RL78/G13

Flash Self Programming: Execution

R01AN0718EJ0102
Rev.1.02
Feb 15, 2013

Introduction

This application note is intended for users who have a basic understanding of the functions of the Type 01 Flash Self Programming Library for the RL78/G13 microcontrollers and who are to design application systems using that library.

The purpose of this application note is to have the user gain an understanding of how to use the Type 01 Flash Self Programming Library which is used to program the code flash memory of the RL78 family.

Target Device

RL78/G13 (R5F100LE)

Contents

Introduction

Target Readers This application note is intended for users who are to design application systems using the Type 01 Flash Self Programming Library for RL78/G13 microcontrollers.

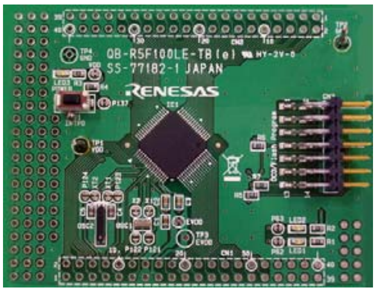
Purpose This application note is intended to give users an understanding of how to use the Flash Self Programming Library for the RL78/G13 to develop programs for rewriting the flash memory.

Organization This application note includes the following sections.

- Overview
- Flash self programming library
- Example of rewriting programs
- Appendix

This application note introduces examples of programs that apply the Flash Self Programming Library on the QB-R5F100LE-TB evaluation board. For this reason, you will need to obtain a QB-R5F100LE-TB if you wish to run the provided sample programs.
Evaluating programs on the QB-R5F100LE-TB also requires other items such as an E1 emulator, UART-RS-232C converter, and external power supply. For how to purchase the E1 emulator and UART-RS-232C converter or other inquiries, contact your local distributor.

Figure QB-R5F100LE-TB Evaluation Board



Conventions

Data significance: Higher-order digits to the left and lower-order digits to the right
Active low representations: $\overline{xxx}$ (overscore over pin and signal name)
Note: Footnote for item marked with **Note** in the text.
Caution: Information requiring particular attention
Remark: Supplementary information
Numeral representation: Binary ... xxxx or xxxxB
Decimal ... xxxx
Hexadecimal ... xxxxH

Related Documents

IDE/Title	Document No.
RL78 Microcontrollers Flash Self Programming Library Type01 User's Manual[Note 1]	R01US0050
RL78 Microcontrollers Flash Self Programming Library Type01 V2.20 Release Note[Note1]	R20UT0777
CubeSuite+ V1.03.00 Release Note[Note 2]	R20UT2259
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: Start[Note 2]	R20UT2133
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: RL78 Design[Note 2]	R20UT2136
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: Analysis[Note 2]	R20UT2146
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: Message[Note 2]	R20UT2147
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: RL78 Debug[Note 2]	R20UT2145
CubeSuite+ RL78,78K0R Compiler CA78K0R V V1.50 Release Note[Note2]	R20UT2261
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: RL78, 78K0R Coding[Note 2]	R20UT2140
CubeSuite+ V1.03.00 Integrated Development Environment User's Manual: RL78, 78K0R Build[Note2]	R20UT2143

Notes: 1. This document should be installed together with Ver. 2.20 of the Type 01 Flash Self Programming Library. For the topics that are not covered in the "Flash Self Programming Library Type01 User's Manual," refer to "RL78 Microcontrollers Flash Self Programming Library Type01 Ver.2.20 Release Note"

2. This document should be downloaded from the web page entitled "CubeSuite+ Integrated Development Environment" at the Renesas website.

Caution: The contents of the above-listed documents are subject to change without notice. Be sure to refer to the latest edition of the relevant documents in the design process etc.

Chapter 1 Overview

This application note introduces the procedures for flash self programming of the code flash memory in an RL78/G13 microcontroller using the RL78 Microcontrollers Flash Self Programming Library Type01 V2.20.

For details on "RL78 Microcontrollers Flash Self Programming Library Type01," refer to the following documents.

- RL78 Microcontrollers Flash Self Programming Library Type01 User's Manual (Document No.: R01US0050)
- RL78 Microcontrollers Flash Self Programming Library Type01 V2.20 Release Note (Document No.: R20UT0777)

This chapter gives an overview of the RL78/G13's flash self programming functions.

1.1 Code Flash Memory in the RL78/G13

The RL78/G13 incorporates code flash memory that allows erasure and programming. The features of the RL78/G13 code flash memory are given below.

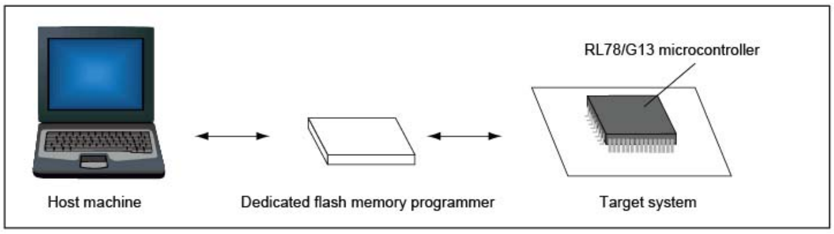
Table 1-1 Features of the RL78/G13 Code Flash Memory

Operating power supply	Erasure and programming are possible on the same power supply.
Minimum erasure unit	1 block (1K = 1024 bytes)
Minimum programming unit	1 word (4 bytes)
Security	Block erase protection, write protection, boot area write protection
	Initial values at shipment are all enabled.
	The flash shield window (FSW) allows the erasure and programming of all the areas except the specified window area to be disabled only during flash self programming.
	Settings can be changed via the Flash Self Programming Library.

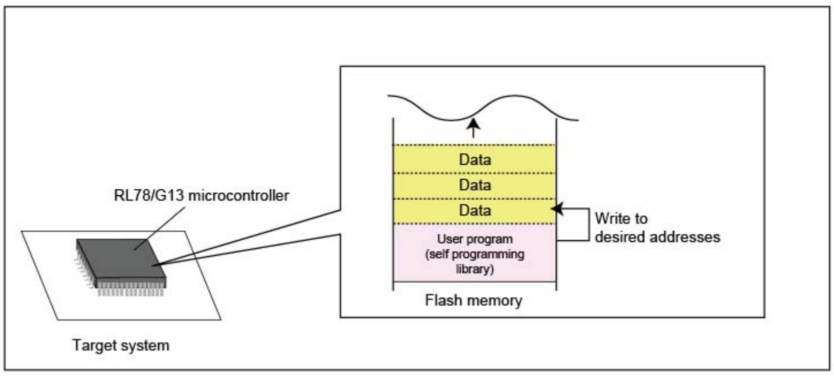
Remark: The write-protection of the boot area and the security settings except the FSW settings are disabled during flash self programming.

The code flash memory can be programmed while the RL78/G13 is installed on the board. The programming of the code flash memory can be accomplished either by a dedicated flash memory programmer or by the flash self programming technique which makes use of a program written in the code flash memory (hereafter referred to as the write program).

Figure 1-1 Means of Programming the Code Flash Memory



Programming using a dedicated flash memory programmer



Programming using self programming

1.2 RL78/G13 Flash Self Programming

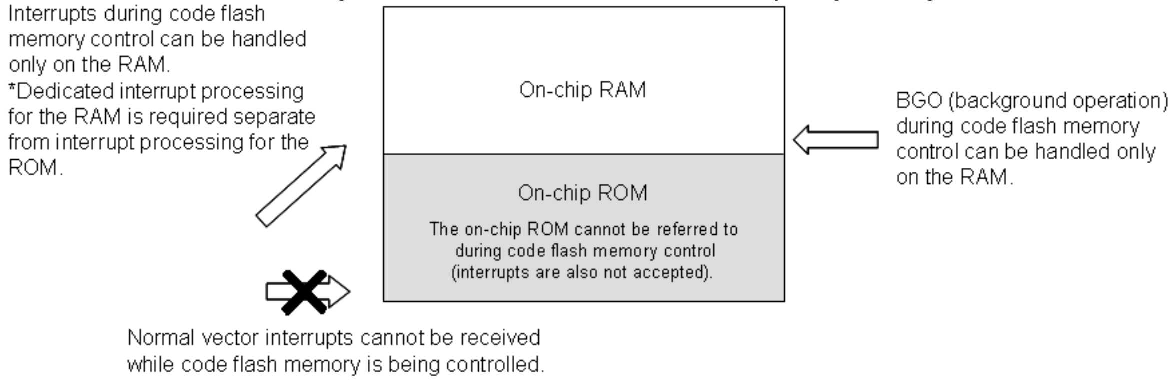
The RL78/G13 is provided with a library for flash self programming. Flash self programming is accomplished by the write program calling functions of the Flash Self Programming Library.

The RL78/G13's flash memory is assigned block numbers in block (1024 bytes) units starting at address 00000H. Erasure of the flash memory is carried out in block units.

The control of RL78/G13 flash self programming is exercised using a sequencer. The code flash memory cannot be referenced while the sequencer is controlling the flash self programming. To run a user program while the control by the sequencer is in progress, it is necessary to relocate some segments of the flash self programming library and the write program to RAM when performing the erasure and programming of the code flash memory and setting of security flags. When it is unnecessary to run any user program while the control by the sequencer is in progress, it is possible to place the flash self programming library and write program in ROM (code flash memory) for execution.

This application note gives examples of placing the flash self programming library and write program in ROM (code flash memory).

Figure 1-2 Outline of Code Flash Memory Programming



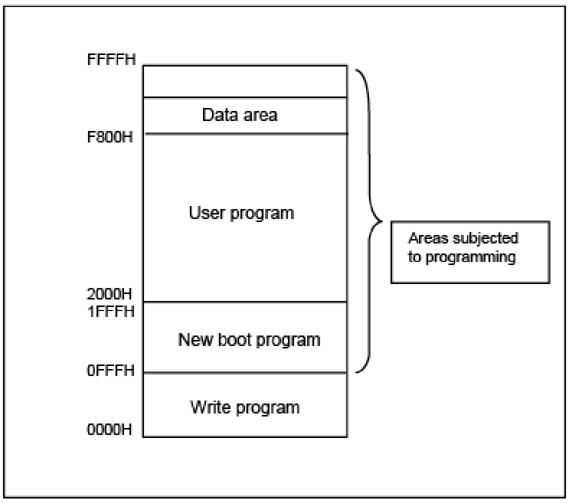
1.3 How to Program the Code Flash Memory

It is possible to rewrite the programs and data in the code flash memory using the flash self programming function.

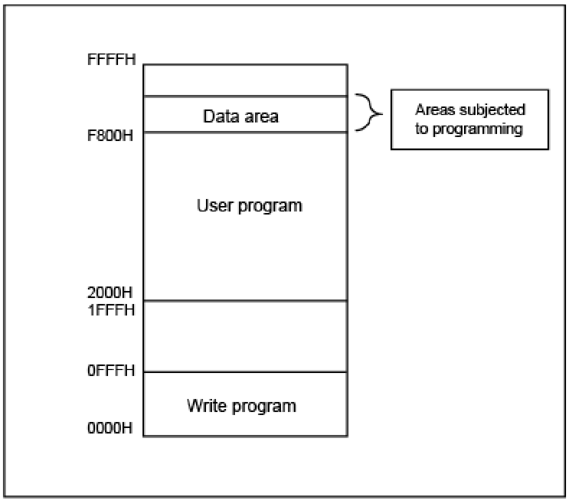
Figure 1-3 shows examples of rewriting the entire section of a program, rewriting data, and rewriting parts of a program that is split and reallocated into two or more parts according to the process to be performed.

When programming the code flash memory using the flash self programming function, it is necessary to allocate the program for flash self programming and the other functional programs to separate blocks.

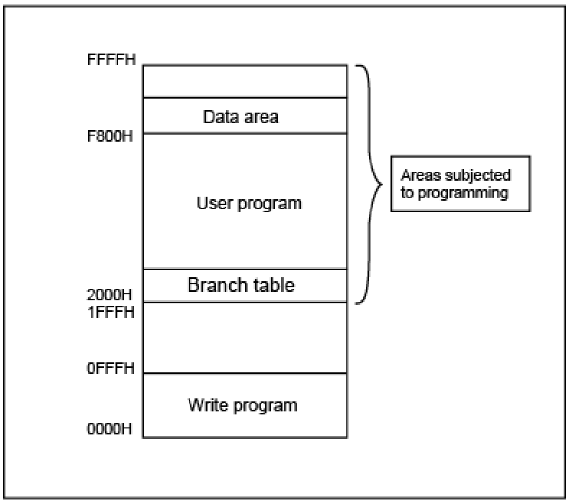
Figure 1-3 Examples of Code Flash Memory Programming



Rewriting a program



Rewriting data



Rewriting parts of a program

1.4 Rewriting Programs and Data

The RL78/G13 provides a boot swap function which serves for safely programming flash memory. The flash area from 0000H to 0FFFH is assigned to boot cluster 0 and the flash area from 1000H to 1FFFH to boot cluster 1. These two areas can be swapped using the boot swap function. The boot swap function allows programs to be swapped safely.

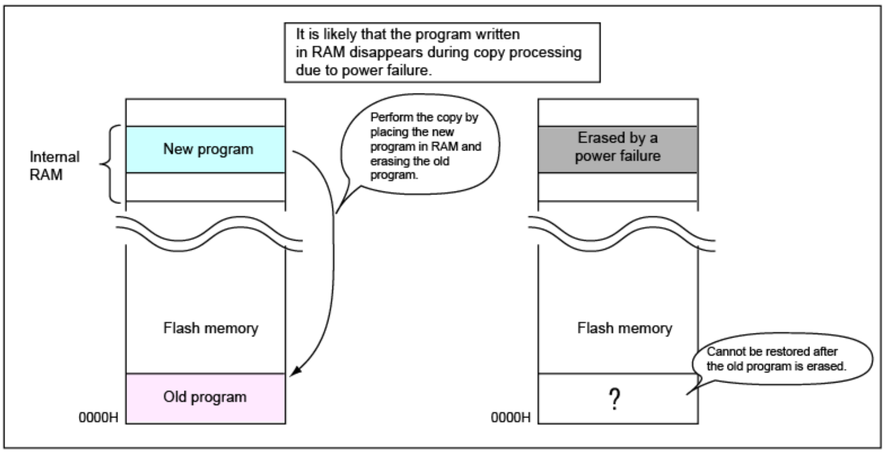
The RL78/G13 comes with library functions for boot swapping. This application note explains how to boot-swap programs using the FSL_InvertBootFlag function and to rewrite programs.

The flash self programming function can also be used to rewrite data to be used by user programs. By allocating a table of rewriting data to fixed addresses, it is possible to rewrite data safely with no modification made to the user program. This application explains how to allocate data tables to fixed addresses for rewriting.

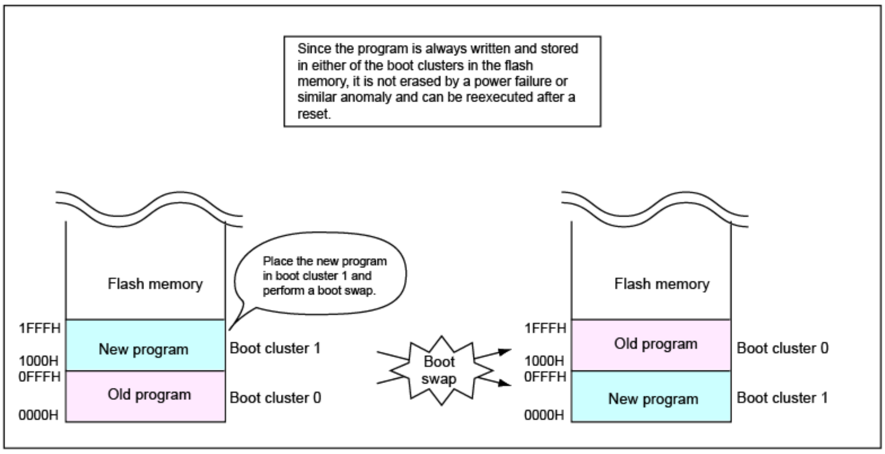
The program that is explained in chapter 2, Example of Configuring and Rewriting a Program, does not rewrite the blocks to which the program for flash self programming is allocated. For details on the sample program, see chapter 2.

Figure 1-4 Rewriting a Program

(Methods of programming in RAM and swapping program areas using the boot swap function)



Rewriting using RAM



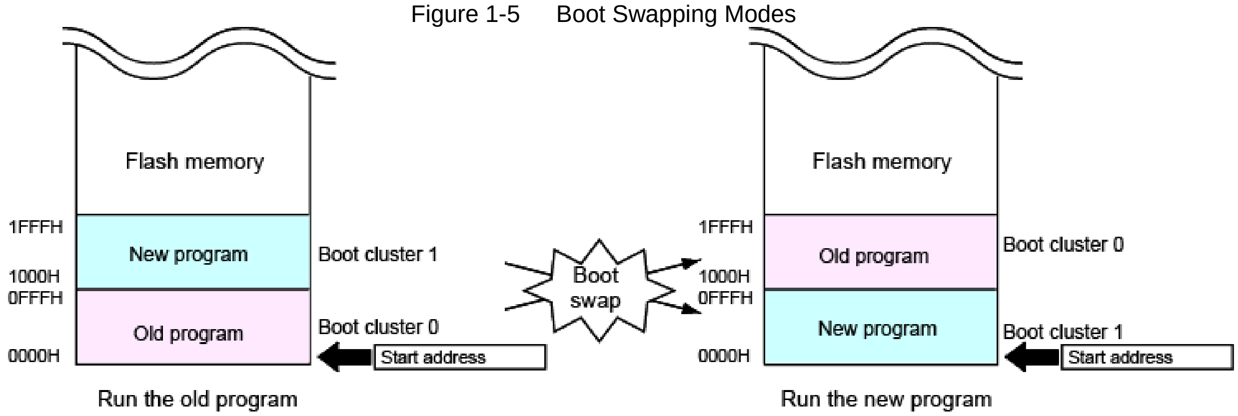
Rewriting a program using the boot swap function

(1) What is a boot swap?

