RL78/F13, F14

Safety Function

R01AN2165EJ0100
Rev.1.00
June 30, 2014

Introduction

This application note describes the safety functions provided in RL78/F13 and RL78/F14.

Target Devices

This application note is applied to RL78/F13 and RL78/F14.

Application conditions

Integrated development environment	Renesas Electronics Corporation CubeSuite+ V1.03.00	
Build tool	Renesas Electronics Corporation CA78K0R V1.50	
Compile option	Use standard startup routine	Yes [Normal]
	Type of memory model	Medium model
	Optimization option	Normal
Link option	On-chip debug option byte The value of 00C3H	 85H
	User option byte The value of 00C0H The value of 00C1H The value of 00C2H	 6EH 63H CAH
High-speed CRC operation	Range of CRC Operation 0000H - 03FFBH (16Kbytes - 4Bytes)	
	CRC result output address 03FFCH - 03FFDH (2Bytes)	
Input voltage	ANI0/AV_{REFP}=V_{DD} ANI1/AV_{REFM}=V_{SS}	5.0V

Contents

1. Overview of safety functions

The following safety functions are provided in the RL78/F13 and F14 microcontrollers in order to comply with the requirements of the IEC60730 and IEC61508 standards plus the ISO26262 automotive functional safety standard.

(1) Flash memory CRC operation function (high-speed CRC, general-purpose CRC)

This function detects data errors in the flash memory by performing CRC operations.

- High-speed CRC: A high-speed check can be executed on the entire code flash memory areas during the initialization routine by stopping the CPU operation.
- General-purpose CRC: This operation can be used for checking various data in addition to the code flash memory area while the CPU is operating.

(2) RAM-ECC function

This function can perform 2-bit error detection and 1-bit error correction.

(3) CPU stack pointer monitor function

This function monitors the stack pointer to detect underflows and overflows.

(4) Clock monitoring function

This function monitors the status of clock oscillation with a low-speed on-chip oscillator by sampling the main system clock (f_{MAIN}) and the PLL clock (f_{PLL}).

(5) RAM guard function

This function prevents RAM data from being rewritten when the CPU runs out of control.

(6) SFR guard function

This function prevents SFRs from being rewritten when the CPU runs out of control.

(7) Invalid memory access detection function

This function detects illegal accesses to invalid memory areas (such as areas where no memory is allocated and areas where access is restricted).

(8) Frequency detection function

This function detects the oscillation frequency using TAU0.

(9) A/D test function

This function is used to perform a self-check of A/D conversion by performing A/D conversion of the internal reference voltage.

(10) Digital output signal level detection function for I/O ports

When the I/O ports are in output mode, the output level of the pin can be read.

2. Flash memory CRC operation function (high-speed CRC)

2.1 Overview of the fault diagnostic function

The IEC60730 standard mandates checking of data in the flash memory, and recommends using CRC operations to do the checking. Faults or abnormalities in the code flash memory can be detected by comparing an expected CRC value which is prepared in advance with a CRC value obtained by the high-speed CRC operation.

The high-speed CRC operation is used before the user program is executed after reset release to check the data of the flash memory is correct. The high-speed CRC operation is executed for programs which are executed during the initial setting (initialization) routine so as to detect any fault (errors) in the entire code flash memory area (excluding the flash memory areas storing the expected CRC values).

The high-speed CRC operation can be executed by the program allocated on the RAM and in HALT mode of the main system clock, featuring its short time operation. (For example, 64 Kbytes of flash memory are checked in 512us when the operating clock is at 32MHz). The CRC generator polynomial used complies with “$X^{16}+X^{12}+X^{5}+1$” of CRC-16-CCITT.

The operation ranges are set by the FEA5 to FEA0 bits in the flash memory CRC control (CRC0CTL) register. The flash memory CRC operation result (PGCRCL) register is read with software, and errors in the entire flash memory areas can be detected by checking whether the read value matches the expected CRC value.

The expected CRC values can be calculated by using the development environment CubeSuite+ (refer to the CubeSuite+ User’s Manual). Calculate expected CRC values in advance and store the calculated values in a flash memory area in which the expected CRC values are to be stored (not an area for the high-speed CRC operation).

2.2 Descriptions of the fault diagnosis-related registers

The registers used for the high-speed CRC operation are described below.

(1) Flash memory CRC control register (CRC0CTL)

This register controls the operation of the high-speed CRC and specifies the operation range.

(2) Flash memory CRC operation result register (PGCRCL)

This register stores the high-speed CRC operation results.

The table below lists the setting examples of the related registers.

Table 2.1 Setting examples of flash memory CRC operation-related registers

Register	Set value	Description
Flash memory CRC control register (CRC0CTL)	03H	Operation range: 00000H - FFFBH (64Kbytes - 4bytes)
Flash memory CRC operation result register (PGCRCL)	-	Stores the results of high-speed CRC operation.

2.3 Program flowchart to execute the diagnostic function

Figure 2.1 is a flowchart showing the overview of the high-speed CRC operation.

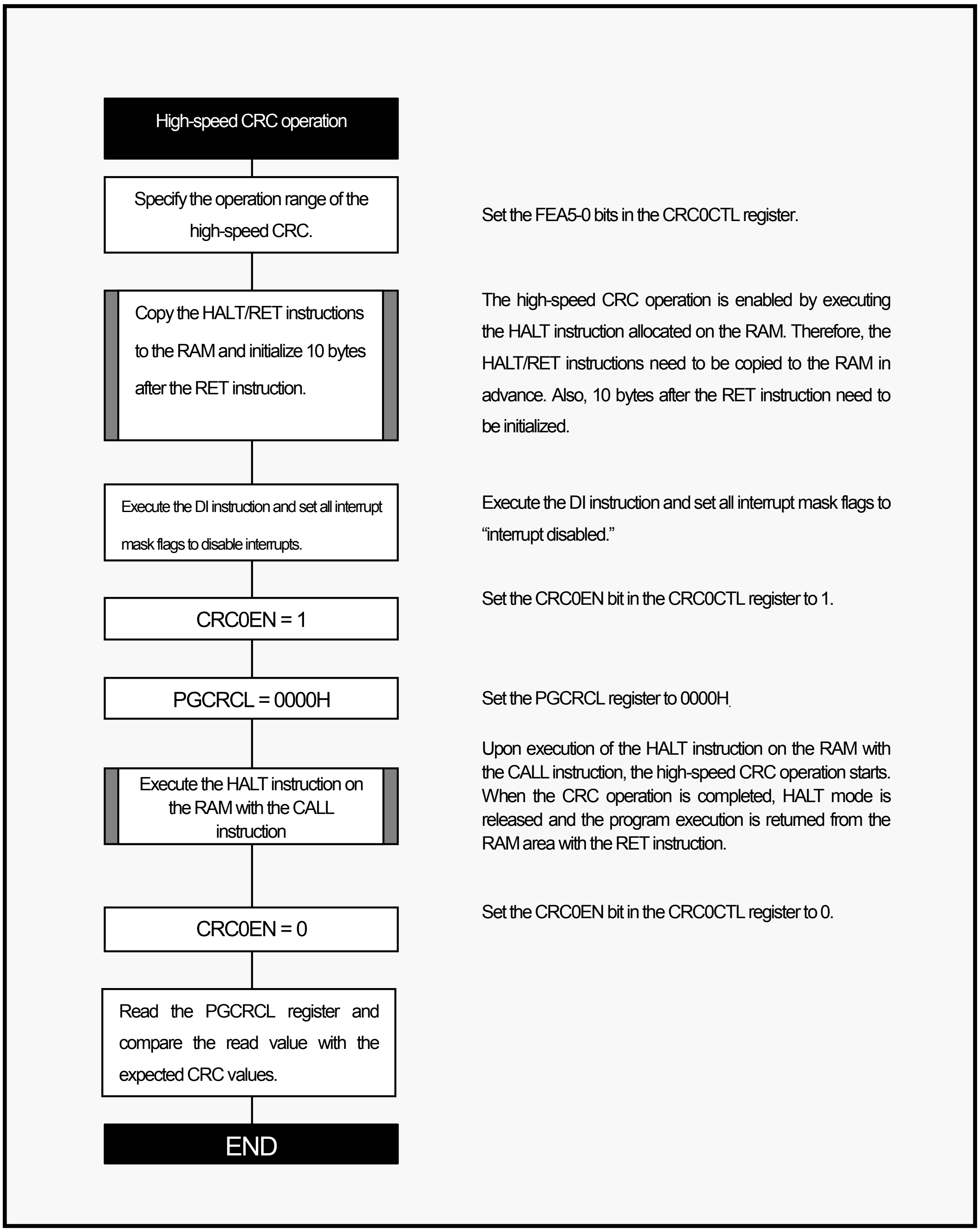

High-speed CRC operation
Specify the operation range of the high-speed CRC.
Set the FEA5-0 bits in the CRC0CTL register.
Copy the HALT/RET instructions to the RAM and initialize 10 bytes after the RET instruction.
The high-speed CRC operation is enabled by executing the HALT instruction allocated on the RAM. Therefore, the HALT/RET instructions need to be copied to the RAM in advance. Also, 10 bytes after the RET instruction need to be initialized.
Execute the DI instruction and set all interrupt mask flags to disable interrupts.
Execute the DI instruction and set all interrupt mask flags to "interrupt disabled."
CRC0EN = 1
Set the CRC0EN bit in the CRC0CTL register to 1.
PGCRCL = 0000H
Set the PGCRCL register to 0000H.
Execute the HALT instruction on the RAM with the CALL instruction
Upon execution of the HALT instruction on the RAM with the CALL instruction, the high-speed CRC operation starts. When the CRC operation is completed, HALT mode is released and the program execution is returned from the RAM area with the RET instruction.
CRC0EN = 0
Set the CRC0EN bit in the CRC0CTL register to 0.
Read the PGCRCL register and compare the read value with the expected CRC values.
END

Figure 2.1 Program flowchart

2.4 Sample program to execute the diagnostic function

The following is a sample program for the high-speed CRC operation.

List 2-1 Sample program (High-speed CRC operation)

```
uint16_t sf_highspeedCRC( uint8_t crc_range )
{
        /*      Flash Memory CRC Control Register(CRC0CTL)
                b7      : Control of CRC ALU operation.
                b6      : Reserved set to 0.
                b5 :b0  : CRC operation range. (FEA5-0) */
        CRC0CTL         = (crc_range & (uint8_t)SF_CRC0CTL_MASK);                    (1)

        CRC0EN          = BIT_SET;      /* Start the operation according to HALT     (2)
                                           instruction execution. */

        PGCRCL          = 0x0000U;      /* Clear the Flash Memory CRC Operation      (3)
                                           Result Register. */

        crc_process();                  // Call the halt and ret codes on the RAM.   (4)

        CRC0EN          = BIT_CLR;      // Stop the operation.                       (5)

        return  (uint16_t)PGCRCL;       // return a high-speed CRC operation result. (6)
}
```

(1) Set the CRC operation range of the flash memory to the CRC0CTL register.

In this example, the CRC operation is performed for the area specified by an argument.

The followings are the values that can be set and their corresponding behaviors.

Table 2.2 CRC operation range

Macro definition	Value	CRC operation range
SF_HS_CRC_16K	0	0H - 3FFBH
SF_HS_CRC_32K	1	0H - 7FFBH
SF_HS_CRC_48K	2	0H - BFFBH
SF_HS_CRC_64K	3	0H - FFFBH
SF_HS_CRC_80K	4	0H - 13FFBH
SF_HS_CRC_96K	5	0H - 17FFBH
SF_HS_CRC_112K	6	0H - 1BFFBH
SF_HS_CRC_128K	7	0H - 1FFFBH
SF_HS_CRC_144K	8	0H - 23FFBH
SF_HS_CRC_160K	9	0H - 27FFBH
SF_HS_CRC_176K	10	0H - 2BFFBH
SF_HS_CRC_192K	11	0H - 2FFFBH
SF_HS_CRC_208K	12	0H - 33FFBH
SF_HS_CRC_224K	13	0H - 37FFBH
SF_HS_CRC_240K	14	0H - 3BFFBH
SF_HS_CRC_256K	15	0H - 3FFFBH

(2) Set the CRC0EN bit to 1 to enter the waiting-for high-speed CRC trigger state (execution of the HALT instruction on RAM).

(3) Set the flash memory CRC operation result register (PGCRCL) to 0000H.

(4) Call a processing function including the HALT instruction allocated on RAM by a CALL instruction.

This example assumes that the program codes have been expanded on RAM.

When the CRC operation is completed, HALT mode is released and the program execution is returned by executing the RET instruction.

(5) Set the CRC0EN bit to 0 to stop the high-speed CRC operation.

(6) Read the PGCRCL register to obtain the high-speed CRC operation result.

In this example, the operation results are treated as a return value of a function.

Note 1: In this example, the DI instruction is executed and also interrupts of all interrupt mask flag registers are disabled.

Note 2: In this example, the CPU operates in main RUN mode.

List 2-2 Sample program (RAM execution program)

```
        PUBLIC  _crc_process

        RAMCODE      CSEG   AT      0FF400H; locate in FF400H.(sample)

_crc_process:
        halt                                ; Trigger of high-speed CRC operation.     (1)
        ret                                 ;                                          (2)

        NOP                                 ; Clear 10 bytes.                          (3)
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP

        end
```

(1) The high-speed CRC operation starts upon execution of the HALT instruction allocated on RAM.

(2) When the high-speed CRC operation is completed, HALT mode is released and the program execution is returned by calling with the RET instruction.

(3) The CPU of the RL78 pre-reads the instruction code. Therefore, to perform RAM fetch, the RAM fetch area + 10bytes need to be initialized. Also, instructions (e.g. NOP) need to be allocated to 10bytes after the RET instruction.

Note: In this example, the codes are allocated at 0FF400$_H$ based on according to CSEG AT.

List 2-3 Sample program (high-speed CRC operation)

```
extern      int                 _rcopy(int16_t);

void main( void )
{
            int16_t             rom_ret;
            uint8_t             int_mask[ USER_INT_MASK_REGISTERS ];
            uint16_t            crc_result;
            uint16_t __far*     rom_crc     =
                                        (uint16_t __far * )USER_FLASH_MEMORY_CRC_SET_ADDRESS;          (1)

            DI();                                               // Disable interrupt.                  (2)
            rom_ret     = _rcopy(1U);                           // ROMtoRAM copy.                      (3)

            /* flash memory CRC check. */
            push_interrupt_mask_flags((uint8_t*)&int_mask);     // MKxx register all mask.             (4)
            crc_result=   sf_highspeedCRC( SF_HS_CRC_16K );     // High-Speed CRC                      (5)
            pop_interrupt_mask_flags((uint8_t*)&int_mask);      // MKxx register all return.           (6)

            if ( *rom_crc != crc_result ) {                     // Check the Flash ROM CRC code.
                        USER_FUNC_FLASH_MEMORY_CRC_ERROR();     // User own code.
            }
            .
            .
            .
```

(1) USER_FLASH_MEMORY_CRC_SET_ADDRESS indicates the address at which the expected CRC values are stored.

(2) Execute the DI instruction.

(3) Expand (copy) the RAM execution program to the RAM area with the _rcopy function.

(4) The push_interrupt_mask_flags function saves all the interrupt mask flag registers and disables interrupts.

(5) Call the sf_highspeedCRC function to obtain the result of the high-speed CRC operation.

(6) The pop_interrupt_mask_flags function restores all the saved interrupt mask flag registers.

Note: The expected CRC values of the corresponding areas can be easily embedded into a HEX file of the corresponding areas with the object converter function provided with the integrated development environment CubeSuite+.
Regarding the object converter, refer to “CubeSuite+ RL78 Help”.

2.5 Cautions when using the high-speed CRC operation function

The followings are the cautions when using the high-speed CRC operation function.

(1) To execute the high-speed CRC operation, disable interrupts (a state in which the DI instruction is executed) and also set all the interrupt mask flag registers to disable interrupts.

(2) The CPU of the RL78 pre-reads the instruction codes. If a program is executed from the RAM area, the RAM execution area + the 10-byte area need to be initialized at first.

The program to be executed on the RAM area can be processed with the _rcopy function expanding the RAM execution program ROMized by the ROMization processing, which is provided by the integrated development environment CubeSuite+, to the RAM area.

Regarding the ROMization processing and the _rcopy function, refer to "CubeSuite+ RL78 Help".

(3) The following cautions may be needed according to the CPU operating frequency.

For example, when the internal high-speed oscillation clock is 32MHz, the operating voltage is 2.7V or higher, but the POR (power on reset) voltage is 1.56V (Typ.). Therefore, the expected CRC operation values cannot be obtained if high-speed CRC operation is carried out before the operating voltage reaches 2.7V. To prevent this, execute the processing with the LVD function until the voltage reaches the specified value.

For details of the LVD function, refer to "RL78/F13, F14 User's Manual: Hardware."

