

## Using M251 LDROM to Log System Events

Example Code Introduction for 32-bit NuMicro® Family

### Document Information

Application	This example code uses M251 LDROM to log system events.
BSP Version	M251_M252_M254_M256_M258_Series_BSP_CMSIS_V3.02.004
Hardware	NuMaker-M258KG V1.1

*The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.*

*Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.*

*All data and specifications are subject to change without notice.*

*For additional information or questions, please contact: Nuvoton Technology Corporation.*

[www.nuvoton.com](http://www.nuvoton.com)

## 1. Overview

This example code uses M251 LDROM to log system events including count of power-on, power-down, hard fault and watchdog timer (WDT) reset. Program execution addresses of hard fault and WDT reset are also recorded.

Users could dump these event logs to check if there is any unexpected behavior of their application. Also, logging execution address of hard fault and WDT reset occurred may help to find root cause of exceptions. This example code does not require additional external hardware modules.

### 1.1 Principle

Since this example code does not require additional external hardware modules, the following sections will focus on the memory layout of log region and how to log each system event in this software project.

The memory layout of the log region is shown in Figure 1-1.

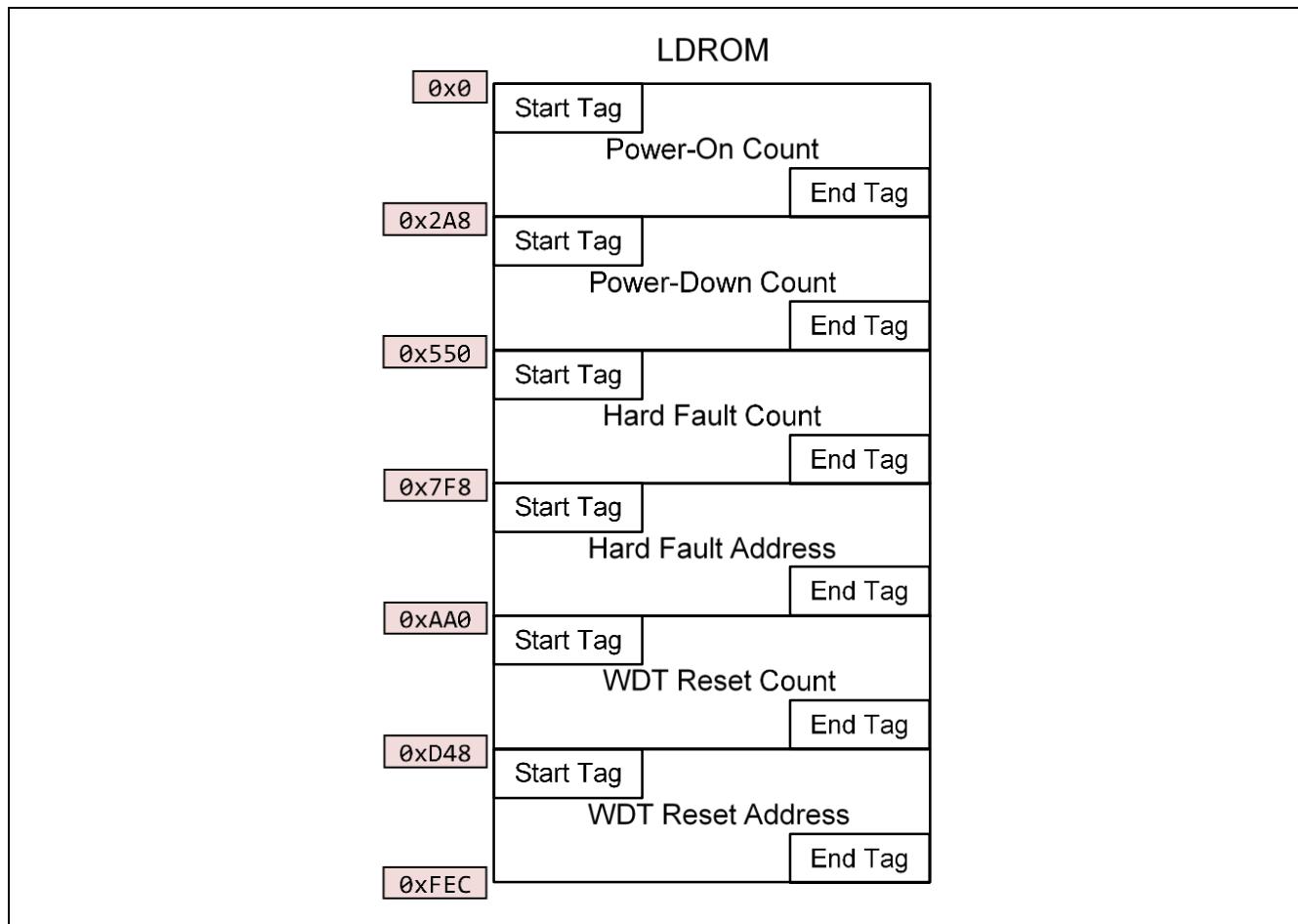


Figure 1-1 Memory Layout of Log Region

## 1.2 Workflow

Figure 1-2 shows the workflow of system event logger example code.

