RX Family

RAM Soft Error Diagnosis Example

R01AN3361EJ0100
Rev.1.00
Aug 10, 2016

Introduction

This document explains a static RAM soft error diagnosis based on a doubly RAM access method and a bit operation.

Target Device

This example supports the following device.

- RX64M Group

- RX71M Group

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Contents

1. Overview

Recently, a soft error of the static RAM (hereafter, SRAM) has become an un-negligible problem proportional to increasing memory capacity and more detailed process scale. The soft error induces inversing bit values of the SRAM memory and is probabilistic phenomena due to alpha particles included as the impurity of package and neutron beams radiated from the cosmic rays.

This document explains the SRAM soft error diagnosis based on a doubly RAM access method (hereafter, double RAM operation [1]) and a bit operation. The bit operation is executed a user's safety data with random (or sequential, constant) bit pattern stored on the Data Flash Memory. Thereafter, those operated data are written in the double RAM area. The bit operation to which applied is exclusive OR makes balance the number of 0's and 1's [2]. As the result, it can be easy to detect 0 or 1 fixed error and enhanced the error detection rate. The bit pattern itself is also applied to the double RAM method for safety viewpoint when it is updated.

This example is released to the project form using plural firmware integration technology (FIT) modules.

1.1 RAM Soft Error Diagnosis Example

This example is implemented in a project and used as the application example of SRAM soft error diagnosis using plural FIT modules.

1.2 Related documents

[1] Functional safety of electrical/electronic/programmable electronic safety-related systems, IEC61508, Edition 2.0, Apr, 2010

[2] Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices, JESD89A, Oct 2006, JEDEC Solid State Technology Association.

[3] RX Family Board Support Package Module Using Firmware Integration Technology, Rev.3.31, Document No. R01AN1685EJ0331, May 19, 2016

[4] RX Family Flash Module Using Firmware Integration Technology, Rev.1.60, Document No. R01AN2184EU0160, Nov 17, 2015

[5] RX Family Open Source FAT File System [M3S-TFAT-Tiny] Module Firmware Integration Technology, Rev.3.02, Document No. R20AN0038EJ0302, Mar 01, 2015

[6] Renesas USB MCU USB Basic Host and Peripheral Driver Using Firmware Integration Technology, Rev.1.11, Document No. R01AN2025EJ0111, Sep 30, 2015

[7] Renesas USB MCU USB Host Mass Storage Class Driver (HMSC) Using Firmware Integration Technology, Rev.1.11, Document No. R01AN2026EJ0111, Sep 30, 2015

[8] RX64M Group Renesas Starter Kit+ User's Manual For e^2 studio, Rev. 1.10, Document No. R20UT2593EG0110, Jun 25, 2015

[9] RX71M Group Renesas Starter Kit+ User's Manual, Rev. 1.00, Document No. R20UT3217EG0100, Jan 23, 2015

1.3 Hardware Structure

This example uses the RAM and Data Flash Memory peripheral modules of the RX64M/71M. As for the RAM, the without ECC error area is only target of this sample and with ECC area is out of scope.

In detail, please refer to RX64M/71M Group User's Manual: Hardware.

1.4 Software Structure

This sample is operations example of the application and middleware layer using the plural FIT modules whose common initial setting are supplied by a Board Support Package [3]. Figure 1.1 shows the typical structure and functional overview of the software. The application manages the operation sequence composed of the Data Flash access, the RAM without ECC part access, the double RAM operation and USB memory access storage system. The double RAM with bit operation (db_ram.c) creates/update/erases an initial value to create random number (hereafter, seed) and a bit pattern, does the bit operation to user data with bit pattern, does the double RAM operation, reads/writes non safety data, and reads/writes safety data. The flash driver (Flash API) [4] erases/writes the seed and the bit pattern from/to the Data Flash. The FAT file system (M3S-TFAT-Tiny) [5] manages the data[1] in the USB memory as the file using the USB Host driver[2]. The USB Host driver [6], [7] accesses the USB memory as the logical block unit.

[1] Assume the safe data transferred from the USB memory is guaranteed

[2] Diagnosis of the USB Host driver is out of scope

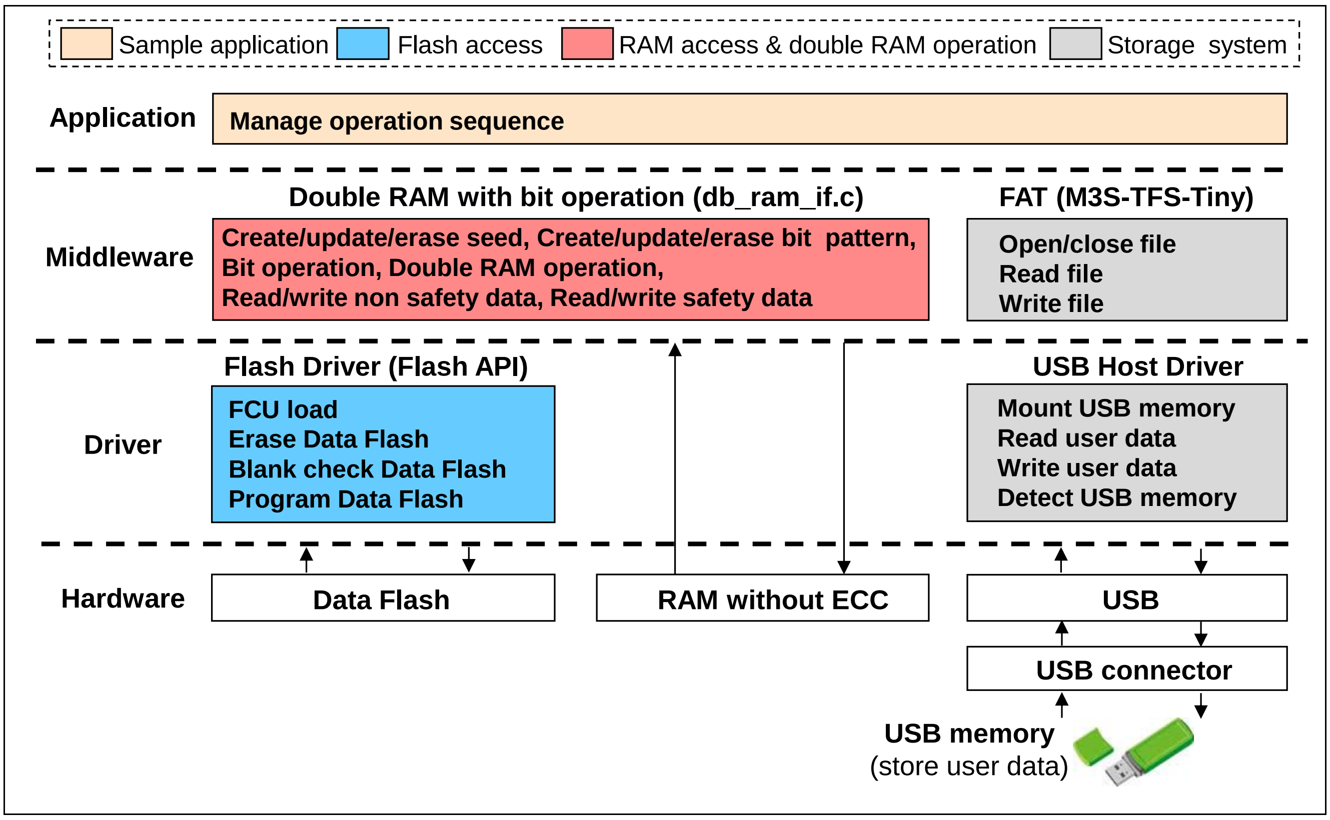


Figure 1.1 Software structure of this sample

1.5 File Structure

This sample codes are stored the “demo_src” and lower hierarchical folders. Figure 1.2 shows the source and header file structures of this sample. The two set samples of non safety and safety data are stored in the USB memory. As for the detailed information of the FIT based modules (BSP, Flash Driver, FAT file system and USB Drivers), please refer to the documentation of the each FIT module.

```
demo_src: main operation
|  sample_main.c
|  sample_main.h
|
+ --- db_ram: Double RAM with bit operation
|      db_ram.c
|      db_ram.h
|
+ --- tfat_if: File system IF to USB driver
|      file_if.c
|      file_if.h
|      r_data_file.c
|      r_data_file.h
|      r_tfat_drv_if.c ;USB driver interface
|
+ --- usb_if: USB Host memory access control
|      r_usb_hmsc_defep.c
|      usb_memory_access.c
|
+ --- usr: LED control
|      led.c
|      led.h
|
+ --- usb_memory_sample: Sample of USB memory data
|      + --- set1, set2 ; two set samples
|           + --- NON: SRC_X.txt ; Non safety data file (X = 1,2,3)
|           + --- SAFE: SRC_X.txt ; Safety data file (X = 0,1,2,3,4,5,6,7)
```

```
r_bsp: BSP (Board Support Package) FIT module

r_config: configuration setting of FIT modules
|      r_bsp_config.h
|      r_bsp_interrupt_config.h
|      r_flash_rx_config.h
|      r_usb_basic_config.h
|      r_usb_hmsc_config.h

r_flash_rx: Flash Driver (Flash API) FIT module

r_tfat_rx: FAT file system (M3S-TFS-Tiny) FIT module
|      r_tfat_lib.h ;FAT library header file
|      + --- lib:  FAT library stored folder
|              r_mw_version.h ; middleware version information
|              r_stdint.h ; integer type definition
|              tfat_rx600_big.lib ; big endian
|              tfat_rx600_little.lib ; little endian
|

r_usb_basic: USB driver (USB basic operation) FIT module

r_usb_hmsc: USB driver (Host Mass Storage Class) FIT module
```

Figure 1.2 File structure of this example

1.6 Outline of Functions

The functions of application layer shows Table 1.1 and the API functions related to the double RAM with bit operation shows Table 1.2.

Table 1.1 Functions of application layer

Item	Contents
main()	Main operation of this project.
led_init()	Initialize user LED.
led_ctrl()	Update user LED pattern.
prm_init()	Initialize operation parameter.
usb_memory_start()	USB memory task start.
Sample_Task()	Sample application task.
file_start()	USB memory start.
file_read()	File reading operation.
file_write()	File writing operation.
file_stop()	File finalizing and operation result save operation.
file_err()	File error retrieval operation.

Table 1.2 API functions (Double RAM with bit operation)

Item	Contents
FlashInit()	Initialize flash driver (flash API).
load_seed()	Load seed from data flash.
upd_seed()	Update seed to data flash.
crt_bitpat()	Create bit pattern data.
load_bitpat()	Load bit pattern from data flash with double RAM operation.
upd_bitpat()	Update bit pattern to data flash with double RAM operation.
ers_bitpat()	Erase bit pattern from data flash.
exec_bitop()	Execute bit operation.
RAM_Init()	Initialize RAM.
RAM_Write()	Write non safety data to RAM.
RAM_SafetyWrite()	Write safety data to RAM.
RAM_Read()	Read non safety data from RAM.
RAM_SafetyRead()	Read safety data from RAM.

2. Functional Information

This example is developed by the following principles.

2.1 Hardware Requirements

This example requires your MCU supports the following feature:

- RAM[1]
- Data Flash
- USB

[1]without ECC error correction only use.

2.2 Hardware Resource Requirements

This section details the hardware peripherals that this example requires. Unless explicitly stated, these resources must be reserved for the following driver, and the user cannot use them.

2.2.1 RAM

This example uses the SRAM without ECC error correction to store non safety data and bit operated safety data.

2.2.2 Data Flash

This example uses the Data Flash to store the seed and the bit pattern.

2.2.3 USB Channel

This example uses the USB 2.0 FS Host/Function Module to read test data from the USB memory and write operation result data to the USB memory.

2.3 Software Requirements

This example depends on the following packages (FIT modules):

- r_bsp
- r_flash_rx
- r_tfat_rx
- r_usb_basic
- r_usb_hmsc

2.4 Limitations

There are following limitations in this example:

- Diagnosis of USB driver is out of scope in this example.
- Assumption the safe data transferred from USB is guaranteed.

2.5 Supported Toolchains

This example is tested and works with the following toolchain:

- Renesas RX Toolchain v2.04.01

2.6 Header Files

Each function call is accessed by including a single file, *sample_main.h, led.h, file_if.h, r_usb_basic_if.h db_ram.h or r_frash_rx_if.h* which is supplied with this project code.

2.7 Integer Types

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

2.8 Configuration Overview

The configuration options in this project are specified in *sample_main.h, db_ram.h and r_data_file.h.* The option names and setting values are listed in the table below.

Configuration options	

`#define NUM_REQ`
`- Default value = 8`
	Specify the total number of safety data record (equal to double RAM operation times).
	- Set 1 to 8 in this example.

`#define MAX_DAT_SIZE`
`- Default value = 16*1024`
	Maximum size of testdata file.
	- Set 16*1024 (=16KB) in this example.

`#define READ_DIR_NON`
`- Default value "NON"`
	String of no safety test data directory.

`#define READ_DIR_SAFE`
`- Default value "SAFE"`
	String of safety test data directory.

`#define READ_FILE`
`- Default value "SRC"`
	The header string of test data file name.
	This string is concatenated the file number assigned access area and the file extension equal to ".txt".
	Ex. "SRC_0.txt", "SRC_1.txt", , "SRC_7.txt"

`#define WRITE_DIR_NCMP`
`- Default value "NCMP"`
	String of no safety operated data directory.

`#define WRITE_DIR_SCMP`
`- Default value "SCMP"`
	String of safety operated data directory.

