To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Application Note

78K0S/Kx1+

8-Bit Single-Chip Microcontrollers

EEPROM™ Emulation

μPD78F9200
μPD78F9201
μPD78F9202
μPD78F9210
μPD78F9211
μPD78F9212
μPD78F9221
μPD78F9222
μPD78F9224
μPD78F9232
μPD78F9234
μPD78F9500
μPD78F9501
μPD78F9502
μPD78F9510
μPD78F9511
μPD78F9512

Document No. U17379EJ3V0AN00 (3rd edition)
Date Published September 2009 NS

[MEMO]

NOTES FOR CMOS DEVICES

(1) VOLTAGE APPLICATION WAVEFORM AT INPUT PIN: Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between VIL (MAX) and VIH (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between VIL (MAX) and VIH (MIN).

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

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

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

(5) POWER ON/OFF SEQUENCE: In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current. The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

(6) INPUT OF SIGNAL DURING POWER OFF STATE : Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

EEPROM is a trademark of NEC Electronics Corporation.
SuperFlash is a registered trademark of Silicon Storage Technology, Inc. in several countries including the United States and Japan.

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

INTRODUCTION

Target readers This application note is intended for users who understand the functions of the 78K0S/Kx1+ with on-chip flash memory and who will use this product to design application systems.

Purpose The purpose of this application note is to inform users concerning the use of the 78K0S/Kx1+ flash memory self programming functions, and the method for storing data (writing constant data using application) during EEPROM emulation of flash memory.

Organization This manual is generally organized into the following sections.

- EEPROM emulation function
- EEPROM emulation program

How to read this manual It is assumed that the reader of this manual has general knowledge in the fields of electrical engineering, logic circuits, and microcontrollers.

☐ To learn more about the 78K0S/Kx1+'s hardware functions:
→ See the user's manual of each 78K0S/Kx1+ product.

☐ To learn more about the 78K0S/Kx1+'s flash memory self programming functions:
→ See the flash memory chapter in the user's manual for each 78K0S/Kx1+ product.

☐ To gain a general understanding of functions:
→ Read this manual in the order of the **CONTENTS**. The mark "<R>" shows major revised points. The revised points can be easily searched by copying an "<R>" in the PDF file and specifying it in the "Find what:" field.

Convention

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

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

Document Name	Document No.
78K0/KB1+ User's Manual	U17446E
78K0/KA1+ User's Manual	U16898E
78K0/KY1+ User's Manual	U16994E
78K0/KU1+ User's Manual	U18172E
78K0S/Kx1+ EEPROM Emulation Application Note	This manual
78K/0S Series Instructions User's Manual	U11047E

CONTENTS

CHAPTER 1 OVERVIEW OF FLASH MEMORY SELF PROGRAMMING

The 78K0S/Kx1+ enables writing from an application program to flash memory (i.e., “flash memory self programming”).

This application note describes how to store (reading or writing as with EEPROM) any data to the flash memory by using the self programming function.

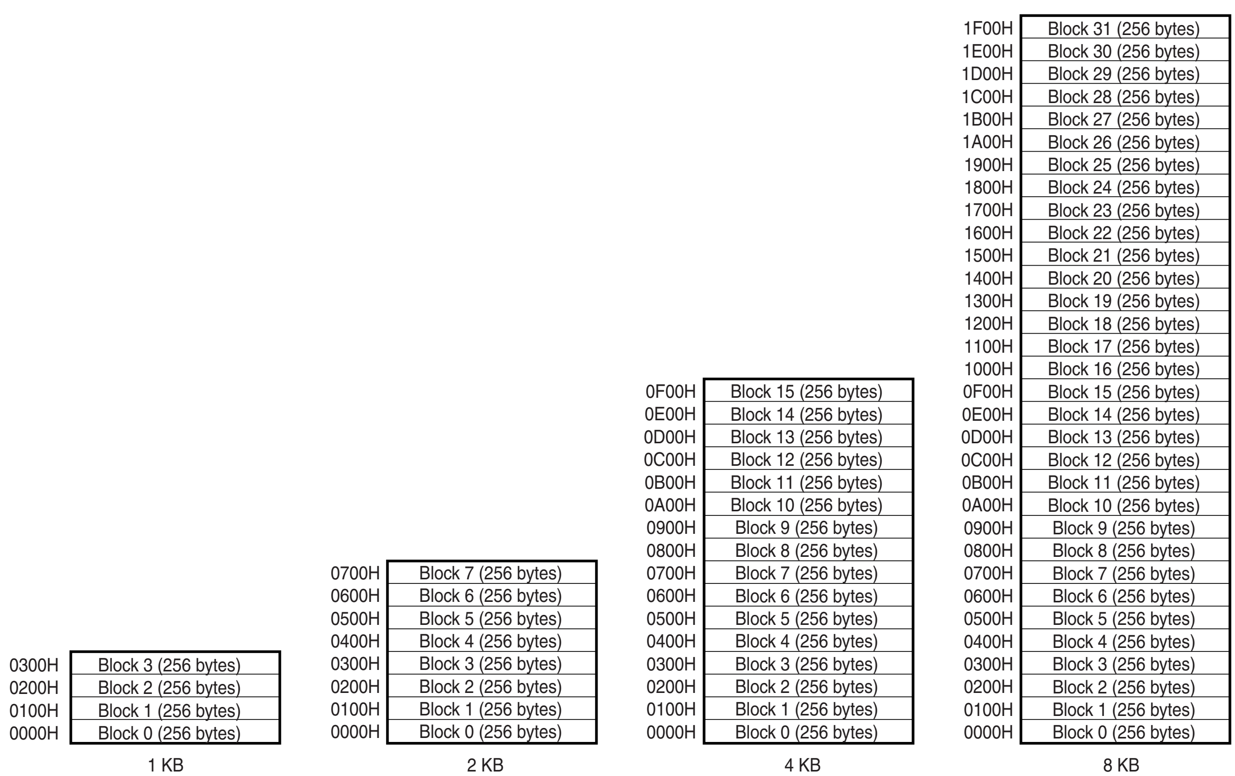
Remark For a more detailed description of flash memory self programming, see the flash memory chapter in the user’s manual for each 78K0S/Kx1+ products.

1.1 Self-Programmable Flash Memory Area

The area that is used for erasing, blank checks, and verification during flash memory control operations is specified in block units. The block numbers that can be specified are listed in Figure 1-1.

Caution Any areas other than the product’s flash memory area cannot be accessed.

Figure 1-1. Allocation of Block Numbers



CHAPTER 2 EEPROM EMULATION FUNCTION (FIXED-LENGTH SINGLE-DATA METHOD)

2.1 Main Specifications for EEPROM Emulation

EEPROM emulation is a function that is used to use a portion of the flash memory as rewritable data ROM, by using the self programming function of the flash memory. The sample program can be used in combination with a user program to perform read or write processing, as with EEPROM.

Note that the data length and number of rewrites is restricted, because the internal flash memory can only be rewritten a limited number of times. Next, the basic specifications of the sample program and how to calculate the number of rewrites is described.

Basic specifications of the sample program and how to calculate the number of rewrites

• Data format for saving

Data (2 bytes)	Delimiter (1 byte)
↑ Any data	↑ Indicates the write status. Used for data search.

Remark The data size can be set, starting from 1 byte. The upper size limit depends on the RAM size.

• Number of flash memory block rewrites

(256 − 2) / 3 = 84 times (rounded to the nearest integer)

- 3: Data size
- 2: Valid or invalid flag (see **2.1.3**)
- 256: Memory size of one block

• Number of blocks to be used as the EEPROM area

Two blocks (MIN.) are used.

Remark These blocks are required to prevent data losses due to problems, such as power cut-off and power interruption during block erasure.

<R> • Number of erasures of one block

1,000 times

<R> • Maximum number of rewrites

$84 \times 2 \times 1{,}000 = 168{,}000$ times

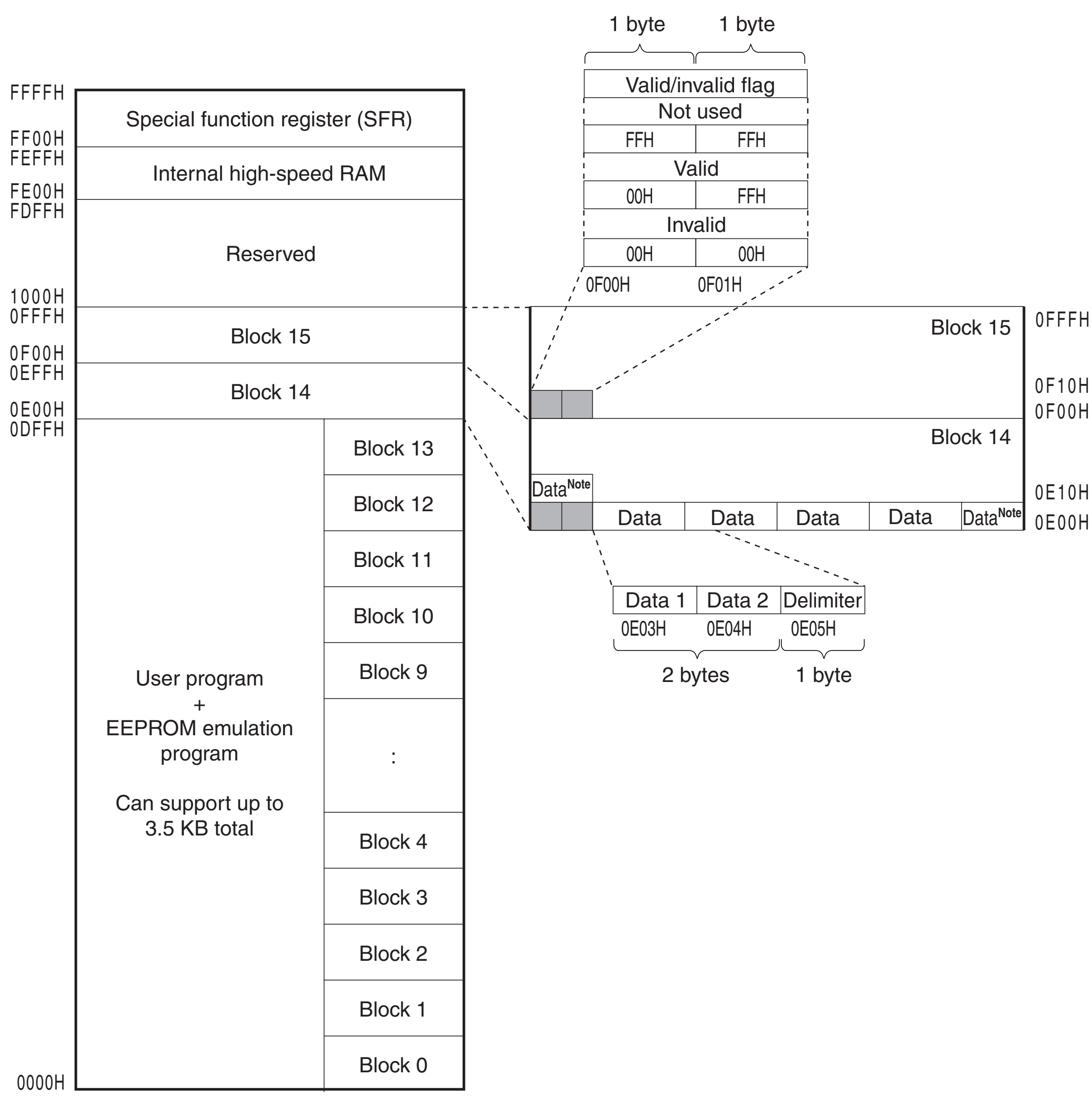
- 1,000: Number of block erasures
- 2: Number of blocks to be used
- 84: Number of rewrites of one block

In the sample program, data is handled in 2-byte units, and 84 rewrites can be performed per block (256 bytes), in combination with a delimiter (1 byte) that indicates the end of data. Furthermore, since an area of at least two <R> consecutive blocks is required to avoid losses caused, for example, by an unexpected power supply voltage drop, a total of 168 rewrites can be performed when two blocks are used. In addition, since a block can be erased up to 1,000 times in the sample program, data can be rewritten up to 168,000 times when two blocks are used.

At least two consecutive blocks must be secured to allocate the flash memory for storing data. These blocks can be set freely by the user.

Figure 2-1 shows a memory map and data structure example in which the program size is 3.5 KB and blocks 14 and 15 are set to be used for EEPROM emulation.

Figure 2-1. Memory Map and Data Structure Example
(When Program Size Is 3.5 KB and Blocks 14 and 15 Are Set as Data Area for Use EEPROM Emulation)



Note Data is stored successively.

Remark Data structure in Figure 2-1 shows the example when used the sample program.

2.1.1 EEPROM emulation data block

The EEPROM emulation program requires at least two consecutive blocks for storing data.
As long as these blocks do not overlap the user program area, they can be freely set by the user.

2.1.2 Data structure

Data that is stored as part of EEPROM emulation consists of data (1-byte unit) and a delimiter (1 byte).

Figure 2-2. Data Structure

1-byte unit	1 byte
Data	Delimiter

(1) Data

Any value from 00H to FFH can be set. The size can be set, starting from 1 byte. Note, however, that, the larger the size, the more the number of rewrites will decrease.

(2) Delimiter

The delimiter's value is fixed as 00H. Delimiters are written to enable detection of unsuccessful data writing, such as in cases where power interruptions or other problems occurred during a data write operation. Whether data writing has completed normally is judged by writing a delimiter area last. If a delimiter (00H) cannot be read correctly, it is likely that a problem will occur when writing data, so the corresponding data is not used.

If a search finds an abnormal delimiter in the latest data, the data written before that data, having a normal delimiter, is read as the latest data.

(3) Normal flow of data storage and search operations

The normal flow of the data storage and search operations are described below (in this example, the blocks specified for EEPROM emulation are blocks 14 and 15).

(Status 1) Block 14 is set as a valid block.

Block 14	+0	
Valid flag	00H	0E00H
Invalid flag	FFH	0E01H
Data	FFH	0E02H

(Status 2) Data (11H and 22H) is written.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data	11H	22H	00H	0E02H
:	FFH			0E06H

(Status 3) Data (22H and 33H) is written.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data	11H	22H	00H	0E02H
Data	22H	33H	00H	0E06H
:	FFH			0E0AH

(Status 4) Data (20H and 30H) is written.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data	11H	22H	00H	0E02H
Data	22H	33H	00H	0E06H
Data	20H	30H	00H	0E0AH
:	FFH			0E0EH

(Status 5) Data (20H and 30H) is read.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data a	11H	22H	00H	0E02H
Data b	22H	33H	00H	0E06H
Data c	20H	30H	00H	0E0AH
Data d	FFH			0E0EH
	• • •			
Erase status	FFH			0EFEH

How to read

<1> Since data a has a different data number, the operation goes to the next data.

<2> Since data b has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, and the operation goes to the next data.

<3> Since data c has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, and the operation goes to the next data.

<4> Check whether the block has been erased to the end.

<5> The read value therefore becomes the latest stored data (data c).

The following describes the flow of operations when a problem such as a power interruption occurs while storing data (in this example, the blocks specified for EEPROM emulation are blocks 14 and 15).