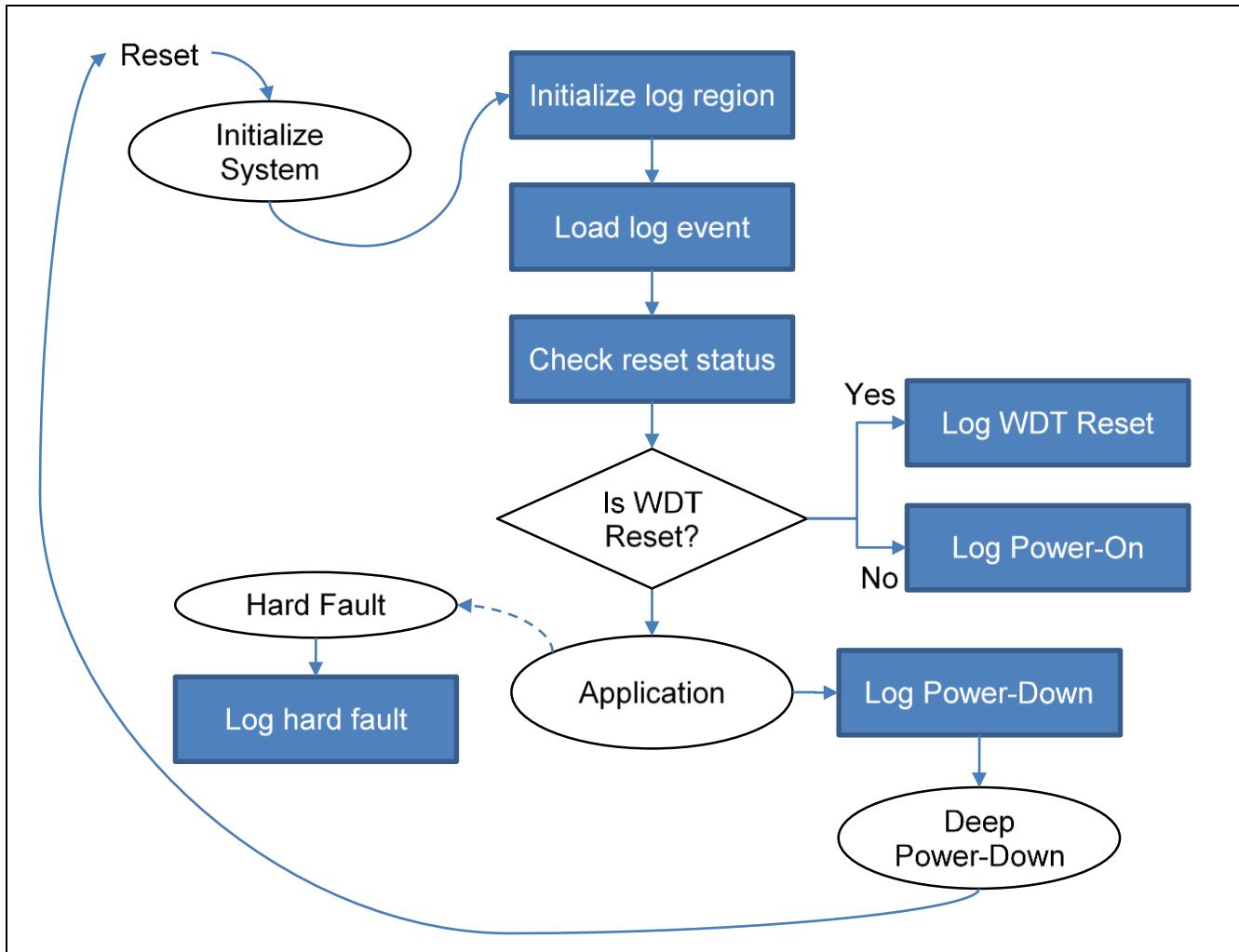


Figure 1-2 Workflow of System Event Logger

- Initialize log region

Initialize a log region with user specified start address and size. If we cannot find valid tags in specified region, try to erase whole region and write start and end tag to each system event region.

To reduce LDROM write time and prevent data loss, there is no erase operation during event log process. When writing to the event log region encounters the end flag, which indicates that the space is full, no further events can be written. In this case, it is necessary to reinitialize the log region through [2] Clear all system events in section 1.5.

- Load system event log
  - Load system event count and address from specified log region.
- Check reset status
  - Check system reset status to identify reset source. If this reset is triggered by WDT, log WDT reset and its execution address.
- Log Power-On/Power-Down
  - Log total count of Power-On or Power-Down.  
Power-On includes all reset sources (POR, nRESET, Chip Reset, DPD Wakeup, etc.) except for WDT reset.
- Log WDT Reset
  - WDT is a timer designed to prevent system halt in an unknown state therefore system needs to reset WDT periodically. If WDT is not reset in predefined timeout duration, WDT will trigger WDT reset.
- Log hard fault
  - Hard fault is an exception raised by an error operation or invalid memory access during code execution. When hard fault is raised, system will enter hard fault exception handler and we can log count and execution address of this event.

### 1.3 Log Event Count

Because the Flash write unit is 32 bits, each bit only allows writing 0 twice.

To utilize the LDROM space, this example code uses a 16-bit value to log the event count and maximum log count is 1344 (= (0x2A8 - 0x8) \* 2, referring to Figure 1-1).

Figure 1-3 shows there are 7 event counts in log.

Addr	0x00100000	0x00100004	0x00100008	0x0010000C
Data	0x00010002	0x00030004	0x00050006	0x0007FFFF
Addr	0x00100010	0x00100014	0x00100018	0x0010001C
Data	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF

Figure 1-3 Example of Count Log in Memory

### 1.4 Log Event Address

This example code supports to log application execution address when hard fault and WDT reset occurred. Users could check these addresses to debug potential issues in application.

Figure 1-4 shows that address 0x0010000C is current write header and the last saved execution address is 0x0000022A located at 0x00100008.

Addr	0x00100000	0x00100004	0x00100008	0x0010000C
Data	0x0000022A	0x0000022C	0x0000022A	0xFFFFFFFF
Addr	0x00100010	0x00100014	0x00100018	0x0010001C
Data	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF

Figure 1-4 Example of Address Log in Memory

Users did not expect hard fault or WDT reset in normal execution; therefore, they can handle it in exception handler or interrupt handler.

To retrieve execution address in exception handler, users need to get return address from stack.

Return address is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

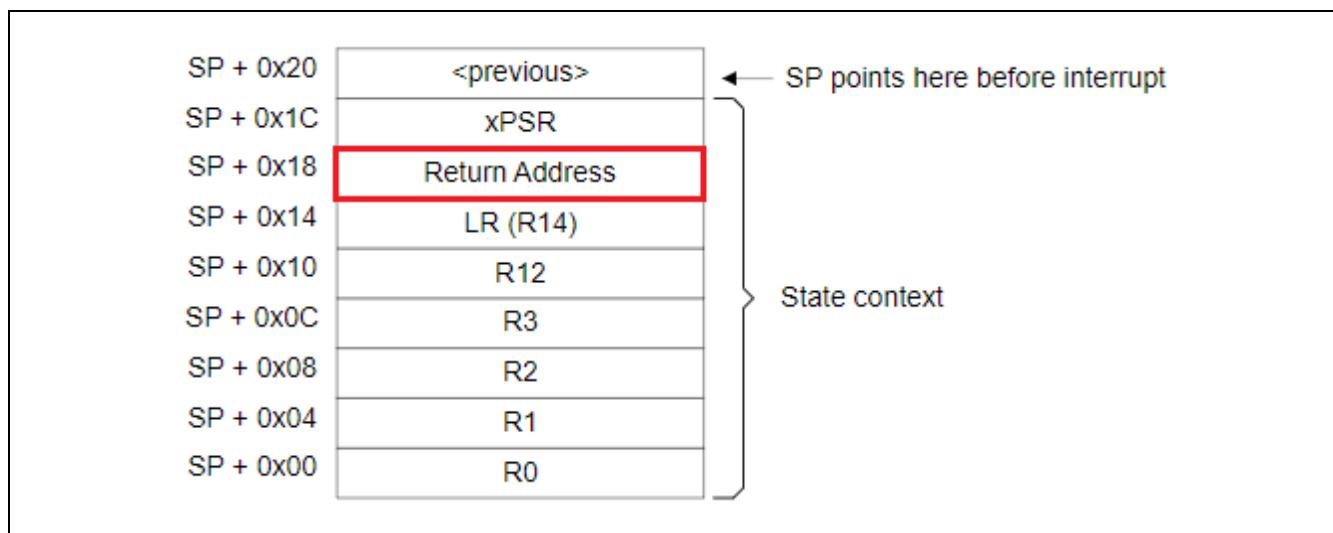


Figure 1-5 Stack Frame

(Exception entry and return: <https://developer.arm.com/documentation/dui1095/a/The-Cortex-M23-Processor/Exception-model/Exception-entry-and-return>)

Basic stack frame is illustrated in Figure 1-5 and return address is located at SP + 0x18. But different compiler's version or options might generate extra stack operations. Users **must** check stack operations in compiler output list file (*KEIL\Objects\M251\_System\_Event\_Logger.txt*) to calculate correct return address offset.