Boot swap is a process of swapping between boot cluster 0 and boot cluster 1. By swapping the clusters, the area that was allocated to boot cluster 0 (0000H-0FFFH) is allocated to boot cluster 1 (1000H-1FFFH) and the area that was allocated to boot cluster 1 (1000H-1FFFH) to boot cluster 0 (0000H-0FFFH).

To perform a boot swap, make required settings for the registers in the code flash memory. There are two modes of boot swapping; the mode in which control is switched to the new program immediately when necessary settings are made and the mode in which only the boot flag is rewritten and the programs are swapped after a reset. This application note explains the latter mode; i.e., the programs are swapped after a reset.

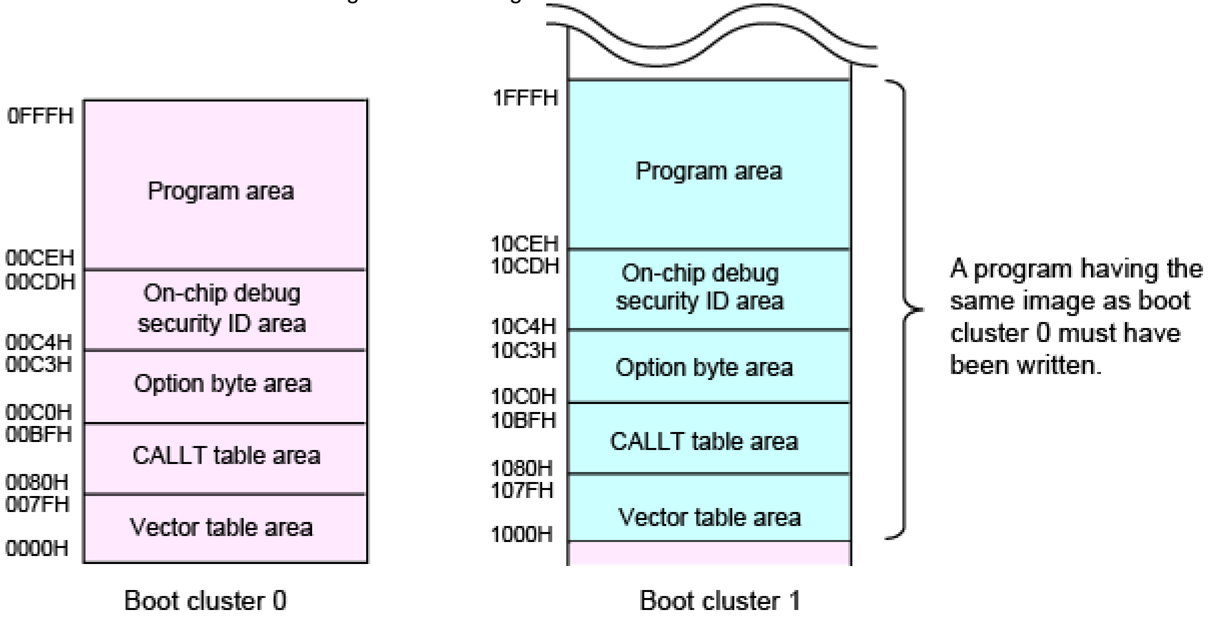
Figure 1-5 Boot Swapping Modes



(2) Program that is to be written into boot cluster 1

Boot cluster 1 is reassigned to boot cluster 0 after the boot swap and its addresses are reset to 0000H-0FFFH. Consequently, the new program to be prewritten in boot cluster 1 must have the same start address and configuration as the program that is written in boot cluster 0.

Figure 1-6 Program to be Written to Boot Cluster 1



(3) What is Flash Shield Window (FSW)?

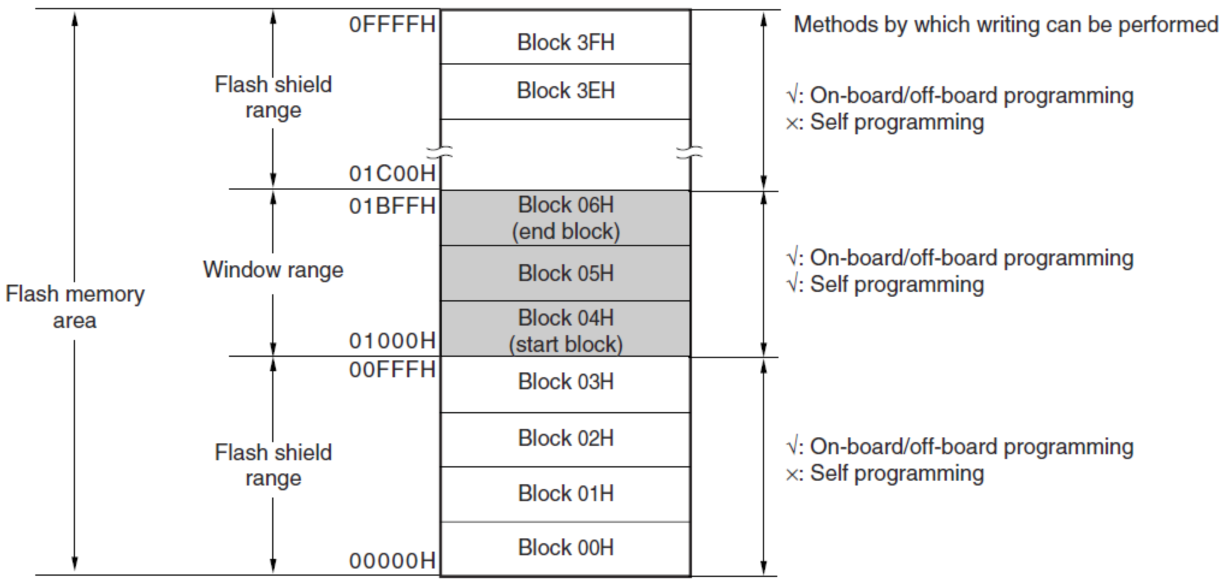
The Flash Shield Window is one of the security facilities that are to be used during flash self programming. It disables all the areas except the one that is defined by a specified window to be written or erased during flash self programming.

The areas other than the one that is specified as a flash shield window can be written or erased during on- or off-board programming.

A flash shield window can be set up using the FSL_SetFlashShieldWindow function. The window range can be defined by specifying the start and end blocks.

Figure 1-7 Example of Flash Shield Window Setup

(Target device: R5F100LE, Start Block: 08H, End Block: 0FH)



1.5 Relink Function

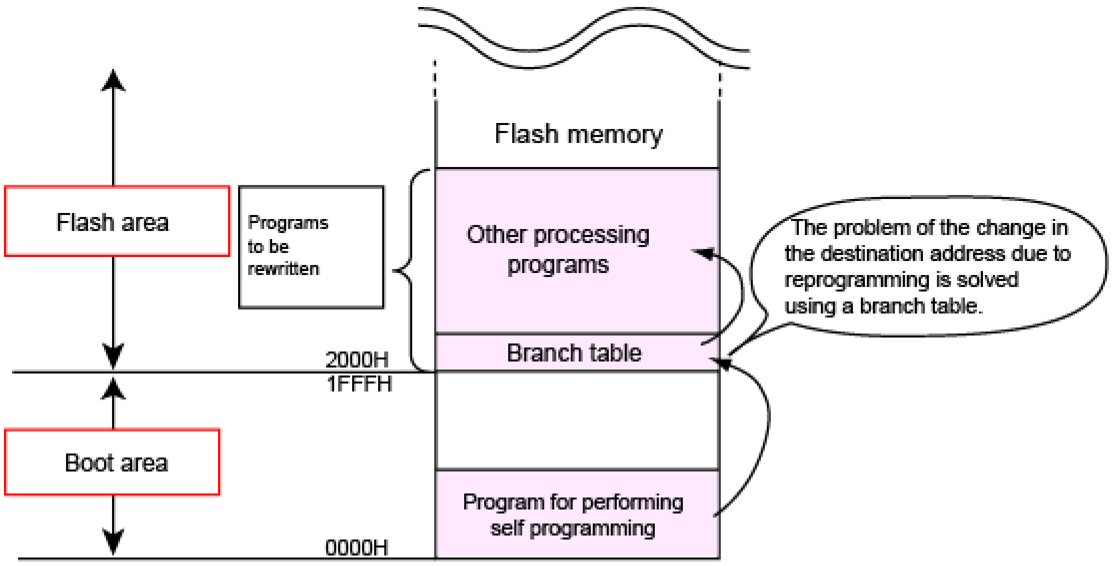
Some systems use areas that can be programmed or swapped (e.g., flash memory and external ROM) in addition to the areas that cannot be programmed or swapped (e.g., boot area).

CubeSuite+ offers a function, called the relink function that allows such a system to execute the function calls between the boot area and flash areas normally without reconfiguring the program in the boot area when only the program in a flash area is to be altered.

Using this relink function on the RL78/G13, it is possible to rewrite only part of the program that is allocated to the code flash memory using the flash self programming function. The code flash memory is split into two areas, i.e., the boot area that lies below the area storing the start addresses for the flash area and the flash area that lies above the area storing the start addresses for the flash area. The flash area can be programmed safely from the boot area side by allocating the program for flash self programming to the boot area and the other programs in the flash area that are subjected to programming.

The sample program covered in this application note makes use of this function to carry out its programming tasks. See chapter 2, Example of Configuring and Rewriting a Program, for instructions to create programs.

Figure 1-8 Outline of the Boot Area and Flash Area Configuration



- Branch table

To perform the processing that is relocated from the boot area to the flash area, it is necessary for the boot side to be aware of the allocation information about the processing to be used on the flash side. CubeSuite+ creates a table that stores a record of allocation information about the processing on the flash side in a specific area. Using the information stored in that table, the CubeSuite+ enables the processing on the boot side to carry out the processing on the flash side. This table is called the branch table.

The branch table contains the start addresses of the programs and interrupt vectors in the flash area that are subjected to programming. The branch table is updated as programs are written. This ensures that the program in the boot area can make function calls normally even when the start address of the functions in the programs in the flash area is altered.

Chapter 2 Example of Configuring a Program to Rewrite the Code Flash Memory

This chapter explains how to configure a program for rewriting flash memory using the flash self programming library RL78/G13 (R5F100LE) and gives an example.

2.1 Operating Environment of the Sample Program

The sample program covered in this application note consists of three components, i.e., the boot program that performs boot-time processing, the write program that rewrites programs and data using the flash self programming function, and a user program (for flashing LEDs).

The boot program performs basic initialization processing on the RL78/G13 (R5F100LE) at the time of booting up and checks the state of the switch on the QB-R5F100LE-TB to determine whether to start the processing for rewriting programs and data or to execute the LED flashing processing.

When the power supply for the target device is turned on or a reset is effected without the press of SW1, LED2 turns on and the processing for rewriting programs and data starts and waits for serial communication. Serial communication is controlled by SelfFlashWriter (see Appendix A for details) and LED1 flashes while communication is in progress. The processing program receives program code data through serial communication under control of SelfFlashWriter and updates the programs with the received data.

When the power supply for the target device is turned on or a reset is effected with SW1 being pressed, the user program starts and flashes LEDs. When SW1 is pressed after the target device is started, ASCII data is sent to the host machine through serial communication and a WDT reset is carried out at a reduced LED flashing speed. Pressing the switch again temporarily clears the WDT and the time up to the reset sequence is elongated.

Table 2-1 summarizes the characteristics of the I/O of the RL78 microcontroller and figure 2-3 shows the program configuration. Figures 2-1 and 2-2 show the operating environment of the sample program.

Figure 2-1 Outline of the QB-R5F100LE-TB Operating Environment of the Sample Program

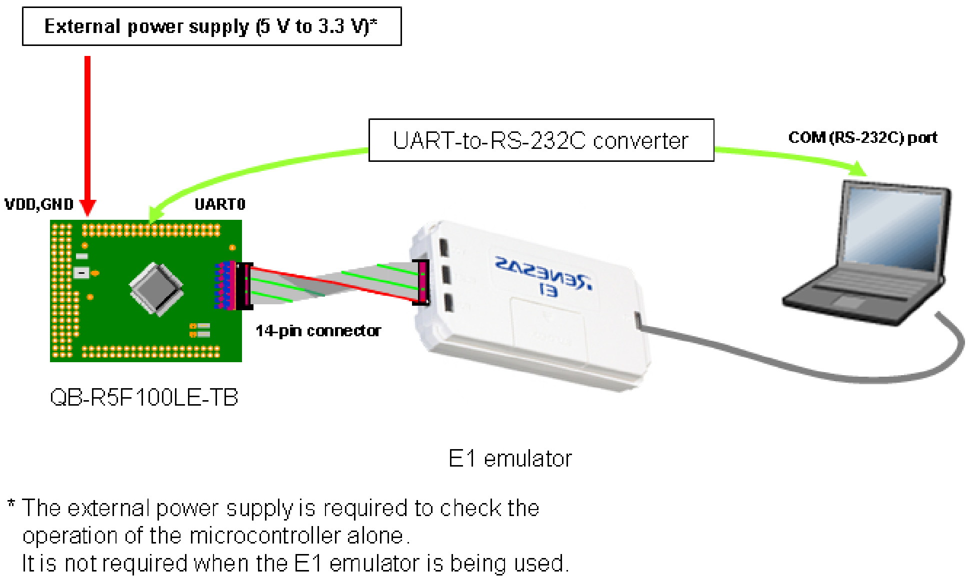
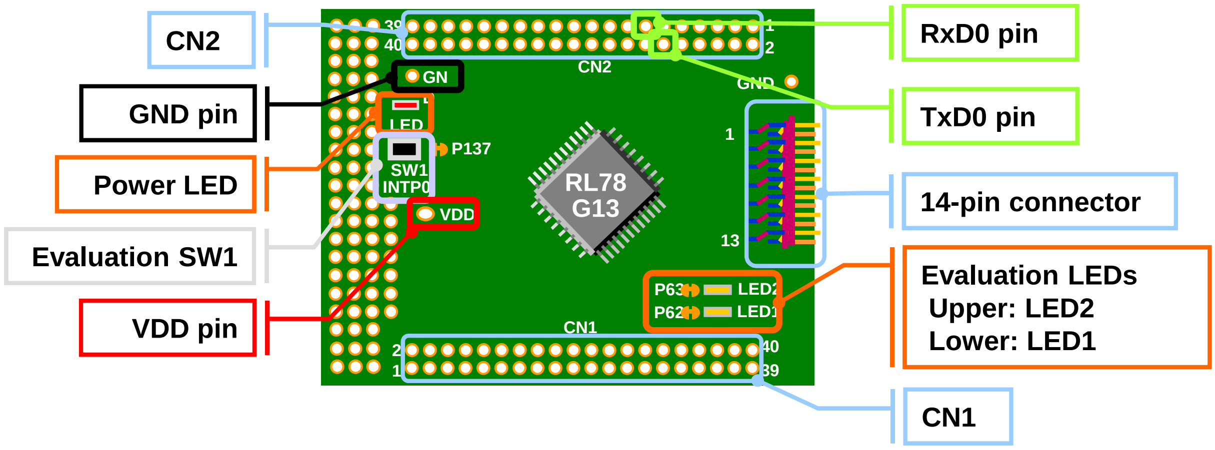


Figure 2-2 QB-R5F100LE-TB Pins that are Used by the Sample Program



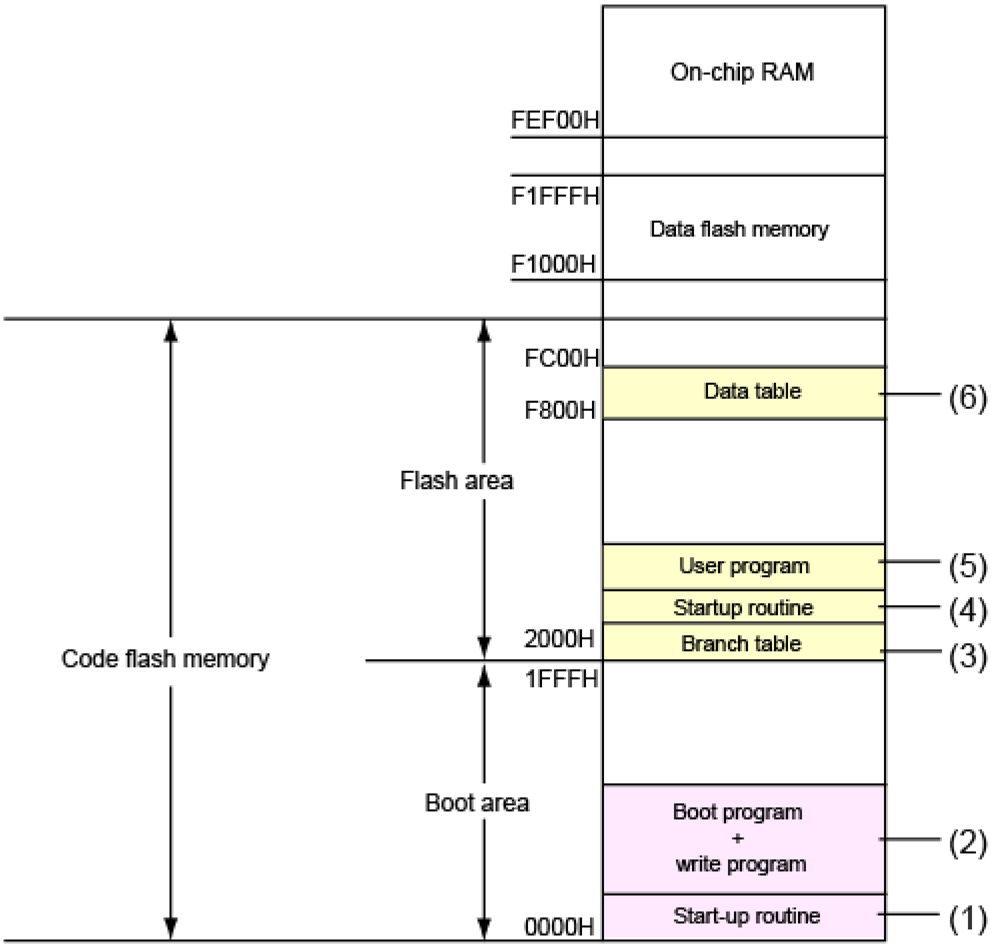
CN1/CN2: Connected to the microcontroller.
Power LED (LED3): Lit in red when power is turned on.
Evaluation LED1: Lit in yellow when port 62 (P62) is set low.
Evaluation LED2: Lit in yellow when port 63 (P63) is set low.
Evaluation SW1: Connected to INTP0.
14-pin connector: Connected to (optional) E1 and used for on-chip debugging or programming.
GND, VDD pins: Used to supply power to the target board.
RxD0 pin: UART RX pin. Connected to a COM port of the host machine for serial communication (data reception).*
TxD0 pin: UART TX pin. Connected to a COM port of the host machine for serial communication (data transmission).*

* A separate level shifter circuit is required to connect to a COM (RS-232C) port of the host machine.

