RX Family

Flash Memory Data Management Module Using Firmware Integration Technology

Summary

This application note describes methods of data management using the on-chip flash memory of RX MCUs from Renesas and how to use them. The flash memory data management module (DATFRX) is an upper-layer software module intended to be used for managing data in on-chip flash memory. A flash FIT module, separate lower-layer software for controlling the flash memory of your specific MCU, is available for download on the Renesas website.

Flash FIT module (on-chip flash programmer), **revision 3.40 or later**

RX Family Flash Module Using Firmware Integration Technology (R01AN2184)

Target Devices

- RX Family

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833)
- RX Family Board Support Package Module Using Firmware Integration Technology (R01AN1685)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723)
- Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)
- Renesas e^2 studio Smart Configurator User Guide (R20AN0451)
- RX Family Flash Module Using Firmware Integration Technology (R01AN2184)

Contents

1. Overview

1.1 DATFRX

Upper-layer software used to manage data in the on-chip flash memory of RX MCUs manufactured by Renesas Electronics.

1.2 Overview of DATFRX

1.2.1 Definitions of Terms

1.2.1.1 Flash Type

Lower-layer flash FIT modules are classified, according to the technology and sequencer used, into four Flash Types: Flash Type 1, Flash Type 2, Flash Type 3, and Flash Type 4.

For details on Flash Type, please obtain and check the latest version of flash FIT module from Renesas Electronics website.

RX Family Flash Module Using Firmware Integration Technology (R01AN2184)

1.2.1.2 Data Flash Memory

This is flash memory for storing data.

The term used to refer to data flash memory differs according to the Flash Type. The terms for data flash memory corresponding to each Flash Type are listed below. In this document the term data flash memory is used.

Table 1.1 Data Flash Memory

Flash Type	Name
Flash Type 1	E2 DataFlash
Flash Type 2	E2 DataFlash
Flash Type 3	Data flash memory
Flash Type 4	Data flash memory

1.2.1.3 Block

The data flash memory is configured as multiple blocks, each of which contains several designated areas.

The block size and count differ depending on the MCU. For details on blocks, refer to the Flash Memory section in the User's Manual: Hardware of the MCU.

1.2.1.4 BGO

BGO stands for "background operation."

This functionality allows a user program allocated in RAM or external memory to run while the data in the data flash memory area subject to data management is being updated.

1.2.2 Overview of Functions

An overview Flash Types and their functions is presented below.

Table 1.2 Overview of Functions

Function			Flash Type 1	Flash Types 2, 3, and 4
Data management	API calls can be used to update and read data associated with user-specified data numbers.		○	○
	User data is managed as DATFRX logical data. The data count and data size can be specified by the user.		○	○
	Data is updated in empty blocks during data update processing. The blocks where data updating takes place are selected by DATFRX. DATFRX sets the block update sequence in a manner that ensures that data updates are not concentrated in specific blocks. Old data is not erased during data update processing.		—	○
	Block erase processing can be used to erase unneeded old data in the blocks being managed. The blocks that are erased are selected by DATFRX.		○	○
	Power cutoff/reset during data update processing	Power cutoff/reset is detected when the initialization function runs after the restart.	○	○
		If the data is not valid, the data from before the update is restored.	○	○
	Power cutoff/reset during block erase processing	Power cutoff or reset during erasure of the blocks being managed is detected when the initialization function runs after the restart.	○	○
		Based on the state of the updated data, a judgment is made as to whether or not the data is valid. If the data is not valid, the data from before the update is restored. Note: If a block is mistakenly judged to be valid, incorrect data could be treated as valid data.	○	○
Data flash memory	BGO support for data in the data flash memory area.		○	○
	Support for operation using either big-endian or little-endian byte order.		○	○

1.2.3 Overview of DATFRX Layers

The relationship between DATFRX and the flash FIT module is illustrated below.

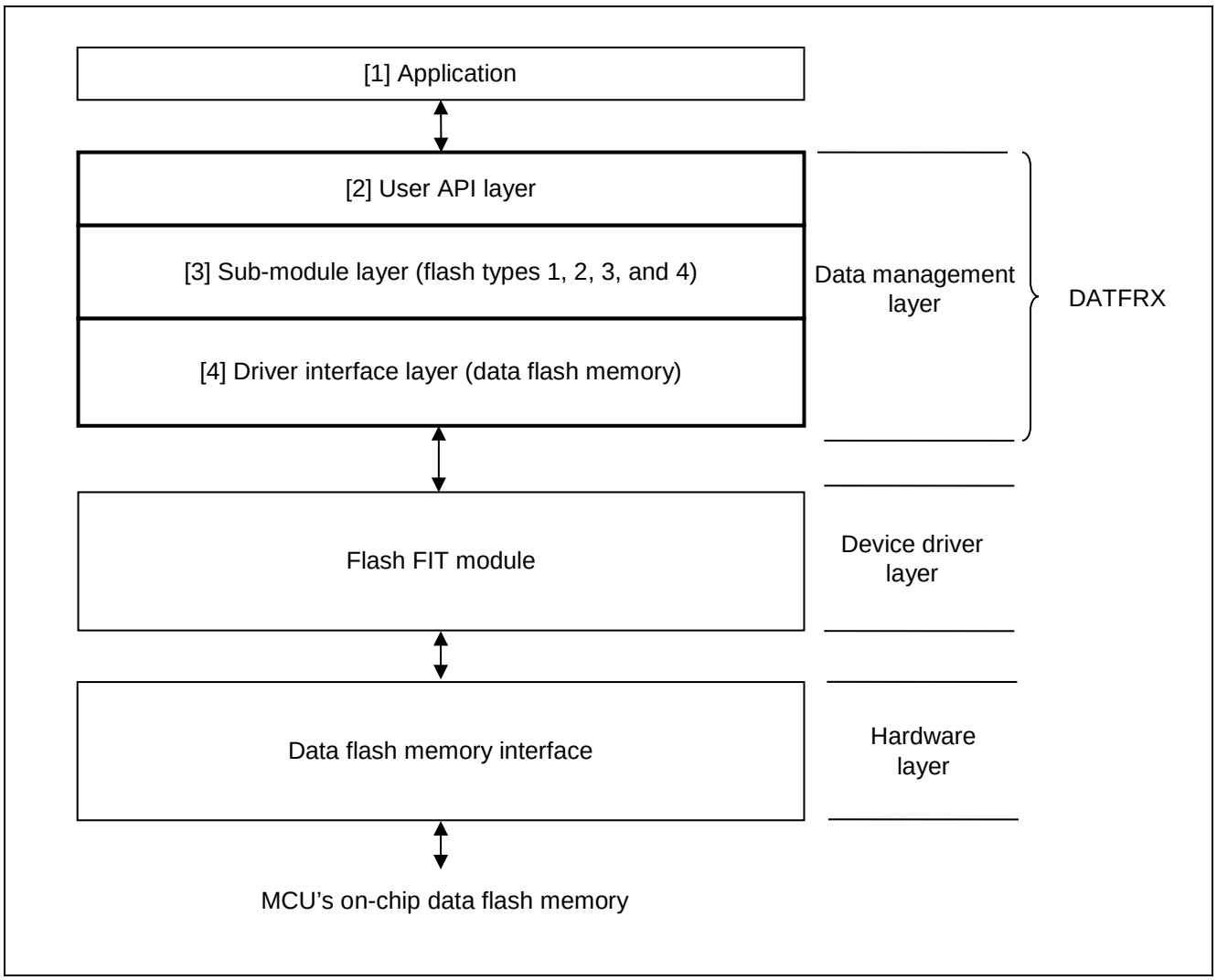

Figure 1.1 Relationship between DATFRX and Flash FIT Module

[1] Application
The FIT module is distributed along with an example program illustrating control of the data flash memory. It can be found in the FITDemos subdirectory.

[2] User API layer
An API for managing data in the data flash memory, it provides functionality not included in the lower-layer device driver.

[3] Sub-module layer
A sub-module for managing data in the data flash memory, it provides functionality not included in the lower-layer device driver.

[4] Driver interface layer
This layer connects to the lower-layer device driver.

1.3 Overview of API

Table 1.3, **API Functions**, lists the API functions contained in DATFRX.

DATFRX supports BGO for the data flash memory. Make sure to perform the necessary settings for BGO and to specify the data flash memory size.

The available functions differ depending on the Flash Type.

Table 1.3 API Functions

Function Name	Description	Flash Type 1	Flash Types 2, 3, and 4
R_FLASH_DM_Open()	DATFRX open processing	○	○
R_FLASH_DM_Close()	DATFRX close processing	○	○
R_FLASH_DM_Init()	Initialization processing (divided)	○	○
R_FLASH_DM_InitAdvance()	Continuation of initialization processing (divided)	—	○
R_FLASH_DM_Format()	Format processing	○	○
R_FLASH_DM_Read()	Data read processing	○	○
R_FLASH_DM_Write()	Data update processing	○	○
R_FLASH_DM_Erase()	Block erase processing	○	○
R_FLASH_DM_Reclaim()	Reclaim processing	○	—
R_FLASH_DM_Control()	State check processing	○	○
R_FLASH_DM_GetVersion()	Version acquisition	○	○

1.3.1 BGO Operating Settings

The flash FIT module supports control utilizing BGO functionality. Make the following settings in r_flash_rx_config.h:

Table 1.4 Flash FIT Module Settings

Flash FIT Module r_flash_rx_config.h #define Definition	With BGO Data Flash Memory	Without BGO Data Flash Memory
FLASH_CFG_CODE_FLASH_ENABLE	0	Not supported.
FLASH_CFG_CODE_FLASH_BGO	0	
FLASH_CFG_DATA_FLASH_BGO	1	

1.4 Processing Example

1.4.1 Flash Type 1

1.4.1.1 Perspective (Processing Example of Main Function)

This is an example from R_FLASH_DM_Open () to R_FLASH_DM_Close () of DATFRX.

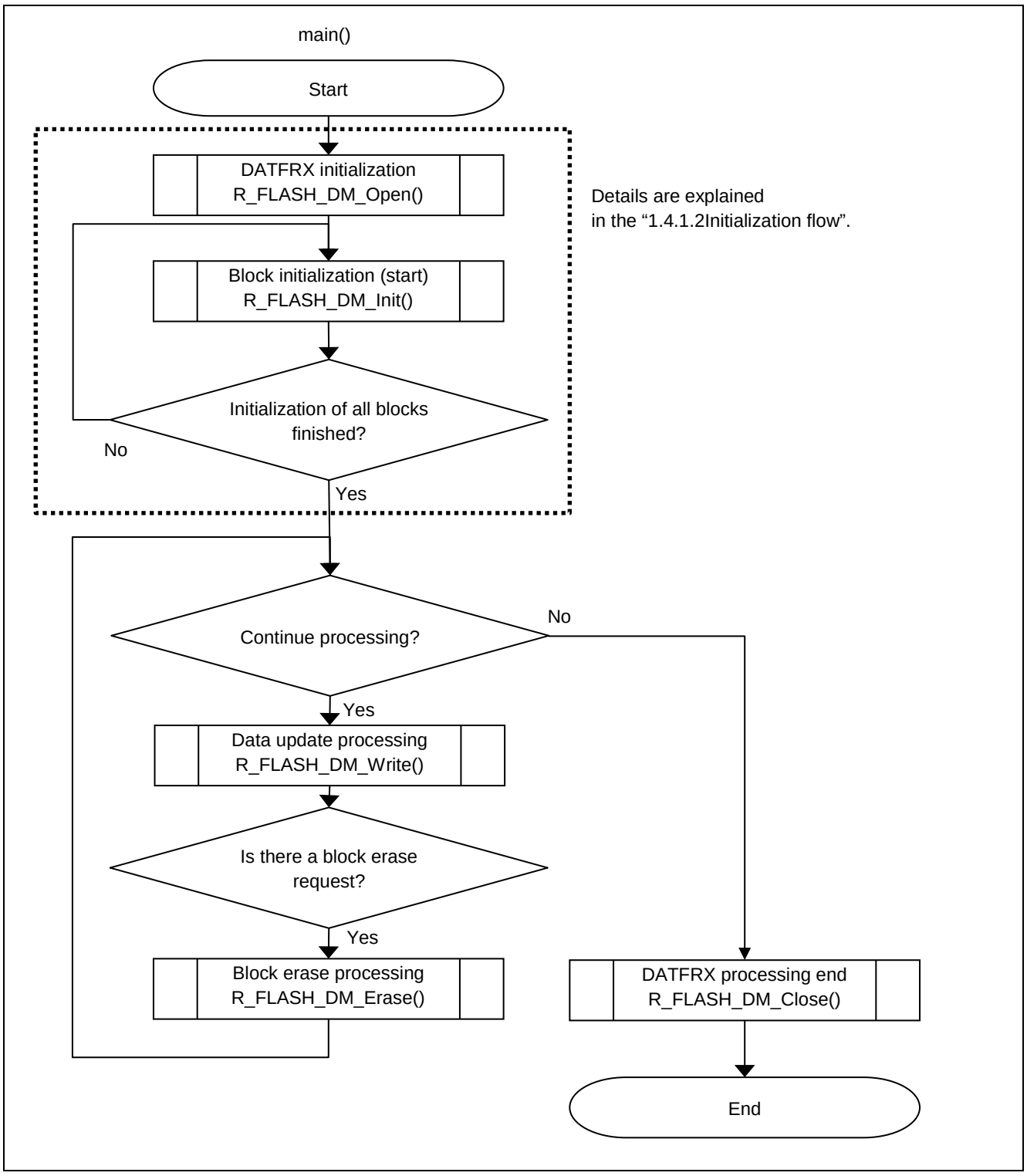

Figure 1.2 Processing Example of Main Function (Flash Type 1)

1.4.1.2 Initialization flow

This is the frow from R_FLASH_DM_Init () to user data processing.