(4) The addresses exceeding the upper address of mirroring overlap the RAM area. When the expected values of the high-speed CRC operation are allocated to this area, reference to the expected address is unavailable by a normal access.

To allow the reference to the address to which the expected high-speed CRC operation value is allocated, program software to execute access to far areas.

(5) With the CubeSuite+, the expected CRC values will be embedded in a HEX file (the values are not embedded in a load module file).

When on-chip debugging is enabled (used), the result of the high-speed CRC operation does not agree with the expected CRC value obtained by CubeSuite+.

2.6 Program size and execution time

Size

- sf_highspeedCRC function: 29 bytes (excluding the RAM execution program)

Execution time

- when the CRC operation range is 16Kbytes: 0.520ms
- when the CRC operation range is 32Kbytes: 1.032ms
- when the CRC operation range is 48Kbytes: 1.546ms
- when the CRC operation range is 64Kbytes: 2.058ms
- when the CRC operation range is 80Kbytes: 2.572ms
- when the CRC operation range is 96Kbytes: 3.084ms

Conditions

- Operation frequency: high-speed on-chip oscillator clock (f_{IH}:8MHz)
- Memory model: medium

3. CRC operation function (general-purpose CRC)

3.1 Overview of the fault diagnostic function

In order to guarantee safety during system's operation, the IEC61508 standard mandates the checking of data even while the CPU is operating.

With the general-purpose CRC operation function, CRC operation is available as a peripheral function in the main system clock operation mode or subsystem clock operation mode while the CPU is operating. This general-purpose CRC can be used for checking various data in addition to data in the code flash memory area.

The CRC generator polynomials used for the general-purpose CRC operation function are:

$X^{16}+X^{12}+X^{5}+1$ of CRC16-CCITT and $X^{4}+X^{3}+X^{2}+1$ which conforms to the Single Edge Nibble Transmission (SENT) protocol.

3.2 Description of the fault diagnosis-related registers

The registers used for the general-purpose CRC operation are described below.

(1) CRC operation mode control register (CRCMD)

This CRCMD register selects the general-purpose CRC operation mode.

(2) CRC input register (CRCIN)

This CRCIN register is an 8-bit register to set the CRC operation data for general-purpose CRC.

(3) CRC data register (CRCD)

This CRCD register stores the results of general-purpose CRC operation.

The table below lists the setting examples of the related registers.

Table 3.1 Setting examples of general-purpose CRC operation-related registers

Register	Set value	Description
CRC operation mode control register (CRCMD)	00H	CRC generator polynomial: $X^{16} + X^{12} + X^{5} + 1$
CRC input register (CRCIN)	-	Sets the general-purpose CRC data.
CRC data register (CRCD)	-	Stores the results of the general-purpose CRC operation.

3.3 Program flowchart to execute the diagnostic function

Figure 3.1 is a flowchart showing the overview of the general-purpose CRC operation function.

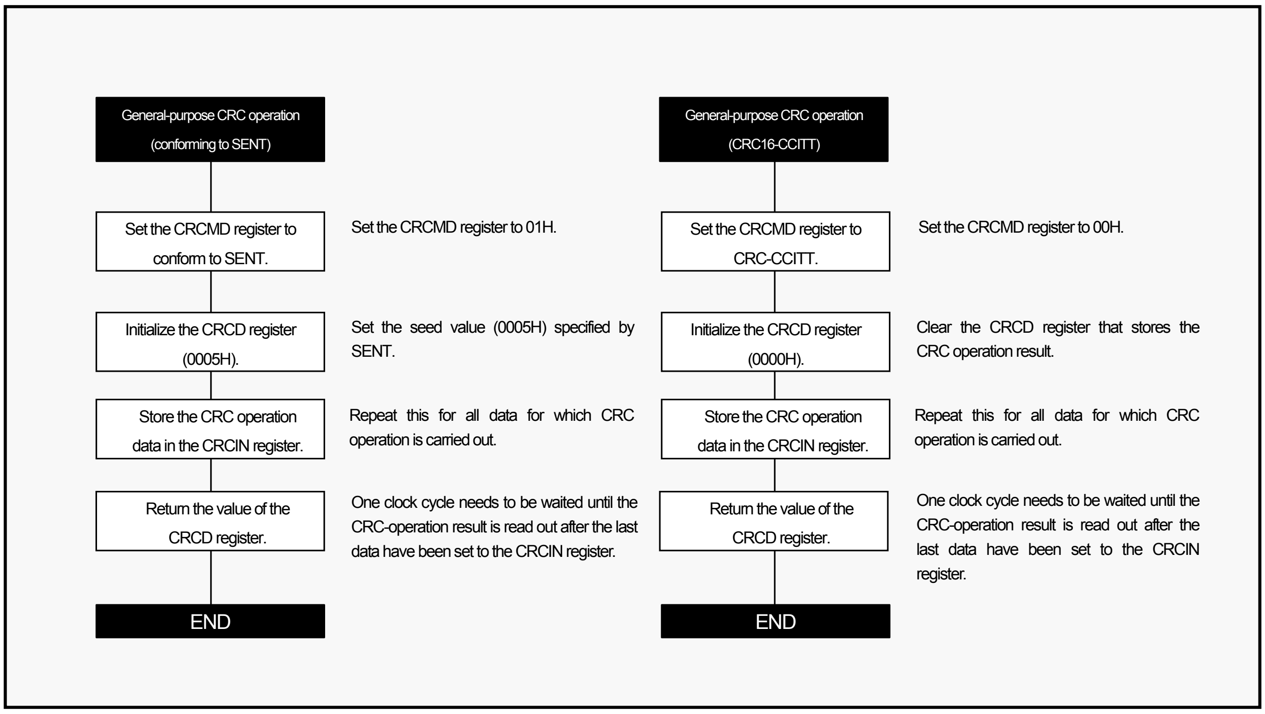

General-purpose CRC operation (conforming to SENT)
Set the CRCMD register to conform to SENT.
Set the CRCMD register to 01H.
Initialize the CRCD register (0005H).
Set the seed value (0005H) specified by SENT.
Store the CRC operation data in the CRCIN register.
Repeat this for all data for which CRC operation is carried out.
Return the value of the CRCD register.
One clock cycle needs to be waited until the CRC-operation result is read out after the last data have been set to the CRCIN register.
END
General-purpose CRC operation (CRC16-CCITT)
Set the CRCMD register to CRC-CCITT.
Set the CRCMD register to 00H.
Initialize the CRCD register (0000H).
Clear the CRCD register that stores the CRC operation result.
Store the CRC operation data in the CRCIN register.
Repeat this for all data for which CRC operation is carried out.
Return the value of the CRCD register.
One clock cycle needs to be waited until the CRC-operation result is read out after the last data have been set to the CRCIN register.
END

Figure 3.1 Program flowchart

3.4 Sample program to execute the diagnostic function

The following is a sample program for the general-purpose CRC operation function.

List 3-1 Sample Program (general-purpose CRC operation)

```
uint16_t sf_generalCRC( uint8_t crc_mode , uint8_t __far *inadr , uint16_t data_len )
{
        uint16_t  lp_cont;

        if ( crc_mode == (uint8_t)SF_CRC_MODE_SENT ) {
                CRCMD           = (uint8_t)SF_CRC_MODE_SENT;                                (1)
                CRCD            = 0x0005U;              // Set the CRC data register.       (2).
                for ( lp_cont=0U; lp_cont<data_len; lp_cont++ ) {
                        CRCIN   = (*inadr & 0x0FU);     // Set the CRC input register.      (3)
                        inadr++;
                }
        } else {
                CRCMD           = (uint8_t)SF_CRC_MODE_CRC16CCITT;                          (4)
                CRCD            = 0x0000U;              // Clear the CRC data register.     (5)
                for ( lp_cont=0U; lp_cont<data_len; lp_cont++ ) {
                        CRCIN   = *inadr++;             // Set the CRC input register.      (6)
                }
        }
        return  (uint16_t)CRCD;         /* return a general-purpose CRC operation result. */ (7)
}
```

(1) Set the CRCMD register to 01H to set the operation mode conforming to SENT.

The function of this sample program is specified by arguments.

(2) Set 0005H to the CRCD register as the initial value of the CRC operation.

(3) Store 4-bit data to be used for the CRC operation to the CRCIN register.

In this sample program, the address at which data to be used for CRC operation is stored and the number of operations are both specified by arguments.

(4) To set the operation mode to CRC-16-CCITT, set 00H to the CRCMD register.

The function of this sample program is specified by arguments.

(5) Set 0000H to the CRCD register as the initial value of the CRC operation.

(6) Store 8-bit data to be used for the CRC operation in the CRCIN register.

In this sample program, the address, at which data to be used for CRC operation is stored, and the number of operations are both specified by arguments.

(7) Return the value of the CRCD register as the operation result.

3.5 Program size and execution time

Size

- sf_generalCRC function: 95bytes

Execution time

- when the CRC operation range is 6bytes (CRC-16-CCITT): 5.38μs

Conditions:

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

4. RAM-ECC function

4.1 Overview of the fault diagnostic function

ISO 26262 has decided the ASIL levels according to the area ratio and fault coverage of each function.

When writing to the RAM area, the RAM-ECC function first generates an ECC code (4 bits) and a parity bit from the data to be written (8 bits), and then writes the generated data to the RAM area as 13-bit data.

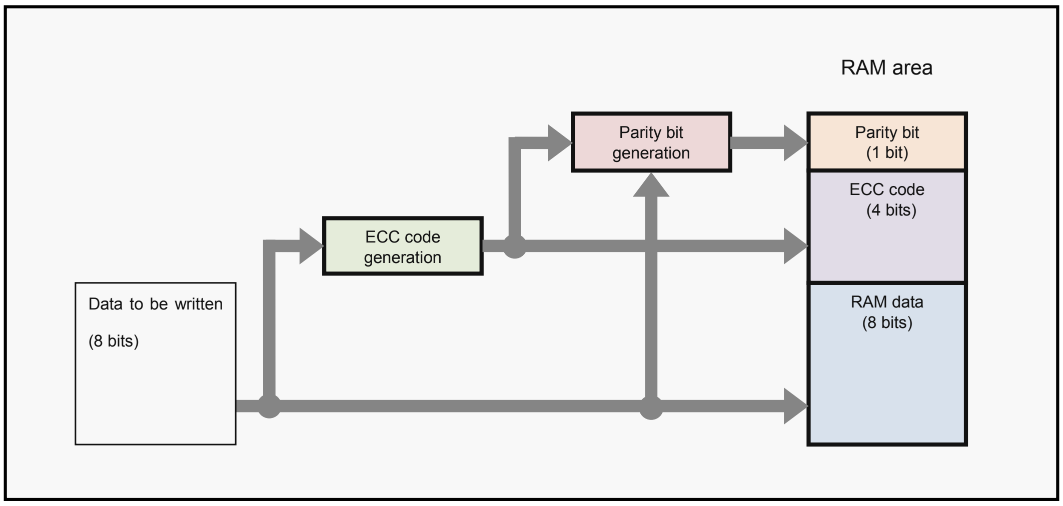


Figure 4.1 Structure of RAM-ECC Function (at write access)

When reading the RAM area, the read data (8 bits), the ECC code (4 bits) and the parity bit (1 bit) are checked to detect a bit error. If a bit error exists, a bit error detection interrupt (INTRAM) is generated and the RAM address where the bit error has occurred is stored in the error address store register (ERADR) register.

If the error detected is one bit, the data to be read will be corrected.

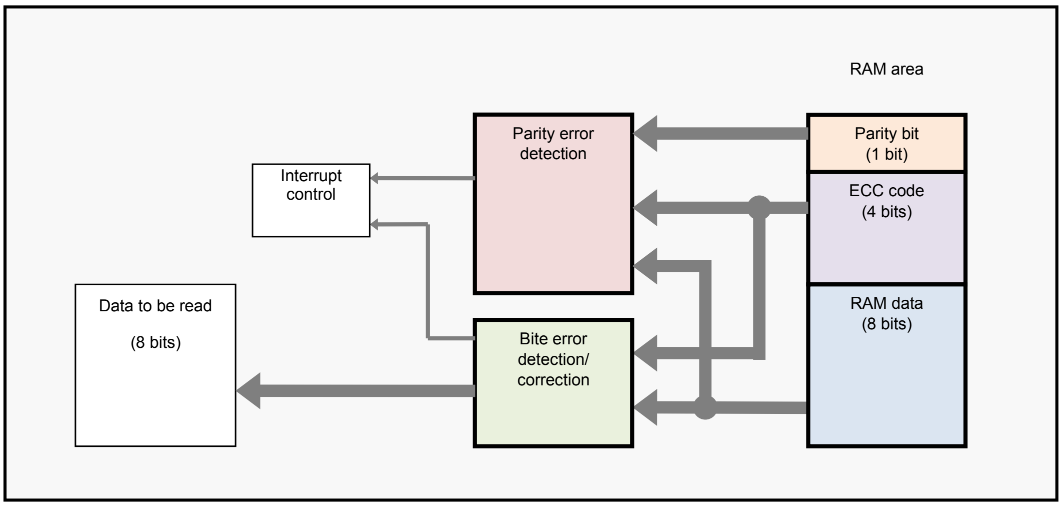


Figure 4.2 Structure of RAM-ECC function (at read access)

The table below shows the operation matrix for bit errors.

Table 4.1 RAM-ECC operation matrix

Types of error bit	Error position	1-bit error interrupt	Error address	Bit error interrupt request	Bit error flag	Parity error	Bit error detection	Bit error correction
No error bit	–	-	No address (previous value)	-	0	-	-	-
1-bit error	Parity	Disabled	No address (previous value)	-	0	Occurs	-	-
	ECC	Disabled	No address (previous value)	-	0	Occurs	✓	-
	Data	Disabled	No address (previous value)	-	0	Occurs	✓	✓
	Parity	Enabled	No address (previous value)	-	0	Occurs	-	-
	ECC	Enabled	Retained	✓	0	Occurs	✓	-
	Data	Enabled	Retained	✓	0	Occurs	✓	✓
2-bit error	Parity +ECC	-	Retained	✓	1	-	✓	-
	Parity +data	-	Retained	✓	1	-	✓	✓
	ECC+data	-	Retained	✓	1	-	✓	Wrong correction
	ECC	-	Retained	✓	1	-	✓	Wrong correction
	Data	-	Retained	✓	1	-	✓	No correction or wrong correction.

Note:

✓ :available

- :not available

4.2 RAM-ECC test mode

RAM-ECC test mode is used as the self-checking for the RAM-ECC function.

By setting the operation mode of the RAM-ECC function to test mode, it becomes possible to inject error data to each of a parity bit, the 4-bit ECC code and 8-bit data, all of which are included in the RAM data, in order to confirm that error bit detection or error bit correction is properly performed.

When writing to the RAM in normal operation mode, the ECC code is generated from data to be written, and the parity bit is generated from both the data to be written and the generated ECC code.

Meanwhile, in RAM-ECC test mode, data is written to the RAM after the values of bits of the data to be written (the parity bit, 4-bit ECC code, and 8-bit data) are inverted according to the values set to the write data inversion register (ECCDWRVR).

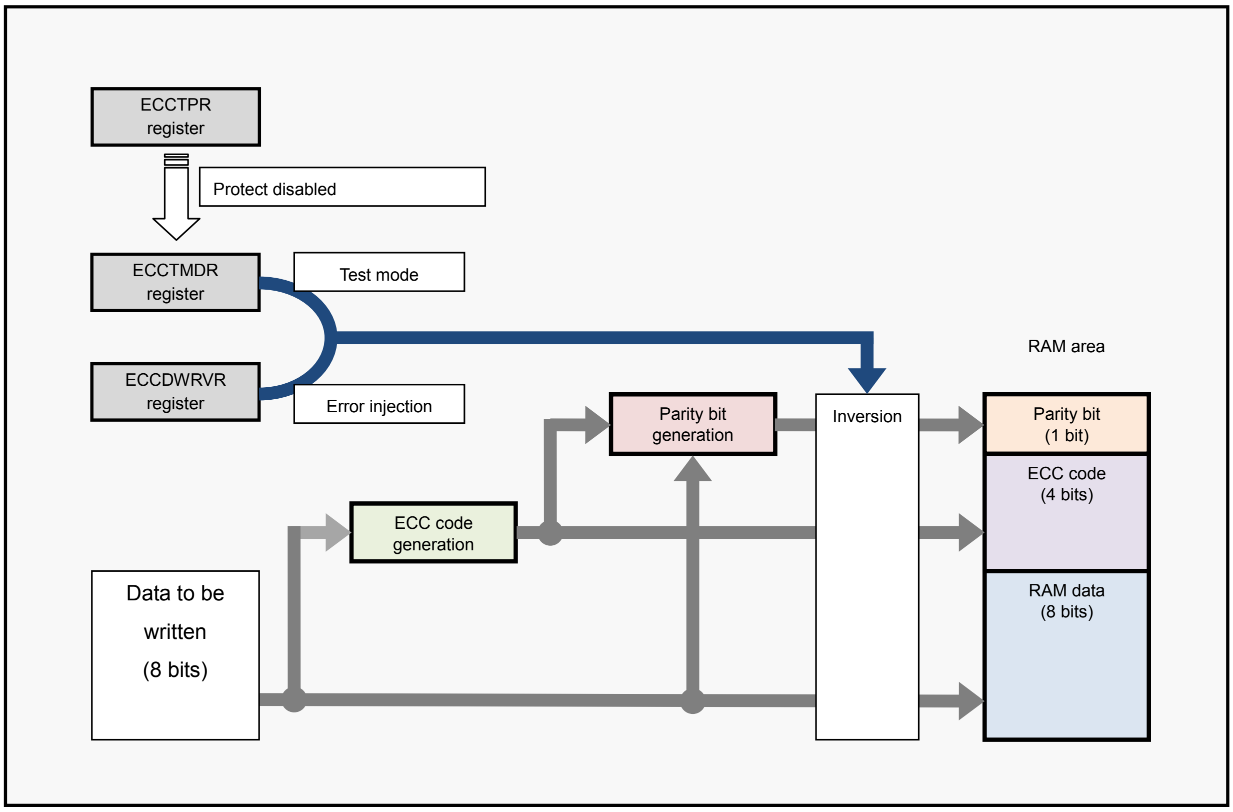


Figure 4.3 Structure of RAM-ECC test mode function

Accordingly, when the data is read from the same RAM area, bit error detection or bit error correction is carried out according to the injected error data (inverted written values). Therefore, when the user system operates in RAM-ECC test mode, the following self checking can be performed for the RAM-ECC function.