`#define RESULT_FILE`
`- Default value "RESULT"`
	Specify the file name of the operation result file to indicate safety data file size, raw record area begin address and inverted record area begin address.
	This string is concatenated to the file extension equal to ".txt".
	Ex. "RESULT.txt"

`#define UPD_NXT_SEED`
`- undefined`
	Update seed value for next operation or not?
	- If defined, update the seed value for next operation after current operation completed.

`#define FORCE_ERASE_BPAT`
`- undefined`
	Erase saved bit pattern from the data flash or not?
	- If defined, forcing erase the bit pattern from the data flash after current operation completed.

`#define BIT_PAT_MODE`
`#define BIT_PAT_RAN (0)`
`#define BIT_PAT_SEQ (1)`
`#define BIT_PAT_CNT (2)`
`- Default value = 0`
	Specify the bit pattern setting.
	- When this is set to 0, the bit pattern is random pattern.
	- When this is set to 1, the bit pattern is sequential pattern.
	- When this is set to 2, the bit pattern is constant.

`#define CONST_PAT`
`- Default value = 0x00`
	Set constant data pattern.
	This setting is only relevant, if the bit pattern is constant (="BIT_PAT_CNT").

`#define SEED_BLOCK_ADDR`
`- Default value = 0x00100000 (FLASH_DF_BLOCK_0)`
	Set the seed value stored address.

`#define NUM_BIT_SEG`
`- Default value = 8`
	Set number of bit pattern segment.
	Please set this default value in this version.

`#define BIT_SEG_SIZE`
`- Default value = 1024`
	Set bit pattern segment size.
	Please set this default value in this version.

`#define NUM_NON_BLK`
`- Default value = 3`
	Set number of no safety areas.
	Please set this default value in this version.

`#define NON1_AREA_SIZE`
`- Default value = 16*1024 (16KB)`
	Set size of no safety data area No.1.
	Please set this default value in this version.

`#define NON2_AREA_SIZE`
`- Default value = 16*1024 (16KB)`
	Set size of no safety data area No.2.
	Please set this default value in this version.

`#define NON3_AREA_SIZE`
`- Default value = 16*1024 (16KB)`
	Set size of no safety data area No.3.
	Please set this default value in this version.

`#define NUM_RAW_REC`
`- Default value = 8`
	Set number of raw record for safety data.
	Please set this default value in this version.

Configuration options	

`#define NUM_INV_REC`
`- Default value = 8`
	Set number of invert record for safety data.
	Please set this default value in this version.

`#define RAW_REC_SIZE`
`- Default value = 1024 (1KB)`
	Set size of raw record for safety data.
	Please set this default value in this version.

`#define INV_REC_SIZE`
`- Default value = 1024 (1KB)`
	Set size of invert record for safety data.
	Please set this default value in this version.

`#define KND_BIT_OP`
`#define OP_NONE (0)`
`#define OP_EXOR (1)`
`- Default value = 1 (OP_EXOR)`
	Specify the kind of bit operation.
	- When this is set to 0, no bit operation is executed.
	- When this is set to 1, exclusive or operation is executed.

`#define FILESIZE`
`- Default value = 2048`
	Specify the FAT file system data buffer size
	- Set 2048 in this sample.

2.9 Data Structures

This section details the data structures that are used with the functions of this example. Those data structures in this project are located in *sample_main.h and db_ram.h* as the prototype declaration.

```
/* USB access state */
typedef enum
{
    APL_START = 0, /* Operation start state */
    APL_NREAD,     /* Non safety data read state */
    APL_NRAM,      /* Non safety data RAM access state */
    APL_NWRITE,    /* Non safety data write state */
    APL_SREAD,     /* Safety data read state */
    APL_SRAM,      /* Safety data RAM access state */
    APL_SWRITE,    /* Safety data write state */
    APL_STOP,      /* Operation stop state */
} APLState;
```

```
/* Kind of RAM area */
typedef enum
{
    AREA_NON1 = 0, /* Non safety data area No.1 */
    AREA_NON2,     /* Non safety data area No.2 */
    AREA_NON3,     /* Non safety data area No.3 */
    AREA_SAFE,     /* Safety data area */
} RAMArea;
```

```
/* Data access information structure */
typedef struct
{
    uint16_t size; /* Data size */
    int8_t *src;   /* Address of read data from USB */
    int8_t *dst;   /* Address of write data to USB */
    int8_t area;   /* Kind of RAM area */
} ACCInfo;
```

```
/* Bit pattern segment address */
const flash_block_address_t bp_addr[NUM_BIT_SEG] =
{ /* Refer to "r_flash_rx/src/targets/rx64m/r_flash_rx64m.h or
rx71m/r_flash_rx71m.h". */
    FLASH_DF_BLOCK_384, /* 0x00106000 to 0x001063FF (1KB) */
    FLASH_DF_BLOCK_400, /* 0x00106400 to 0x001067FF (1KB) */
    FLASH_DF_BLOCK_416, /* 0x00106800 to 0x00106BFF (1KB) */
    FLASH_DF_BLOCK_432, /* 0x00106C00 to 0x00106FFF (1KB) */
    FLASH_DF_BLOCK_448, /* 0x00107000 to 0x001073FF (1KB) */
    FLASH_DF_BLOCK_464, /* 0x00107400 to 0x001077FF (1KB) */
    FLASH_DF_BLOCK_480, /* 0x00107800 to 0x00107BFF (1KB) */
    FLASH_DF_BLOCK_496, /* 0x00107C00 to 0x00107FFF (1KB) */
};
```

```
/* Bit pattern for double RAM segment address */
const flash_block_address_t bpc_addr[NUM_BIT_SEG] =
{ /* Refer to "r_flash_rx/src/targets/rx64m/r_flash_rx64m.h or
rx71m/r_flash_rx71m.h". */
    FLASH_DF_BLOCK_512, /* 0x00108000 to 0x001083FF (1KB) */
    FLASH_DF_BLOCK_528, /* 0x00108400 to 0x001087FF (1KB) */
    FLASH_DF_BLOCK_544, /* 0x00108800 to 0x00108BFF (1KB) */
    FLASH_DF_BLOCK_560, /* 0x00108C00 to 0x00108FFF (1KB) */
    FLASH_DF_BLOCK_576, /* 0x00109000 to 0x001093FF (1KB) */
    FLASH_DF_BLOCK_592, /* 0x00109400 to 0x001097FF (1KB) */
    FLASH_DF_BLOCK_608, /* 0x00109800 to 0x00109BFF (1KB) */
    FLASH_DF_BLOCK_624, /* 0x00109C00 to 0x00109FFF (1KB) */
};
```

2.10 Return Values

This section describes return values of the functions of this example. Those return values in the test project are located in *db_ram.h and file_if.h* as the prototype declarations.

```
/* Double RAM operation return value */
typedef enum
{
    DBRAM_ERR_FLERASE = -8,  /* Flash erase error */
    DBRAM_ERR_FLWRITE = -7,  /* Flash write error */
    DBRAM_ERR_FLVERIFY = -6, /* Flash verify error */
    DBRAM_ERR_FLDBRAM = -5,  /* Flash double RAM error */
    DBRAM_ERR_PARAM = -4,    /* Parameter error */
    DBRAM_ERR_BOP = -3,      /* Bit operation error */
    DBRAM_ERR_CMP = -2,      /* Double RAM compare error */
    DBRAM_ERR = -1,         /* General error */
    DBRAM_OK = 0,
} dbram_t;
```

```
/* File access return value */
typedef enum
{
    FLIF_ERR = -1, /* General error */
    FLIF_OK = 0,
} flif_t;
```

3. Specification of This Example

3.1 Environment and Execution

Execution of this example needs a RX64M/71M RSK boards[1] and a USB memory.

The outline of the execution sequence is following.

- Write the project execution code to the code Flash of in the RX64M/71M RSK board (hereafter, RSK board).
- Insert the USB memory to the USB port in the RSK board. The USB memory stores the test data composed of the not safety data and safety ones to/from which wrote/read RAM without ECC. In case of safety data, double Ram with bit operation is applied.
- The test data sample is prepared in the project and it is stored the demo_src/usb_memory_sample/set1 or set2 folder. You can use it copying it to root layer of the USB memory.
- Power on the RSK board.
- When the RSK board finishes the initialization and start process of FAT file system and USB driver, creation to bit pattern data, updating bit pattern with double RAM operation, and loading seed value, the user LED composed of LED3, LED2, LED1 and LED0 shows the "1" pattern (LED3: OFF, LED2: OFF, LED1: OFF, LED0: ON).
- Push the SW1 switch of the RSK board and starts the operation described in the Section 3.2.
- When the double RAM operation of the safety data to each area is completed, the user LED shows the last access record number from "0" to "7". As for the detail of RAM area, please refer to the Section 3.2.
- Push the SW2 switch of the RSK board and starts the double RAM operation to next area.
- When the operation finished without any error, the user LED shows the all-on pattern (LED3: ON, LED2: ON, LED1: ON, LED0: ON).
- If any error detected during the operation, the user LED shows the following pattern depend on the error.

  Flash access error – LED3: ON, LED2: OFF, LED1: OFF, LED0: OFF. ("8" pattern)

  Safety access error – LED3: ON, LED2: OFF, LED1: OFF, LED0: ON. ("9" pattern)

  Not safety access error – LED3: ON, LED2: OFF, LED1: ON, LED0: OFF. ("A" pattern)

  Seed creation error – LED3: ON, LED2: OFF, LED1: ON, LED0: ON. ("B" pattern)

  Bit pattern erasing error – LED3: ON, LED2: ON, LED1: OFF, LED0: OFF. ("C" pattern)

[1] Product name is a Renesas Starter Kit+ for RX64M [8] or a Renesas Starter Kit+ for RX71M [9].

Figure 3.1 shows the environments of this example.

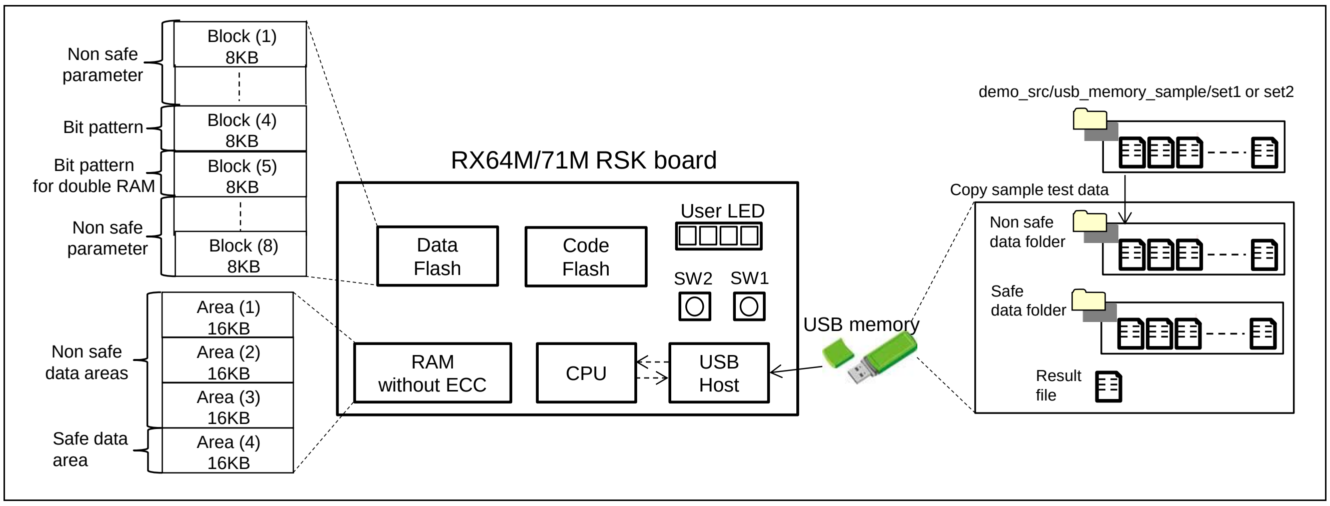
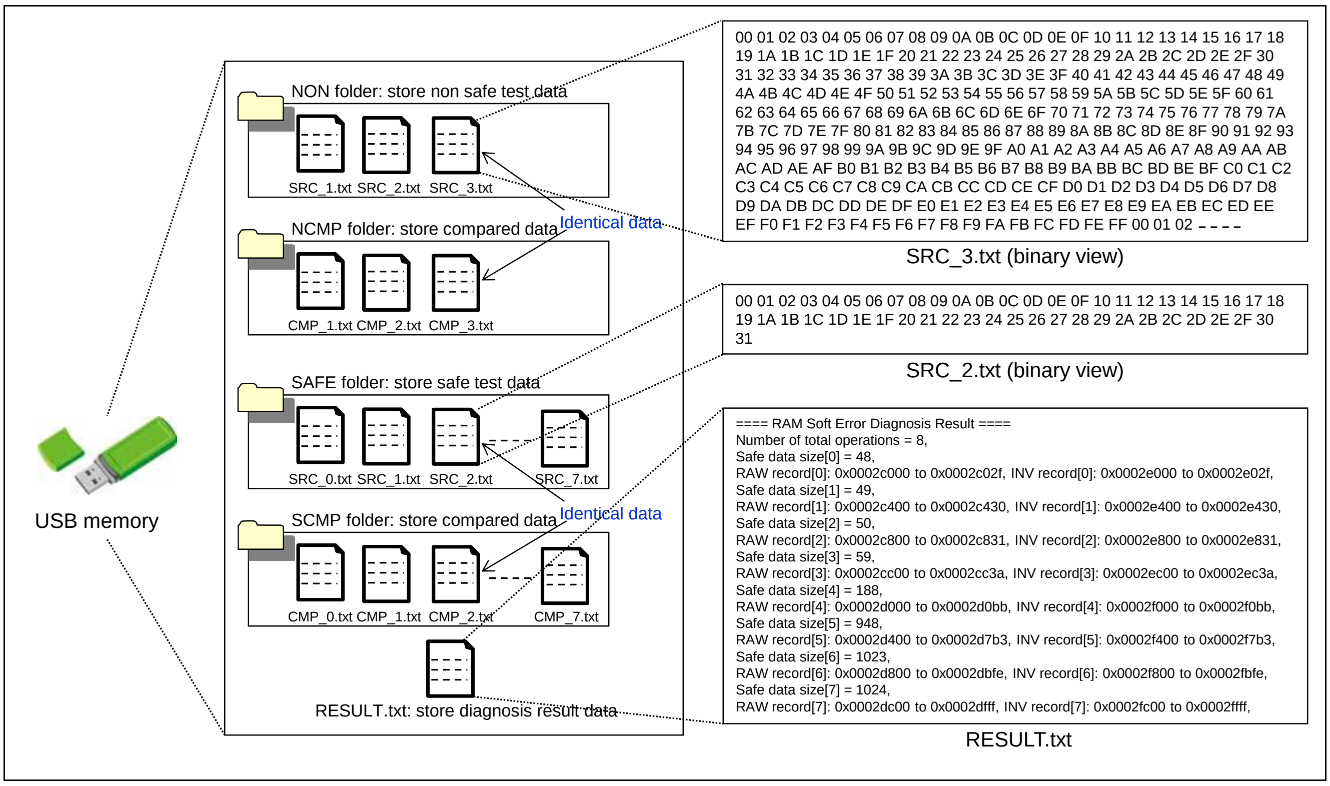


Figure 3.1 Environment

3.2 Operation Sequence

In this section, explain the operation sequence in this example when the total number of safety data record are 8 specified to “NUM_REC”.