(Status 1) Block 14 is set as a valid block.

Block 14	+0	
Valid flag	00H	0E00H
Invalid flag	FFH	0E01H
Data	FFH	0E02H

(Status 2) Data number 1 (for data values 11H, 22H) is written.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data	11H	22H	00H	0E02H
:	FFH			0E06H

(Status 3) Power interruption occurs while data number 1 (for data values 22H, 33H) is being written and delimiter cannot be written correctly (value other than 00H is written)

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data	11H	22H	00H	0E02H
Data	22H	33H	01H	0E06H
:	FFH			0E0AH

(Status 4) Data number 1 is read.

Block 14	+0	+1	+2	
Valid flag	00H			0E00H
Invalid flag	FFH			0E01H
Data a	11H	22H	00H	0E02H
Data b	22H	33H	01H	0E06H
Data c	FFH			0E0AH
	• • •			
Erase status	FFH			0EFEH

How to read

<1> Since data a has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, the operation goes to the next data.

<2> Since data b has a matching data number, its delimiter is checked, and since the delimiter value is 01H (abnormal), the operation goes to the next data.

<3> Check whether the block has been erased to the end.

<4> The read value therefore becomes the latest stored data (data a).

2.1.3 Valid and invalid flags

Valid and invalid flags are placed at the start of the block as a total of 2 bytes of data specified in 1-byte units. As such, valid and invalid flags indicate the valid or invalid status of data stored in the corresponding block.

When the valid flag's value is 00H and the invalid flag's value is FFH, the corresponding block is valid. In all other cases, the block is invalid.

Data is stored sequentially to a valid block, and if that block becomes full, the valid/invalid flag setting makes the next block valid and the previously valid block invalid. In the event that the next block becomes full or a power interruption or other problem occurs while transferring data to the next block, this procedure enables the data up to that point to be saved in order to prevent loss of data.

The operation flow of valid and invalid flags is described below.

(Status 1) Initial status

	Block n		Block n + 1
Valid flag	FFH	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 2) Write 00H to valid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 3) Write data to block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 4) Data is full in block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	FFH
:	:	:	:
Data	Data	Data	FFH

(Status 5) Write latest data to block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 6) Write 00H to valid flag for block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 7) Write 00H to invalid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	00H	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 8) Data is full in block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	00H	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	Data

(Status 9) Erase block n

	Block n		Block n + 1
Valid flag	FFH	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 10) Write latest data to block n

	Block n		Block n + 1
Valid flag	FFH	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 11) Write 00H to valid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 12) Write 00H invalid flag for block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	00H
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

2.2 EEPROM Emulation Program Execution Conditions

Be sure to meet all of the conditions listed in Table 2-1 before executing the EEPROM emulation program.

Table 2-1. Conditions for EEPROM Emulation Operations

Item	Description
Secure stack area (Assembly language: 22 bytes)	During EEPROM emulation program operations, the stack used by the user program is inherited and used. In addition, the stack area described on the left starting from the stack address at the start of EEPROM emulation program execution is required. See **3.2 Resources Used by EEPROM Emulation Program** for further description of this stack.
EEPROM emulation program RAM (Assembly language: 8 bytes)	The area must be secured as RAM dedicated to EEPROM emulation, where read and write data are stored temporarily. Secure the area described on the left in the internal high-speed RAM as a data buffer.
Operation of watchdog timer (WDT)	Since no instruction can be executed while flash memory control processing is being performed during execution of the EEPROM emulation program, flash memory control processing clears the WDT counter. At this time, set the overflow time to 10 ms or longer so that no overflow occurs in WDT.
Prohibit reset	Do not reset this microcontroller during EEPROM emulation program operations. When a reset occurs, any data in the flash memory being accessed becomes undefined.
Prohibit power cut-off or interruption	Be sure to apply a stable voltage to the microcontroller during EEPROM emulation program operations. When a power cut-off or interruption occurs, any data in the flash memory being accessed becomes undefined.

Cautions 1. All interrupts are disabled during write processing of the EEPROM emulation program. After completion of EEPROM emulation program write processing, the interrupt mask status returns to the status before the EEPROM emulation program write processing, and interrupts are enabled.

<R> **2. When using the on-chip debug function, do not allocate areas such as the EEPROM emulation data area to the area where the monitor program for debugging is allocated.**

Example: Allocating a 2-block data area in a 4 KB flash memory product

Blocks 14 and 15 are used for the on-chip debug function, so allocate the data area to flash memory area other than blocks 14 and 15.

Remark For details on the watchdog timer operation and prohibitions on power cut-off or interruption, refer to **Cautions on self programming function** in **the flash memory chapter** in the user's manual for each 78K0S/Kx1+ product.

<R> For details on the on-chip debug function, refer to **the on-chip debug function chapter** in the user's manual for each 78K0S/Kx1+ product.

<R>

2.3 How to Get the Sample Program

Download the sample program from the URL below.

http://www.necel.com/micro/en/designsupports/sampleprogram/78k0s/low_pin_count/index.html

CHAPTER 3 EEPROM EMULATION PROGRAM (FIXED-LENGTH SINGLE-DATA METHOD, ASSEMBLYLANGUAGE)

This is an application program that uses the self programming function of the flash memory in order to use the flash memory as EEPROM memory for storing data, etc.

3.1 Configuration of EEPROM Emulation Program

Table 3-1 lists the files that comprise this program.

Table 3-1. File Configuration

File Name	Function	Type
EEPROM.asm	EEPROM emulation processing This processing includes not only read and write operations for EEPROM emulation but also data search and block transfer processing.	Assembler source

3.2 Resources Used by EEPROM Emulation Program

The resources used by this program are listed in Table 3-2 below.

Table 3-2. Resources

	Resource	Description		
	RAM	RAM for EEPROM emulation		8 bytes
		Stack		22 bytes
			EEPROM write processing	22 bytes
			EEPROM read processing	12 bytes
	ROM	EEPROM emulation processing		295 bytes
<R>		Flash memory control processing		177 bytes
<R>		Total		472 bytes

3.3 Use of EEPROM Emulation Program

EEPROM emulation described in this chapter uses at least two blocks of flash memory. Two bytes of data can be referred and updated to flash memory during EEPROM emulation.

When this EEPROM emulation program is embedded in the user application, the conditions are met (see **2.2 EEPROM Emulation Program Execution Conditions**), and the specified program is executed, EEPROM emulation can be performed.

The following explains how the “fixed-length data method, assembly language” program can be used to perform EEPROM emulations.

3.3.1 Initial values for user settings

The user must set the following items as initial values for the EEPROM emulation program.

- The first block number used as EEPROM
- The number of blocks used as EEPROM

These initial value items are included in EEPROM.asm.

(1) Block numbers used as EEPROM

Specify the block numbers to be used for EEPROM emulation. The set blocks must be consecutive for more than 2 blocks. Set the blocks so that they do not overlap the user program area.

The number of EEPROM rewrite cycles can be increased by increasing the number of blocks used as EEPROM. Regardless of the amount of data used for EEPROM emulation, we recommend that any area that is not being used as a program area should be set for use in EEPROM emulation.

Example 1. When using two blocks (blocks 14 and 15) as EEPROM blocks

```
EEPROM_BLOCK EQU (14)
EEPROM_BLOCK_NO EQU (2)
```

2. When using four blocks (blocks 12, 13, 14, and 15) as EEPROM blocks

```
EEPROM_BLOCK EQU (12)
EEPROM_BLOCK_NO EQU (4)
```

(2) Data length used by user

The data length to be stored in the EEPROM must be set by the user.

Set the data length with the data size and the delimiter (1 byte), because EEPROM emulation requires a delimiter.

Example When the data to be stored is 2 bytes

```
LENG  EQU  (3)     ; Data length (including the delimiter)
```

<R> **(3) Number of erase retrials**

The number of retrials is set in accordance with the time (MAX. value) required for the number of flash memory block erasures performed.

In the sample program of this manual, it is set (4.9 seconds) under the conditions, T_A = –40 to +85°C, 4.5 V ≤ V_{DD} ≤ 5.5 V, and NERASE ≤ 1,000 times.

To set it under different conditions, set it larger than the block erasure time divided by 8.5 ms. The time for one erase is 8.5 ms.

3.3.2 Calling of user processing for EEPROM emulation

Two types of processing are provided for use when performing EEPROM emulations in a user program: EEPROM read and write processing.

<EEPROM read and write processing>

EEPROM read and write processing facilitates EEPROM emulation by setting and calling a specific argument of a data address.

The assembler version and C language version of EEPROM read and write processing is included in main.asm and main.c, respectively.

The following variable and structure (RAM) are used both for reading and writing in EEPROM emulation.

For main.asm (assembler version), use the variable EEPROM_DATA defined below.

```
EEPROM_DATA:       DS      2       ; Data
EEPROM_DELIMITER:  DS      1       ; Delimiter
```

For main.c (C language version), use the structure eeprom_data defined below.

```
Struct eeprom_data{
  unsigned char uc_eeprom_data[2]       ; Data
  unsigned char uc_delimiter            ; Delimiter
};
```

(1) EEPROM read processing (__eeprom_read): Reads from the EEPROM area the data of the set size.

For main.asm (assembler version)

- Argument:
  Store the EEPROM_DATA address to the AX register and execute subroutine call the _eeprom_read function.
- Return value (CY flag):
  The return value is either CY=0 indicating normal completion of data read or CY=1 indicating abnormal completion. If the result is an abnormal completion, an error will occur if data with the specified number is not written even once.

For main.c (C language version)

- Argument:
  Execute the _eeprom_read function by using the address to the eeprom_data structure as the argument.
- Return value (error flag):
  The return value is either return value=0 indicating normal completion of data read or return value =1 indicating abnormal completion. If the result is an abnormal completion, an error will occur if data with the specified number is not written even once.

(2) EEPROM write processing (EEPROMWrite): Writes to the EEPROM area the data of the set size.

For main.asm (assembler version)

- Argument:
  Store the EEPROM_DATA address to the AX register and execute the _eeprom_write function after setting the data to be written to EEPROM_DATA and the delimiter to EEPROM_DELIMITER.
- Return value (CY flag):
  The return value is either CY=0 indicating normal completion of data write or CY=1 indicating abnormal completion. If the result is an abnormal completion, an error will occur if data with the specified number is not written even once.

For main.c (C language version)

- Argument:
  Execute the _eeprom_write function by using the address to the eeprom_data structure as the argument.
- Return value (error flag):
  The return value is either return value=0 indicating normal completion of data write or return value =1 indicating abnormal completion.

3.4 Description of EEPROM Emulation Program

3.4.1 User access processing for EEPROM emulation

Tables 3-3 and 3-4 list the processing that is accessed by users and used to perform read and write operations as part of EEPROM emulation.

Table 3-3. EEPROM Read Processing

(a) Assembler version

Processing name	__eeprom_read (user access function)
ROM size	29 bytes
Stack size	5 levels (10 bytes)
Input	AX: Address of variable
Return value	Normal completion: CY=0 Abnormal completion: CY=1
Description of operation	The latest data at the specified address is read from the EEPROM to the storage address. 1: Searches for blocks used as EEPROM. 2: Searches for address of latest data from valid blocks. 3: Reads latest data from searched addresses.

(b) C language version

Processing name	_eeprom_read (user access function)
ROM size	29 bytes
Stack size	5 levels (10 bytes)
Input	AX: Pointer of structure
Return value	Normal completion: error flag=0 Abnormal completion: error flag=1
Description of operation	The latest data at the specified address is read from the EEPROM to the storage address. 1: Searches for blocks used as EEPROM. 2: Searches for address of latest data from valid blocks. 3: Reads latest data from searched addresses.

Table 3-4. EEPROM Write Processing

(a) Assembler version

Processing name	__eeprom_write (user access function)
ROM size	58 bytes
Stack size	11 levels (22 bytes)
Input	AX: Address of variable
Return value	Normal completion: CY=0 Abnormal completion: CY=1
Description of operation	The data is written from to the storage address the EEPROM. 1: Searches for blocks used as EEPROM. 2: Sets as valid the block specified first, if there are no valid blocks. 3: Searches for addresses of valid blocks which can be written. 4: Performs an operation to shift to the next block, if the valid blocks are full and cannot be written. 5: Creates write data. 6: Writes to valid blocks.

(b) C language version

Processing name	_eeprom_write (user access function)
ROM size	58 bytes
Stack size	11 levels (10 bytes)
Input	AX: Pointer of structure
Return value	Normal completion: error flag=0 Abnormal completion: error flag=1
Description of operation	The data is written from the storage address to the EEPROM. 1: Searches for blocks used as EEPROM. 2: Sets as valid the block specified first, if there are no valid blocks. 3: Searches for addresses of valid blocks which can be written. 4: Performs an operation to shift to the next block, if the valid blocks are full and cannot be written. 5: Creates write data. 6: Writes to valid blocks..

3.4.2 EEPROM emulation control processing (for internal processing)

Tables 3-5 to 3-9 list the processing used to control emulation as part of EEPROM emulation.

Table 3-5. EEPROM Block Search Processing

Processing name	EEPROMUseBlockSearch
ROM size	28 bytes
Stack size	2 levels (4 bytes)
Input	None
Output	Normal completion: CY=0, A=Block table number (01H to FEH) Abnormal completion: CY=1, A=The next end block
Registers used	A
Description of operation	Searches for currently used blocks in flash memory allocated as EEPROM.

Table 3-6. EEPROM Block Initialize Processing

Processing name	EEPROMBlockInit
ROM size	17 bytes
Stack size	6 levels (12 bytes)
Input	None
Output	Normal completion: CY=0 Abnormal completion: CY=1
Registers used	A, X, B, C, D, E
Description of operation	If there are no valid blocks among the blocks specified for EEPROM, the first specified block is set as currently being used (valid). Returns with CY = 0 if secured normally. Returns with CY = 1 if not secured normally.

Table 3-7. EEPROM Block Change Processing

Processing name	EEPROMUseBlockChange
ROM size	59 bytes
Stack size	6 levels (12 bytes)
Input	A=Currently used block number
Output	Normal completion: CY=0, C=Block table number (02H to FEH), Zero flag (Z)=1 Abnormal completion: CY=1, Zero flag (Z)=0
Registers used	A, X, B, C, D, E
Description of operation	If the currently used blocks are full of data, this function searches for the next block to be used and copies data to that block. 1: Sets block to be used next. 2: Erases block to be used next. 3: Transfers the latest data from a valid block to the next block. 4: Sets the next block to be used as valid. 5: Sets currently valid blocks as invalid. 6: Stores to the new block number “CurrentB_No”.

Table 3-8. EEPROM Block Data Write Top Search Processing

Processing name	EEPROMWriteTopSearch
ROM size	44 bytes
Stack size	3 levels (6 bytes)
Input	A: Currently searched block table number
Output	Successful search: CY = 0, sets the write address to AX. Search failure: CY = 1
Registers used	A, X, B, D, E
Description of operation	Searches for specified block's write address. Completes normally only if the data area fits within the block at 0FFH.