- Detects a 1-bit or 2-bit error.
- Confirms that the detected 1-bit or 2-bit error matches the injected error data.
- Confirms that the RAM address where the 1-bit or 2-bit error has been detected is the same with the address where the error data had been injected.
- Confirms that when a 1-bit error is detected the RAM data read have been properly corrected.
- Confirms that when a 2-bit error is detected the RAM data read have been properly corrected. (This case applies to the followings only: a 2-bit error caused by a parity bit and ECC bits or by a parity bit and data bits).

4.3 Description of the fault diagnosis-related registers

The registers used for the RAM-ECC function are described below.

(1) Error address storage register (ERADR)

This ERADR register stores the address at which a bit error detection interrupt request (INTRAM) has been generated.

(2) 1-bit error detection interrupt enable register (ECCIER)

This ECCIER register specifies whether generation of the INTRAM interrupt is enabled or disabled when a 1-bit error is detected.

(3) Bit error detection register (ECCER)

This ECCER register confirms whether the detected bit error is a 1-bit error or 2-bit error.

(4) ECC test protect register (ECCTPR)

This ECCTPR register enables/disables the protection function of the ECCTMDR register that switches the operation mode between normal mode and test mode.

(5) ECC test mode register (ECCTMDR)

This ECCTMDR register sets the RAM-ECC function to test mode.

(6) Write data inversion register (ECCDWRVR)

This ECCDWRVR register performs error data injection for the RAM-ECC function operating in test mode.

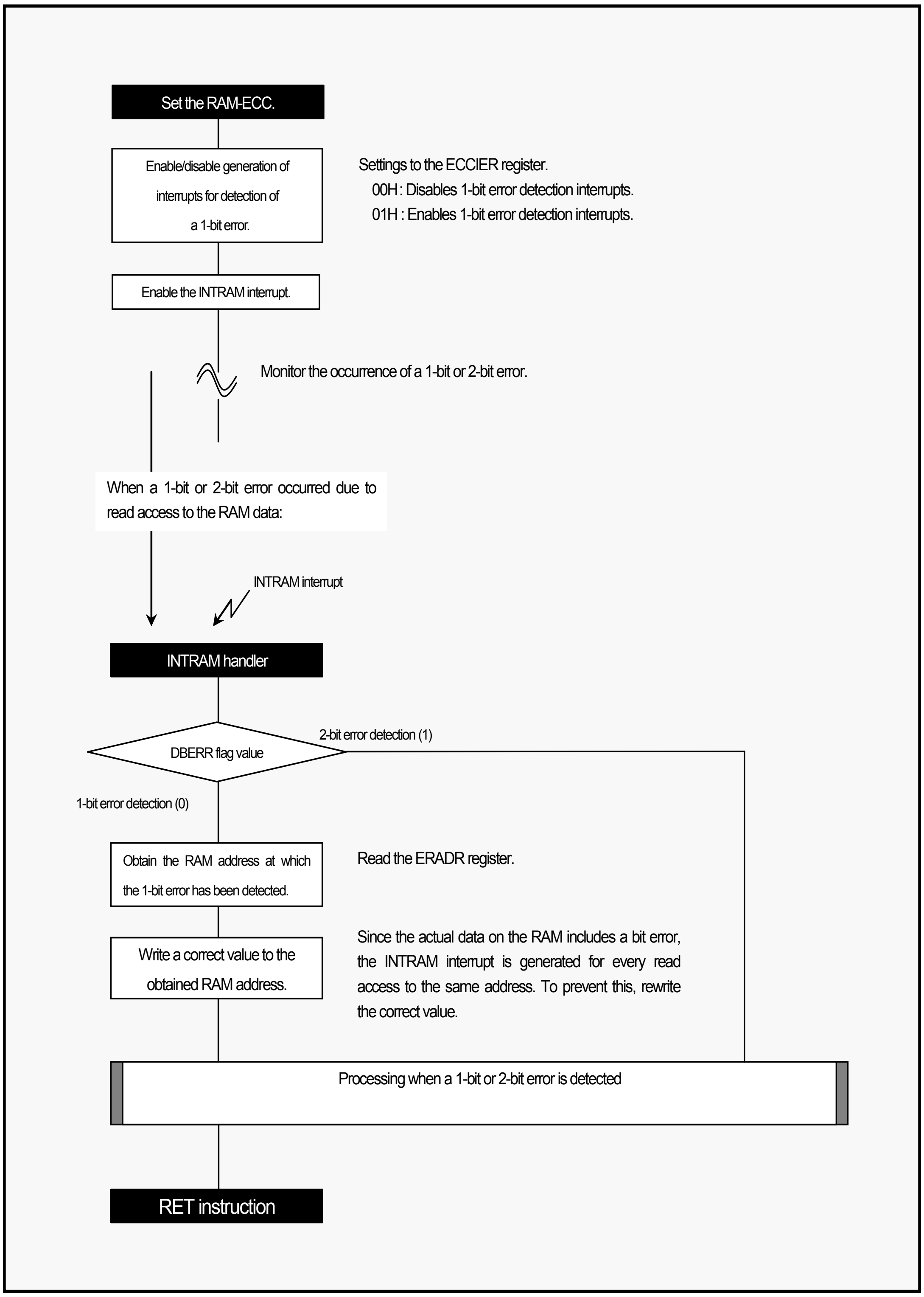
The table below shows the setting examples of the related registers.

Table 4.2 Setting examples of RAM-ECC-related registers

Register	Set value	Description
1-bit error detection interrupt enable register (ECCIER)	00H	Disables interrupts for the detection of a 1-bit error.
Error address store register (ERADR)	-	Stores the address of a 1-bit or 2-bit error detected.
Bit error detection register (ECCER)	-	Stores the bit error detected.
ECC test protect register (ECCTPR)	07H	Enables the access to the ECCTMDR register.
ECC test mode register (ECCTMDR)	01H	Sets the RAM-ECC function to test mode.
Write data inversion register (ECCDWRVR)	1001H	At write access to RAM, write the inverted values of a parity bit and bit 0 of data.

4.4 Program flowchart to execute the diagnostic function

Figure 4.4 is a flowchart showing the overview of the RAM-ECC function.



Set the RAM-ECC.
Enable/disable generation of interrupts for detection of a 1-bit error.
Settings to the ECCIER register.
00H : Disables 1-bit error detection interrupts.
01H : Enables 1-bit error detection interrupts.
Enable the INTRAM interrupt.
Monitor the occurrence of a 1-bit or 2-bit error.
When a 1-bit or 2-bit error occurred due to read access to the RAM data:
INTRAM interrupt
INTRAM handler
DBERR flag value
2-bit error detection (1)
1-bit error detection (0)
Obtain the RAM address at which the 1-bit error has been detected.
Read the ERADR register.
Write a correct value to the obtained RAM address.
Since the actual data on the RAM includes a bit error, the INTRAM interrupt is generated for every read access to the same address. To prevent this, rewrite the correct value.
Processing when a 1-bit or 2-bit error is detected
RET instruction

Figure 4.4 Program flowchart

Figure 4.5 is a flowchart showing the overview of the processing in ECC-RAM test mode.

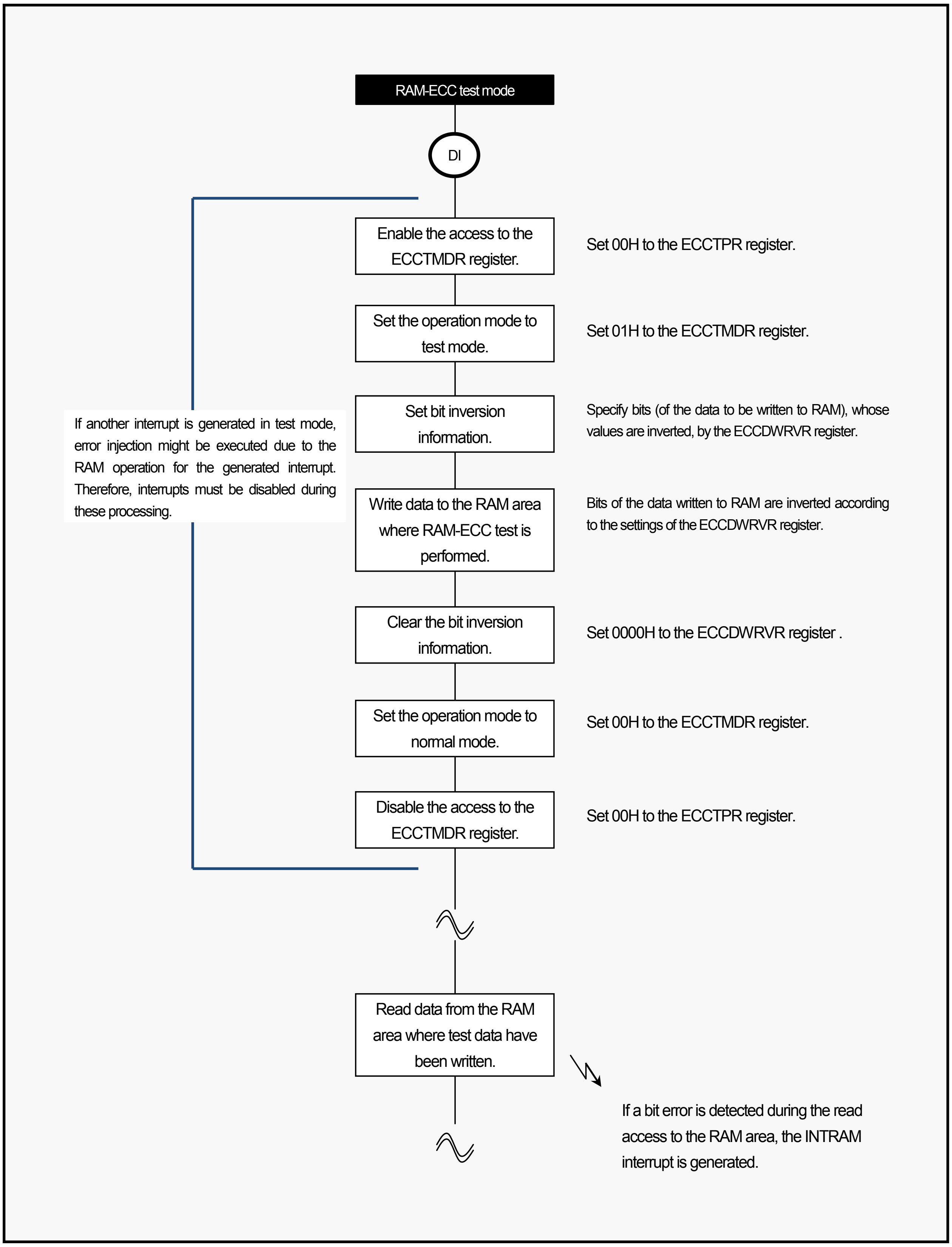

RAM-ECC test mode
DI
Enable the access to the ECCTMDR register.
Set 00H to the ECCTPR register.
Set the operation mode to test mode.
Set 01H to the ECCTMDR register.
Set bit inversion information.
Specify bits (of the data to be written to RAM), whose values are inverted, by the ECCDWRVR register.
If another interrupt is generated in test mode, error injection might be executed due to the RAM operation for the generated interrupt. Therefore, interrupts must be disabled during these processing.
Write data to the RAM area where RAM-ECC test is performed.
Bits of the data written to RAM are inverted according to the settings of the ECCDWRVR register.
Clear the bit inversion information.
Set 0000H to the ECCDWRVR register .
Set the operation mode to normal mode.
Set 00H to the ECCTMDR register.
Disable the access to the ECCTMDR register.
Set 00H to the ECCTPR register.
Read data from the RAM area where test data have been written.
If a bit error is detected during the read access to the RAM area, the INTRAM interrupt is generated.

Figure 4.5 Program flowchart

4.5 Sample program to execute the diagnostic function

The following is a sample program for the RAM-ECC function.

List 4-1 Sample program (Enable/disable 1-bit error detection interrupts.)

```
void sf_ram_ecc_set_int_mode( uint8_t int_mode )
{
        if ( int_mode==SF_RAMECC_ENABLE_1BIT_ERROR_INTERRUPT ) {
                ECCIER = (uint8_t)SF_RAMECC_ENABLE_1BIT_ERROR_INTERRUPT;        (1)
        } else {
                ECCIER = (uint8_t)SF_RAMECC_DISABLE_1BIT_ERROR_INTERRUPT;       (2)
        }
}
```

(1) Setting 01H to the ECCIER register enables the generation of the 1-bit error detection interrupt.

(2) Setting 00H to the ECCIER register disables the generation of the 1-bit error detection interrupt.

List 4-2 Sample program (INTRAM interrupt enabled)

```
void sf_ram_ecc_start( void )
{
        RAMIF           = BIT_CLR;      // Clear the INTRAM interrupt request flag.     (1)
        RAMMK           = BIT_CLR;      // Clear the INTRAM interrupt mask flag.        (2)
}
```

(1) Clear the INTRAM interrupt request flag.

(2) Clear the INTRAM interrupt mask flag to enable the INTRAM interrupt.

List 4-3 Sample program (Clear bit error detection flag.)

```
void sf_ram_ecc_clear_bit_error( void )
{
        ECCER  = 0x00U;                                                         (1)
}
```

(1) Set 0000H to the ECCER register.

List 4-4 Sample program (INTRAM interrupt handler)

```
__interrupt void sf_INTRAM_int( void )
{
        uint8_t             ecc_error_address_data;

        /* bit error infomation read. */
        if ( ECCER == SF_RAMECC_2BIT_ERROR_DETECT ) {                               (1)
                NOP();
                /* Start a user's program in case of the INTRAM (2bit error detect) interrupt. */
                /* Finish a user's program in case of the INTRAM (2bit error detect) interrupt. */
        } else {
                /* Write back process. */
                ecc_error_address_data          = *(uint8_t __far * )(ERADR | 0xf0000);     (2)

                NOP();                                          // RAMIF set wait.
                NOP();
                RAMIF                           = BIT_CLR;      // clear RAMIF.         (3)

                *(uint8_t __far * )(ERADR | 0xf0000)  = ecc_error_address_data;             (4)
                /* Start a user's program in case of the INTRAM (1bit error detect) interrupt. */
                /* Finish a user's program in case of the INTRAM (1bit error detect) interrupt. */
        }
}
```

(1) When the value of the ECCER register is 00H, a 1-bit error has been detected. Meanwhile, when the value of the ECCER register is 01H, a 2-bit error has been detected.

(2) When a 1-bit error is detected, the RAM address of the 1-bit error, which is indicated by the ERADR register, will be read out.

(3) To prevent the INTRAM interrupt from being generated once again after the step (2) above, wait two clock cycles and then clear the INTRAM interrupt request flag (RAMIF).

(4) Write back the read data to the same address. This prevents the generation of the INTRAM interrupt at the subsequent read access to the same address.