In HardFault\_Handler of this example, its stack pointer offset is 6 (Basic) + 2 (*PUSH {r7,lr}*) + 4 (*SUB sp,sp,#0x10*) = 12 as Figure 1-6. To get return address in C code, refer to HardFault\_Handler in chapter 2 Code Description.

## HardFault\_Handler

0x00000f2c:	b580	..	PUSH	{r7,lr}
0x00000f2e:	b084	..	SUB	sp,sp,#0x10
0x00000f30:	4671	qF	MOV	r1,lr
0x00000f32:	4815	.H	LDR	r0,[pc,#84] ; [0xf88] = 0x200000e8
0x00000f34:	6001	.`	STR	r1,[r0,#0]
0x00000f36:	6800	.h	LDR	r0,[r0,#0]
...				

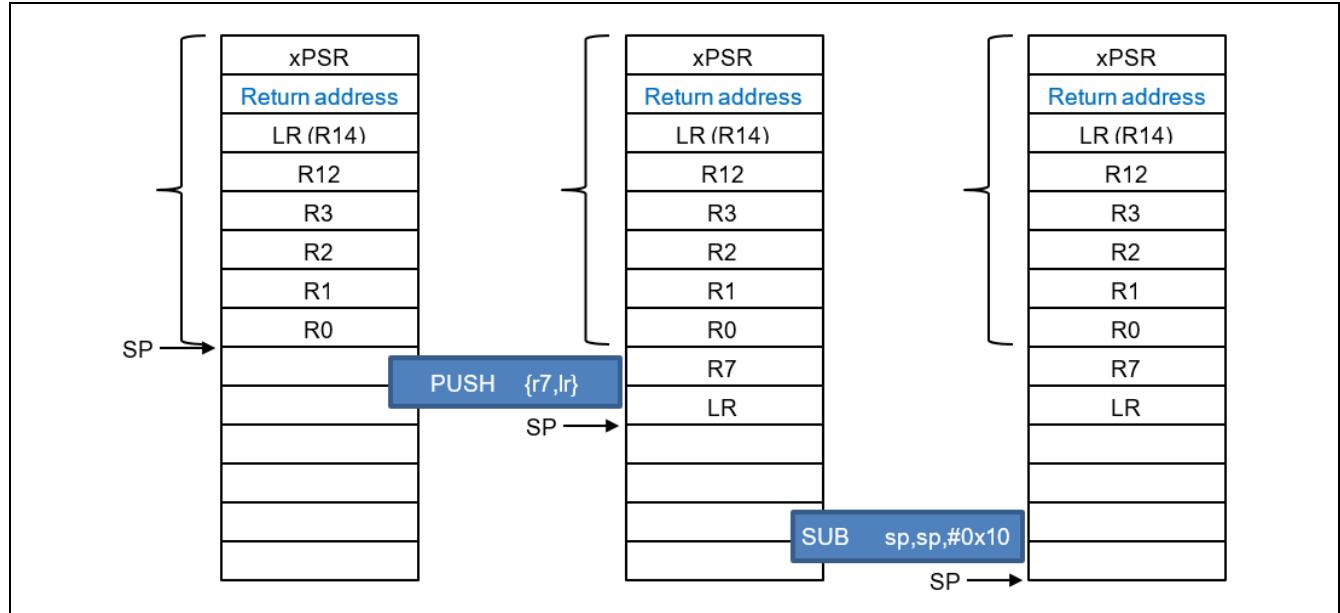


Figure 1-6 Stack Operation of HardFault\_Handler

In WDT\_IRQHandler of this example, its stack pointer offset is 6 (Basic) + 2 (SUB sp,sp,#8) = 8 as Figure 1-7. To get return address in C code, refer to WDT\_IRQHandler in chapter 2 Code Description.

## WDT\_IRQHandler

0x00001b28:	b082	..	SUB	sp,sp,#8
0x00001b2a:	4671	qF	MOV	r1,lr
0x00001b2c:	481a	.H	LDR	r0,[pc,#104] ; [0x1b98] = 0x200000e8
0x00001b2e:	6001	.`	STR	r1,[r0,#0]
0x00001b30:	6800	.h	LDR	r0,[r0,#0]
...				

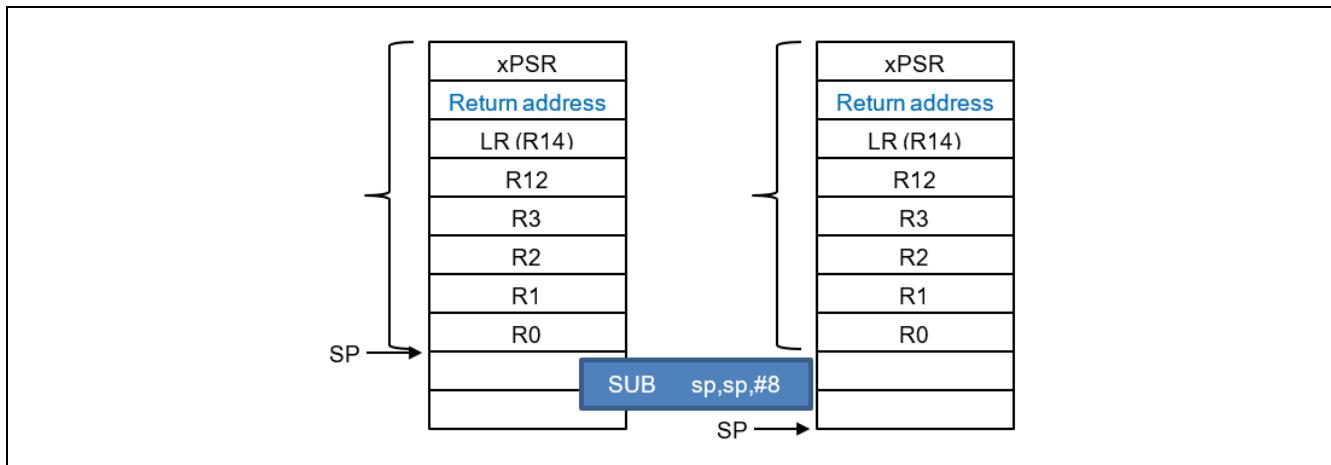


Figure 1-7 Stack Operation of WDT\_IRQHandler

## 1.5 Demo Result

Use terminal software through UART and set its baud rate to 115200. The execution menu and results will be printed out as shown below.

```
=====
System Event Logger
=====
=====
[0] Dump all system events
[1] Dump system event summary
[2] Clear all system events
[3] Trigger hard fault
[4] Trigger WDT reset
[5] Trigger power down
[6] Reset chip
=====
```