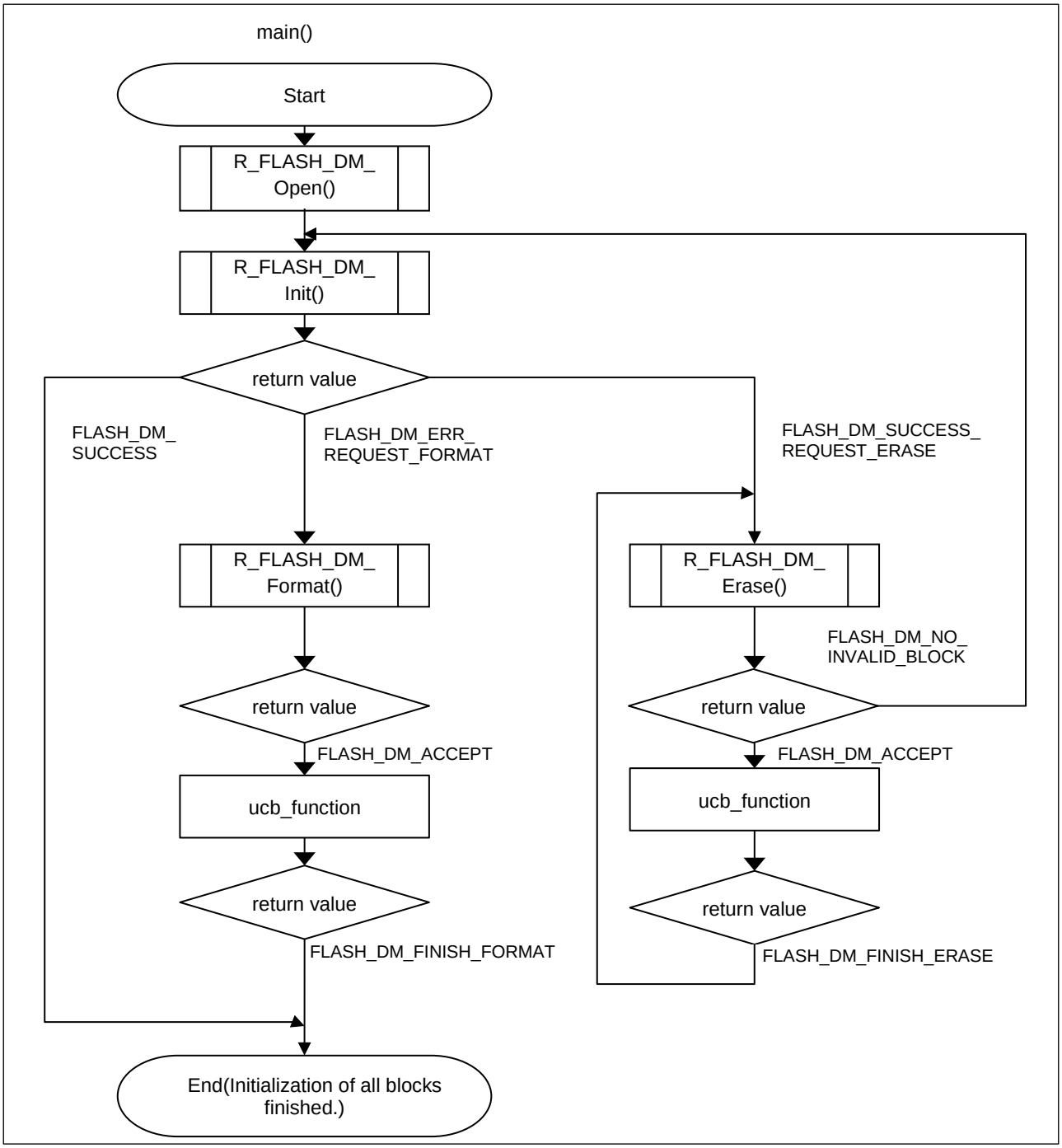

Figure 1.3 Example of processing after R_FLASH_DM_Init() (Flash Type 1)

1.4.2 Flash Type 2, 3 and 4

1.4.2.1 Perspective (Processing Example of Main Function)

This is an example from R_FLASH_DM_Open () to R_FLASH_DM_Close () of DATFRX.

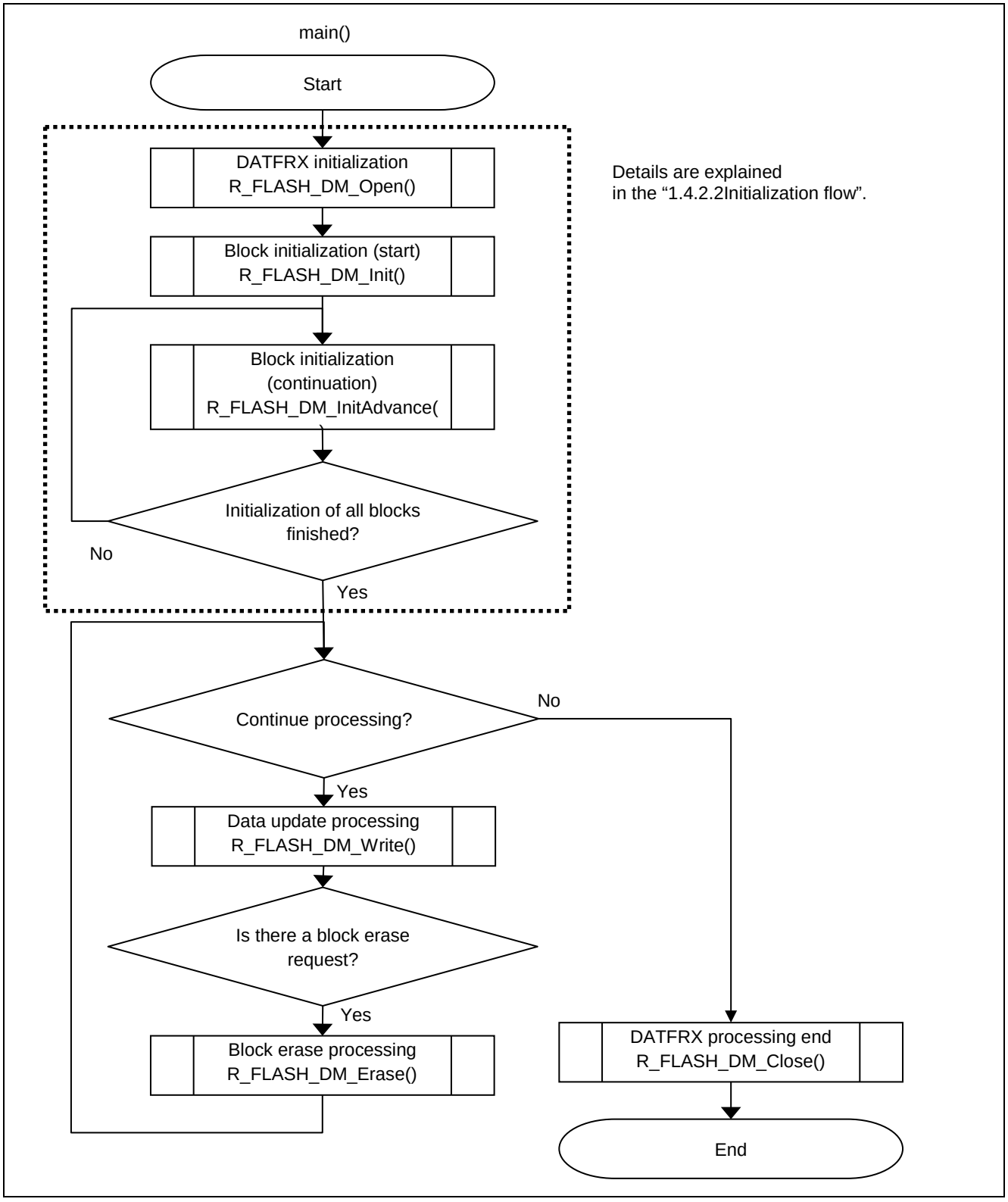

Figure 1.4 Processing Example of Main Function (Flash Type 2, 3 and 4)

1.4.2.2 Initialization flow

This is the frow from R_FLASH_DM_Init () to user data processing.

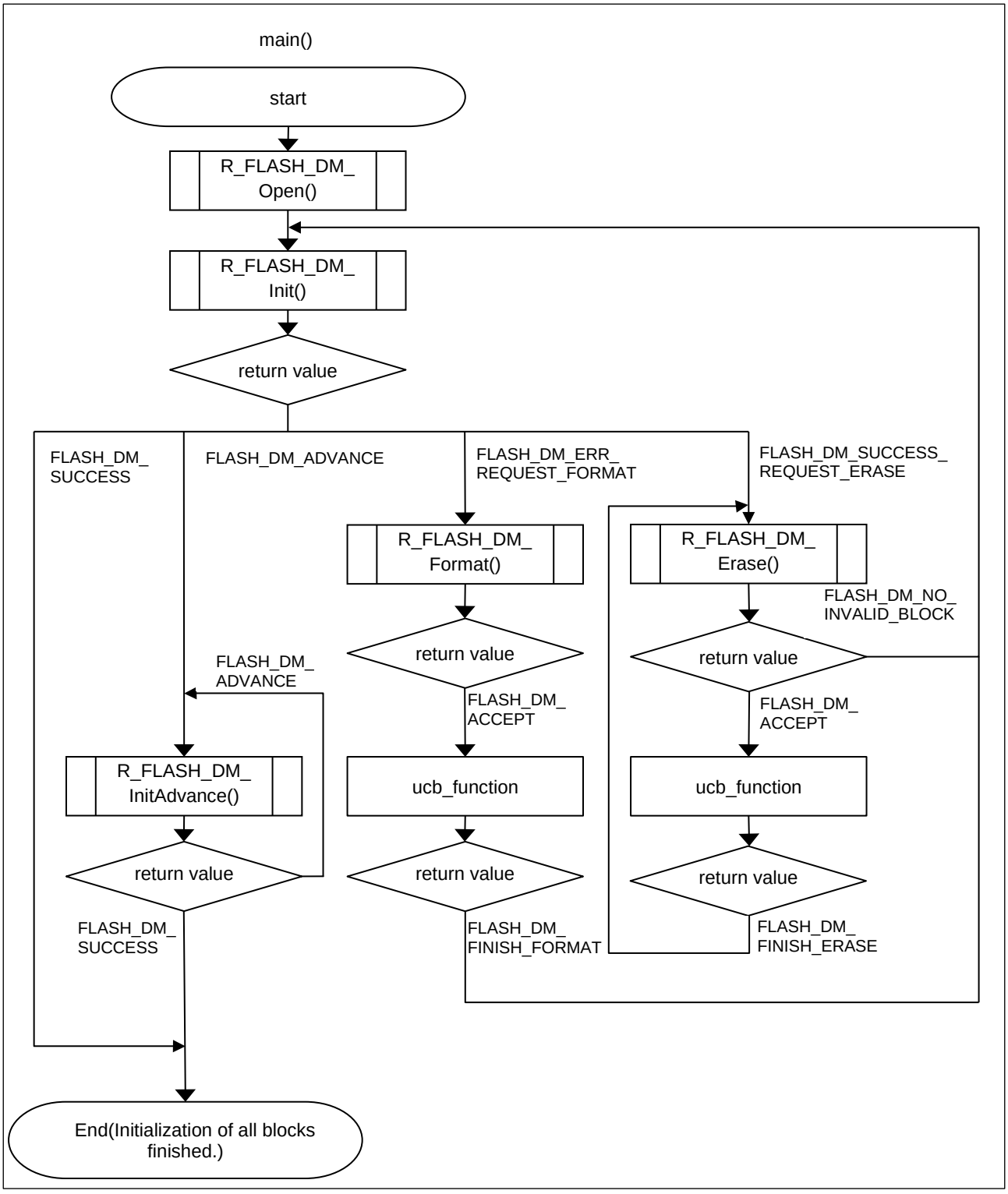

Figure 1.5 Example of processing after R_FLASH_DM_Init() (Flash Type 2, 3 and 4)

1.4.3 Callback Function

The contents of the callback function are created by the user.

