Mixed Signal MCU

MD6603 Silicon Errata

Contents

This document describes all the silicon updates thus far to the functional specifications for the MD6603. The following usage notes and restrictions must be taken into account when designing your product.

1. Restrictions on Flash Memory Controller

1.1. Usage Notes on Erasing or Programming (Writing) Immediately after Re-protection

- **Details**

  Erasing or programming (writing) immediately after re-protection of the flash memory normally starts after a wait period. The duration of this wait period depends on the value* of the FMTIME register. Even after a wait period, erasing and programming (writing) to the flash memory can be successfully executed. If not re-executing the re-protection, no wait period occurs before subsequent erasing or programming (writing) is performed.

  * Specified by $\mathrm{FMTIME} = \frac{\mathrm{CLKFAST}}{1\times10^{6}} - 1$; if CLKFAST = 60 MHz and FMTIME = 59, the wait period is 32 ms.

- **Workaround**

  This malfunction can be avoided if not executing the re-protection.

  The LSI can perform erasing and programming (writing) to the flash memory without protection release when operated according to programs in the flash memory itself. Consequently, for the operations executed by the programs in the flash memory, the LSI does not require any protection release or re-protection on the flash memory.

- **Tool-based Workaround**

  None.

2. Restrictions on 8051 CPU

2.1. Restrictions on XDATA BUS Buffer

- **Details**

  While a program in a RAM on a bus is being executed, the functions of the XDATA BUS buffer cannot be used.

- **Workaround**

  None.

- **Tool-based Workaround**

  None.

2.2. Conflict between 16-bit Register Write and Interrupt

● **Details**

Writing to the 16-bit registers must not conflict with any interrupt acceptance; therefore, before writing to the higher bytes of the 16-bit registers (SFR BUS, XDATA BUS), set the INTMST.INTME bit to 0 to disable any interrupt acceptance. Then, insert two NOP instructions and write to the higher bytes. After writing to the higher bytes, set the INTMST.INTME bit to 1 to re-enable interrupt acceptance.

● **Workaround**

When writing data in C to a 16-bit width register which is assigned to the same address of the higher and lower bits in the SFR or the XDATA space (or both), implement the workarounds listed below.

(1) In the first line of codes in the defines.h file, define the following macro (hereafter the “macro (1)” for short).

```
#define WRITE16(RegisterName,Value)
  do{
    const uint16_t value = (Value);
    _UNSAFE_##RegisterName = (uint8_t)value;
    if(INTMST & 1){
      INTMST = INTMST & 0xfe;
      __asm__("nop");
      __asm__("nop");
      _UNSAFE_##RegisterName = (uint8_t)(value >> 8);
      INTMST = INTMST | 0x01;
    }else{
      _UNSAFE_##RegisterName = (uint8_t)(value >> 8);
    }
  }while(0)
```

(2) To prevent the 16-bit-width register from being written without the macro (1), specify const within the code that defines the registers in the defines.h file as below. By specifying the const as follows, a compile error occurs when the 16-bit width register is written without the macro (1).

```
const __sfr __at (0xC4) DSP0_R0  ;
__sfr __at (0xC4) _UNSAFE_DSP0_R0 ;
...
const __xdata __at(0xF788) uint8_t DSP0R8;
__xdata __at(0xF788) uint8_t _UNSAFE_DSP0R8;
```

(3) When writing data to the 16-bit width register, use the macro (1) as shown below.

```
WRITE16(BUF_A0_LH,  0x1234);      //SFR
WRITE16(DSP0_C0_HL, 0xabcd);      //XDATA
```

● **Tool-based Workaround**

MD Studio provides the prewritten codes (in the workarounds (1) and (2) above) within the defines.h file, which is generated when you create a new project. However, note that the defines.h file is not supplied when you create a new project without templates.

2.3. Access to RAM1 Area

- **Details**
  During the EPU operation, accessing to the 512-byte RAM1 area (0x0400 to 0x05FF) from the CPU may result in conflicts, thus causing malfunctions.

- **Workaround**
  Do not access the 512-byte RAM1 area (0x0400 to 0x05FF) from the CPU during the EPU operation.

- **Tool-based Workaround**
  None.

2.4. Access to XDATA Space

- **Details**
  During the EPU operation, accessing to addresses of the XDATA space, ranging from 0xC000 to 0xDFFF, by the CPU may result in conflicts, thus causing malfunctions.