Figure 3.2 shows the USB memory contents. There are NON folder, NCMP folder, SAFE folder, SCMP folder, and RESULT.txt file in the USB memory. The NON folder stores the non safe test data in the files from “SRC_1.txt” to “SRC_3.txt”. The NCMP folder stores the compared data after the normal read/write operation in the files from “CMP_1.txt” to “CMP_3.txt”. The SAFE folder stores the safe test data in the files from “SRC_0.txt” to “SRC_7.txt”. The SCMP folder stores the compared data after the double RAM operation in the files from “CMP_0.txt” to “CMP_7.txt”. The contents of SAFE (NON) folder and SCMP (NCMP) folder are identical each other if operation finished without error. Figure 3.2 also shows the contents of SRC_3.txt of the non safe test data and SRC_2.txt of the safe test data respectively as example. The RESULT.txt stores the number of the total test times, sizes of the safe test data, and the addresses of raw and invert records of the safe test data.


NON folder: store non safe test data
SRC_1.txt SRC_2.txt SRC_3.txt
Identical data
NCMP folder: store compared data
CMP_1.txt CMP_2.txt CMP_3.txt
SAFE folder: store safe test data
SRC_0.txt SRC_1.txt SRC_2.txt SRC_7.txt
Identical data
SCMP folder: store compared data
CMP_0.txt CMP_1.txt CMP_2.txt CMP_7.txt
RESULT.txt: store diagnosis result data
USB memory

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18
19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30
31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49
4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61
62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A
7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93
94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB
AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2
C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8
D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE
EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 00 01 02 – – – –
```

SRC_3.txt (binary view)

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18
19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30
31
```

SRC_2.txt (binary view)

```
==== RAM Soft Error Diagnosis Result ====
Number of total operations = 8,
Safe data size[0] = 48,
RAW record[0]: 0x0002c000 to 0x0002c02f, INV record[0]: 0x0002e000 to 0x0002e02f,
Safe data size[1] = 49,
RAW record[1]: 0x0002c400 to 0x0002c430, INV record[1]: 0x0002e400 to 0x0002e430,
Safe data size[2] = 50,
RAW record[2]: 0x0002c800 to 0x0002c831, INV record[2]: 0x0002e800 to 0x0002e831,
Safe data size[3] = 59,
RAW record[3]: 0x0002cc00 to 0x0002cc3a, INV record[3]: 0x0002ec00 to 0x0002ec3a,
Safe data size[4] = 188,
RAW record[4]: 0x0002d000 to 0x0002d0bb, INV record[4]: 0x0002f000 to 0x0002f0bb,
Safe data size[5] = 948,
RAW record[5]: 0x0002d400 to 0x0002d7b3, INV record[5]: 0x0002f400 to 0x0002f7b3,
Safe data size[6] = 1023,
RAW record[6]: 0x0002d800 to 0x0002dbfe, INV record[6]: 0x0002f800 to 0x0002fbfe,
Safe data size[7] = 1024,
RAW record[7]: 0x0002dc00 to 0x0002dfff, INV record[7]: 0x0002fc00 to 0x0002ffff,
```

RESULT.txt

Figure 3.2 USB memory Contents

This example uses the 16KB data flash area and 64KB RAM without ECC area. The erase unit and program unit of RX64M/71M's data flash are 64B and 4B respectively[1].

A portion of the data flash divided by eight defines the block (Block(1) to Block(8)). Furthermore, each 1KB unit of the block defines the segment (Segment(0) to Segment(7)). Allocating one block as the bit pattern stored area, the other block as the bit pattern for double RAM stored area, and remaining blocks are used as the not safety parameter stored areas.

The last 16KB RAM area, defining Area(4), assigns the safe data stored area and remaining 3x16KB areas uses the not safety data stored areas (Area(1) to Area(4)). Forward 8KB of the Area(4) allocates a raw data area (Raw record(0) to Raw record(7)) and backward 8KB uses a bit invert area (Invert record(0) to Invert record(7)). Each 1KB unit of them defines a raw record or invert record.

Figure 3.3 explains the operation overview in relation to the data allocation when 1byte safety data (hereafter, Data A) write after read operation. And then, Data A is stored at Nth byte of 7th raw record and exclusive OR operation (hereafter, XOR) is applied. (1 <= N <= 1024)

(1) Update bit pattern in the data flash. Thereafter, verify updated bit pattern applied to double RAM operation.

(2) Execute XOR Data A with Bit pattern(6,N). This operated data defines Data A'.

(3) Write Data A' to Nth byte of 7th raw record.

(4) Execute bit invert operation to Data A'. This operated data defines Data A".

(5) Write Data A" to (1024 – N + 1)th byte of 2nd invert record.

(6) Read Data A" from (1024 – N + 1)th byte of 2nd invert record.

(7) Execute bit invert operation to Data A''.

(8) Read Data A' to Nth byte of 7th raw record.

(9) Compare the bit invert Data A" at sequence(7) to the Read Data A' at sequence(8). If not compare match each other, any RAM soft error detected.

(10) Execute XOR Data A' with Bit pattern(6,N). This operated data retrieves Data A due to XOR feature. And then, return Data A to user application.

[1] As for more detailed feature, please refer to RX64M/71M user's manual.

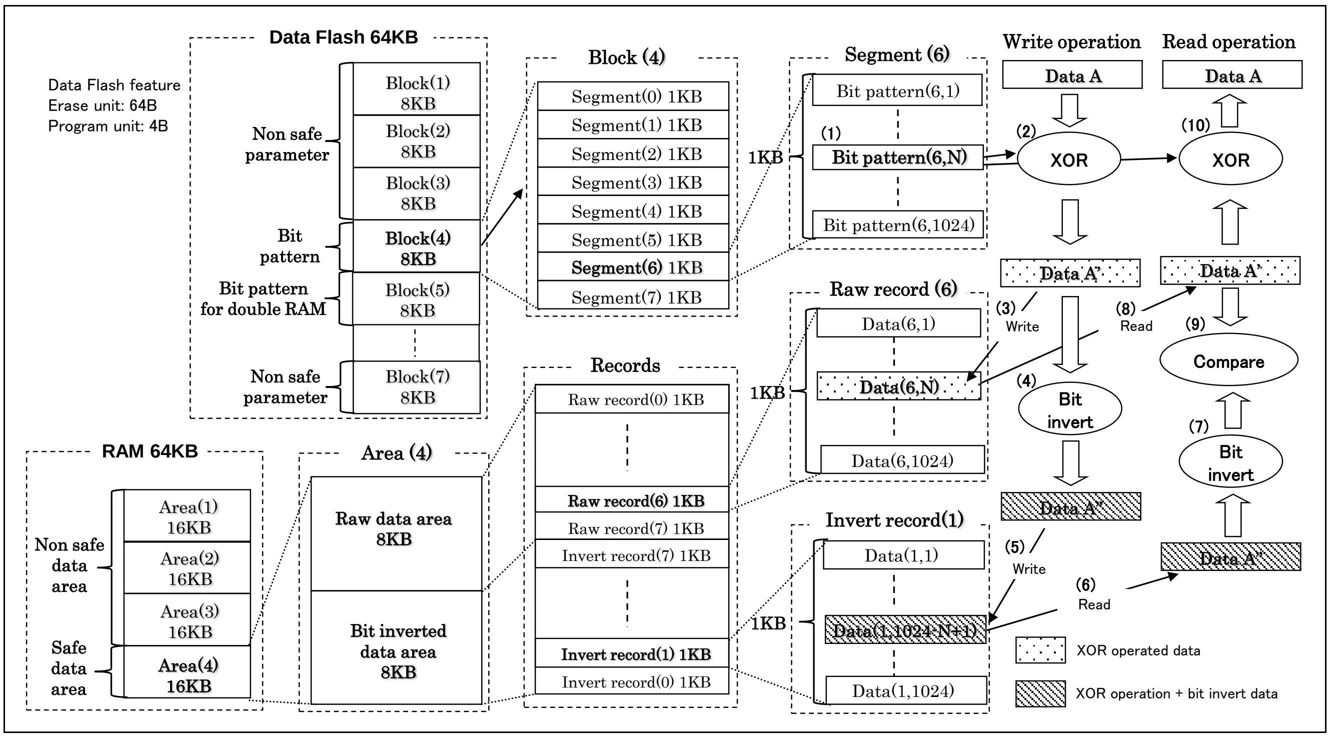


Figure 3.3 Operation overview and data allocation

3.3 Reducing Biased Data

In general, the safe data of user application are not balanced (biased 0's or 1's) and difficult to detect RAM's 0 or 1 fixed error effectively. Therefore, XOR make equivalent data pattern of safe data stored in the RAM and enhances the error detection rate.

Figure 3.4 shows the example of reducing user application's biased data by XOR and the operation is same as described Figure 3.3. Store the sequential bit pattern such as 0x00 to 0xFF by 256 byte unit in the Segment(6). In this case, both of P_N and P_I closes nearly 1/2 because the number of b'0 and b'1 are 4096 unit respectively and each safe data is applied to XOR with the bit pattern (P_N: probability preserving the current bit value, P_I: the probability inverting the current bit value). If the number of b'0 and b'1 in the user application data are N unit and (8192 – N) unit respectively, XOR makes the number of b'0 or b'1 approximately becomes 4092 unit and reduces the biased data in view of probabilistically (N = 0, 1, 2,,, 8192).

The error detection rate improves proportional to the data size of user application because the more data size, P_N and P_I become closer to 1/2.

If bit pattern is created from random data, the same degree of result can be expected because the number of b'0 and b'1 are approximately equivalent.

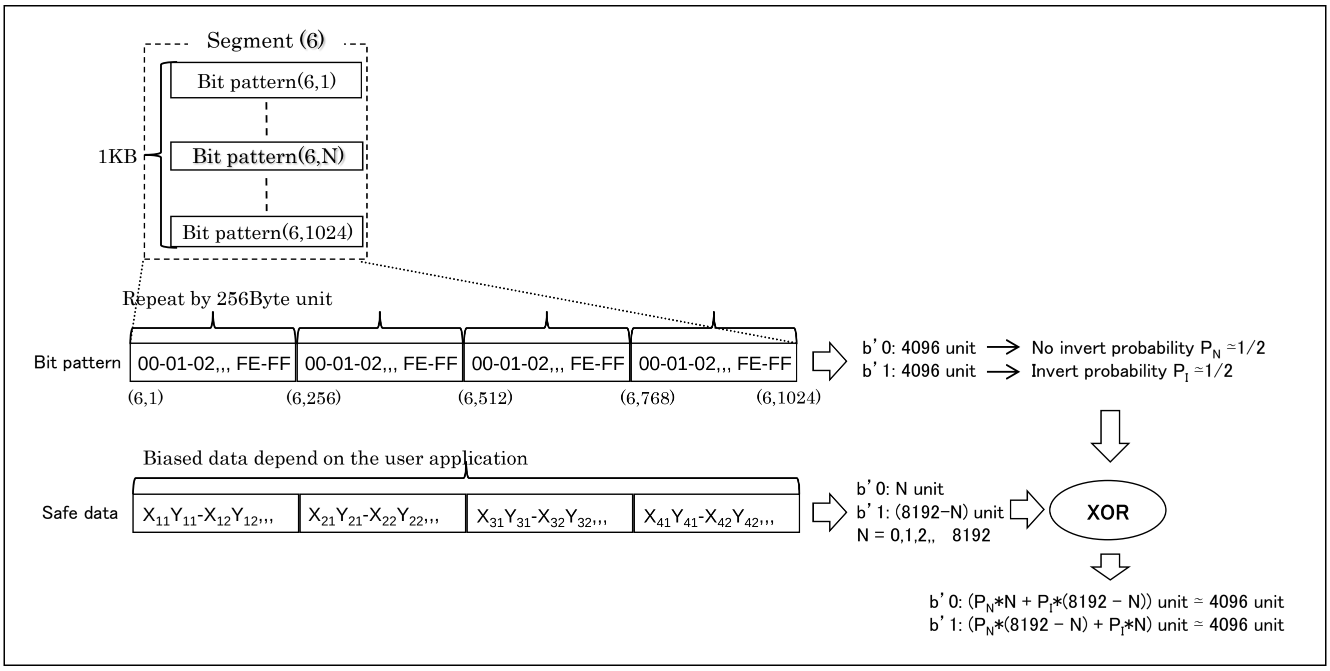

Segment (6)
Bit pattern(6,1)
Bit pattern(6,N)
Bit pattern(6,1024)
1KB
Repeat by 256Byte unit
Bit pattern
00-01-02,,, FE-FF
00-01-02,,, FE-FF
00-01-02,,, FE-FF
00-01-02,,, FE-FF
(6,1)
(6,256)
(6,512)
(6,768)
(6,1024)
b' 0: 4096 unit → No invert probability P_N ≈1/2
b' 1: 4096 unit → Invert probability P_I ≈1/2
Biased data depend on the user application
Safe data
$X_{11}Y_{11}$-$X_{12}Y_{12}$,,,
$X_{21}Y_{21}$-$X_{22}Y_{22}$,,,
$X_{31}Y_{31}$-$X_{32}Y_{32}$,,,
$X_{41}Y_{41}$-$X_{42}Y_{42}$,,,
b' 0: N unit
b' 1: (8192−N) unit
N = 0,1,2,, 8192
XOR
b' 0: (P_N*N + P_I*(8192 − N)) unit ≈ 4096 unit
b' 1: (P_N*(8192 − N) + P_I*N) unit ≈ 4096 unit

Figure 3.4 Reducing biased data by XOR

3.4 Performance of Safety Data Access (Measurement Example)

Table 3.1 and Table 3.2 show the access time of RX64M and RX71M respectively as the typical examples. Those access time are measured by the execution interval of safety write function (RAM_SafetyWrite) or safety read function (RAM_SafetyRead) with bit operation and those safety data are used the safety data set[1] stored in the SAFE folder.