Table 2-1 List of QB-R5F100LE-TB Pins that are Used by the Sample Program

Location	Use	I/O
Indicator 1 LED1	<Boot program> Initializes or turn off LED1 and LED2. <Write program> Turns on LED2 and flashes LED1 during communication. <User program> Turns on LED1 and flashes LED2 at constant intervals.	Port 62 (P62 pin)
Indicator 2 LED2		Port 63 (P63 pin)
Switch 1 SW1	<Boot program> Used to determine whether to run the write program or user program at boot time. <User program> Sends ASCII data to the host machine through serial communication, decreases the LED1 flashing interval, and performs a WDT reset. Pressing this switch again temporarily clears the WDT.	Port 137 (P137 pin)
Serial communication RxD0: P11 (CN2: 13 pin) TxD0: P12 (CN2: 12 pin)	Performs serial communication with the host machine using the RL78/G13 microcontroller's UART0 port. A separate level shifter circuit is required to connect to a COM (RS-232C) port of the host machine. <Communications specifications> Bits/s: 115200 Data bits: 8 Parity: None Stop bits: 1 Flow control: None	RXD0 (P11 pin) TXD0 (P12 pin)

Figure 2-3 Outline of the Sample Program in the Code Flash Memory



(1) Boot area startup routine

Startup routine for initializing the boot area.

(2) Boot program + write program

A program that performs initialization program at the time of booting up and a program that rewrites the flash memory in the boot area using the flash self programming library.

(3) Branch table

Branch table used to make accesses from the boot area to the flash area.

(4) Flash area startup routine

Startup routine for initializing the flash area.

(5) User program

A program in the flash area that turns on LED1 and flashes LED2 at constant intervals.

(6) Data table

Area that is used by the user program and that contains ASCII data to be sent to the host machine.

Figure 2-4 Operating Environment (Outline) of the Sample Program (in Programming Mode)

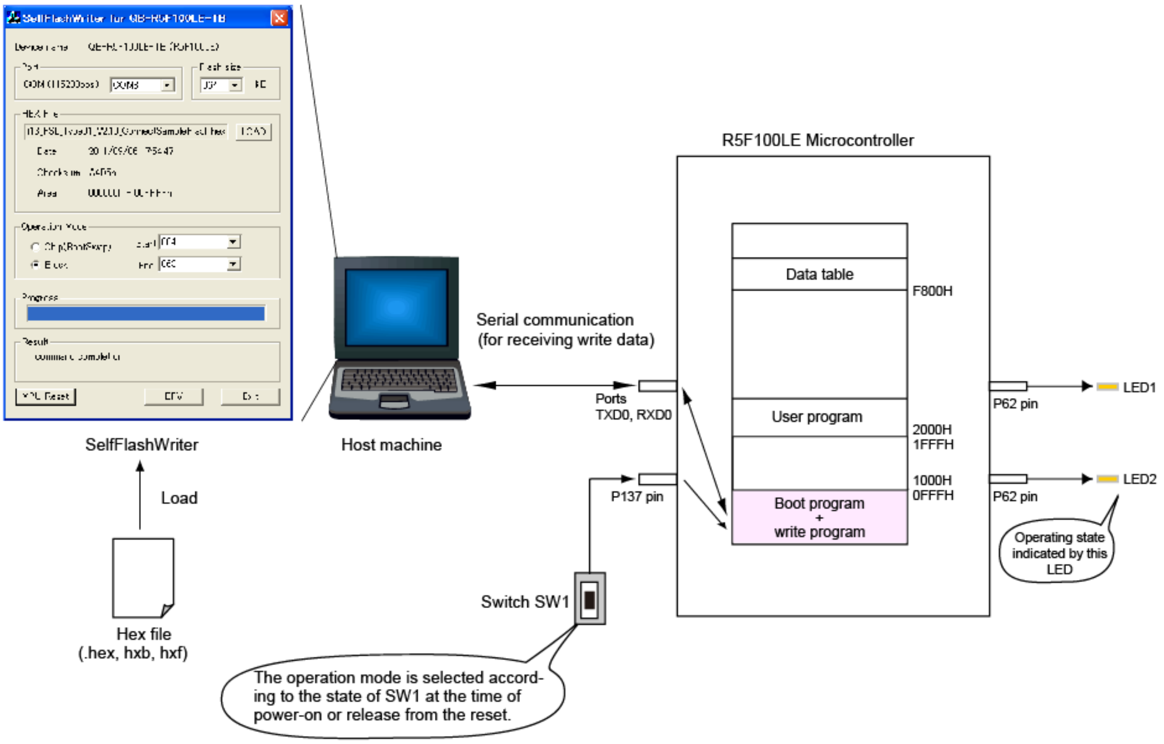
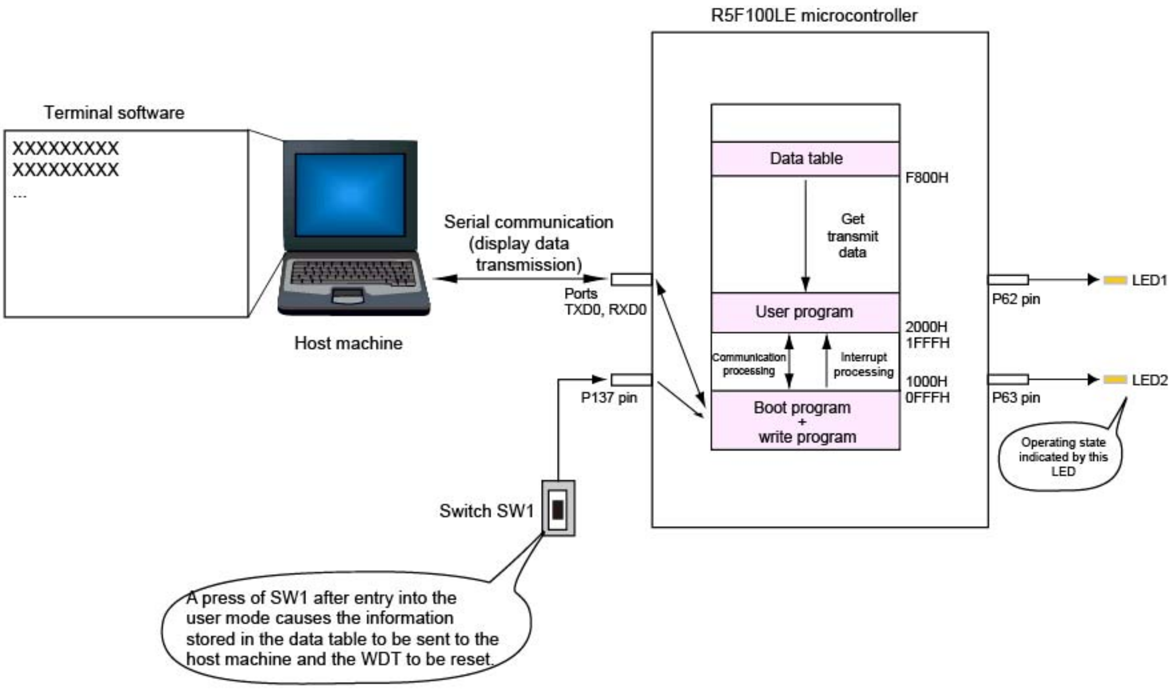


Figure 2-5 Operating Environment (Outline) of the Sample Program (in User Mode)

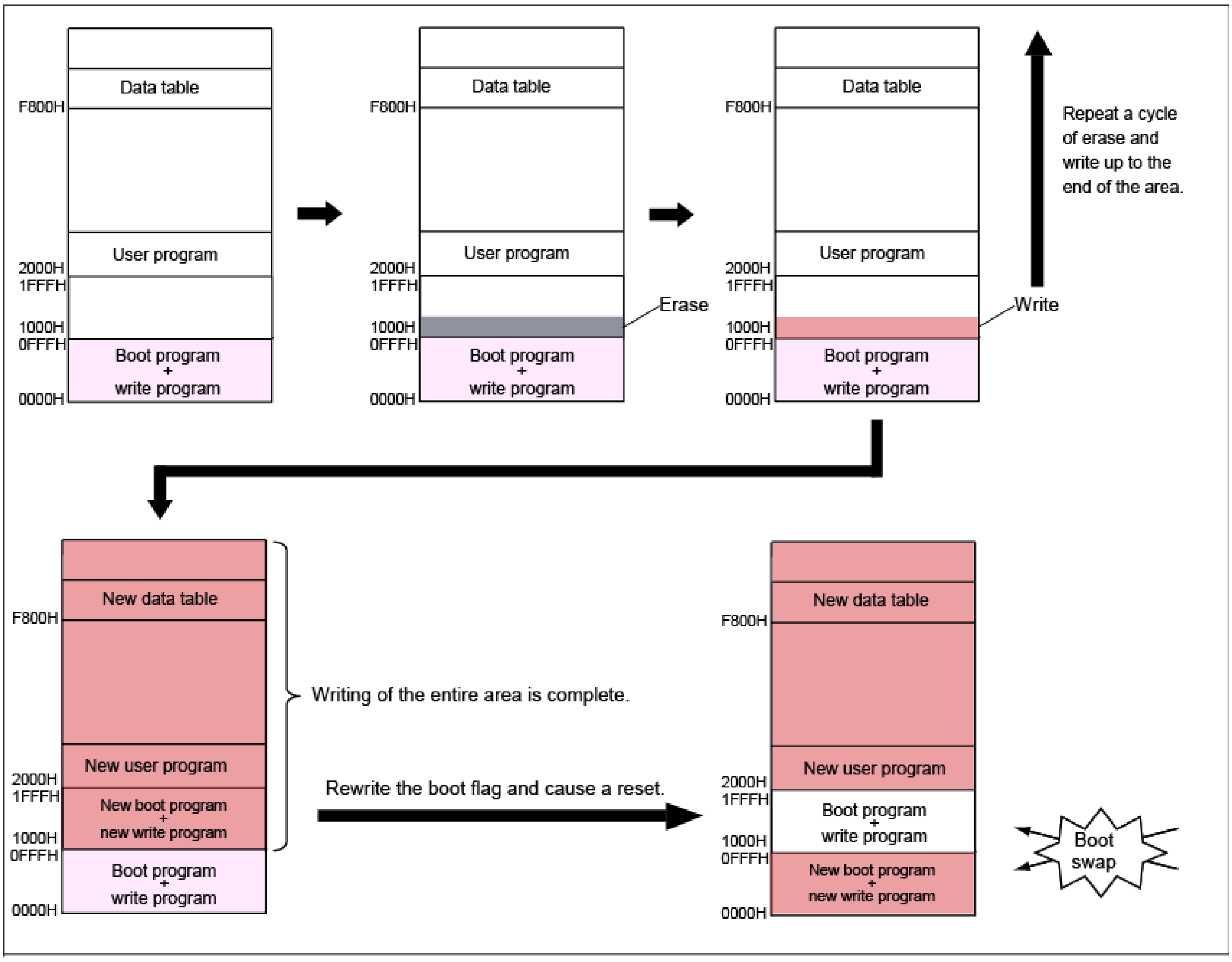


2.2 Flash Programming Operation Flow

The operation flows of rewriting programs and data by the sample program are given in figures 2-4, 2-5, and 2-6.

The write program that performs flash self programming is placed in the boot cluster area (blocks 0 to 3).

Figure 2-6 Rewriting the Entire Program (Outline)



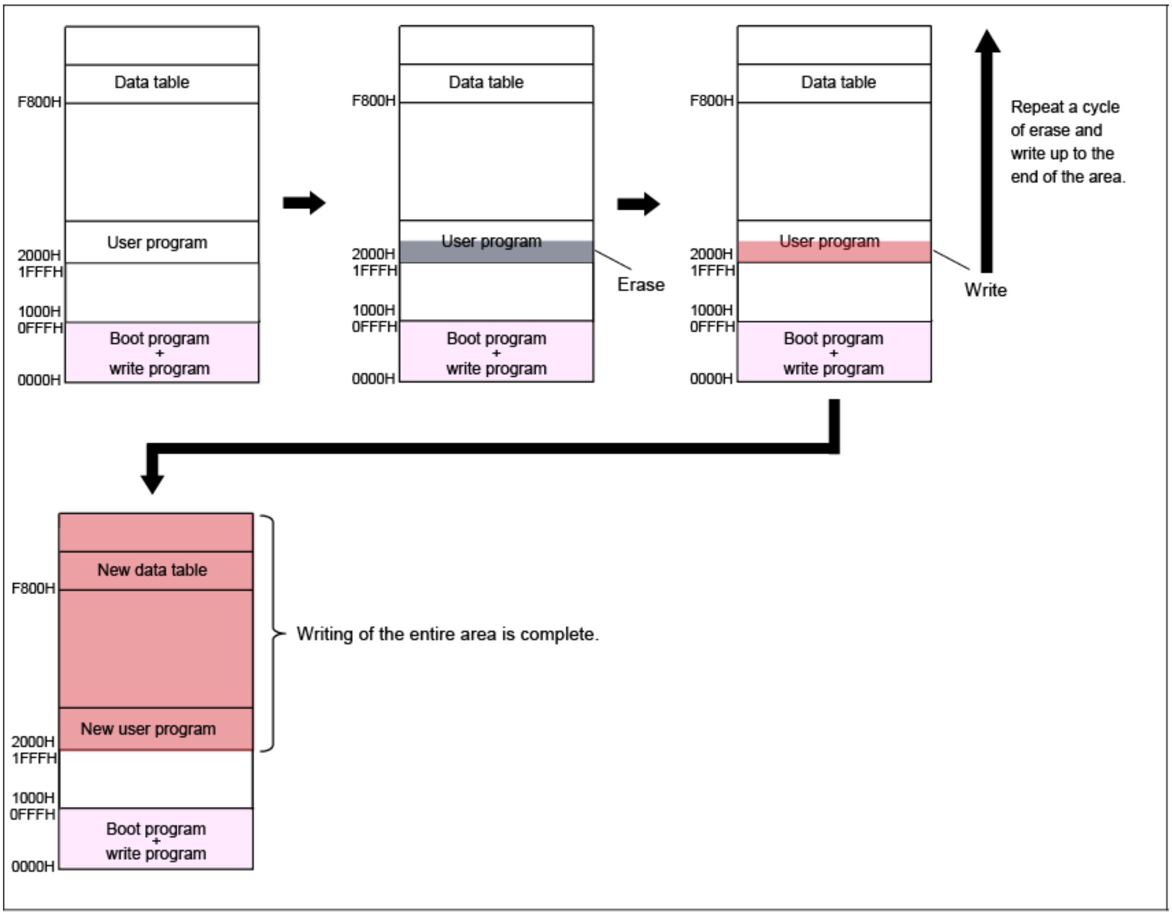
(1) Writing all program data that is received

The RL78/G13 (R5F100LE)'s write program (a program that performs flash self programming) writes the new boot program and write program that are received from SelfFlashWriter into blocks 4 to 7 (1000H-1FFFH) and the user program and data table into block 8 (2000H-FFFFH) and later blocks.

(2) Performing a boot swap

SelfFlashWriter, after confirming that the programming of all data is completed, sends out a BOOTSWAP command. The RL78/G13 (R5F100LE)'s write program makes boot swap settings using the flash self programming library and effects a reset upon completion. After the reset is effected, a boot swap is carried out and the new boot program is executed.

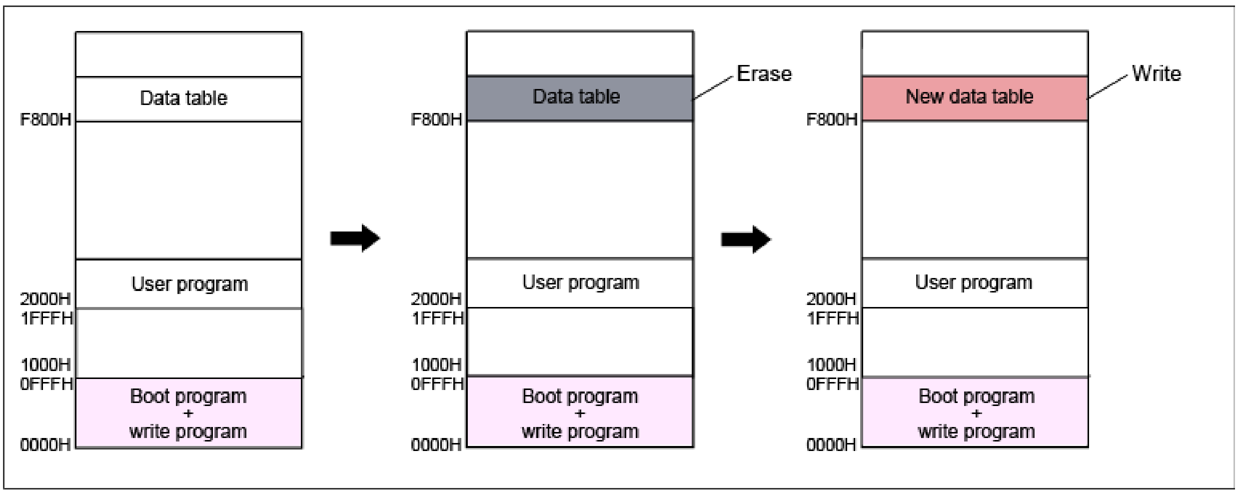
Figure 2-7 Rewriting Parts of a Program (Flash Area) (Outline)



- Writing all program data that is received

  The RL78/G13 (R5F100LE)'s write program (a program that performs flash self programming) writes the new user program and data table that is received from SelfFlashWriter into block 8 (2000H-FFFFH) and later blocks.

