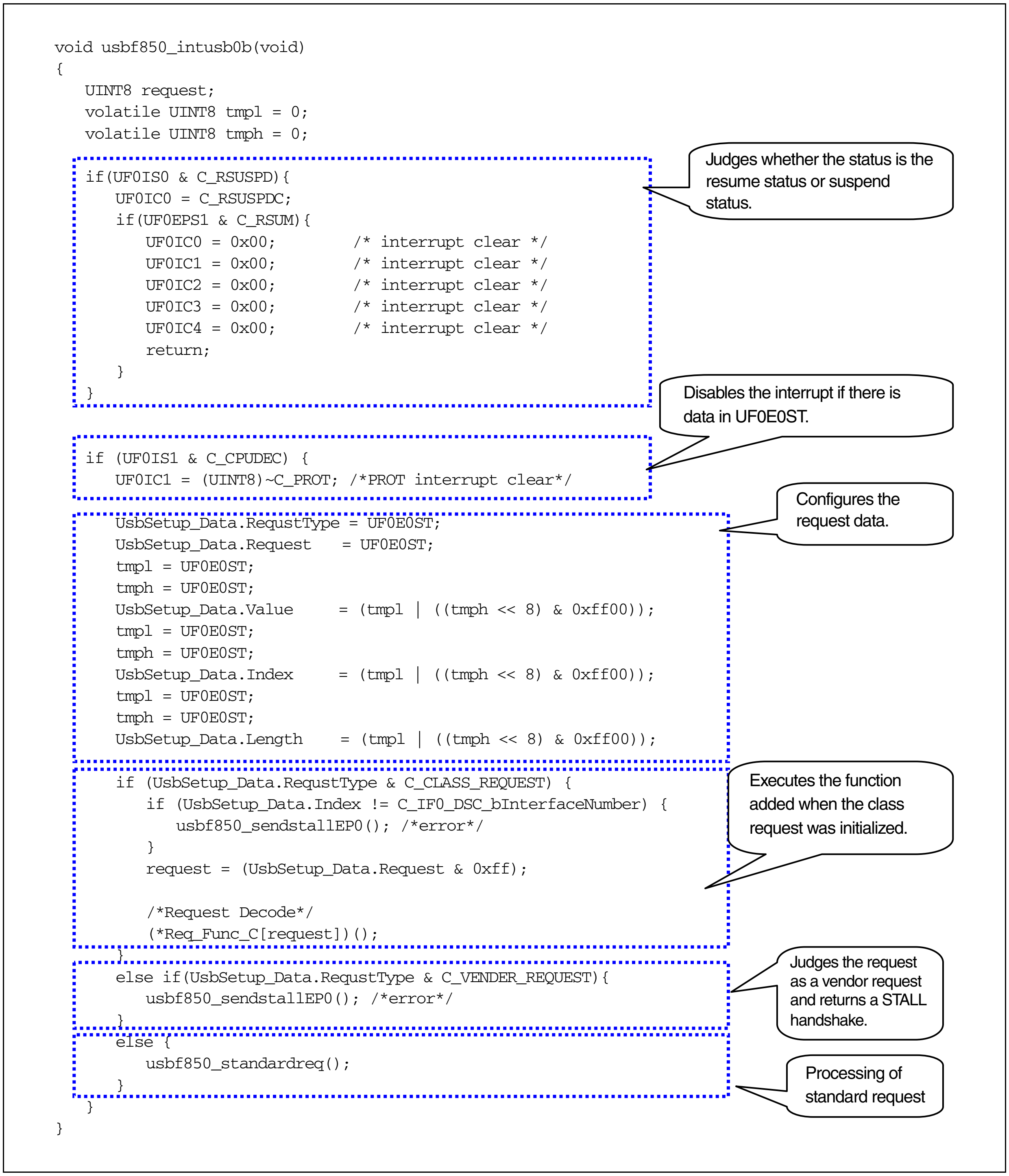
Executing `cdc_funcs->int0b();` in the function calls the `usbf850_intusb0b` function, according to the initial settings. Similarly, executing `cdc_funcs->int1b();` calls the `usbf850_intusb1b` function.

### 3.7.2 Monitoring EP0

EP0 is the endpoint for control transfers. EP0 is monitored to detect standard requests, class requests, and vendor requests that cannot be detected by the hardware.

The processing for monitoring EP0 is shown below.

**Figure 3-29. Monitoring EP0**



```
void usbf850_intusb0b(void)
{
    UINT8 request;
    volatile UINT8 tmpl = 0;
    volatile UINT8 tmph = 0;

    if(UF0IS0 & C_RSUSPD){
        UF0IC0 = C_RSUSPDC;
        if(UF0EPS1 & C_RSUM){
            UF0IC0 = 0x00;          /* interrupt clear */
            UF0IC1 = 0x00;          /* interrupt clear */
            UF0IC2 = 0x00;          /* interrupt clear */
            UF0IC3 = 0x00;          /* interrupt clear */
            UF0IC4 = 0x00;          /* interrupt clear */
            return;
        }
    }

    if (UF0IS1 & C_CPUDEC) {
        UF0IC1 = (UINT8)~C_PROT; /*PROT interrupt clear*/
        UsbSetup_Data.RequstType = UF0E0ST;
        UsbSetup_Data.Request    = UF0E0ST;
        tmpl = UF0E0ST;
        tmph = UF0E0ST;
        UsbSetup_Data.Value     = (tmpl | ((tmph << 8) & 0xff00));
        tmpl = UF0E0ST;
        tmph = UF0E0ST;
        UsbSetup_Data.Index     = (tmpl | ((tmph << 8) & 0xff00));
        tmpl = UF0E0ST;
        tmph = UF0E0ST;
        UsbSetup_Data.Length    = (tmpl | ((tmph << 8) & 0xff00));
        if (UsbSetup_Data.RequstType & C_CLASS_REQUEST) {
            if (UsbSetup_Data.Index != C_IF0_DSC_bInterfaceNumber) {
                usbf850_sendstallEP0(); /*error*/
            }
            request = (UsbSetup_Data.Request & 0xff);

            /*Request Decode*/
            (*Req_Func_C[request])();
        }
        else if(UsbSetup_Data.RequstType & C_VENDER_REQUEST){
            usbf850_sendstallEP0(); /*error*/
        }
        else {
            usbf850_standardreq();
        }
    }
}
```

Judges whether the status is the resume status or suspend status.
Disables the interrupt if there is data in UF0E0ST.
Configures the request data.
Executes the function added when the class request was initialized.
Judges the request as a vendor request and returns a STALL handshake.
Processing of standard request

### (1) Standard requests

Standard requests are used to obtain descriptors.

**Figure 3-30. Processing a Standard Request**

```
void usbf850_standardreq(void)
{
   if (UsbSetup_Data.Request == GETDESC) { /*GetDescriptor[String/Class]*/
      usbf850_getdesc();
   }
   else { /*error*/
      usbf850_sendstallEP0();
   }
}
```

If the request is for a descriptor, the descriptor is returned. In all other cases, a STALL handshake is returned.

**Figure 3-31. Transmitting a Descriptor**

```
void usbf850_getdesc(void)
{
   UINT8 len;
   UINT8 value;
   UINT8* tmp;

   if ((UsbSetup_Data.Value & 0xff00) == STRDESC) { /*String Descriptor*/
      value = (UINT8)(UsbSetup_Data.Value & 0xff);
      if (value >= (sizeof(USB_strings)/sizeof(USB_strings[0]))) {
         /*EP0 STALL*/
         usbf850_sendstallEP0();
         return ;
      }
      len = USB_strings[value][0];
      tmp = &(USB_strings[value][0]);
   }
   else {
      /*error*/
      usbf850_sendstallEP0();
      return ;
   }
   if (UsbSetup_Data.Length < len) {
      len = UsbSetup_Data.Length;
   }
   usbf850_data_send(tmp,len,C_EP0);
}
```

Sets a string descriptor.

Obtains the string length.

Transmits the descriptor from EP0.

The descriptor data (`USB_string`) is defined below. DSTR and USTR are macros for specifying the locale and Unicode settings.

**Figure 3-35. Definition of Descriptor Data**

```
/* 0 : Language Code*/
DSTR(LangString, 2, (0x09,0x04));
/* 1 : Manufacturer*/
USTR(ManString, 19, ('N','E','C',' ','E','l','e','c','t','r','o','n','i','c','s',' ','C','o','.'));
/* 2 : Product*/
USTR(ProductString, 10, ('U','S','B',' ','C','o','m','D','r','v'));
/* 3 : Serial Number*/
USTR(SerialString,  10, ('0','_','9','8','7','6','5','4','3','2'));

unsigned char *USB_strings[]={LangString,ManString,ProductString,SerialString};
```

### (2) Class requests

The class requests in the `Req_Func_C` file are listed in the table below. The issuance of each request causes the corresponding function to be executed.

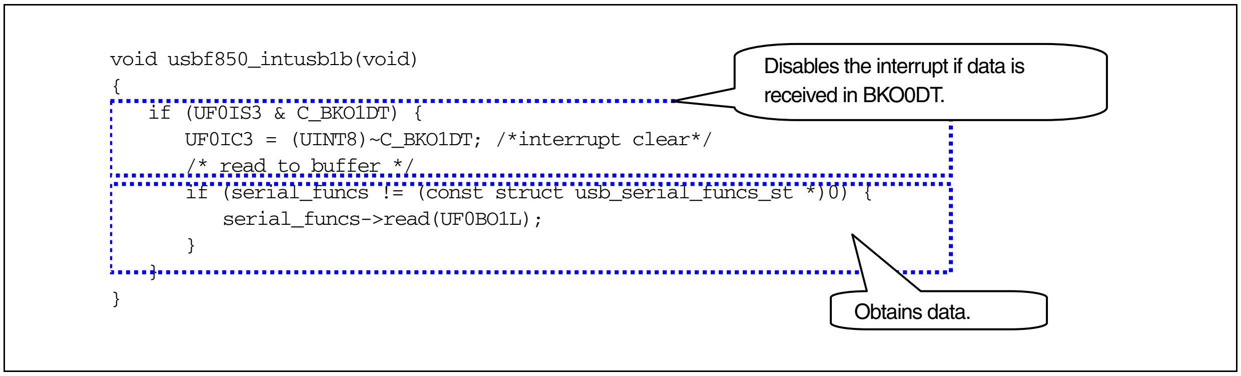
**Table 3-7. Class Requests**

| Function Name | Corresponding Request and Processing |
|---|---|
| `usbf850_send_encapsulated_command` | `SendEncapsulatedCommand`<br>Data is received from EP0. |
| `usbf850_get_encapsulated_response` | `GetEncapsulatedResponse`<br>No processing occurs. |
| `usbf850_set_line_coding` | `SetLineCoding`<br>Data for specifying the UART communication settings is received in EP0.<br>Processing to transmit the EP0NULL packet is executed. |
| `usbf850_get_line_coding` | `GetLineCoding`<br>Data for specifying the UART communication settings is transmitted from EP0. |
| `usbf850_set_control_line_state` | `SetControlLineState`<br>Processing to transmit the EP0NULL packet is executed. |
| `usbf850_sstall_ctrl` | STALL processing is executed. |

### 3.7.3 Monitoring EP1

The processing for monitoring EP1 is shown below.