Table 3-9. EEPROM Latest Data Search Processing

Processing name	EEPROMDataSearch
ROM size	33 bytes
Stack size	3 levels (6 bytes)
Input	A: Currently used block table number
Output	Normal completion: CY=0, DE=Address of the latest data Abnormal completion: CY=1, E=0
Registers used	A, X, D, E
Description of operation	Reads the storage address of the latest data.

3.4.3 Flash memory control processing

Tables 3-10 to 3-17 list the processing used to control the flash memory as part of EEPROM emulation.

Table 3-10. Block Erase

Processing name	SelfFlashBlockErase
<R> ROM size	29 bytes
Stack size	3 levels (6 bytes)
Input	A: Number of block to be erased
Output	Normal completion: CY=0 Abnormal completion: CY=1
Registers used	B
Description of operation	Erases the specified block and performs a blank check.

Table 3-11. Mode Transition Processing (from Self Programming Mode to Normal Mode)

Processing name	SelfFlashModeOff
ROM size	31 bytes
Stack size	1-level (2 bytes)
Input	None
Output	None
Registers used	A, X
Description of operation	Releases self programming mode.

Table 3-12. Mode Transition Processing (from Normal Mode to Self Programming Mode)

Processing name	SelfFlashModeOn
ROM size	35 bytes
Stack size	1-level (2 bytes)
Input	None
Output	None
Registers used	A, X
Description of operation	Sets self programming mode.

Table 3-13. Block Erase Processing

Processing name	FlashBlockErase
ROM size	15 bytes
Stack size	1-level (2 bytes)
Input	A: Block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Erases the specified block.

Table 3-14. Flash Self Programming Function Calling Processing

Processing name	SubFlashSelfPrg
ROM size	12 bytes
Stack size	1-level (2 bytes)
Input	None
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Calls flash self programming function.

Table 3-15. Block Blank Check Processing

Processing name	FlashBlockBlankCheck
ROM size	17 bytes
Stack size	1 level (2 bytes)
Input	A: Specified block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Performs a blank check of the specified block.

Table 3-16. Processing for Setting Block as Valid

Processing name	SetValid
ROM size	9 bytes
Stack size	1 level (2 bytes)
Input	A: Block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A, X, B, C, D, E
Description of operation	Sets as valid the block used.

Table 3-17. EEPROM Data Write Processing

	Processing name	FlashEEPROMWrite
<R>	ROM size	56 bytes
	Stack size	5 levels (10 bytes)
	Input	DE: Write start address C: Write data count AX: Write data storage address
	Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
	Registers used	D, E
	Description of operation	Writes data to EEPROM and internally verifies the data.

3.5 Flowchart of EEPROM Emulation Program

3.5.1 Flowcharts of EEPROM emulation access processings

Figures 3-1 and 3-2 show flowcharts of access processings called by users to perform read or write operations as part of EEPROM emulation.

Figure 3-1. Flowchart of EEPROM Read Processing

[Overview]

The data defined with the structure is read from specified storage address.

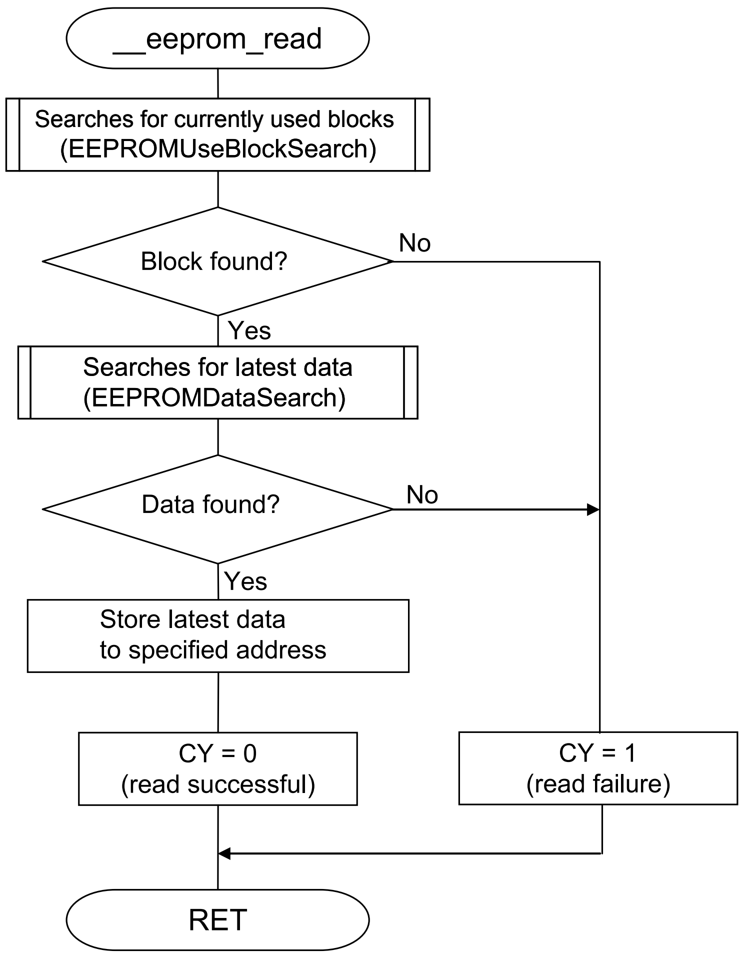
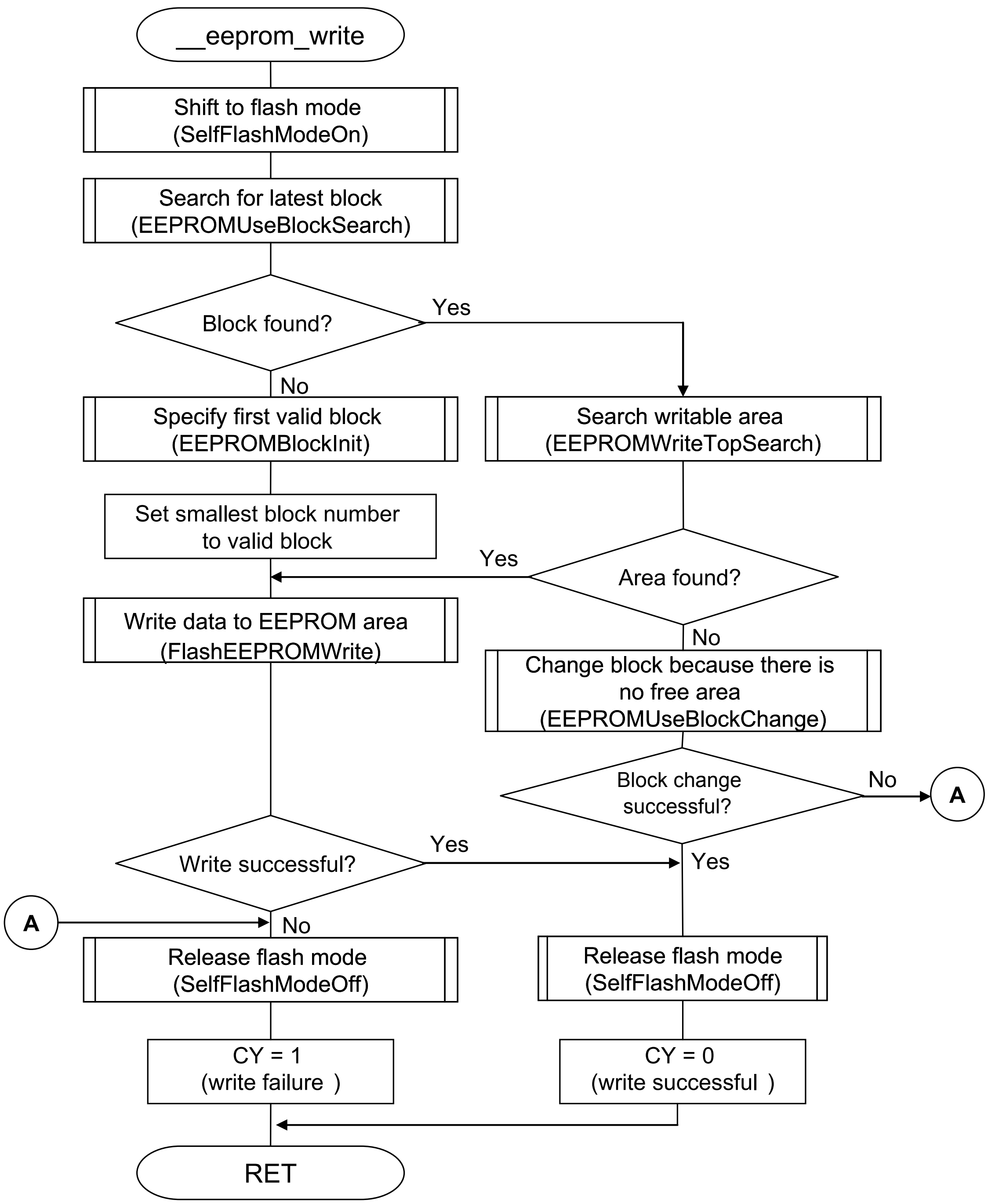


Figure 3-2. Flowchart of EEPROM Write Processing

[Overview]

The data of the specified number is written to a valid block from the storage address.



3.5.2 Flowcharts of EEPROM emulation control processings

Figures 3-3 to 3-7 show flowcharts of emulation control processings used during EEPROM emulation.

Figure 3-3. Flowchart of Currently Used EEPROM Block Search Function

[Overview]

The currently used blocks of the flash memory that is allocated as EEPROM is searched.

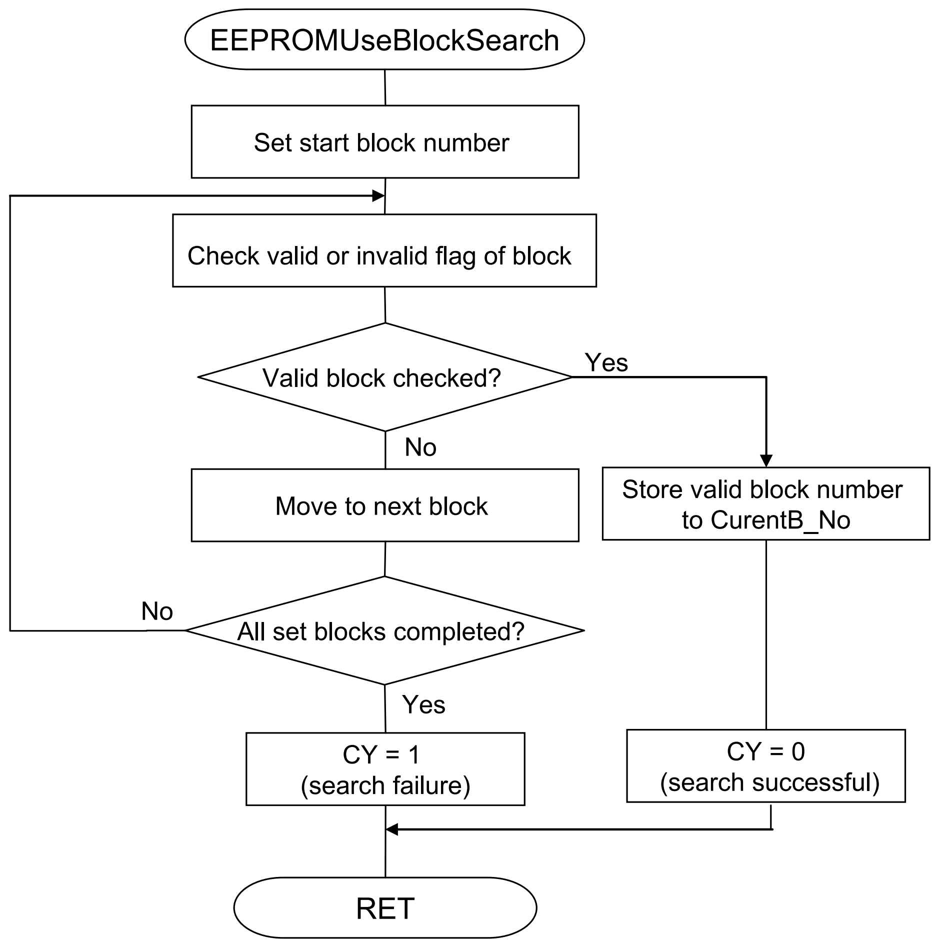


Figure 3-4. Flowchart of EEPROM Block Initialize Processing

[Overview]

If there are no valid blocks among the blocks specified for EEPROM, the first specified block is set as valid.

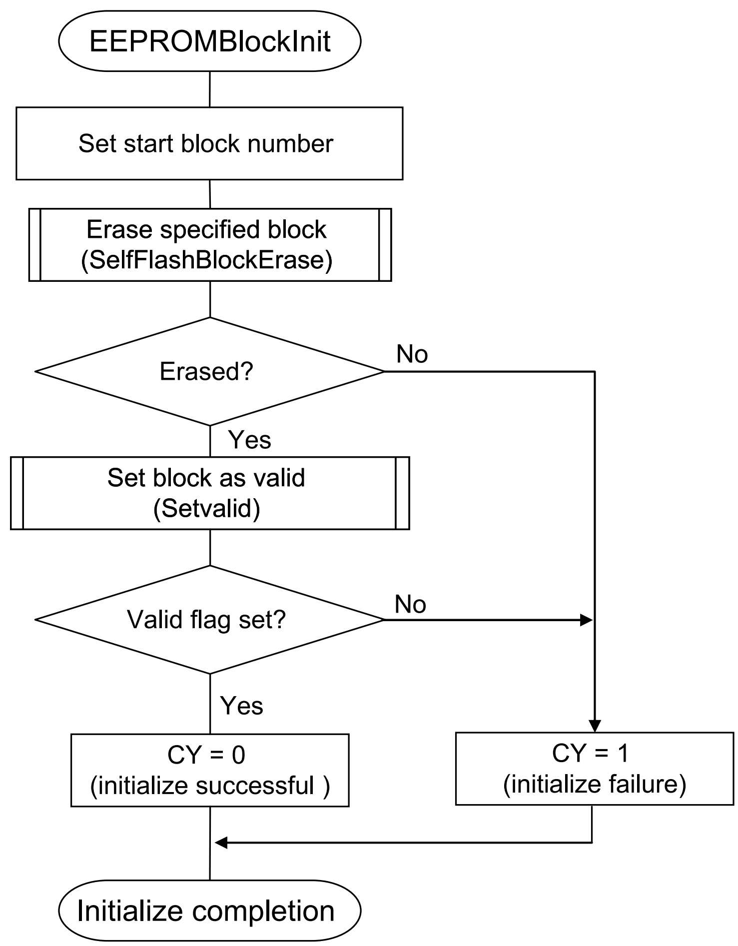


Figure 3-5. Flowchart of EEPROM Use Block Change Processing

[Overview]

If the currently used blocks are full of data, this function searches for the next block to be used and copies data to new block.