- **Workaround**
  Do not access the XDATA space addresses (0xC000 to 0xDFF) from the CPU during the EPU operation.

- **Tool-based Workaround**
  None.

2.5. MOVX Instructions to Peripheral Registers of XDATA Space

- **Details**
  During the EPU operation, executing two consecutive MOVX instructions by the CPU to a peripheral register of the XDATA space may result in conflicts, thus causing malfunctions. The examples below show code sequences using two consecutive MOVX instructions.

  Example 1: Write the same data consecutively to the same address of the XDATA peripheral register from the CPU.

```
movx @dptr, a   (Write)
movx @dptr, a   (Write)
```

  Example 2: Write to/read from the same address of the XDATA peripheral register consecutively by the CPU.

```
movx @dptr, a   (Write)
movx a, @dptr   (Read)
```

- **Workaround**
  Inserting two or more execution cycles of a different instruction other than MOVX (e.g., NOP) between the two MOVX instructions can avoid the malfunction mentioned above.
  When the program is written in C, implement the workaround as follows. During the EPU operation, when the CPU attempts two consecutive accesses to a peripheral register of the XDATA space with the MOVX instruction, insert two NOP instructions, "`__asm__("nop");`", between the consecutive accesses by the CPU.

  Example 1: Write the same data consecutively to the same address of the XDATA peripheral register from the CPU.

```
movx @dptr, a   (Write)
nop
nop                          ← Insert two NOP instructions.
movx @dptr, a   (Write)
```

Example 2: Write to/read from the same address of the XDATA peripheral register consecutively by the CPU.

```
movx @dptr, a   (Write)
nop
nop                       ← Insert two NOP instructions.
movx a, @dptr   (Read)
```

● **Tool-based Workaround**

When you use MD Studio to implement the workaround listed below, the user-programmable workarounds described above are unnecessary.

(1) In SDCC, the code "`--peep-file md6603.peep`" is added as a compile option.

(2) The md6603.peep file is automatically created from a skeleton code and placed under the hierarchy of the source file. Make sure that the md6603.peep file is placed under the hierarchy of the source file. The md6603.peep file contains the codes as given below.

```
replace {
  movx @dptr,a
  movx @dptr,a
} by {
  movx @dptr,a
  nop
  nop
  movx @dptr,a
}
replace {
  movx @dptr,a
  movx a,@dptr
} by {
  movx @dptr,a
  nop
  nop
  movx a,@dptr
}
```

2.6. EPU Operation While CPU Is Accessing UART Register

2.6.1. EPU Access Failure by XDATA BUS Conflict

● **Details**

If the EPU attempts to access the XDATA space while the CPU is accessing the UART register, the buses will have conflicts thus the EPU access fails.

● **Workaround**

(1) Access the UART by the EPU (alternative access for the CPU; see Table 2-1)
This workaround is to permit the EPU to access the UART registers instead of the CPU. Handshaking with the CPU can control one thread of the EPU, thus permitting an alternative access to the UART registers by the EPU. Flags for handshaking, data addresses to be accessed are then read/written by the CPU and EPU from/to the following: SPR on the SFR BUS; the RAM0 (0x0000 to 0x03FF).

(2) Access the UART by the CPU (see Table 2-1)
This workaround is to perform two consecutive accesses to the XDATA space by the EPU, thus avoiding access fails due to bus conflicts. The first access succeeds or fails, but the second access succeeds. In this description, "fail" means that no access by the EPU is issued to the XDATA space. If an access fails, no writing will be executed when a write access is attempted, whereas undefined values will be read when a read access is attempted. When writing the instructions which activate the hardware (e.g. ADC trigger, EPU activation, UART transmission

and reception, DSAC trigger, PWM re-trigger, etc.) to the peripheral function registers, write an ineffective value at the first time or implement the workaround (1) above.

(2)-1 In the case where thread switching occurs while an EPU thread is being processed:

- Perform byte access to the XDATA space by the EPU
  Read/write bytes by using the word access instruction W_SA mode (i.e., two consecutive accesses to the same address). For the usage notes on the word access instruction, see Section 2.6.2 below.
- Perform word access to the XDATA space by the EPU
  Accessing the 16-bit-width peripheral function registers (i.e. consecutive accessing to the lower and higher bytes of the registers) is prohibited. When accessing the 16-bit-width peripheral function registers, implement the workaround (1) above.