**Figure 3-32. Monitoring EP1**



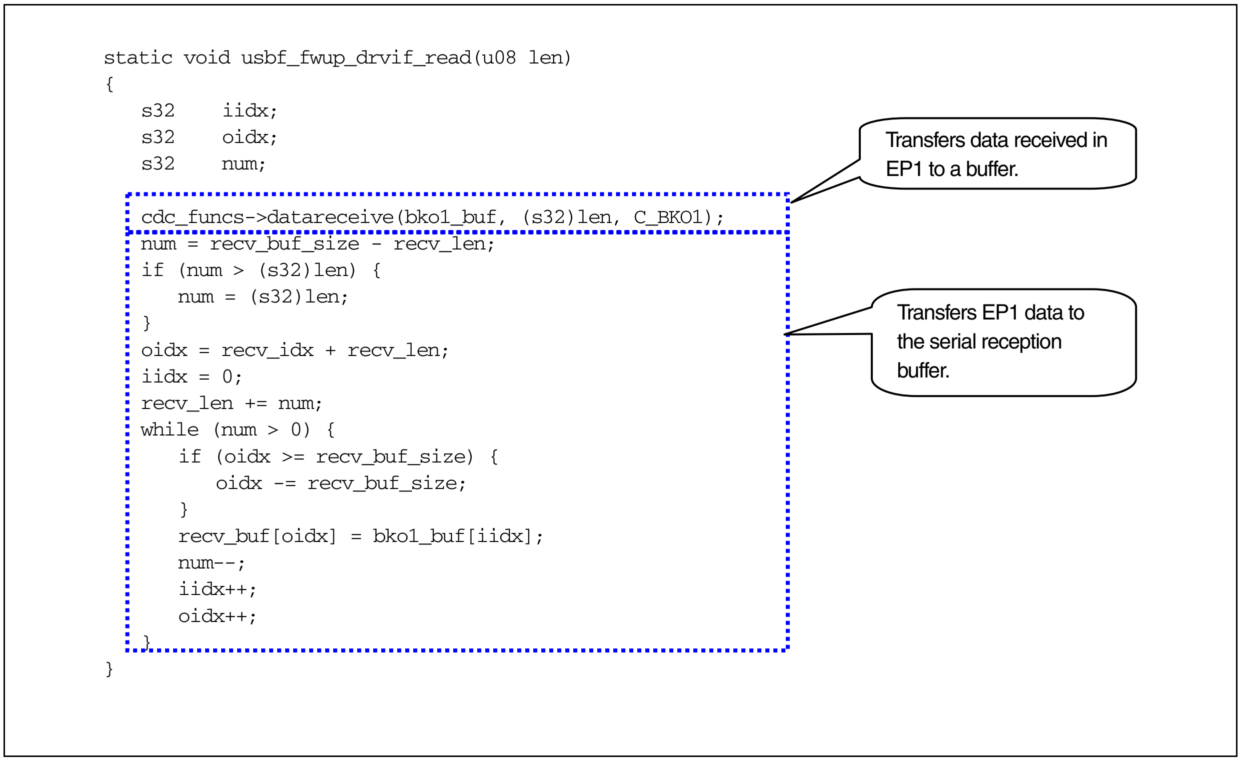
```
void usbf850_intusb1b(void)
{
    if (UF0IS3 & C_BKO1DT) {
        UF0IC3 = (UINT8)~C_BKO1DT; /*interrupt clear*/
        /* read to buffer */
        if (serial_funcs != (const struct usb_serial_funcs_st *)0) {
            serial_funcs->read(UF0BO1L);
        }
    }
}
```

Disables the interrupt if data is received in BKO0DT.

Obtains data.

Executing `serial_funcs->read(UF0BO1L);` in the function calls the `usb_fwup_drvif_read` function, according to the initial settings.

**Figure 3-33. Receiving Data in EP1**



```
static void usbf_fwup_drvif_read(u08 len)
{
   s32    iidx;
   s32    oidx;
   s32    num;

   cdc_funcs->datareceive(bko1_buf, (s32)len, C_BKO1);
   num = recv_buf_size - recv_len;
   if (num > (s32)len) {
      num = (s32)len;
   }
   oidx = recv_idx + recv_len;
   iidx = 0;
   recv_len += num;
   while (num > 0) {
      if (oidx >= recv_buf_size) {
         oidx -= recv_buf_size;
      }
      recv_buf[oidx] = bko1_buf[iidx];
      num--;
      iidx++;
      oidx++;
   }
}
```

Transfers data received in EP1 to a buffer.

Transfers EP1 data to the serial reception buffer.

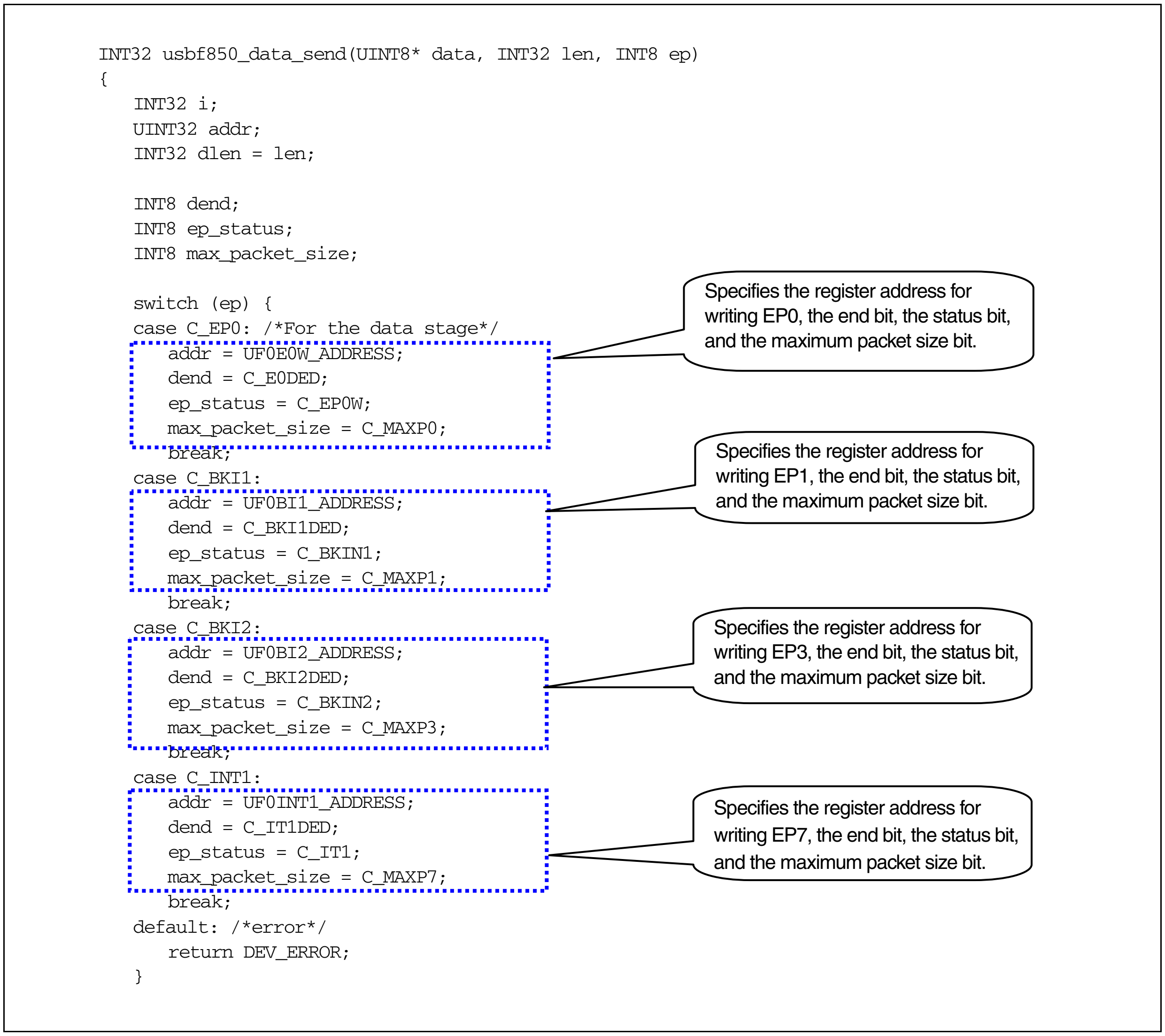
Executing `cdc_funcs->datareceive(bko0_buf, IINT32)len, C_BKO1);` in the function calls the `usbf850_data_receive` function, according to the initial settings.

### 3.7.4 Transmitting and receiving USB data

The processing for transmitting and receiving USB data, transmitting NULL packets, and returning a STALL handshake is shown below.