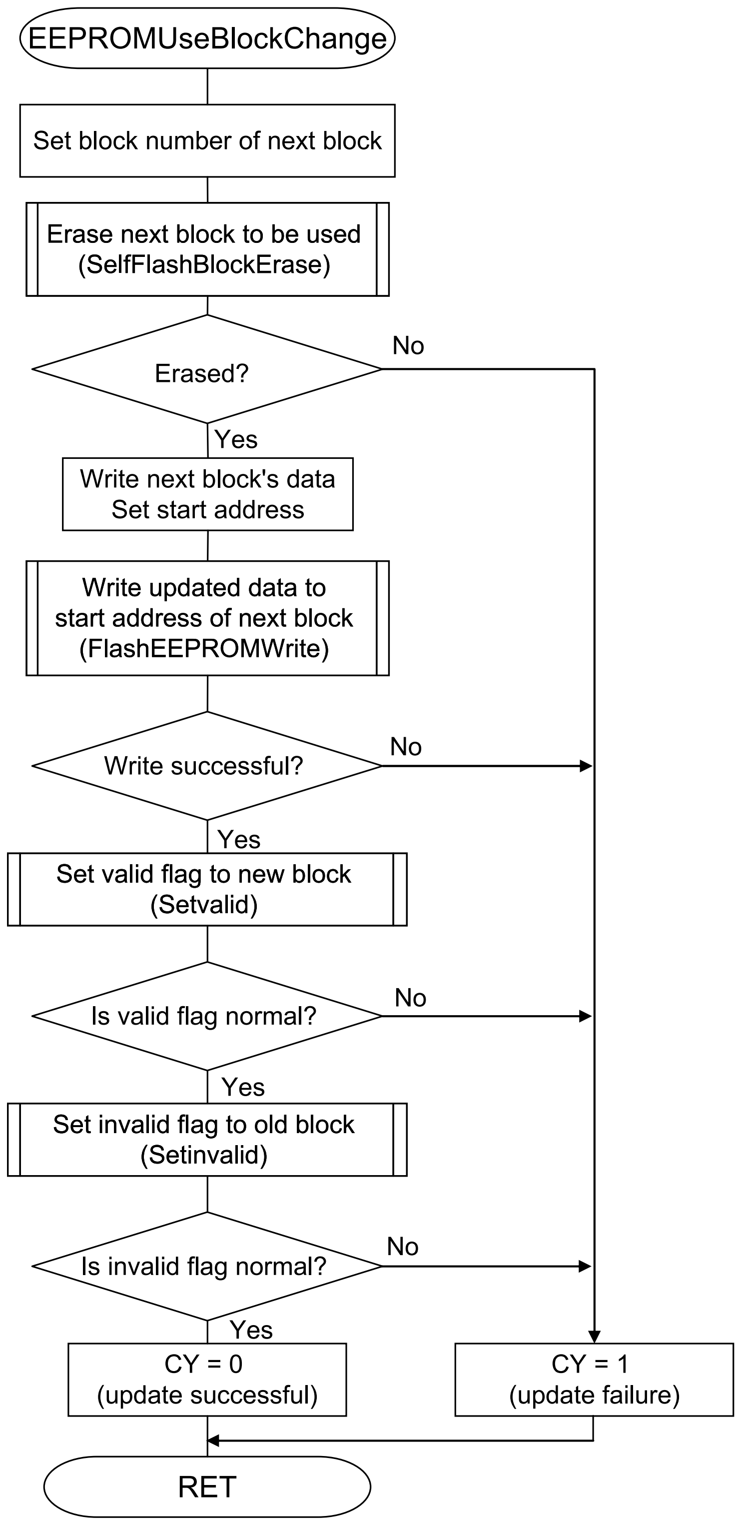


Figure 3-6. Flowchart of EEPROM Use Block Data Write Top Search Processing

[Overview]

Searches for specified block's write area.

Completes normally only if the data area fits within the block at 0xFFH.

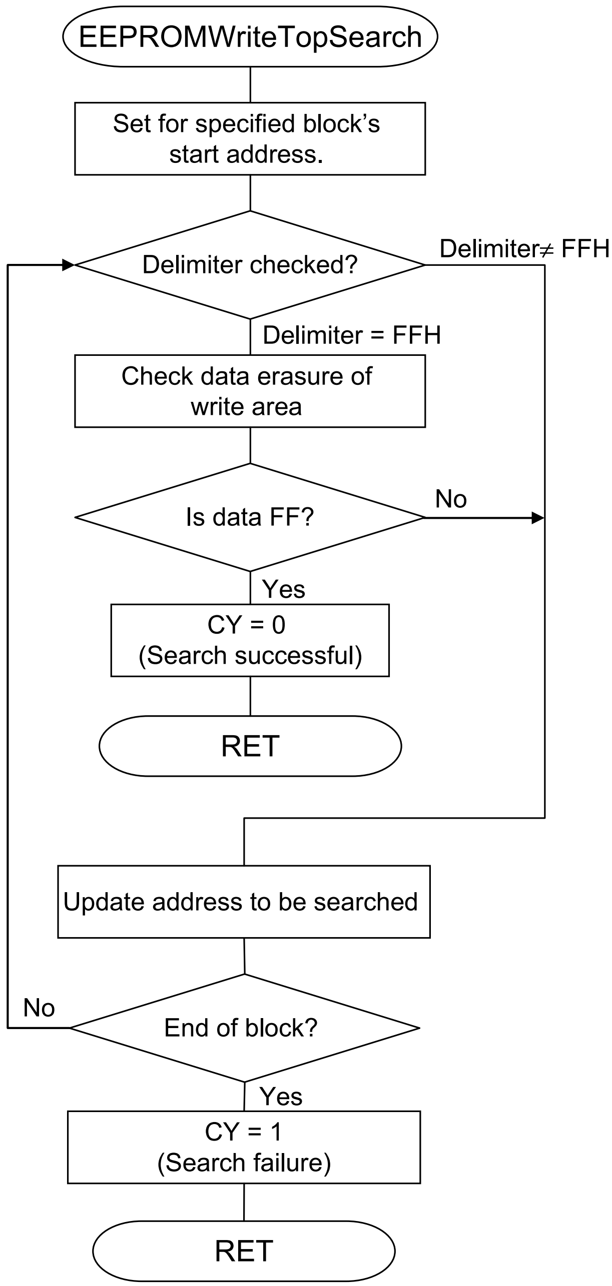
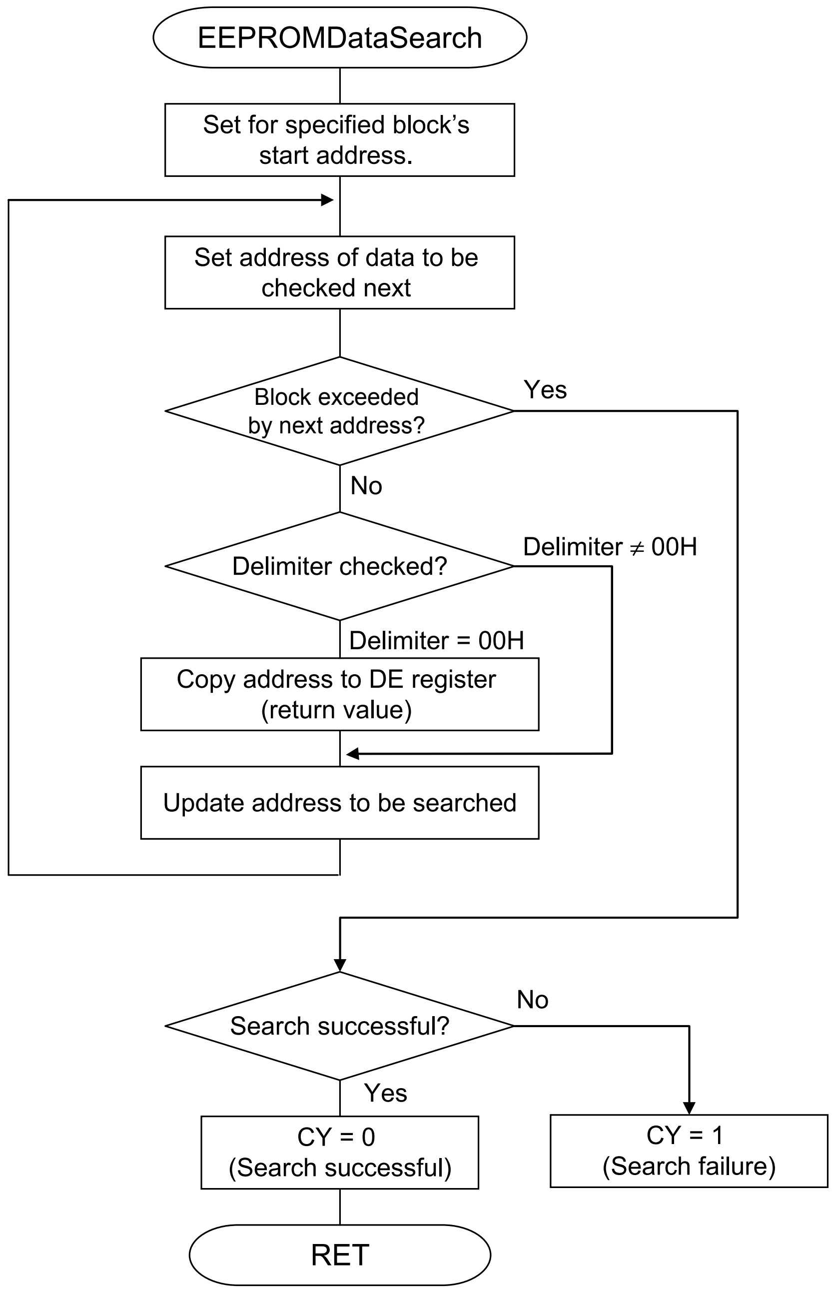


Figure 3-7. Flowchart of EEPROM Latest Data Search Processing

[Overview]

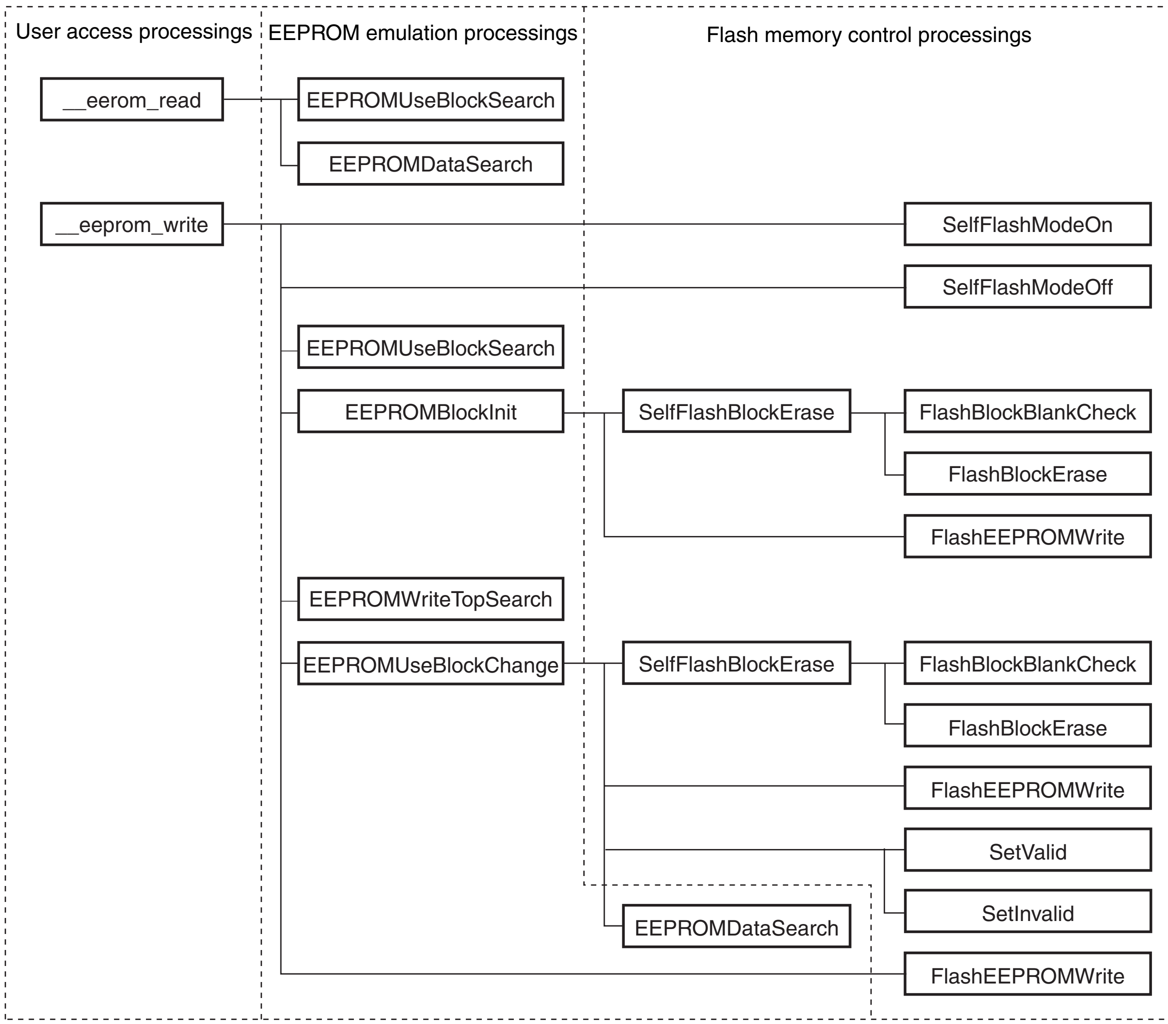
Reads the storage address of the latest data.



3.6 List of EEPROM Emulation Processings

A call tree of EEPROM emulation processings is shown below.

Figure 3-8. Call Tree



CHAPTER 4 EEPROM EMULATION FUNCTION (FIXED-LENGTH MULTIPLE-DATA METHOD)

4.1 Main Specifications for EEPROM Emulation

EEPROM emulation is a function that is used to use a portion of the flash memory as rewritable data ROM, by using the self programming function of the flash memory. The sample program can be used in combination with a user program to perform read or write processing, as with EEPROM.

Note that the data length and number of rewrites is restricted, because the internal flash memory can only be rewritten a limited number of times. Next, the basic specifications of the sample program and how to calculate the number of rewrites is described.

Basic specifications of the sample program and how to calculate the number of rewrites

• Data format for saving

Data number (1 byte)	Data (2 bytes)	Delimiter (1 byte)

↑ Number distinguishing the data to be saved (Two types of data numbers are used in the sample program.)

Remark The data size can be set, starting from 1 byte. The upper size limit depends on the RAM size.

• Number of flash memory block rewrites

(256 − 2) / 4 − 1 = 62 times (rounded to the nearest integer)

- 1: The number of data copied in the switching of block
- 4: Data size
- 2: Valid or invalid flag (see **4.1.3**)
- 256: Memory size of one block

• Number of blocks to be used as the EEPROM area

Two blocks (MIN.) are used.

Remark These blocks are required to prevent data losses due to problems, such as power cut-off and power interruption during block erasure.