Please keep your mind those result are depend on the condition and environment.

[1] Stored at demo_src/usb_memory_sample/set1 and set2.

Table 3.1 RX64M access time

- RX64M Measurement conditions

Board	CPU frequency	Code area	Data area	Compiler	Optimization setting
RX64M RSK board (R0K50564MC001BR)	120MHz	Internal ROM Read: 1cycle	Internal RAM Read/Write: 1cycle	CC-RX V2.04.01	Default setting - Opt level: 2 - Opt method: size priority - Inter module opt: none

- set1 with bit operation

Data size (Byte)	48	49	50	59	188	948	1023	1024
Write (ns)	22,150	22,616	23,083	27,141	85,008	426,216	459,941	460,333
Read (ns)	22,891	23,375	23,858	28,066	87,941	441,025	475,916	476,325

- set2 with bit operation

Data size (Byte)	150	1024	256	127	991	992	723	27
Write (ns)	67,983	460,333	115,533	57,675	445,575	445,966	325,250	12,775
Read (ns)	70,316	476,325	119,533	59,658	461,050	461,458	336,541	13,200

Table 3.2 RX71M access time

- RX71M Measurement conditions

Board	CPU frequency	Code area	Data area	Compiler	Optimization setting
RX64M RSK board (R0K50571MC000BR)	240MHz	Internal ROM Read: 2cycles	Internal RAM Read/Write: 1cycle	CC-RX V2.04.01	Default setting - Opt level: 2 - Opt method: size priority - Inter module opt: none

- set1 with bit operation

Data size (Byte)	48	49	50	59	188	948	1023	1024
Write (ns)	17,983	18,350	18,733	22,033	69,250	347,516	375,000	375,333
Read (ns)	18,316	18,708	19,083	22,458	70,666	354,875	382,950	383,300

- set2 with bit operation

Data size (Byte)	150	1024	256	127	991	992	723	27
Write (ns)	55,350	375,333	94,141	46,933	363,283	363,625	265,150	10,316
Read (ns)	56,483	383,300	96,108	47,891	370,983	371,333	270,766	10,491

3.5 Software Operation Flow

In this section, describes the software operation flow of this sample.

Figure 3.5 shows from the initialize process to the infinite loop. This sample task (Sample_Task function) is called via USB application task showed by Figure 3.6 within the infinite loop. Figure 3.7 and Figure 3.8 show the each task structure managed by the application state. Figure 3.9 to Figure 3.20 are software operation flows related to the double RAM with bit operation.

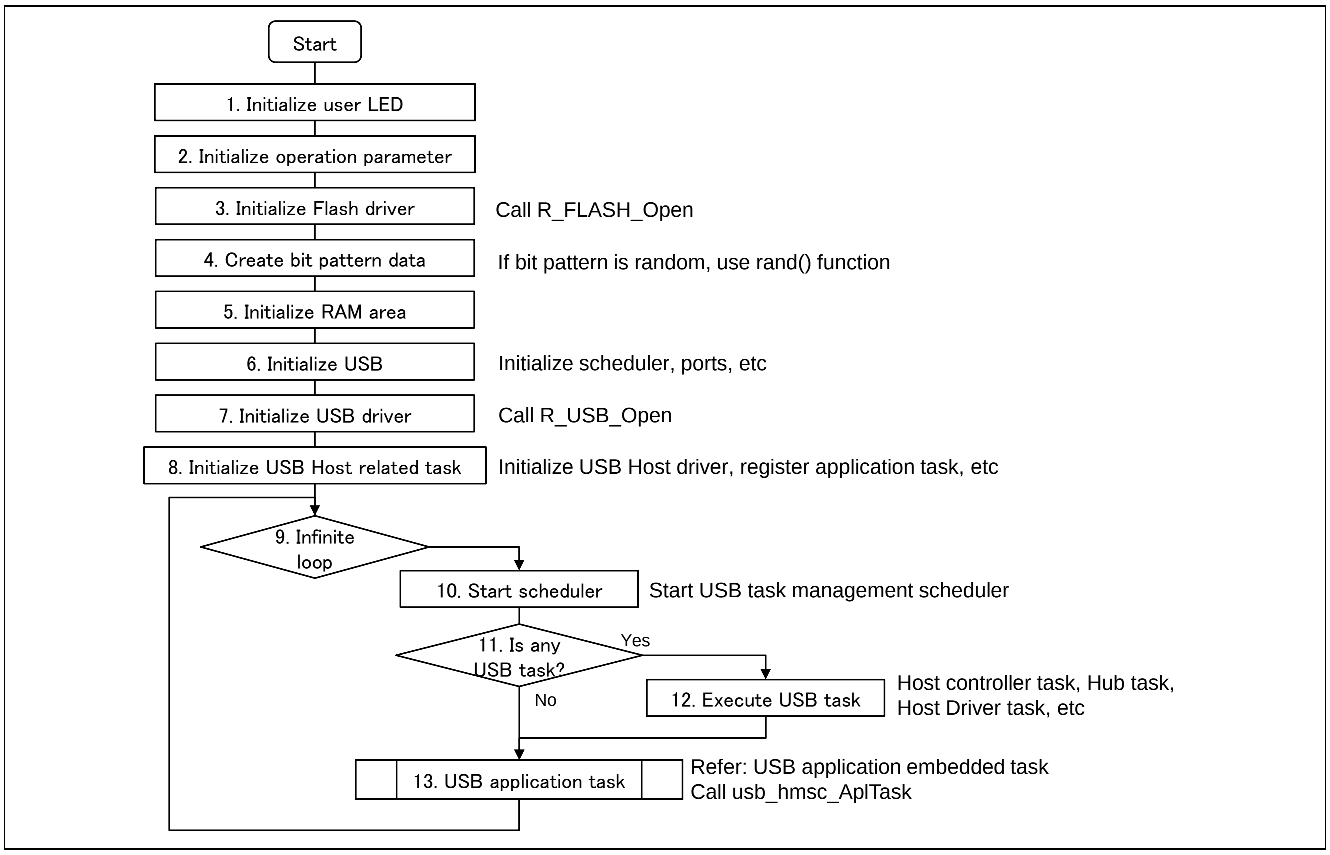

Start
1. Initialize user LED
2. Initialize operation parameter
3. Initialize Flash driver
Call R_FLASH_Open
4. Create bit pattern data
If bit pattern is random, use rand() function
5. Initialize RAM area
6. Initialize USB
Initialize scheduler, ports, etc
7. Initialize USB driver
Call R_USB_Open
8. Initialize USB Host related task
Initialize USB Host driver, register application task, etc
9. Infinite loop
10. Start scheduler
Start USB task management scheduler
11. Is any USB task?
Yes
No
12. Execute USB task
Host controller task, Hub task, Host Driver task, etc
13. USB application task
Refer: USB application embedded task Call usb_hmsc_AplTask

Figure 3.5 Initial operation

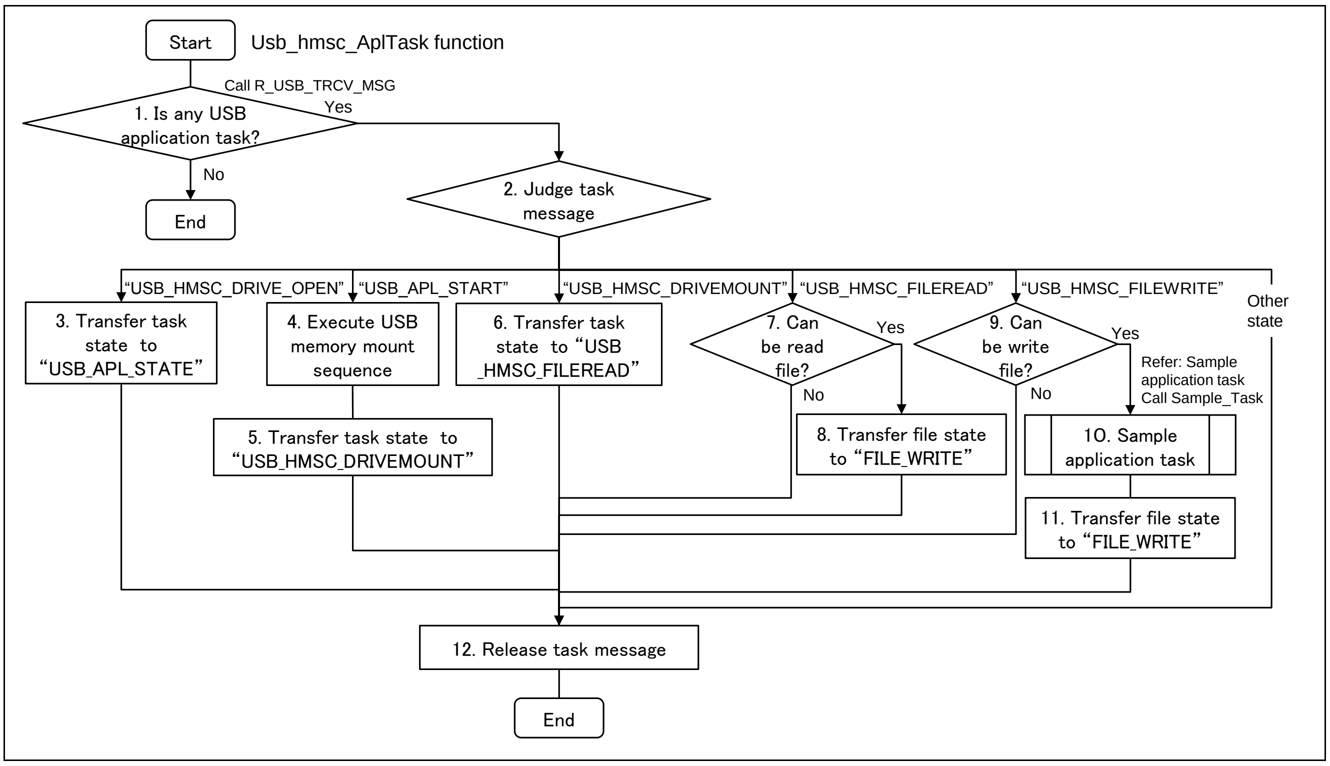

Start
Usb_hmsc_AplTask function
Call R_USB_TRCV_MSG
1. Is any USB application task?
Yes
No
End
2. Judge task message
"USB_HMSC_DRIVE_OPEN"
"USB_APL_START"
"USB_HMSC_DRIVEMOUNT"
"USB_HMSC_FILEREAD"
"USB_HMSC_FILEWRITE"
Other state
3. Transfer task state to "USB_APL_STATE"
4. Execute USB memory mount sequence
6. Transfer task state to "USB_HMSC_FILEREAD"
7. Can be read file?
Yes
No
9. Can be write file?
Yes
No
Refer: Sample application task
Call Sample_Task
5. Transfer task state to "USB_HMSC_DRIVEMOUNT"
8. Transfer file state to "FILE_WRITE"
10. Sample application task
11. Transfer file state to "FILE_WRITE"
12. Release task message
End