(2)-2 In the case where no thread switching occurs while an EPU thread is being processed:

- When either of the following two conditions is met, perform two consecutive accesses to the XDATA space regardless of the byte or word access instruction: the condition in which only one thread is processed at once; or the condition in which no round-robin thread switching occurs while multiple threads are processed.

Table 2-1. List of Workarounds

Workaround		Instructions to Be Used	
		Byte Access Instruction by EPU	Word Access Instruction by EPU
(1)	Access by EPU to UART (Alternative Access)	Available	Available
(2)-1	Access by CPU to UART (Thread Switching Occurs)	Substitute the word access instruction (see Section 2.6.2)	Prohibited
(2)-2	Access by CPU to UART (No Thread Switching Occurs)	Execute two consecutive byte instructions	Execute two consecutive word instructions

- **Tool-based Workaround**
  MD Studio supports the restriction on the EPU operation mentioned in this section (but offers only the workaround (1) above). MD Studio provides the workaround (1) through a program automatically created from a skeleton code to permit the EPU to access the UART register.
  When accessing the UART register from the CPU, be sure to use the following functions to avoid the UART register being conflict-prone. The workaround program that permits the EPU to access the UART register consumes 44 bytes of the RAM1. Here are detailed descriptions how to use the EPU:

- This workaround program requires the EPU Thread5. Set the Thread5 priority control to fixed priority. In addition, do not write to modify the Threa5 such as for adding an instruction. Note that the Thread5 is also used for the workaround program described in Section 4.1.
- The EPU Thread5 that allows the EPU to access the UART register is normally stopped. The Thread5 operates only when the CPU calls the functions, then stops again. (This means that it stops an instruction fetch of the EPU and lowers power consumption.)
- Write an address of the UART register as an argument, addr, used in the functions provided by MD Studio. When defining a register address, use the “`ADDR_`*`UART register name`*” code defined in the errata.h file.
  The functions to be provided are as follows:

```
//Read a value from the UART register and return the value.
uint8_t UART_Reg_RD(uint16_t addr);
//Write data to the UART register.
void UART_Reg_WR(uint16_t addr, uint8_t data);
```

- Including the errata files:
  To implement workaround programs appropriately, select a setting from the following three combinations indicating which program must be enabled or disabled. The workaround programs include the UART access workaround and the sleep mode workaround (see Section 4.1). However, do not select a setting not listed below (i.e., the combination of "UART access workaround disabled; sleep mode workaround enabled"). If the improper combination is selected, the UART access workaround attempts to use the functions even during sleep mode operation, thus causing a build error.

  Enable-or-disable combinations of workaround programs:

  - ● UART access workaround enabled; sleep mode workaround enabled
    - Code used in the epu.easm file:

```
.include "errata_uart.easm"
.include "errata_sleep.easm"
```

  - ● UART access workaround enabled; sleep mode workaround disabled
    (When a new project is created, the default epu.easm file contains the workaround code below.)
    - Code used in the epu.easm file:

```
.include "errata_uart.easm"
#.include "errata_sleep.easm"
```

  - ● UART access workaround disabled; sleep mode workaround disabled
    - Code used in the epu.easm file:

```
#thread(5){
#    #Please set proper priority of thread 5.
#    priority:None
#    launch:None
#    pc:ERRATA_TH5_START
#    EIC:0
#}
#.include "errata_uart.easm"
#.include "errata_sleep.easm"
```

2.6.2. Usage Notes on Two Consecutive Accesses with Word Access Instruction

When you perform two consecutive accesses to the same address by using the word access instruction, each access will end up with two different results according to which of the accesses, read (LOADX) or write (STOREX), has been attempted.

Read Access: `LOADX.W_SA Rn, @AddrX`

- The first and second accesses succeed
- The first access fails (a LOAD value is undefined); the second access succeeds

Write Access: `STOREX.W_SA @AddrX, Rn`

- The first and second accesses succeed
- The first access fails (access is lost); the second access succeeds

If both the first and second accesses succeeded, the first write access may cause malfunctions. Therefore, the following workarounds must be implemented, for proper writing to the registers by the EPU.