Figure 2-8 Rewriting of Data (Outline)



- Writing received data table and program

  The RL78/G13 (R5F100LE)'s write program (a program that performs flash self programming) writes parts of the new data table and program received from SelfFlashWriter.

2.3 File Configuration of the Sample Program

The file configuration of the sample program is given in table 2-2. To load a project into CubeSuite+, start CubeSuite+ and open the file r_fsl_praxis01.mtpj.

The sample project file has been generated in the folder C:\Program Files\Renesas Electronics\Flash Libraries. It assumes that the Flash Self Programming Library Type01 V2.20 is installed. If the installation folder of the library is different, change the registered destinations of the library-related files accordingly after starting the project.

Table 2-2 Sample Program File Configuration (Folder Name: R01AN0718_PRAXIS01)

File Name			Description
root		r_fsl_praxis01.mtpj	Project file
		r_fsl_praxis01_boot.mtsp	Boot area sub-project file
		r_fsl_praxis01_flash.mtsp	Flash area sub-project file
\DefaultBuild		r_fsl_praxis01_boot.hex	Boot area project HEX format ifle
		r_fsl_praxis01_flash.hex	Flash area project hex file • All-area hex file (hex)
		r_fsl_praxis01_flash.hxb	Flash area project hex file • Boot area hex file (hxb)
		r_fsl_praxis01_flash.hxf	Flash area project hex file • Flash area hex file (hxf)
\TestData		r_fsl_praxis01_write_test.hex	Test HEX format file • Programming check hex file (Invert LED1/LED2 display mode)
		r_fsl_praxis01_boot_write_test.hxb	Test HEX (hxb) format file • Boot area hex file (hxb) (Invert LED1/LED2 display mode)
		r_fsl_praxis01_flash_write_test.hxf	Test HEX (hxf) format file • Flash area hex file (hxf) (Invert LED1/LED2 display mode)
\inc		r_fsl_praxis01_com.h	Program common header file
		r_fsl_praxis01_BranchTable.h	Branch table configuration file
		r_fsl_praxis01_BootSection.h	Boot area section configuration file
		r_fsl_praxis01_FlashSection.h	Flash area section configuration file
\src	\boot	r_fsl_praxis01_boot_main.c	Boot area main processing
		r_fsl_praxis01_boot_write.c	Boot area write processing
	\flash	r_fsl_praxis01_flash_main.c	Flash area main processing
\dr		r_fsl_praxis01_boot_map.dr	Boot area link directive file
		r_fsl_praxis01_flash_map.dr	Flash area link directive file

2.4 Resources of the Sample Program

The reference information for resources of the sample program is given in tables 2-3 to 2-5.

These information are reference information, it is different from the actual values.Please check the map file that generated at compiling about detail information.

Table 2-3 Overall Resources of the Sample Program(Reference information)

Area Name	ROM Area Range	Occupied ROM Size	Occupied RAM Size	Remarks
Boot area	0H to FFFH	3000 bytes	900 bytes	Including the size of areas for vector bytes, option bytes, and the library
Flash area	2000H to FBFFH	600 bytes	6 bytes	Excluding the size of OCD monitor area

Table 2-4 Resource for the Sample Program's Boot Area(Reference information)

Area Used		Item	Total Size
ROM		Area for interrupt vector and option bytes	3000 bytes
		Startup routine and run-time library	
		Standard library (memcpy_f, memset_f)	
		Flash self programming library	
		Boot program, write program, interrupt processing	
RAM		Startup routine and run-time library	750bytes
		Boot program, write program, interrupt processing	
	Stack	Boot program, write program	150 bytes
		Interrupt processing	
		Flash self programming library	

Table 2-5 Resource for the Sample Program's Flash Area(Reference information)

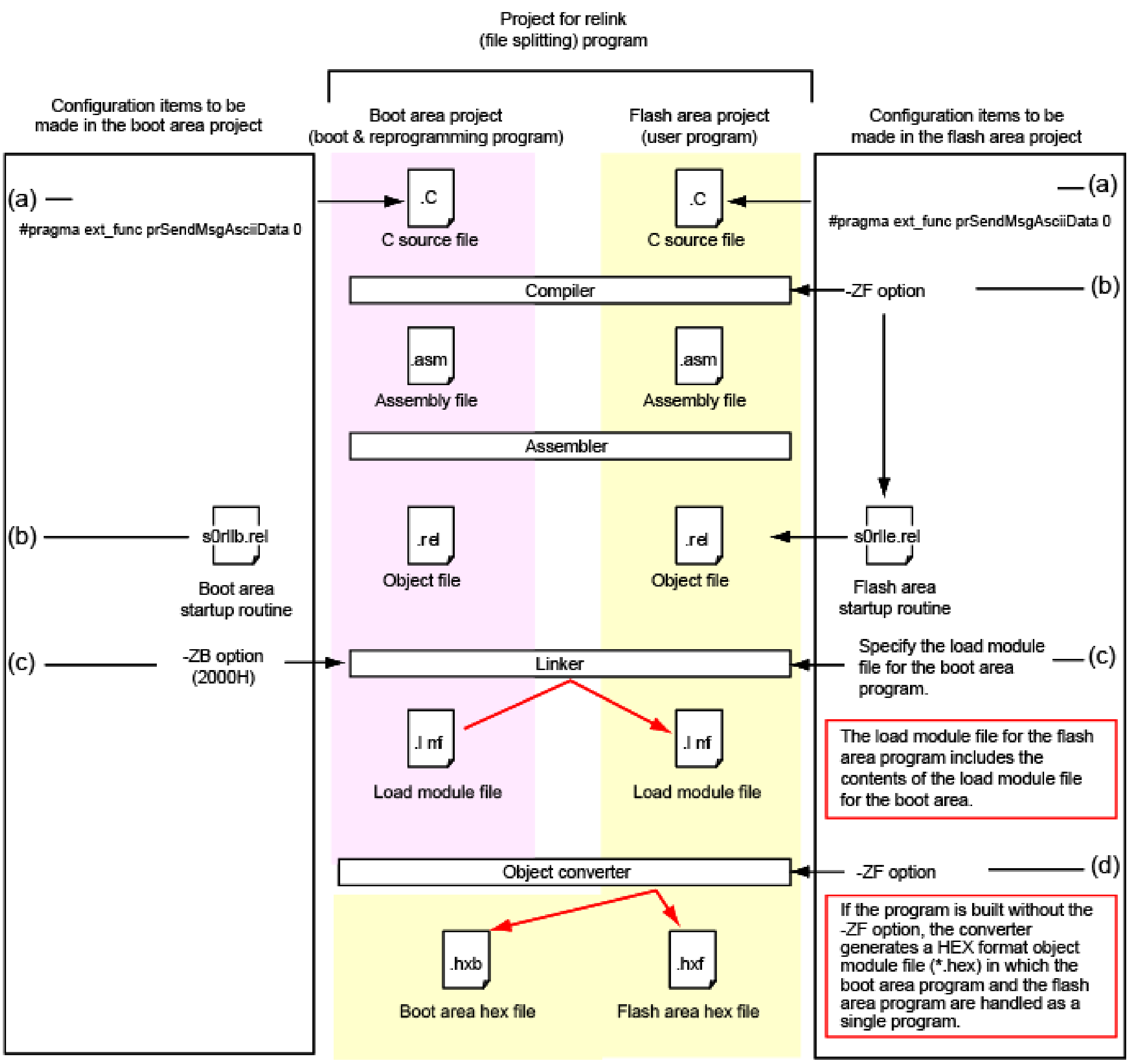
Area Used		Item	Total Size
ROM		Branch table (vector addresses, branch-table-registered functions)	600 bytes
		Start-up routine and run-time libraries	
		User program	
		Data table(F800H to FBFFH)	
RAM		User program	50 bytes
	Stack	User program	50 bytes
		Interrupt processing	

2.5 Configuring Projects (Relink Function Configuration)

One program can be split into the boot and flash areas during the development stage by specifying certain options of the RL78's assembler/compiler.

When developing a program under CubeSuite+, use two separate projects. Since the load module file (.lmf) for the program in the boot area is linked with the load module file for the program in the flash area during the program generation stage, it is necessary to have the program in the boot area built in advance. Figure 2-9 shows the outline of the relink function and figures 2-10 to 2-19 illustrate the steps for manipulating CubeSuite+ that are necessary for the relink function configuration.

Figure 2-9 Example of Project Configuration under CubeSuite+ and Relink Function Configuration



(1) Configuration for the boot area project (boot program and write program)

(a) Setting up a function call from the boot area to the flash area using an extension function (#pragma)

Describe the entry of the function to the branch table in the C source code. The function to be added to the table is the SW1 interrupt function which is also used by the user program.

This setting enables the program in the boot area to run the programs (functions) in the flash area.

Figure 2-10 Calling a Function from the Boot Area to the Flash Area (r_fsl_praxis01_BranchTable.h)

```
.
.
.
/*------------------------------------------------------------------------*/
/* (#pragma) branch table functions (ext_func)                          */
/*------------------------------------------------------------------------*/
#pragma ext_func prSendMsgAsciiData 0
.
.
```

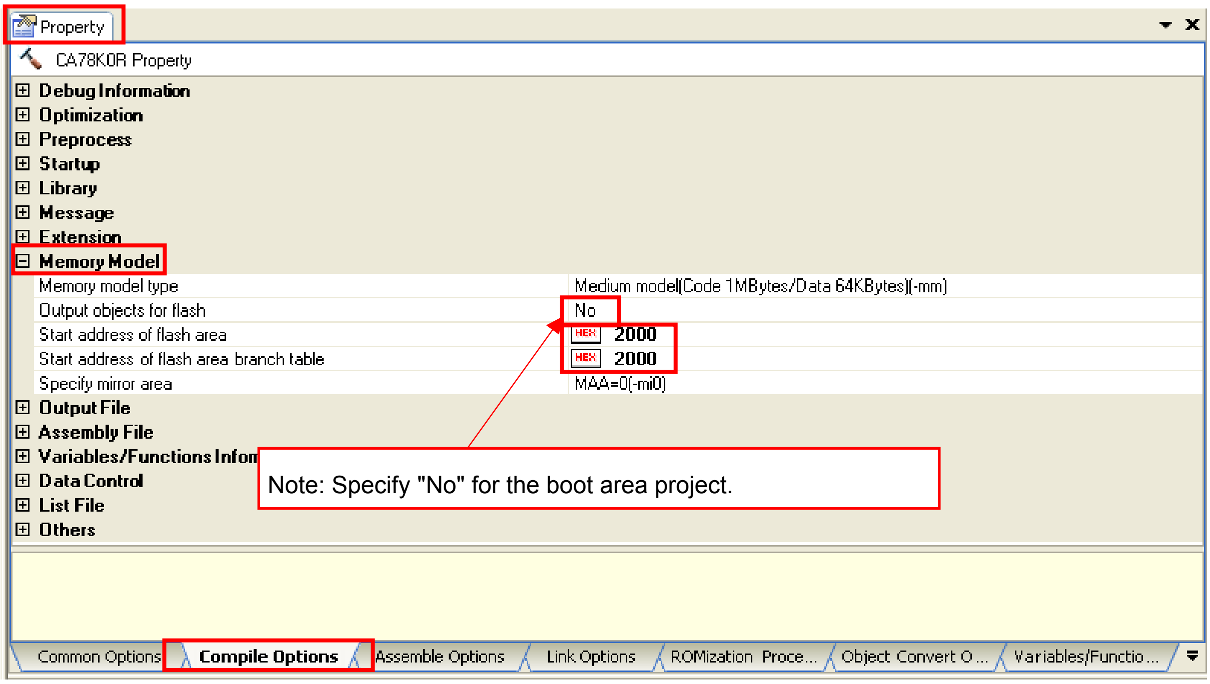
Add function to branch table.
#pragma ext_func func_name ID No.

Describe the name of the desired function allocated to the flash area
* A prototype declaration is separately required.

(b) Specifying the start addresses of the branch table and flash area

Specify the start addresses of the branch table and flash area with the CA78K0R's compiler option.

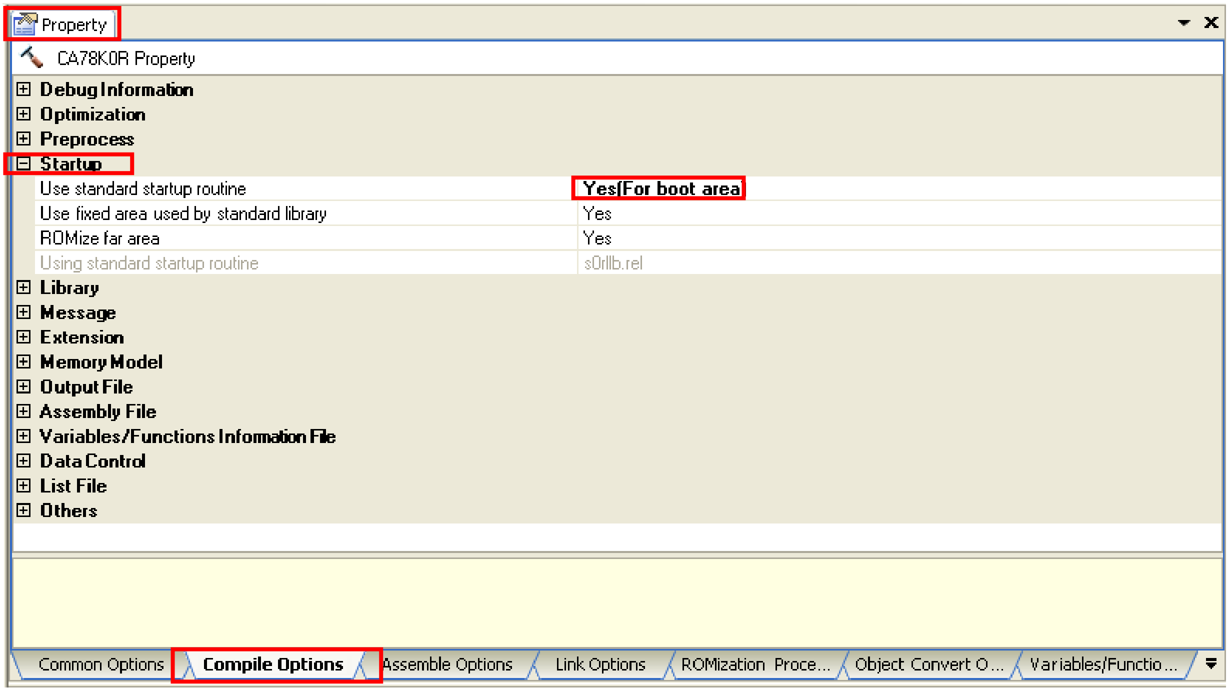
Figure 2-11 Specifying the Start Addresses of the Branch Table and Flash Area



(c) Specifying the startup routine for the boot area project

Specify the startup routine for the boot area project.

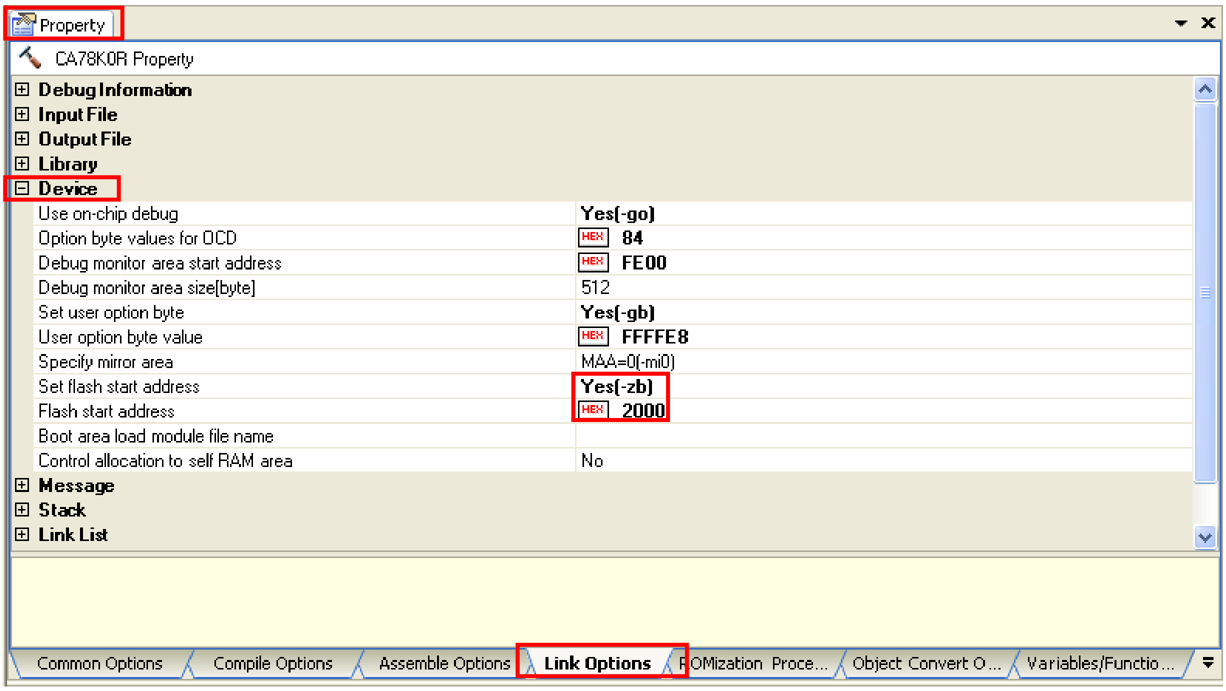
Figure 2-12 Specifying the Startup Routine for the Boot Area



(d) Setting the –ZB linker option

Set the linker's –ZB option to specify the start address of the flash area.