#### (1) Transmitting USB data

**Figure 3-34. Transmitting Data (1/2)**



```
INT32 usbf850_data_send(UINT8* data, INT32 len, INT8 ep)
{
    INT32 i;
    UINT32 addr;
    INT32 dlen = len;

    INT8 dend;
    INT8 ep_status;
    INT8 max_packet_size;

    switch (ep) {
    case C_EP0: /*For the data stage*/
        addr = UF0E0W_ADDRESS;
        dend = C_E0DED;
        ep_status = C_EP0W;
        max_packet_size = C_MAXP0;
        break;
    case C_BKI1:
        addr = UF0BI1_ADDRESS;
        dend = C_BKI1DED;
        ep_status = C_BKIN1;
        max_packet_size = C_MAXP1;
        break;
    case C_BKI2:
        addr = UF0BI2_ADDRESS;
        dend = C_BKI2DED;
        ep_status = C_BKIN2;
        max_packet_size = C_MAXP3;
        break;
    case C_INT1:
        addr = UF0INT1_ADDRESS;
        dend = C_IT1DED;
        ep_status = C_IT1;
        max_packet_size = C_MAXP7;
        break;
    default: /*error*/
        return DEV_ERROR;
    }
```

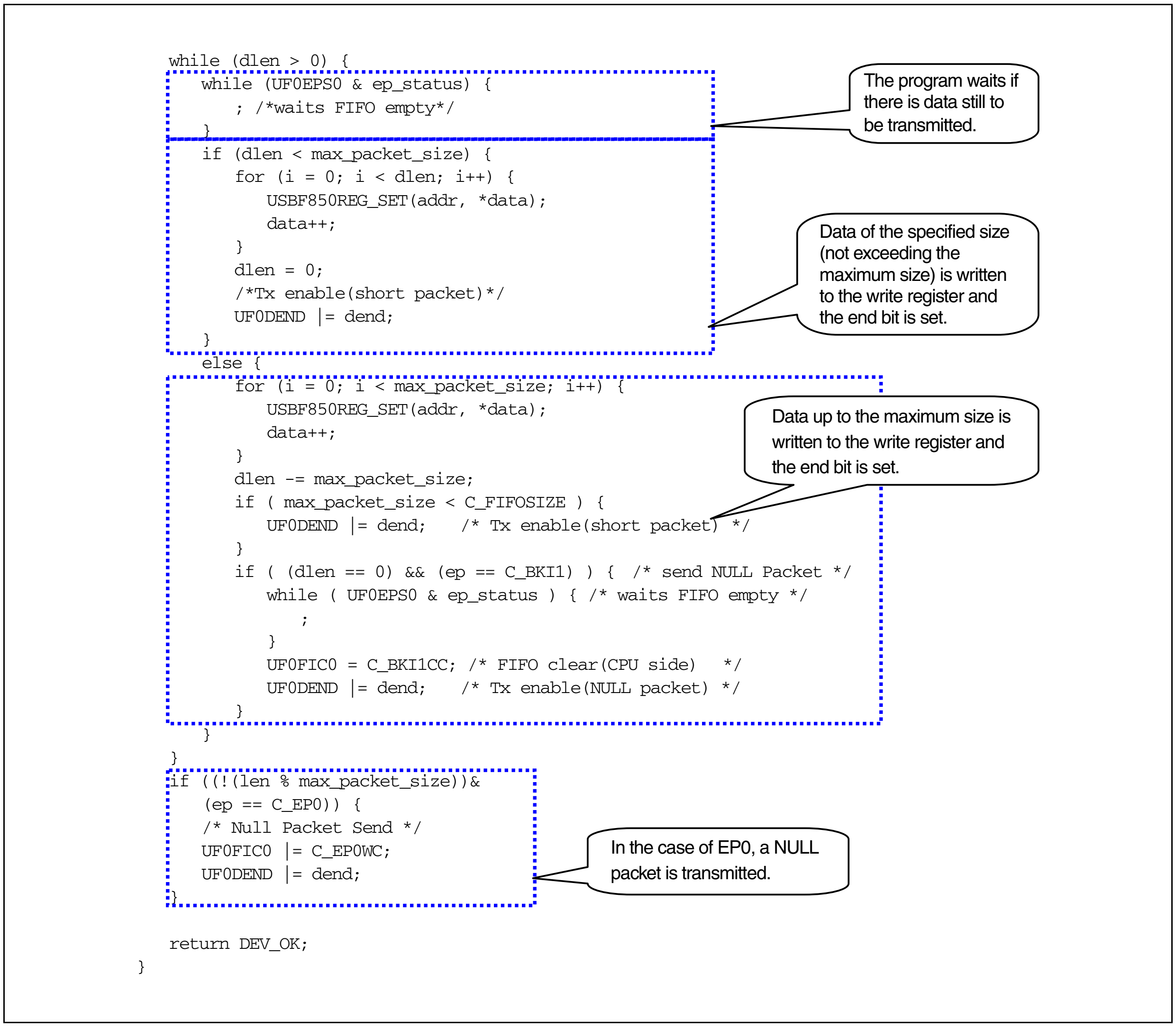
Specifies the register address for writing EP0, the end bit, the status bit, and the maximum packet size bit.

Specifies the register address for writing EP1, the end bit, the status bit, and the maximum packet size bit.

Specifies the register address for writing EP3, the end bit, the status bit, and the maximum packet size bit.

Specifies the register address for writing EP7, the end bit, the status bit, and the maximum packet size bit.

**Figure 3-34. Transmitting Data (2/2)**



```
    while (dlen > 0) {
        while (UF0EPS0 & ep_status) {
            ; /*waits FIFO empty*/
        }
        if (dlen < max_packet_size) {
            for (i = 0; i < dlen; i++) {
                USBF850REG_SET(addr, *data);
                data++;
            }
            dlen = 0;
            /*Tx enable(short packet)*/
            UF0DEND |= dend;
        }
        else {
            for (i = 0; i < max_packet_size; i++) {
                USBF850REG_SET(addr, *data);
                data++;
            }
            dlen -= max_packet_size;
            if ( max_packet_size < C_FIFOSIZE ) {
                UF0DEND |= dend;    /* Tx enable(short packet) */
            }
            if ( (dlen == 0) && (ep == C_BKI1) ) {  /* send NULL Packet */
                while ( UF0EPS0 & ep_status ) { /* waits FIFO empty */
                    ;
                }
                UF0FIC0 = C_BKI1CC; /* FIFO clear(CPU side)   */
                UF0DEND |= dend;    /* Tx enable(NULL packet) */
            }
        }
    }
    if ((!(len % max_packet_size))&
        (ep == C_EP0)) {
        /* Null Packet Send */
        UF0FIC0 |= C_EP0WC;
        UF0DEND |= dend;
    }

    return DEV_OK;
}
```

The program waits if there is data still to be transmitted.

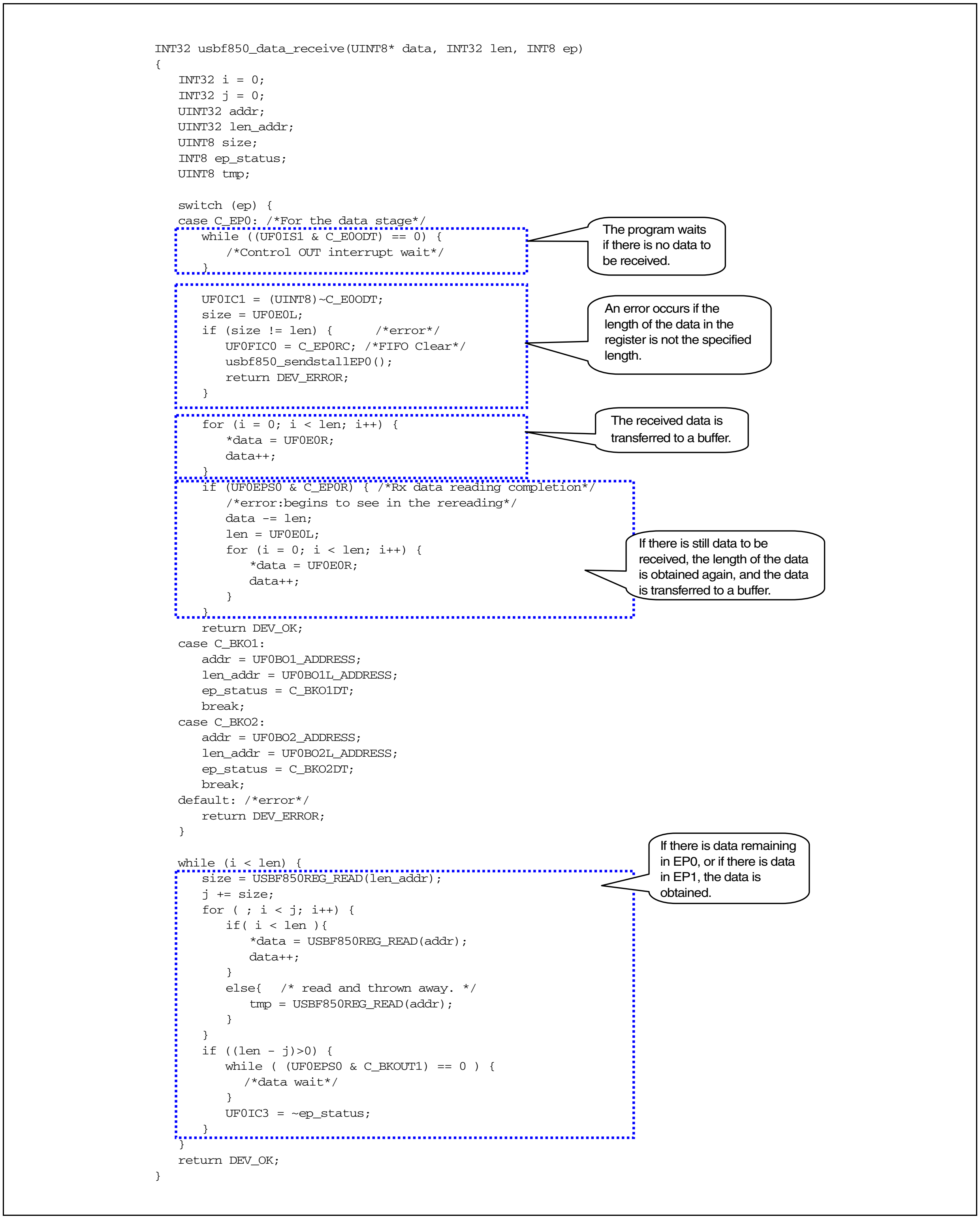
Data of the specified size (not exceeding the maximum size) is written to the write register and the end bit is set.

Data up to the maximum size is written to the write register and the end bit is set.

In the case of EP0, a NULL packet is transmitted.