List 4-5 Sample program (Error data injection in test mode)

```
void sf_ram_ecc_test_mode(      uint8_t   __far*    ram_address ,
                                uint8_t             write_data ,
                                uint16_t            error_pattern )
{
            ECCTPR          = 0x07U;        // Enable the access to the ECCTMDR register.       (1)
            ECCTMDR         = 0x01U;        // Set RAM-ECC function in test mode.               (2)

                                            // Set the bit-error pattern.
            ECCDWRVR        = (error_pattern & (uint16_t)SF_ECCDWRVR_MASK);                     (3)

            *ram_address    = write_data;   // Data and bit-error.                              (4)

            ECCDWRVR        = 0x0000U;      // Clear the bit-error pattern.                     (5)

            ECCTMDR         = 0x00U;        // Set RAM-ECC function in normal mode.             (6)
            ECCTPR          = 0x00U;        // Enable the access to the ECCTMDR register.       (7)
}
```

(1) Set 07H to the ECCTPR register to enable the access to the ECCTMDR register.

(2) Setting 01H to the ECCTMDR register enables the operation in RAM-ECC test mode.

(3) Specify the bits of the parity bit, ECC code and data, which are to be inverted (error data are injected), by the ECCDWRVR register.

In this example, the bits to which error data are to be injected are specified by arguments.

(4) Write data to the RAM area where error data are to be injected.

In this example, the address, at which the data is written (the RAM address where the self-checking is executed), and the data to be written to the RAM are both indicated by arguments.

(5) Set 0000H to the ECCDWRVR register.

(6) Set 00H to the ECCTMDR register to return to normal mode.

(7) Set 00H to the ECCTPR register to protect the ECCTMDR register.

Note: This processing is performed when the DI instruction is executed (interrupt disabled).

4.6 Cautions when using the RAM-ECC function

The followings are the cautions when using the RAM-ECC function.

(1) With the 1-bit/2-bit error detection/correction function, the RAM data read are corrected. However, the actual RAM data remain uncorrected.

Every time the address including a bit error is read, a bit error detection interrupt (INTRAM) is generated.

(2) When on-chip debugging is enabled, bit error detection or correction of the RAM-ECC function is not performed. (Also, the INTRAM interrupt will not be generated.)

Error data injection can be executed in RAM-ECC test mode. However, communication between the E1 debugger and some bits where error data have been injected is disconnected, which disturbs the continuous debugging.

Therefore, the RAM-ECC-related programs need to be debugged in the stand-alone operation.

(3) Bit error detection will not be carried out for the instruction codes while RAM fetch is being performed. However, bit error detection is carried out while RAM data is being read by fetching an instruction from RAM.

4.7 Program size and execution time

Size

・ sf_ram_ecc_set_int_mode function: 16bytes

Execution time

・ 718ns

Size

・ sf_ram_ecc_start function: 7bytes

Execution time

・ 591ns

Size

・ sf_ram_ecc_clear_bit_error function :4bytes

Execution time

・ 373ns

Size

・ sf_ram_ecc_test_mode function: 47bytes

Execution time

・ 1.85μs

Conditions

・ Operating frequency: PLL clock (32MHz)

・ Memory model: medium

4.8 Sample program using RAM-ECC test mode

The RAM-ECC test operation in RAM-ECC test mode is described using a sample program.

-Sample specification-

- The INTRAM interrupt is not used.
- The RAM area to be tested is the entire area (512bytes) protected by the RAM guard function.
- The following error patterns will be injected to the RAM area.
  - Test operation 1 (2-bit error test operation): parity bit and ECC (bit 2)
  - Test operation 2 (2-bit error test operation): parity bit and data (bit 6)
  - Test operation 3 (2-bit error test operation): ECC (bit 1) and data (bit 5)
  - Test operation 4 (2-bit error test operation): ECC (bit 3) and ECC (bit 0)
  - Test operation 5 (2-bit error test operation): data (bit 4) and data (bit 0)
  - Test operation 6 (1-bit error test operation): ECC (bit 3)
  - Test operation 7 (1-bit error test operation): data (bit 7)

In this sample specification, the information (data) above are allocated in the table below.

List 4-6 Sample program (error data injection in test mode)

```
#define             USER_RAM_ECC_TEST_PHASE                                      (   7U )
static      const   uint16_t   ram_ecc_error_patern_array[ USER_RAM_ECC_TEST_PHASE ][ 3U ] = {
            /* Test operation 1 */
            (uint16_t)(SF_RAMECC_TEST_PARITY     | SF_RAMECC_TEST_ECC_2BIT ) ,  // Invert a parity bit and the ECC (bit 2).
            (uint8_t)SF_RAMECC_2BIT_ERROR_DETECT ,                             // The bit error detected is a 2-bit error.
            (uint8_t) USER_RAM_ECC_CASE_NO_ERROR_CORRECTIMG ,                  // Correction will not be carried out.
            /* Test operation 2 */
            (uint16_t)(SF_RAMECC_TEST_PARITY     | SF_RAMECC_TEST_DATA_6BIT) ,  // Invert a parity bit and data (bit 6).
            (uint8_t)SF_RAMECC_2BIT_ERROR_DETECT ,                             // The bit error detected is a 2-bit error.
            (uint8_t)USER_RAM_ECC_CASE_ERROR_CORRECTIMG ,                      // The data read has been corrected.
            /* Test operation 3 */
            (uint16_t)(SF_RAMECC_TEST_ECC_1BIT   | SF_RAMECC_TEST_DATA_5BIT) ,  // Inver the ECC (bit 1) and data (bit 5).
            (uint8_t)SF_RAMECC_2BIT_ERROR_DETECT ,                             // The bit error detected is a 2-bit error.
            (uint8_t)USER_RAM_ECC_CASE_NO_ERROR_CORRECTIMG ,                   // Correction of the read data is wrong.
            /* Test operation 4 */
            (uint16_t)(SF_RAMECC_TEST_ECC_3BIT   | SF_RAMECC_TEST_ECC_0BIT ) ,  // Invert the ECC (bit 3 and bit 0).
            (uint8_t)SF_RAMECC_2BIT_ERROR_DETECT ,                             // The bit error detected is a 2-bit error.
            (uint8_t)USER_RAM_ECC_CASE_NO_ERROR_CORRECTIMG ,                   // Correction of the read data is wrong.
            /* Test operation 5 */
            (uint16_t)(SF_RAMECC_TEST_DATA_4BIT | SF_RAMECC_TEST_DATA_0BIT) ,   // Invert data (bit 4 and bit 0).
            (uint8_t)SF_RAMECC_2BIT_ERROR_DETECT ,                             // The bit error detected is a 2-bit error.
            (uint8_t)USER_RAM_ECC_CASE_NO_ERROR_CORRECTIMG ,                   // Correction of the read data is wrong or will not be carried out.

            /* Test operation 6 */
            (uint16_t)(SF_RAMECC_TEST_ECC_3BIT                         ) ,     // Invert the ECC (bit 3).
            (uint8_t)SF_RAMECC_1BIT_ERROR_DETECT ,                             // The bit error detected is a 1-bit error.
            (uint8_t) USER_RAM_ECC_CASE_NO_ERROR_CORRECTIMG                    // Correction will not be carried out.
            /* Test operation 7 */
            (uint16_t)(SF_RAMECC_TEST_DATA_7BIT                        ) ,     // Invert data (bit 7).
            (uint8_t)SF_RAMECC_1BIT_ERROR_DETECT ,                             // The bit error detected is a 1-bit error.
            (uint8_t)USER_RAM_ECC_CASE_ERROR_CORRECTIMG                        // The read data has been corrected.
`
```

Figure 4.6 is a sample flowchart showing the overview of the RAM-ECC test operation.

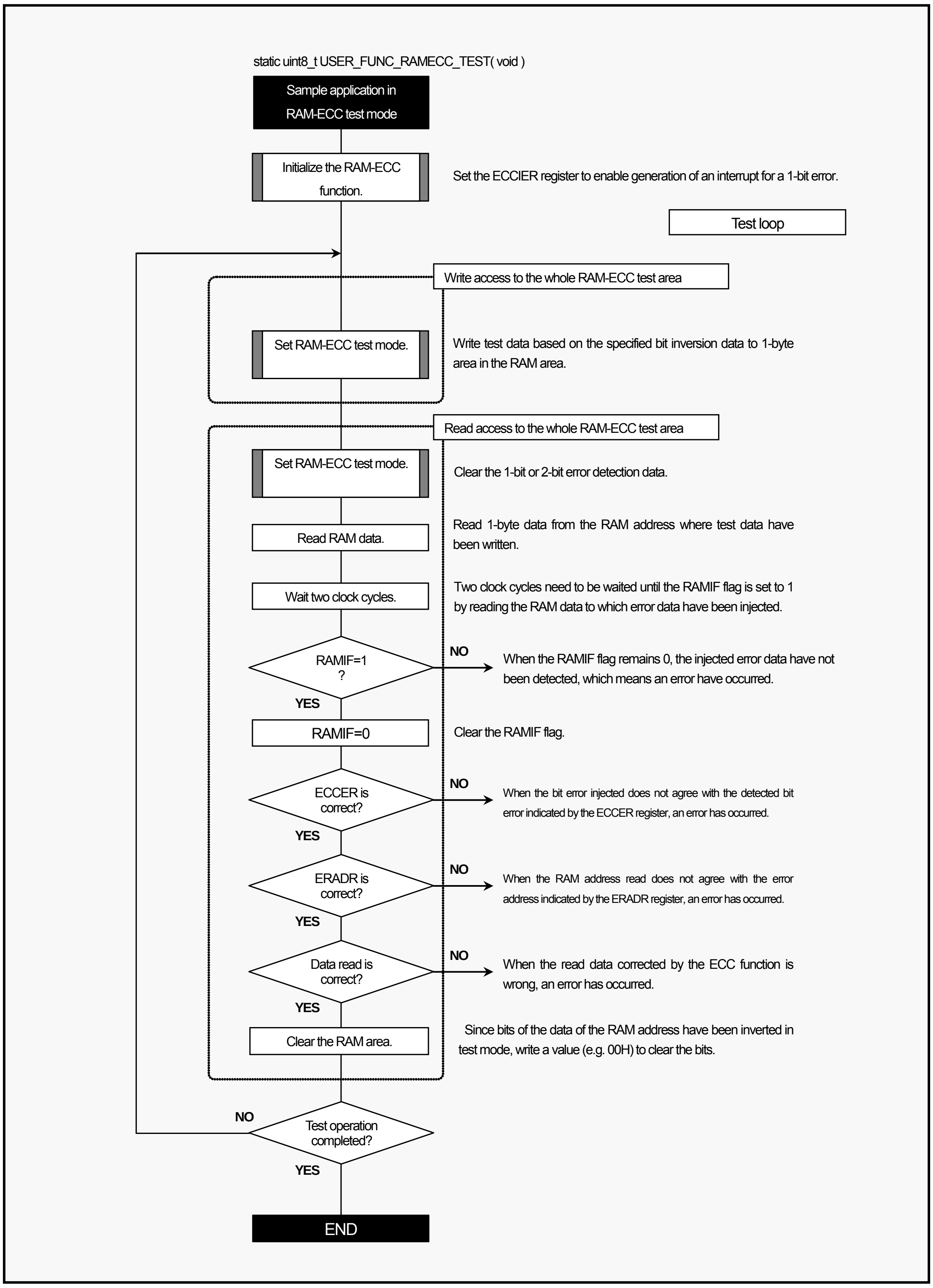

static uint8_t USER_FUNC_RAMECC_TEST(void)
Sample application in RAM-ECC test mode
Initialize the RAM-ECC function.
Set the ECCIER register to enable generation of an interrupt for a 1-bit error.
Test loop
Write access to the whole RAM-ECC test area
Set RAM-ECC test mode.
Write test data based on the specified bit inversion data to 1-byte area in the RAM area.
Read access to the whole RAM-ECC test area
Set RAM-ECC test mode.
Clear the 1-bit or 2-bit error detection data.
Read RAM data.
Read 1-byte data from the RAM address where test data have been written.
Wait two clock cycles.
Two clock cycles need to be waited until the RAMIF flag is set to 1 by reading the RAM data to which error data have been injected.
RAMIF=1 ?
NO
When the RAMIF flag remains 0, the injected error data have not been detected, which means an error have occurred.
YES
RAMIF=0
Clear the RAMIF flag.
ECCER is correct?
NO
When the bit error injected does not agree with the detected bit error indicated by the ECCER register, an error has occurred.
YES
ERADR is correct?
NO
When the RAM address read does not agree with the error address indicated by the ERADR register, an error has occurred.
YES
Data read is correct?
NO
When the read data corrected by the ECC function is wrong, an error has occurred.
YES
Clear the RAM area.
Since bits of the data of the RAM address have been inverted in test mode, write a value (e.g. 00H) to clear the bits.
NO
Test operation completed?
YES
END

Figure 4.6 Program flowchart

List 4-7 Sample program (RAM-ECC test mode operation)

```
static uint8_t USER_FUNC_RAMECC_TEST( void )
{
        uint16_t            lp;
        uint8_t             read_data;
        uint8_t             reg_value;
        uint8_t             ret_value;
        uint16_t            ecc_test_phase;

        ret_value           = USER_RAM_ECC_TEST_OK;
        RAMIF               = BIT_CLR;

        /* RAM-ECC 1 bit error detect interrupt enable.*/
        sf_ram_ecc_set_int_mode( SF_RAMECC_ENABLE_1BIT_ERROR_INTERRUPT );                          (1)

        /* Test loop start. */
        for ( ecc_test_phase=0U; ecc_test_phase<USER_RAM_ECC_TEST_PHASE; ecc_test_phase++ ) {      (2)
                /* write start. */
                for ( lp=0U; lp<sizeof(GUARD_MEMORY); lp++ ) {          // 512 bytes loop.
                        // error data write (1byte)
                        sf_ram_ecc_test_mode(                                                      (3)
                          (uint8_t __far *)&GUARD_MEMORY[lp] ,
                          (uint8_t)USER_RAM_ECC_TEST_WRITE_DATA ,
                          (uint16_t)ram_ecc_error_patern_array[ecc_test_phase][USER_RAM_ECC_ERROR_PATERN] );
                }

                /* read check start. */
                for ( lp=0U; lp<sizeof(GUARD_MEMORY); lp++ ) {
                                                                                                   (4)
                        sf_ram_ecc_clear_bit_error();                   // bit error information clear.

                        read_data       =   GUARD_MEMORY[lp];           // 1 byte read.            (5)

                        /* RAMIF flag set check. */                                                (6)
                        NOP();                                          // RAMIF set wait.
                        NOP();
                        if ( !RAMIF ) {
                                ret_value   = USER_RAM_ECC_TEST_FAULT;
                                break;
                        }
                        RAMIF           = BIT_CLR;                      // clear RAMIF.

                        /* bit error infomation check. */                                          (7)
                        if ( ram_ecc_error_patern_array[ecc_test_phase][USER_RAM_ECC_ERROR_BIT] != ECCER ) {
                                ret_value   = USER_RAM_ECC_TEST_FAULT;
                                break;
                        }

                        /* bit error address check. */                                             (8)
                        if ( (uint16_t __far *)&GUARD_MEMORY[lp] != (uint16_t __far *)(ERADR | 0xf0000) ) {
                                ret_value   = USER_RAM_ECC_TEST_FAULT;
                                break;
                        }

                        /* error correct check. */                                                 (9)
                        if ( ram_ecc_error_patern_array[ecc_test_phase][USER_RAM_ECC_ERROR_CORRECT]
                                == USER_RAM_ECC_CASE_ERROR_CORRECTIMG ) {
                                if ( read_data != USER_RAM_ECC_TEST_WRITE_DATA ) {
                                        ret_value   = USER_RAM_ECC_TEST_FAULT;
                                        break;
                                }
                        }
                        GUARD_MEMORY[lp]  = 0x00;               // clear RAM.                      (10)
                }
                if ( ret_value != USER_RAM_ECC_TEST_OK ) {
                        break;
                }
        }
        return ret_value;
}
```

(1) Call the sf_ram_ecc_set_int_mode function to enable the 1-bit error detection interrupt.

(2) Repeat the test operation in test mode from Step (1) to Step (7) indicated in List 4-7.

-Bit error injection-

(3) Call the sf_ram_ecc_test_mode function and inject error data to each 1 byte of the RAM area indicated by GUARD_MEMORY so that the entire 512 bytes of the area will be entered into the error status. (For details, refer to the ram_ecc_error_patern_array table.)

-Bit error confirmation-

(4) Call the sf_ram_ecc_clear_bit_error function to clear the information on detection of a 1-bit or 2-bit error.

(5) Read one byte data of the RAM address (where the error data have been injected) indicated by GUARD_MEMORY.

(6) Confirm that bit errors can be properly detected by read access.

When two clock cycles have passed after the read access to the target RAM, the INTRAM interrupt request flag (RAMIF) is set to 1,which can confirm a bit error has been detected.

(7) Check whether the bit error detected (ECCER register) agrees with the bit error injected (the ram_ecc_error_patern_array).

(8) Check whether the address (the ERADR register) where the bit error has been detected agrees with the RAM address read.

(9) Confirm that the read RAM data have been properly corrected by the ECC function.