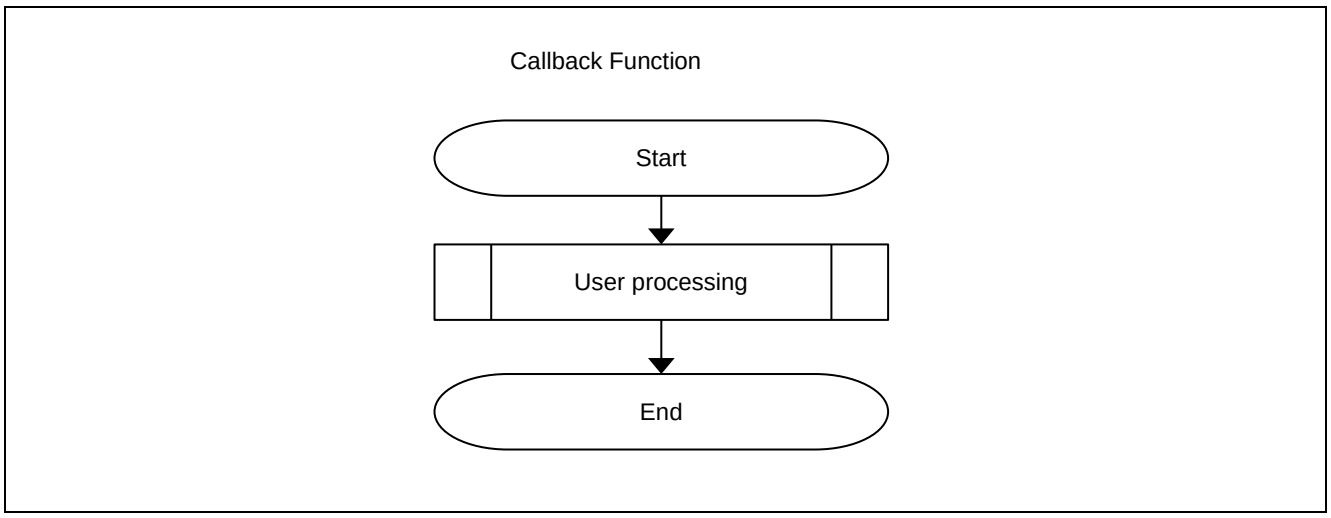

Figure 1.6 Callback Function

1.5 State Transition Diagram

Illustrates the state transitions of the FIT module.

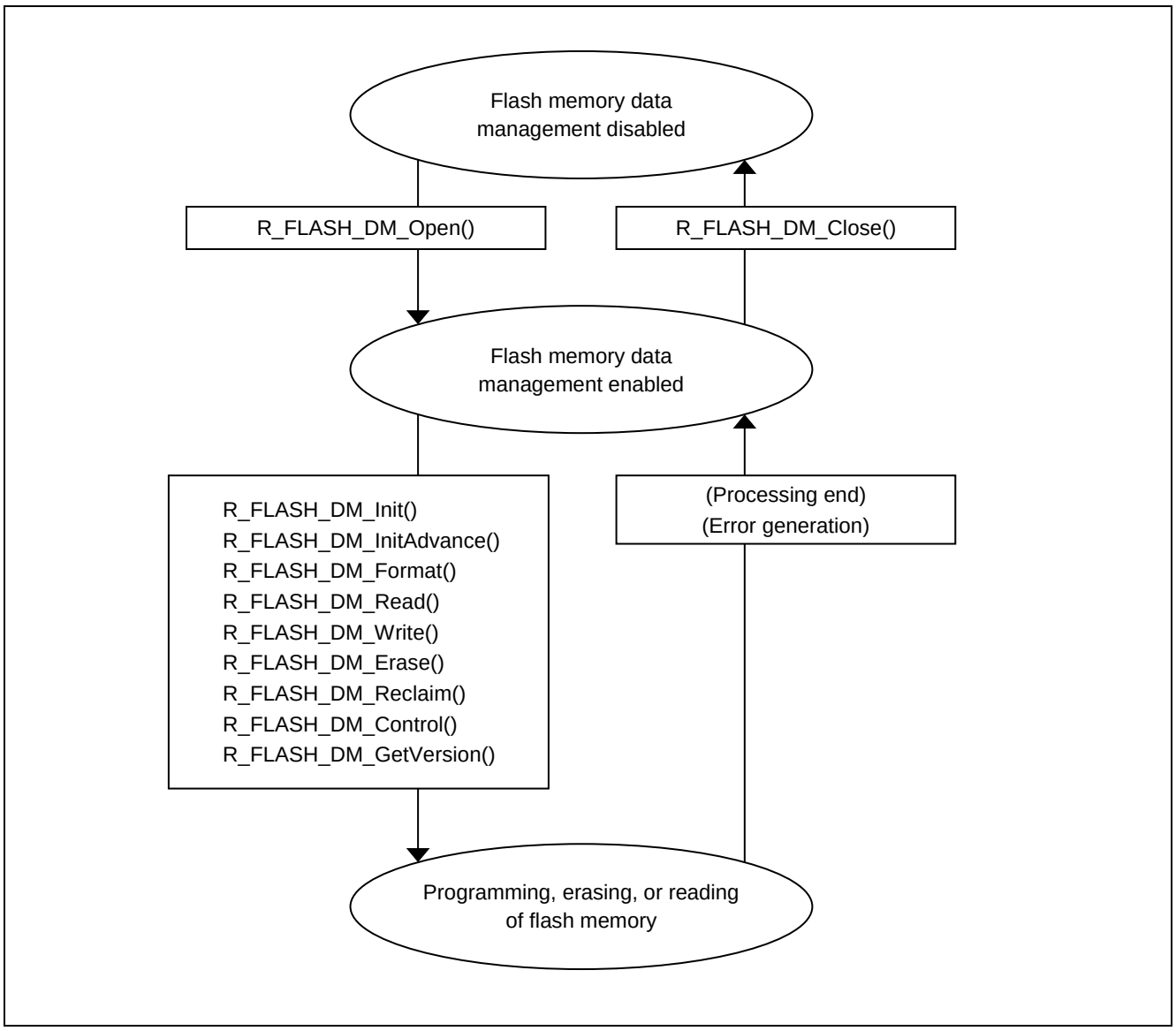

Figure 1.7 State Transition Diagram

1.6 Limitations

Limitations are listed below.

Table 1.5 Limitations

Item	Description
Power supply voltage	DATFRX uses the flash FIT module to perform programming and block erasing of the data flash memory. Ensure that the power supply voltage conditions specified in the MCU's User's Manual: Hardware are met before using API functions that execute program commands or block erase commands.
Data flash memory specifications	For the data flash memory specifications, including details of the flash memory control registers and electrical characteristics, refer to the MCU's User's Manual: Hardware.
Exclusion from other user programs	Accessing the flash memory registers is prohibited until processing by DATFRX finishes. Proper operation is not possible if these registers are accessed before processing completes. DATFRX uses the flash FIT module to access the flash memory registers as necessary. No consideration has been given to parallel operation of user programs that access the flash memory in the same manner during DATFRX processing.
Resets during programming, erasing, or block checking	The user system must meet the following requirements until programming or block erasing has completely finished: Even if the power supply voltage drops, the MCU's operating voltage must be maintained until processing of program commands or block erase commands finishes through the use of super capacitors, etc. For the operating voltage and hold time, refer to the electrical characteristics in the MCU's User's Manual: Hardware. If a drop in the power supply voltage is detected during execution of a program command or block erase command, refrain from calling an API function to ensure that the next command is not issued. Also, the MCU and flash memory may enter an unstable state if the power supply voltage drops below the rated operating voltage. In this case, reset the MCU and run DATFRX initialization processing.
Formatting and initialization by the driver	Make sure that power is not interrupted and no reset occurs during formatting. If a power cutoff or reset occurs, the block could be misidentified during subsequent initialization processing because its format is in an unfinished state.
Initialization processing	Make sure that power is not interrupted and no reset occurs during initialization processing. If a power cutoff or reset occurs, data loss could occur during subsequent initialization processing.
API call limitations	Calls to the DATFRX API functions should be issued by an application program written in the C language. Operation cannot be guaranteed if such calls are issued by an interrupt handler or cycle handler.
Argument setting conventions and register guarantee conventions	The argument setting conventions and register guarantee conventions of DATFRX conform to the setting conventions and guarantee conventions of the C compiler. Refer to the related manuals for details.
Sections	Sections in areas with no initial value should be initialized to 0.

2. API Information

The operation of the FIT module has been confirmed under the conditions listed below.

2.1 Hardware Requirements

The microcontroller used must support the following functionality.

- Data flash memory

2.2 Software Requirements

DATFRX is dependent on the following packages when used with FIT support.

- r_bsp
- r_flash_rx

2.3 Supported Toolchain

The operation of the FIT module has been confirmed with the toolchain listed in 6.1, Confirmed Operation Environment.

2.4 Interrupt Vector

The FRDYI or FRDYIE interrupt is used to detect the completion of data writes to, and erasures from, the data flash memory. Enable interrupts on the system before calling the DATFRX open processing function R_FLASH_DM_Open(). For details of the FRDYI and FRDYIE interrupts, refer to application note for the flash FIT module.

2.5 Header Files

All the API calls and interface definitions used are listed in r_flash_dm_rx_if.h.

Select the structure options per build by r_datfrx_rx_config.h, and include them according to the following order.

2.6 Integer Types

This project uses ANSI C99. These types are defined in stdint.h.

2.7 Compile Settings

The configuration option settings for the control software are specified in r_datfrx_rx_config.h.

The option names and setting values are described below.

Set #define definitions for the flash FIT module r_flash_rx_config.h as follows:

#define FLASH_CFG_CODE_FLASH_ENABLE (0)

#define FLASH_CFG_CODE_FLASH_BGO (0)

#define FLASH_CFG_DATA_FLASH_BGO (1)

Table 2.1 Configuration options

Configuration options in r_datfrx_rx_config.h	
Definition	**Description**
FLASH_DM_CFG_FRDYI_INT_PRIORITY The default value is “1”	When using data flash memory BGO, define the FRDYI/FRDYIE interrupt priority level. The allowable setting range is 1 to 15.
FLASH_DM_CFG_DF_BLOCK_NUM The default value is “4”	Define the number of sblocks of data flash memory to be managed. The allowable setting ranges are: For Flash Type 1: 2 to 8 For Flash Types 2, 3, and 4: 2 to 1024 For information on blocks subject to data management, refer to 6.3, Data Management. For the maximum block count value, refer to the User’s Manual: Hardware of the MCU.
FLASH_DM_CFG_DF_DATA_NUM The default value is “10”	Define the number of data of data flash memory to be managed. The allowable setting ranges are: For Flash Type 1: 1 to 255 (data numbers managed: No. 0 to No. 254) For Flash Types 2, 3, and 4: 1 to 1024 (data numbers managed: No. 0 to No. 1023)
FLASH_DM_CFG_DF_SIZE_NOx The default value of data 0 is “1”. “x” represents the data number.	Define the data size of each data number in the data flash memory to be managed. The allowable setting ranges are: For Flash Type 1: 1 to 256 For Flash Types 2, 3, and 4: 1 to 1024 The setting values for unused data numbers are ignored. However, No. 40 and above are not defined, so it is necessary to add definitions separately. Alternatively, it is necessary to rewrite a portion of the source code.
FLASH_DM_CFG_CRC_HARDWARE The default value is “0”	This definition applies to Flash Types 2, 3, and 4. Specify whether or not the CRC unit of the RX MCU will be used. A setting of “0” means that CRC operations are performed in software. A setting of “1” means that CRC operations are performed by the RX MCU’s on-chip CRC unit, resulting in faster CRC performance. Note: This is for future deployment. In this version, only "0" is supported.

2.7.1 Adding Data Numbers

In order to use data numbers of No. 40 and above when managing the data flash memory, it is necessary to add separate definitions or to rewrite a portion of the source code.

In the example below, data numbers from No. 40 to No. 47 are added for data flash memory control.

2.7.1.1 Example Modification of r_datfrx_rx_config.h

Add the following lines of code.

Specify a data size value of your choice between the parentheses ( ).

[DATA FLASH : SET THE DATA LENGTH FOR THE DATA NUMBER]

```
#define FLASH_DM_CFG_DF_SIZE_NO40    (4)
#define FLASH_DM_CFG_DF_SIZE_NO41    (4)
#define FLASH_DM_CFG_DF_SIZE_NO42    (4)
#define FLASH_DM_CFG_DF_SIZE_NO43    (4)
#define FLASH_DM_CFG_DF_SIZE_NO44    (4)
#define FLASH_DM_CFG_DF_SIZE_NO45    (4)
#define FLASH_DM_CFG_DF_SIZE_NO46    (4)
#define FLASH_DM_CFG_DF_SIZE_NO47    (4)
```

2.7.1.2 Example Modification of r_dm_1.c, r_dm_2.c, r_dm_3.c, and r_dm_4.c

Remove the comment-start and comment-stop characters from const variable gc_dm_data_size[] for No. 40 to No. 47.