### (2) Receiving USB data

**Figure 3-35. Receiving Data**



```
INT32 usbf850_data_receive(UINT8* data, INT32 len, INT8 ep)
{
   INT32 i = 0;
   INT32 j = 0;
   UINT32 addr;
   UINT32 len_addr;
   UINT8 size;
   INT8 ep_status;
   UINT8 tmp;

   switch (ep) {
   case C_EP0: /*For the data stage*/
      while ((UF0IS1 & C_E0ODT) == 0) {
         /*Control OUT interrupt wait*/
      }

      UF0IC1 = (UINT8)~C_E0ODT;
      size = UF0E0L;
      if (size != len) {      /*error*/
         UF0FIC0 = C_EP0RC; /*FIFO Clear*/
         usbf850_sendstallEP0();
         return DEV_ERROR;
      }
      for (i = 0; i < len; i++) {
         *data = UF0E0R;
         data++;
      }
      if (UF0EPS0 & C_EP0R) { /*Rx data reading completion*/
         /*error:begins to see in the rereading*/
         data -= len;
         len = UF0E0L;
         for (i = 0; i < len; i++) {
            *data = UF0E0R;
            data++;
         }
      }
      return DEV_OK;
   case C_BKO1:
      addr = UF0BO1_ADDRESS;
      len_addr = UF0BO1L_ADDRESS;
      ep_status = C_BKO1DT;
      break;
   case C_BKO2:
      addr = UF0BO2_ADDRESS;
      len_addr = UF0BO2L_ADDRESS;
      ep_status = C_BKO2DT;
      break;
   default: /*error*/
      return DEV_ERROR;
   }

   while (i < len) {
      size = USBF850REG_READ(len_addr);
      j += size;
      for ( ; i < j; i++) {
         if( i < len ){
            *data = USBF850REG_READ(addr);
            data++;
         }
         else{   /* read and thrown away. */
            tmp = USBF850REG_READ(addr);
         }
      }
      if ((len - j)>0) {
         while ( (UF0EPS0 & C_BKOUT1) == 0 ) {
           /*data wait*/
         }
         UF0IC3 = ~ep_status;
      }
   }
   return DEV_OK;
}
```

The program waits if there is no data to be received.

An error occurs if the length of the data in the register is not the specified length.

The received data is transferred to a buffer.

If there is still data to be received, the length of the data is obtained again, and the data is transferred to a buffer.

If there is data remaining in EP0, or if there is data in EP1, the data is obtained.

### (3) Transmitting an EP0NULL packet

**Figure 3-36. Transmitting an EP0NULL Packet**

```
void usbf850_sendnullEP0(void)
{
  UF0FIC0 = C_EP0WC;  /*FIFO Clear*/
  UF0DEND |= C_E0DED; /*data send(Null Packet)*/
}
```

Clears the FIFO and transmits the NULL packet

### (4) Returning a STALL handshake

**Figure 3-37. Returning a STALL Response**

```
void usbf850_sendstallEP0(void)
{
  UF0SDS = C_SNDSTL; /*send STALL*/
}
```

STALL handshake response

# CHAPTER 4 FILE TRANSFER APPLICATION

This chapter describes the file transfer application that runs on the host.

## 4.1 Development Environment

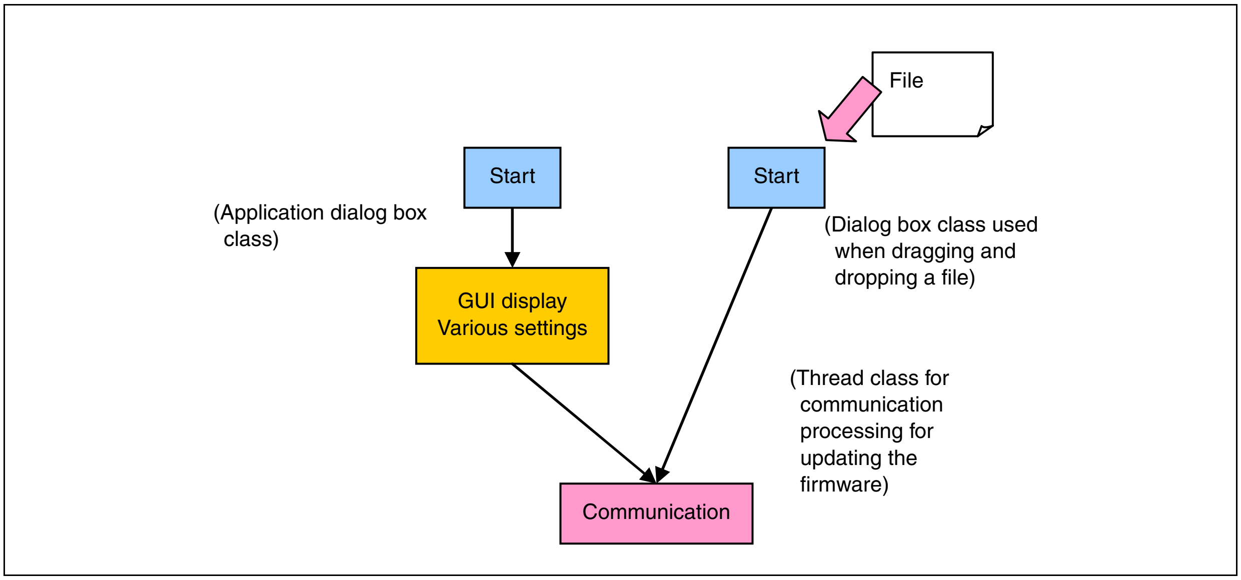
The file transfer application must be set up in the following environment.

OS: Windows XP
Development software: Microsoft Visual C++ 6.0 (MFC)

## 4.2 Operation Overview

When the file transfer application is run with the target file to use to update the firmware specified as a parameter (option), the application immediately begins updating the firmware. If no file is specified, the configuration dialog box is displayed.

**Figure 4-1. File Transfer Application Operation Overview**


File
Start
Start
(Application dialog box class)
(Dialog box class used when dragging and dropping a file)
GUI display
Various settings
(Thread class for communication processing for updating the firmware)
Communication

## 4.3 Organization of Files

The main files included in the file transfer application are as follows.

**Table 4-1. Main Files Included in the File Transfer Application**

| File Name | Description |
|---|---|
| `FlashSelfRewriteGUI.dsw` | Workspace file |
| `FlashSelfRewriteGUI.dsp` | Project file |
| `FlashSelfRewriteGUI.clw` | File for the class wizard |
| `FlashSelfRewriteGUI.rc` | Resource file |
| `FlashSelfRewriteGUI.cpp` | Source file containing the application class |
| `FlashSelfRewriteGUI.h` | Header file defining the application class |
| `FlashSelfRewriteGUIDlg.cpp` | Source file containing the dialog box class for the application |
| `FlashSelfRewriteGUIDlg.h` | Header file defining the dialog box class for the application |
| `FlashSelfRewriteGUIDrop.cpp` | Source file containing the dialog box class used when dragging and dropping a file |
| `FlashSelfRewriteGUIDrop.h` | Header file defining the dialog box class used when dragging and dropping a file |
| `CommandThread.cpp` | Source file containing the thread class that performs communication processing to update the firmware |
| `CommandThread.h` | Header file defining the thread class that performs communication processing to update the firmware |
| `CommonProc.cpp` | Source file containing the class for common processing |
| `CommonProc.h` | Header file defining the class for common processing |
| `SerialPort.cpp` | Source file containing the class for serial communication with the COM port |
| `SerialPort.h` | Header file defining the class for serial communication with the COM port |
| `Resource.h` | Header file defining resources |
| `UsbfUpdate.ini` | Configuration file for using the application |

### 4.3.1 Application class (`FlashSelfRewriteGUI`)

Upon being executed for the first time, this class checks the parameters (options) and then calls the dialog box class used when dragging and dropping a file if a file has been specified or calls the normal dialog box class if no file has been specified.

The execution options that can be specified for this class are as follows.

**Table 4-2. Application Class Execution Options**

| Option | Description |
|---|---|
| `/M` [`chip`\|`address`] | Specify either the chip or address operating mode. |
| `/S` *nnnnnn* | Specify the hexadecimal address at which to begin updating the firmware. |
| `/C` *nn* | Specify the number of the connected COM port. |
| `filename` | Specify the path of the file used to update the firmware. |

### 4.3.2 Application dialog box class (`FlashSelfRewriteGUIDlg`)

This class is used to display the dialog box in which settings for updating the firmware are specified. (For details, see **CHAPTER 2 EXECUTING THE SAMPLE PROGRAM FOR UPDATING THE USB FUNCTION FIRMWARE**.) This dialog box is used to specify the operating mode, address, file, and COM port to use for updating the firmware. Note that, when this dialog box is displayed, the configuration file for using the application is read, and, if the file contains any settings, these are used as the default display settings.

If you click the **Update** button, the thread class that performs communication processing to update the firmware is called.

The application dialog box class includes the following member variables.

**Table 4-3. Member Variables in the Application Dialog Box Class**

| Member Variables | | Description |
|---|---|---|
| Data Type | Member Name | |
| int | m_nCOM | Number of the COM port to which to connect |
| TCHAR | m_tcAppDir[_MAX_PATH] | Directory from which the application is run |
| int | m_nCurTargetID | Current target ID |
| CString | m_strCurTarget | Current target name |
| CString | m_strCurDevice | Current device |
| CStringArray | m_arDeviceVal | List of devices |
| CStringArray | m_arDeviceText | List of device names |
| int | m_nDevSize | Current device ROM size |
| CWinThrread* | m_pCommandThread | Pointer to the thread class |
| BOOL | m_bExistThread | Indicates whether the thread exists |
| BOOL | m_bStartUp | Indicates initial startup |
| CArray<int,int> | m_arBlockStart | Array containing starting block numbers |
| CArray<int,int> | m_arBlockEnd | Array containing ending block numbers |
| CArray<int,int> | m_arBlockUnit | Array containing the number of bytes for each block |
| COleDateTime | m_dtStart | Date and time when updating the firmware started |
| COleDateTime | m_dtEnd | Date and time when updating the firmware finished |

The member functions are as follows.