According to the bit error injected, the ECC function performs the following:

1. Corrects the error,
2. corrects the error but the corrosion is wrong, or
3. the error will not be corrected.

In this example, only when a correctable error is injected, it is possible to confirm that the written value agrees with the read value by referencing to the ram_ecc_error_patern_array table.

(10) To clear the injected bit error, rewrite the corresponding RAM data with a value (e.g. 00H).

For the 512-byte data, the processing will be repeated from Step (4) above.

Caution: Before performing this processing, disable the generation of interrupts.

Note: During this processing, the on-chip debugging cannot be performed.

5. CPU stack pointer monitor function

5.1 Overview of the fault diagnostic function

The automotive functional safety standard ISO 26262 stipulates self-diagnostic programs and stack pointer (SP) monitoring as methods for detecting faults in a processor.

The CPU stack pointer monitor function monitors stack pointers, and generates an interrupt when an SP exceeds the highest/lowest thresholds.

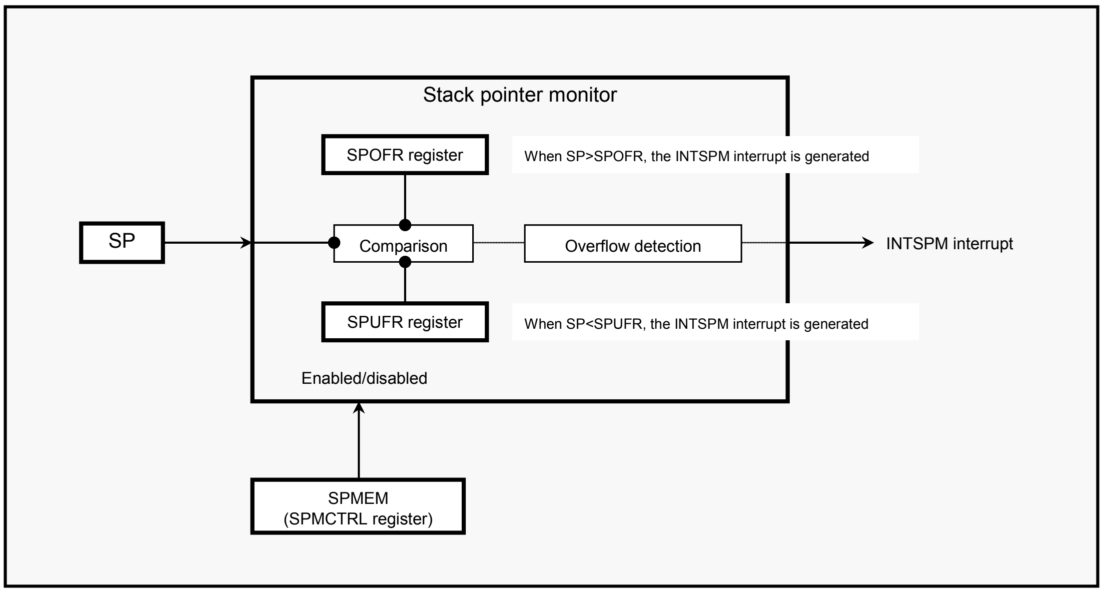


Figure 5.1 Structure of CPU stack pointer monitor function

5.2 Description of the fault diagnosis-related registers

The registers used for the CPU stack pointer monitor function are described below.

(1) SPM control register (SPMCTRL)

This SPMCTRL register controls the operation of the CPU stack pointer monitor function.

(2) SP overflow address setting register (SPOFR)

This SPOFR register sets the threshold to detect an SP overflow.

(3) SP underflow address setting register (SPUFR)

This SPUFR register sets the threshold to detect an SP underflow.

Table 5.1 lists the setting examples of the related registers.

Table 5.1 Setting examples of the CPU stack pointer monitor function-related registers

Register	**Set value**	**Description**
SPM control register (SPMCTRL)	80H	Activates the stack pointer monitor function.
SP overflow address setting register (SPOFR)	F2EAH	Sets the threshold "F2EAH" to detect an overflow.
SP underflow address setting register (SPUFR)	F4EAH	Sets the threshold "F4EAH" to detect an underflow.

5.3 Program flowchart to execute the diagnostic function

Figure 5.2 is a flowchart showing the overview of the CPU stack pointer monitoring.

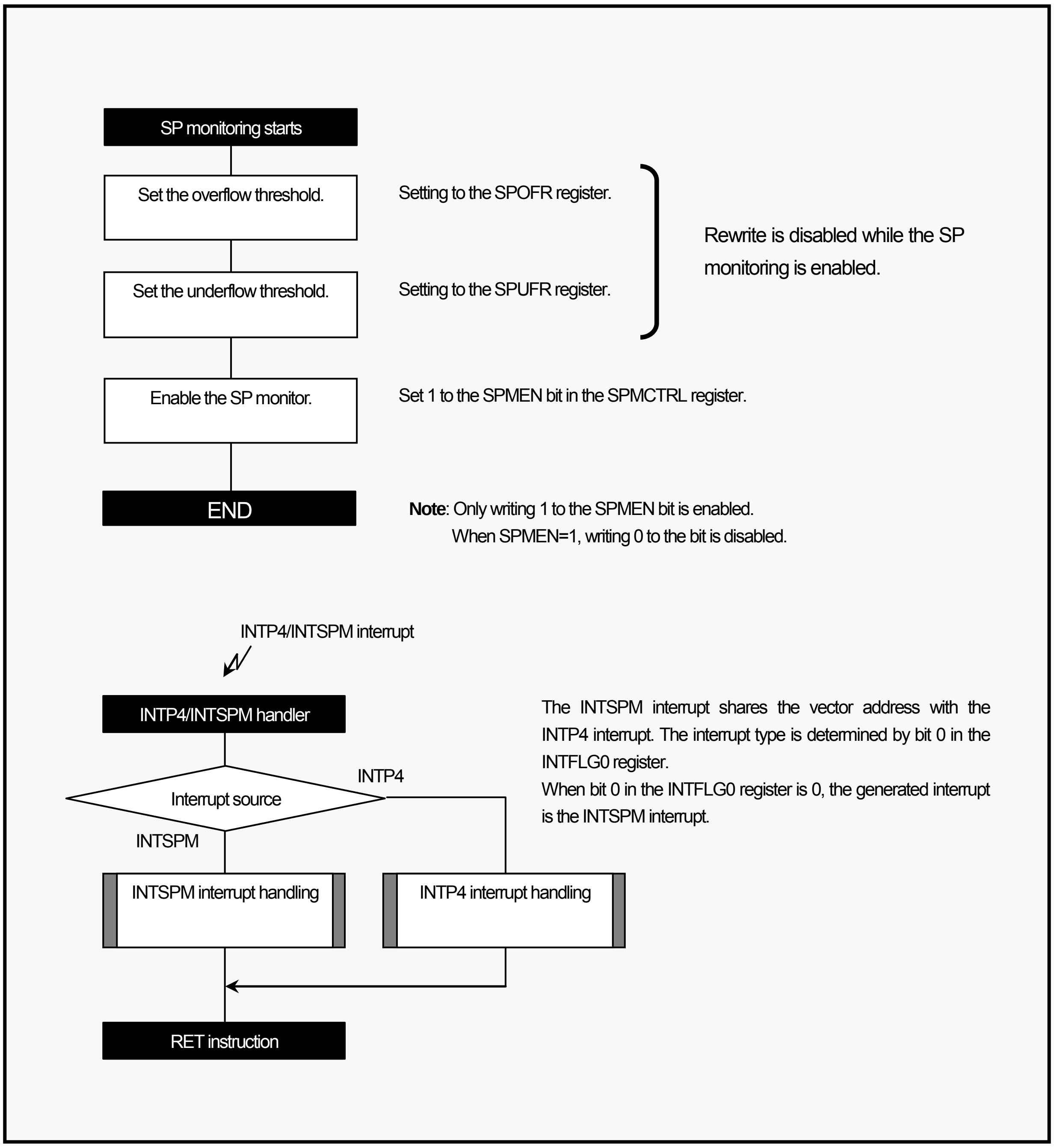

SP monitoring starts
Set the overflow threshold.
Setting to the SPOFR register.
Set the underflow threshold.
Setting to the SPUFR register.
Rewrite is disabled while the SP monitoring is enabled.
Enable the SP monitor.
Set 1 to the SPMEN bit in the SPMCTRL register.
END
Note: Only writing 1 to the SPMEN bit is enabled. When SPMEN=1, writing 0 to the bit is disabled.
INTP4/INTSPM interrupt
INTP4/INTSPM handler
Interrupt source
INTP4
INTSPM
INTSPM interrupt handling
INTP4 interrupt handling
RET instruction
The INTSPM interrupt shares the vector address with the INTP4 interrupt. The interrupt type is determined by bit 0 in the INTFLG0 register.
When bit 0 in the INTFLG0 register is 0, the generated interrupt is the INTSPM interrupt.

Figure 5.2 Program flowchart

5.4 Sample program to execute the diagnostic function

The following is a sample program for the CPU stack pointer monitor function.

List 5-1 Sample program (SP monitor)

```
void sf_spmon_start( uint16_t sp_ovf_address , uint16_t sp_udf_address )
{
        SPOFR       = ( sp_ovf_address & 0xFFFEU );                                          (1)
        SPUFR       = ( sp_udf_address & 0xFFFEU );                                          (2)

        SPMIF       = BIT_CLR;         // Clear the INTSPM interrupt request flag.           (3)
        SPMMK       = BIT_CLR;         // Clear the INTSPM interrupt mask flag.              (4)
        SPMCTRL     = 0x80U;                                                                 (5)
}
```

(1) Set the SP overflow threshold address to the SPOFR register.

(2) Set the SP underflow threshold address to the SPUFR register.

(3) Clear the INTSPM interrupt request flag (SPMIF).

(4) Clear the INTSPM interrupt mask flag (SPMMK) to enable the INTSPM interrupt.

(5) Set 80H to the SPMCTL register to activate the SP monitoring.

5.5 Cautions when using the CPU stack pointer monitor function

The following is the caution when using the CPU stack pointer monitor function.

(1) When on-chip debugging is enabled, the stack pointer monitor function cannot be properly debugged.

Due to the debugging operation using the on-chip debugging function such as stepwise execution or debugging break, an SP overflow or underflow detection interrupt (INTSPM) will be generated in an unintended timing.

5.6 Program size and execution time

Size

- sf_spmon_start function: 37bytes

Execution time

- 1.251µs

Conditions

- Operating frequency: PLL clock (32MHz)
- memory model: medium

6. Clock monitoring function

6.1 Overview of the fault diagnostic function

The clock monitoring function monitors the oscillation status of the clock by sampling the main system clock (f_{MAIN}) and the PLL clock using a low-speed on-chip oscillator.

When the CLKMB bit (00C1H) of the option byte is cleared, the clock monitoring function is enabled. When the CSS bit in the CKC register is set to 1 (when f_{CLK}=f_{SUB} or f_{IL}), the clock monitoring function is disabled.

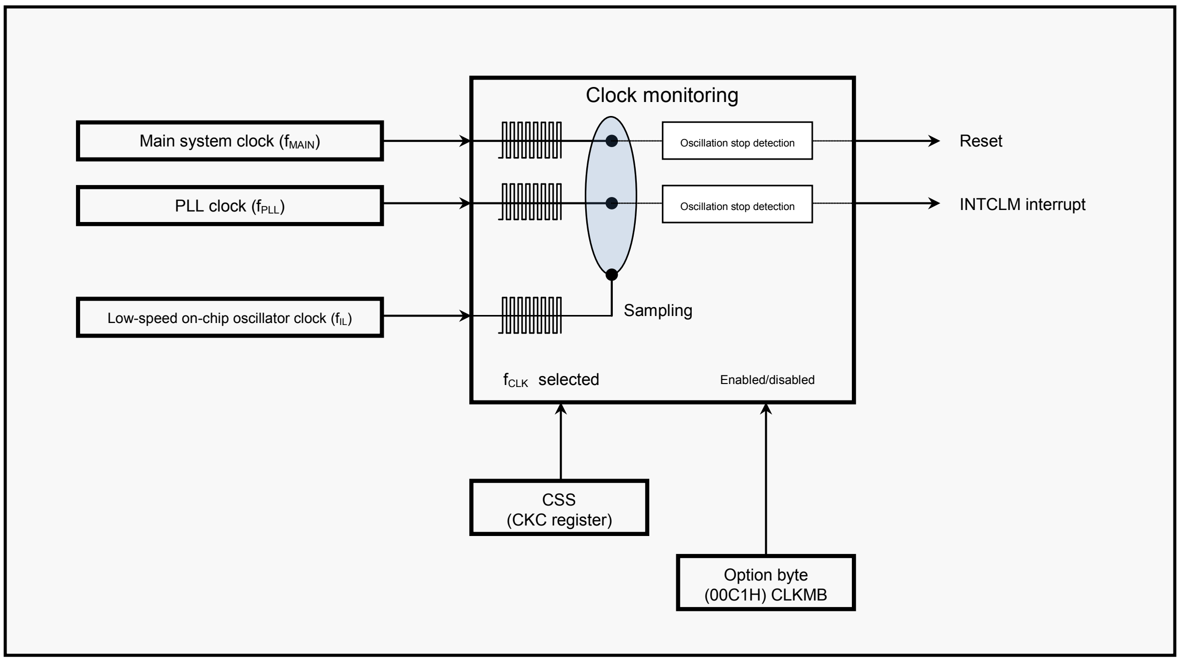

Clock monitoring
Main system clock (fMAIN)
PLL clock (fPLL)
Low-speed on-chip oscillator clock (fIL)
Oscillation stop detection
Oscillation stop detection
Reset
INTCLM interrupt
Sampling
fCLK selected
Enabled/disabled
CSS (CKC register)
Option byte (00C1H) CLKMB

Figure 6.1 Structure of clock monitoring function

When oscillation of the main system clock stops while the low-speed on-chip oscillator is sampling the clock, a reset request signal will be generated.

When the PLL clock stops, clock through mode is forcibly selected (the SELPLLS bit in the PLLSTS register is cleared, and the SELPLL bit in the PLLCTL register is not cleared). Simultaneously, the INTCLM interrupt is generated.

6.2 Program flowchart to execute the diagnostic function

Figure 6.2 is a flowchart showing the overview of the clock monitoring function.

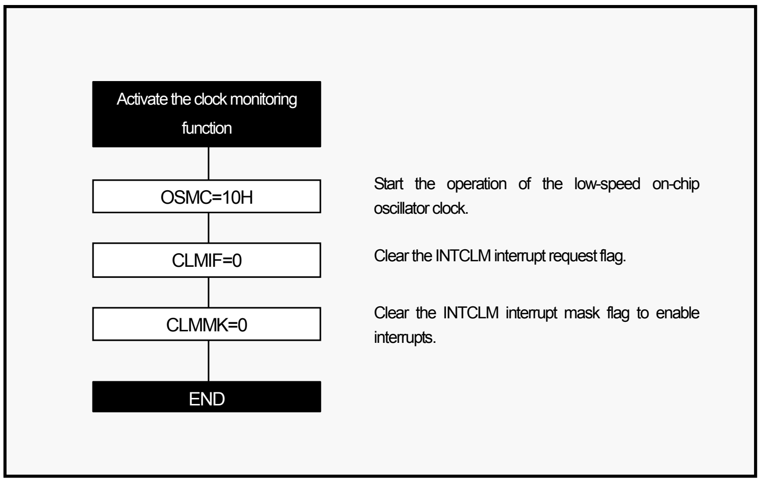

Activate the clock monitoring function
OSMC=10H
Start the operation of the low-speed on-chip oscillator clock.
CLMIF=0
Clear the INTCLM interrupt request flag.
CLMMK=0
Clear the INTCLM interrupt mask flag to enable interrupts.
END

Figure 6.2 Program flowchart

6.3 Sample program to execute the diagnostic function

The following is a sample program for the clock monitoring function.

List 6-1 Sample Program (clock monitor function)

```
void sf_oscmon_start( void )
{
        /*      Operation speed mode control register (OSMC)
                b7       : Setting in STOP mode or HALT mode while subsystem clock
                             is selected as CPU clock.
                b6 :b5   : Reserved set to 0.
                b4       : Selection of operation clock for real-time clock and
                             interval timer.
                b3 :b0   : Reserved set to 0. */
        OSMC             = 0x10U;                                                      (1)

        CLMIF            = BIT_CLR;          // Clear the INTCLM interrupt request flag.   (2)
        CLMMK            = BIT_CLR;          // Clear the INTCLM interrupt mask flag.      (3)
}
```

(1) Start the operation of the low-speed on-chip oscillator clock.

(2) Clear the INTCLM interrupt request flag (CLMIF).

(3) Clear the INTCLM interrupt mask flag (CLMMK) to enable the INTCLM interrupt.

6.4 Cautions when using clock monitoring function

The following is the caution when using the clock monitoring function.

(1) On-chip debugging using the E1 emulator is not available.