- [0] Dump all system events

Dump event counts and all event occurred execution addresses.

```
Select: 0
-----
[ PowerOn Count ] 2
[ PowerDown Count ] 2
[ HardFault Count ] 5
[ HardFault Addr ]
  [ 5 ] 0x0000188C  [ 4 ] 0x0000188C
  [ 3 ] 0x0000188C  [ 2 ] 0x00001886
  [ 1 ] 0x0000187E
[ WDT Reset Count ] 2
[ WDT Reset Addr ]
  [ 2 ] 0x00000F22  [ 1 ] 0x00000F14
-----
```

- [1] Dump system event summary

Only dump event count and last event occurred execution address.

```
Select: 1
```

```
[ PowerOn Count ] 2
[ PowerDown Count ] 2
[ HardFault Count ] 5
[ HardFault Addr ] 0x0000188C
[ WDT Reset Count ] 2
[ WDT Reset Addr ] 0x00000F22
```

- [2] Clear all system events

Erase whole log region and re-initialize it.

- [3] Trigger hard fault

This example demos three methods to raise hard fault.

To raise hard fault – Input 0 ~ 2 to test log different hard fault execution address.

```
Select: 3
```

```
[0] M32(0) = 0;
[1] M32(FMC_LDROM_BASE + FMC_LDROM_SIZE) = 0;
[2] M32(FMC_APROM_BASE + 0x1);
```

- [4] Trigger WDT reset

To raise WDT reset without output.

- [5] Trigger power down

Enter Deep Power-down mode and set PC.0 from 0 to 1 to exit power down.

```
Select: 5
```

```
Enter deep power down mode
```

```
Set PC.0 from 0 to 1 to exit power down
```

- [6] Reset chip

Reset whole chip without output.

## 2. Code Description

This example code provides basic functions for event log operation which are located in *System\_Event\_Logger.c* and *System\_Event\_Logger.h*.

- Constant Definition

```
/* Specify default data region tag */
#define SEL_DEFAULT_TAG      0x4E554843
/*
 * TRUE: Skip duplicated system event in the same boot cycle
 * FALSE: Log all system events in the same boot cycle
 */
#define SEL_SKIP_DUP_EVENT   (FALSE)
/* Dump how many address per line */
#define SEL_DUMP_ADDR_CNT    (2)
```

- Enumeration

```
/* Error code */
typedef enum
{
    eSEL_ERRCODE_SUCCESS          = 0,
    eSEL_ERRCODE_INVALID_BASE_ADDR = -1,
    eSEL_ERRCODE_ERASE_FAIL       = -2,
    eSEL_ERRCODE_WRITE_FAIL       = -3,
    eSEL_ERRCODE_INVALID_TAG      = -4,
    eSEL_ERRCODE_DATA_FULL        = -5,
    eSEL_ERRCODE_SKIP_WRITE       = -6,
    eSEL_ERRCODE_INVALID_PARAM    = -7,
} E_SEL_ERRCODE;

/* Data region type */
typedef enum
{
    eSEL_EVENT_TYPE_POWER_ON_CNT   = 0,
    eSEL_EVENT_TYPE_POWER_DOWN_CNT,
    eSEL_EVENT_TYPE_ADDR_IDX,
    eSEL_EVENT_TYPE_HARD_FAULT_CNT = eSEL_EVENT_TYPE_ADDR_IDX,
    eSEL_EVENT_TYPE_HARD_FAULT_ADDR,
    eSEL_EVENT_TYPE_WDT_RESET_CNT,
    eSEL_EVENT_TYPE_WDT_RESET_ADDR,
    eSEL_EVENT_TYPE_CNT
} E_SEL_EVENT_TYPE;
```

- Data Structure

```

typedef struct
{
    uint32_t u32BaseAddr;          /* Base address to save system event */
    uint32_t u32ByteSize;          /* Byte size to save system event */
    uint32_t u32WriteHead;         /* Current write header */
    uint16_t u16DataLogCount;      /* Current data log count */
    uint16_t bLogAddr;             /* TURE: Log address, FALSE: Log count */
} S_SEL_DATA_REGION;

/* Total need 16 + 16 * eSEL_EVENT_TYPE_CNT = 112 bytes */
typedef struct
{
    uint32_t u32Tag;               /* Tag to identify start/end of region */
    uint32_t u32BaseAddr;          /* Base address to save system event */
    uint32_t u32ByteSize;          /* Byte size to save system event */
    uint32_t u32LogMask;           /* To prevent multiple log in the same boot */
    S_SEL_DATA_REGION sDataRegion[eSEL_EVENT_TYPE_CNT];
} S_SEL_DATA_LOGGER;

```

- Global variables

s\_sDataLogger is internal data structure to keep log status.

g\_astrDataType is constant string array of each data region for log output.

```

static S_SEL_DATA_LOGGER s_sDataLogger = { 0 };
const char *g_astrDataType[eSEL_EVENT_TYPE_CNT] =
{
    "PowerOn Count", "PowerDown Count", "HardFault Count", "HardFault Addr", "WDT Reset
    Count", "WDT Reset Addr"
};

```

- Initialize log region

Erase whole region and write start/end tag to each data region.

```

/**
 * @details This function is used to init specified log region in APROM or LDROM to
 save system event logs.
 *
 * @param[in] u32BaseAddr Page alignment base address of log region.
 * @param[in] u32ByteSize Page alignment byte size of log region.
 * @param[in] u32Tag Specify custom tag or use default tag \ref SEL_DEFAULT_TAG to
 indicate start and end of different log.
 * @param[in] bForceInit Force to initialize log region.
 *
 * @retval eSEL_ERRCODE_SUCCESS
 *         Init log region successfully
 * @retval eSEL_ERRCODE_INVALID_BASE_ADDR
 *         Invalid base address
 * @retval eSEL_ERRCODE_ERASE_FAIL
 *         Failed to erase log region
 */
int32_t SEL_Init(uint32_t u32BaseAddr, uint32_t u32ByteSize, uint32_t u32Tag, uint32_t
bForceInit)
{
    int32_t i;
    uint32_t u32Addr, u32DataSize;

    if ((u32BaseAddr % FMC_FLASH_PAGE_SIZE) != 0)
        return eSEL_ERRCODE_INVALID_BASE_ADDR;
}