< Near line 314 in r_dm_1.c and near line 60 in r_dm_2.c, r_dm_3.c, and r_dm_4.c >

```
const uint16 gc_dm_data_size[] =
{
   FLASH_DM_CFG_DF_SIZE_NO0, FLASH_DM_CFG_DF_SIZE_NO1,
FLASH_DM_CFG_DF_SIZE_NO2, FLASH_DM_CFG_DF_SIZE_NO3,
   FLASH_DM_CFG_DF_SIZE_NO4, FLASH_DM_CFG_DF_SIZE_NO5,
FLASH_DM_CFG_DF_SIZE_NO6, FLASH_DM_CFG_DF_SIZE_NO7,

(Omitted)

/* FLASH_DM_CFG_DF_SIZE_NO40, FLASH_DM_CFG_DF_SIZE_NO41,
FLASH_DM_CFG_DF_SIZE_NO42, FLASH_DM_CFG_DF_SIZE_NO43, */ ← Remove the comment-start
and comment-stop characters at the beginning and end of the line.
/* FLASH_DM_CFG_DF_SIZE_NO44, FLASH_DM_CFG_DF_SIZE_NO45,
FLASH_DM_CFG_DF_SIZE_NO46, FLASH_DM_CFG_DF_SIZE_NO47, */ ← Remove the comment-start
and comment-stop characters at the beginning and end of the line.
```

2.8 Memory Usage

2.8.1 Flash Type 1

Table 2.2 lists the required memory sizes for data flash memory BGO.

Table 2.2 Memory Sizes (Data Flash Memory BGO)

MCU	Memory	Size (1), (2), (3), (4), (5), (6), (7),(8)
RX111	ROM	5045 bytes + (4 bytes × n Number of blocks) + (2 bytes × m Number of management data)
	RAM	31 bytes + (4bytes × n Number of blocks) + (2bytes × m Number of management data)
	Maximum user stack used	120 bytes
	Maximum interrupt stack used	56 bytes
RX231	ROM	5039 bytes + (4bytes × n Number of blocks) + (2bytes × m Number of blocks)
	RAM	33 bytes + (4bytes × n Number of blocks) + (2bytes × m Number of management data)
	Maximum user stack used	120 bytes
	Maximum interrupt stack used	56 bytes

Note 1. This is the value if the default setting from “2.7 Compile Settigns” is selected. The code size differs depending on the definitions selected.

Note 2. The execution conditions are as follows.

- r_flash_dm_rx_if.c
- r_dispatch_1.c
- r_dm_1.c

Note 3. The required memory size differs depending on the version of the C compiler, the compiler options, and the like.

Note 4. This is the value in the case of little-endian. The memory sizes indicated above differ depending on the endian order.

Note 5. n = 2 to 8

Note 6. m = 1 to 255

Note 7. The size of the Flash FIT module (On-chip flash memory Programing) is not included.

Note 8. The maximum use interrupt stack is the value when Excep_FCU_FRDYI() is called.

2.8.2 Flash Type 2

Table 2.3 Memory Sizes (Data Flash Memory BGO)

MCU	Memory	Size (1), (2), (3), (4), (5), (6), (7)
RX210	ROM	5099 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	192 bytes
	Maximum interrupt stack used	248 bytes
RX63N	ROM	5026 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	192 bytes
	Maximum interrupt stack used	248 bytes

Note 1. This is the value if the default setting from “2.7 Compile Settigns” is selected. The code size differs depending on the definitions selected.

Note 2. The execution conditions are as follows.

- r_flash_dm_rx_if.c
- r_dispatch_2.c
- r_datf_crc.c
- r_dm_2.c

Note 3. The required memory size differs depending on the version of the C compiler, the compiler options, and the like.

Note 4. This is the value in the case of little-endian. The memory sizes indicated above differ depending on the endian order.

Note 5. (RX210) n = 4 to 64 ,(RX63N) n = 4 to 1024

Note 6. The size of the Flash FIT module (On-chip flash memory Programing) is not included.

Note 7. The maximum use interrupt stack is the value when Excep_FCU_FRDYI() is called.

2.8.3 Flash Type 3

Table 2.4 Memory Sizes (Data Flash Memory BGO)

MCU	Memory	Size (1), (2), (3), (4), (5), (6), (7)
RX64M	ROM	4999 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	160 bytes
	Maximum interrupt stack used	72 bytes
RX71M	ROM	4999 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	160 bytes
	Maximum interrupt stack used	72 bytes
RX66T	ROM	4999 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	160 bytes
	Maximum interrupt stack used	72 bytes

Note 1. This is the value if the default setting from “2.7 Compile Settigns” is selected. The code size differs depending on the definitions selected.

Note 2. The execution conditions are as follows.

- r_flash_dm_rx_if.c
- r_dispatch_2.c
- r_datf_crc.c
- r_dm_3.c

Note 3. The required memory size differs depending on the version of the C compiler, the compiler options, and the like.

Note 4. This is the value in the case of little-endian. The memory sizes indicated above differ depending on the endian order.

Note 5. (RX64M) n = 4 to 1024 ,(RX71) n = 4 to 1024 ,(RX66T) n = 4 to 512

Note 6. The size of the Flash FIT module (On-chip flash memory Programing) is not included.

Note 7. The maximum use interrupt stack is the value when Excep_FCU_FRDYI() is called.

2.8.4 Flash Type 4

Table 2.5 Memory Sizes (Data Flash Memory BGO)

MCU	Memory	Size (1), (2), (3), (4), (5), (6), (7)
RX65N-2MB	ROM	4999 bytes
	RAM	18 bytes + (3 bytes × n Number of blocks)
	Maximum user stack used	156 bytes
	Maximum interrupt stack used	68 bytes

Note 1. This is the value if the default setting from “2.7 Compile Settigns” is selected. The code size differs depending on the definitions selected.

Note 2. The execution conditions are as follows.
- r_flash_dm_rx_if.c
- r_dispatch_2.c
- r_datf_crc.c
- r_dm_4.c

Note 3. The required memory size differs depending on the version of the C compiler, the compiler options, and the like.

Note 4. This is the value in the case of little-endian. The memory sizes indicated above differ depending on the endian order.

Note 5. (RX65N-2MB) n = 4 to 512

Note 6. The size of the Flash FIT module (On-chip flash memory Programing) is not included.

Note 7. The maximum use interrupt stack is the value when Excep_FCU_FRDYI() is called.

2.9 Arguments

The structure for the arguments of the API functions is shown below. This structure is listed in r_flash_dm_rx_if.h, along with the prototype declarations of the API functions.

```
typedef struct _flash_dm_info
{
    uint8_t     data_no;
    uint8_t     rsv[3];
    uint8_t   * p_data;
} st_flash_dm_info_t;
```

2.10 Return Values

The API function return values and error codes are shown below. This enumerated type is listed in r_flash_dm_rx_if.h, along with the prototype declarations of the API functions.

Table 2.6 Return Values

Return Value	Description
FLASH_DM_SUCCESS	Processing successful
FLASH_DM_ACCEPT	Accept processing successful
FLASH_DM_SUCCESS_REQUEST_ERASE	Processing successful, erase request
FLASH_DM_ADVANCE	Advance request
FLASH_DM_FINISH_FORMAT	Format successful
FLASH_DM_FINISH_WRITE	Data update successful
FLASH_DM_FINISH_ERASE	Block erase successful
FLASH_DM_FINISH_RECLAIM	Reclaim successful
FLASH_DM_FINISH_INITIALIZE	Initialization successful
FLASH_DM_NO_INVALID_BLOCK	No invalid blocks
FLASH_DM_ERR_INIT	Initialization processing failure
FLASH_DM_ERR_BUSY	Busy state
FLASH_DM_ERR_ARGUMENT	Parameter error
FLASH_DM_ERR_REQUEST_INIT	Initialization request
FLASH_DM_ERR_REQUEST_FORMAT	Format request
FLASH_DM_ERR_REQUEST_ERASE	Block erase request
FLASH_DM_ERR_DATA_NOT_PRESENT	Data number mismatch
FLASH_DM_ERR_CANT_RECLAIM	Cannot run reclaim processing
FLASH_DM_ERR_REQUEST_RECLAIM	Reclaim request
FLASH_DM_ERR_FORMAT	Format failure
FLASH_DM_ERR_WRITE	Data update failure
FLASH_DM_ERR_ERASE	Block erase failure
FLASH_DM_ERR_RECLAIM	Reclaim failure
FLASH_DM_ERR_OPEN	Open failure
FLASH_DM_ERR_CLOSE	Close failure
FLASH_DM_ERR	Error

2.11 Callback function

Upon the successful or error completion of format processing, data update processing, block erase processing, and reclaim processing, the user calls the specified callback function.

For information regarding how to register callback functions, see “3 API Functions”

Format

```
void user_cb_function(
  void* event
)
```

Parameters

*event

Stores the command result.

Return Values

None

Properties

Prototype declarations are contained in the user program.

Description

This function reports the end of format processing, data update processing, block erase processing, or reclaim processing.

The report details are stored in argument void* event. For instructions for obtaining the report details, refer to 3.1, R_FLASH_DM_Open().

Table 2.7 Return value to callback function

Value Stored in Argument	Meaning	Flash Type
FLASH_DM_FINISH_FORMAT	Format finished	1, 2, 3, 4
FLASH_DM_FINISH_WRITE	Data update finished	1, 2, 3, 4
FLASH_DM_FINISH_ERASE	Block erase finished	1, 2, 3, 4
FLASH_DM_FINISH_RECLAIM	Reclaim finished	1
FLASH_DM_ERR_FORMAT	Format failure	1, 2, 3, 4
FLASH_DM_ERR_WRITE	Data update failure	1, 2, 3, 4
FLASH_DM_ERR_ERASE	Block erase failure	1, 2, 3, 4
FLASH_DM_ERR_RECLAIM	Reclaim failure	1
FLASH_DM_ERR	Processing failure	1, 2, 3, 4

2.12 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends using “Smart Configurator” described in (1) or (3). However, “Smart Configurator” only supports some RX devices. Please use the methods of (2) or (4) for unsupported RX devices.

(1) Adding the FIT module to your project using “Smart Configurator” in e^2 studio
By using the “Smart Configurator” in e^2 studio, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using “FIT Configurator” in e^2 studio
By using the “FIT Configurator” in e^2 studio, the FIT module is automatically added to your project. Refer to “Adding Firmware Integration Technology Modules to Projects (R01AN1723)” for details.

(3) Adding the FIT module to your project using “Smart Configurator” on CS+
By using the “Smart Configurator Standalone version” in CS+, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(4) Adding the FIT module to your project in CS+
In CS+, please manually add the FIT module to your project. Refer to “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

2.13 *for*, *while*, and *do while* Expressions

This module uses *for*, *while*, and *do while* expressions (loop processing) for standby states such as waiting for register values to be updated. These instances of loop processing are indicated by the keyword WAIT_LOOP in the comments. Therefore, if you wish to incorporate failsafe processing into the instances of loop processing, you can locate them in the code by searching for the keyword WAIT_LOOP.

3. API Functions

R_FLASH_DM_Open()

This function is used first when starting data management processing. It reserves the work area used by DATFRX and registers the callback function.

Format

```
e_flash_dm_status_t R_FLASH_DM_Open(
    uint32_t* p_flash_dm_work,
    p_flash_dm_callback func
)
```

Parameters

** p_flash_dm_work*

Pointer to work area

The size of the area is as follows:

Flash Type 1: 140 bytes

Flash Types 2, 3, and 4: 261 + 2 bytes × FLASH_DM_CFG_DF_DATA_NUM

Prepare a work area that satisfies the above size requirement.

func

Pointer to callback function

Called when format processing, data update processing, block erase processing, or reclaim processing completes normally or with an error.

Return Values

FLASH_DM_SUCCESS	*Normal end*
FLASH_DM_ERR_ARGUMENT	*Parameter error* *→After checking the arguments, call the open function again.*
FLASH_DM_ERR_OPEN	*Flash FIT module open function error* *→Call the open function again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Calls the flash FIT module open function R_FLASH_Open().

Reentrant

Reentrancy is not supported.

Example

```
 static uint32_t g_flash_dm_work[314/sizeof(uint32_t)];

 void user_cb_function(void * event) /* callback function */
 {
     e_flash_dm_status_t callback_event = (e_flash_dm_status_t)event;

     /* Perform callback functionality here */
     switch(callback_event)
     {
         case FLASH_DM_FINISH_FORMAT:
         {
             nop();
         }
         break;
         case FLASH_DM_FINISH_WRITE:
         {
             nop();
         }
         break;
         /* : */
         /* : */
         default:
         {
             nop();
         }
         break;
     }
 }

 void main(void)
 {
     if (FLASH_DM_SUCCESS != R_FLASH_DM_Open(&g_flash_dm_work, &user_cb_function))
     {
         /* Error */
     }
}
```

Special Notes:

None

R_FLASH_DM_Close()

Ends data management and releases the work area used by DATFRX.