<R> • Number of erasures of one block

1,000 times

<R> • Maximum number of rewrites

62 × 2 × 1,000 = 124,000 times

- 1,000: Number of block erasures
- 2: Number of blocks to be used
- 62: Number of rewrites of one block

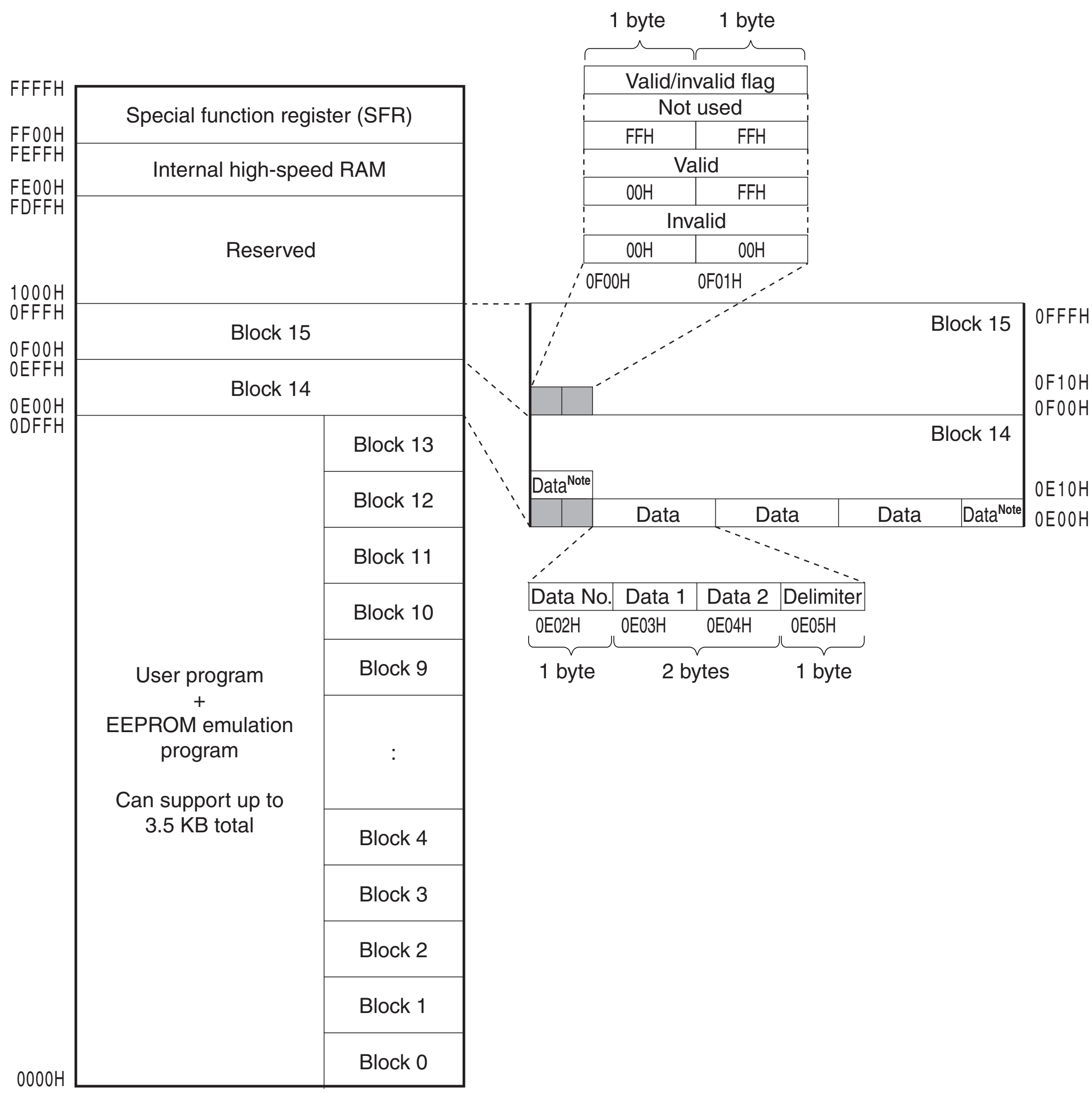
In the sample program, data is handled in 2-byte units, and 62 rewrites can be performed per block (256 bytes), in combination with a delimiter (1 byte) that indicates the end of data. Furthermore, since an area of at least two <R> consecutive blocks is required to avoid losses caused, for example, by an unexpected power supply voltage drop, a total of 124 rewrites can be performed when two blocks are used. In addition, since a block can be erased up to 1,000 times in the sample program, data can be rewritten up to 124,000 times when two blocks are used.

If, however, the total number of bytes of all EEPROM storage data (data + delimiter) types is not sufficiently large with respect to the storage block size (i.e., the total number of bytes of the data is larger than half the block size), EEPROM emulation may not function correctly. In particular, the transfer of data between blocks may occur frequently and the number of data rewrites may significantly decrease.

At least two consecutive blocks must be secured to allocate the flash memory for storing data. These blocks can be set freely by the user.

Figure 4-1 shows a memory map and data structure example in which the user program size is 3.5 KB and blocks 14 and 15 are set to be used for EEPROM emulation.

Figure 4-1. Memory Map and Data Structure Example
(When User Program Size Is 3.5 KB and Blocks 14 and 15 Are Set as Data Area for Use EEPROM Emulation)



Note Data is stored successively.

Remark Data structure in Figure 2-1 shows the example when used the sample program.

4.1.1 EEPROM emulation data block

The EEPROM emulation program requires at least two consecutive blocks for storing data.

As long as these blocks do not overlap the user program area, they can be freely set by the user.

4.1.2 Data structure

Data that is stored as part of EEPROM emulation consists of a data number (1 byte), data (1-byte unit), and a delimiter (1 byte).

Figure 4-2. Data Structure

1 byte	1-byte unit	1 byte
Data No.	Data	Delimiter

(1) Data number

Data numbers are used to distinguish data to be read or written. Any data number value from 00H to FEH is valid.

The user should assign a data number to each particular type of data before using the data.

Basically, each time a write program is executed, data is written in 4-byte units, beginning from the start of the block.

To read the latest data, search from the start of the block in 4-byte units to see whether or not the same data number exists. If several instances of data having the same data number are found, the data that is closest to the data end point is regarded as the latest data.

If the data number is FFH, it is judged as the data end point. After flash memory is erased, all data is assigned the value FFH, so when FFH is read at the location of a data number, it is judged as the data end point and the search is terminated. Accordingly, only data numbers from 00H to FEH are valid.

Remark Two types of data numbers are used in the sample program.

(2) Data

00H to FFH can be set by any value that is to be stored. The data size can be set, starting from 1 byte, but the upper size limit depends on the RAM size that can be used. Note, however, that, the larger the size, the more the number of rewrites will decrease. It is specified as 2 bytes (the data length (LENG) is defined as 4) in the sample program.

(3) Delimiter

The delimiter's value is fixed as 00H. Delimiters are written to enable detection of unsuccessful data writing, such as in cases where power interruptions or other problems occurred during a data write operation. Whether data writing has completed normally is judged by writing a delimiter area last. If a delimiter (00H) cannot be read correctly, it is likely that a problem will occur when writing data, so the corresponding data is not used.

If a search finds an abnormal delimiter in the latest data, the data written before that data (followed by the data closest to the next end point), having a normal delimiter, is read as the latest data.

(4) Normal flow of data storage and search operations

The normal flow of the data storage and search operations are described below (in this example, the blocks specified for EEPROM emulation are blocks 14 and 15).

(Status 1) Block 14 is set as a valid block.

Block 14	+0	
Valid flag	00H	0E00H
Invalid flag	FFH	0E01H
Data	FFH	0E02H

(Status 2) Data number 1 (for data values 11H and 22H) is written.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
:	FFH				0E06H

(Status 3) Data number 2 (for data values 22H, 33H) is written.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
Data	02H	22H	33H	00H	0E06H
:	FFH				0E0AH

(Status 4) Data number 2 (for data values 20H, 30H) is written.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
Data	02H	22H	33H	00H	0E06H
Data	02H	20H	30H	00H	0E0AH
:	FFH				0E0EH

(Status 5) Data number 2 is read.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data a	01H	11H	22H	00H	0E02H
Data b	02H	22H	33H	00H	0E06H
Data c	02H	20H	30H	00H	0E0AH
Data d	FFH				0E0EH

<1> Since data a has a different data number, the operation goes to the next data.

<2> Since data b has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, and the operation goes to the next data.

<3> Since data c has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, and the operation goes to the next data.

<4> The data number for data d is FFH (end point), so the read operation is terminated.

<5> The read value therefore becomes the latest stored data (data c).

The following describes the flow of operations when a problem such as a power interruption occurs while storing data (in this example, the blocks specified for EEPROM emulation are blocks 14 and 15).

(Status 1) Block 14 is set as a valid block.

Block 14	+0	
Valid flag	00H	0E00H
Invalid flag	FFH	0E01H
Data	FFH	0E02H

(Status 2) Data number 1 (for data values 11H, 22H) is written.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
:	FFH				0E06H

(Status 3) Power interruption occurs while data number 1 (for data values 22H, 33H) is being written and delimiter cannot be written correctly (value other than 00H is written)

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
Data	01H	22H	33H	01H	0E06H
:	FFH				0E0AH

(Status 4) Data number 1 is read.

Block 14	+0	+1	+2	+3	
Valid flag	00H				0E00H
Invalid flag	FFH				0E01H
Data	01H	11H	22H	00H	0E02H
Data	01H	22H	33H	01H	0E06H
:	FFH				0E0AH

How to read

<1> Since data a has a matching data number, its delimiter is checked, and since the delimiter value is 00H (normal), data of 2 bytes is stored as the latest data, the operation goes to the next data.

<2> Since data b has a matching data number, its delimiter is checked, and since the delimiter value is 01H (abnormal), the operation goes to the next data.

<3> The data number for data c is FFH (end point), so the read operation is terminated.

<4> The read value therefore becomes the latest stored data (data a).

4.1.3 Valid and invalid flags

Valid and invalid flags are placed at the start of the block as a total of 2 bytes of data specified in 1-byte units. As such, valid and invalid flags indicate the valid or invalid status of data stored in the corresponding block.

When the valid flag's value is 00H and the invalid flag's value is FFH, the corresponding block is valid. In all other cases, the block is invalid.

Data is stored sequentially to a valid block, and if that block becomes full, a search is executed to find blocks (at least two blocks are required) for storing the subsequent data, after which data is transferred to those blocks (this data is only the latest data for each data number). After the data transfer is completed, the valid/invalid flag setting makes the next block valid and the previously valid block invalid. In the event that the next block becomes full or a power interruption or other problem occurs while transferring data to the next block, this operation enables the data up to that point to be saved in order to prevent loss of data.

The operation flow of valid and invalid flags is described below.

(Status 1) Initial status

	Block n		Block n + 1
Valid flag	FFH	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 2) Write 00H to valid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 3) Write data to block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	FFH
:	:	:	:
Data	FFH	Data	FFH

(Status 4) Data is full in block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	FFH
:	:	:	:
Data	Data	Data	FFH

(Status 5) The latest data to be updated is written after the latest data whose data number has not been updated is transferred to block n + 1.

	Block n		Block n + 1
Valid flag	00H	Valid flag	FFH
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 6) Write 00H to valid flag for block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 7) Write 00H to invalid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	00H	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	FFH

(Status 8) Data is full in block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	00H	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	Data	Data	Data

(Status 9) Erase block n

	Block n		Block n + 1
Valid flag	FFH	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	FFH	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 10) The latest data to be updated is written after the latest data whose data number has not been updated is transferred to block n.

	Block n		Block n + 1
Valid flag	FFH	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 11) Write 00H to valid flag for block n

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	FFH
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

(Status 12) Write 00H invalid flag for block n + 1

	Block n		Block n + 1
Valid flag	00H	Valid flag	00H
Invalid flag	FFH	Invalid flag	00H
Data	Data	Data	Data
:	:	:	:
Data	FFH	Data	Data

4.2 EEPROM Emulation Program Execution Conditions

Be sure to meet all of the conditions listed in Table 4-1 before executing the EEPROM emulation program.

Table 4-1. Conditions for EEPROM Emulation Operations

Item	Description
Secure stack area (Assembly language: 22 bytes)	During EEPROM emulation program operations, the stack used by the user program is inherited and used. In addition, the stack area described on the left starting from the stack address at the start of EEPROM emulation program execution is required. See **5.2 Resources Used by EEPROM Emulation Program** for further description of this stack.
EEPROM emulation program RAM (Assembly language: 11 bytes)	The area must be secured as RAM dedicated to EEPROM emulation, where read and write data are stored temporarily. Secure the area described on the left in the internal high-speed RAM as a data buffer. In addition to the RAM for the program described on the left, only the stack area is used by the EEPROM emulation program.
Operation of watchdog timer (WDT)	Since no instruction can be executed while flash memory control processing is being performed during execution of the EEPROM emulation program, flash memory control processing clears the WDT counter. At this time, set the overflow time to 10 ms or longer so that no overflow occurs in WDT.
Prohibit reset	Do not reset this microcontroller during EEPROM emulation program operations. When a reset occurs, any data in the flash memory being accessed becomes undefined.
Prohibit power cut-off or interruption	Be sure to apply a stable voltage to the microcontroller during EEPROM emulation program operations. When a power cut-off or interruption occurs, any data in the flash memory being accessed becomes undefined.

Cautions 1. All interrupts are disabled during write processing of the EEPROM emulation program. After completion of EEPROM emulation program write processing, the interrupt mask status returns to the status before the EEPROM emulation program write processing, and interrupts are enabled.

<R> **2. When using the on-chip debug function, do not allocate areas such as the EEPROM emulation data area to the area where the monitor program for debugging is allocated.**

Example: Allocating a 2-block data area in a 4 KB flash memory product
Blocks 14 and 15 are used for the on-chip debug function, so allocate the data area to flash memory area other than blocks 14 and 15.

Remark For details on the watchdog timer operation and prohibitions on power cut-off or interruption, refer to **Cautions on self programming function** in **the flash memory chapter** in the user's manual for each 78K0S/Kx1+ product.

<R> For details on the on-chip debug function, refer to **the on-chip debug function chapter** in the user's manual for each 78K0S/Kx1+ product.

<R> 4.3 How to Get the Sample Program

Download the sample program from the URL below.
http://www.necel.com/micro/en/designsupports/sampleprogram/78k0s/low_pin_count/index.html

CHAPTER 5 EEPROM EMULATION PROGRAM (FIXED-LENGTH MULTIPLE-DATA METHOD, ASSEMBLY LANGUAGE)

This is an application program that uses the self programming function of the flash memory in order to use the flash memory as EEPROM memory for storing data, etc.

5.1 Configuration of EEPROM Emulation Program

Table 5-1 lists the files that comprise this program.

Table 5-1. File Configuration

File Name	Function	Type
EEPROM.asm	EEPROM emulation processing This processing includes not only read and write operations for EEPROM emulation but also data search and block transfer processing.	Assembler source

5.2 Resources Used by EEPROM Emulation Program

The resources used by this program are listed in Table 5-2 below.

Table 5-2. Resources

	Resource	Description		
	RAM	RAM for EEPROM emulation		11 bytes
		Stack		22 bytes
			EEPROM write processing	22 bytes
			EEPROM read processing	12 bytes
	ROM	EEPROM emulation processing		303 bytes
<R>		Flash memory control processing		177 bytes
<R>		Total		480 bytes

5.3 Use of EEPROM Emulation Program

EEPROM emulation described in this chapter uses at least two blocks of flash memory. Two bytes of data can be referred and updated to flash memory during EEPROM emulation.