**Table 4-4. `Read_DeviceInfo` Function**

| | | |
|---|---|---|
| Function Name | | Read_DeviceInfo |
| Specification Format | | BOOL Read_DeviceInfo ( VOID ) |
| Description | | Acquires information from the configuration file for using the application. |
| Input/Output | Input | None |
| | Output | TRUE (success) or FALSE (failure) |

**Table 4-5. `Write_DeviceInfo` Function**

| | | |
|---|---|---|
| Function Name | | `Write_DeviceInfo` |
| Specification Format | | `BOOL Write_DeviceInfo ( VOID )` |
| Description | | Updates the configuration file for using the application. |
| Input/Output | Input | None |
| | Output | `TRUE` (success) or `FALSE` (failure) |

**Table 4-6. `Update_Message` Function**

| | | |
|---|---|---|
| Function Name | | `Update_Message` |
| Specification Format | | `VOID Update_Message ( LPCTSTR )` |
| Description | | Displays a message in the message display field. |
| Input/Output | Input | A pointer to the message string |
| | Output | None |

**Table 4-7. `Get_BlockAddress` Function**

| | | |
|---|---|---|
| Function Name | | `Get_BlockAddress` |
| Specification Format | | `DWORD Get_BlockAddress( int nBlk, EnBlockAddress opt )` |
| Description | | Returns the memory address of the specified block number. |
| Input/Output | Input | `nBlk:` A block number<br>`opt:` `START` or `END` (for the starting or ending block, respectively) |
| | Output | A memory address |

**Table 4-8. `Get_AddressBlock` Function**

| | | |
|---|---|---|
| Function Name | | `Get_AddressBlock` |
| Specification Format | | `int Get_AddressBlock( DWORD dwAddress )` |
| Description | | Returns the block number that has the specified address. |
| Input/Output | Input | `dwAddress:` A memory address |
| | Output | A block number |

**Table 4-9. `Initialize_Device` Function**

| | | |
|---|---|---|
| Function Name | | `Initialize_Device` |
| Specification Format | | `VOID Initialize_Device( VOID )` |
| Description | | Performs initialization processing. |
| Input/Output | Input | None |
| | Output | None |

Table 4-10. **`AppStatus`** Function

| | | |
|---|---|---|
| Function Name | | `AppStatus` |
| Specification Format | | `VOID AppStatus( BOOL stu )` |
| Description | | Specifies the status when the firmware is updated. |
| Input/Output | Input | `stu: TRUE` (The dialog box can be used.)<br>`FALSE` (The dialog box cannot be used.) |
| | Output | None |

### 4.3.3 Dialog box class used when a file is dragged and dropped (`FlashSelfRewriteGUIDrop`)

Immediately after the dialog box for this class is displayed, the thread class that performs communication processing to update the firmware is called, and the update begins. Only a progress bar is displayed in this dialog box.

The member variables are shown below. (Member variables included in the dialog box class for the application have been omitted.)

Table 4-11. Member Variables in the Dialog Box Class Used When a File Is Dragged and Dropped

| Member Variables | | Description |
|---|---|---|
| Data Type | Member Name | |
| `CString` | `m_strFileName` | Target file path |
| `EnMode` | `m_enMode` | Updating mode |
| `DWORD` | `m_dwStartAddress` | Address at which to start the update |

The member functions are as follows.

Table 4-12. **`Execute`** Function

| | | |
|---|---|---|
| Function Name | | `Execute` |
| Specification Format | | `VOID Execute( VOID )` |
| Description | | Performs update processing. |
| Input/Output | Input | None |
| | Output | None |

### 4.3.4 Thread class that performs communication processing to update the firmware (CommandThread)

This class uses the class for serial communication with the COM port to connect to the target evaluation board and transmit or receive the specified file in accordance with the interface specifications. If a HEX file is specified, this class analyzes the file.

The member variables are shown below. (Member variables included in the dialog box class for the application have been omitted.)

**Table 4-13. Member Variables in the Thread Class That Performs Processing to Update the Firmware**

| Member Variables | | Description |
|---|---|---|
| Data Type | Member Name | |
| CDialog* | m_pAppDlg | Pointer to the dialog box class, which calls the thread class |
| CString | m_strAppDir | Directory in which the application resides |
| BOOL* | m_pbExistThread | Pointer to a flag indicating whether the thread exists |
| CSerialPort | m_Serial | Instance of the class for serial communication with the COM port |
| int | m_nCOM | Number of the COM port to which to connect |
| CString | m_strFileName | Target file path |
| EnMode | m_enMode | Updating mode |
| DWORD | m_dwStartAddress | Address at which to start updating the firmware |
| DWORD | m_dwROMStartAddress | First ROM address |
| DWORD | m_dwROMEndAddress | Last ROM address |

The member functions are as follows.

**Table 4-14. Cal_CheckSum Function**

| | | |
|---|---|---|
| Function Name | | Cal_CheckSum |
| Specification Format | | BYTE Cal_CheckSum( LPBYTE bytes, LONG size ) |
| Description | | Calculates the checksum. |
| Input/Output | Input | bytes: A pointer to a data string<br>size: The length of the data string |
| | Output | The calculated checksum |

**Table 4-15. Change_strHex2Binary Function**

| | | |
|---|---|---|
| Function Name | | Change_strHex2Binary |
| Specification Format | | VOID Change_strHex2Binary( LPCSTR strHex, LPBYTE pbytes, LONG size ) |
| Description | | Converts a hexadecimal character string into a binary data string. |
| Input/Output | Input | strHex: A pointer to a hexadecimal character string<br>pbytes: A pointer to the beginning of a data string<br>size: The size of the data to convert |
| | Output | None |

Table 4-16. **Upsets_DWORD** Function

| | | |
|---|---|---|
| Function Name | | Upsets_DWORD |
| Specification Format | | DWORD Upsets_DWORD( DWORD dwVal ) |
| Description | | Reverses a DWORD value in byte units as follows:<br>0x*aabbccdd* is converted to 0x*ddccbbaa*. |
| Input/Output | Input | dwVal: The DWORD value to reverse |
| | Output | The reversed value |

Table 4-17. **SET_StartRecord** Function

| | | |
|---|---|---|
| Function Name | | SET_StartRecord |
| Specification Format | | VOID SET_StartRecord ( LPVOID lpRecord ) |
| Description | | Creates the start record for updating the firmware. |
| Input/Output | Input | lpRecord: A pointer to a stored record |
| | Output | None |

Table 4-18. **SET_EndRecord** Function

| | | |
|---|---|---|
| Function Name | | SET_EndRecord |
| Specification Format | | VOID SET_EndRecord ( LPVOID lpRecord ) |
| Description | | Creates the end record for updating the firmware. |
| Input/Output | Input | lpRecord: A pointer to a stored record |
| | Output | None |

### 4.3.5 Common Processing Class (CommonProc)

This class defines commonly used processing.

The member functions are as follows.

Table 4-19. **GetAppDir** Function

| | | |
|---|---|---|
| Function Name | | GetAppDir |
| Specification Format | | static VOID GetAppDir( LPTSTR path, int sw = 0 ) |
| Description | | Acquires the execution address for the application. |
| Input/Output | Input | path: A pointer to the character string to acquire<br>sw: 0 Acquires the path without conversion.<br>1 Converts the path to a short path during acquisition. |
| | Output | None |

Table 4-20. **Change_Hex2Val** Function

| | | |
|---|---|---|
| Function Name | | Change_Hex2Val |
| Specification Format | | static DWORD Change_Hex2Val( LPCSTR pHex ) |
| Description | | Converts a 1-byte (2-digit hexadecimal) character string to a number. |
| Input/Output | Input | pHex: A pointer to a 2-digit hexadecimal character string |
| | Output | The converted value |

**Table 4-21. `IsNumeric` Function**

| | | |
|---|---|---|
| Function Name | | `IsNumeric` |
| Specification Format | | `static BOOL IsNumeric( LPCTSTR lpNum, LONG size, int type = 10 )` |
| Description | | Checks whether the parameter is a number. |
| Input/Output | Input | `lpNum:` A pointer to a character string representing a number<br>`size:` The number of digits in the parameter to check<br>`type: 10` Checks whether the parameter is a decimal number.<br>`16` Checks whether the parameter is a hexadecimal number. |
| | Output | `TRUE` (which indicates that the parameter is a number) or `FALSE` (which indicates that the parameter is not a number) |

**Table 4-22. `IsExistFile` Function**

| | | |
|---|---|---|
| Function Name | | `IsExistFile` |
| Specification Format | | `static BOOL IsExistFile( LPCTSTR lpszFileName, BOOL bDirectory = FALSE )` |
| Description | | Checks whether a file exists. |
| Input/Output | Input | `lpszFileName:` The file path to check<br>`bDirectory: FALSE` (checking for a file)<br>`TRUE` (checking for a directory) |
| | Output | `TRUE` (which indicates that the file exists) or `FALSE` (which indicates that the file does not exist) |

### 4.3.6 Class for serial communication with the COM port (`SerialPort`)

This class is used to perform serial communication with the COM port. The communication settings, which are fixed, are as follows.

**Table 4-23. Serial Communication Settings**

| Setting | Value |
|---|---|
| Baud rate | 115,200 bps |
| Data size | 8 bits |
| Parity | None |
| Stop bit | 1 bit |
| Start bit | LSB |
| Flow control | None |

The member variables are as follows.

**Table 4-24. Member Variables in the Class for Serial Communication with the COM Port**

| Member Variables | | Description |
|---|---|---|
| Data Type | Member Name | |
| `HANDLE` | `m_hCom` | Handle acquired when a connection is established |
| `DCB` | `m_Dcb` | Device control block structure |
| `COMMTIMEOUTS` | `m_TimeoutSts` | Structure for specifying timeout settings |
| `INT` | `m_nCOM` | Port number for connecting |

The member functions are as follows.

**Table 4-25. `Port_Open` Function**

| | | |
|---|---|---|
| Function Name | | `Port_Open` |
| Specification Format | | `LONG Port_Open(INT com )` |
| Description | | Connects to the specified COM port. |
| Input/Output | Input | `com:` The COM port number |
| | Output | `0` Connection success<br>`−1` Connection failure |

**Table 4-26. `Port_Close` Function**

| | | |
|---|---|---|
| Function Name | | `Port_Close` |
| Specification Format | | `VOID Port_Close( VOID )` |
| Description | | Closes a connected port. |
| Input/Output | Input | None |
| | Output | None |

### Table 4-27. `Port_Write` Function

| | | |
|---|---|---|
| Function Name | | `Port_Write` |
| Specification Format | | `LONG Port_Write(LPCVOID buf, LONG cnt )` |
| Description | | Transmits data by performing serial communication. |
| Input/Output | Input | `buf:` A pointer to the string of data to transmit<br>`cnt:` The length of the data to transmit (in bytes) |
| | Output | The number of transmitted bytes. `-1` is returned if data could not be transmitted. |

### Table 4-28. `Port_Read` Function

| | | |
|---|---|---|
| Function Name | | `Port_Read` |
| Specification Format | | `LONG Port_Read(LPVOID buf, LONG cnt )` |
| Description | | Receives data by performing serial communication. |
| Input/Output | Input | `buf:` A pointer to the string of data in which to store the received data<br>`cnt:` The length of the received data (in bytes) |
| | Output | The number of received bytes. `-1` is returned if data could not be received. |

### Table 4-29. `Get_PortNumber` Function

| | | |
|---|---|---|
| Function Name | | `Get_PortNumber` |
| Specification Format | | `INT Get_PortNumber( VOID )` |
| Description | | Acquires the number of the currently connected port. |
| Input/Output | Input | None |
| | Output | The number of the currently connected port |

### Table 4-30. `AutoScanCom` Function

| | | |
|---|---|---|
| Function Name | | `AutoScanCom` |
| Specification Format | | `INT AutoScanCom ( LPCTSTR pszService, LPCTSTR pszInterface, INT nNo = 0 )` |
| Description | | Detects the number of a COM port that can be connected. |
| Input/Output | Input | `pszService:` The name of the service for which the COM port is running<br>`pszInterface:` The interface name<br>`nNo:` Specify whether to search for numbers later than this number. |
| | Output | The detected COM port number. 0 is returned if no number is found. |

### 4.3.7 Configuration file for using the application (`UsbfUpdate.ini`)

This `ini` file is used to retain settings or device information. This file is located in the same folder as the `exe` file. The definitions in this `ini` file are as follows.