```

```
/* Region size is divided equally to data region size (4 bytes alignment) */
u32DataSize = (u32ByteSize / eSEL_EVENT_TYPE_CNT) & ~0x3;

/* Forced to erase whole log region to re-initialize if bForceInit == TRUE */
if (bForceInit)
{
    /* Erase specified flash region */
    for (u32Addr = u32BaseAddr; u32Addr < (u32BaseAddr + u32ByteSize); u32Addr += FMC_FLASH_PAGE_SIZE)
    {
        if (FMC_Erase(u32Addr) != 0)
        {
            return eSEL_ERRCODE_ERASE_FAIL;
        }
    }

    memset((void *)&s_sDataLogger, 0x0, sizeof(s_sDataLogger));
}

/* Initialize base address and size of each data log region */
for (i = eSEL_EVENT_TYPE_POWER_ON_CNT; i < eSEL_EVENT_TYPE_CNT; i++)
{
    s_sDataLogger.sDataRegion[i].u32ByteSize = u32DataSize;

    if (i > 0)
    {
        /* Other data region => Base address = end address of previous data region */
        s_sDataLogger.sDataRegion[i].u32BaseAddr = s_sDataLogger.sDataRegion[i - 1].u32BaseAddr + s_sDataLogger.sDataRegion[i - 1].u32ByteSize;
    }
    else
    {
        /* First data region => Base address = whole log region base address */
        s_sDataLogger.sDataRegion[i].u32BaseAddr = u32BaseAddr;
    }

    /* Check start/end tag of each data log region */
    if ((inp32(s_sDataLogger.sDataRegion[i].u32BaseAddr) != u32Tag) ||
        (inp32(s_sDataLogger.sDataRegion[i].u32BaseAddr +
s_sDataLogger.sDataRegion[i].u32ByteSize - 4) != (u32Tag + 1)))
    )
    {
        /* No valid start/end tag => Write start/end tag */
        FMC_Write(s_sDataLogger.sDataRegion[i].u32BaseAddr, u32Tag);
        /* Simply set start tag + 1 as end tag */
        FMC_Write((s_sDataLogger.sDataRegion[i].u32BaseAddr +
s_sDataLogger.sDataRegion[i].u32ByteSize - 4), (u32Tag + 1));
        s_sDataLogger.sDataRegion[i].u32WriteHead =
s_sDataLogger.sDataRegion[i].u32BaseAddr + 4;
    }

    /* Set data region type to log address */
    if (i >= eSEL_EVENT_TYPE_ADDR_IDX)
        s_sDataLogger.sDataRegion[i].bLogAddr = TRUE;
}

s_sDataLogger.u32BaseAddr = u32BaseAddr;
```

```
s_sDataLogger.u32ByteSize = u32ByteSize;
s_sDataLogger.u32Tag      = u32Tag;
s_sDataLogger.u32LogMask = 0;

return eSEL_ERRCODE_SUCCESS;
}
```

- Load system event log

```
/** @details This function is used to load system event log from log region in APROM or LDROM.
 * @retval eSEL_ERRCODE_SUCCESS:
 *          Load
 *          eSEL_ERRCODE_INVALID_TAG
 *          Cannot find valid tag
 */
int32_t SEL_LoadSystemEvent(void)
{
    int32_t i;
    uint32_t u32Offset, u32Addr, u32Data;

    /* Traverse each data region to get write header and data count */
    for (i = eSEL_EVENT_TYPE_POWER_ON_CNT; i < eSEL_EVENT_TYPE_CNT; i++)
    {
        u32Offset = s_sDataLogger.sDataRegion[i].u32BaseAddr;

        /* Check base address of this data region contains start tag */
        if (inp32(u32Offset) != s_sDataLogger.u32Tag)
            return eSEL_ERRCODE_INVALID_TAG;

        /* Traverse whole data region until read end tag or 0xFFFFFFFF */
        for (u32Addr = u32Offset + 4; u32Addr < (u32Offset +
s_sDataLogger.sDataRegion[i].u32ByteSize - 4); u32Addr += 4)
        {
            u32Data = inp32(u32Addr);

            /* Reach end of data region */
            if (u32Data == (s_sDataLogger.u32Tag + 1))
                break;

            if (s_sDataLogger.sDataRegion[i].bLogAddr)
            {
                if (u32Data != 0xFFFFFFFF)
                {
                    /* Valid address log => data log count + 1 */
                    s_sDataLogger.sDataRegion[i].u16DataLogCount++;
                }
                else
                {
                    /* Set write header */
                    s_sDataLogger.sDataRegion[i].u32WriteHead = u32Addr;
                    break;
                }
            }
            else
            {
                if ((u32Data & 0xFFFF) != 0xFFFF)

```

```

    {
        /* Valid count is in low 16 bits */
        s_sDataLogger.sDataRegion[i].u16DataLogCount = u32Data & 0xFFFF;
    }
    else
    {
        if ((u32Data >> 16) != 0xFFFF)
        {
            /* Valid count is in high 16 bits */
            s_sDataLogger.sDataRegion[i].u16DataLogCount = u32Data >> 16;
        }
        /* Set write header */
        s_sDataLogger.sDataRegion[i].u32WriteHead = u32Addr;
        break;
    }
}
}

return eSEL_ERRCODE_SUCCESS;
}

```

- Dump system event log

User can customize log output in this function. Please refer to in section 1.5.

[0] Dump all system events or [1] Dump system event summary

```

/**
 * @details This function is used to dump system event log from log region in APROM or
 * LDROM.
 *
 * @param[in] bDumpAll To dump detail log or not
 *                   TRUE: Dump all logs
 *                   FALSE: Dump log count and latest address
 *
 * @retval eSEL_ERRCODE_SUCCESS:
 *         Success
 */
int32_t SEL_DumpSystemEvent(uint32_t bDumpAll)
{
    int32_t i, j;

    printf("-----\n");

    for (i = eSEL_EVENT_TYPE_POWER_ON_CNT; i < eSEL_EVENT_TYPE_CNT; i++)
    {
        /* Check if this region is for address and its count > 0 */
        if (s_sDataLogger.sDataRegion[i].bLogAddr &&
s_sDataLogger.sDataRegion[i].u16DataLogCount)
        {
            /* If bDumpAll == TRUE, print all log address */
            if (bDumpAll)
            {
                printf(" [ %-15s ]\n", g_astrDataType[i]);

                for (j = 0; j < s_sDataLogger.sDataRegion[i].u16DataLogCount; j++)
                {
                    if ((j > 0) && (j % SEL_DUMP_ADDR_CNT) == 0)