When this EEPROM emulation program is embedded in the user application, the conditions are met (see **4.2 EEPROM Emulation Program Execution Conditions**), and the specified program is executed, EEPROM emulation can be performed.

The following explains how the “fixed-length multiple-data method, assembly language” program can be used to perform EEPROM emulations.

5.3.1 Initial values for user settings

The user must set the following items as initial values for the EEPROM emulation program.

- The first block number of blocks used as EEPROM (defined as constant EEPROM_BLOCK)
- The number of blocks used as EEPROM (defined as constant EEPROM_BLOCK_NO)
- Amount of data used by user, data length (defined as constant DATA_NO_MAX and LENG, respectively)

These initial value items are included in EEPROM.asm.

(1) The block numbers and the number of blocks used as EEPROM

Specify the block numbers of blocks to be used for EEPROM emulation. The set blocks must be consecutive for more than 2 blocks. Set the blocks so that they do not overlap the user program area.

The number of EEPROM rewrite cycles can be increased by increasing the number of blocks used as EEPROM. Regardless of the amount of data used for EEPROM emulation, we recommend that any area that is not being used as a program area should be set for use in EEPROM emulation.

Example 1. When using two blocks (blocks 14 and 15) as EEPROM blocks

```
EEPROM_BLOCK EQU (14)
EEPROM_BLOCK_NO EQU (2)
```

2. When using four blocks (blocks 12, 13, 14, and 15) as EEPROM blocks

```
EEPROM_BLOCK EQU (12)
EEPROM_BLOCK_NO EQU (4)
```

(2) Amount of data and data length used by user

The amount of data and data length to be stored in the EEPROM must be set by the user.

Set the data length with the data size, data number (1 byte), and the delimiter (1 byte), because EEPROM emulation requires data number and a delimiter.

Example When using two types of 2 bytes data

```
DATA_NO_MAX  EQU (2) ; When two units of data (data numbers 0 and 1) are to be used
LENG  EQU (4)        ; Data length (size including data number and delimiter
                       (2 bytes, total))
```

<R> **(3) Number of erase retrials**

The number of retrials is set in accordance with the time (MAX. value) required for the number of flash memory block erasures performed.

In the sample program of this manual, it is set (4.9 seconds) under the conditions, T_A = –40 to +85°C, 4.5 V ≤ V_{DD} ≤ 5.5 V, and N_{ERASE} ≤ 1,000 times.

To set it under different conditions, set it larger than the block erasure time divided by 8.5 ms. The time for one erase is 8.5 ms.

5.3.2 Calling of user processing for EEPROM emulation

Two types of processing are provided for use when performing EEPROM emulations in a user program: EEPROM read and write processing.

<EEPROM read and write processing>

EEPROM read and write processing facilitates EEPROM emulation by calling address of variable or structure as specific argument.

The assembler version and C language version of EEPROM read and write processing is included in main.asm and main.c, respectively.

The following variable and structure (RAM) are used both for reading and writing in EEPROM emulation.

For main.asm (assembler version), use the variable EEPROM_DATA defined below.

```
EEPROM_DO:         DS     1      ; Data number
EEPROM_DATA:       DS     2      ; Data
EEPROM_DELIMITER:  DS     1      ; Delimiter
```

For main.c (C language version), use the structure eeprom_data defined below.

```
Struct eeprom_data{
  unsigned char uc_data_no             ; Data number
  unsigned char uc_eeprom_data[2]      ; Data
  unsigned char uc_delimiter           ; Delimiter
};
```

(1) EEPROM read processing (__eeprom_read): Reads from the EEPROM area the data of the set size.

For main.asm (assembler version)

- Argument:
  Store the EEPROM_NO address to the AX register and call the __eeprom_read function as a subroutine, after setting to EEPROM_NO the data number of the data to be read.
- Return value (CY flag):
  The return value is either CY=0 indicating normal completion of data read or CY=1 indicating abnormal completion. If the result is an abnormal completion, be sure to check whether or not the data number is within the set range. An error will occur if data with the specified number is not written even once.

For main.c (C language version)

- Argument:
  Execute the _eeprom_read function by using the address to the eeprom_data structure as the argument, after setting to uc_data_no of eeprom_data structure the data number of the data to be read.

- Return value (error flag):
  The return value is either return value=0 indicating normal completion of data read or return value =1 indicating abnormal completion. If the result is an abnormal completion, be sure to check whether or not

the data number is within the set range. An error will occur if data with the specified number is not written even once.

(2) EEPROM write processing (__eeprom_write): Writes to the EEPROM area the data of the set size.

For main.asm (assembler version)

- Argument:
  Store the EEPROM_NO address to the AX register and execute the __eeprom_write function, after setting the data number of the data to be written, data, and delimiter to EEPROM_NO, EEPROM_DATA, and EEPROM_DELIMITER, respectively.
- Return value (CY flag):
  The return value is either CY=0 indicating normal completion of data write or CY=1 indicating abnormal completion. If the result is an abnormal completion, be sure to check whether or not the data number is within the set range. An error will occur if data with the specified number is not written even once.

For main.c (C language version)

- Argument:
  Store the EEPROM_NO address to the AX register and execute the __eeprom_write function, after setting the data number of the data to be written, data, and delimiter to uc_data_no eeprom_data structure, uc_eeprom_data, and uc_delimiter, respectively.
- Return value (error flag):
  The return value is either return value=0 indicating normal completion of data write or return value =1 indicating abnormal completion.

5.4 Description of EEPROM Emulation Program

5.4.1 User access processing for EEPROM emulation

Tables 5-3 and 5-4 list the processing that is accessed by users and used to perform read and write operations as part of EEPROM emulation.

Table 5-3. EEPROM Read Processing

(a) Assembler version

Processing name	__eeprom_read (user access function)
ROM size	31 bytes
Stack size	6 levels (12 bytes)
Input	AX: Address of variable
Return value	Normal completion: CY=0 Abnormal completion: CY=1
Description of operation	The latest data at the specified data number is read from the EEPROM to the storage address. 1: Searches for blocks used as EEPROM. 2: Searches for address of latest data from valid blocks. 3: Reads latest data from searched addresses.

(b) C language version

Processing name	_eeprom_read (user access function)
ROM size	31 bytes
Stack size	6 levels (12 bytes)
Input	AX: Pointer of structure
Return value	Normal completion: error flag=0 Abnormal completion: error flag=1
Description of operation	The latest data at the specified data number is read from the EEPROM to the storage address. 1: Searches for blocks used as EEPROM. 2: Searches for address of latest data from valid blocks. 3: Reads latest data from searched addresses.

Table 5-4. EEPROM Write Processing

(a) Assembler version

Processing name	__eeprom_write (user access function)
ROM size	63 bytes
Stack size	11 levels (22 bytes)
Input	AX: Address of variable
Return value	Normal completion: CY=0 Abnormal completion: CY=1
Description of operation	The data of specified number is written from the storage address to the EEPROM. 1: Searches for blocks used as EEPROM. 2: Sets as valid the block specified first, if there are no valid blocks. 3: Searches for addresses of valid blocks which can be written. 4: Performs an operation to shift to the next block, if the valid blocks are full and cannot be written. 5: Creates write data. 6: Writes to valid blocks.

(b) C language version

Processing name	_eeprom_write (user access function)
ROM size	63 bytes
Stack size	11 levels (10 bytes)
Input	AX: Pointer of structure
Return value	Normal completion: error flag=0 Abnormal completion: error flag=1
Description of operation	The data of specified number is written from the storage address to the EEPROM. 1: Searches for blocks used as EEPROM. 2: Sets as valid the block specified first, if there are no valid blocks. 3: Searches for addresses of valid blocks which can be written. 4: Performs an operation to shift to the next block, if the valid blocks are full and cannot be written. 5: Creates write data. 6: Writes to valid blocks..

5.4.2 EEPROM emulation control processing (for internal processing)

Tables 5-5 to 5-10 list the processing used to control emulation as part of EEPROM emulation.

Table 5-5. EEPROM Parameter Acquisition Processing

Processing name	getprara
ROM size	8 bytes
Stack size	1-level (2 bytes)
Input	AX: pointer of structure
Output	A=data number. Copies also to RQDATA_No. HL=data address
Description of operation	Reads from the argument (pointer) for the user to call a function, the content of its structure and acquires the required parameters.

Table 5-6. EEPROM Block Search Processing

Processing name	EEPROMUseBlockSearch
ROM size	27 bytes
Stack size	2-level (4 bytes)
Input	None
Output	Normal completion: CY=0, A=Block table number (01H to FEH) Abnormal completion: CY=1, A=The next end block
Registers used	A
Description of operation	Searches for currently used blocks in flash memory allocated as EEPROM.

Table 5-7. EEPROM Block Initialize Processing

Processing name	EEPROMBlockInit
ROM size	19 bytes
Stack size	6-level (12 bytes)
Input	None
Output	Normal completion: CY=0 Abnormal completion: CY=1
Registers used	A, X, B, C, D, E
Description of operation	If there are no valid blocks among the blocks specified for EEPROM, the first specified block is set as currently being used (valid). Returns with CY = 0 if secured normally. Returns with CY = 1 if not secured normally.

Table 5-8. EEPROM Block Change Processing

Processing name	EEPROMUseBlockChange
ROM size	88 bytes
Stack size	6 levels (12 bytes)
Input	A=Currently used block number
Output	Normal completion: CY=0 Abnormal completion: CY=1
Registers used	A, X, B, C, D, E
Description of operation	If the currently used blocks are full of data, this function searches for the next block to be used and copies data to that block. 1: Sets block to be used next. 2: Erases block to be used next. 3: Transfers the latest data from a valid block to the next block. 4: Sets the next block to be used as valid. 5: Sets currently valid blocks as invalid. 6: Stores to the new block number “CurrentB_No”.

Table 5-9. EEPROM Block Data Write Top Search Processing

Processing name	EEPROMWriteTopSearch
ROM size	26 bytes
Stack size	3 levels (6 bytes)
Input	A: Currently searched block table number
Output	Successful search: CY = 0, sets the write address to AX. Search failure: CY = 1
Registers used	A, AX
Description of operation	Searches for specified block’s write address. Completes normally only if the data area fits within the block at 0FFH.

Table 5-10. EEPROM Latest Data Search Processing

Processing name	EEPROMDataSearch
ROM size	41 bytes
Stack size	2 levels (4 bytes)
Input	A: Currently used block table number, X: Search data number 5
Output	Normal completion: CY=0, DE=Address of the latest data Abnormal completion: CY=1, E=0
Registers used	A, D, E
Description of operation	Reads the storage address of the latest data corresponding to the specified number.

5.4.3 Flash memory control processing

Tables 5-11 to 5-18 list the processing used to control the flash memory as part of EEPROM emulation.

Table 5-11. Block Erase

	Processing name	SelfFlashBlockErase
<R>	ROM size	29 bytes
	Stack size	3 levels (8 bytes)
	Input	A: Number of block to be erased
	Output	Normal completion: CY=0 Abnormal completion: CY=1
	Registers used	B
	Description of operation	Erases the specified block and performs a blank check.

Table 5-12. Mode Transition Processing (from Self Programming Mode to Normal Mode)

Processing name	SelfFlashModeOff
ROM size	31 bytes
Stack size	1 level (2 bytes)
Input	None
Output	None
Registers used	A, X
Description of operation	Releases self programming mode.

Table 5-13. Mode Transition Processing (from Normal Mode to Self Programming Mode)

Processing name	SelfFlashModeOn
ROM size	35 bytes
Stack size	1 level (2 bytes)
Input	None
Output	None
Registers used	A, X
Description of operation	Sets self programming mode.

Table 5-14. Block Erase Processing

Processing name	FlashBlockErase
ROM size	15 bytes
Stack size	1 level (2 bytes)
Input	A: Block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Erases the specified block.

Table 5-15. Flash Self Programming Function Calling Processing

Processing name	SubFlashSelfPrg
ROM size	12 bytes
Stack size	1 level (2 bytes)
Input	None
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Calls flash self programming function.

Table 5-16. Block Blank Check Processing

Processing name	FlashBlockBlankCheck
ROM size	17 bytes
Stack size	1 level (2 bytes)
Input	A: Specified block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A
Description of operation	Performs a blank check of the specified block.

Table 5-17. Processing for Setting Block as Valid

Processing name	SetValid
ROM size	9 bytes
Stack size	1 level (2 bytes)
Input	A: Block number
Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
Registers used	A, X, C, D, E
Description of operation	Sets as valid the block used.

Table 5-18. EEPROM Data Write Processing

	Processing name	FlashEEPROMWrite
<R>	ROM size	56 bytes
	Stack size	5 levels (10 bytes)
	Input	DE: Write start address C: Write data count AX: Write data storage address
	Output	Normal completion: Zero flag (Z)=1 Abnormal completion: Zero flag (Z)=0
	Registers used	D, E
	Description of operation	Writes data to EEPROM and internally verifies the data.

5.5 Flowchart of EEPROM Emulation Program

5.5.1 Flowcharts of EEPROM emulation access processings

Figures 5-1 and 5-2 show flowcharts of access processings called by users to perform read or write operations as part of EEPROM emulation.

Figure 5-1. Flowchart of EEPROM Read Processing

[Overview]
The data defined with the structure is read from specified storage address.