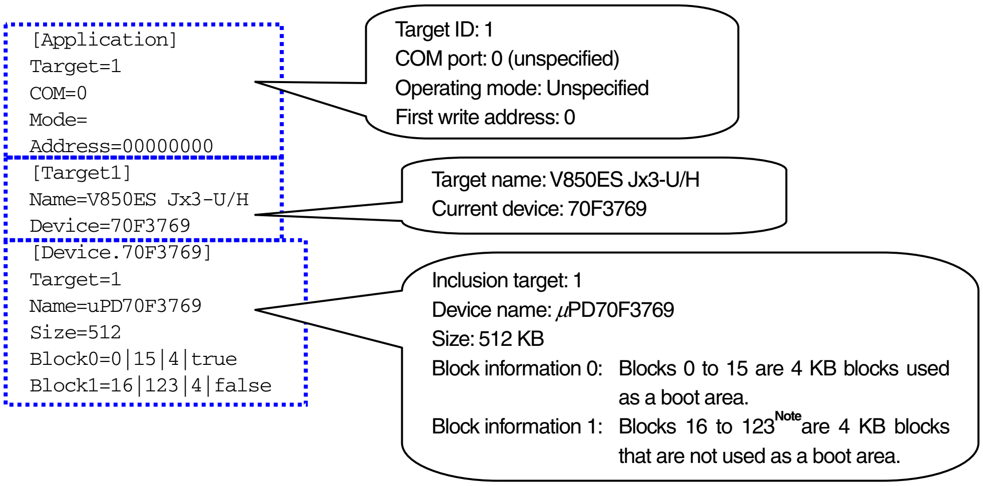
**Table 4-31. Sections in the Configuration File for Using the Application**

| Section | Description |
|---|---|
| `Application` | Indicates the currently specified values for the application. |
| `Tartget1` | Indicates the target ID. |
| `Device.70F3769` | Indicates the device information.<br>Multiple settings can be specified. |

**Table 4-32. Items in the Configuration File for Using the Application**

| Section | Key | Value | Description |
|---|---|---|---|
| `Application` | `Target` | `1` or greater | The currently specified ID number |
| | `COM` | `1` to `20` | The number of the connected COM port or COM port to connect |
| | `Mode` | `chip` or `address` | Indicates the currently specified operating mode.<br>`chip:` Updates the firmware with a user-created program using boot swapping<br>`address:` Updates the firmware using a specified address. |
| | `Address` | `FFFFFFFF` | The first address to write to (in hexadecimal) |
| `Target1` | `Name` | *XXX* | Indicates the name of this target. |
| | `Device` | *XXX* | The device specified for this target |
| `Device.70F3769` | `Target` | `1` or greater | The ID of the target to which this device belongs |
| | `Name` | *XXX* | The name of this device |
| | `Size` | `999` | The ROM size of this target |
| | `Block0` | *XXX\|XXX\|XXX\|XXX* | Block information delimited using vertical bars (\|)<br>*First block number\|last block number\|size of each block (in KB)\|whether this is a booting area*<br>Mutiple blocks can be specified by using `Block1`, `Block2`, ..., `Block`*n*. |

**Figure 4-2. `ini` Configuration File for Using the Application**



```
[Application]
Target=1
COM=0
Mode=
Address=00000000
[Target1]
Name=V850ES Jx3-U/H
Device=70F3769
[Device.70F3769]
Target=1
Name=uPD70F3769
Size=512
Block0=0|15|4|true
Block1=16|123|4|false
```

Target ID: 1
COM port: 0 (unspecified)
Operating mode: Unspecified
First write address: 0

Target name: V850ES Jx3-U/H
Current device: 70F3769

Inclusion target: 1
Device name: μPD70F3769
Size: 512 KB
Block information 0: Blocks 0 to 15 are 4 KB blocks used as a boot area.
Block information 1: Blocks 16 to 123[Note] are 4 KB blocks that are not used as a boot area.

**Note** Blocks 124 to 127 are allocated by the firmware updating program and cannot be specified.

## 4.4 Operating Mode

This section describes the operating modes.

**(1) Chip**

The specified HEX file must be in the Motorola S-record format or Intel extended format. If a file that has any other format is specified, an error occurs during analysis. Because the file is written to the first memory address, any specified address is ignored.

**(2) Address**

A file image is transferred, and then writing is performed starting at the specified address.

## 4.5 Display of Messages

The following table describes the messages displayed in the message display field and when they are displayed.

**Table 4-33. Displayed Messages**

| | Message | When Displayed |
|---|---|---|
| 1 | `Updating the firmware will now start.` | This message is displayed when the processing to update the firmware starts |
| 2 | `Updating has finished successfully.` | This message is displayed when the processing to update the firmware finishes successfully |
| 3 | `Specify the file.` | This message is displayed if no file is specified for updating the firmware or the specified file does not exist. |
| 4 | `Specify the mode.` | This message is displayed if no mode is specified for updating the firmware. |
| 5 | `Specify the correct address.` | This message is displayed if the correct address is not specified while updating the firmware in the address mode. |
| 6 | `Specify the COM port.` | This message is displayed if the COM port is not correctly specified. |
| 7 | `ERR: An error occurred while opening the file.` | This message is displayed if an error occurred while opening the file. |
| 8 | `ERR: A file format error occurred.` | This message is displayed if a file other than a Motorola S-record format file or Intel extended format file is specified when mode=chip is specified. |
| 9 | `ERR: COM port n could not be connected.` | This message is displayed if COM port *n* could not be connected. |
| 10 | `ERR: A data transmission error occurred.` | This message is displayed if data transmission failed. |
| 11 | `ERR: A data reception error occurred.` | This message is displayed if data reception failed (for all three retry attempts). |
| 12 | `ERR: Processing to update the firmware stopped.` | This message is displayed if an NAK error was received from the evaluation board. |
| 13 | `ERR: A file size error occurred.` | This message is displayed if the data is found to exceed the ROM area during the file size check. |

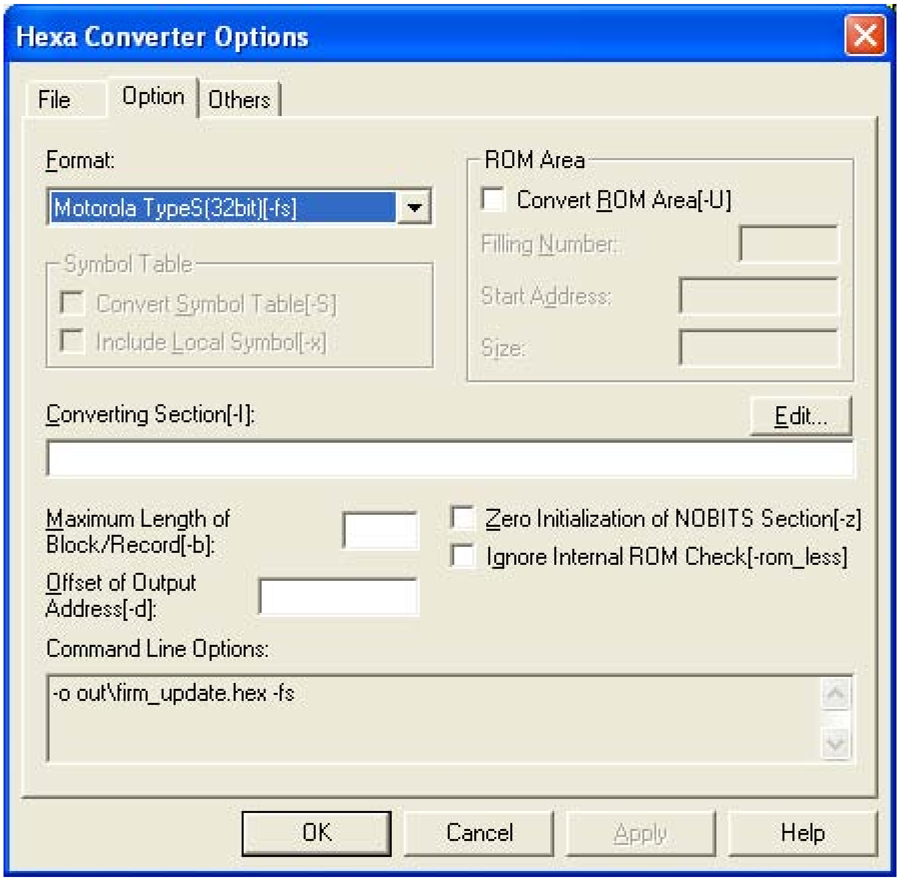
This chapter provides notes to keep in mind when creating a program.

## 5.1 Setting Up PM+ (Specifying the HEX File Format)

Only HEX files in the Motorola S-record format (32 bit) or Intel extended format can be used for the USB function firmware updating program (the file transfer application) when updating a user-created program. Specify the format in the **Hexa Converter Options** dialog box on the **Option** tab.

In the following example, **Motorola TypeS(32bit)[-fs]** is selected from the **Format** drop-down list.

**Figure 5-1. Example of Specifying the HEX File Format**



Hexa Converter Options

Format: Motorola TypeS(32bit)[-fs]

Command Line Options: -o out\firm_update.hex -fs

File
Option
Others
Symbol Table
☐ Convert Symbol Table[-S]
☐ Include Local Symbol[-x]
ROM Area
☐ Convert ROM Area[-U]
Filling Number:
Start Address:
Size:
Converting Section[-l]:
Edit...
Maximum Length of Block/Record[-b]:
Offset of Output Address[-d]:
☐ Zero Initialization of NOBITS Section[-z]
☐ Ignore Internal ROM Check[-rom_less]
OK
Cancel
Apply
Help

## 5.2 Boot Processing (Reset Vector Section)

Because the self-update program assumes that vector processing is performed at the start of memory (starting at the address 00000000) following a reset, use the start of memory for the reset section in user-created programs.