- Writing to the setting register of the peripheral function:
  Write the same values at the first and second accesses.
- Clearing the flags of the peripheral function registers:
  Write 0 at the first time, then clear at the second time.
- Writing the instructions which activate the hardware (e.g., ADC trigger, EPU activation, UART transmission and reception, DSAC trigger, PWM re-trigger, etc.) to the peripheral function registers:
  Write an ineffective value, or implement the workaround (1) in Section 2.6.1.

2.7. CPU and EPU Simultaneous Access to XDATA Space

● **Details**

Malfunctions may occur when the CPU and EPU attempt simultaneous access to the XDATA space, and then the EPU accesses the XDATA space after a single-cycle instruction. This is applicable to all of the the XDATA BUS access instructions by the EPU: STOREX (B_LO, B_HI, W, W_SA) and LOADX (B_SE, B_ZE, W, W_SA). In the descriptions hereafter, the term "STOREX/LOADX instruction" represents these XDATA BUS access instructions by the EPU. The term "MOVX instruction" represents all of the XDATA BUS access instructions by the CPU (MOVX @DPTR, A; MOVX @Rm, A; MOVX A, @DPTR; MOVX A, @Rm), which are also subject to the malfunction mentioned above.

The failure example below explains that the EPU may hang up, or the CPU or EPU may lose its access after a single-cycle instruction according to the EPU and CPU access destinations in the XDATA space (Table 2-2). Losing an access means that no writing will be executed when a write access is attempted, and undefined values will be read when a read access is attempted.

[Failure Example]

When the STOREX instruction (E-1) and the MOVX instruction (C-1) attempt simultaneous access to any of the peripheral registers of the XDATA space, the XDATA BUS access of the MOVX instruction (C-3) will be lost.

The EPU attempts the second access to a peripheral register of the XDATA space after a single-cycle instruction.

```
(E-1)  STOREX.B_LO @addr, Rm
(E-2)  MOV Rn, Rm
(E-3)  STOREX.B_LO @addr, Rm
```

At the same time, the CPU also attempts the second access to a peripheral register of the XDATA space after a single-cycle instruction.

```
(C-1)  movx @dptr, a
(C-2)  nop
(C-3)  movx @dptr, a
```

Table 2-2. Access Destinations and Phenomena to Be Occurred

EPU Instruction (E-1) Access Destination	EPU Instruction (E-3) Access Destination	CPU Instruction (C-1) (Access Destinations of 1st Instruction)		
		Peripheral Register	UART	RAM0/1
Peripheral register (incl. UART)	Peripheral register (incl. UART)	(C-3) Lost[1]	(E-3) Lost[3]	Access succeeded
	RAM0/1	EPU hanged up[2]	(E-3) Lost[3]	Access succeeded
RAM0/1	XDATA space (Peripheral register, UART, RAM0/1)	Access succeeded	(E-3) Lost[3]	Access succeeded

[1] Refers that the CPU's second XDATA BUS access (C-3) after a single-cycle instruction is lost.
[2] Refers that the EPU remains hanged up until the CPU's next XDATA BUS access occurs from the EPU's second XDATA BUS access ((E-3); to the RAM0/1) after a single-cycle instruction.
[3] Refers that the EPU's second XDATA BUS access (E-3) after a single-cycle instruction is lost (i.e., the phenomenon described in Section 2.6).
When the CPU's first XDATA BUS access destination is the UART register, the workaround for the EPU's two

consecutive accesses (see Section 2.6) does not prevent losing the CPU's second XDATA BUS access (C-3) after a single-cycle instruction, as shown in the following failure example. Therefore, implement the workarounds listed in this section in addition to the workarounds for the UART listed in Section 2.6.

[Failure Example]

When the STOREX instruction (E-1) and the MOVX instruction (C-1) attempt simultaneous access to the XDATA space, the XDATA BUS accesses of the STOREX instruction (E-3) and the MOVX instruction (C-3) will be lost.