Format

```
e_flash_dm_status_t R_FLASH_DM_Close(
    void
)
```

Parameters

None

Return Values

FLASH_DM_SUCCESS	*Normal end*
FLASH_DM_ERR_CLOSE	*Close failure* *→ Call R_FLASH_DM_Close() once again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Ends data management and releases the work area used by DATFRX.

Calls the flash FIT module close function R_FLASH_Close().

Reentrant

Reentrancy is not supported.

Example

```
if (FLASH_DM_SUCCESS != R_FLASH_DM_Close())
{
   /* Error */
}
```

Special Notes:

None

R_FLASH_DM_Init()

Initializes the driver.

Format

```
e_flash_dm_status_t R_FLASH_DM_Init(
    void
)
```

Parameters

None

Return Values

FLASH_DM_SUCCESS	*Normal end (Flash Type 1)* *→ Flash Type 1 initialization has completed.*
FLASH_DM_ADVANCE	*Initialization processing in progress (Flash Types 2, 3, and 4)* *→ For Flash Types 2, 3, and 4, call FLASH_DM_InitAdvance() to complete initialization until FLASH_DM_SUCCESS is returned.*
FLASH_DM_SUCCESS_REQUEST_ERASE	*Normal end and erase request (Flash Type 1)* *→ Call the block erase function.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory in busy state* *→ After the API execution in progress completes, call the initialization function again.*
FLASH_DM_ERR_REQUEST_FORMAT	*Unformatted or block header error (Flash Type 1)* *→ Call format function.*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*
FLASH_DM_ERR_INIT	*Initialization error* *→ Confirm that R_FLASH_DM_Open() is running and call R_FLASH_DM_Init() again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Initializes DATFRX.

Run this function before running any API functions other than the format function R_FLASH_DM_Format().

For Flash Types 2, 3, and 4, initialization processing is divided because it takes a long time to complete.

After a value of FLASH_DM_ADVANCE is returned, call R_FLASH_DM_InitAdvance() to finish initialization processing.

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_status_t ret;

do
{
   ret = R_FLASH_DM_Init();
}while (FLASH_DM_ERR_BUSY == ret);
if(ret == FLASH_DM_ERR_REQUEST_FORMAT)
{
    ret = R_FLASH_DM_Format();
}
else if(ret == FLASH_DM_SUCCESS_REQUEST_ERASE)
{
    ret = R_FLASH_DM_Erase();
}
else if(ret != FLASH_DM_ADVANCE)
{
    ret = R_FLASH_DM_InitAdvance();
}
else
{
}
```

Special Notes:

The initialization processing is not non-blocking. The API waits internally for processing to finish.

Do not call this function from a callback function.

R_FLASH_DM_InitAdvance()

Continues execution of initialization processing.

This function is for Flash Types 2, 3, and 4 only.

Format

```
e_flash_dm_status_t R_FLASH_DM_InitAdvance(
    void
)
```

Parameters

None

Return Values

FLASH_DM_SUCCESS	*Normal end*
FLASH_DM_ADVANCE	*Initialization in progress (start next processing)* *→ Call the R_FLASH_DM_InitAdvance function again.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the API execution in progress completes, call the initialization function again.*
FLASH_DM_ERR_REQUEST_FORMAT	*Unformatted or block header error* *→ Call format function.*
FLASH_DM_ERR_REQUEST_INIT	*Initialization error* *→ Call the initialization function.*
FLASH_DM_ERR_INIT	*Initialization error* *→ Confirm that R_FLASH_DM_Open() is running and call R_FLASH_DM_Init() again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Continues DATFRX initialization processing.

After starting initialization processing with R_FLASH_DM_Init(), call R_FLASH_DM_InitAdvance() to finish it.

Finish initialization processing before running any API functions other than the format function R_FLASH_DM_Format().

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_status_t ret;

do
{
    ret = R_FLASH_DM_InitAdvance();
}
while (FLASH_DM_ERR_BUSY == ret);
```

Special Notes:
The initialization processing is not non-blocking. The API waits internally for processing to finish.

Do not call this function from a callback function.

R_FLASH_DM_Format()

Erases the data in the data flash memory.

Puts the data flash memory into a state in which initialization processing can be run.

(For Flash Type 1, it is not necessary to run initialization processing after format processing ends normally.)

Format

e_flash_dm_status_t R_FLASH_DM_Format(
    void
)

Parameters

None

Return Values

FLASH_DM_ACCEPT	*Processing accepted*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory in busy state →After the currently running processing finishes, call the format function again.*
FLASH_DM_ERR_FORMAT	*Format failure (problem with work area)*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Starts format processing.

Calls the callback function when an error occurs or format processing finishes.

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_status_t ret;

ret = FLASH_DM_Format();
if (FLASH_DM_ACCEPT != ret)
{
   /* Error */
}
else
{
   /* Initialization processing  */
}
```

Special Notes:

For Flash Types 2, 3, and 4, format processing does not initialize the driver. After format processing, immediately run driver initialization processing.

R_FLASH_DM_Read()

Reads the data associated with the specified data number.

Format

```
e_flash_dm_status_t R_FLASH_DM_Read(
    st_flash_dm_info_t * p_flash_dm_info
)
```

Parameters

** p_flash_dm_info*

DATFRX information structure

data_no

Read target data number

The allowable range of data numbers is 0 to (FLASH_DM_CFG_DF_DATA_NUM − 1).

**p_data*

Storage destination buffer for read data

The size of the area is the value specified by FLASH_DM_CFG_DF_SIZE_NOx.

Return Values

FLASH_DM_SUCCESS	*Normal end*
FLASH_DM_ERR_ARGUMENT	*Parameter error* *→ If the open function has not been called, call the open function and then call this function. If the open function has been called, check the arguments, then call the data read function again.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the currently running processing finishes, call the data read function again.*
FLASH_DM_ERR_DATA_NOT_PRESENT	*No data at specified data number in flash memory* *→ Check the data number, then call the data read function again.*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Reads the specified data from the data flash memory and stores it in the specified buffer.

If the data read function is run with a data number specified that is currently having its data updated, the old data previously written to the data flash memory is read because the new data from the update has not been established.

Reentrant

Reentrancy is not supported.

Example

```
st_flash_dm_info_t flash_dm_info;
static uint8_t g_test_r_buff[FLASH_DM_CFG_DF_SIZE_NO0];

flash_dm_info.data_no = 0;
flash_dm_info.p_data  = &g_test_r_buff[0];
if (FLASH_DM_SUCCESS != R_FLASH_DM_Read(&flash_dm_info))
{
   /* Error */
}
```

Special Notes:

None

R_FLASH_DM_Write()

Updates the data associated with the specified data number.

Format

```
e_flash_dm_status_t R_FLASH_DM_Write(
    st_flash_dm_info_t * p_flash_dm_info
)
```

Parameters

** p_flash_dm_info*

DATFRX information structure

data_no

Update target data number

The allowable range of data numbers is 0 to (FLASH_DM_CFG_DF_DATA_NUM − 1).

**p_data*

Update data storage source buffer

The size of the area is the value specified by FLASH_DM_CFG_DF_SIZE_NOx.

Return Values

FLASH_DM_ACCEPT	*Processing accepted*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*
FLASH_DM_ERR_ARGUMENT	*Parameter error* *→ If the open function has not been called, call the open function and then call this function. If the open function has been called, check the arguments, then call the data read function again.*
FLASH_DM_ERR_REQUEST_ERASE	*No erased blocks, so data update processing not possible (Flash Types 2, 3, and 4)* *→ Call the block erase function.*
FLASH_DM_ERR_REQUEST_RECLAIM	*No writable area within the active block for updating the data of the specified data number (Flash Type 1)* *→ Call the reclaim function.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the currently running processing finishes, call the data update function again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Starts updating the data associated with the specified data number. Writes the data in the specified buffer to the data flash memory.

Calls the callback function when an error occurs or the data update finishes.

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_status_t ret;
uint32_t status;
st_flash_dm_info_t flash_dm_info;
static uint8_t g_test_w_buff[FLASH_DM_CFG_DF_SIZE_NO0];

flash_dm_info.data_no = 0;
flash_dm_info.p_data  = &g_test_w_buff[0];
if (FLASH_DM_ACCEPT != R_FLASH_DM_Write(&flash_dm_info))
{
   /* Reclaim or error */
}
do
{
   ret = R_FLASH_DM_Control(FLASH_DM_GET_STATUS, &status);
   if (FLASH_DM_SUCCESS == ret)
   {
        if(status == FLASH_DM_ACT_IDLE)
        {
            break;
        }
   }
}while(1);
```

Special Notes:

Data update error end is returned if an error occurs during programming. Call the data update function again. The write destination address is updated and data update processing runs.

Do not change the value of p_data until data update processing finishes. If the value is changed, the update data may be incorrect.

R_FLASH_DM_Erase()

Erases a block.

Format

```
e_flash_dm_status_t R_FLASH_DM_Erase(
    void
)
```

Parameters

None

Return Values

FLASH_DM_ACCEPT	*Processing accepted*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*
FLASH_DM_NO_INVALID_BLOCK	*No invalid blocks*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the currently running processing finishes, call the block erase function again.*
FLASH_DM_ERR_ERASE	*Block erase failure (problem with work area)*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Erases an invalid block to create an erased block.

If this function is called when there are no invalid blocks, block erase processing does not run.

Calls the callback function when an error occurs or block erase finishes.

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_driver_status_t ret = FLASH_DM_SUCCESS;

if (FLASH_DM_ACCEPT != R_FLASH_DM_Erase())
{
   /* Error */
}
do
{
   ret = R_FLASH_DM_Control(FLASH_DM_GET_STATUS, &status);
   if (FLASH_DM_SUCCESS == ret)
   {
        if(status == FLASH_DM_ACT_IDLE)
        {
            break;
        }
   }
}
while(1);
```

Special Notes:

Block erase error end is returned if an error occurs during programming or block erasing. Call the block erase function again. Block erase processing is performed on the physical block where the error occurred. Note that repeated incidences of block erase error end may indicate deterioration of the data flash memory.

R_FLASH_DM_Reclaim()

Starts reclaim processing.

Reserves the capacity needed for data updating.

This function is for Flash Type 1 only.

Format

```
e_flash_dm_status_t R_FLASH_DM_Reclaim(
    void
)
```

Parameters

None

Return Values

FLASH_DM_ACCEPT	*Processing accepted*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*
FLASH_DM_ERR_REQUEST_ERASE	*No erased block, so reclaim processing cannot run* *→ Call the block erase function.*
FLASH_DM_ERR_CANT_RECLAIM	*No erased block or erase target block, so reclaim processing cannot run* *→ Call the format function.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the currently running processing finishes, call the reclaim function again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Switches the active block and copies all valid data from the oldest reclaim block to the new active block. Sets the reclaim block that was the source of the copied data as a garbage block.

Reclaim processing includes copying of all valid data in the block, so it takes some time to finish.

Reentrant

Reentrancy is not supported.

Example

```
e_flash_dm_driver_status_t ret = FLASH_DM_SUCCESS;

if (FLASH_DM_ACCEPT != R_FLASH_DM_Reclaim())
{
   /* Error */
}
do
{
   ret = R_FLASH_DM_Control(FLASH_DM_GET_STATUS, &status);
   if (FLASH_DM_SUCCESS == ret)
   {
        if(status == FLASH_DM_ACT_IDLE)
        {
            break;
        }
   }
}
while(1);
```

Special Notes:

When the reclaim processing function R_FLASH_DM_Reclaim() is called, reclaim processing starts, regardless of whether any writeable area remains in the active block. To increase the data update count, call the reclaim function R_FLASH_DM_Reclaim() when there is no writeable area remaining in the active block.*[1]

When reclaim processing ends normally, the setting of the copy source reclaim block is changed from garbage block to erase target block. It is then necessary to run block erase processing before running reclaim processing the next time.*[2]

If an error occurs during programming, reclaim error end is returned.*[3]

Note 1. This can be determined from the return value after calling the data update function R_FLASH_DM_Write().

Note 2. It is possible to determine the appropriate timing for block erase from the return value after calling the reclaim function R_FLASH_DM_Reclaim(). It is recommended that block erase processing be run periodically when there is extra time available on the user system.

Note 3. To reconstruct the data when a reclaim error end occurs, run initialization processing and then run the appropriate processing based on the return value. Afterwards, call the reclaim function R_FLASH_DM_Reclaim() or the data update function R_FLASH_DM_Write(). Note that calling the data update function R_FLASH_DM_Write() will return a value that serves as a call request for the reclaim function R_FLASH_DM_Reclaim().

R_FLASH_DM_Control()

The control function is used to embed various functionalities.

Format