## 5.3 Linker Directives (Restriction on Allocating User-Created Programs)

User-created programs cannot be allocated where the firmware updating program resides (starting at the address 0007C000H). Therefore, when specifying the segments in the linker directive file, specify an address such that user-created programs are not allocated where the self-update program resides. (For details, see **3.2 Memory Map**.)

# CHAPTER 6 CUSTOMIZATION

This chapter describes how to port the USB function firmware update program to another environment. The TK-850/JG3H board is used as an example.

The memory capacities for the CPU (μPD70F3760) used for the TK-850/JG3H board are as follows.

- Internal flash memory: 256 KB (blocks 0 to 63)
- Internal RAM: 32 KB

## 6.1 Modifying Files

The following files must be modified:

- `firm_update.dir`
- `usbf_fwup_mem_def_usr.h`
- `usbf_fwup_pwonchk_usr.c`
- `UsbfUpdate.ini`

### 6.1.1 Modifying the self-update program

Modify the firmware update program by customizing the following files (which are in the `firm_update` directory) in accordance with the environment to which the program is to be ported.

**Table 6-1. Files to Customize for the Firmware Update Program**

| File Name | Description |
|---|---|
| `firm_update.dir` | Linker directive file |
| `include\usbf_fwup_mem_def_usr.h` | Flash memory environment definitions |
| `src\usbf_fwup_pwonchk_usr.c` | Source file for selecting the program to execute |

### (1) `firm_update.dir`

Modify the addresses at which segments are allocated in accordance with the CPU (μPD70F3760) used for the TK-850/JG3H board.

The firmware update program must be allocated at the end of the flash memory and uses four blocks (16 KB). The `CONST` and `TEXT` segments use a total of three blocks, and the `APSTART` segment uses one block. The `FLASHTEXT` and `DATA` segments are allocated at the beginning of the internal RAM.

**Figure 6-1. flash_update.dir**

```
CONST   : !LOAD ?R V0x3c000 {
        .const                    = $PROGBITS   ?A .const;
};

TEXT    : !LOAD ?RX {
        SelfLib_Rom.text          = $PROGBITS   ?AX SelfLib_Rom.text;
        .text                     = $PROGBITS   ?AX .text;
};

APSTART : !LOAD ?RX V0x3f000 {
        apstart.text              = $PROGBITS   ?AX apstart.text;
};

FLASHTEXT: !LOAD ?RX V0x3ff7000 {
        SelfLib_ToRamUsrInt.text  = $PROGBITS   ?AX SelfLib_ToRamUsrInt.text;
        SelfLib_ToRamUsr.text     = $PROGBITS   ?AX SelfLib_ToRamUsr.text;
        SelfLib_RomOrRam.text     = $PROGBITS   ?AX SelfLib_RomOrRam.text;
        SelfLib_ToRam.text        = $PROGBITS   ?AX SelfLib_ToRam.text;
        flash.text                = $PROGBITS   ?AX flash.text;
};

DATA    : !LOAD ?RW {
        .data                     = $PROGBITS   ?AW  .data;
        .sdata                    = $PROGBITS   AWG .sdata;
        .sbss                     = $NOBITS     ?AWG .sbss;
        .bss                      = $NOBITS     ?AW  .bss;
        SelfLib_RAM.bss           = $NOBITS     ?AW  SelfLib_RAM.bss;
};

__tp_TEXT @ %TP_SYMBOL;
__gp_DATA @ %GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA @ %EP_SYMBOL;
```

Specifies the starting address of block 60.

Specifies the starting address of block 63.

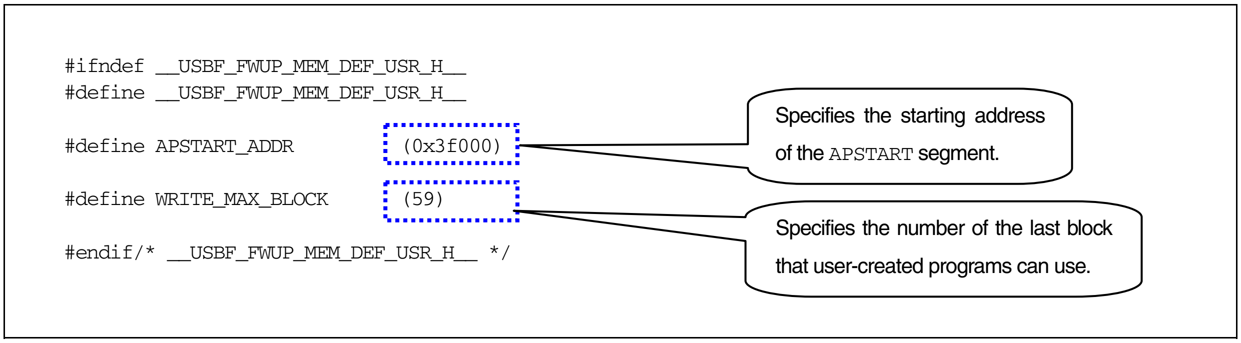
Specifies the starting address of internal RAM.

### (2) `usbf_fwup_mem_def_usr.h`

This header file defines the flash memory environment used for the firmware update program.

For `APSTART_ADDR`, specify the address at which the `APSTART` segment is allocated, which is specified in the linker directive file. For `WRITE_MAX_BLOCK`, specify the number of the last block that can be used for a user-created program. Because the firmware update program uses blocks 60 to 63, user-created programs can only use blocks 0 to 59.

**Figure 6-2. `usbf_fwup_mem_def_usr.h`**



```
#ifndef __USBF_FWUP_MEM_DEF_USR_H__
#define __USBF_FWUP_MEM_DEF_USR_H__

#define APSTART_ADDR        (0x3f000)

#define WRITE_MAX_BLOCK     (59)

#endif/* __USBF_FWUP_MEM_DEF_USR_H__ */
```

Specifies the starting address of the `APSTART` segment.

Specifies the number of the last block that user-created programs can use.

### (3) `usbf_fwup_pwonchk_usr.c`

When power is supplied or a reset occurs, this source file is used to determine whether to execute the firmware update program or a user-created program.

Because the SW3 and SW4 on the TK-850/JG3H board have the same configuration as those on the TK-850/JH3U-SP board, this source file must not be modified.

**Figure 6-3. `usbf_fwup_pwonchk_usr.c`**

```
#pragma ioreg

#include "nec_types.h"

#define SW_PUSHED   (0x00)     /* pushed switch SW3 and SW4 */
#define SW_STATUS   (0x03)     /* switch status SW3 and SW4 */

s32 usbf_fwup_pwonchk_usr(void);

s32 usbf_fwup_pwonchk_usr(void)
{
   s32    ret = -1;
   u08    sts;

   sts = P9H;
   if ((sts & SW_STATUS) == SW_PUSHED) {
      ret = 0;
   }

   return ret;
}
```

### 6.1.2 Modifying the `ini` file for the file transfer application

Customize the `UsbfUpdate.ini` file in the **FirmupdateGUI** directory in accordance with the environment to which the program is to be ported.

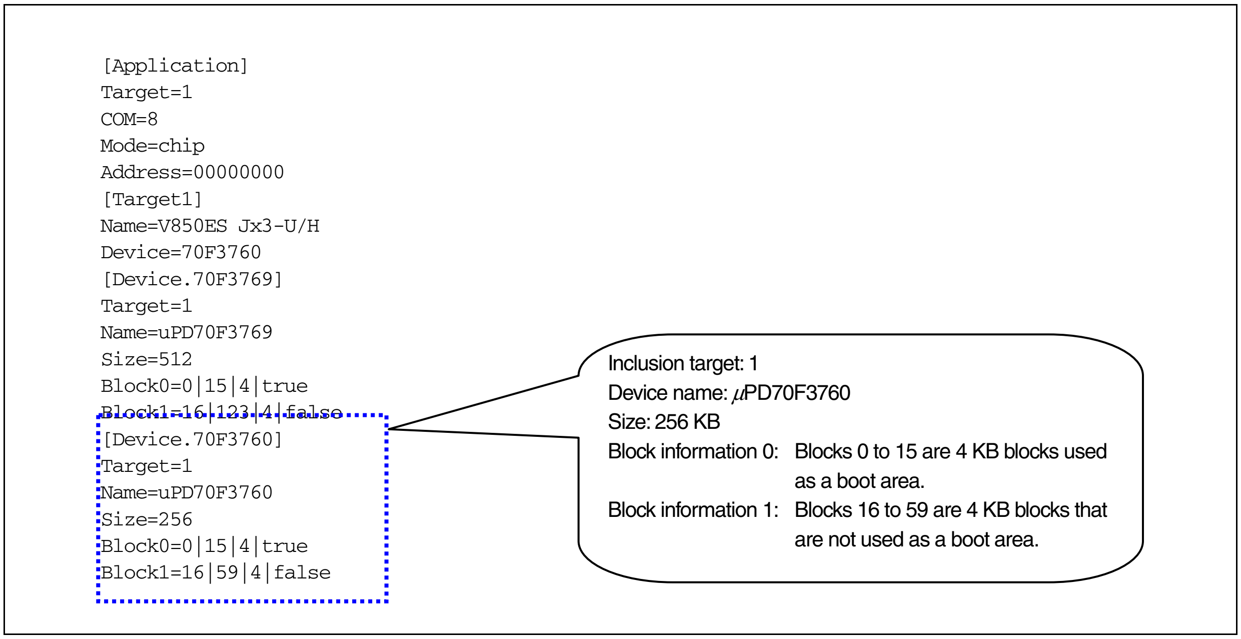
**Table 6-2. File to Customize for the File Transfer Application**

| File Name | Description |
|---|---|
| `UsbfUpdate.ini` | Settings for the file transfer application |

#### (1) `UsbfUpdate.ini`

The following figure shows how to add the μPD70F3760 settings.

**Figure 6-4. `UsbUpdate.ini`**



```
[Application]
Target=1
COM=8
Mode=chip
Address=00000000
[Target1]
Name=V850ES Jx3-U/H
Device=70F3760
[Device.70F3769]
Target=1
Name=uPD70F3769
Size=512
Block0=0|15|4|true
Block1=16|123|4|false
[Device.70F3760]
Target=1
Name=uPD70F3760
Size=256
Block0=0|15|4|true
Block1=16|59|4|false
```

Inclusion target: 1
Device name: μPD70F3760
Size: 256 KB
Block information 0: Blocks 0 to 15 are 4 KB blocks used as a boot area.
Block information 1: Blocks 16 to 59 are 4 KB blocks that are not used as a boot area.