Figure 3.6 USB application embedded task

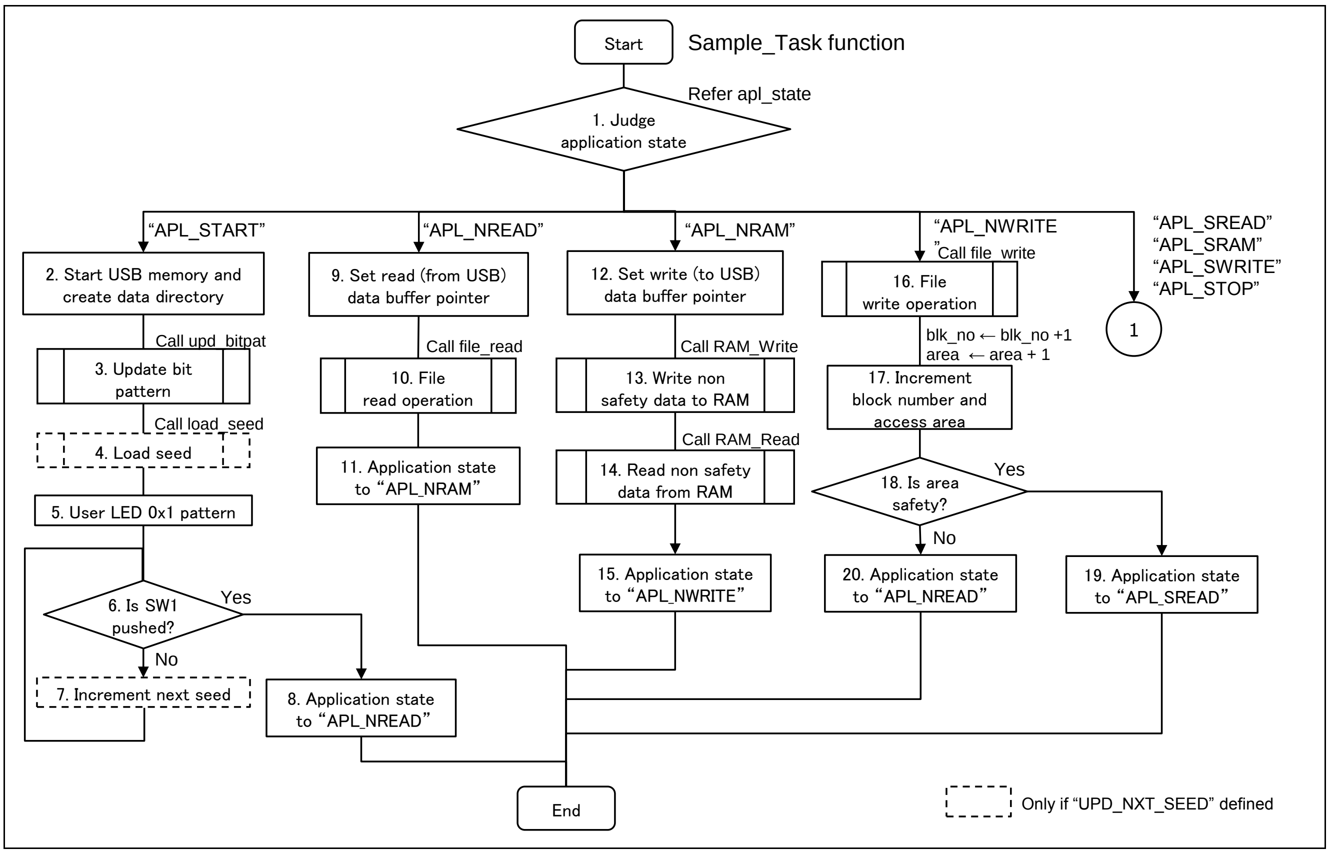

Start
Sample_Task function
Refer apl_state
1. Judge application state
"APL_START"
"APL_NREAD"
"APL_NRAM"
"APL_NWRITE"
"APL_SREAD"
"APL_SRAM"
"APL_SWRITE"
"APL_STOP"
2. Start USB memory and create data directory
Call upd_bitpat
3. Update bit pattern
Call load_seed
4. Load seed
5. User LED 0x1 pattern
6. Is SW1 pushed?
Yes
No
7. Increment next seed
8. Application state to "APL_NREAD"
9. Set read (from USB) data buffer pointer
Call file_read
10. File read operation
11. Application state to "APL_NRAM"
12. Set write (to USB) data buffer pointer
Call RAM_Write
13. Write non safety data to RAM
Call RAM_Read
14. Read non safety data from RAM
15. Application state to "APL_NWRITE"
Call file_write
16. File write operation
blk_no ← blk_no +1
area ← area + 1
17. Increment block number and access area
18. Is area safety?
Yes
No
20. Application state to "APL_NREAD"
19. Application state to "APL_SREAD"
1
End
Only if "UPD_NXT_SEED" defined

Figure 3.7 Sample application task (1)

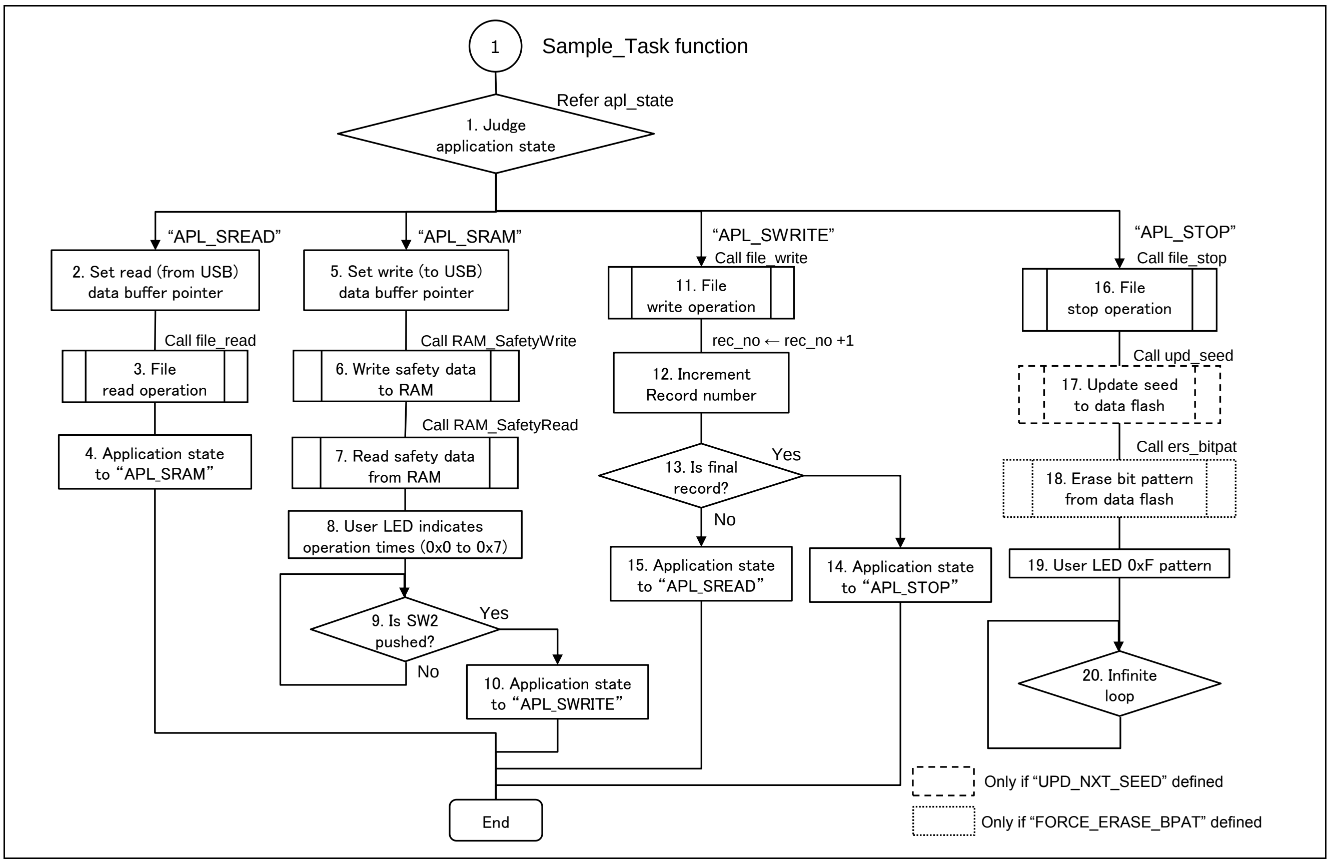

1
Sample_Task function
Refer apl_state
1. Judge application state
"APL_SREAD"
"APL_SRAM"
"APL_SWRITE"
"APL_STOP"
2. Set read (from USB) data buffer pointer
Call file_read
3. File read operation
4. Application state to "APL_SRAM"
5. Set write (to USB) data buffer pointer
Call RAM_SafetyWrite
6. Write safety data to RAM
Call RAM_SafetyRead
7. Read safety data from RAM
8. User LED indicates operation times (0x0 to 0x7)
9. Is SW2 pushed?
Yes
No
10. Application state to "APL_SWRITE"
Call file_write
11. File write operation
rec_no ← rec_no +1
12. Increment Record number
13. Is final record?
Yes
No
15. Application state to "APL_SREAD"
14. Application state to "APL_STOP"
Call file_stop
16. File stop operation
Call upd_seed
17. Update seed to data flash
Call ers_bitpat
18. Erase bit pattern from data flash
19. User LED 0xF pattern
20. Infinite loop
End
Only if "UPD_NXT_SEED" defined
Only if "FORCE_ERASE_BPAT" defined

Figure 3.8 Sample application task (2)