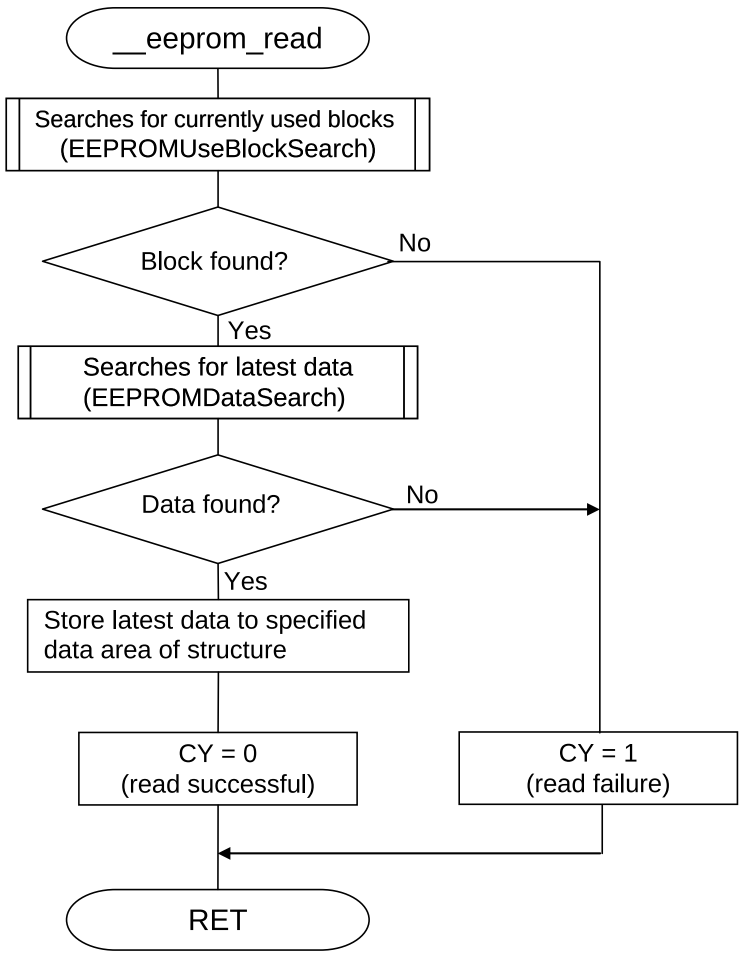
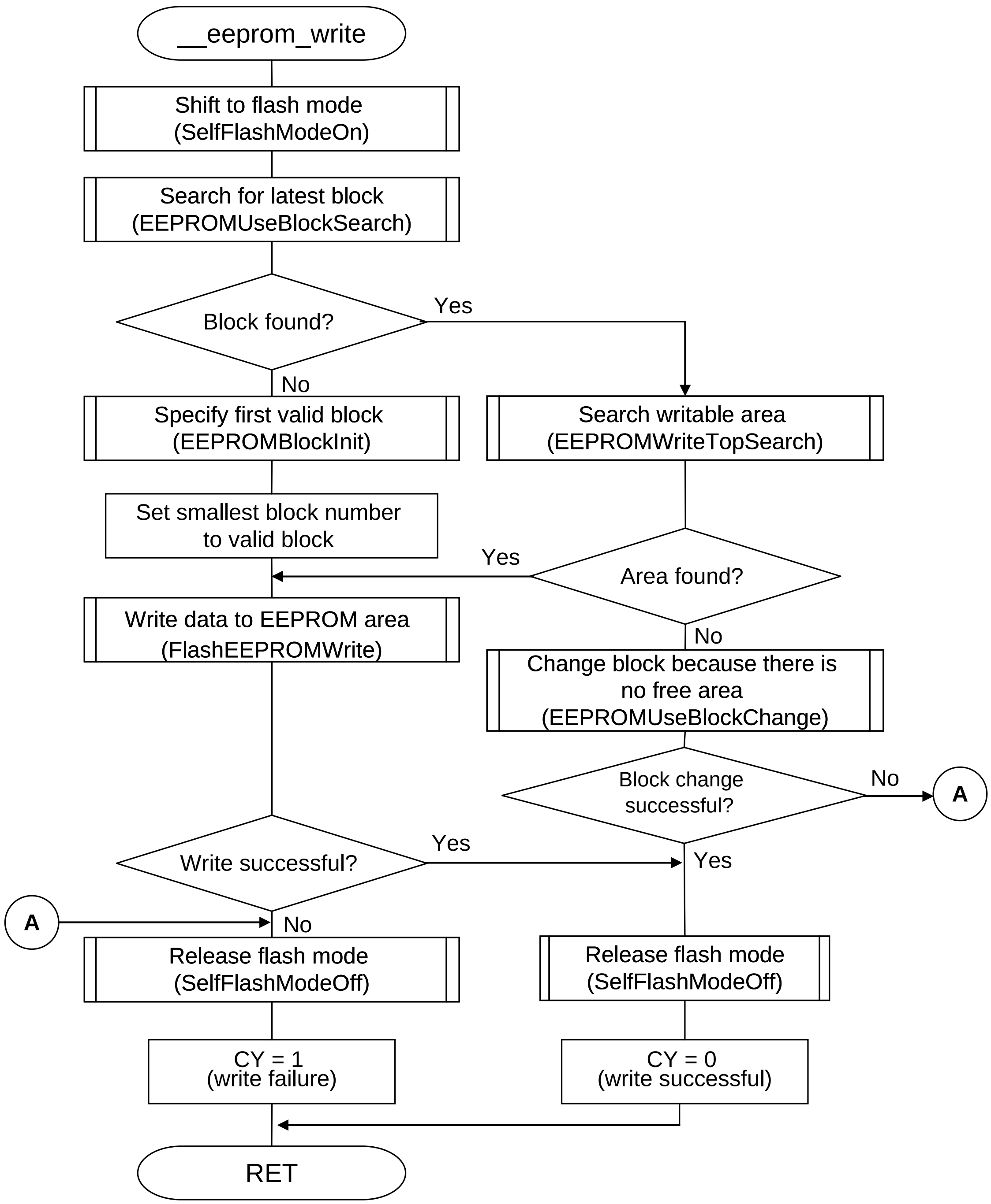


Figure 5-2. Flowchart of EEPROM Write Processing

[Overview]

The data of the specified number is written to a valid block from the storage address.



5.5.2 Flowcharts of EEPROM emulation control processings

Figures 5-3 to 5-7 show flowcharts of emulation control processings used during EEPROM emulation.

Figure 5-3. Flowchart of Currently Used EEPROM Block Search Function

[Overview]

The currently used blocks of the flash memory that is allocated as EEPROM is searched.

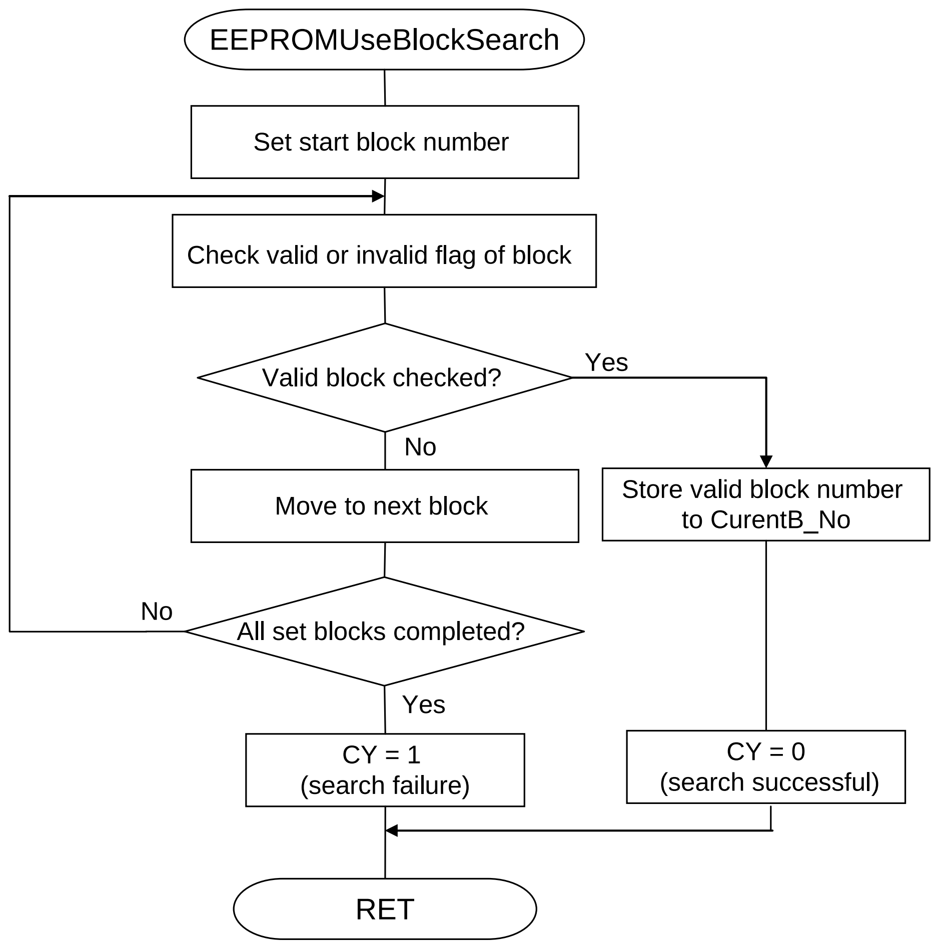


Figure 5-4. Flowchart of EEPROM Block Initialize Processing

[Overview]

If there are no valid blocks among the blocks specified for EEPROM, the first specified block is set as valid.

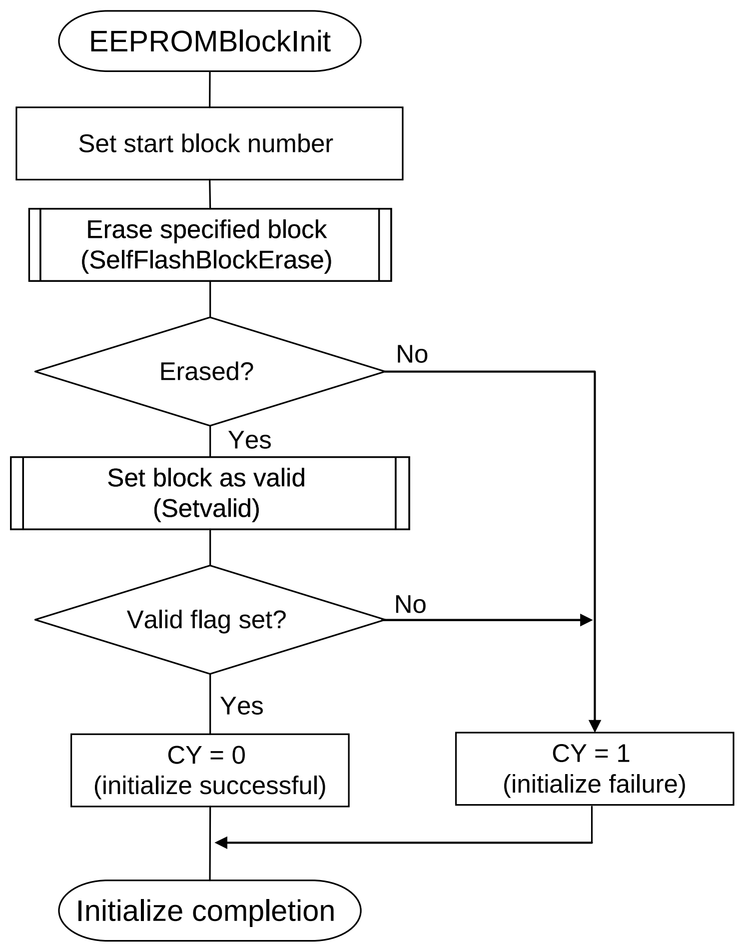


Figure 5-5. Flowchart of EEPROM Use Block Change Processing (1/2)

[Overview]

If the currently used blocks are full of data, this function searches for the next block to be used and copies data to new block.

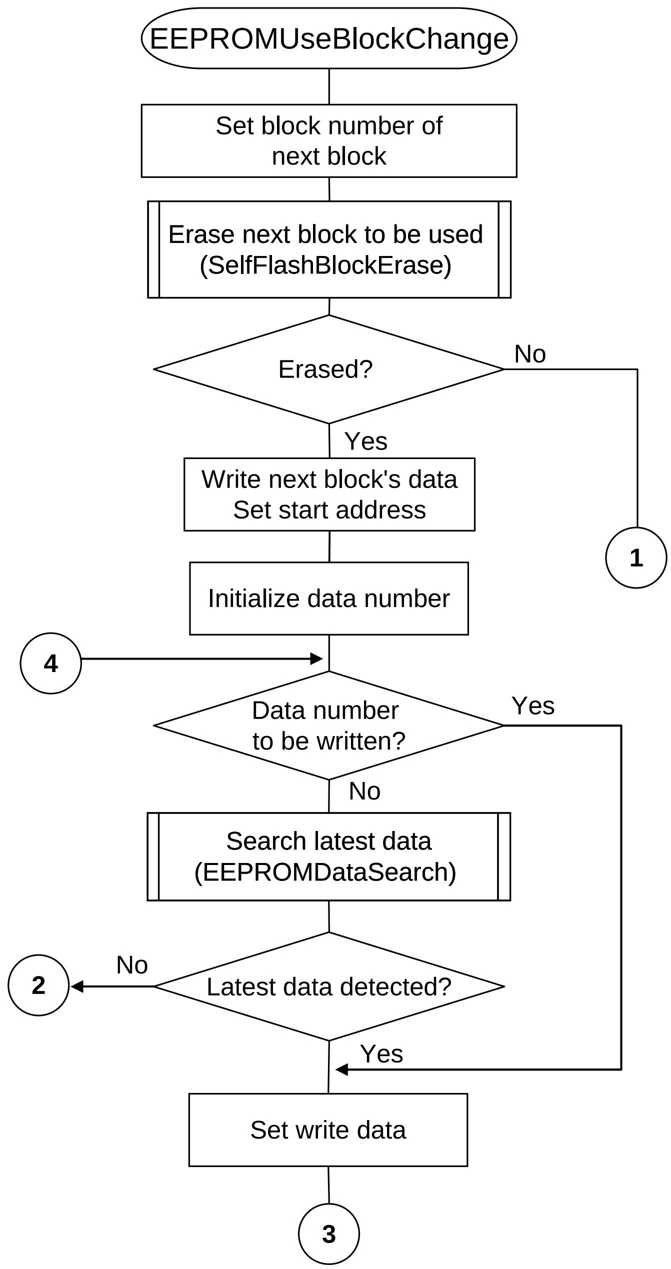


Figure 5-5. Flowchart of EEPROM Use Block Change Processing (2/2)

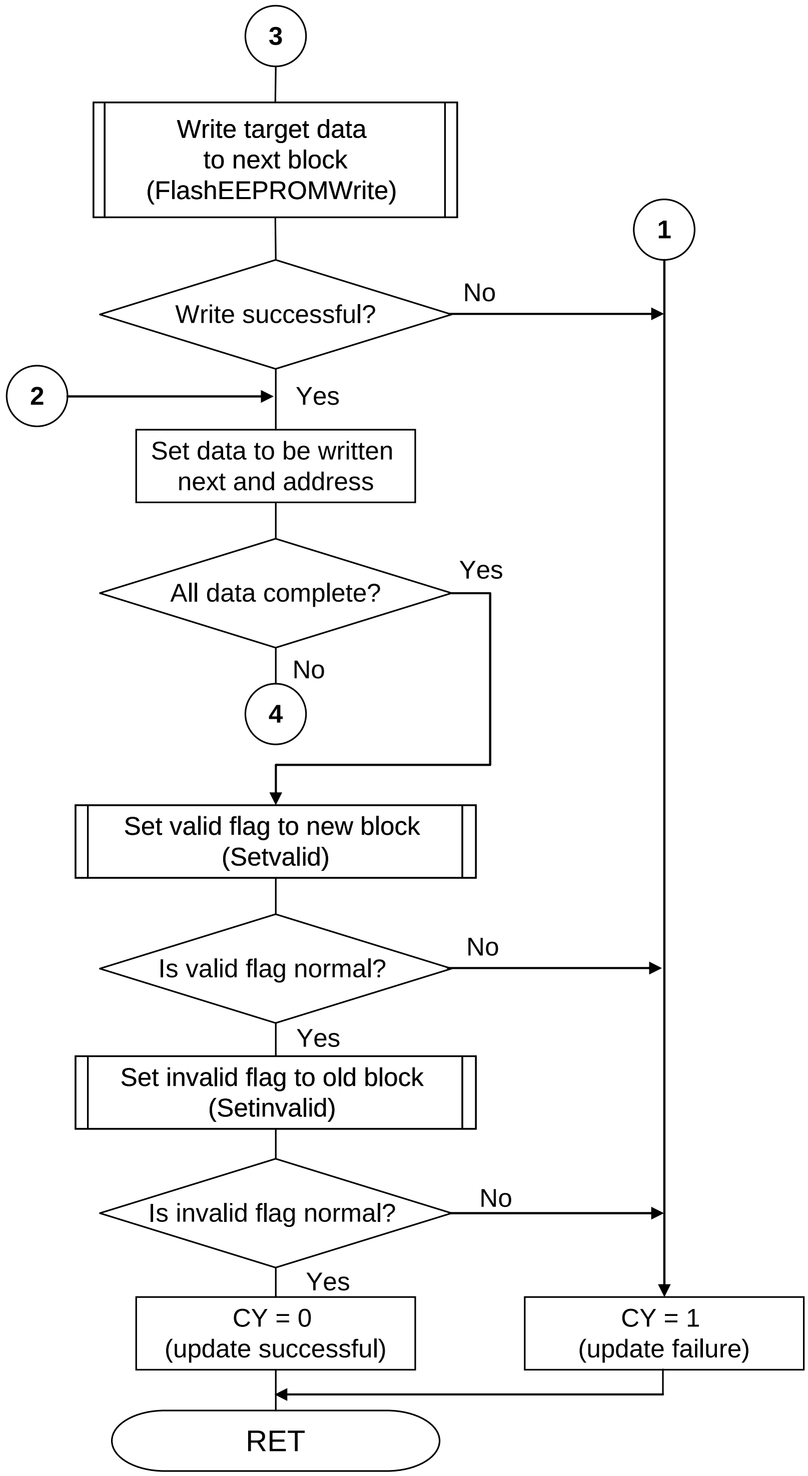


Figure 5-6. Flowchart of EEPROM Use Block Data Write Top Search Processing

[Overview]
Searches for specified block's write area.
Completes normally only if the data area fits within the block at 0xFFH.