# CHAPTER 7 DATA COMMUNICATION SPECIFICATIONS

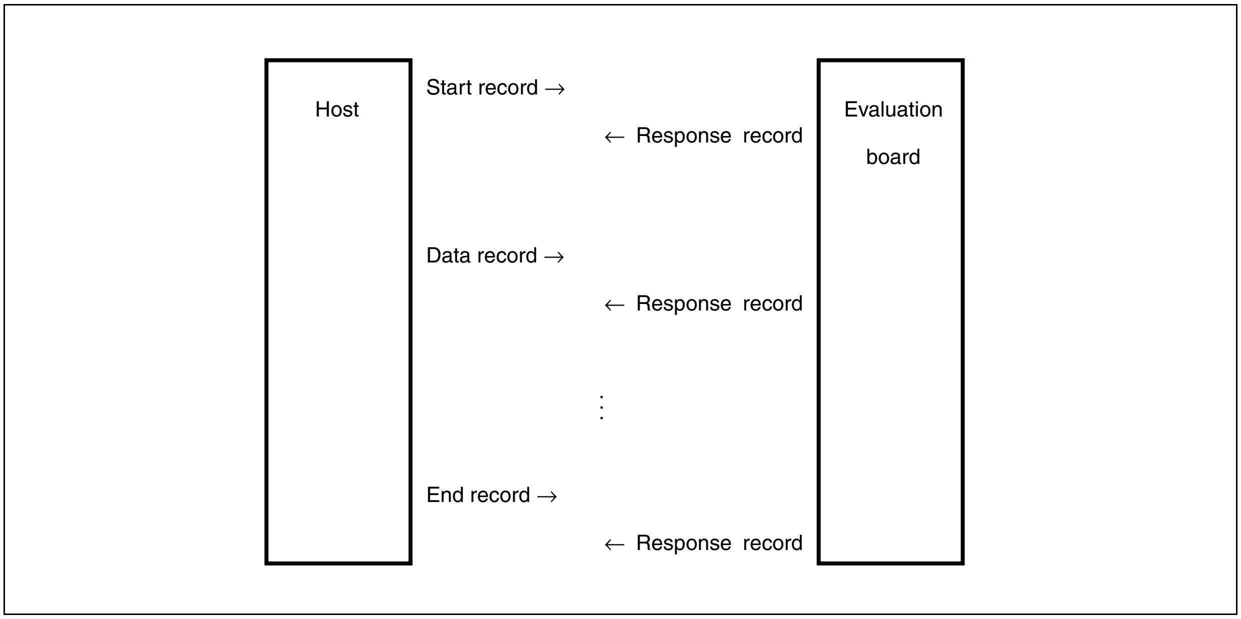
## 7.1 Specifications of the Communication Interface for Updating the Firmware

This section describes the communication between the host on which the firmware update program runs and the evaluation board.

### 7.1.1 Communication data sequence

The host transmits a start record at the beginning of communication and an end record at the end. Data loaded into the flash memory is transmitted as a series of data records.

**Figure 7-1. Communication Data Sequence**



### 7.1.2 Data transmitted by the host

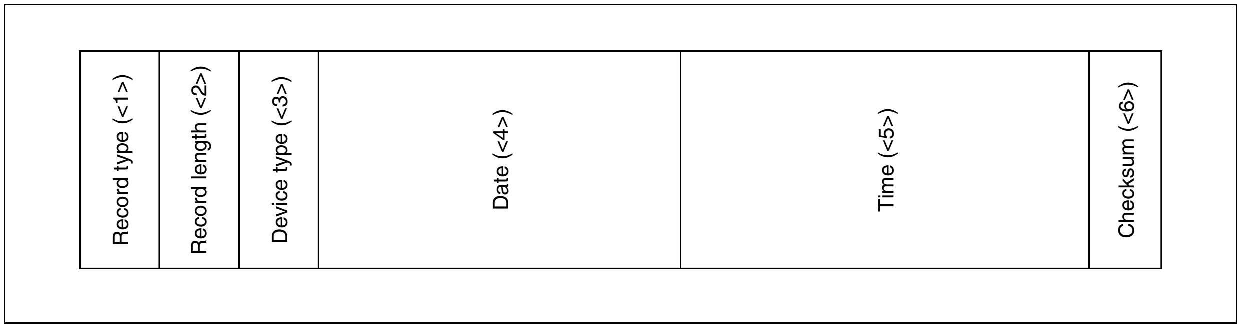
The host transmits a start record, data records, and an end record.

Records are transmitted one by one, and the next record is not transmitted until a response record is received.

#### (1) Start record

This record is transmitted first when updating the firmware.

**Figure 7-2. Start Record Format**



**<1> Record type**

The type of record

1 byte

The record type of the start record is 0x00.

**<2> Record length**

The number of bytes for the device type and later

1 byte

**<3> Device type**

The type of device

1 byte

**<4> Date**

The current date

The year, month, and day require 1 byte each.

The last two digits of the year are specified (based on the Western calendar).

**<5> Time**

The current time

The hour, minute, and second require 1 byte each.

**<6> Checksum**

The record checksum

1 byte

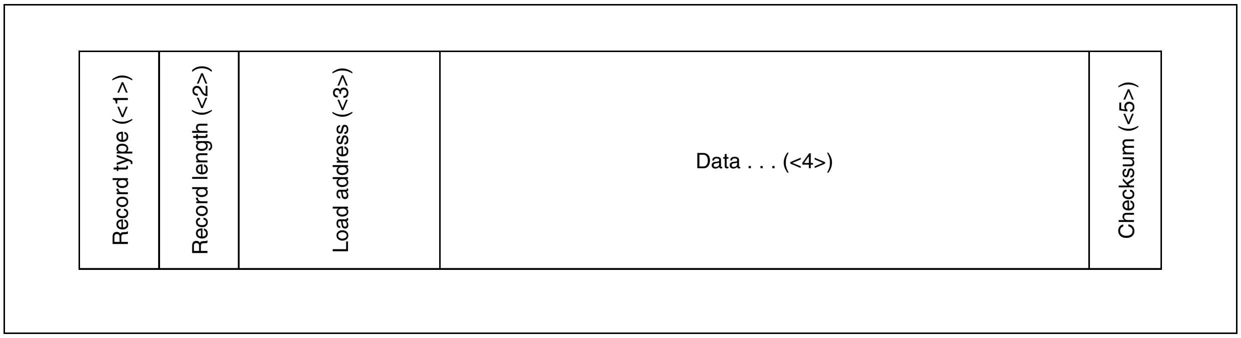
This is the checksum for the record length, device type, date, and time.

The checksum is the lower 8 bits of the one's complement of the sum of each byte value.

### (2) Data records

These records contain the data to be loaded into the flash memory.

**Figure 7-3. Data Record Format**



**<1> Record type**

The type of record

1 byte

The record type of a data record is 0x0f.

**<2> Record length**

The number of bytes for the load address and later

1 byte

**<3> Load address**

A flash memory address

4 bytes

Data is loaded into the flash memory starting at this address.

The load address is a 32-bit number in little endian format.

**<4> Data**

The data to load into the flash memory

Each record can contain up to 256 bytes.

**<5> Checksum**

The record checksum

1 byte

This is the checksum for the record length, load address, and data.

The checksum is the lower 8 bits of the one’s complement of the sum of each byte value.

### (3) End record

This record is transmitted after all other records.

**Figure 7-4. End Record Format**

| | | | |
|---|---|---|---|
| Record type (<1>) | Record length (<2>) | Device type (<3>) | Checksum (<4>) |

**<1> Record type**

The type of record

1 byte

The record type of the end record is 0xf0.

**<2> Record length**

The number of bytes for the device type and later

1 byte

**<3> Device type**

The type of device

1 byte

**<4> Checksum**

The record checksum

1 byte

This is the checksum for the record length and device type.

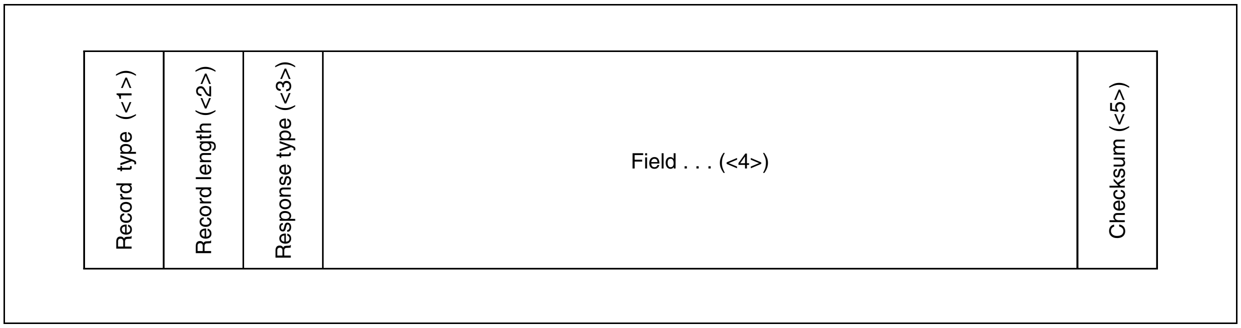
The checksum is the lower 8 bits of the one’s complement of the sum of each byte value.

### 7.1.3 Data transmitted by the evaluation board

The evaluation board transmits records in response to records from the host.

#### (1) Response records

**Figure 7-5. Response Record Format**



**<1> Record type**

The type of record

1 byte

This is the type of record for which this response record is returned.

**<2> Record length**

The number of bytes for the response type and later

1 byte

**<3> Response type**

The response type

1 byte

The following three types are available:

| | |
|---|---|
| 0x00: | ACK |
| 0x0f: | NAK (a request to transmit the record again) |
| 0xf0: | NAK (an error termination) |

**<4> Field**

If an error occurs, the field is a 1-byte error code.

If no error occurs, the contents vary depending on the record type as follows.

| | |
|---|---|
| Start record: | Device type |
| Data record: | Load address |
| End record: | Device type |

**<5> Checksum**

The record checksum

1 byte

This is the checksum for the record length, response type, and field.

The checksum is the lower 8 bits of the one's complement of the sum of each byte value.

*For further information, please contact:*

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

**[America]**

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

**[Europe]**

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

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

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

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

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

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

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

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

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

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

**[Asia & Oceania]**

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

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

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

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

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

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

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