```
e_flash_dm_status_t R_FLASH_DM_Control(
    e_flash_dm_cmd_t cmd,
    uint32_t* pcfg
)
```

Parameters

cmd

Command to be run

*pcfg

Argument for specifying a setting passed to the command as a request. This argument may be set to NULL if there is no command request.

Return Values

FLASH_DM_SUCCESS	*Protection processing normal end*
FLASH_DM_ERR_REQUEST_INIT	*Uninitialized state* *→ Call the initialization function.*
FLASH_DM_ERR_ARGUMENT	*Parameter error* *→ If the open function has not been called, call the open function and then call this function. If the open function has been called, check the arguments, then call the data read function again.*
FLASH_DM_ERR_BUSY	*API execution in progress or flash memory busy state* *→ After the currently running processing finishes, call the control function again.*
FLASH_DM_ERR	*Module run error* *→ Call the control function again.*

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

This is an extended function for embedding sequencer functionality other than read, write, and block erase. The argument type differs depending on the command type.

Table 3.1 Control Function Options

Command	Argument	Operation
FLASH_DM_GET_WRITABLE_SIZE	uint32_t*	Flash Type 1: Gets the size of the writeable area within the active block*1 (unit: bytes), and stores it in an argument. Data updating can be performed on user data equal to or less than this value.*2 Flash Types 2, 3, and 4: Gets the size of the writeable area based on the erased blocks (unit: bytes), and stores it in an argument.
FLASH_DM_GET_STATUS	uint32_t*	Determines whether or not an API function called by the user is currently running, and stores the result in an argument. 0x00:FLASH_DM_ACT_IDLE (idle) 0x01:FLASH_DM_ACT_WRITING (data update processing in progress) 0x02:FLASH_DM_ACT_RECLAIMING (reclaim processing in progress) 0x04:FLASH_DM_ACT_ERASING (block erase processing in progress) 0x08:FLASH_DM_ACT_FORMATTING (format processing in progress) 0x10:FLASH_DM_ACT_INITIALIZING (initialization processing in progress)
FLASH_DM_GET_DATA_SIZE	uint32_t*	Gets the data size of the data number specified by an argument, and stores the result in an argument.
FLASH_DM_GET_DATA_NUM	uint32_t*	Gets the user setting data count, and stores the result in an argument.

Note 1. The programming unit size is necessary as a separator between the data header and the user data. The size of the writeable area is equal to the size of the empty area, minus the programming unit size and the data header size.

Note 2. If the block is uninitialized, it is not possible to get the size of the writeable area when format processing, data update processing, or reclaim processing is in progress. Also, the size of the writeable area cannot be obtained when any other API function is running.
In the case of data flash memory, the returned value has had a total of 8 bytes subtracted: 7 bytes for the data header required for data update processing and 1 byte for the separator between the data header and user data.

Reentrant

This function does not support reentrancy.

Example 1: Obtaining the API Status (Flash Types 1, 2, 3, and 4)

```
e_flash_dm_status_t ret;
uint32_t* pcfg;

ret = R_FLASH_DM_Control(FLASH_DM_GET_STATUS, pcfg);
if (FLASH_DM_SUCCESS != ret)
{
   /* Error */
}
```

Example 2: Obtaining the Data Count (Flash Types 1, 2, 3, and 4)

```
e_flash_dm_status_t ret;
uint32_t* pcfg;

ret = R_FLASH_DM_Control(FLASH_DM_GET_DATA_NUM, pcfg);
if (FLASH_DM_SUCCESS != ret)
{
   /* Error */
}
```

Special Notes:

None

R_FLASH_DM_GetVersion()

Gets the DATFRX version information.

Format

```
uint32_t R_FLASH_DM_GetVersion(
    void
)
```

Parameters

None

Return Values

Upper 2 bytes:	Major version (decimal notation)
Lower 2 bytes:	Minor version (decimal notation)

Properties

Prototype declarations are contained in r_flash_dm_rx_if.h.

Description

Returns the version information.

Reentrant

Reentrancy is supported.

Example