```

```

        printf("\n");

        printf("      [%3d ] 0x%08X",
(s_sDataLogger.sDataRegion[i].u16DataLogCount - j),
inp32(s_sDataLogger.sDataRegion[i].u32WriteHead - 4 * (j + 1)));
    }

        printf("\n");
    }
else
{
    /* If bDumpAll == FALSE, print latest log address */
    printf(" [ %-15s ] 0x%08X\n", g_astrDataType[i],
inp32(s_sDataLogger.sDataRegion[i].u32WriteHead - 4));
}
else
{
    /* Print data log count only */
    printf(" [ %-15s ] %d\n", g_astrDataType[i],
s_sDataLogger.sDataRegion[i].u16DataLogCount);
}
}

printf("-----\n");

return eSEL_ERRCODE_SUCCESS;
}

```

- Log count

Invoke this function to log count of specified data event.

```

/** @details This function is used to log system event occurrence count.
 * @param[in] eEventType System event type
 * @retval eSEL_ERRCODE_SUCCESS:
 *           Log count successfully
 *           eSEL_ERRCODE_DATA_FULL
 *           Data region is full
 *           eSEL_ERRCODE_WRITE_FAIL
 *           Failed to write data
 */
int32_t SEL_LogCount(E_SEL_EVENT_TYPE eEventType)
{
    uint32_t u32Offset, u32Data;
    uint16_t u16Count;

    if (s_sDataLogger.sDataRegion[eEventType].bLogAddr == TRUE)
        return eSEL_ERRCODE_INVALID_PARAM;

    /* Set SEL_SKIP_DUP_EVENT to TRUE to skip duplicated event log */
#if (SEL_SKIP_DUP_EVENT == TRUE)
    if (s_sDataLogger.u32LogMask & (1 << eEventType))
        return eSEL_ERRCODE_SKIP_WRITE;
#endif

```

```

u16Count = s_sDataLogger.sDataRegion[eEventType].u16DataLogCount + 1;
u32Offset = s_sDataLogger.sDataRegion[eEventType].u32WriteHead;
u32Data = inp32(u32Offset);

/* Check if log reaches end of data region */
if (u32Data == (s_sDataLogger.u32Tag + 1))
    return eSEL_ERRCODE_DATA_FULL;

if (u32Data == 0xFFFFFFFF)
{
    /* Write count to high 16 bits */
    u32Data = ((u16Count << 16) | 0xFFFF);
}
else
{
    /* Write count to low 16 bits */
    u32Data = (u32Data & 0xFFFF0000) | u16Count;
}

FMC_Write(u32Offset, u32Data);

if (FMC_GET_FAIL_FLAG() || (g_FMC_i32ErrCode != eFMC_ERRCODE_SUCCESS))
{
    FMC_CLR_FAIL_FLAG();
    return eSEL_ERRCODE_WRITE_FAIL;
}

/* Set new write header and update data log count */
/* Check low 16 bits of this word has been updated => Switch to next word */
if ((u32Data & 0xFFFF) != 0xFFFF)
    s_sDataLogger.sDataRegion[eEventType].u32WriteHead = u32Offset + 4;

s_sDataLogger.sDataRegion[eEventType].u16DataLogCount = u16Count;

/* Set SEL_SKIP_DUP_EVENT to TRUE to skip duplicated event log */
#if (SEL_SKIP_DUP_EVENT == TRUE)
    s_sDataLogger.u32LogMask |= (1 << eEventType);
#endif

return eSEL_ERRCODE_SUCCESS;
}

```

- Log address

Invoke this function to log address of specified data event.

```

/** 
 * @details This function is used to log system event occurred address.
 *
 * @param[in] eEventType System event type
 * @param[in] u32PC System event occurred address
 *
 * @retval eSEL_ERRCODE_SUCCESS:
 *         Success
 *         eSEL_ERRCODE_INVALID_PARAM
 *         Specify region is not for address
 *         eSEL_ERRCODE_DATA_FULL
 *         Data region is full
 *         eSEL_ERRCODE_SKIP_WRITE
 *         Skip this write if u32PC == 0xFFFFFFFF or u32LogMask is set
 */

```

```
*          eSEL_ERRCODE_WRITE_FAIL
*          Failed to write data
*/
int32_t SEL_LogAddr(E_SEL_EVENT_TYPE eEventType, uint32_t u32PC)
{
    uint32_t u32Offset, u32Data;

    /* Check specified event type is for address */
    if (s_sDataLogger.sDataRegion[eEventType].bLogAddr == FALSE)
        return eSEL_ERRCODE_INVALID_PARAM;

    /* Set SEL_SKIP_DUP_EVENT to TRUE to skip duplicated event log */
#if (SEL_SKIP_DUP_EVENT == TRUE)
    if (s_sDataLogger.u32LogMask & (1 << eEventType))
        return eSEL_ERRCODE_SKIP_WRITE;
#endif

    u32Offset = s_sDataLogger.sDataRegion[eEventType].u32WriteHead;
    u32Data = inp32(u32Offset);

    /* Check not reach end of data region */
    if (u32Data == (s_sDataLogger.u32Tag + 1))
        return eSEL_ERRCODE_DATA_FULL;

    if (u32Data != 0xFFFFFFFF)
        return eSEL_ERRCODE_SKIP_WRITE;

    FMC_Write(u32Offset, u32PC);

    if (FMC_GET_FAIL_FLAG() || (g_FMC_i32ErrCode != eFMC_ERRCODE_SUCCESS))
    {
        FMC_CLR_FAIL_FLAG();
        return eSEL_ERRCODE_WRITE_FAIL;
    }

    /* Set new write header and add data log count */
    s_sDataLogger.sDataRegion[eEventType].u32WriteHead = u32Offset + 4;
    s_sDataLogger.sDataRegion[eEventType].u16DataLogCount++;

    /* Set SEL_SKIP_DUP_EVENT to TRUE to skip duplicated event log */
#if (SEL_SKIP_DUP_EVENT == TRUE)
    s_sDataLogger.u32LogMask |= (1 << eEventType);
#endif

    return eSEL_ERRCODE_SUCCESS;
}
```

- HardFault\_Handler

Use SEL\_LogCount and SEL\_LogAddr to log count and address in handler.

```
void HardFault_Handler(void)
{
    __ASM volatile("mov %0, lr\n" : "=r"(g_u32LR));

    if (g_u32LR & BIT2)
        g_pu32SP = (uint32_t*) __get_PSP();
    else
        g_pu32SP = (uint32_t*) __get_MSP();
```

Refer to section 1.4 to know how to get return address from stack.

```
/* Get information from stack */
/* Check assembly code to get correct stack pointer offset to get return address */
g_u32HardFaultPC = g_pu32SP[12];
SEL_LogCount(eSEL_EVENT_TYPE_HARD_FAULT_CNT);
SEL_LogAddr(eSEL_EVENT_TYPE_HARD_FAULT_ADDR, g_u32HardFaultPC);
g_pu32SP[12] += 2;
```

- WDT\_IRQHandler

WDT is a dedicated timer used to monitor application execution and perform WDT reset when system runs into an unknown state.