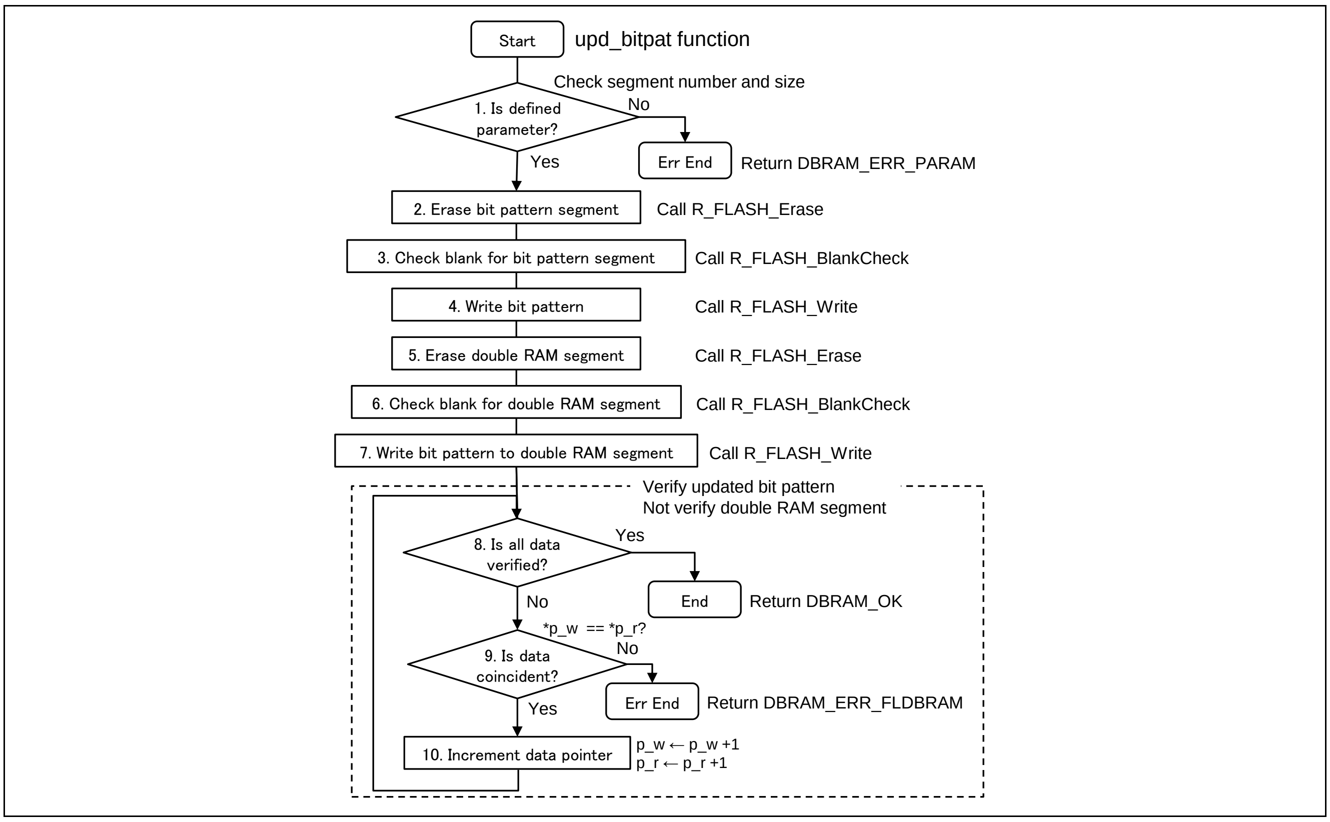


Figure 3.9 Update bit pattern operation

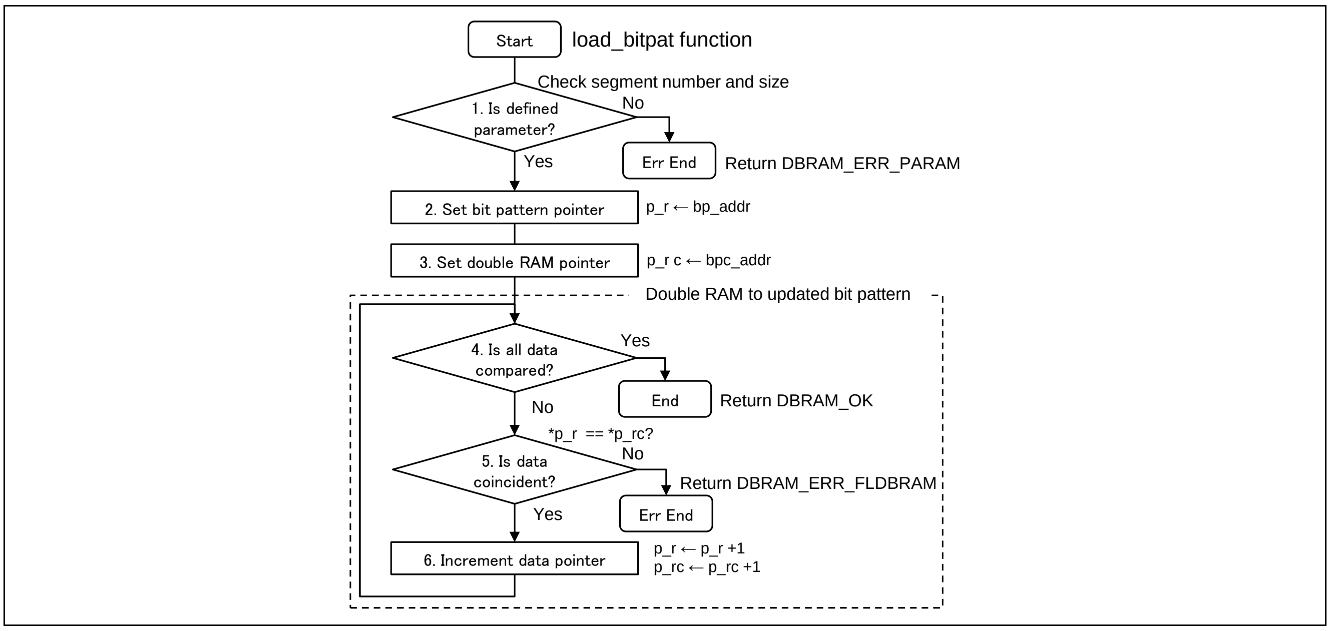


Figure 3.10 Load bit pattern operation

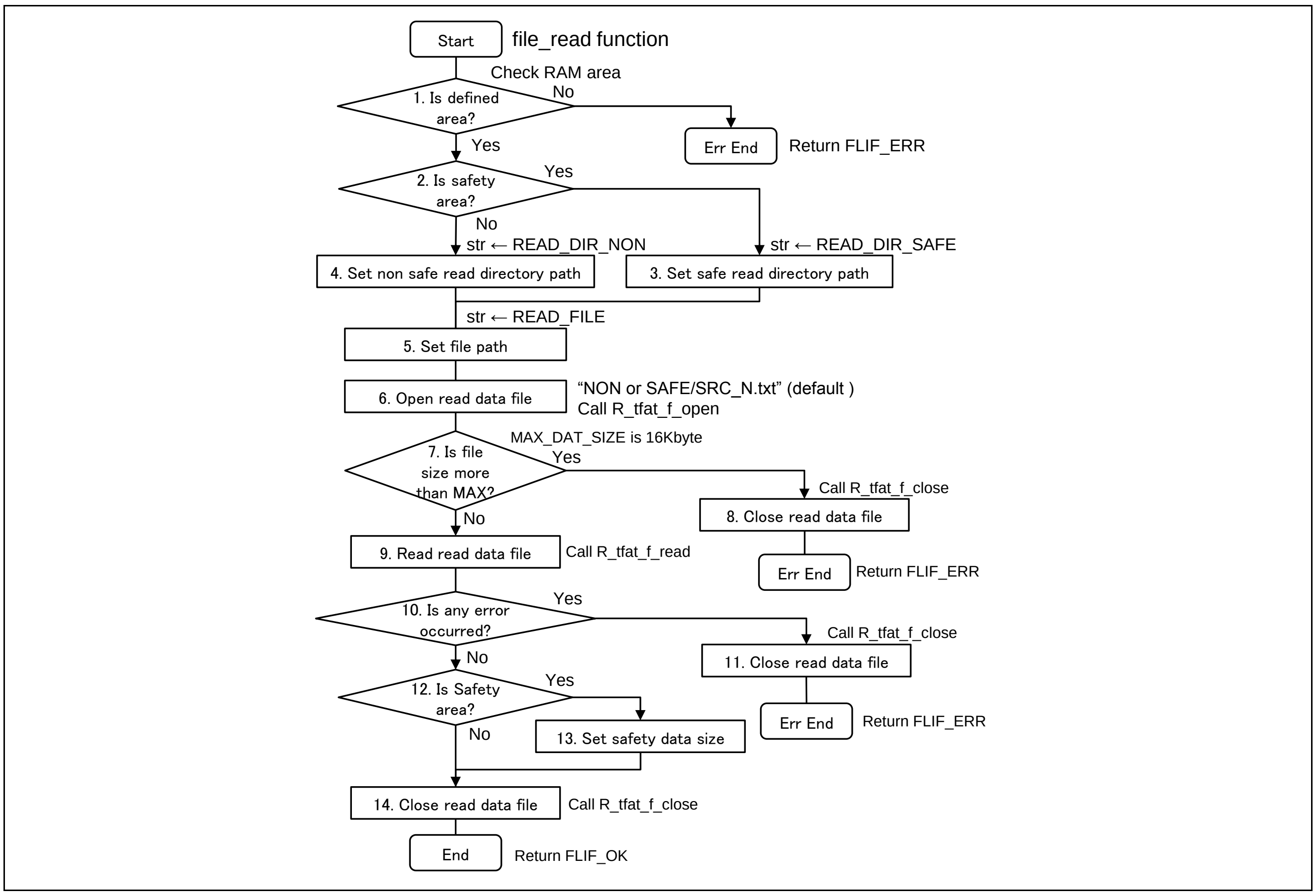


Figure 3.11 File read operation

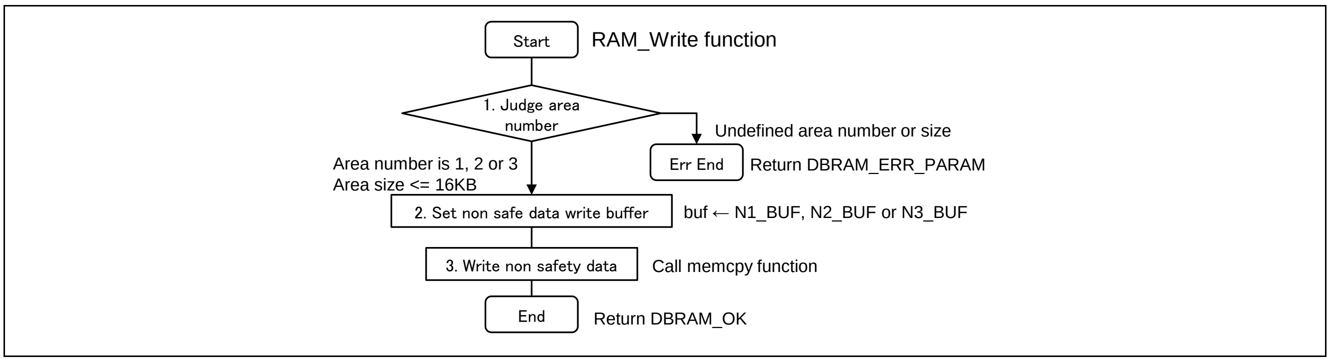


Figure 3.12 Write non safety data operation

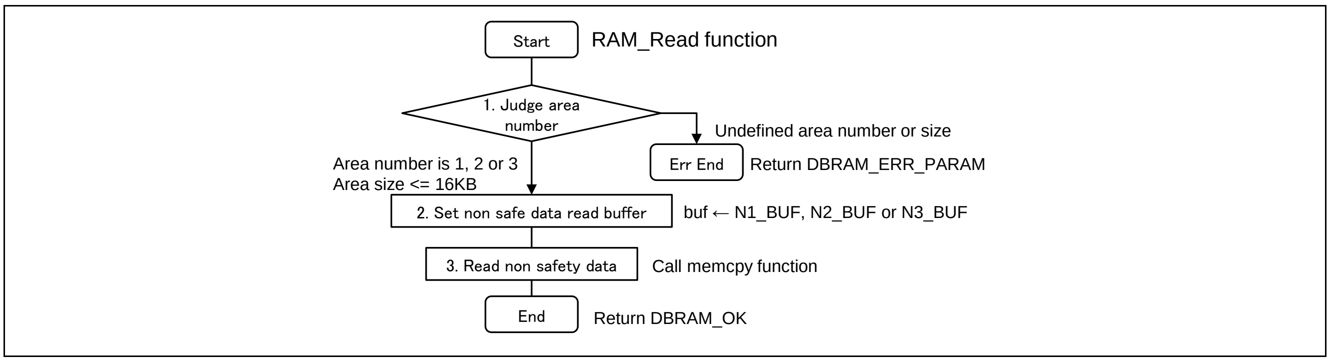


Figure 3.13 Read non safety data operation

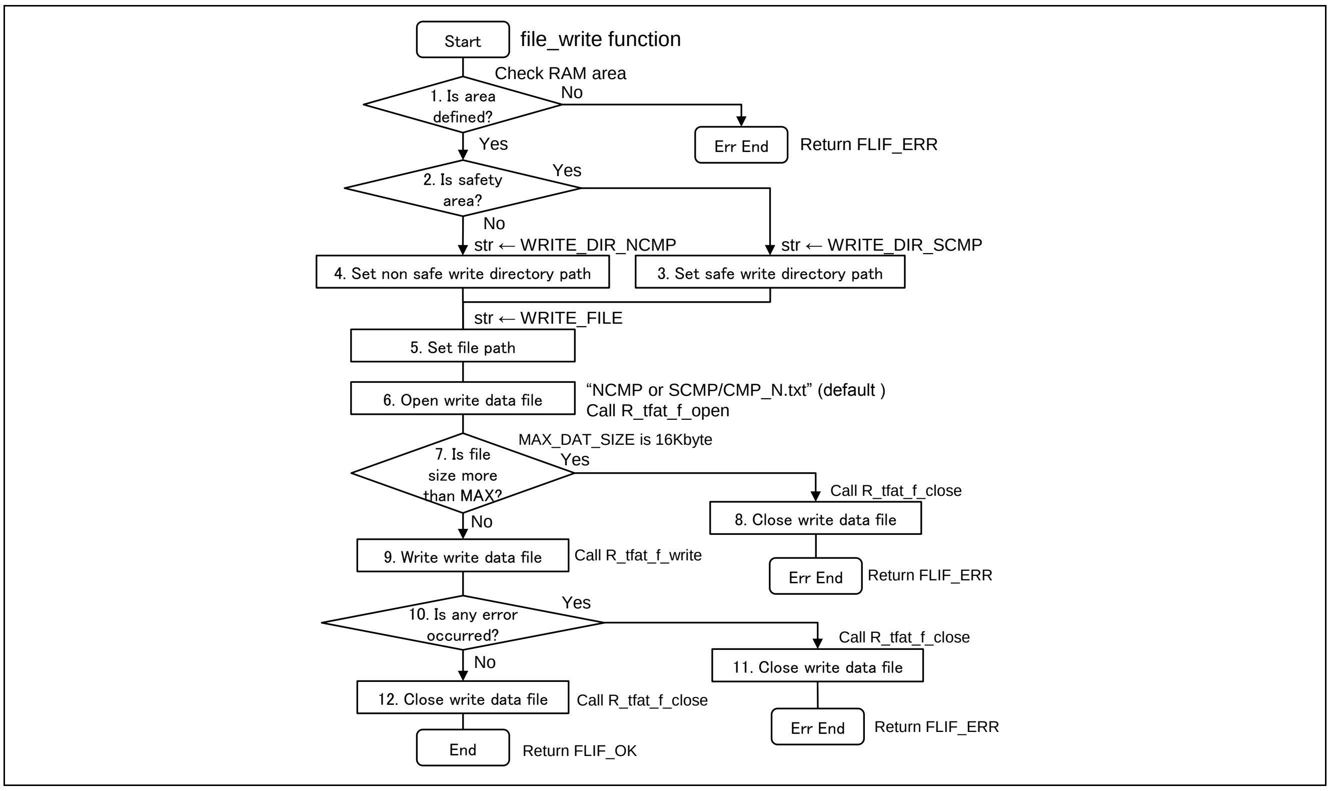


Figure 3.14 File write operation

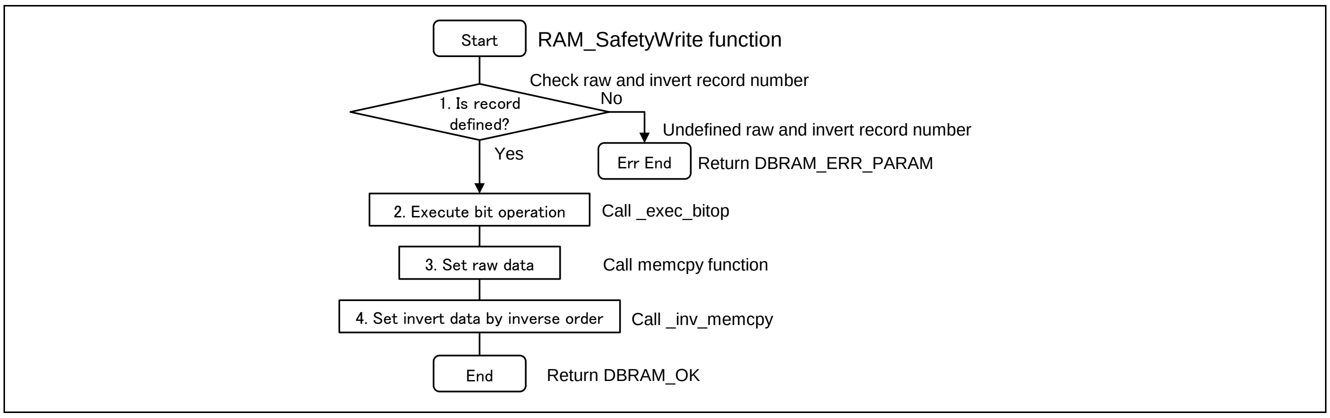


Figure 3.15 Write safety data operation

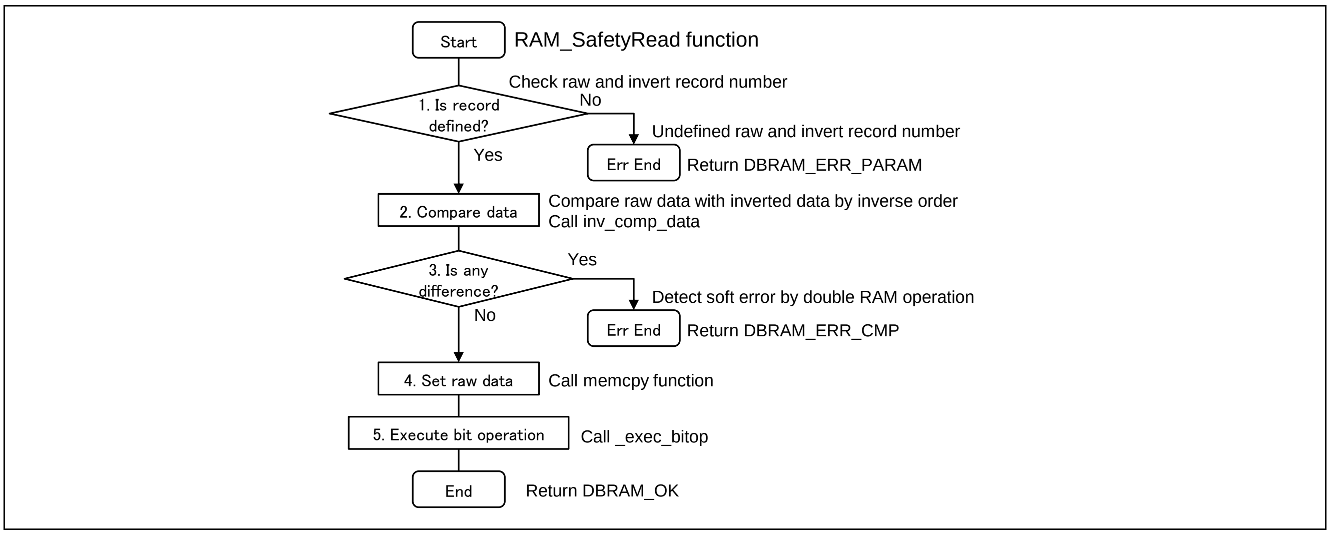


Figure 3.16 Read safety data operation

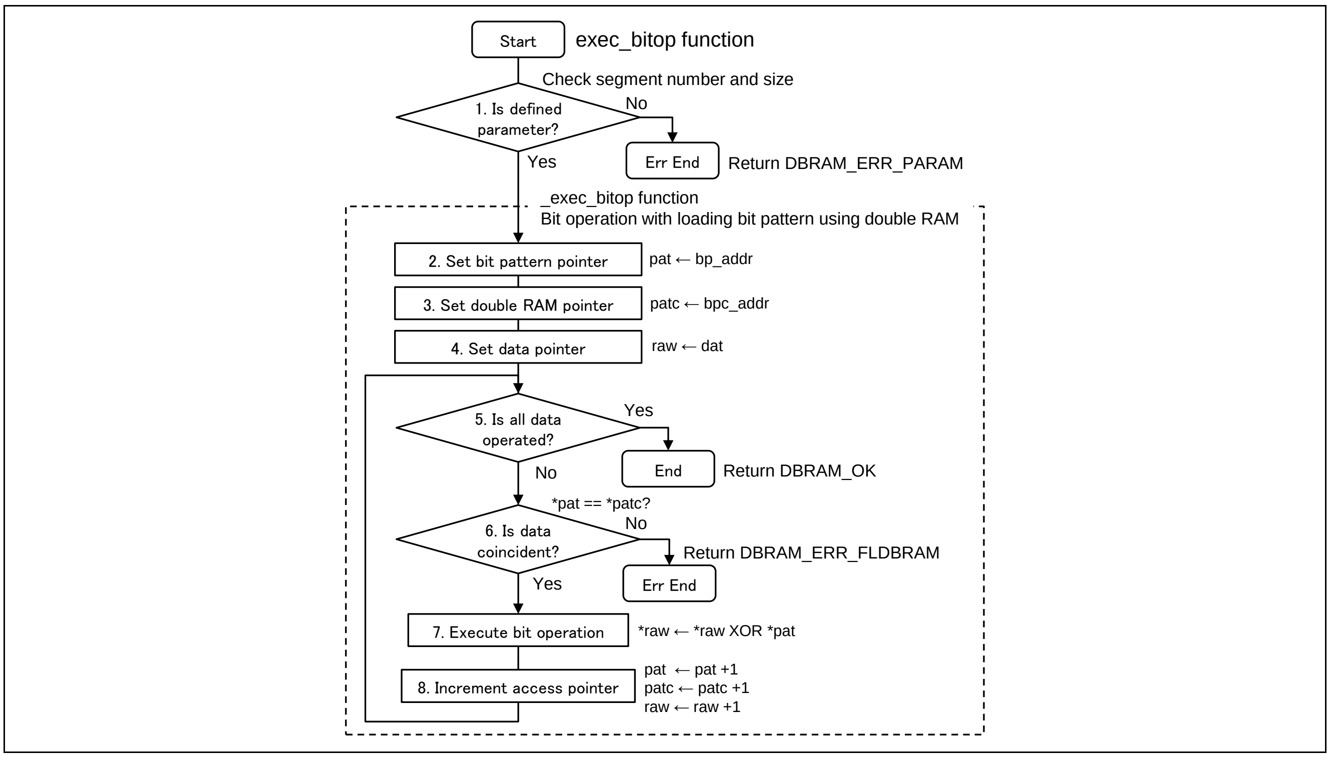


Figure 3.17 Execute bit operation

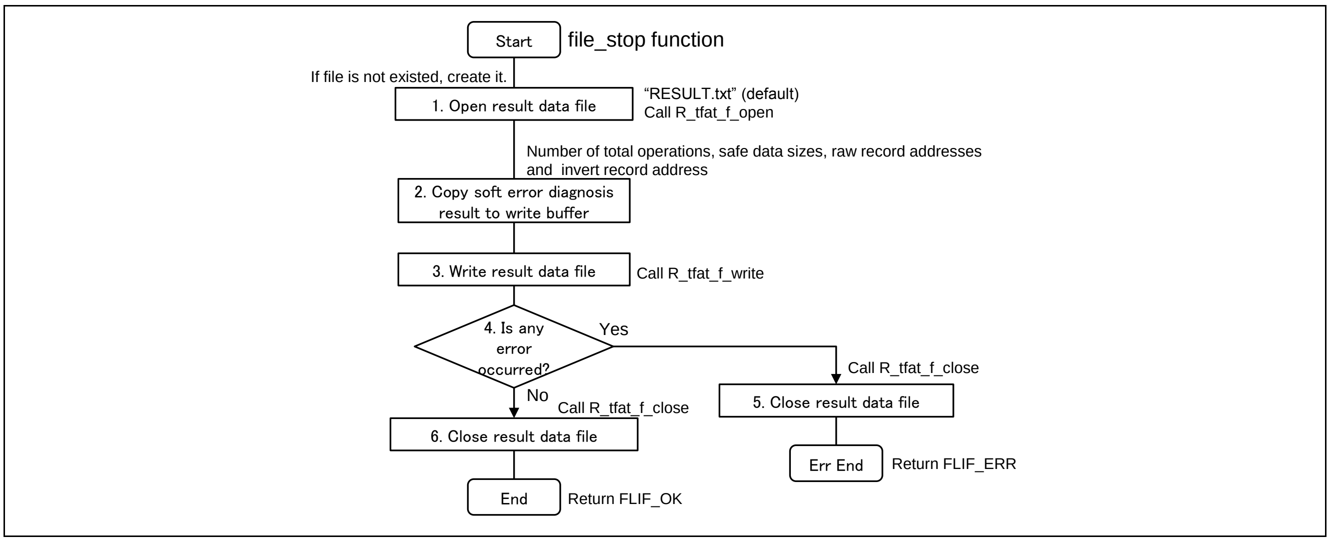

Start
file_stop function
If file is not existed, create it.
1. Open result data file
"RESULT.txt" (default)
Call R_tfat_f_open