6.5 Program size and execution time

Size

- sf_oscmon_start function: 11bytes

Execution time

- 620ns

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

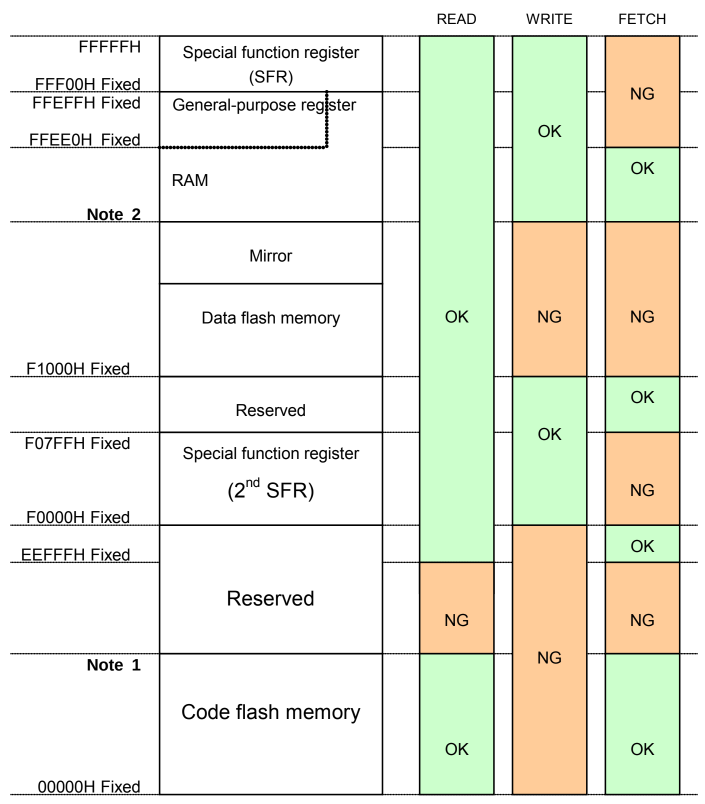
7. Invalid memory access detection function

7.1 Overview of the diagnostic function

The IEC 60730 standard mandates checking that the CPU and interrupts are operating correctly. With this invalid memory access detection function, a reset signal is generated when a memory space that is specified as an access-prohibited area is accessed.

7.2 Invalid memory access detection area

The invalid memory access detection areas are marked with “NG” in Figure 7.1.



Note 1: The code flash memory size varies by product.

Note 2: The RAM size varies by product.

Regarding the size of the code flash memory and RAM and the invalid memory access detection areas for each product, refer to “RL78/F13, F14 User’s Manual: Hardware”

Figure 7.1 Invalid memory access detection areas

7.3 Description of the fault diagnosis-related registers

The register used for the invalid memory access detection function is described below.

(1) Invalid memory access detection control register (IAWCTL)

This IAWCTL register controls the detection of invalid memory access and the RAM/SFR guard function.

The table below shows the setting example of the related register.

Table 7.1 Setting example of invalid memory access detection-related register

Register	Set value	Description
Invalid memory access detection control register (IAWCTL)	80H	Enables the invalid memory access detection function.

7.4 Program flowchart to execute the diagnostic function

Figure 7.2 is a flowchart showing the overview of the invalid memory access detection function.

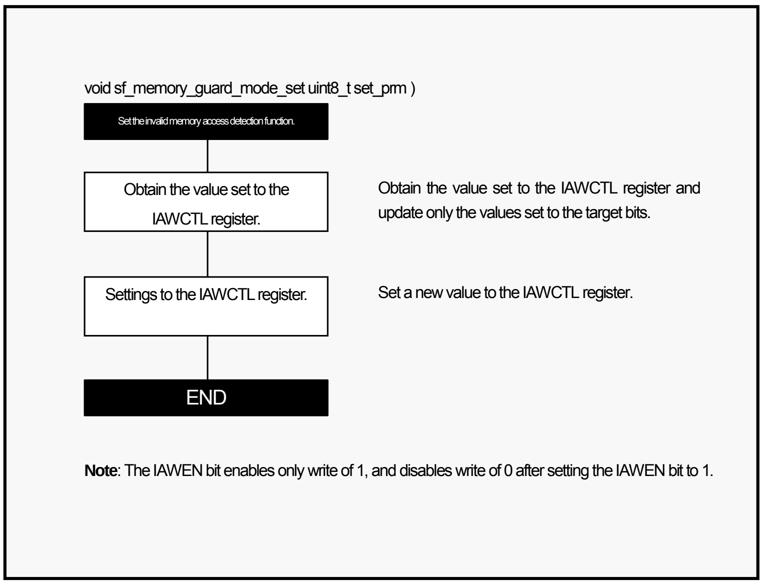

void sf_memory_guard_mode_set uint8_t set_prm)
Set the invalid memory access detection function.
Obtain the value set to the IAWCTL register.
Obtain the value set to the IAWCTL register and update only the values set to the target bits.
Settings to the IAWCTL register.
Set a new value to the IAWCTL register.
END
Note: The IAWEN bit enables only write of 1, and disables write of 0 after setting the IAWEN bit to 1.

Figure 7.2 Program flowchart

7.5 Sample program to execute the diagnostic function

The following is a sample program for the invalid memory access detection function.

List 7-1 Sample program (Invalid memory access detection)

```
void sf_memory_guard_mode_set( uint8_t set_prm )
{
        uint8_t         reg_value;

        reg_value       = IAWCTL;                                          (1)
        set_prm         = set_prm & (uint8_t)SF_MEMORY_GUARD_MASK;
        IAWCTL          = ( set_prm | reg_value ) & (uint8_t)SF_IAWCTL_MASK;     (2)
}
```

(1) Obtain the value set to the IAWCTL register.

(2) Based on the obtained IAWCTL register value and a value to be set for the invalid memory access detection function, a value to be set to the IAWCTL register will be updated.

7.6 Program size and execution time

Size

- sf_memory_guard_mode_set function: 18bytes

Execution time

- 1.5μs

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

8. RAM guard function

8.1 Overview of the diagnostic function

Since the IEC 61508 standard requires safety during system’s operation be guaranteed, important data stored in RAM needs to be protected even when the CPU runs out of control. This RAM guard function protects data in the specified memory area.

Once the RAM guard function is enabled, write access to the specified RAM area is disabled and read access to the area is enabled.

8.2 Description of the fault diagnosis-related registers

The register used for the RAM guard function is described below.

(1) Invalid memory access detection control register (IAWCTL)

This IAWCTL register controls the detection of invalid memory access and the RAM/SFR guard function.

Table 8.1 shows the setting example of the related register.

Table 8.1 Setting example of RAM guard related register

Register	Set value	Description
Invalid memory access detection control register (IAWCTL)	20H	Protects 256 bytes starting at the lower RAM address.

8.3 Program flowchart to execute the diagnostic function

Figure 8.1 is a flowchart showing the overview of the RAM guard function.

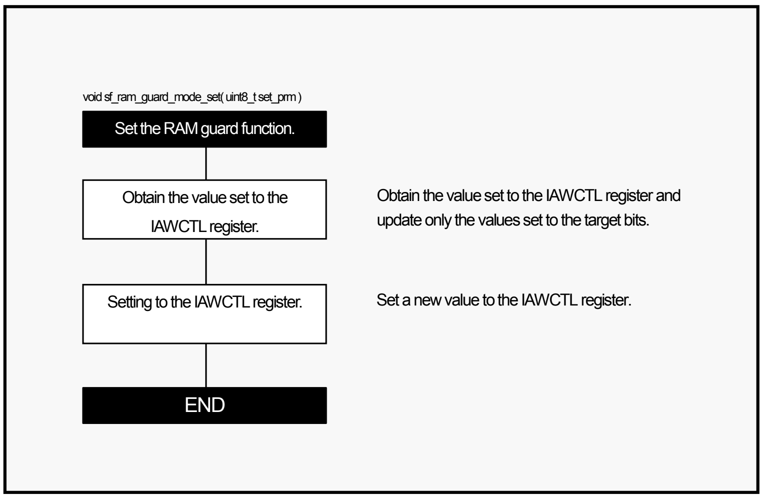

void sf_ram_guard_mode_set(uint8_t set_prm)
Set the RAM guard function.
Obtain the value set to the IAWCTL register.
Obtain the value set to the IAWCTL register and update only the values set to the target bits.
Setting to the IAWCTL register.
Set a new value to the IAWCTL register.
END

Figure 8.1 Program flowchart

8.4 Sample program to execute the diagnostic function

The following is a sample program for the RAM guard function.

List 8-1 Sample program (RAM guard)

```
void sf_ram_guard_mode_set( uint8_t set_prm )
{
            uint8_t         reg_value;

            reg_value       = IAWCTL;                                                    (1)
            reg_value       = reg_value & (uint8_t)(~SF_RAM_GUARD_MASK);
            set_prm         = set_prm & (uint8_t)SF_RAM_GUARD_MASK;
            IAWCTL          = ( set_prm | reg_value ) & (uint8_t)SF_IAWCTL_MASK;         (2)
}
```

(1) Obtain the value set to the IAWCTL register.

(2) Based on the obtained IAWCTL register value and the value to be set for the RAM guard function, a value to be set to the IAWCTL register will be updated.

8.5 Cautions when using the RAM guard function

The followings are the cautions when using the RAM guard function.

(1) The RAM guard function does not work at stepwise execution of on-chip debugging by the integrated development environment CubeSuite+. (Even when the guard function is set, the protection is not available.)

(2) When the position of a stack pointer overlaps with the area where the RAM guard function is effective, write access to the RAM is blocked for the processing with a stack pointer. Therefore, the subsequent operation is not guaranteed.

8.6 Program size and execution time

Size

・ sf_ram_guard_mode_set function: 20bytes

Execution time:

・ 1.531μs

Conditions

・ Operating frequency: PLL clock (32MHz)

・ Memory model: medium

9. SFR guard function

9.1 Overview of the fault diagnostic function

In order to guarantee safety during system's operation, the IEC 61508 standard requires that the SFRs be protected so that their important data will not be rewritten even when the CPU runs out of control. The SFR guard function protects data in the control registers used by the port function, interrupt function, and clock control function.

Once the SFR guard function is enabled, write access to the protected SFRs is disabled and read access to the registers is enabled.

The following registers are protected by the SFR guard function.

[Port registers to be protected]
PMxx, PUxx, PIMxx, POMxx, PMCxx, PITHLxx, ADPC, PIOR

[Interrupt control registers to be protected]
IFxx MKxx, PRxx, EGPx, EGNx

[CSC registers to be protected]
CMC, CSC, OSTS, CKC, PERx, OSMC, LVIM, LVIS,
CANCKSEL, LINCKSEL, CKSEL, PLLCTL, MDIV, RTCCL, POCRES, STPSTC,

9.2 Description of the fault diagnosis-related registers

The register used for the SFR guard function is described below.

(1) Invalid memory access detection control register (IAWCTL)

This IAWCTL register controls the detection of invalid memory access and the RAM/SFR guard function.

Table 9.1 shows the setting example of the related register.

Table 9.1 Setting examples of the SFR guard function related register

Register	**Set value**	**Description**
Invalid memory access detection control register (IAWCTL)	07H	The following registers are protected by the SFR guard function: PMxx, PUxx, PIMxx, POMxx, PMCxx, PITHLxx, ADPC, PIOR IFxx, MKxx, PRxx, EGPx, EGNx CMC, CSC, OSTS, CKC, PERx, OSMC, LVIM, LVIS CANCKSEL, LINCKSEL, CKSEL, PLLCTL, MDIV, RTCCL, POCRES, STPSTC

9.3 Program flowchart to execute the diagnostic function

Figure 9.1 is a flowchart showing the overview of the SFR guard function.

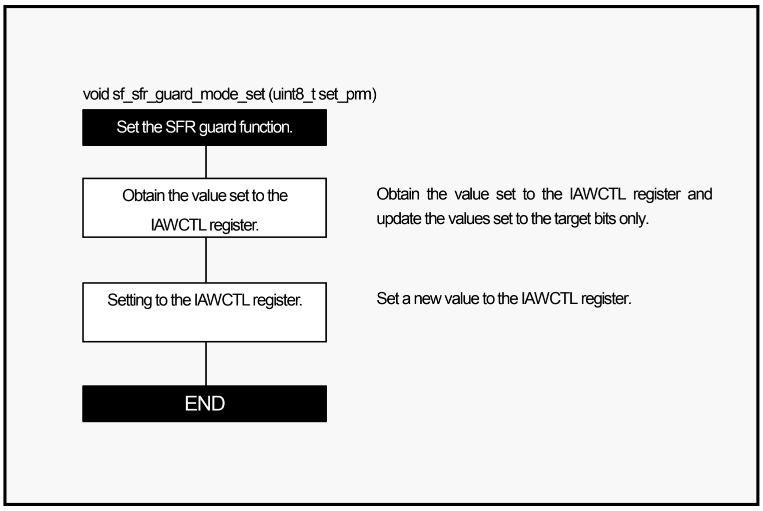

void sf_sfr_guard_mode_set (uint8_t set_prm)
Set the SFR guard function.
Obtain the value set to the IAWCTL register.
Obtain the value set to the IAWCTL register and update the values set to the target bits only.
Setting to the IAWCTL register.
Set a new value to the IAWCTL register.
END

Figure 9.1 Program flowchart

9.4 Sample program to execute the diagnostic function

The following is a sample program for the SFR guard function.

List 9-1 Sample program (SFR guard processing)

```
void sf_sfr_guard_mode_set( uint8_t set_prm )
{
        uint8_t         reg_value;

        reg_value       = IAWCTL;                                            (1)
        reg_value       = reg_value & (uint8_t)(~SF_SFR_GUARD_MASK);
        set_prm         = set_prm & (uint8_t)SF_SFR_GUARD_MASK;
        IAWCTL          = ( set_prm | reg_value ) & (uint8_t)SF_IAWCTL_MASK;   (2)
}
```

(1) Obtain the value set to the IAWCTL register.

(2) Based on the obtained IAWCTL register value and the value to be set to the SFR guard function, a value to be set to the IAWCTL register will be updated.

9.5 Program size and execution time

Size

- sf_sfr_guard_mode_set function: 20bytes

Execution time:

- 1.54μs

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

10. Frequency detection function

10.1 Overview of the fault diagnostic function

The IEC 60730 standard requires to check the clock oscillation frequency be normal. By using the timer array unit 0 (ATU0), the frequency detection function can detect an abnormal clock frequency by comparing the high-speed on-chip oscillator clock or external X1 oscillation clock with the low-speed on-chip oscillator clock (15kHz).

TAU0 activates timer channel 1 as the input pulse interval measurement function. Whether the clock frequency is appropriate or not can be detected by measuring the pulse width under the following conditions: The low-speed on-chip oscillator clock is selected as a clock input to TAU0 channel 1 and one clock of the CPU/peripheral hardware clock (f_{CLK}) of the low-speed on-chip oscillator clock is counted. The measurement of the pulse width can determine whether the clock frequency is normal or abnormal.

Figure 10.1 is the structure of the frequency detection function.

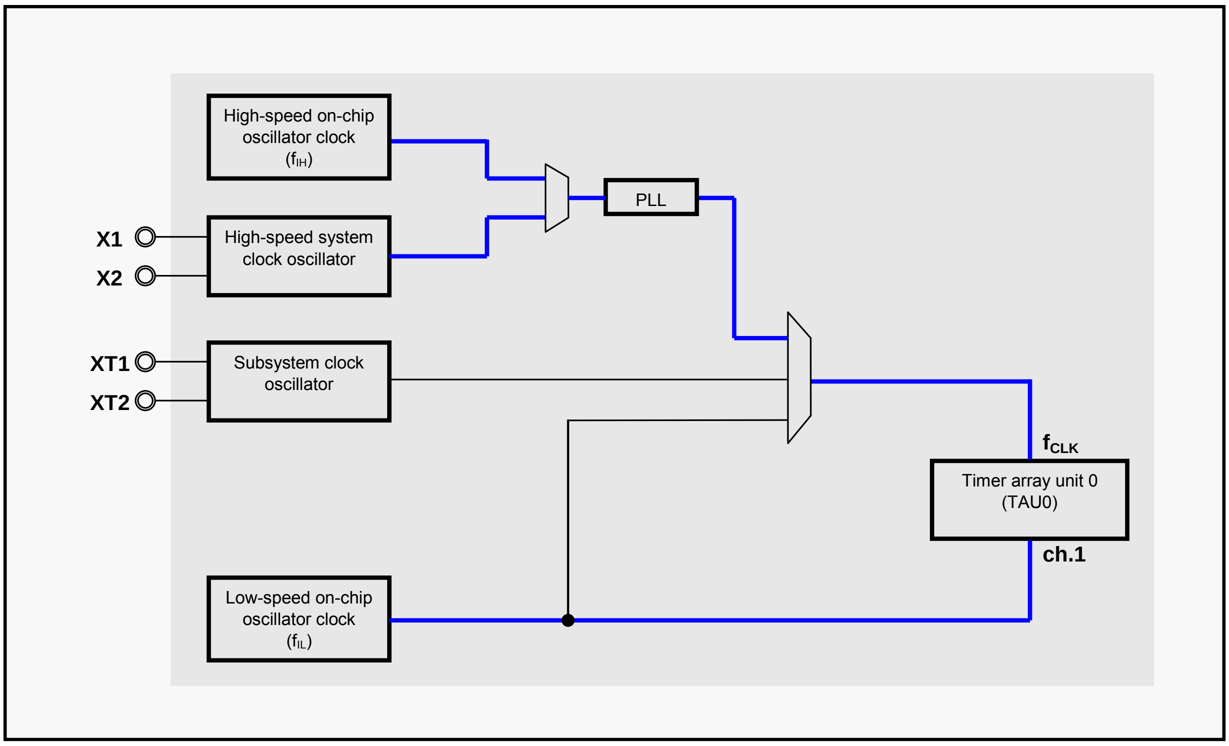

High-speed on-chip oscillator clock (f_{IH})
X1
X2
High-speed system clock oscillator
PLL
XT1
XT2
Subsystem clock oscillator
f_{CLK}
Timer array unit 0 (TAU0)
ch.1
Low-speed on-chip oscillator clock (f_{IL})

Figure 10.1 Overview of frequency detection function

Note: The measurement error is about one clock of the TAU0 operation clock (f_{CLK}).