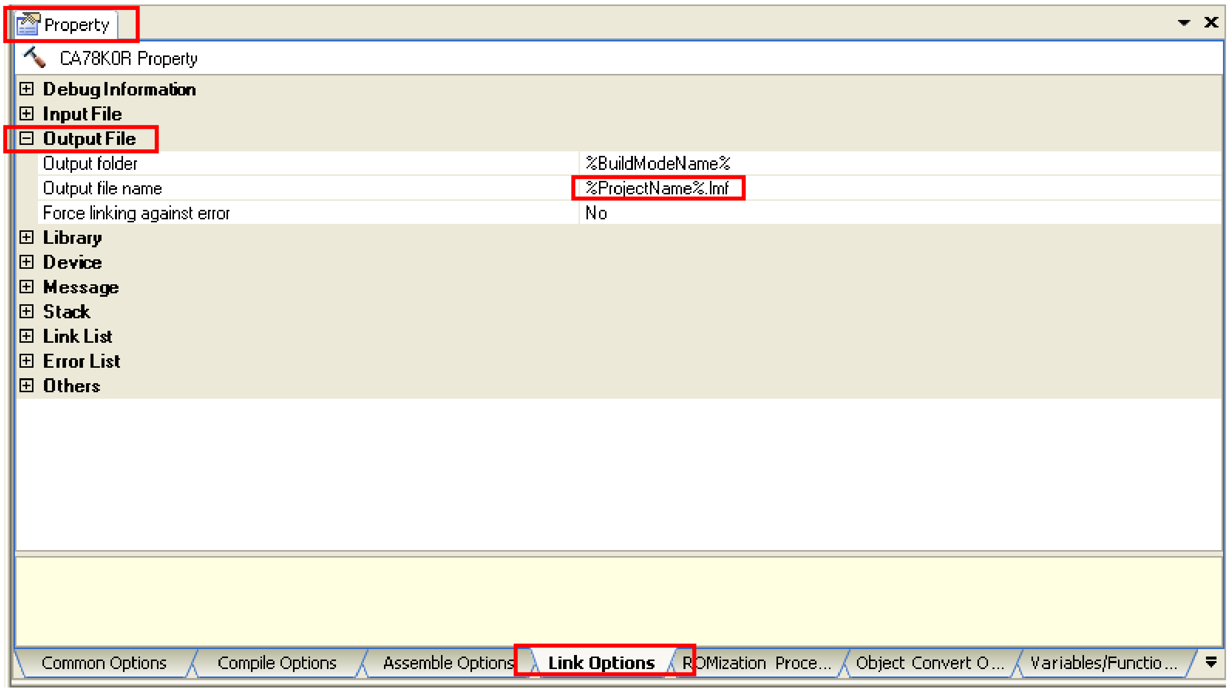
Figure 2-13 Specifying the Linker –ZB Option



(e) Setting the name of the load module for the boot area project

Since the flash area project generates the total module using the load modules in the boot area project, if a specific name is necessary for the load module in the boot area project, specify it so that the load module can be identified by the flash area project. By default, the name is "<project name>.lmf."

Figure 2-14 Specifying the Name of the Load Module in the Boot Area Project



(f) Confirming the boot area program to be run from the flash area

For the program allocated to the boot area to be run from the flash area, the program that is to be allocated to the boot area needs to be included in the load module file (*.lmf) for the boot area. If the target program is not used on the boot area, however, it may not be linked at the time of linkage on the boot area.

If an attempt is made to run the target program from the flash area under this situation, the building of the program for the flash area fails leading to a linkage error because the target program is not included in the load module file (*.lmf) for the boot area.

If there is a boot area program that needs to be run from the flash area and the target program is not used on the boot area, it is necessary to make the program so that the target program can be included in the load module file (*.lmf) for the boot area by, for example, creating a dummy function that runs the target program.

(2) Configuration for the flash area project (user program)

(a) Setting up a function call from the boot area to the flash area using an extension function (#pragma)

Describe the entry of the function to the branch table in the C source code. The function to be added to the table is the SW1 interrupt function which is also used by the user program. Add the same description in the program for the flash area.

Figure 2-15 Calling a Function from the Boot Area to the Flash Area (r_fsl_praxis01_flash_main.c)

r_fsl_praxis01_BranchTable.h :

```
.
.
/*----------------------------------------------------------------------*/
/* (#pragma) branch table functions (ext_func)                         */
/*----------------------------------------------------------------------*/
#pragma ext_func prSendMsgAsciiData 0
.
.
```

Add function to branch table.
#pragma ext_func func name ID

Function body to add

r_fsl_praxis01_flash_main.c:

```
.
/*----------------------------------------------------------------------*/
/* Include common files                                                 */
/*----------------------------------------------------------------------*/
/*  */
#include "r_fsl_praxis01_BranchTable.h"
```

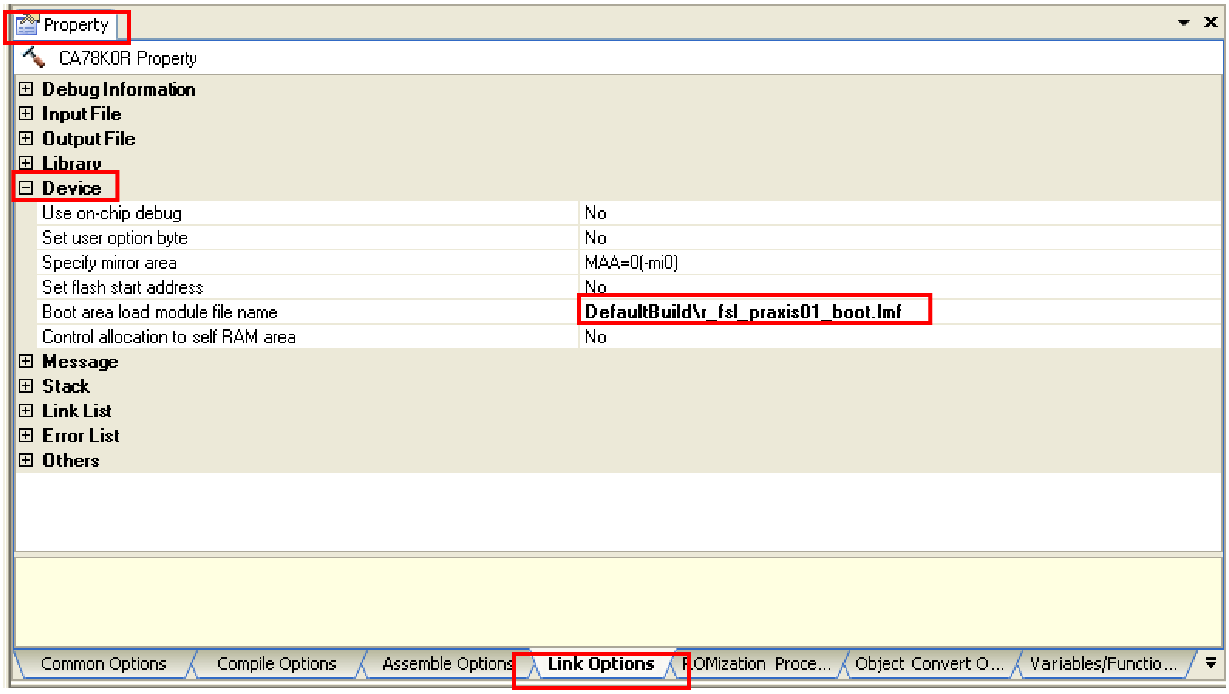
Omitted

```
. void prSendMsgAsciiData( void )
{
    UH  duh_i;
    UB  dubSendData[5];

    /*  */
    for( duh_i = 0 ; duh_i < sizeof( prFcubSendMsgData ) ; duh_i++ )
    {
        dubSendData[0] = prFcubSendMsgData[ duh_i ];
        prUartSendData( &dubSendData[0] );
    }

    /*  */
    prDuhSwNum++;
    if( prDuhSwNum > 999 )
    {
        prDuhSwNum = 0;
    }

    /*  */
    dubSendData[0] = (UB)( prDuhSwNum / 100 ) | 0x30;
    dubSendData[1] = (UB)( ( prDuhSwNum % 100 ) / 10 ) | 0x30;
    dubSendData[2] = (UB)( prDuhSwNum % 10  ) | 0x30;
    dubSendData[3] = '\n';
    dubSendData[4] = '\r';

    /*  */
    for( duh_i = 0 ; duh_i < 5 ; duh_i++ )
    {
        prUartSendData( &dubSendData[duh_i] );
    }

#ifdef PR_USE_OCD_MODE

#else
    /*  */
    PR_WD_INT_OFF();

    /*  */
    if( prDuwLedTime <= PR_LED_DEFAULT_WAIT )
    {
        prDuwLedTime = PR_LED_DEFAULT_WAIT * PR_LED_WAIT_MAG;
    }
#endif
}
```

(b) Setting up the load module file for the boot area project

Set up the load module file (.lmf) for the boot area project that is to be used in the flash area project.

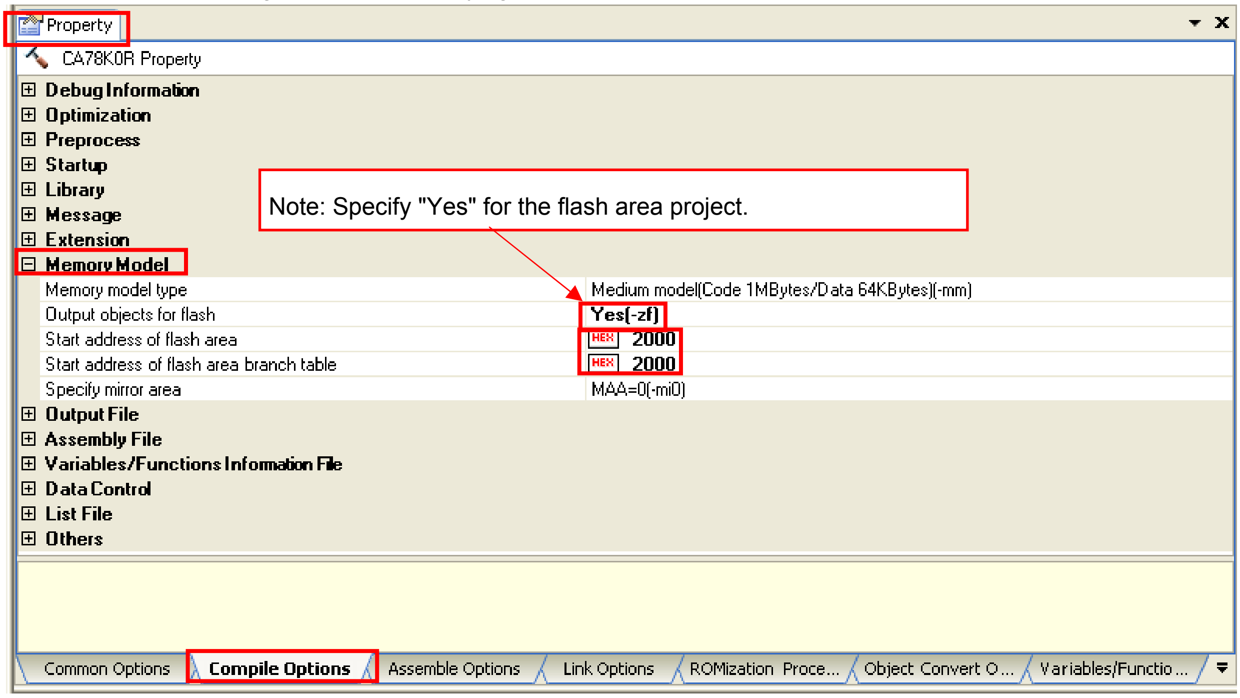
Figure 2-16 Setting up the Load Module File for the Boot Area Project



(c) Specifying the start addresses of the branch table and flash area

Specify the start addresses of the branch table and flash area to be used in the flash area project.

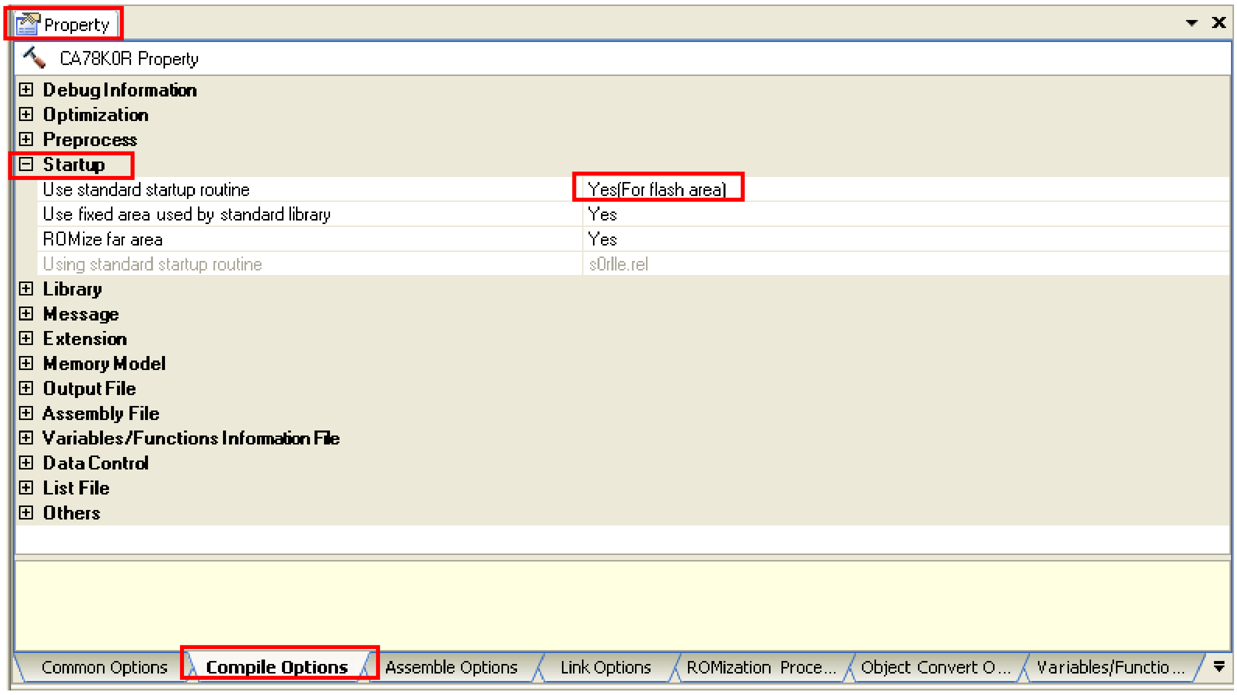
Figure 2-17 Specifying the Start Addresses of the Branch Table and Flash Area



(d) Specifying the startup routine for the flash area project

Specify the startup routine for the flash area project.

Figure 2-18 Specifying the Startup Routine for the Flash Area Project

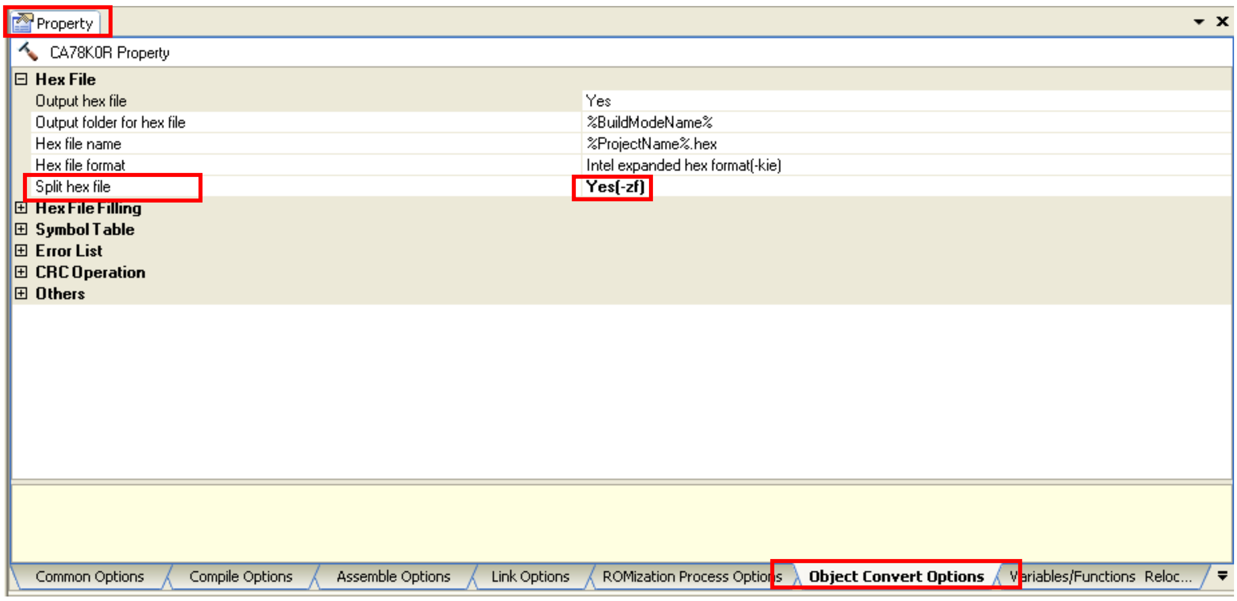


(e) Setting up a hex file (only when the hex file is to be split on output)

Specify the –ZF option of the object converter. When this option is specified, the program in the boot area and the program in the flash area are output to separate hex format object module files.

The output file for the boot area program is given a file extension of .hxb and the output file for the flash area program a file extension of .hxf.