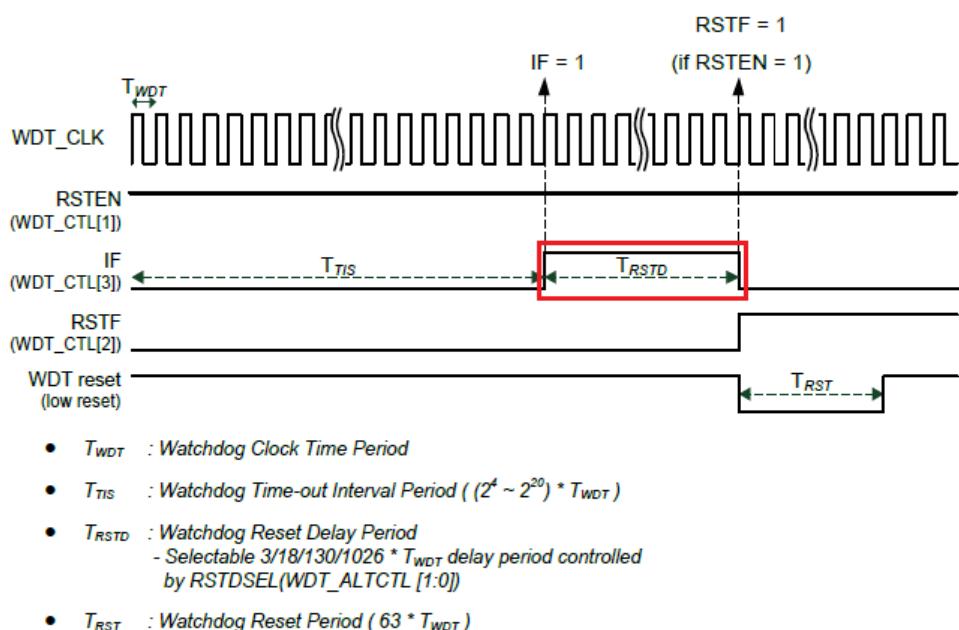


Figure 2-1 WDT Reset Period Timing

In Figure 2-1, when  $T_{TIS}$  timeout occurred WDT will trigger the timeout interrupt (IF = 1) and application has to reset WDT counter in  $T_{RSTD}$  period to prevent WDT to reset system. If system runs into unknown state or halt due to some errors, system cannot reset WDT counter in time and WDT trigger system immediately after  $T_{RSTD}$  timeout.

It might not have sufficient time to log count or address to Flash. Execution address is saved into a temporary variable **g\_u32WDTResetPC** placed in section “**.bss.noinit**” and logged after WDT reset.

```
volatile uint32_t g_u32WDTResetPC __attribute__((section(".bss.noinit")));
```

In some cases, execution address cannot be saved correctly. e.g. System is blocking in other exception/IRQ handler with higher priority than WDT\_IRQHandler.

```
void WDT_IRQHandler(void)
{
    __ASM volatile ("mov %0, lr\n" : "=r"(g_u32LR));

    if (g_u32LR & BIT2)
        g_pu32SP = (uint32_t*) __get_PSP();
    else
        g_pu32SP = (uint32_t*) __get_MSP();
```

Refer to section 1.4 to know how to get return address from stack.

```
/* Get information from stack */
/* Check assembly code to get correct stack pointer offset to get return address */
g_u32WDTResetPC = g_pu32SP[8];

if (WDT_GET_TIMEOUT_INT_FLAG() == 1)
{
    /* For demo purpose - simply skip reset WDT counter to trigger WDT reset */
    if (g_u32TriggerWDTReset == FALSE)
        WDT_RESET_COUNTER();

    /* Clear WDT time-out interrupt flag */
    WDT_CLEAR_TIMEOUT_INT_FLAG();
}
```

- Test menu

```
void TestMenu(void)
{
    char chItem, chSubItem;

    printf("=====\\n");
    printf(" [0] Dump all system events          \\n");
    printf(" [1] Dump system event summary       \\n");
    printf(" [2] Clear all system events         \\n");
    printf(" [3] Trigger hard fault             \\n");
    printf(" [4] Trigger WDT reset              \\n");
    printf(" [5] Trigger power down            \\n");
    printf(" [6] Reset chip                   \\n");
    printf("=====\\n");
    chItem = getchar();
    printf("Select: %c\\n", chItem);

    switch(chItem - '0')
    {
        case 0:
            SEL_DumpSystemEvent(TRUE);
```

```
        break;
    case 1:
        SEL_DumpSystemEvent(FALSE);
        break;
    case 2:
        if (SEL_Init(SEL_LOG_BASE_ADDR, SEL_LOG_BYTE_SIZE, SEL_DEFAULT_TAG, TRUE) != eSEL_ERRCODE_SUCCESS)
            printf("Failed to init system event logger !\n");
        break;
    case 3:
        /* Test three ways to trigger hard fault */
        printf("=====\\n");
        printf(" [0] M32(0) = 0; \\n");
        printf(" [1] M32(FMC_LDROM_BASE + FMC_LDROM_SIZE) = 0; \\n");
        printf(" [2] M32(FMC_APROM_BASE + 0x1); \\n");
        printf("=====\\n");
        chSubItem = getchar();
        printf("Select: %c\\n", chSubItem);
        switch (chSubItem - '0')
        {
            case 0:
                M32(0) = 0;
                break;
            case 1:
                M32(FMC_LDROM_BASE + FMC_LDROM_SIZE) = 0;
                break;
            default:
                M32(FMC_APROM_BASE + 0x1);
                break;
        }
        break;
    case 4:
        g_u32TriggerWDTRestart = TRUE;
        break;
    case 5:
    {
        uint32_t u32PowerDownMode = CLK_PMUCTL_PDMSEL_DPD;

        SEL_LogCount(eSEL_EVENT_TYPE_POWER_DOWN_CNT);
        CLK_SetPowerDownMode(u32PowerDownMode);
        CLK_EnableDPDWKPIn(CLK_DPDWKPIN_0, CLK_DPDWKPIN_RISING);

        printf("Enter deep power down mode\\n");
        printf(" Set PC.0 from 0 to 1 to exit power down \\n");
        UART_WAIT_TX_EMPTY(UART0);

        /* Enter Power-down mode */
        CLK_PowerDown();
        printf("Wake up\\n");
        break;
    }
    default:
        UART_WAIT_TX_EMPTY(UART0);
        SYS_ResetChip();
        break;
}
```