```
uint32_t version;
version = R_FLASH_DM_GetVersion();
```

Special Notes:

None

4. Pin Settings

DATFRX does not use any pins, so no pin settings are required.

5. Demo Project

The project listed below is contained in the FITDemos folder. This folder also contains the sample program r_datfrx_rx_main.c.

Table 5.1 Project List

User Conditions					
MCU	**Flash Type**	**Flash Memory**	**Block Size**	**IDE**	**Project**
RX66T	Flash Type 3	Data flash	64 bytes	e² studio	type3_rx66t_rsk_sample
RX65N-2MB	Flash Type 4	Data flash	64 bytes	e² studio	type4_rx65n_2mb_rsk_sample

The sample program opens DATFRX and performs initialization processing; performs data update, data read, and verification check in sequence 50 times; then closes DATFRX.

When one set of data update, data read, and verification check completes, the data number is incremented by 1. (In the demo, the data number rises in sequence from 0 to 13.)

During initialization processing, formatting is performed if required.

During data update processing, block erasing is performed if required.

5.1 Adding the Demo to the Workspace

The demo project is contained in the FITDemos subdirectory, which is created when the archive file in which this application note is distributed is opened. To add the demo project to the workspace, select **File → Import**, then select **Add Existing Project to Workspace** under **General** in the Import dialog box and click the **Next** button. In the Import dialog box select the **Select Archive File** radio button, click the Browse button, open the FITDemos subdirectory, select the zip file containing the demo, and click **Done**.

5.2 Downloading the Demo

The demo project is not included in the RX Driver Package. In order to use the demo project, you must download the necessary FIT modules separately. From the **Application Notes** tab under **Smart Browser**, right-click this application note and select **Sample Code (Download)** to begin the download.

6. Appendix

6.1 Confirmed Operation Environment

Table 6.1 shows the environment in which operation has been confirmed.

Table 6.1 Confirmed Operation Environment

Item	Description
Integrated development envronment	Renesas Electronics e^2 studio V7.2.0 Renesas Electronics CS+ V8.0.0
C compiler	Renesas Electronics C/C++ compiler for RX Family V.3.00.00
	Compile option: The following option is added to the default settings of the integrated development environment. -lang = c99
Endian order	Big-endian/little-endian
Module revision	Ver. 2.01
Board used	Renesas Starter Kit for RX231 (product No.: R0K505231xxxxxx) Renesas Starter Kit for RX210 (product No.: R0K505210xxxxxx) Renesas Starter Kit for RX66T (product No.: RTK50566T0Cxxxxxxx) Renesas Starter Kit+ for RX65N-2MB (product No.: RTK50565N2S1xxxxxx)

6.2 Troubleshooting

1. Q: I added the FIT module to my project, but when I build it I get the error “Could not open source file ‘platform.h’.”
   A: The FIT module may not have been added to the project properly. Refer to the documents listed below to confirm the method for adding FIT modules:
      1. Using CS+
         Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)
      2. Using e^2 studio
         Adding Firmware Integration Technology Modules to Projects (R01AN1723)

      When using the FIT module, the RX Family board support package FIT module (BSP module) must also be added to the project. Refer to the application note “RX Family: Board Support Package Module Using Firmware Integration Technology” (R01AN1685) for instructions for adding the BSP module.

2. Q: I added the FIT module to the project, but when I build it I get the error “This MCU is not supported by the current r_datfrx_rx module.”
   A: The FIT module you added may not support the target device chosen in the user project. Check to make sure the FIT module supports the target device.

6.3 Data Management

The DATFRX data management areas are described below.

6.3.1 DATFRX Areas

The block areas managed by DATFRX are illustrated below.

The read addresses in the boxes below match those indicated in User's Manual: Hardware. Block A to block H are the DATFRX management numbers.

6.3.1.1 1 Block = 1,024 Bytes (Flash Type 1)

Table 6.2 1 Block = 1,024 Bytes (Flash Type 1)

1 block = 1,024 bytes Read address	Blocks managed by flash FIT module	Blocks managed by DATFRX
0x0010 0000 0x0010 03FF	DB0000	Block A (start block)
0x0010 0400 0x0010 07FF	DB0001	Block B
0x0010 0800 0x0010 0BFF	DB0002	Block C
0x0010 0C00 0x0010 0FFF	DB0003	Block D
0x0010 1000 0x0010 13FF	DB0004	Block E
0x0010 1400 0x0010 17FF	DB0005	Block F
0x0010 1800 0x0010 1BFF	DB0006	Block G
0x0010 1C00 0x0010 1FFF	DB0007	Block H (end block)

6.3.1.2 1 Block = 128 Bytes (Flash Type 2)

Table 6.3 1 Block = 128 Bytes (Flash Type 2)

1 block = 128 bytes Read address	Blocks managed by flash FIT module	Blocks managed by DATFRX
0x0010 0000 0x0010 007F	Block 0 (128 bytes)	Block A
0x0010 0080 0x0010 00FF	Block 1 (128 bytes)	Block B
:	:	:
0x0010 7F80 0x0010 1FFF	Block 63 (128 bytes)	Block Z

6.3.1.3 1 Block = 64 Bytes (Flash Types 3 and 4)

Table 6.4 1 Block = 64 Bytes (Flash Types 3 and 4)

1 block = 64 bytes Read address	Blocks managed by flash FIT module	Blocks managed by flash FIT module	Blocks managed by DATFRX
0x0010 0000 0x0010 003F	Block 0 (64 bytes)	Block 0 (64 bytes)	Block A
0x0010 0040 0x0010 007F	Block 1 (64 bytes)	Block 1 (64 bytes)	Block B
:	:	:	:
0x0010 7FC0 0x0010 7FFF	:	Block 511 (64 bytes)	Block Y
0x0010 FFC0 0x0010 FFFF	Block 1023 (64 bytes)	—	Block Z

6.3.1.4 1 Block = 32 Bytes (Flash Type 2)

Table 6.5 1 Block = 32 Bytes (Flash Type 2)

1 block = 32 bytes Read address	Blocks managed by flash FIT module	Blocks managed by DATFRX
0x0010 0000 0x0010 001F	Block 0 (32 bytes)	Block A
0x0010 0020 0x0010 003F	Block 1 (32 bytes)	Block B
:	:	:
0x0010 7FD0 0x0010 7FE0	Block 1,023 (32 bytes)	Block Z

6.3.2 Block Areas (Flash Type 1)

The block format is described below. Each block is divided into a block header area, data header area, and user data area. There is also an empty area between the data header area and user data area.

6.3.2.1 Block Header (Flash Type 1)

The block header area is used to manage the block. It contains the erase start flag, erase end flag, initialized flag, etc.

The block is managed by the block header area. The flags listed below are recorded in the block header area. The state of the block flags is checked during initialization processing to determine the block type.

Table 6.6 Data (Flash Type 1)

Flag Name	Processing	Command 1	Command 2
Erase start flag	Block erase processing	Before erase command runs 0x00	Erase command successful 0xFF
Erase end flag	Block erase processing	Block erase end 0xAA	—
Initialized flag	Block erase processing	Initialized block creation end 0x00	—
Active flag	Reclaim processing	Active flag switching end 0x00	—
Full flag	Reclaim processing	Reclaim processing start 0x00	—
Reclaim flag	Reclaim processing	Reclaim processing end	—

6.3.2.2 Data Header (Flash Type 1)

The data header area is used to manage the user data. A data header is created each time a data update request is received. If a second data update request is received for the same data number, a new data header with the same data number is created within the active block. The data header with the higher address is determined to be newest (valid). During initialization processing the data headers of reclaim blocks and the active block are checked, and data headers are programmed such that their addresses increase starting immediately after the block header. The data header format is shown below.

Table 6.7 Data Flash Memory (Flash Type 1)

(1 block: 1,024 bytes) program unit: 1 byte

Offset	Description	Size
0x000	Data header update start flag	1 byte
0x001	Data number	1 byte
0x002	Data address (lower bits)*1	1 byte
0x003	Data address (upper bits)*1	1 byte
0x004	Data header update end flag	1 byte
0x005	Data update end flag	1 byte
0x006	Valid flag	1 byte

Note 1. This is the data allocation when little-endian byte order is used. When big-endian byte order is used, the data allocation for the data address (lower bits) and data address (upper bits) is reversed.

Table 6.8 Flag Types

Flag Name	Processing
Data header update start flag	A value of 0x7F is programmed when a data header write starts. This indicates the existence of a data header. The flag is checked during initialization processing, and if the value is other than 0x7F or 0xFF, the data header is determined to be invalid.
Data number	Data numbers are set by the user in r_datfrx_config.h. A separate data size can be specified for each data number. For the setting method, refer to 2.7, Compile Settings.
Data address	The actual address at which user data is stored.
Data header update end flag	After the data number and data address are programmed, a value of 0xBF is programmed to this flag. This indicates that the data number and data address have been programmed. The flag is checked during initialization processing, and if the value is other than 0xBF, the data header is determined to be unprogrammed and is treated as invalid.
Data update end flag	After the user data is programmed, a value of 0xDF is programmed to this flag. This indicates that programming of user data has finished. The flag is checked during initialization processing, and if the value is other than 0xDF, programming of the user data is determined to be incomplete and the data header is treated as invalid.
Valid flag	After the data update end flag is programmed, a value of 0x0F is programmed to this flag. This indicates the validity of the data header. The flag is checked during initialization processing, and if the value is other than 0x0F, data update processing is determined not to have finished and the data header is treated as invalid.

6.3.2.3 Data (Flash Type 1)

Table 6.9 Data Flash Memory (Flash Type 1)

(1 block: 1,024 bytes) program unit: 1 byte

Offset	Description	Size	Area
0x000	Erase start flag	2 bytes	Block header area
0x002	Erase end flag	8 bytes	
0x00A	Initialized flag	2 bytes	
0x00C	Active flag	2 bytes	
0x00E	Full flag	2 bytes	
0x010	Reclaim flag	2 bytes	
0x012	Data header (a)	7 bytes	Data header area
0x019	Data header (b)	7 bytes	
0x020	Data header (c)	7 bytes	
:	:		
:	Data header (n)	7 bytes	
	↓ ↑		Empty area
0x400 - Size (a..n)	User data (n)	Size (n)	User data area
:	:		
0x400 - Size (a..c)	User data (c)	Size (c)	
0x400 - Size (a..b)	User data (b)	Size (b)	
0x400 - Size (a)	User data (a)	Size (a)	

Size (a): Data size of user data a

Size (a..c): Total data size of user data a to user data n

6.3.3 Block Management (Flash Types 2, 3, and 4)

6.3.3.1 Block Header (Flash Types 2, 3, and 4)

Flash Types 2, 3, and 4 do not have a block header area.

6.3.3.2 Data Header (Flash Types 2, 3, and 4)

The state of the block is checked directly during initialization processing to determine the block type.

The management information within each block is summarized below.

Table 6.10 Data Flash Memory (Flash Types 2, 3, and 4)

Flag Name	Processing	Command
data_type	Identification of the data type.	data_type value*1 1: First block in long format 2: Intermediate block in long format 4: End block in long format 8: Short format block For format 1, data_type = 1 or data_type = 8. For format 2, data_type = 2 or data_type = 4.
chain	Extraction of the block number containing the next portion of user data for cases where the user data will not fit in a single block.	For long format, stores the block number information of the next block where user data is stored. When data_type = 4 or data_type = 8, stores a value of 0xFFFF because there is no next block number containing user data.
data_No	Extraction of the data number where the user data is stored.	Can be set such that data number < data count.
ser_No	Identification of whether user data is new or old.	The serial number is incremented for each data update, regardless of the pass/fail status. Update count: 0xFFFFFFFF max. The maximum value of 0xFFFFFFFF is the upper limit imposed by the software. It is not equivalent to the maximum number of data updates supported by the data flash memory.