The EPU attempts two consecutive accesses to the XDATA space after a single-cycle instruction.

```
(E-1)  STOREX.B_LO @addr, Rm
(E-2)  MOV Rn, Rm
(E-3)  STOREX.B_LO @addr, Rm
(E-4)  STOREX.B_LO @addr, Rm
```

At the same time, the CPU attempts the second access to the XDATA space after a single-cycle instruction.

```
(C-1)  movx @dptr, a   (Access destination is the UART register.)
(C-2)  nop
(C-3)  movx @dptr, a
```

- **Workaround**

(1) Workaround by the EPU

As indicated in the instructions (1) to (3) in the failure examples 1 to 4, the following execution order may cause the malfunction mentioned above: (1) the STOREX/LOADX instruction, (2) a single-cycle instruction other than the STOREX/LOADX instruction, then (3) the STOREX/LOADX instruction. To avoid the malfunction, make the EPU execute the STOREX/LOADX instructions consecutively, or insert one or more instructions between the MOVX instructions so that there are at least two consecutive bus-idle-state cycles between the XDATA BUS accesses (i.e., insert two single-cycle instructions or one double-cycle instruction). For the execution cycle counts defined for each instruction, refer to the section describing the EPU operations in the MD6603 data sheet.

[Failure Examples]

Example 1:

```
(1)  STOREX.B_LO @addr, Rm
(2)  MOV Rn, Rm
(3)  STOREX.B_LO @addr, Rm
```

Example 2:

```
(1)  STOREX.B_LO @addr, Rm
(2)  ADD Rn, Rm
(3)  LOADX.B_SE Rn, @addr
```

Example 3:

```
(1)  LOADX.B_SE Rn, @addr
(2)  OR Rn, Rm
(3)  STOREX.W @addr, Rm
```

Example 4:

```
(1)  LOADX.W_SA Rn, @addr
(2)  INC Rn, Rm
(3)  LOADX.B_SE Rn, @addr
```

(1)-1 Workround for the fixed priority setting

Do not put the following in the same thread: the STOREX/LOADX instruction, a single-cycle instruction other than the STOREX/LOADX instruction, and a sequence of the STOREX/LOADX instructions.

As the workaround example 1 shows, put the STOREX/LOADX instructions consecutively, or insert two single-cycle instructions or one double-cycle instruction. The workaround example 2 is for inserting two single-cycle instructions: insert an instruction that has no operational influence, or change the instruction order if no operational problem is found.

[Workaround Examples]

Example 1:

```
(1) STOREX.B_LO @addr, R0
(2) LOADX.B_SE R1, @addr
```

Example 2:

```
(1) STOREX.B_LO @addr, R1
(2) MOV R0, R0
(3) MOV R1, R1
(4) STOREX.B_LO @addr, R0
```

When thread switching occurs, be sure to pay attention to the following condition: a certain thread contains any of the instruction sequences listed in the following failure examples and a thread with lower priority than the certain thread contains the STOREX/LOADX instruction. If the upper-priority thread is executed immediately after the STOREX/LOADX instruction in the lower-priority thread is executed, an instruction sequence leading to malfunctions may occur depending on a release timing of the WAIT instruction. To avoid such failure, check operation timings and insert either of the following between the EVTWAIT/TIMWAIT instruction and the STOREX/LOADX instruction as needed: two or more single-cycle instructions; one or more double-cycle instructions.

[Failure Examples]

Example 1:

```
(1) EVTWAIT #evt
(2) STOREX.B_LO @addr, Rm
```

Example 2:

```
(1) EVTWAIT #evt
(2) MOV Rn, Rm
(3) STOREX.B_LO @addr, Rm
```

Example 3:

```
(1) TIMWAIT #time   (Failure occurs even when #time = 0.)
(2) STOREX.B_LO @addr, Rm
```

Example 4:

```
(1) TIMWAIT #time   (Failure occurs even when #time = 0.)
(2) MOV Rn, Rm
(3) STOREX.B_LO @addr, Rm
```

[Workaround Examples]

Example 1:

```
(1) EVTWAIT #evt
(2) MOV R0, R0
(3) MOV R0, R0
(4) STOREX.B_LO @addr, R1
```

Example 2:

```
(1) TIMWAIT #time
(2) MOV R1, R1
(3) MOV R1, R1
(4) STOREX.B_LO @addr, R0
```

(1)-2 Workaround for the round-robin setting

Do not use the round-robin method when the STOREX/LOADX instruction is used in multiple threads.