{}

- Initialize WDT

```
void WDT_Init(void)
{
    /* Reset temporary address */
    g_u32WDTResetPC = 0xFFFFFFFF;
    /* Because of all bits can be written in WDT Control Register are write-protected;
       To program it needs to disable register protection first. */
    SYS_UnlockReg();

    /* Configure WDT settings and start WDT counting */
    WDT_Open(WDT_TIMEOUT_2POW16, WDT_RESET_DELAY_1026CLK, TRUE, FALSE);

    /* Enable WDT interrupt function */
    WDT_EnableInt();
    /* Enable WDT NVIC */
    NVIC_EnableIRQ(WDT_IRQn);
}
```

- main function

Figure 1-2 illustrates basic workflow.

```
int main()
{
    SYS_Init();
    /* Init UART to 115200-8n1 for print message */
    UART_Open(UART0, 115200);
    SYS_UnlockReg();
    /* Enable ISP and CONFIG/LDROM/APROM update function */
    FMC_Open();
    FMC_ENABLE_CFG_UPDATE();
    FMC_ENABLE_LD_UPDATE();
    FMC_ENABLE_AP_UPDATE();
    /* Config wake up pin 0 - PC.0 as input */
    GPIO_SetMode(PC, BIT0, GPIO_MODE_INPUT);
    GPIO_SetPullCtl(PC, BIT0, GPIO_PUSEL_PULL_UP);

    /* Enable IAP function to access LDROM */
    if (((FMC->ISPSTS & FMC_ISPSTS_CBS_Msk) >> FMC_ISPSTS_CBS_Pos) & 1)
    {
        uint32_t au32Config[3];

        FMC_ReadConfig(au32Config, 3);
        if (au32Config[0] & 0x40)
        {
            au32Config[0] &= ~0x40;
            FMC_Erase(FMC_CONFIG_BASE);
            FMC_WriteConfig(au32Config, 3);

            printf("Wait chip reset done to enable IAP to access LDROM ... \n");
            UART_WAIT_TX_EMPTY(UART0);
            SYS_ResetChip();
            while (1)
                ;
        }
    }
}
```

```
printf("\n\n");
printf("=====System Event Logger\n");
printf("=====

if (SEL_Init(SEL_LOG_BASE_ADDR, SEL_LOG_BYTE_SIZE, SEL_DEFAULT_TAG, FALSE) != eSEL_ERRCODE_SUCCESS)
    printf("Failed to init system event logger !\n");

if (SEL_LoadSystemEvent() != eSEL_ERRCODE_SUCCESS)
    printf("Failed to load system event !\n");
```

If reset is triggered by WDT, log count and address. Or log count as normal power on.

```
if ((SYS_GetResetSrc() & SYS_RSTSTS_WDTRF_Msk) && WDT_GET_RESET_FLAG())
{
    /* Log WDT reset count and address if reset is trigger by WDT */
    SEL_LogCount(eSEL_EVENT_TYPE_WDT_RESET_CNT);
    SEL_LogAddr(eSEL_EVENT_TYPE_WDT_RESET_ADDR, g_u32WDTRestPC);
    g_u32WDTRestPC = 0xFFFFFFFF;
    SYS_ClearResetSrc(SYS_RSTSTS_WDTRF_Msk);
    WDT_CLEAR_RESET_FLAG();
}
else
    SEL_LogCount(eSEL_EVENT_TYPE_POWER_ON_CNT);

WDT_Init();
/* Generate Systick interrupt each 10 ms */
SysTick_Config(SystemCoreClock / 100);

while (1)
{
    /* TestMenu is for demo purpose and can be replaced by user application code. */
    TestMenu();
}
```

### 3. Software and Hardware Requirements

#### 3.1 Software Requirements

- BSP version
  - M251\_M252\_M254\_M256\_M258\_Series\_BSP\_CMSIS\_V3.02.004
- IDE version
  - Keil uVersion 5.37

#### 3.2 Hardware Requirements

- Circuit components
  - NuMaker-M258KG V1.1
- Pin Connect
  - Connect UART0 TX (PB.13) pin to PC UART RX to show menu and execution results of this example code and UART0 RX (PB.12) pin to PC UART TX to select demo test item.
  - Connect Wake Up PIN0 (PC.0) to GND.

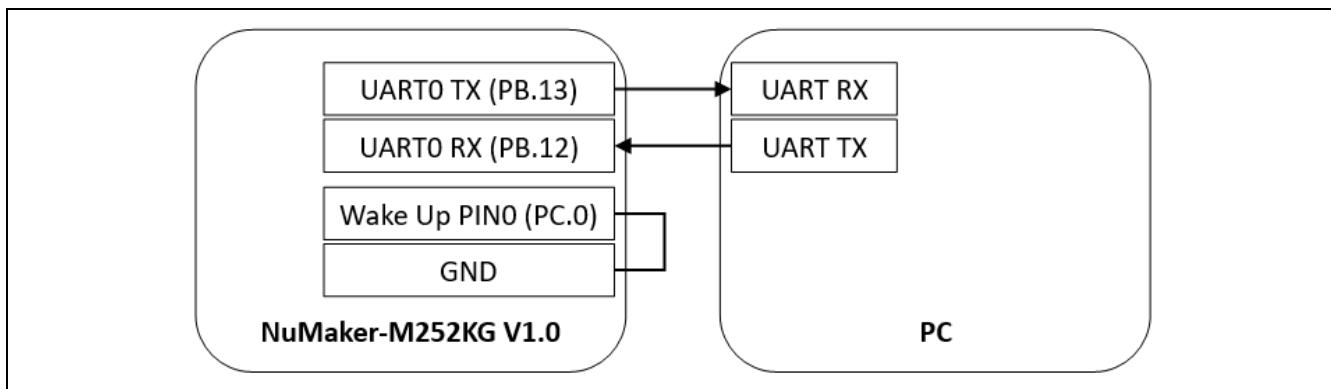


Figure 3-1 Pin Connect

## 4. Directory Information

The directory structure is shown below.

 EC_M251_System_Event_Logger_V1.00	
 Library	Sample code header and source files
 CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
 Device	CMSIS compliant device header file
 StdDriver	All peripheral driver header and source files
 SampleCode	
 ExampleCode	Source file of example code

Figure 4-1 Directory Structure

## 5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M251\_System\_Event\_Logger.uvprojx*.
2. Enter Keil compile mode.
  - Build
  - Download
  - Start/Stop debug session
3. Enter debug mode.
  - Run

## 6. Revision History

Date	Revision	Description
2023.07.15	1.00	Initial version.

### Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

---

Please note that all data and specifications are subject to change without notice.  
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.