crc_ccitt	Determining if stored management information is correct.	CRC codes are generated for the following management information: (1) data_No (2) ser_No
write_end	Confirming that data updating has finished.	This flag is programmed as the last step of data update processing. A data value of 0x0000 is programmed to the flag. When data update processing ends successfully, this flag remains in the unerased state.
erase_start	Determining whether or not block erase processing ended with an error.	This flag is programmed before erasing starts. A data value of 0x0000 is programmed to the flag. When this flag is in the unerased state, block erase processing is determined to have ended with an error. The flag is erased when block erase processing finishes successfully.

Flag Name	Processing	Command
erase_end	Confirming completion of block erase processing.	This flag is programmed as the last step of block erase processing. A data value of 0x0000 is programmed to the flag. When block erase processing ends successfully, this flag remains in the unerased state.

Note 1. Short format is suitable for cases where the user data will fit in a single block, and it is configured as format 1. Long format is used when the user data extends over more than one block (cases where the user data will not fit in a single block). They are configured as format 1 and format 2.

6.3.3.3 Data (Flash Types 2, 3, and 4)

Short format is suitable for cases where the user data will fit in a single block, and it is configured as format 1. Long format is used when the user data extends over more than one block (cases where the user data will not fit in a single block). They are configured as format 1 and format 2.

(1) Format 1

Format 1 is suitable for the start block when storing one unit of user data.

Table 6.11 Format 1

Flash Type 2				Flash Type 3		Flash Type 4			
128 Bytes		32 Bytes		64 Bytes		64 Bytes			
Address	Size	Address	Size	Address	Size	Address	Size	Data	Symbol
0x00	1	0x00	1	0x00	1	0x00	1	Data type	data_type
0x01	2	0x01	2	0x01	2	0x01	2	Chain information	chain
0x03	2	0x03	2	0x03	2	0x03	2	Data number	data_No
0x05	4	0x05	4	0x05	4	0x05	4	Serial number	ser_No
0x09	2	0x09	2	0x09	2	0x09	2	CRC code	crc_ccitt
0x0B	111	0x0B	15	0x0B	41	0x0B	41	User data	data
0x7A	2	0x1A	2	0x34	4	0x34	4	Write end flag	write_end
0x7C	2	0x1C	2	0x38	4	0x38	4	Erase start flag	erase_start
0x7E	2	0x1E	2	0x3C	4	0x3C	4	Erase end flag	erase_end

(2) Format 2

Format 2 suitable for the second and subsequent blocks in cases where the user data extends over more than one block (long format).

Table 6.12 Format 2

Flash Type 2				Flash Type 3		Flash Type 4			
128 Bytes		32 Bytes		64 Bytes		128 Bytes			
Address	Size	Address	Size	Address	Size	Address	Size	Data	Symbol
0x00	1	0x00	1	0x00	1	0x00	1	Data type	data_type
0x01	2	0x01	2	0x01	2	0x01	2	Chain information	chain
0x03	121	0x03	25	0x03	53	0x03	53	User data	data
0x78	2	0x18	2	0x38	4	0x38	4	Erase start flag	erase_start
0x7C	2	0x1C	2	0x3C	4	0x3C	4	Erase end flag	erase_end

6.3.4 Block States and How They Are Determined

6.3.4.1 Flash Type 1

(1) Block States (Flash Type 1)

Each physical block (Flash Type 1) managed by DATFRX is classified into one of the states listed below.

Table 6.13 Block States and How They Are Determined

Block State	Description	Determination
Erase target	A block in this state has been set as a block erase target, meaning that it needs to be erased. Block erase processing changes the state from erase target block to initialized block.	When the value of the erase start flag is other than 0xFF, the block is determined to be an erase target block. If even one of the bytes of data in the erase end flag has a value other than 0xAA, the block is determined to be an erase target block.
Initialized	A block in this state has no user data written to it and is ready to be used. One initialized block is necessary. Format processing sets block B as the initialized block. Block erase processing creates an initialized block.	If the block is not an erased block and the value of the active flag is 0xFF, the block is determined to be the initialized block.
Active	A block in this state has user data written to it and is the target block for data updates. One active block is necessary. Format processing sets block A as the active block. Reclaim processing (switching the active block and copying valid data) changes the state from active block to full block. Then the initialized block is changed to the active block. Consequently, after formatting the active block changes in the following sequence: block A → block B → block C → block D → block E → block F → block G → block H → block A.	If the block is not the initialized block and the value of the full flag is 0xFF, the block is determined to be the active block. If more than one block is determined to be set as the active block, an error is determined to have occurred. Format processing is necessary to correct such errors.
Full	A block in this state is an active block on which no additional area remains for data updates. This setting is only used during reclaim processing. After reclaim processing ends the state changes from full block to reclaim block.	If the block is not the active block and the value of the reclaim flag is 0xFF, the block is determined to be the full block. If more than one block is determined to be set as the full block, an error is determined to have occurred. Format processing is necessary to correct such errors.
Reclaim	A block in this state is a read-only block on which no additional area remains for data updates. Reclaim processing uses the oldest reclaim block as the copy source block and copies the valid data to the new active block. When reclaim processing ends, the copy source block is set as a garbage block.	If a block cannot be determined to be in any of the other states, it is determined to be a reclaim block.
Garbage	A block in this state is a copy source block which no longer contains any valid data due to reclaim processing, or a block in which a block header error was detected during initialization processing. Block erase processing treats garbage blocks as erase target blocks.	If the next block after the active block is not the initialized block, it is set as a garbage block. When there is one full block and one active block, the active block is set as a garbage block.

Block State	Description	Determination
Erased	A block in this state has only the erase end flag programmed after the block erase command completes. If block erase processing is halted midway, the block is treated as one determined to have a block header error by initialization processing.	If the block is not an erase target block and the value of the initialized flag is 0xFF, the block is determined to be an erased block.

(2) Block State Transitions (Flash Type 1)

The state transitions of the physical blocks (Flash Type 1) managed by DATFRX are shown below.

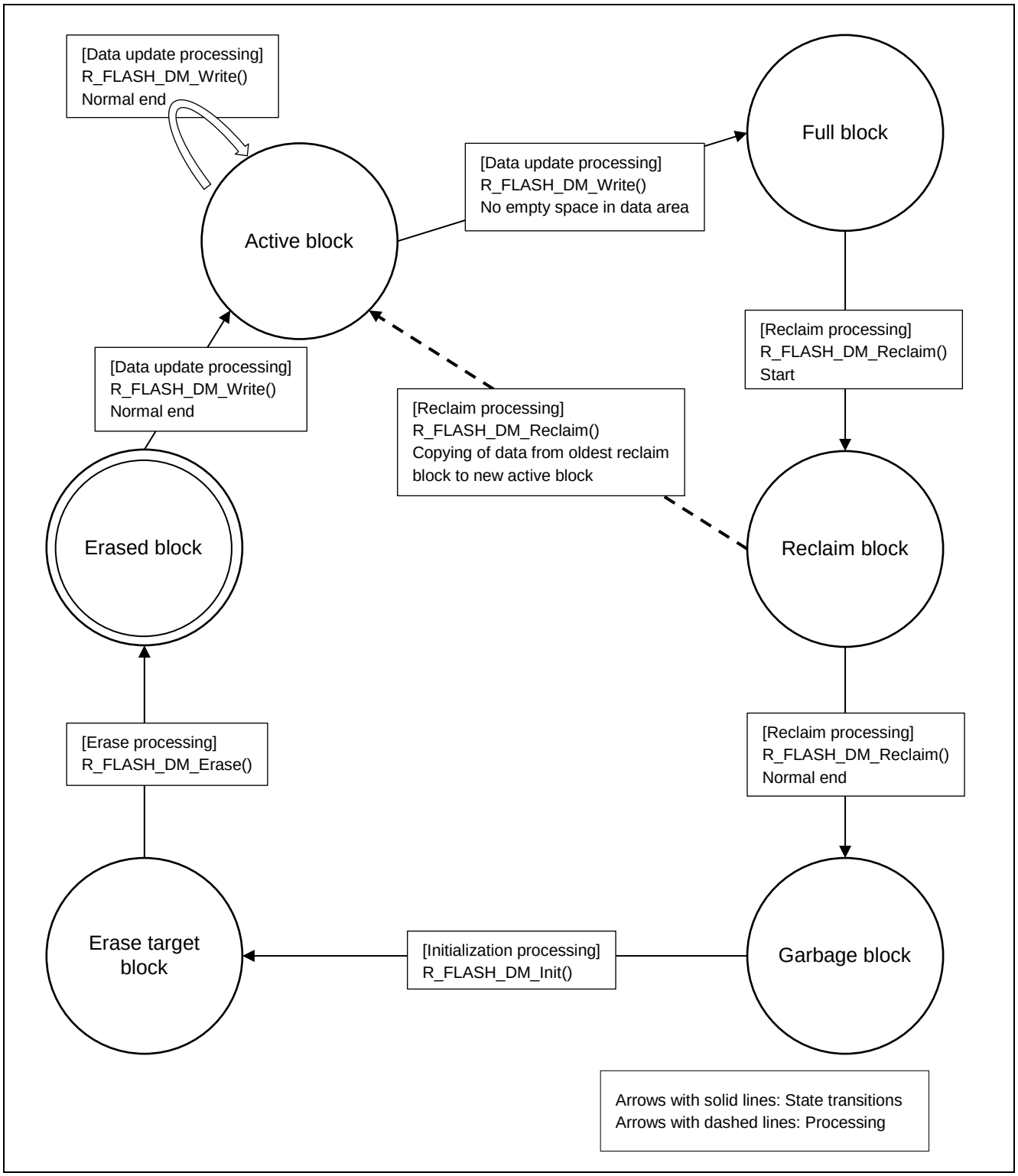

Figure 6.1 Block State Transitions (Flash Type 1)

(3) Block State Determination Flowchart (Flash Type 1)

The states of physical blocks (Flash Type 1) managed by DATFRX are determined as shown below.

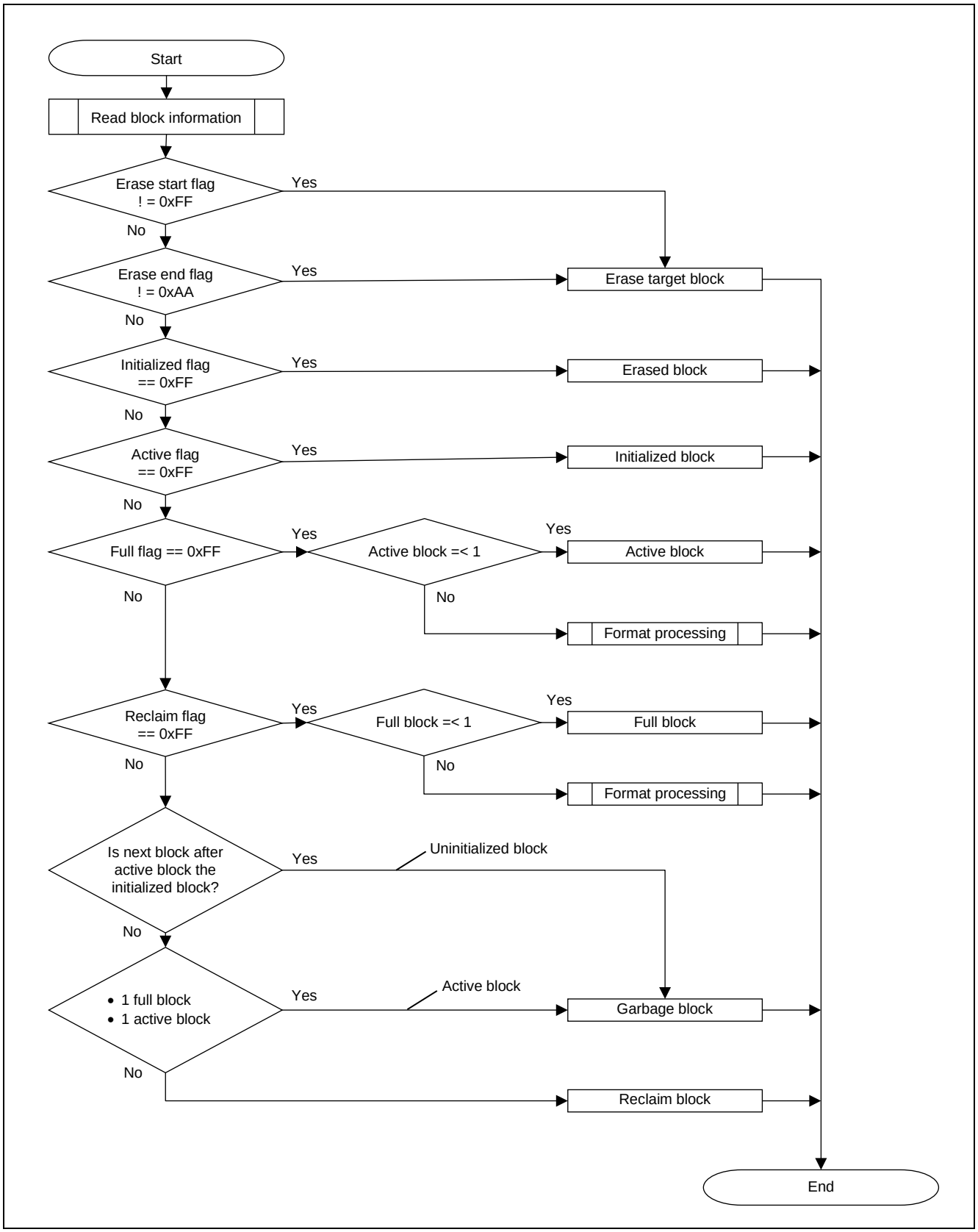

Figure 6.2 Block State Determination Flowchart (Flash Type 1)

6.3.4.2 Flash Types 2, 3, and 4

(1) Block States (Flash Types 2, 3, and 4)

Each physical block (Flash Types 2, 3, and 4) managed by DATFRX is classified into one of the states listed below.

Table 6.14 Block States (Flash Types 2, 3, and 4)

Block State	Description	Flag	Flag State Used to Determine
Valid block	A block in which valid user data is stored.*1	data_type	Block is in unerased state, and value matches specifications.
		chain	Block is in unerased state, and block number is less than the total block count.
		data_No	Block is in unerased state, and data number is less than the registered data count.
		ser_No	Block is in unerased state, and the number is the highest among those with the same data number.
		crc_ccitt	Block is in unerased state, and the codes generated from data_No and ser_No match.
		data	A block in which the portions where data is stored are in the unerased state, and the other portions are in the erased state.
		write_end	Block is in unerased state.
		erase_start	Block is in erased state.
		erase_end	Block is in unerased state.
Erased block	A block from which the data has been successfully erased. Data update processing is performed on erased blocks.	data_type	Block is in erased state.
		chain	Block is in erased state.
		data_No	Block is in erased state.
		ser_No	Block is in erased state.
		crc_ccitt	Block is in erased state.
		data	Block is in erased state.
		write_end	Block is in erased state.
		erase_start	Block is in erased state.
		erase_end	Block is in unerased state.
Invalid block	A block of a state other than the above. Block erase processing is performed on invalid blocks.	A block that cannot be determined to be a valid block or an erased block is determined to be an invalid block.	

Note 1. For format 2, if the start block is a valid block, the intermediate blocks and end block are also determined to be valid blocks.

(2) Block State Transitions (Flash Types 2, 3, and 4)

The state transitions of the physical blocks (Flash Types 2, 3, and 4) managed by DATFRX are shown below.

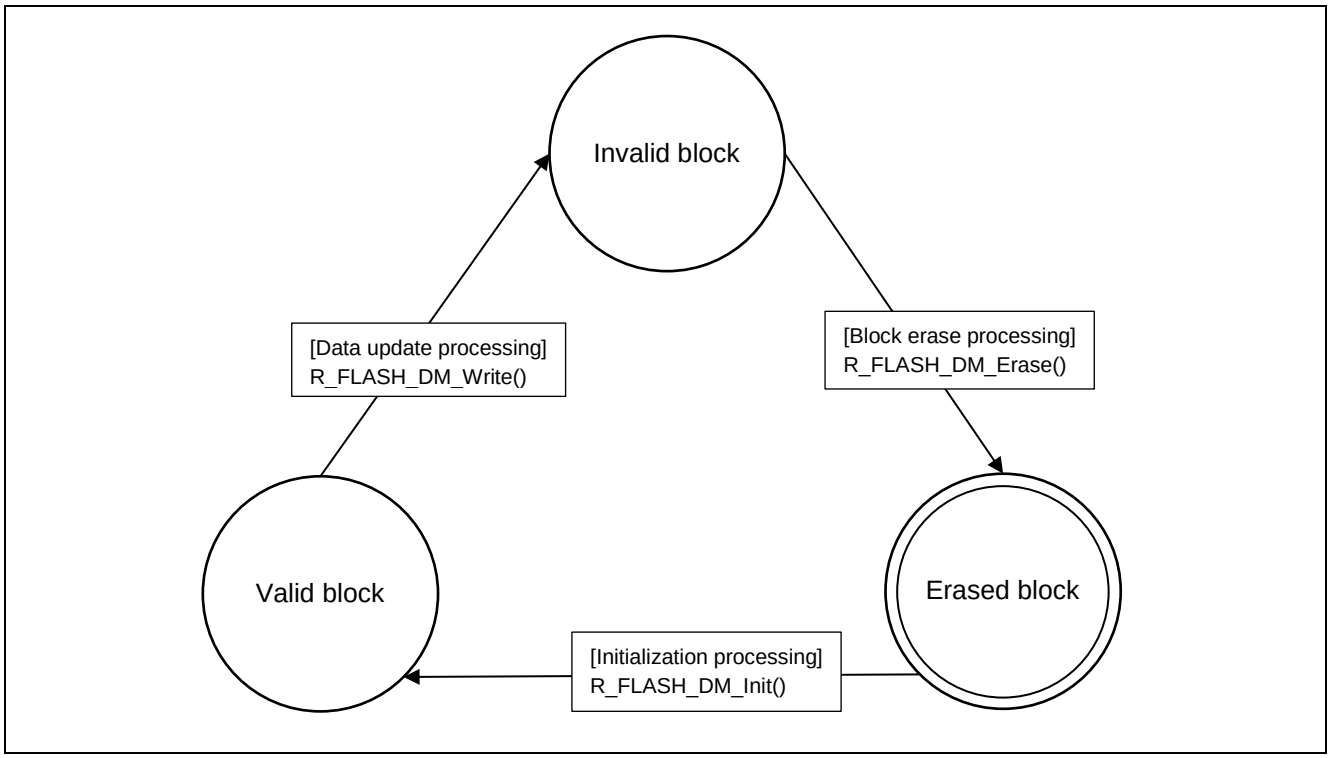

Figure 6.3 Block States (Flash Types 2, 3, and 4)

(3) Block State Determination Flowchart (Flash Types 2, 3, and 4)

The states of physical blocks (Flash Types 2, 3, and 4) managed by DATFRX are determined as shown below.

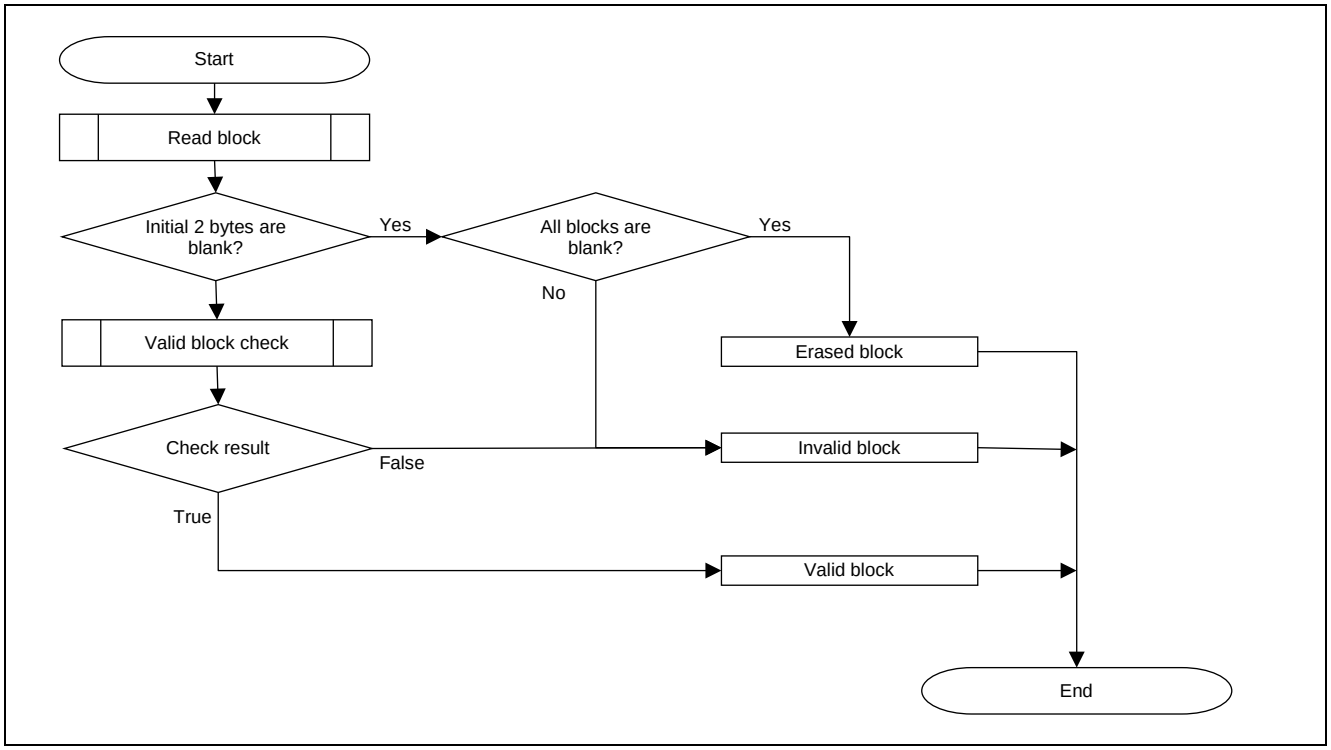

Figure 6.4 Block State Determination Flowchart (Flash Types 2, 3, and 4)

7. Reference Documents

User’s Manual: Hardware
The latest version can be downloaded from the Renesas Electronics website.

Technical Update/Technical News
The latest information can be downloaded from the Renesas Electronics website.

User’s Manual: Development Tools
RX Family C/C++ Compiler CC-RX User's Manual (R20UT3248)
The latest version can be downloaded from the Renesas Electronics website.

Support for Technical Updates

The FIT module reflects the contents of the following technical updates:

None

Website and Support

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

Revision History

Rev.	Date	Description	
		Page	Summary
2.01	Feb. 01, 2019	—	First edition issued

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can 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 must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time 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 time 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 time 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 time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. 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. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance 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 the 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.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is 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. Additionally, 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.
6. 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, for example, 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.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. 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 or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving 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, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, 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 and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall 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. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, 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.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.