Figure 2-19 Specifying the Object Converter –ZF Option

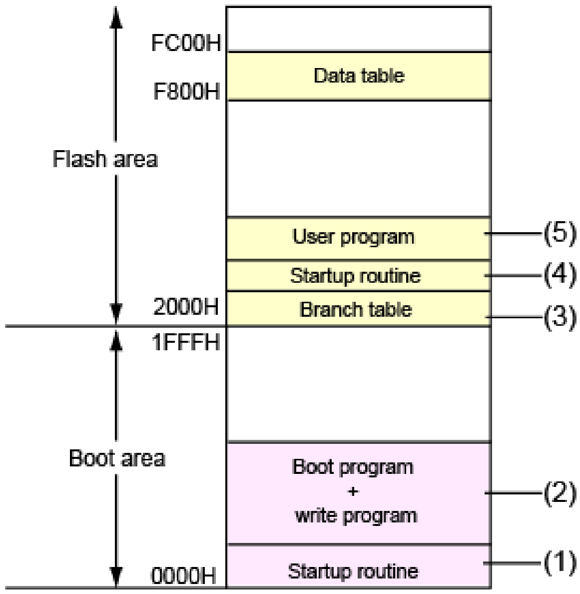


2.6 Configuration for Processing from Reset Release to Main Processing

Since a program that uses the relink function has its startup routine placed in both the boot and flash areas, its behavior during the period from immediately after the reset is released up to their main processing differs from that of ordinary programs. Such a program must be coded so that the startup routine for the boot area and the startup routine for the flash area are executed without fail as shown in (1) to (3) in figure 2-20.

The main function and subsequent functions ((5) in the figure below) in the flash area must be executed according to their programming specifications.

Figure 2-20 Sequence of Program Execution



(1) Startup routine for the boot area

The startup routine for the boot area is executed after the reset is released. After the data for the boot area is initialized, the main function (boot_main() = main function that is started by the startup routine for the boot area) of the program in the boot area is executed.

(2) Main function (boot_main()) on the boot area

The main function (boot_main()) performs the QB-R5F100LE-TB's basic initialization processing as a boot program and checks the state of the switches on the QB-R5F100LE-TB to determine whether rewriting of data is to be executed. When the user program is to be run, the main function ends and returns control to the startup routine for the boot area, without doing anything.

(3) Jumping into the branch table

When the main function (boot_main()) ends, control is returned to the startup routine for the boot area, then to the branch table on the flash area. No normal processing can be continued if the branch table does not exist in the required location.

(4) Branch table

Execution branches to the startup routine for the flash area.

(5) Startup routine for the flash area

After initializing the data for the boot area, the startup routine for the flash area causes a jump to the main function for the flash area (main() = main function that is started by the startup routine for the flash area).

Subsequently, code the program according to their programming specifications. Figures 2-21 and 2-22 show the processing of the sample program.

Figure 2-21 Program Execution Sequence

Boot area startup routine (Initializing the area to be used for the boot area)

↓

Program for the boot area

```
void boot_main(void)
{
      if( PR_BOOT_SW != PR_SW_ON )
      {
            UB dubRetCode;

            prBootTargetStart();                     /* Target start processing */

            dubRetCode = prBootWriteProgram(); /* Run the write program (flash self programming).
*/

            prBootTargetEnd( dubRetCode );           /* Target termination processing */
      }
}
```

↓

Boot area startup routine (Jump to branch table)

↓

Branch table (Jump to flash area startup routine)

↓

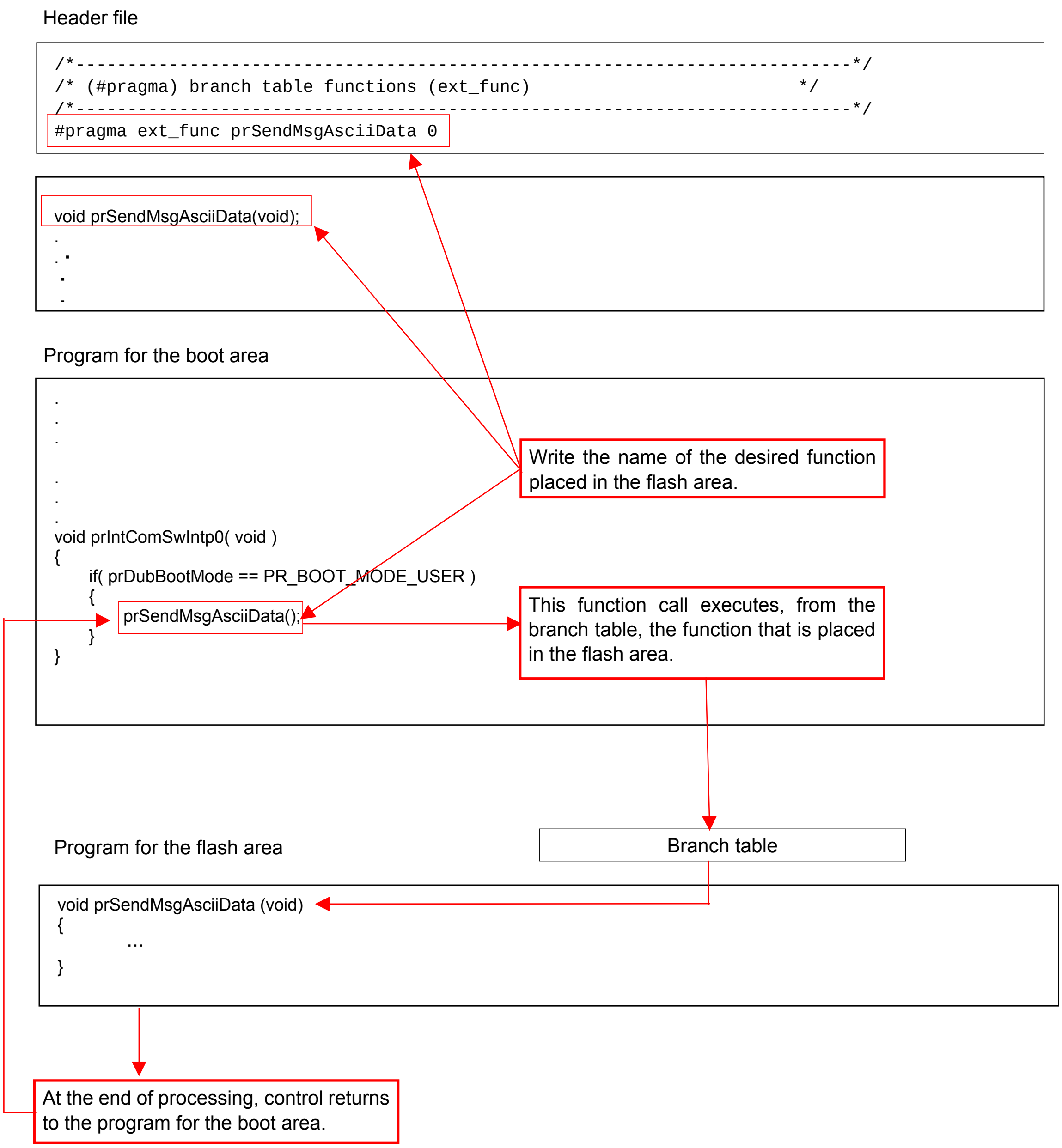
Flash area startup routine (Initializing the area to be used for the flash area)

↓

Program for the flash area

```
void main( void )
{
      UB dubLed2flag;
      UW duw_i;

      prFlashTargetStart();
      dubLed2flag    = true;

      /* Main loop */
      while(1)
      {
            /* LED turn-on processing */
            if( dubLed2flag == false )
            {
                  dubLed2flag = true;
                  PR_LED2_ON();
            }
            else {
                  dubLed2flag = false;
                  PR_LED2_OFF();
            }

            /*    */
            for( duw_i = 0 ; duw_i < prDuwLedTime ; duw_i++ )
            {
                  prWait10clcok();
            }
      }

      prFlashTargetEnd();
}
```

Figure 2-22 Example of Using the Branch Table



2.7 Details of the Main and Other Functions

A program listing of the sample program is given below. (Mainly, the program code that is related to flash self programming is given. For the other processing, refer to the sample program code itself.)

Some programs such as the one for switch testing can be altered through the header file (r_fsl_praxis01_com.h). By using programs of different specifications, you can see if flash self programming has successfully completed rewriting of the original program (for the procedure, see section 2.9, How to Evaluate Rewriting of Programs).

Listing 2-1 Header File (r_fsl_praxis01_com.h)

```
.
.
.
/**********************/
/* Definitions common to all sample programs */
/**********************/
/* Area definitions */
#define PR_MAX_BLOCK_NUM       64        /* Maximum number of blocks   */
#define PR_BLOCK_SIZE          0x400     /* Block size */
#define PR_WORD_SIZE           4         /* Word size */

/* SW test definitions */
#define PR_SW_ON               0         /* Switch polarity  */

/* operating mode */
#define PR_BOOT_MODE_UNKNOWN    0         /* Boot unknown         */
#define PR_BOOT_MODE_WRITE      1         /* Programming mode */
#define PR_BOOT_MODE_USER       2         /* User mode */
.
.
.
```

Change the switch polarity when it is to be changed at the time of booting up.

According to the settings defined in the link directive file, the write program that performs flash self programming is placed in blocks 0 to 3 (boot cluster 0) and the user program is placed in block 8 and subsequent blocks.

When performing flash self programming, it is necessary to configure the link directive file so that these programs be placed in areas other than the RAM areas whose use is restricted by the flash self programming library. For the RAM areas that are to be used with the flash self programming library, refer to the flash self programming library user's manual.

Listing 2-2 Boot Area Link Directive File (r_fsl_praxis01_boot_map.dr)

```
;************************************************************
; Redefined ROM area
;************************************************************
; ----------------------------------------------------------
; Redefined default data segment ROM
; ----------------------------------------------------------
MEMORY ROM      : ( 000000H, 001000H )     Boot area definition

; ----------------------------------------------------------
; Define new memory entry for OCD Monitor area
; ----------------------------------------------------------
MEMORY OCD_ROM   : ( 00FC00H, 000400H )    OCD monitor area definition

;************************************************************
; Redefined RAM area
;************************************************************
; ----------------------------------------------------------
; Define new memory entry for self-RAM
; ----------------------------------------------------------   Self RAM area
MEMORY SELFRAM   : ( 0FEF00H, 000400H )    → Separate the self restricted area from the standard RAM area.

; ----------------------------------------------------------
; Redefined default data segment RAM
; ----------------------------------------------------------
MEMORY RAM      : ( 0FF300H, 000B20H )     Standard RAM area redefinition

; ----------------------------------------------------------
; Define new memory entry for saddr area
; ----------------------------------------------------------   SADDR (short addressing register) area
MEMORY RAM_SADDR : ( 0FFE20H, 0001E0H )    →Separate the self restricted area from the standard RAM area.
```

Listing 2-3 Flash Area Link Directive File (r_fsl_praxis01_flash_map.dr)

```
;************************************************************
; Redefined ROM area
;************************************************************
; ----------------------------------------------------------
; Redefined default data segment ROM
; ----------------------------------------------------------
MEMORY ROM      : ( 000000H, 00F800H )     Flash area definition (need to be defined so that it overlaps with
                                           the boot area.)
; ----------------------------------------------------------
; Define new memory entry for OCD Monitor area
; ----------------------------------------------------------
MEMORY ROM_DATA  : ( 00F800H, 000400H )    Data table area definition

; ----------------------------------------------------------
; Define new memory entry for OCD Monitor area
; ----------------------------------------------------------
MEMORY OCD_ROM   : ( 00FC00H, 000400H )    OCD monitor area definition (also need to be defined for the flash
                                           area.)
;************************************************************
; flash segment
;************************************************************
; ----------------------------------------------------------
; Merge FLAS_CNF segment
; ----------------------------------------------------------
MERGE FLAS_CNF := ROM_DATA                 Data table placement setting
```

The main function of the boot program starts either the write program or user program according to the state of the switch SW1.

Listing 2-4 Main Function (r_fsl_praxis01_boot_main.c)

```
/*******************************************************************************
* Outline           : boot_main
* Include           : none
* Declaration       : void boot_main(void)
* Function Name     : boot_main
* Description       : none
* Argument          : none
* Return Value      : none
* Calling Functions : start-up routine( boot project )
*******************************************************************************/
void boot_main(void)
{
    if( PR_BOOT_SW != PR_SW_ON )
    {
        UB dubRetCode;

        prBootTargetStart();              /* Target start processing */

        dubRetCode = prBootWriteProgram();  /* Run the write program (flash self programming). */

        prBootTargetEnd( dubRetCode );      /* Target termination processing */
    }

}
```

Write program (flash self programming program)

Transits to user program (LED lighting program).

Since the write program presumes the use of the flash self programming library, at initiation it initializes the flash self programming library and transits to the state in which programming is enabled.

When the initialization of the flash self programming library terminates normally, the program sets up the timer and communications ports and transits to the state in which it waits for a command.

Listing 2-5 Write Program Main Function (r_fsl_praxis01_boot_write.c)

```
UB prBootWriteProgram( void )
{
    Omitted
    dtyWriteBuff.fsl_data_buffer_p_u08 = prDubWriteBuffer;
    dtyWriteBuff.fsl_word_count_u08    = PR_WRITE_SIZE;

    /*  */
    dubSelfResult = prFslStart();

    if( dubSelfResult == FSL_OK )
    {
        /*-- Initialize UART1 ports for communication, 11
        prUartinit();

        /* When using memset on the RL78 assuming that flash area is located below 2000H, */
        /* it must be specified as a far standard function. Consequently, */
        /* memset_f needs to be used instead of memset.          */
        memset_f(prDubWriteBuffer, 0x00, PR_MSG_PACKET_SIZE);

        /* Communication loop */
        while( duhSelfLoop == true )
        {
            /*-- Receive from SelfFlashWriter --*/
            do
            {   /* Receive Uart command message */
                dubMsgResult = prUartRcvMsg( &prDubMsgBuffer[0], &dubCommnad);

                if( dubMsgResult != PR_MSG_RET_NORM_END )
                {   /* Send error to SelfFlashWriter if an error is found in the command. */
                    prUartSendMsg( dubCommnad, dubMsgResult );
                }
            }
            while( dubMsgResult != PR_MSG_RET_NORM_END );

            /*-- Process according to the type of the comm
            switch( dubCommnad )
            {
                    Omitted
                /* Discard any irrelevant command. */
                default:
                    /*  */
                   dubMsgResult = PR_MSG_RET_PRM_ERR;
                   prUartSendMsg( dubCommnad, dubMsgResult );
                   break;

            }
        }
     }
     else {
         dubReturn = false;
     }

     /* Terminate flash self programming. */
     prFslEnd();
     return dubReturn;
}
```

Because this program is intended for writing, perform initialization processing so that flash self programming is enabled at the beginning. If the initialization fails, terminate without performing any communication.

memset_f must be executed instead of memset if the mirror area is not contained in the boot area.

Cause a branch according to the command received from SelfFlashWriter. Command processing to be executed is listed on the next and subsequent pages.

When a command is received from SelfFlashWriter, the main function takes necessary actions as directed by the command. The supported commands are WRITE, DATA, IVERIFY, BOOTSWAP, and RESET. The main function returns an error for the other commands.

The processing described below is performed when a WRITE command is received.

- Transfers the block, address, and size information that is received from the receive buffer to memory.
- Checks the parameters for the received data and, if no problem is found, checks the specified blocks for blank blocks and, if necessary, performs erase processing, then sends the execution result to SelfFlashWriter.

<Write command from SelfFlashWriter>

Sends the block to be programmed, its address and size.

WRITE command format

Start Code	Data Length	Command	Data			Checksum
0x01	0x0008	0x05	Block	Address	Size	1 byte

Listing 2-6 Write Program's WRITE Command Processing (r_fsl_praxis01_boot_write.c)

```
        /*-- WRITE command --*/
        case PR_MSG_COMM_WRITE:
        {
            UB  dub_i;
            UB  dubStartEraseBlock;
            UW  duwStartWriteAddress;
            UB  dubBlockLength;

            /*-- Store received data (Block to program, address, size) from buffer. --*/
            dubStartEraseBlock    = prDubMsgBuffer[ PR_MSG_BLOCK_NUM ];
            duwStartWriteAddress  = ( (UW)( prDubMsgBuffer[ PR_MSG_ADDR_HI  ] ) ) << 16;
            duwStartWriteAddress |= ( (UW)( prDubMsgBuffer[ PR_MSG_ADDR_MID ] ) ) <<  8;
            duwStartWriteAddress |= ( (UW)( prDubMsgBuffer[ PR_MSG_ADDR_LOW ] ) );
            duwWriteSize          = ( (UW)( prDubMsgBuffer[ PR_MSG_SIZE_HI  ] ) ) << 8;
            duwWriteSize         |= ( (UW)( prDubMsgBuffer[ PR_MSG_SIZE_LOW ] ) );
            duwEndWriteAddress    = duwStartWriteAddress + duwWriteSize  - 1;
            dubBlockLength        = (UB)( ( duwWriteSize - 1 ) / PR_BLOCK_SIZE ) + 1;

            /* Parameter check (blocks 0-3 protection, inhibit 0 size writing, etc.) */
            if( ( dubStartEraseBlock >= 4                  ) &&   /*  */
                ( dubStartEraseBlock <  PR_MAX_BLOCK_NUM ) &&   /*  */
                ( duwWriteSize      != 0                 ) )    /*  */
            {
                /* Check state of blocks subject to programming and erase processing */
                for( dub_i = 0 ; dub_i < dubBlockLength ; dub_i++ )
                {
#ifdef PR_USE_OCD_MODE     /* Do nothing on monitor area in OCD mode. */
                    if( ( dubStartEraseBlock + dub_i ) != PR_OCD_MONITOR_BLOCK )
                    {
#endif
                        DI();
                        dubSelfResult = FSL_BlankCheck( dubStartEraseBlock + dub_i );

                        /* If the target block is nonblank. */
                        if( dubSelfResult == FSL_ERR_BLANKCHECK )
                        {

                            dubSelfResult = FSL_Erase( dubStartEraseBlock + dub_i );
                        }

                        EI();
#ifdef PR_USE_OCD_MODE     /* Do nothing on monitor area in OCD mode.  */
                    }
                    else {
                        dubSelfResult = FSL_OK;
                    }
#endif
                }

                /* Set address to write. */
                duwWriteAdressIndex = duwStartWriteAddress;

                /* Convert flash self programming result to a transmit parameter. */
                dubMsgResult = prFslErrorCheck( dubSelfResult );
            }
            else {
                dubMsgResult = PR_MSG_RET_PRM_ERR;
            }

            /* Send result. */
            prUartSendMsg( dubCommnad, dubMsgResult );
            break;
        }
```

Check blocks to program and erase.