Number of total operations, safe data sizes, raw record addresses and invert record address
2. Copy soft error diagnosis result to write buffer
3. Write result data file
Call R_tfat_f_write
4. Is any error occurred?
Yes
Call R_tfat_f_close
5. Close result data file
Err End
Return FLIF_ERR
No
Call R_tfat_f_close
6. Close result data file
End
Return FLIF_OK

Figure 3.18 File stop operation

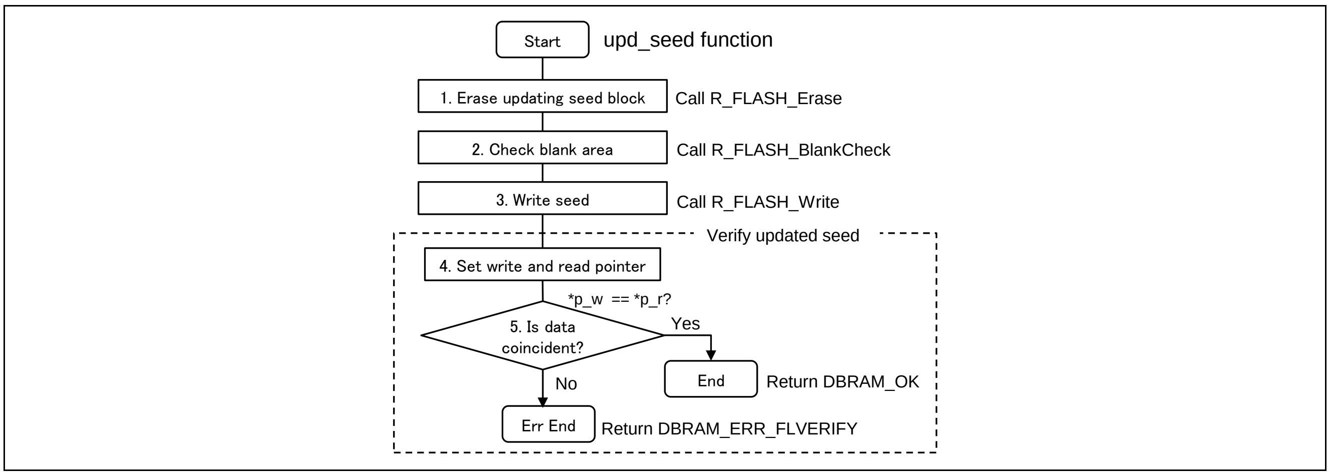


Figure 3.19 Update seed operation

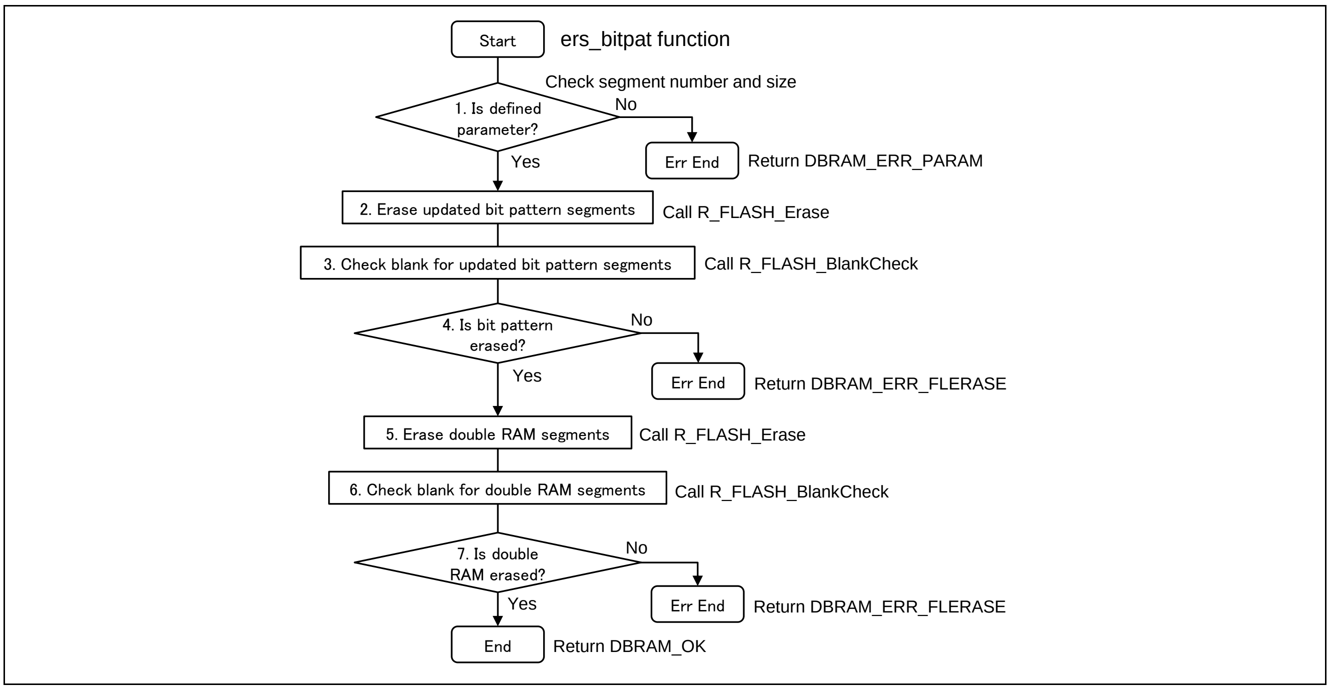


Figure 3.20 Erase bit pattern operation

3.6 Board Setting

There are two jumpers changing from the default setting of the RX64M/71M RSK board to execute this example. When the product name of the RX64M/71M RSK board is R0K50564MC001BR or R0K5RX71MC010BR, Table 3.3 indicates their changing. And when the product name of the RX71M RSK board is R0K50571MC000BR, Table 3.4 indicates their changing.