10.2 Description of the fault diagnosis-related registers

The register used for the frequency detection function is described below.

(1) Timer input select register 0 (TIS0)

This TIS0 register selects the timer input of channel 1.

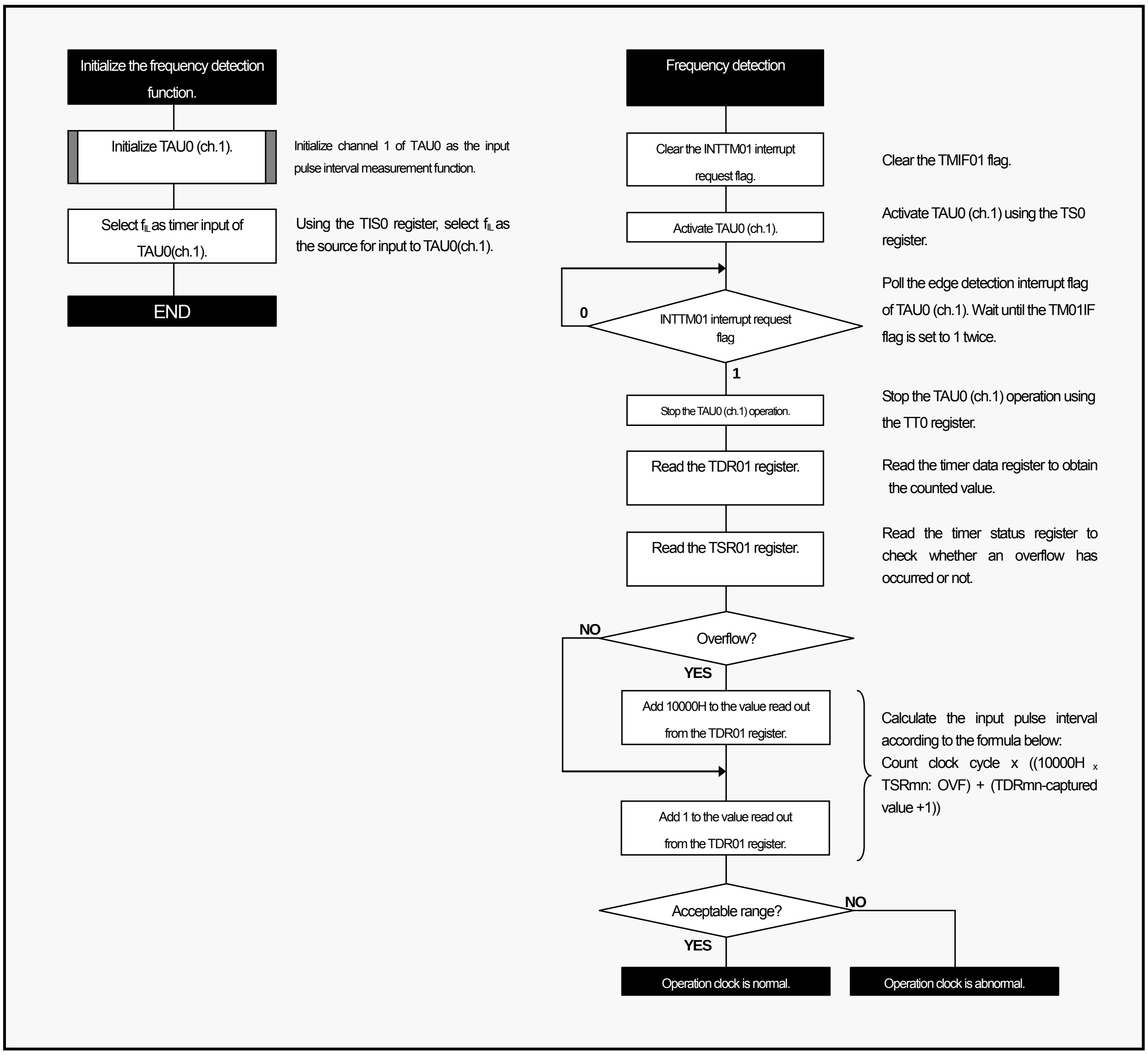
The table below shows the setting example of the related register.

Table 10.1 Setting example of frequency detection-related register

Register	**Set value**	**Description**
Timer input select register 0 (TIS0)	04H	Selects the low-speed on-chip oscillator clock (f_{IL}) as the timer input used for channel 1.

10.3 Program flowchart to execute the diagnostic function

Figure 10.2 is a flowchart showing the overview of the frequency detection function.



Initialize the frequency detection function.
Initialize TAU0 (ch.1).
Initialize channel 1 of TAU0 as the input pulse interval measurement function.
Select f_{IL} as timer input of TAU0(ch.1).
Using the TIS0 register, select f_{IL} as the source for input to TAU0(ch.1).
END
Frequency detection
Clear the INTTM01 interrupt request flag.
Clear the TMIF01 flag.
Activate TAU0 (ch.1).
Activate TAU0 (ch.1) using the TS0 register.
0
INTTM01 interrupt request flag
1
Poll the edge detection interrupt flag of TAU0 (ch.1). Wait until the TM01IF flag is set to 1 twice.
Stop the TAU0 (ch.1) operation.
Stop the TAU0 (ch.1) operation using the TT0 register.
Read the TDR01 register.
Read the timer data register to obtain the counted value.
Read the TSR01 register.
Read the timer status register to check whether an overflow has occurred or not.
NO
Overflow?
YES
Add 10000H to the value read out from the TDR01 register.
Add 1 to the value read out from the TDR01 register.
Calculate the input pulse interval according to the formula below: Count clock cycle x ((10000H x TSRmn: OVF) + (TDRmn-captured value +1))
Acceptable range?
NO
YES
Operation clock is normal.
Operation clock is abnormal.

Figure 10.2 Program flowchart

Note: The firstly-obtained value of the TDR01 register after the timer is activated is not a counted value of a valid edge. This is because a count start trigger is generated by the setting of TS0 by software (=rising of the TE0) not by detection of a valid edge.
In the flowchart above, the program waits until the second setting of the INTTM01 interrupt request flag (TMIF01) is done.

10.4 Sample program to execute the diagnostic function

The following is a sample program for the frequency detection function.

List 10-1 Sample program (Timer initial setting)

```
void sf_frqdet_initialize( void )
{
        TMMK01          = BIT_SET;              // SET the INTTM01 interrupt mask flag.         (1)
        TMIF01          = BIT_CLR;              // Clear the INTTM01 interrupt request flag.

        /*      Operation speed mode control register (OSMC)
                b7      : Setting in STOP mode or HALT mode while subsystem clock
                            is selected as CPU clock.
                b6 :b5  : Reserved set to 0.
                b4      : Selection of operation clock for real-time clock and
                            interval timer.
                b3 :b0  : Reserved set to 0. */
        OSMC            = 0x10U;                                                                (2)

        TAU0EN          = BIT_SET;              // Supplies input clock for TAU0.               (3)

        /*      Timer input select register 0 (TIS0)
                b7:b3   : Reserved set to 0.
                b2:b0   : Selection of timer input used with channel 1. */
        TIS0            = 0x04U;                                                                (4)

        /*      Timer clock select register 0 (TPS0)
                b15:b14 : Reserved set to 0.
                b13:b12 : Interval Times Available for Operation Clock CKS03.
                b11:b10 : Reserved set to 0.
                b9 :b8  : Interval Times Available for Operation Clock CKS02.
                b7 :b4  : Selection of operation clock (CK01).
                b3 :b0  : Selection of operation clock (CK00). */
        TPS0            = 0x0000U;                                                              (5)

        /*      Timer mode register 0n (TMR0n)
                b15:b14 : Selection of operation clock (fMCK) of channel n.
                b13     : Reserved set to 0.
                b12     : Selection of count clock (fTCLK) of channel n.
                b11     : Selection between using channel n independently or
                            simultaneously with another channel(as a slave or master).
                b10:b8  : Setting of start trigger or capture trigger of channel n.
                b7 :b6  : Selection of TI0n pin input valid edge.
                b5 :b4  : Reserved set to 0.
                b3 :b0  : Operation mode of channel n. */
        TMR01           = 0x0144U;                                                              (6)

}
```

(1) Set the INTTM01 interrupt mask flag (TMMK01) to 1 to disable the INTTM01 interrupt.
Also, clear the INTTM01 interrupt request flag (TMIF01).

(2) Start the operation of the low-speed on-chip oscillator clock.

(3) Start the clock supply to TAU0.

(4) Select the low-speed on-chip oscillator clock as timer input to TAU0 (channel 1).

(5) Set the clock to be supplied to TAU0 (channel 1) to 32MHz.

(6) Set the operation mode of TAU0 (channel 1).

List 10-2 Sample program (frequency detection)

```
uint32_t sf_frqdet_get_frequency( void )
{
        uint8_t         lp;
        uint8_t         reg_data;
        uint32_t        tdr01_value;

        TMIF01          = BIT_CLR;              // Clear the INTTM01 interrupt request flag.   (1)
        TS0             = 0x0002U;              // TAU0(ch.1) start.                           (2)

        for ( lp=0U ; lp<2U ; lp++ ) {                                                         (3)
                do {
                        reg_data = TMIF01;
                } while ( reg_data!=BIT_SET );
                TMIF01  = BIT_CLR;              // Clear the INTTM01 interrupt request flag.
        }

        TT0             = 0x0002U;              // TAU0(ch.1) stop.                            (4)

        tdr01_value     = (uint32_t)TDR01;      // Timer Data Register.                        (5)
        if ( TSR01 ) {                          // Timer Status Register.
                tdr01_value     += 0x10000UL;   // Overflow.
        }
        tdr01_value             += 1UL;

        return  tdr01_value;
}
```

(1) Clear the INTTM01 interrupt request flag (TMIF01).

(2) Start the operation of TAU0 (channel1).

(3) Wait until the INTTM01 interrupt request flag (TMIF01) is set to 1 upon detection of a valid edge of the low-speed on-chip oscillator clock. (Wait until the second setting to the flag is done.)

(4) Stop the operation of TAU0 (channel 1).

(5) Read the timer data register to obtain the counted value and calculate an input pulse interval according to the formula below.

a period of count clock × ((10000H×TSR01: OVF) + (capture value of TDR01+1))

10.5 Program size and execution time

Size

- sf_frqdet_initialize function: 29bytes

Execution time

- 834ns

Size:

- sf_frqdet_get_frequency function: 95bytes

Execution time

- 91.52μs

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

11. A/D test function

11.1 Overview of the fault diagnostic function

The IEC 60730 standard requires that the A/D converters be tested. This A/D test function checks whether the A/D converter is operating properly by A/D-converting the AV_{REFP} voltage, the AV_{REFM} voltage, and the internal reference voltage (1.45 V).

Figure 11.1 is the structure of the A/D test function.

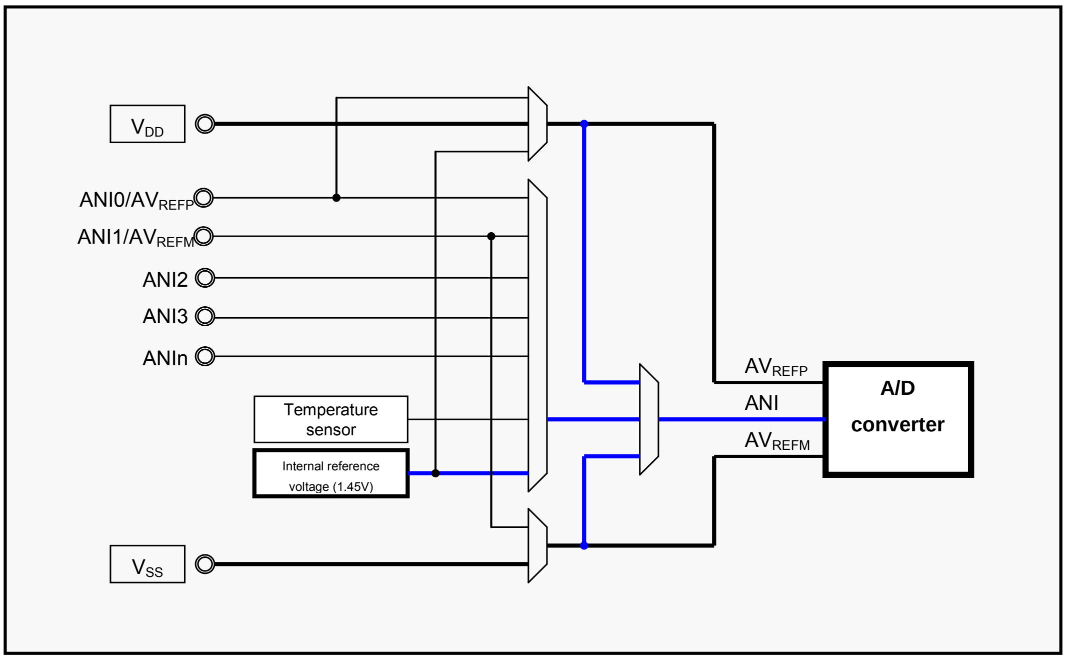


Figure 11.1 Structure of A/D test function

11.2 Description of the fault diagnosis-related registers

The registers used for the A/D test function are described below.

(1) A/D test register (ADTES)

This ADTES register selects the A/D converter's positive reference voltage AV_{REFP}, the A/D converter's negative reference voltage AV_{REFM}, or the analog input channel (ANIxx) as the target of A/D conversion.

(2) Analog input channel specification register (ADS)

This ADS register specifies the input channel of the analog voltage to be A/D converted.

The table below shows the related registers.

Table 11.1 Setting examples of A/D test function-related registers

Register	**Set value**	**Description**
A/D test register (ADTES)	02H 03H 00H	Selects the AV_{REFM} voltage as the target to be A/D converted. Selects the AV_{REFP} voltage as the target to be A/D converted. Selects the internal reference voltage (1.45V) as the target to be A/D converted
Analog input channel specification register (ADS)	81H	Measures the internal reference voltage output (1.45V).

11.3 Program flowchart to execute the diagnostic function

Figure 11.2 is a flowchart showing the overview of the A/D test function.