(2) Workaround by the CPU

Insert one or more instructions between the MOVX instructions so that there are at least two consecutive bus-idle-state cycles between the XDATA BUS accesses when the following execution order exists in the CPU: (1) the MOVX instruction, (2) a single-cycle instruction, then (3) the MOVX instruction. Note that when the workaround by the EPU is already implemented, no workaround by the CPU is required. For the execution cycle counts defined for each instruction, refer to the section providing the 8051 CPU instruction code map in the MD6603 data sheet. However, note that the workaround by the EPU must be implemented when the EPU may hang up (as described in the footnote (2), Table 2-2) due to the following execution order existing in the EPU: (1) the STOREX/LOADX instruction, (2) a single-cycle instruction other than the STOREX/LOADX instruction, then (3) the STOREX/LOADX instruction (with the access destination to the RAM0/1).

[Workaround Example]

```
movx @dptr,a
nop              (Insert two single-cycle instructions b/w the MOVX instructions, e.g., two NOP instructions.)
nop
movx @dptr,a
```

- **Tool-based Workaround**

MD Studio provides the md6603.peep file as the workaround for when the CPU's second access is lost. This file does not contain the workaround against the possible EPU hang-up at the EPU's second access to the RAM0/1. Therefore, be sure to implement the workaround by the EPU.

(1) In SDCC, the code "`--peep-file md6603.peep`" is added as a compile option.

(2) The md6603.peep file is automatically created from a skeleton code and placed under the hierarchy of the source file.
Make sure that the md6603.peep file is placed under the hierarchy of the source file.
This .peep file replaces either of the following with the code of two single-cycle instructions inserted between the MOVX instructions: a code in which one single-cycle instruction is inserted between the MOVX instructions; a code in which the MOVX instructions are put consecutively. Here is a replacement example.

[Replacement Example]

Before:

```
    movx @dptr,a
    clr a
    movx @dptr,a
```

After:

```
    movx @dptr,a
    nop
    clr a
    movx @dptr,a
```

3. Restrictions on POC (PWM Output Controller)

3.1. Operational Restrictions on Using Clear-wait Function

- **Details**

The following restrictions are implemented when clearing the POC control in the case where the control delay addition function is used and the clear-wait function is enabled (i.e., setting the POCDTCn register to either of 0x81, 0x82, or 0x83):

(1) Auto-release by PWM is not available. Be sure to set POCCRn.ARELEN = 0.

(2) Before clearing the POC control by writing 1 to the POCSTS.PWMnF bit, read the POCSTS register to make sure that the bit to be cleared is set to 1 (or, currently being controlled). Otherwise, the LSI may result in unexpected behaviors thus malfunctions.

- **Workaround**

None.

- **Tool-based Workaround**

  None.

4. Restrictions on System Controller (SYSC)

4.1. Usage Notes on Returning from Sleep Mode

- **Details**

  When the LSI falls into the operation states listed below, the CPU does not return from the sleep mode.

(1) The condition in which the CPU receives a high-priority interrupt signal before or after writing an instruction enabling the sleep mode to a register.
(2) The condition in which the CPU writes an instruction enabling the sleep mode to a register while a low-priority interrupt is being processed.

Under the operation states listed below, the CPU returns from the sleep mode only when it receives a high-priority interrupt. However, the CPU does not return from the sleep mode when it receives a low-priority interrupt.

(1) The condition in which the CPU receives a low-priority interrupt signal before or after writing an instruction enabling the sleep mode to a register.

(2) The condition in which the CPU writes an instruction enabling the sleep mode to a register while a low-priority interrupt is being processed.

In a normal operation state (i.e., no interrupt is being processed), when the CPU receives no interrupt signal before or after writing a sleep-mode-enabling instruction to a register, it returns from the sleep mode by an interrupt signal regardless of the signal's priority level.
A high-priority interrupt is an interrupt whose bit corresponding to the INTLVLn register of the INTC is set to 1. A low-priority interrupt is an interrupt whose bit corresponding to the INTLVLn register of the INTC is set to 0.

- **Workaround**

  The following workarounds should be implemented when the sleep mode is used.