Library function calls

Send the status.

The main function receives a DATA command after it received the WRITE command. It receives 256 bytes of data with a single DATA command; it receives a total of 1,024 bytes (1 block) of data with four DATA commands.

The write data received with the DATA command is written into the target block that is specified by the WRITE command.

- Writes 256 bytes of data into the block specified by the WRITE command.
- Increments the start address by 256 bytes for the next write operation.
- Sends the execution result to SelfFlashWriter.

<DATA command from SelfFlashWriter>

Sends write data.

DATA command format

Start Code	Data Length	Command	Data	Checksum
0x01	0x0102	0x06	256 bytes	1 bytes

Listing 2-7 Write Program's DATA Command Processing (r_fsl_praxis01_boot_write.c)

```
    /*-- DATA command --*/
            /*-- DATA command --*/
            case PR_MSG_COMM_DATA:

                /* Check whether the target write address is smaller than the end address */
                if( duwWriteAdressIndex <= duwEndWriteAddress )
                {
#ifdef PR_USE_OCD_MODE  /* Do nothing on monitor area in OCD mode.  */
                    if( duwWriteAdressIndex < PR_OCD_MONITOR_ADDR )
                    {
#endif
                        /* Copy write data into data buffer. */
                        memcpy_f( &prDubWriteBuffer[0], &prDubMsgBuffer[0], PR_MSG_PACKET_SIZE );

                        dtyWriteBuff.fsl_destination_address_u32 = duwWriteAdressIndex;

                        DI();
                        dubSelfResult = FSL_Write( &dtyWriteBuff );
                        EI();

#ifdef PR_USE_OCD_MODE  /* Do nothing on monitor area in OCD mode.  */
                    }
                    else {
                        dubSelfResult = FSL_OK;
                    }
#endif

                    /* Convert flash self programming result to a transmit parameter. */
                    dubMsgResult = prFslErrorCheck( dubSelfResult );

                    /* Increment target write address by write size */
                    duwWriteAdressIndex += PR_MSG_PACKET_SIZE;
                }
                else {
                    dubMsgResult = PR_MSG_RET_ERR_END;
                }

               /* Send result. */
                prUartSendMsg( dubCommnad, dubMsgResult );
                break;
```

memset_f must be executed instead of memset if the mirror area is not contained in the boot area.

Library function call

When the programming of one block is finished, SelfFlashWriter sends an IVERIFY command for the target block. Upon receipt of the IVERIFY command, the main function performs IVERIFY processing on the target block.

- Performs IVERIFY processing on the block that is specified by the WRITE command.
- Sends the execution result to SelfFlashWriter.

<IVERIFY command from SelfFlashWriter>

Sends the number of block to be subjected to IVERIFY processing.

IVERIFY command format

Start Code	Data Length	Command	Data	Checksum
0x01	0x0003	0x0B	Block	1 byte

Listing 2-8 Write Program's IVERIFY Command Processing (r_fsl_praxis01_boot_write.c)

```
/*-- IVERIFY command --*/
case PR_MSG_COMM_IVERIFY:

    /* Verify processing */
    DI();
    dubSelfResult = FSL_IVerify( prDubMsgBuffer[ PR_MSG_IVERIFY_BLOCK ] );
    EI();

    /* Convert flash self programming result to a transmit parameter and send result. */
    dubMsgResult = prFslErrorCheck( dubSelfResult );
    prUartSendMsg( dubCommnad, dubMsgResult );
    break;
```

Library function call

SelfFlashWriter sends the whole write data when programming is done in Chip Mode. When the transmission of the whole write data is carried out and its completion is confirmed, SelfFlashWriter sends out a BOOTSWAP command.

The processing described below is performed when a BOOTSWAP command is received.

- Rewrites the boot flag.
- Sends the execution result to SelfFlashWriter.
- Effects a reset.

<BOOTSWAP command from SelfFlashWriter>

Sends a BOOTSWAP command.

BOOTSWAP command format

Start Code	Data Length	Command	Data	Checksum
0x01	0x0002	0x08	None	1 byte

Listing 2-9 Write Program's BOOTSWAP Command Processing (r_fsl_praxis01_boot_write.c)

```
              /*-- BOOTSWAP command --*/
              case PR_MSG_COMM_BOOTSWAP:

              /* Disabled in OCD mode. */
#ifdef PR_USE_OCD_MODE
                  /* Do nothing and end normally in OCD mode. */
                  prUartSendMsg( dubCommnad, PR_MSG_RET_NORM_END );
#else
                  /* Perform processing if not in OCD mode. */
                  /* Boot flag rewrite processing */
                  DI();
                  dubSelfResult = FSL_InvertBootFlag();
                  EI();

                  /* Convert flash self programming result to a transmit parameter and send result. */
                  dubMsgResult  = prFslErrorCheck( dubSelfResult );
                  prUartSendMsg( dubCommnad, dubMsgResult );

                  /* Upon completion, perform forced reset processing. */
                  if( dubMsgResult == PR_MSG_RET_NORM_END )
                  {
                      /* UART communication termination processing */
                      prUartEnd();

                      /* Forced reset processing */
                      FSL_ForceReset();
                  }
#endif
                  break;
```

Library function calls

In addition, the press of the Reset button on SelfFlashWriter causes a RESET command to be sent. The processing described below is performed when a RESET command is received.

- Sends the execution result to SelfFlashWriter.
- Effects a reset.

<RESET command SelfFlashWriter>

Sends a RESET command.

RESET command format

Start Code	Data Length	Command	Data	Checksum
0x01	0x0002	0x07	None	1 byte

Listing 2-10 Write Program's RESET Command Processing (r_fsl_praxis01_boot_write.c)

```
              /*-- RESET command --*/
              case PR_MSG_COMM_RESET:
              /* Disabled in OCD mode. */
#ifdef PR_USE_OCD_MODE
                 /* Do nothing and end normally in OCD mode. */
                 prUartSendMsg( dubCommnad, PR_MSG_RET_NORM_END );
#else
                 /* Perform processing if not in OCD mode. */
                 /* Send result of reception.  */
                 dubMsgResult = PR_MSG_RET_NORM_END;
                 prUartSendMsg( dubCommnad, dubMsgResult );

                 /* UART communication termination processing */
                 prUartEnd();

                 /* Formced reset processing */
                 FSL_ForceReset();            Library function call
#endif
                 break;
```

2.8 Precautions to be Taken when Debugging

The precautions described in paragraphs (1), (2), (3), and (4) below should be taken when evaluating the sample program.

(1) Boot swapping during on-chip debugging

Since the addresses of the programs that are placed in boot clusters 0 and 1 are changed after a swap, their execution cannot be monitored under the debugger unless the programs that are held in boot clusters 0 and 1 are the same.

If the programs can no longer run normally, temporarily terminate the debugger and turn off the power to the target device, then reconnect the target device.

(2) Checking of the reset state and operation of the write program during on-chip debugging

When a reset is to be effected not by the debugger but by the program during on-chip debugging, no software reset can be accomplished using the FSL_ForceReset() function or by executing an invalid instruction.

When using on-chip debugging, it is necessary to place the monitor program in part of the code flash memory area. If that area is programmed, the on-chip debugger will not be able to run normally.

Since the sample program is implemented on the assumption that it is to run on ROM, if an on-chip debugger is used to check rewriting of the entire area of the sample program, the sample program may not run normally depending on the debugger that is used, for the above-mentioned reason.

To check the sample program using an on-chip debugger, enable "#define PR_USE_OCD_MODE" in the header file (r_fsl_praxis01_com.h).

If this setting is used, however, the reset processing specified within the program and the programming of the monitor program area which is reserved for on-chip debugging are not performed.

Listing 2-11 Write Program's RESET Command Processing (r_fsl_praxis01_com.h)

```
/*********************************************************/
/*                                               */
/*********************************************************/
/* QB-R5F100LE-TB */
#if 1
#define __QB_R5F100LE_TB__              /*  */
/*#define PR_USE_OCD_MODE*/             /*  */

/*  */
#else
#define __NON_TARGET__
#endif

/*  */
#ifdef PR_USE_OCD_MODE
   #define PR_OCD_MONITOR_BLOCK    0x3F
   #define PR_OCD_MONITOR_ADDR     0xFC00
#endif
```

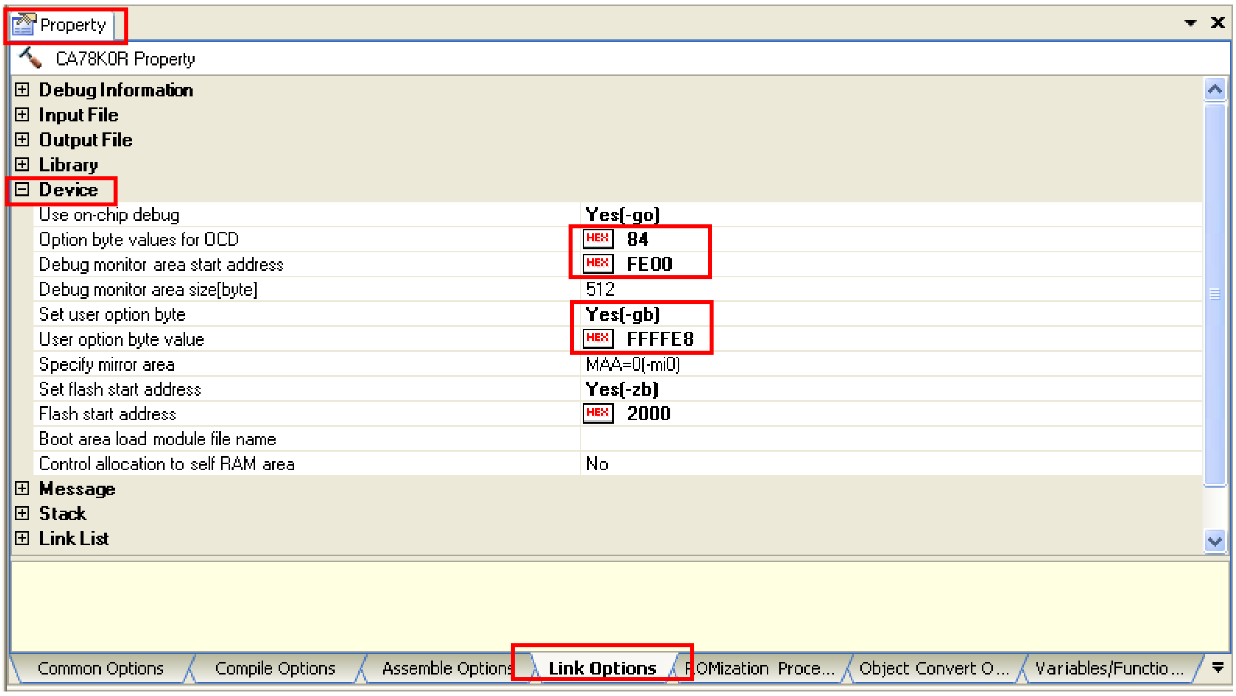
Uncomment this line.

(3) Option byte and on-chip debug settings

The communication processing of the sample program runs normally when the high-speed on-chip oscillator (high-speed OCO) is set to 32 MHz. In the attached project files, the option byte for the boot area project is set to "FFFFE8," the high-speed on-chip oscillator is set to 32 MHz, and the WDT is set up. For newly created projects, however, these items are not set by default. If the sample program needs to be loaded into a user-supplied project, set the option byte for the boot area to "FFFFE8."

Before on-chip debugging, set the on-chip debugging setting for the boot area to "Yes."

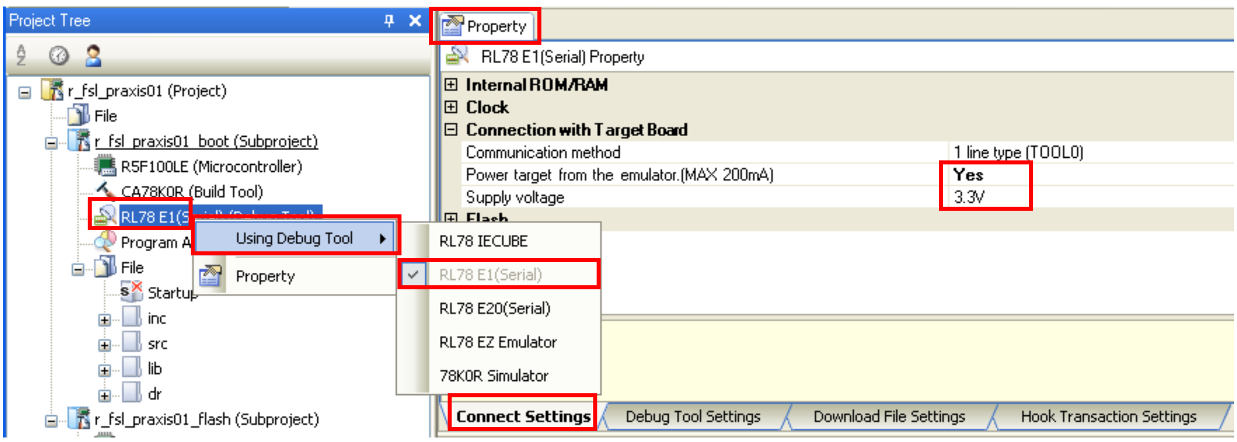
Figure 2-23 Option Byte Setting



(4) E1 emulator and power supply settings

To use the E1 emulator during debugging, change the value of the debugging tool setting in the project file from the default simulator to the E1 emulator. When power is to be supplied from the E1 emulator, set "Power target from the emulator (MAX 200mA)" to "Yes."

Figure 2-24 E1 Emulator Settings



2.9 How to Evaluate Rewriting of Programs

By using programs of different specifications, you can see if flash self programming has successfully completed rewriting of the original program. Follow the procedures described in paragraphs (1) and (2) below when evaluating a program.

(1) Writing the hex file for the whole area program into the QB-R5F100LE-TB

Write the file r_fsl_praxis01_flash.hex in the <project folder for the flash area programs>\DefaultBuild folder into the RL78/G13 using a flash memory programmer. (This file is a hex file that contains the programs in both the boot and flash areas.)

Reset the QB-R5F100LE-TB while holding down the SW1 on the QB-R5F100LE-TB, and the user program will start and turn on LED1 and flash LED2. Press SW1 in this state, and ASCII data will be sent to the host machine. Check the data that is displayed on the host machine with terminal software or similar tool. Subsequently, a reset is automatically effected by the WDT and the QB-R5F100LE-TB waits for communication with SelfFlashWriter, LED2 stays on, and LED1 turns off.

(2) Writing the program with SelfFlashWriter

Perform programming with the program that has been written in step (1).

Specify the write program "r_fsl_praxis01_write_test.hex" from SelfFlashWriter as shown in figure 2-25. When programming the flash area, check "Block" under "Operation Mode" and set the "Start" block number to 008 and the "Stop" block number to 063 (specify the entire flash area).

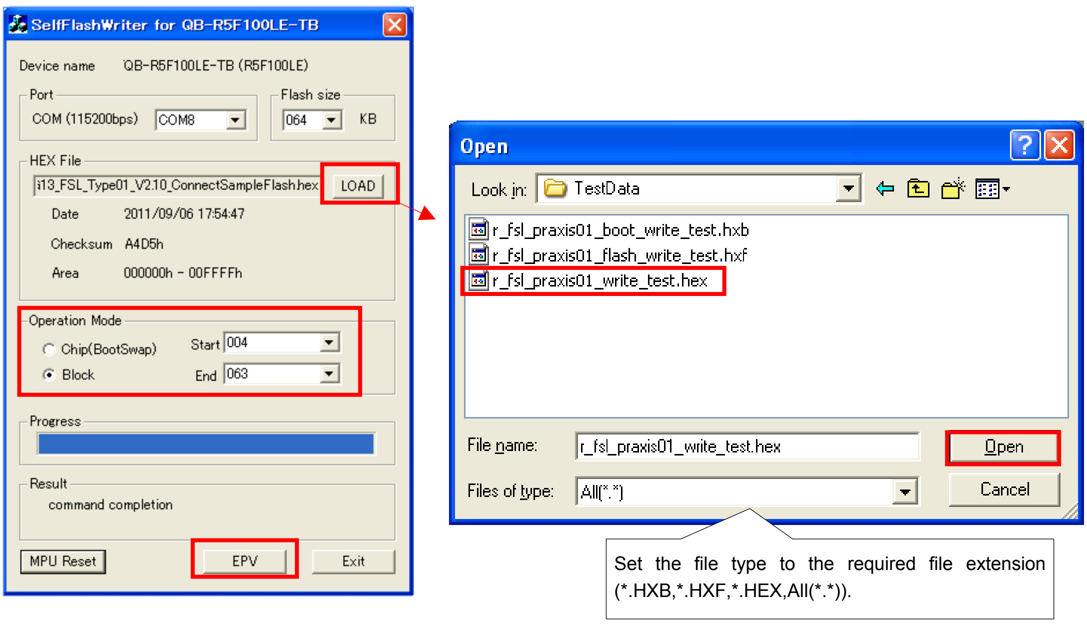
Click the EPV button, and communication will start and the only the program in the flash area be rewritten.

After programming ends, reset the QB-R5F100LE-TB while holding down the SW1 on the QB-R5F100LE-TB to confirm that only the mode of displaying LED1 and LED2 by the user program has been changed. Note that the mode of displaying LED1 and LED2 by the write program is not changed.

When programming the entire area, check "Chip" under "Operation Mode" and click the EPV button to start programming. After programming ends, reset is automatically effected. After the programming, make sure that the LED display modes of the user program and write program are reversed.

Note: Click on the, "EXIT" button after running "EPV", please terminate once the SelfFlashWrite.