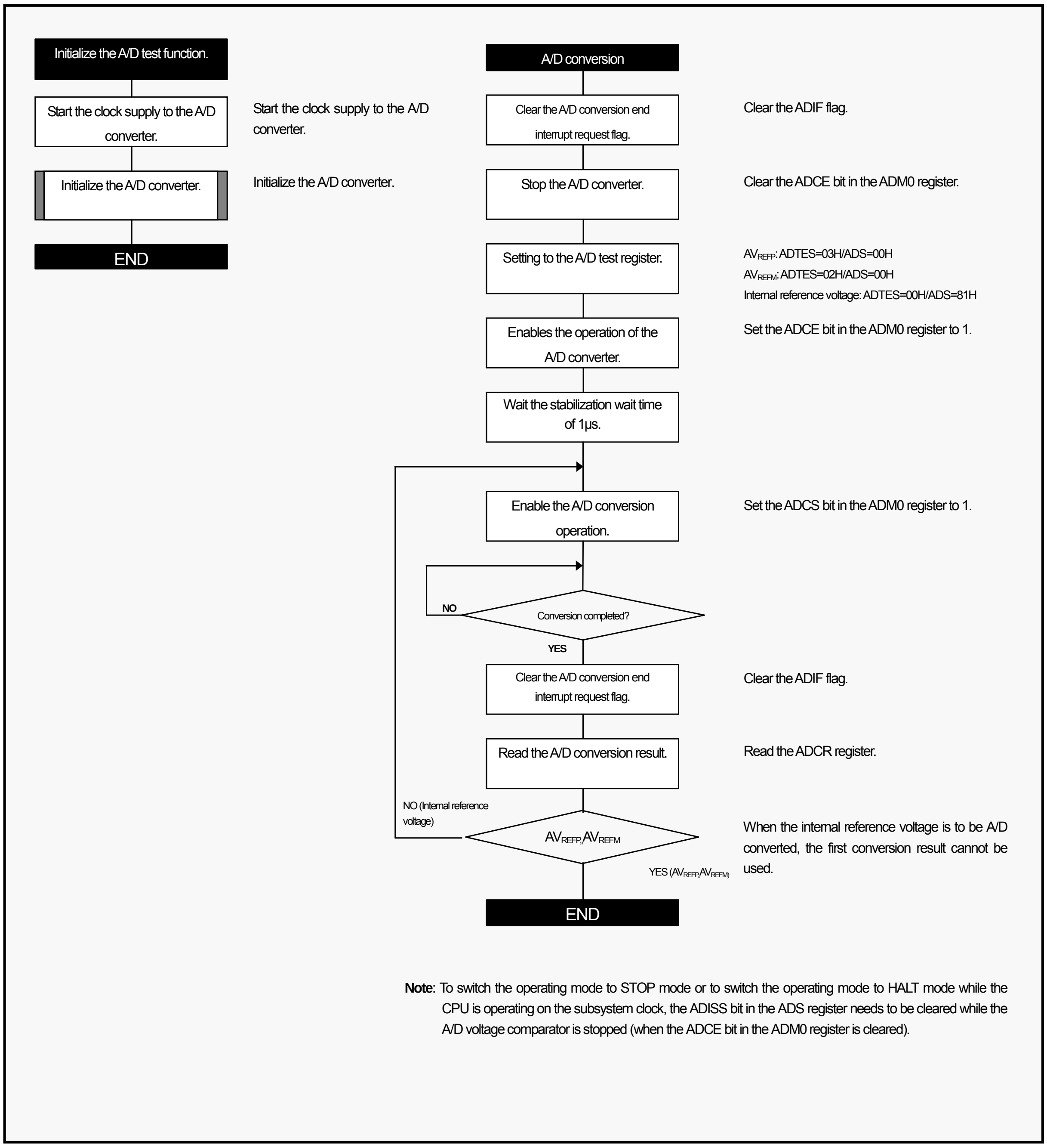

Initialize the A/D test function.
Start the clock supply to the A/D converter.
Start the clock supply to the A/D converter.
Initialize the A/D converter.
Initialize the A/D converter.
END
A/D conversion
Clear the A/D conversion end interrupt request flag.
Clear the ADIF flag.
Stop the A/D converter.
Clear the ADCE bit in the ADM0 register.
Setting to the A/D test register.
AVREFP: ADTES=03H/ADS=00H
AVREFM: ADTES=02H/ADS=00H
Internal reference voltage: ADTES=00H/ADS=81H
Enables the operation of the A/D converter.
Set the ADCE bit in the ADM0 register to 1.
Wait the stabilization wait time of 1μs.
Enable the A/D conversion operation.
Set the ADCS bit in the ADM0 register to 1.
NO
Conversion completed?
YES
Clear the A/D conversion end interrupt request flag.
Clear the ADIF flag.
Read the A/D conversion result.
Read the ADCR register.
NO (Internal reference voltage)
AVREFP, AVREFM
YES (AVREFP, AVREFM)
When the internal reference voltage is to be A/D converted, the first conversion result cannot be used.
END
Note: To switch the operating mode to STOP mode or to switch the operating mode to HALT mode while the CPU is operating on the subsystem clock, the ADISS bit in the ADS register needs to be cleared while the A/D voltage comparator is stopped (when the ADCE bit in the ADM0 register is cleared).

Figure 11.2 Program flowchart

11.4 Sample program to execute the diagnostic function

The following is a sample program for the A/D test function.

List 11-1 Sample program (A/D converter initialization)

```
void sf_adtest_initialize( void )
{
        ADMK            = BIT_SET;              // Set the INTAD interrupt mask flag.          (1)
        ADIF            = BIT_CLR;              // Clear the INTAD interrupt request flag.

        ADCEN           = BIT_SET;              // Supplies input clock for ADC.               (2)

        /*      A/D converter mode register 0 (ADM0)
                b7      : A/D conversion operation control.
                b6      : Specification of the A/D conversion channel selection mode.
                b5 :b3  : Conversion Time Selection.
                b2 :b1  : Conversion Time Selection.
                b0      : A/D voltage comparator operation control. */
        ADM0            = 0x28U;                                                               (3)

        /*      A/D converter mode register 1 (ADM1)
                b7 :b6  : Selection of the A/D conversion trigger mode.
                b5      : Specification of the A/D conversion mode.
                b4 :b2  : Reserved set to 0.
                b1 :b0  : Selection of the hardware trigger signal. */
        ADM1            = 0x20U;                                                               (4)

        /*      A/D converter mode register 2 (ADM2)
                b7 :b6  : Selection of the + side reference voltage source of
                            the A/D converter
                b5      : Selection of the - side reference voltage source of
                            the A/D converter
                b4      : Reserved set to 0.
                b3      : Checking the upper limit and lower limit conversion
                            result values
                b2      : Specification of the wakeup function (SNOOZE mode)
                b1      : Reserved set to 0.
                b0      : Selection of the A/D conversion resolution. */
        ADM2            = 0x00U;                                                               (5)

        /* Conversion result comparison upper limit setting register (ADUL) */
        ADUL            = 0xFFU;                // Set default.                                (6)

        /* Conversion result comparison lower limit setting register (ADLL) */
        ADLL            = 0x00U;                // Set default.

}
```

(1) Set the INTAD interrupt mask flag (ADMK) to 1 to disable the INTAD interrupt.
Also, clear the INTAD interrupt request flag (ADIF).

(2) Start the clock supply to the A/D converter.

(3) Set the A/D conversion time and set the A/D converter to select mode.

(4) Select software trigger mode and one-shot conversion mode.

(5) Select V_{DD} or V_{SS} as the internal reference voltage to set to 10-bit resolution.

(6) Set the ADUL register and the ADLL register to FFH and 00H, respectively.

List 11-2 Sample program (A/D conversion operation)

```
uint16_t sf_adtest_convert( uint8_t conv_sel )
{
                        uint16_t  lp_cont;
                        uint16_t  ad_value;
                        uint8_t   reg_data;
        static  uint8_t           conv_sel_old     = 0xFFU;

        if ( (conv_sel != SF_ADTEST_AVREFP) &&                                    (1)
                 (conv_sel != SF_ADTEST_AVREFM) &&
                 (conv_sel != SF_ADTEST_IAVREF) ){
                return 0xFFFFU;                 // Parameter error.
        }
        ADIF    = BIT_CLR;                      // Clear the INTAD Interrupt Request Flag.     (2)
        ADCE    = BIT_CLR;                      // Stops A/D voltage comparator operation.
        /*      A/D test register (ADTES)
                b7 :b3   : Reserved set to 0.
                b2 :b0   : A/D conversion target. */
        if ( conv_sel == SF_ADTEST_AVREFP ) {                                     (3)
                ADTES   = 0x03U;                // AD conversion of the VDD.
        } else if ( conv_sel == SF_ADTEST_AVREFM ) {
                ADTES   = 0x02U;                // AD conversion of the VSS.
        } else {
                ADTES   = 0x00U;                /* AD conversion of the internal
                                                        reference voltage. */
        }
        /*      Analog input channel specification register (ADS)
                b7       : Analog Dis-charge.
                b6 :b5   : Reserved set to 0.
                b4 :b0   : Analog input channel. */
        if ( (conv_sel == SF_ADTEST_AVREFP) || (conv_sel == SF_ADTEST_AVREFM)) {  (4)
                /* AD conversion of the VDD or VSS. */
                ADS             = 0x00U;
        } else {
                /* AD conversion of the internal reference voltage. */
                ADS             = 0x81U;
        }
        ADCE            = BIT_SET;              // Enables A/D voltage comparator operation.   (5)

        /* The software counts up to the stabilization wait time (1us). */
        for( lp_cont=0U ; lp_cont<SF_PERIOD_1US ; lp_cont++ ) {                  (6)
                NOP();
        }
        ADCS            = BIT_SET;              // Enables conversion operation.               (7)

        /* Waiting AD conversion completion. */
        do {                                                                      (8)
                reg_data = ADIF;
        } while ( reg_data!=BIT_SET );

        ADIF            = BIT_CLR;              // Clear the INTAD Interrupt Request Flag.     (9)
        ad_value = (ADCR >> 6U);                                                  (10)

        /* Ignore the first conversion data.(internal reference voltage only) */
        if ( (conv_sel == SF_ADTEST_IAVREF) &&                                    (11)
                 (conv_sel_old != SF_ADTEST_IAVREF) ) {
                ADCS    = BIT_SET;              // Enables conversion operation.
                /* Waiting AD conversion completion. */
                do {
                        reg_data = ADIF;
                } while ( reg_data!=BIT_SET );
                ADIF    = BIT_CLR;              // Clear the INTAD Interrupt Request Flag.
                ad_value = (ADCR >> 6U);
        }
        conv_sel_old    = conv_sel;
        return  ad_value;
}
```

(1) Check the arguments.

(2) Clear the INTAD interrupt request flag and the ADCE bit to stop the voltage comparator.

(3) The values set to the ADTES register vary according to the target to be A/D converted.

- A/D conversion of AV_{REFP}: 03H
- A/D conversion of AV_{REFM}: 02H
- A/D conversion of the internal reference voltage: 00H

(4) The values set to the ADS register vary according to the target to be A/D converted.

- A/D conversion of AV_{REFP}/AV_{REFM}: 00H
- A/D conversion of the internal reference voltage: 81H

(5) Set the ADCE bit to 1 to activate the voltage comparator.

(6) Wait the stabilization wait time of 1μs.

(7) Set the ADCS bit to 1 to start the A/D conversion.

(8) Monitor the INTAD interrupt request flag and wait until the A/D conversion is completed.

(9) Clear the INTAD interrupt request flag (ADIF).

(10) Read the A/D conversion result from the ADCR register.

(11) When the A/D conversion of the internal reference voltage is executed and also the previous A/D conversion is not of the internal reference voltage, repeat the procedures from Step 7 above and do not use the result of the first A/D conversion.

11.5 Program size and execution time

Size

- sf_adtest_initialize function: 27bytes

Execution time

- 812ns

Size

- sf_adtest_convert function: 169bytes

Execution time

- A/D conversion time of AV_{REFP}: 6.23μs
- A/D conversion time of AV_{REFM}: 6.62μs
- A/D conversion time of internal reference voltage: 10.40μs

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

12. Digital output signal level detection function for I/O ports

12.1 Overview of the fault diagnostic function

The IEC 60730 standard requires the validity of digital I/O ports be checked.

With the digital output signal level detection function for I/O ports, the digital output level of the pin when the port is in output mode can be read.

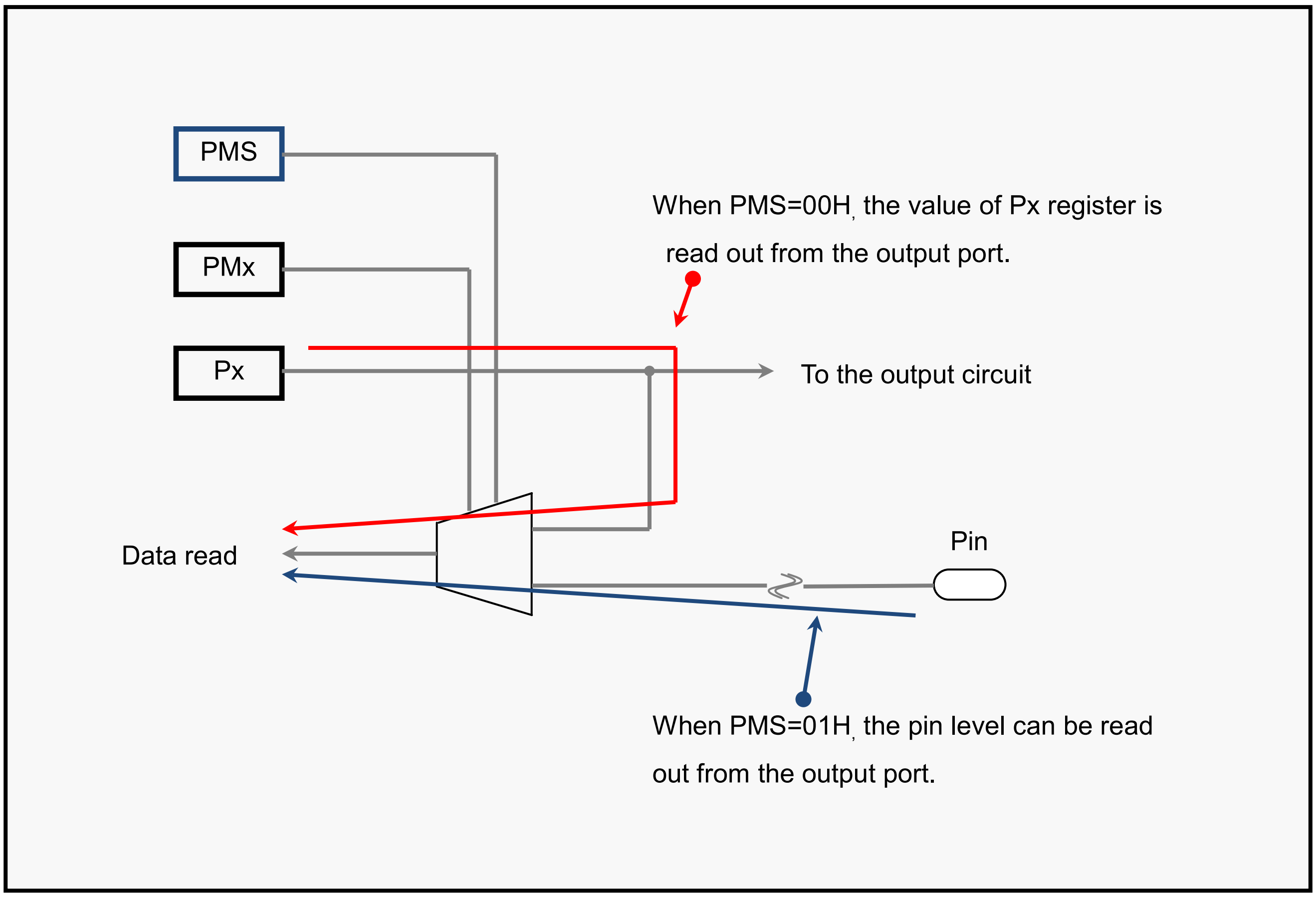

PMS
PMx
Px
When PMS=00H, the value of Px register is read out from the output port.
To the output circuit
Data read
Pin
When PMS=01H, the pin level can be read out from the output port.

Figure 12.1 Structure of digital output signal level detection function for I/O Ports

This function can check whether the values output to output ports are appropriate or not. This function can confirm whether the value (HIGH level or LOW level) output to the output port is appropriately output.

12.2 Description of the fault diagnosis-related register

The register used for the digital output signal level detection function for I/O ports is described below.

(1) Port mode select register (PMS)

This PMS register selects reading of the output latch level of a port or reading of the output level of a pin.

The table below shows the setting example of the related register.

Table 12.1 Setting example of digital output signal level detection function for I/O ports-related register

Register	Set value	Description
Port mode select register (PMS)	01H	Reads the digital output level of pins.

12.3 Sample program to execute the diagnostic function

The following is a sample program for the digital output signal level detection function for I/O ports.

List 12-1 Sample program (digital output signal level detection for I/O ports)

```
void sf_set_port_mode( uint8_t port_mode )
{
            if ( port_mode==SF_OUTPUT_PORT_READ_MODE_LATCH ) {
                        PMS         = (uint8_t)SF_OUTPUT_PORT_READ_MODE_LATCH;                   (1)
            } else {
                        PMS         = (uint8_t)SF_OUTPUT_PORT_READ_MODE_TERMINAL;                (2)
            }
}
```

(1) Setting 00H to the PMS register can read the output latch level.

(2) Setting 01H to the PMS register can read the pin level.

12.4 Program size and execution time

Size

- sf_set_port_mode function: 16bytes

Execution time

- 688ns

Conditions

- Operating frequency: PLL clock (32MHz)
- Memory model: medium

Website and Support

- Renesas Electronics Website
  http://www.renesas.com/

- Inquiries
  http://www.renesas.com/inquiry

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	June 30, 2014	-	First edition issued.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this 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 an MPU or MCU 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.

RENESAS

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

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

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

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

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

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

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

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Ku, Seoul, 135-920, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 4.0