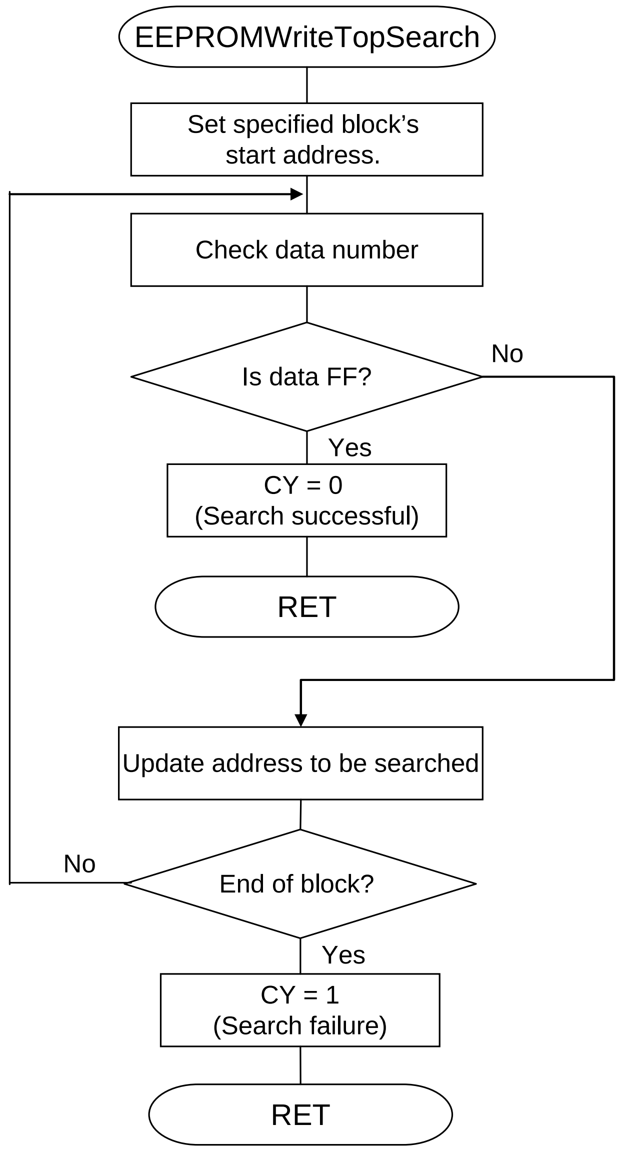
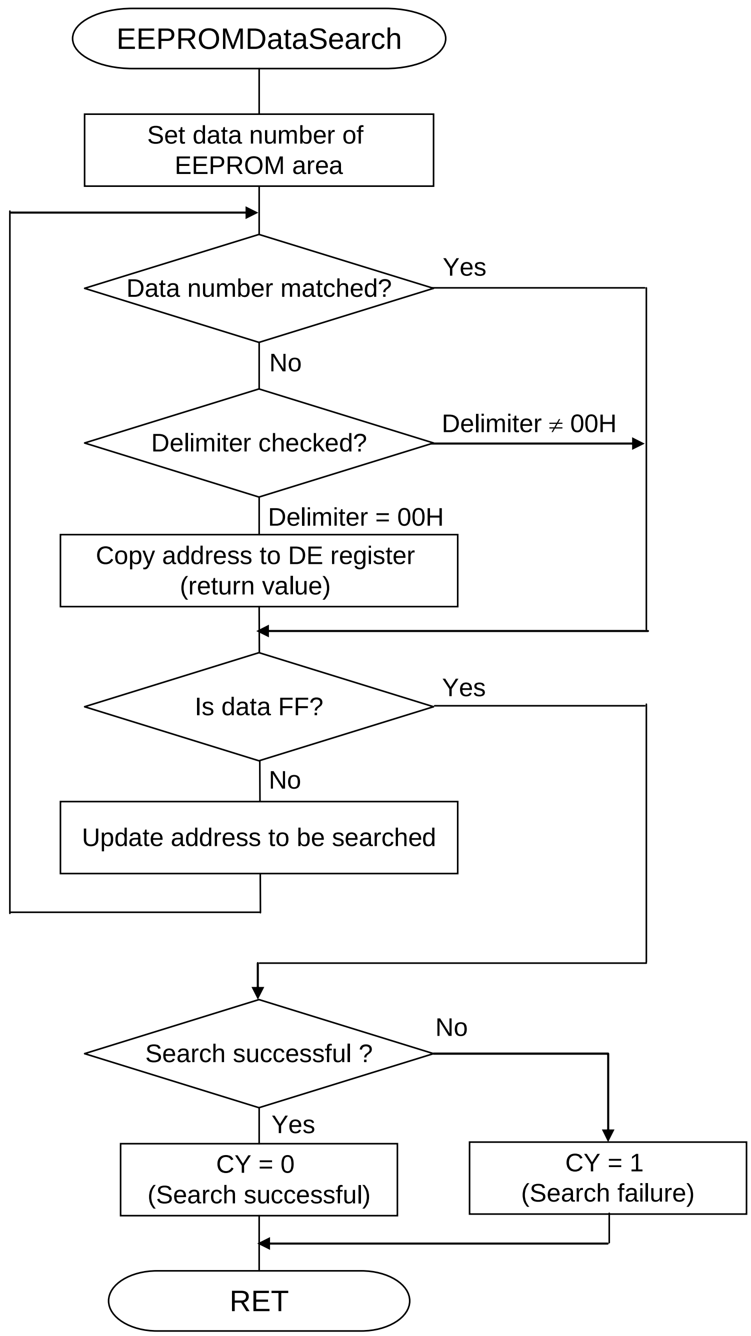


Figure 5-7. Flowchart of EEPROM Latest Data Search Processing

[Overview]

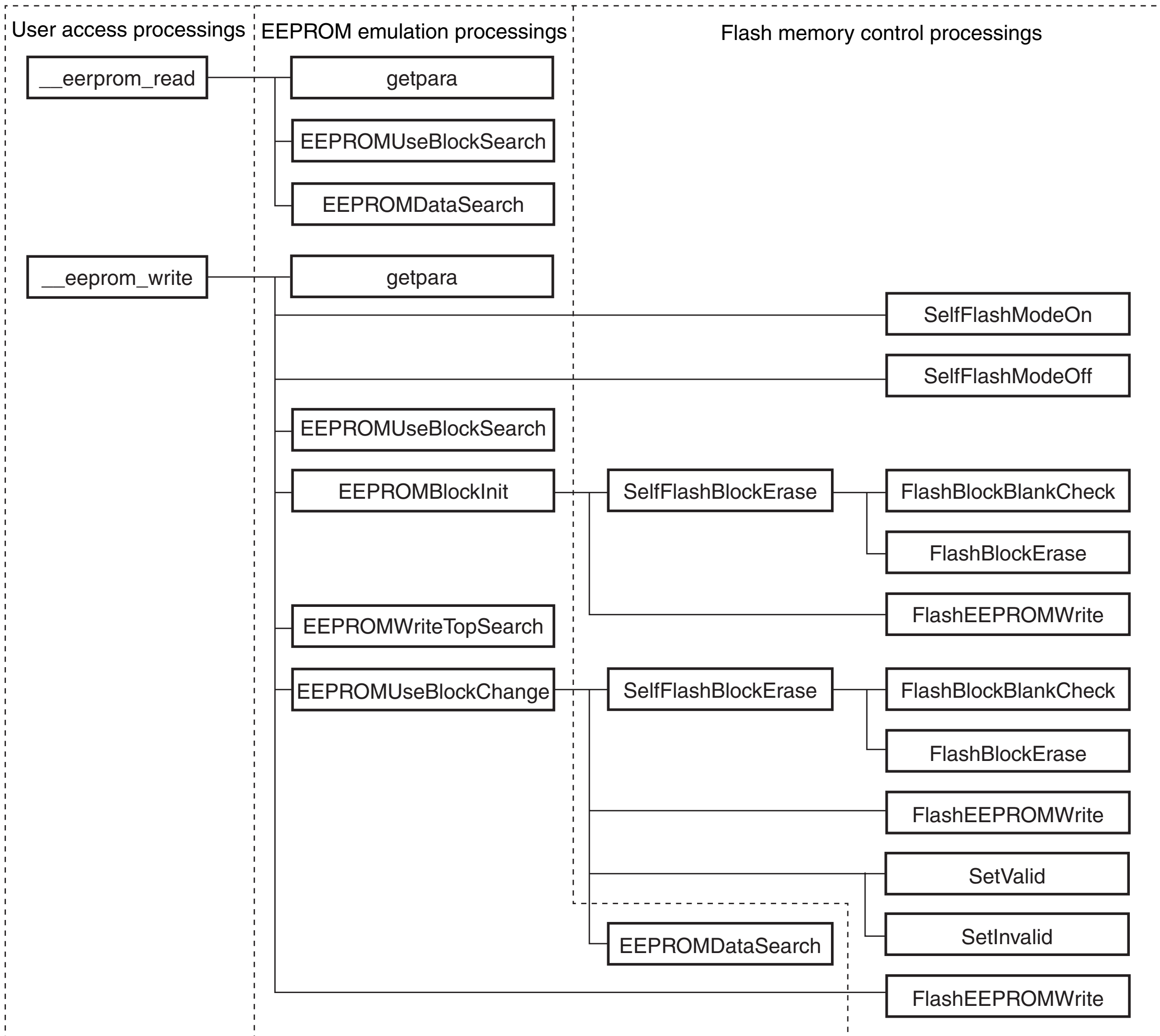
Reads the storage address of the latest data.



5.6 List of EEPROM Emulation Processings

A call tree of EEPROM emulation processings is shown below.

Figure 5-8. Call Tree



APPENDIX A REVISION HISTORY

A.1 Major Revisions in This Edition

Page	Description
p. 66 in old edition	Deletion of **APPENDIX A SAMPLE PROGRAM LIST (FIXED-LENGTH SINGLE-DATA METHOD)** from old edition
p. 84 in old edition	Deletion of **APPENDIX B SAMPLE PROGRAM LIST (FIXED-LENGTH MULTIPLE-DATA METHOD)** from old edition
CHAPTER 2 EEPROM EMULATION FUNCTION (FIXED-LENGTH SINGLE-DATA METHOD)	
pp. 9, 10	Modification of the number of erasures of one block, and the maximum number of rewrites in **2.1 Main Specifications for EEPROM Emulation**
p. 17	Addition of **caution 2** to, and modification of **Remark** in **Table 2-1 Conditions for EEPROM Emulation Operations**
p. 17	Addition of **2.3 How to Get the Sample Program**
CHAPTER 3 EEPROM EMULATION PROGRAM (FIXED-LENGTH SINGLE-DATA METHOD, ASSEMBLYLANGUAGE)	
p. 18	Modification of **Table 3-2 Resources**
p. 19	Modification of **3.3.1 (3) Number of erase retrials**
pp. 26, 28	Modification of **Table 3-10 Block Erase** and **Table 3-17 EEPROM Data Write Processing**
CHAPTER 4 EEPROM EMULATION FUNCTION (FIXED-LENGTH MULTIPLE-DATA METHOD)	
p. 37	Modification of the number of erasures of one block, and the maximum number of rewrites in **4.1 Main Specifications for EEPROM Emulation**
p. 45	Addition of **caution 2** to, and modification of **Remark** in **Table 4-1 Conditions for EEPROM Emulation Operations**
p. 45	Addition of **4.3 How to Get the Sample Program**
CHAPTER 5 EEPROM EMULATION PROGRAM (FIXED-LENGTH MULTIPLE-DATA METHOD, ASSEMBLY LANGUAGE)	
p. 46	Modification of **Table 5-2 Resources**
p. 47	Modification of **5.3.1 (3) Number of erase retrials**
pp. 54, 56	Modification of **Table 5-11 Block Erase** and **Table 5-18 EEPROM Data Write Processing**
APPENDIX A REVISION HISTORY	
p. 67	Addition of **A.2 Revision History of Preceding Editions**

<R>

A.2 Revision History of Preceding Editions

Here is the revision history of the preceding editions. Chapter indicates the chapter of each edition.

Edition	Description	Chapter
2nd Edition	Full modification of chapter.	**CHAPTER 2 EEPROM EMULATION FUNCTION (FIXED-LENGTH SINGLE-DATA METHOD)**
		CHAPTER 3 EEPROM EMULATION PROGRAM (FIXED-LENGTH SINGLE-DATA METHOD, ASSEMBLYLANGUAGE)
	Addition of chapter.	**CHAPTER 4 EEPROM EMULATION FUNCTION (FIXED-LENGTH MULTIPLE-DATA METHOD)**
		CHAPTER 5 EEPROM EMULATION PROGRAM (FIXED-LENGTH MULTIPLE-DATA METHOD, ASSEMBLY LANGUAGE)
	Full modification of chapter.	**APPENDIX A SAMPLE PROGRAM LIST (FIXED-LENGTH SINGLE-DATA METHOD)**
	Addition of chapter.	**APPENDIX B SAMPLE PROGRAM LIST (FIXED-LENGTH MULTIPLE-DATA METHOD)**
		APPENDIX C RIVISION HISTORY

Published by: NEC Electronics Corporation (http://www.necel.com/)
Contact: http://www.necel.com/support/