Table 3.3 Jumper setting

- USB access setting

Jumper	Board default setting	This example	Functional use
J2	2-3	**1-2**	**USB Enables Host Mode**
J6	1-2	**2-3**	**USB USB0VBUSEN**

Table 3.4 Jumper setting

- USB access setting

Jumper	Board default setting	This example	Functional use
J1	2-3	**1-2**	**USB Enables Host Mode**
J3	1-2	**2-3**	**USB USB0VBUSEN**

4. API Functions

4.1 FlashInit ()

This function initializes flash driver (flash API).

Format

dbram_t FlashInit(void);

Parameters

None

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR: Any error occurred

Properties

Prototyped in “db_ram.h”.

Description

This function initializes flash driver (flash API).

- If bit pattern setting is random (BIT_PAT_RAN == BIT_PAT_MODE), load seed from data flash and set initial seed by srand function.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <stdlib.h>
#include "db_ram.h"
#include "r_flash_rx_if.h"

static int8_t BPT_BUF[BIT_SEG_SIZE]; /* Bit pattern data buffer */

   dbram_t db_ret;

   /* Initialize flash API and set initial seed */
   db_ret = FlashInit();
   if (DBRAM_OK != db_ret)
   {
      goto Err_end; /* error */
   }

   /* Create bit pattern data */
   crt_bitpat(BPT_BUF, sizeof(BPT_BUF));

   /* Initializes RAM area */
   RAM_Init();

   /* Update bit pattern to 7th segment */
   db_ret = upd_bitpat(6, (uint8_t*)BPT_BUF, BIT_SEG_SIZE);
   if (DBRAM_OK != db_ret)
   {
      goto Err_end; /* error */
   }

   return;
```

Special Notes

This function need to be executed at least once after the system was started, if bit pattern setting is random.

4.2 load_seed ()

This function loads seed from data flash.

Format

uint32_t load_seed(void);

Parameters

None

Return Values

Seed data

Properties

Prototyped in “db_ram.h”.

Description

This function loads seed from data flash.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include "db_ram.h"

   dbram_t db_ret;
   uint32_t seed;

   /* Load seed from data flash */
   seed = load_seed();

   /* Update bit pattern to 7th segment */
   db_ret = upd_seed(&next_seed);
   if (DBRAM_OK != db_ret)
   {
      goto Err_end; /* error */
   }

   return;
```

Special Notes

This function is only used, if bit pattern setting is random (BIT_PAT_RAN == BIT_PAT_MODE).

4.3 upd_seed ()

This function updates seed to data flash.

Format

dbram_t upd_seed(uint32_t *dat);

Parameters

dat – seed data.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_FLERASE: Flash erase error
DBRAM_ERR_FLWRITE: Flash write error
DBRAM_ERR_FLVERIFY: Flash verify error

Properties

Prototyped in “db_ram.h”.

Description

This function updates seed to data flash.

Reentrant

Function is reentrant.

Example

Example is same as “4.2 load_seed”.

Special Notes

This function is only used, if bit pattern setting is random (BIT_PAT_RAN == BIT_PAT_MODE).

4.4 crt_bitpat ()

This function creates bit pattern data.

Format

void crt_bitpat(int8_t *dat, int32_t size);

Parameters

dat - bit pattern data.

size – bit pattern size.

Return Values

None

Properties

Prototyped in “db_ram.h”.

Description

This function creates bit pattern data.

- If bit pattern setting is random (BIT_PAT_RAN == BIT_PAT_MODE), create random data by rand function.

- If bit pattern setting is sequential (BIT_PAT_SEQ == BIT_PAT_MODE), create sequential data composed of repeating 0x00 to 0xff by 256 byte unit.

- If bit pattern setting is constant (BIT_PAT_CNT == BIT_PAT_MODE), create constant data whose value is specified by constant data pattern (CONST_PAT).

Reentrant

Function is reentrant.

Example

Example is same as “4.1 FlashInit”.

Special Notes

Need to allocate buffer more than creating bit pattern.

4.5 load_bitpat ()

This function loads bit pattern from data flash with double RAM operation.

Format

dbram_t load_bitpat(int32_t seg_no, uint8_t *dat, int32_t size);

Parameters

seg_no – segment number.

dat – bit pattern data.

size – bit pattern size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_FLDBRAM: Flash double RAM error
DBRAM_ERR_PARAM: Parameter error

Properties

Prototyped in “db_ram.h”.

Description

This function loads bit pattern from data flash with double RAM operation.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <stdio.h>
#include "db_ram.h"

static uint8_t BUF[4]; /* Bit pattern data buffer */

   dbram_t db_ret;

   /* Load bit pattern from 3rd segment,  */
   db_ret = load_bitpat(2, (uint8_t*)BUF, 4);
   if (DBRAM_OK != db_ret)
   {
      goto Err_end; /* error */
   }

   printf("Bit pattern data = %8x\n", BUF);

   return;
```

Special Notes

None.

4.6 upd_bitpat ()

This function updates bit pattern to data flash with double RAM operation.

Format

dbram_t upd_bitpat(int32_t seg_no, uint8_t *dat, int32_t size);

Parameters

seg_no – segment number.

dat – bit pattern data.

size – bit pattern size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_PARAM: Parameter error
DBRAM_ERR_FLERASE: Flash erase error
DBRAM_ERR_FLWRITE: Flash write error
DBRAM_ERR_FLVERIFY: Flash verify error

Properties

Prototyped in “db_ram.h”.

Description

This function updates bit pattern to data flash with double RAM operation following procedures.

- Erase bit pattern segment, check blank for bit pattern segment and write bit pattern.

- Erase double RAM segment, check blank for double RAM segment and write bit pattern to double RAM segment.

- Verify updated bit pattern (Not verify double RAM segment).

Reentrant

Function is reentrant.

Example

Example is same as “4.1 FlashInit”.

Special Notes

None.

4.7 ers_bitpat ()

This function erases bit pattern from data flash.

Format

dbram_t ers_bitpat(int32_t seg_no, int32_t num_seg);

Parameters

seg_no – erase segment number.

num_seg - number of erase segment.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_PARAM: Parameter error
DBRAM_ERR_FLERASE: Flash erase error

Properties

Prototyped in “db_ram.h”.

Description

This function erases bit pattern from data flash with double RAM blocks.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <stdio.h>
#include "db_ram.h"

   dbram_t db_ret;

   /* Erase bit pattern of 4th segment,  */
   db_ret = ers_bitpat(3, 1);
   if (DBRAM_OK != db_ret)
   {
      goto Err_end; /* error */
   }

   printf("Bit pattern erased of 4th segment\n");

   return;
```

Special Notes

This function erases not only bit pattern data itself but also the double RAM data.

4.8 exec_bitop ()

This function executes bit operation.

Format

dbram_t exec_bitop(int32_t seg_no, int8_t *dat, uint16_t size);

Parameters

seg_no – segment number.

dat - bit operation data.

size - bit operation data size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_FLDBRAM: Flash double RAM error
DBRAM_ERR_PARAM: Parameter error

Properties

Prototyped in “db_ram.h”.

Description

This function executes bit operation with loading bit pattern using double RAM.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <string.h>
#include "db_ram.h"

static int8_t USER_BUF[1024];

#pragma section _RAW_AREA
static int8_t R_BUF[NUM_RAW_REC][RAW_REC_SIZE]; /* Raw data area (8*1024) byte
*/
#pragma section

#pragma section _INV_AREA
static int8_t I_BUF[NUM_INV_REC][INV_REC_SIZE]; /* Invert data area (8*1024)
byte */
#pragma section

   int32_t ret;
   int32_t *dat;

   /* Compare 1st record raw data with inverted data by 1KB */
   ret = inv_comp_data(R_BUF[0], &(I_BUF[0][BIT_SEG_SIZE-1]), 1024);
   if ((-1) != ret)
   {
      ErrPtr = ret; /* set error byte */
      goto Err_end; /* compare error */
   }

   /* Set user buffer pointer */
   dat = USER_BUF;

   /* Set raw data */
   memcpy(dat, R_BUF[0], 1024);

   /* Execute bit operation */
```

```
    _exec_bitop(rec_no, dat, size);

    return;
```

Special Notes

No bit operation is executed, if kind of bit operation is no operation (OP_NONE == KND_BIT_OP).

4.9 RAM_Init ()

This function initializes RAM.

Format

void RAM_Init(void);

Parameters

None

Return Values

None

Properties

Prototyped in “db_ram.h”.

Description

This function initializes following RAM areas.

- Non safe areas: Area(1), Area(2) and Area(3).

- Safe areas: raw data area and bit inverted data area.

Reentrant

Function is reentrant.

Example

Example is same as “4.1 FlashInit”.

Special Notes

None.

4.10 RAM_Write ()

This function writes non safety data to RAM.

Format

dbram_t RAM_Write(int32_t area_no, int8_t *dat, uint16_t size);

Parameters

area_no - area number.

dat - write data.

size - data size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_PARAM: Parameter error

Properties

Prototyped in “db_ram.h”.

Description

This function writes non safety data to RAM specified by area number and size.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <stdio.h>
#include "db_ram.h"

static int8_t W_BUF[1024]; /* write buffer */
static int8_t R_BUF[1024]; /* read buffer */

   dbram_t db_ret;
   int32_t i;

   /* Write non safe data to Area(1) by 1KB */
   db_ret = RAM_Write(1, W_BUF, sizeof(W_BUF));
   if (DBRAM_OK != db_ret)
   {
      printf("Write error occurred\n");
      goto Err_end;
   }

   /* ==== Add user operation ==== */

   /* Read non safe data from Area(1) by 1KB */
   db_ret = RAM_Read(1, R_BUF, sizeof(R_BUF));
   if (DBRAM_OK != db_ret)
   {
      printf("Read error occurred\n");
      goto Err_end;
   }

   /* Compare read and write data by 1KB */
   for (i = 0; i < 1024; i++)
   {
      if (R_BUF[i] != W_BUF[i])
      {
         printf("Data error detected at %d\n", i);
         goto Err_end; /* compare error */
```

```
        }
    }

    printf("Non safe data access completed\n");

    return;
```

Special Notes

None.

4.11 RAM_SafetyWrite ()

This function writes safety data to RAM.

Format

dbram_t RAM_SafetyWrite(int32_t rec_no, int8_t *dat, uint16_t size);

Parameters

rec_no - record number.

dat - write data.

size - data size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_FLDBRAM: Flash double RAM error
DBRAM_ERR_PARAM: Parameter error

Properties

Prototyped in "db_ram.h".

Description

This function writes safety data to RAM specified by record number and size.

- Execute bit operation with loading bit pattern using double RAM.

- Write the bit operated data to raw records.

- Write bit inverting data after bit operation to invert records.

Reentrant

Function is reentrant.

Example

Example showing this function being used.

```
#include <stdio.h>
#include "db_ram.h"

static int8_t W_BUF[128]; /* write buffer */
static int8_t R_BUF[128]; /* read buffer */

   dbram_t db_ret;
   int32_t i;

   /* Write safe data to record(6)by 128B */
   db_ret = RAM_SafetyWrite (6, W_BUF, sizeof(W_BUF));
   if (DBRAM_OK != db_ret)
   {
      printf("Write error occurred\n");
      goto Err_end;
   }

   /* ==== Add user operation ==== */

   /* Read safe data from record(6)by 128B */
   db_ret = RAM_SafetyRead(6, R_BUF, sizeof(R_BUF));
   if (DBRAM_OK != db_ret)
   {
      printf("Read error occurred\n");
      goto Err_end;
   }

   /* Compare read and write data by 128B */
```

```
    for (i = 0; i < 1024; i++)
    {
        if (R_BUF[i] != W_BUF[i])
        {
            printf("Data error detected at %d\n", i);
            goto Err_end; /* compare error */
        }
    }

    printf("Safe data access completed\n");

    return;
```

Special Notes

None.

4.12 RAM_Read ()

This function reads non safety data from RAM.

Format

dbram_t RAM_Read(int32_t area_no, int8_t *dat, uint16_t size);

Parameters

area_no - area number.

dat - read data.

size - data size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_PARAM: Parameter error

Properties

Prototyped in “db_ram.h”.

Description

This function reads non safety data from RAM specified by block number and size.

Reentrant

Function is reentrant.

Example

Example is same as “4.10 RAM_Write”.

Special Notes

None.

4.13 RAM_SafetyRead ()

This function reads safety data from RAM.

Format

dbram_t RAM_SafetyRead(int32_t rec_no, int8_t *dat, uint16_t size);

Parameters

rec_no - record number.

dat - read data.

size - data size.

Return Values

DBRAM_OK: Processing completed successfully
DBRAM_ERR_FLDBRAM: Flash double RAM error
DBRAM_ERR_PARAM: Parameter error
DBRAM_ERR_CMP: Double RAM compare error

Properties

Prototyped in “db_ram.h”.

Description

This function reads safety data from RAM specified by record number and size.

- Compare data read from raw record with data read by inverse address order from inverted record. If there was any difference, a soft error was detected by the double RAM.

- Copy data from raw record to user buffer.

- Execute bit operation with loading bit pattern using double RAM.

Reentrant

Function is reentrant.

Example

Example is same as “4.11 RAM_SafetyWrite”.

Special Notes

None.

5. Reference Documents

User’s Manual: Hardware

RX64M Group User’s Manual: Hardware Rev.1.00 (R01UH0377EJ)
RX71M Group User’s Manual: Hardware Rev.1.00 (R01UH0493EJ)
The latest version can be downloaded from the Renesas Electronics website.

User’s Manual: Software

RX Family RXv2 Instruction Set Architecture User’s Manual: Hardware Rev.1.00 (R01US0071EJ)
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.

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

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

Revision History

		Description	
Rev.	Date	Page	Summary
1.00	Aug 10, 2016	—	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. 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 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 a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the 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.