(1) Workaround before CPU entering into sleep mode
Before the CPU enters the sleep mode, disable interrupts whose generation timing is non-CPU-controllable, such as those generated by external signals input to the comparators and GPIO. When using interrupts whose generation timing is CPU-controllable, set them not to be generated during the execution of a sleep-mode-enabling instruction.
(2) Workaround for sources used when CPU returning from sleep mode
Use interrupts whose generation timing is CPU-controllable as sources for returning from the sleep mode. When using interrupts whose timing is uncontrollable as recovery sources, enable interrupts by the EPU after the CPU enters the sleep mode. The EPU cannot access the registers of the INTC; therefore, the interrupt functions of the peripheral modules which issue interrupts should be enabled or disabled individually. It is recommended to define an interrupt used for a recovery source from the sleep mode as a high-priority interrupt.

- **Tool-based Workaround**

  MD Studio provides the following functions as skeleton codes: "`void Sleep_Init(void)`" as the initialization function which executes the Goto_Sleep function; "`void Goto_Sleep(void)`" as the sleep-mode-enabling function. When using these functions, include the following two files into the epu.easm file (i.e., an EPU assembly file): the errata_uart.easm and errata_sleep.easm files.
  When the errata_sleep.easm file is output from the skeleton code, it contains the include statement coded as follows: "`#include errata_sleep.easm`". When using the sleep mode, delete the "#" symbol in the include statement mentioned above to make its code description executable.epu. When the errata_sleep.easm file is included into the epu.easm file, the LSI automatically executes the Sleep_Init() within the main function. At this time, this workaround program consumes 74 bytes of the RAM0 area. By using the user-written code that calls the Goto_Sleep() function, the LSI can properly enter and return from the sleep mode by an interrupt.

  The Goto_Sleep() function runs as follows:

(1) Obtains the status data of all the enabled interrupts from peripheral modules, then stores it into the RAM0.
(2) Disables all of the interrupts from the peripheral modules.
(3) Activates the EPU Thread5, which is also used in the workaround program in Section 2.6.1.
(4) The CPU enters the sleep mode.
(5) The Thread5 re-enables the interrupts whose status values were enabled at the time when stored in the RAM0. Then, the Thread5 enters a wait state.
(6) Returns to the main function after waking from the sleep mode.

When the errata_sleep.easm file is included, the sleep mode workaround uses 100 bytes of the RAM0. The EPU Thread5 is used as a dedicated thread for the workaround programs provided by MD Studio. Set the Thread5 priority control to fixed priority, and do not change the other settings.

When the WDT module is used as an interval timer, the Goto_Sleep() function stops the counting operation of the WDT to disable interrupts. Therefore, when the Goto_Sleep() function is executed, the count period of the WDT's interval timer becomes longer than its original setting.

However, do not execute the Goto_Sleep() function within the interrupt sequence. Otherwise, the LSI may not be able to return from the sleep mode. When implementing the sleep mode workaround, be sure to use the functions defined in Section 2.6.1, including "`UART_Reg_RD(uint16_t addr`" and "`UART_Reg_WR(uint16_t addr, uint8_t data)`", to access the UART register.

5. Restrictions on 12-bit SAR ADC

5.1. Restrictions on Sampling Cycle Settings

- **Details**

The sampling cycles listed below cannot provide proper converted values when used in the SHTIME bit, which specifies a sampling cycle of the ADNSMPmn register.

5, 37, 69, 101, 133, 165, 197, 229

- **Workaround**

The sampling cycles listed below are prohibited from using in the SHTIME bit, which specifies a sampling cycle of the ADNSMPmn register.

5, 37, 69, 101, 133, 165, 197, 229

- **Tool-based Workaround**

None.

Important Notes

Revision History

Revision	Date of Issue	Description
1.1	Apr. 06, 2018	Initial release
1.2	May 07, 2021	● Section 2.5: Workaround (Corrected) - Incorrect: Inserting a different instruction other than MOVX (e.g., NOP) between the two MOVX instructions can avoid malfunctions from such code sequences. - Correct: Inserting two or more execution cycles of a different instruction other than MOVX (e.g., NOP) between the two MOVX instructions can avoid the malfunction mentioned above. ● Section 2.6.1: Tool-based Workaround (Corrected) - Incorrect: Round-robin settings on the Thread5 and other threads are user-selectable. - Correct: Set the Thread5 priority control to fixed priority. ● Section 2.7 (Newly added) ● Section 4.1: Tool-based Workaround (Corrected) - Incorrect: Do not change the settings other than thread priorities. - Correct: Set the Thread5 priority control to fixed priority, and do not change the other settings. ● Modified some expressions and notations according to the changes in the Japanese source document.