Figure 2-25 Programming from SelfFlashWriter



2.10 How to Evaluate Rewriting of Data

Use the flash self programming function to update the data table and verify that the data being used in the program for the flash area (user program) is altered. Follow the procedures described in paragraphs (1) and (2) below when evaluating rewriting of data.

(1) Writing the hex file for the whole area program into the QB-R5F100LE-TB

Write the file r_fsl_praxis01_flash.hex in the <project folder for the flash area programs>\DefaultBuild folder into the RL78/G13 using a flash memory programmer. (This file is a hex file that contains the programs in both the boot and flash areas.)

Reset the QB-R5F100LE-TB while holding down the SW1 on the QB-R5F100LE-TB, and the user program will start and turn on LED1 and flash LED2. Press SW1 in this state, and ASCII data will be sent to the host machine. Check the data that is displayed on the host machine with terminal software or similar tool. Subsequently, a reset is automatically effected by the WDT and the QB-R5F100LE-TB waits for communication with SelfFlashWriter, LED2 stays on, and LED1 turns off.

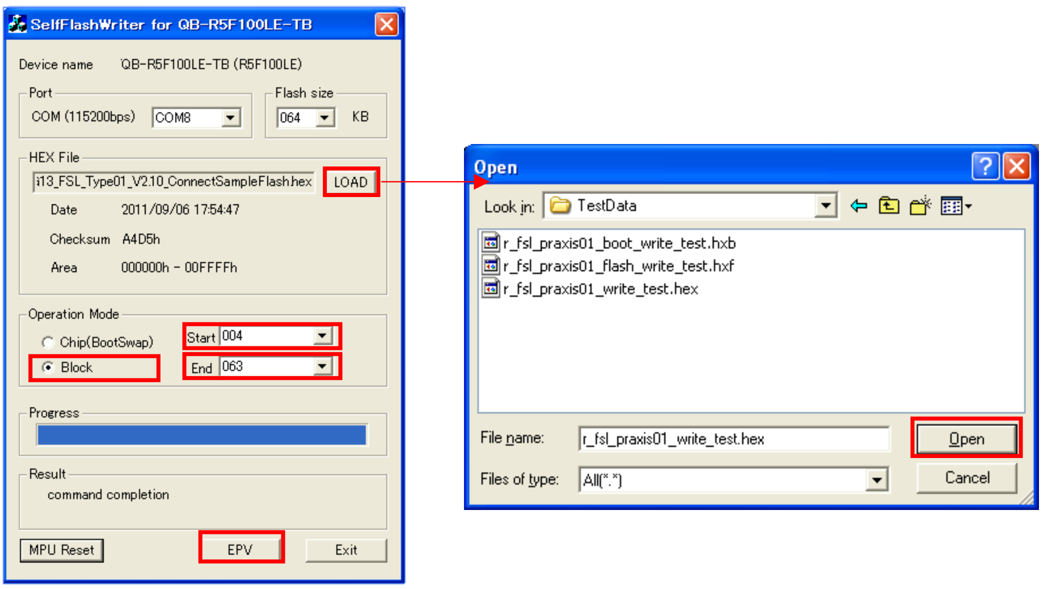
(2) Updating the data table in the flash area with SelfFlashWriter

Perform rewriting with the program that has been written in step (1). Specify the flash area program "r_fsl_praxis01_write_test.hex" from SelfFlashWriter as shown in figure 2-26. Change the "Operation Mode" setting to the block programming mode, set both the "Start" block and "End" block to 62, and click the EPV button, and communication will start. The data table for the programs is rewritten by the flash self programming program.

After programming ends, reset the QB-R5F100LE-TB while holding down the SW1 on the QB-R5F100LE-TB to confirm that the user program is started and that only the ASCII data to be sent to the host machine by pressing the SW1 is changed.

Note: Click on the, "EXIT" button after running "EPV", please terminate once the SelfFlashWrite.

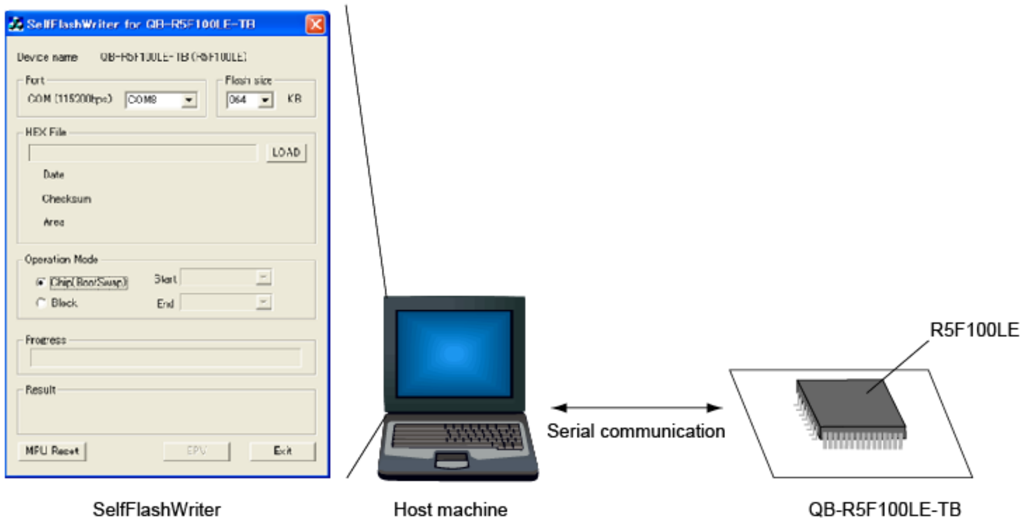
Figure 2-26 Programming by SelfFlashWriter



Appendix A SelfFlashWriter

SelfFlashWriter is a GUI for splitting machine-language files (*.hex) specified by the host machine into minimum flash memory erasure units or blocks (1 block = 1024 bytes) and sending them through serial communication. It can serve as a virtual tool for evaluating flash self programming.

Figure A-1 Outline of the Connection between SelfFlashWriter and QB-R5F100LE-TB



(1) Operating environment

SelfFlashWriter must be used in the environment described in the table below.

Table A-1 SelfFlashWriter's Operating Environment

CPU	Pentium® III 500 MHz or faster
Supported OS	Windows® 2000/ Windows XP®/ Windows Vista®/ Windows® 7
Memory size	512 Mbytes or more
HDD capacity	Approx. 7 Mbytes

(2) Communication specifications

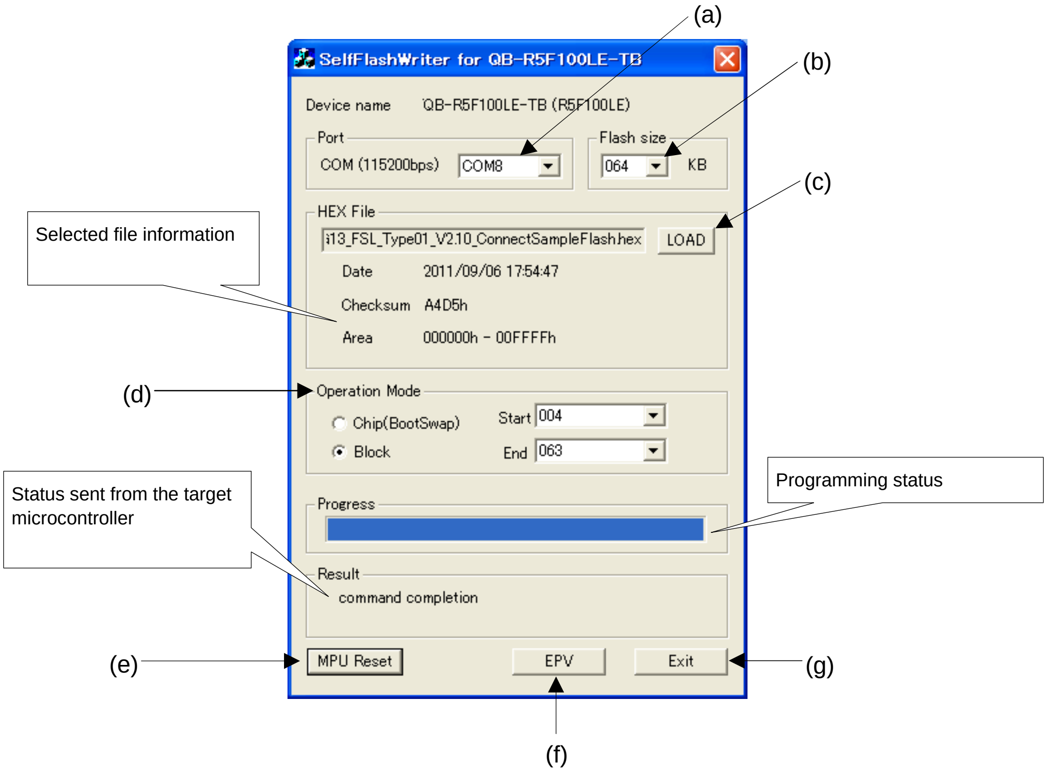
SelfFlashWriter communicates with the RL78/G13 (R5F100LE) according to the following serial communication specifications:

Table A-2 SelfFlashWriter's Communication Specifications

Bits/second	115200
Data bit length	8
Parity	None
Stop bits	1
Flow control	None

(3) Functions

Figure A-2 SelfFlashWriter Functions



(a) Selecting the COM port

Select a communication port No. 1 to 16 from the pull-down menu.

(b) Setting the flash memory size

Specify the size of flash memory for the RL78/G13 (64 Kbytes or less).

(c) LOAD button

Specify the machine-language file (*.hex) to be written to the RL78/G13 (R5F100LE).

(d) Checking the Operation Mode

Chip: Select this option when programming the entire chip. Pressing the EPV button with this mode checked causes SelfFlashWriter to send the write data. When programming is finished, SelfFlashWriter sends a BOOTSWAP command.

NOTE: Check "Block" and set "Start" block number to 004 and "Stop" block number to 063 (the last block number).

Block: Select this option when programming in block units. Select the range of blocks to be subjected to programming through the START and END pull-down menus. SelfFlashWriter programs the range of addresses specified in START and END regardless of the start and end addresses specified in the machine-language file (*.hex). Any extra range of addresses that does not exist in the machine-language file is erased (i.e. programmed with FFh). Blocks 0 to 3 are not subjected to programming.

(e) MPU Reset button

Press this button to send a RESET command to the RL78/G13 (R5F100LE) so that the RL78/G13 (R5F100LE) will be reset.

(f) EPV (Erase-Program-Verify) button

Press this button to send a command to the RL78/G13 (R5F100LE). The RL78/G13 (R5F100LE) erases the flash memory, writes data, and performs internal verification.

Note: Click on the, "EXIT" button after running "EPV", please terminate once the SelfFlashWrite.

(g) Exit button

Press this button to quit SelfFlashWriter.

(4) Communication commands

The function and format of the communication commands are given below.

(a) Commands that SelfFlashWriter sends

＜WRITE command>

Sends information on the block, address, and size of data to be written.

Start Code	Data Length	Command	Data			Checksum
0x01	0x0008	0x05	Block	Address	Size	1 byte

＜DATA command>

Sends 256 bytes of write data.

Start Code	Data Length	Command	Data	Checksum
0x01	0x0102	0x06	256 bytes	1 byte

＜IVERIFY command>

Sends an IVERIFY command.

Start Code	Data Length	Command	Data	Checksum
0x01	0x0003	0x0B	Block	1 byte

＜RESET command>

Sends a RESET command.

Start Code	Data Length	Command	Data	Checksum
0x01	0x0002	0x07	None	1 byte

<BOOTSWAP command>

Sends a BOOTSWAP command. This command is used when "Chip" is selected as the Operation Mode.

Start Code	Data Length	Command	Data	Checksum
0x01	0x0002	0x08	None	1 byte

(b) Command that the RL78/G13 (R5F100LE) sends

The command given below is returned by RL78/G13 (R5F100LE) in response to the commands sent from SelfFlashWriter. Code you program so that it returns the command in the format given below.

<DATA_REV command>

This is an ACK (response) command to be sent in response to a programming-related command (WRITE or DATA command) from SelfFlashWriter.

Start Code	Data Length	Command	Data	Checksum
0x01	0x0003	Response command	Status[Note]	1 byte

Note: Status indicates the result of executing the command sent from SelfFlashWriter. See table A-3, List of Status, for details.

Table A-3 List of Status

Status Name	Value	Description
Normal end	0x00	Normal end
Abnormal end	0x01	Abnormal end
Parameter error	0x05	Parameter error in communication format
Protect error	0x10	The specified block falls within the boot area and the boot area is protected against programming.
Erasure error	0x1A	Erasure error
Programming error	0x1C	Could not program data properly.
Verify error	0x1D	Verify error found after programming
Checksum error	0xFF	Checksum error in communication format

(5) Exchange of communication commands

Exchange of communication commands between SelfFlashWriter and RL78/G13 (R5F100LE) proceeds as described in (a) and (b).

Code your program so that the RL78/G13 (R5F100LE) sends its commands as exactly specified in (a) and (b).

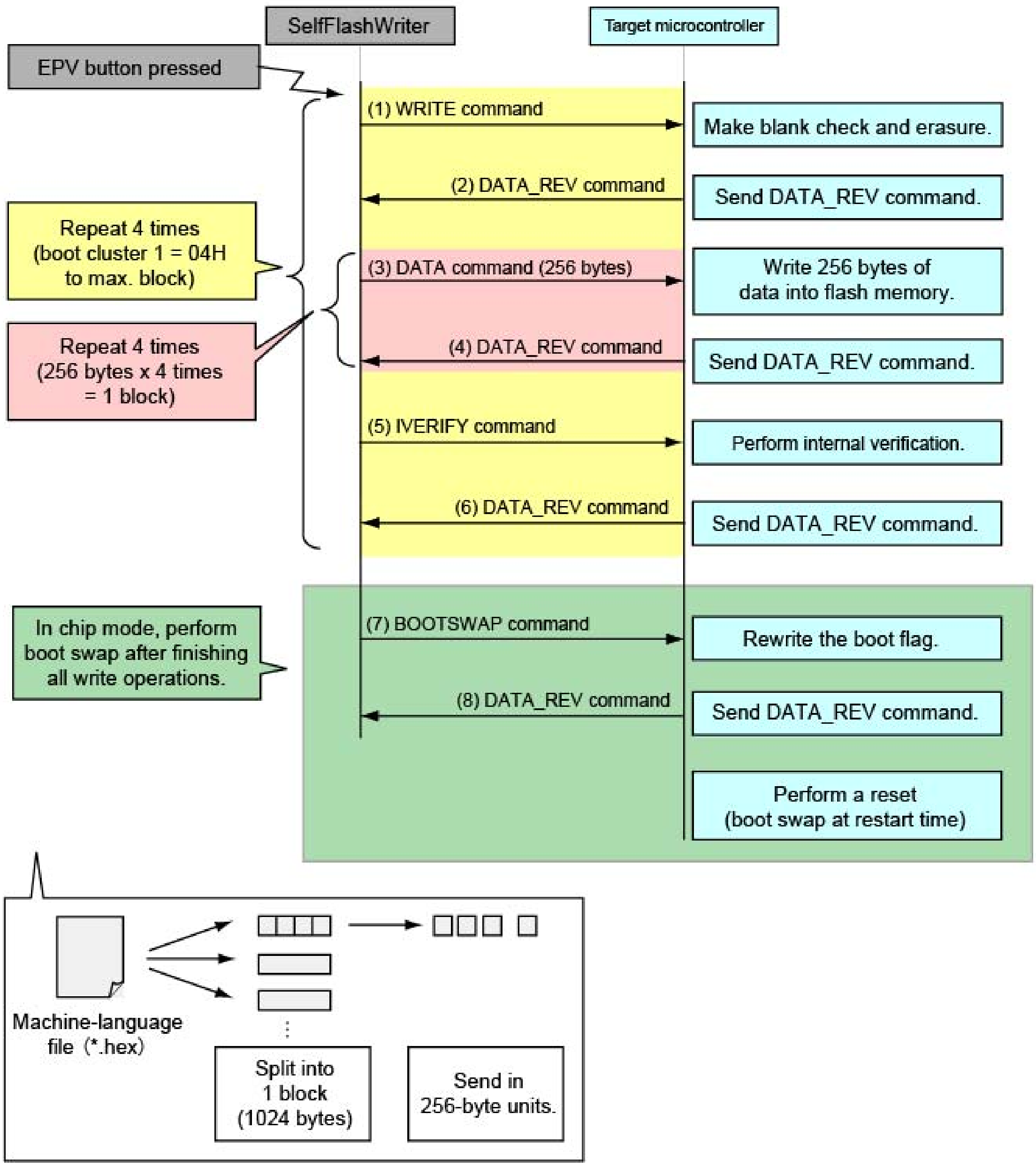
(a) EPV execution flow to be followed when "Block" is selected as the Operation Mode

Write the machine-language file specified through SelfFlashWriter. The machine-language file is split into blocks (1024 bytes) and one block of data is written with a single WRITE command. On transmission, one block is further split into four 256-byte sub-blocks and sent in four transmission operations. After SelfFlashWriter sends each command, it waits for a DATA_REV response from the RL78/G13 (R5F100LE). It signals a timeout error if no response is received.

(b) EPV execution flow to be followed when "Chip" is selected as the Operation Mode

In this mode, the RL78/G13 (R5F100LE) programs the entire program area using the boot swap function. After the programming of the entire area is finished, SelfFlashWriter sends a BOOTSWAP command. After SelfFlashWriter sends each command, it waits for a DATA_REV response from the RL78/G13 (R5F100LE). It signals a timeout error if no response is received.

Figure A-3 EPV Execution Flow



Note

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep. 30, 2011	-	First edition issued
1.01	Dec. 28, 2012	All page	Change format of document.
1.02	Feb. 15,2013	All page	the flash self-programming library Type01 Ver.2.10 -> V2.20 Development environment CubeSuite+ V1.00 -> V1.03

General Precautions in the Handling of MPU/MCU Products

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

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

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

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

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

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

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

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

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

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

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

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

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

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

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

Colophon 2.2