使用M251 LDROM记录重要系统事件

NuMicro® 32位系列微控制器范例代码介绍

文件信息

应用简述	本范例代码展示使用 M251 LDROM 记录重要系统事件
BSP 版本	M251_M252_M254_M256_M258_Series_BSP_CMSIS_V3.02.004
开发平台	NuMaker-M258KG V1.1

1. 概述

以下范例程序是使用 M251 LDROM 用于记录系统事件，包括开机次数、休眠次数、硬件错误 (Hard fault) 和看门狗定时器 (WDT) 重置次数。同时也会记录硬件错误和看门狗定时器重置的程序执行地址。

用户可以将这些事件记录输出以检查应用程序是否有任何意外行为。而记录硬件错误和看门狗定时器重置发生时的程序执行地址，有助于寻找异常的根本原因。

这个范例程序代码不需要额外的外部硬件模块。

1.1 原理

由于这个范例程序代码不需要额外的外部硬件模块，接下来的部分将着重介绍记录区域的内存布局以及如何在这个软件项目中记录每个系统事件。记录区域的内存布局如图 1-1 所示。

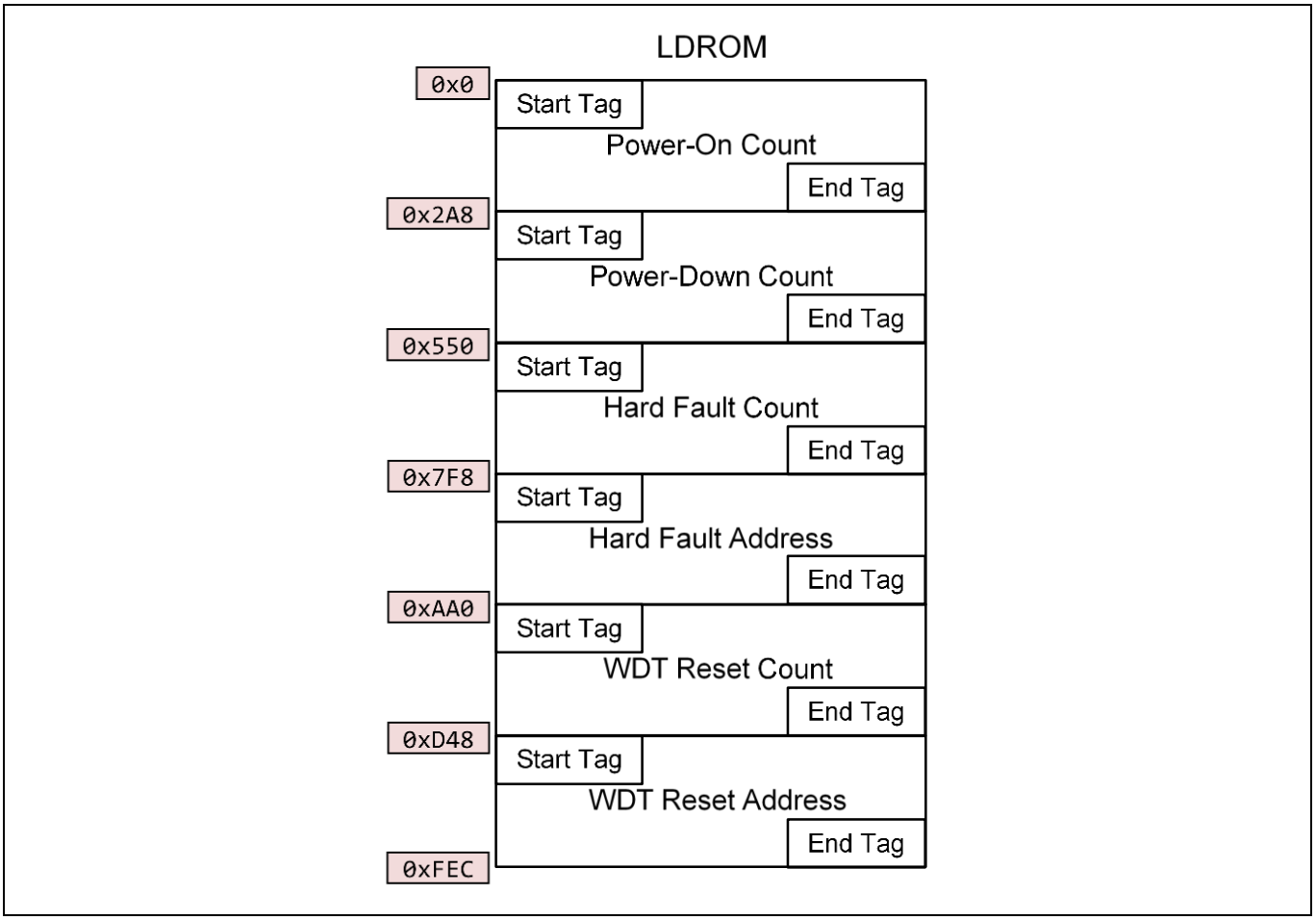

图 1-1 日志区域的内存布局

1.2 工作流程

图 1-2 展示系统事件记录器范例程序代码工作流程

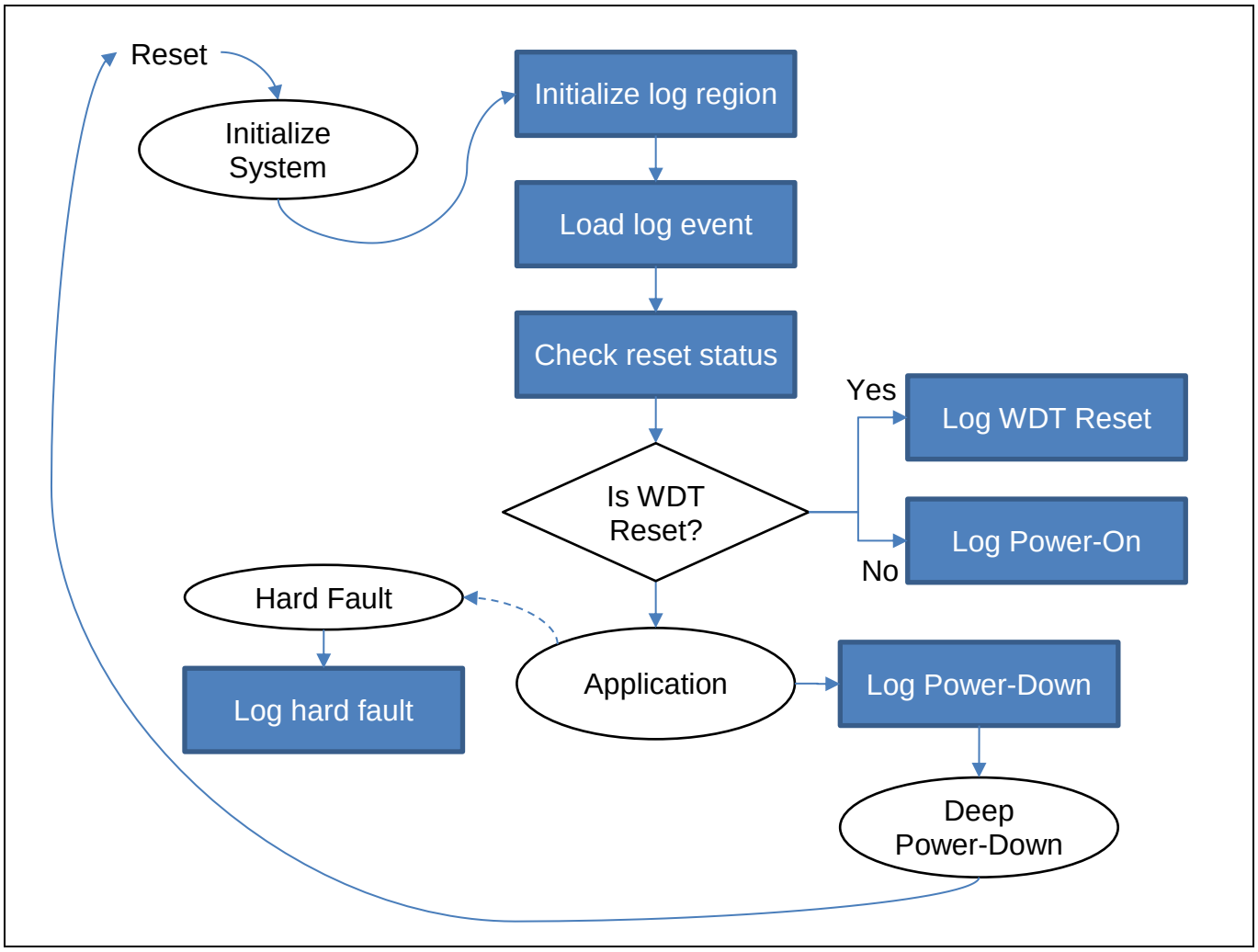

图 1-2 系统事件记录器工作流程

- 初始化日志区域 (Initialize log region)

  用户可以指定起始地址和大小来初始化日志区域。如果在指定的区域找不到有效的卷标，则尝试清除整块区域并在每块系统事件区域写入起始和结束标志。

  为了减少 LDROM 的写入时间和避免数据遗失，在事件日志记录过程中不会进行擦除操作。当写入事件日志区域碰到结束标志表示空间已满，便无法再写入新事件，需要重新初始化日志区域。在范例程序中，可使用章节 1.5 執行結果的选项 [2] Clear all system events 来清除资料。

- 加载系统事件日志 (Load log event)

  从指定的日志区域加载系统事件计数和其发生地址。

- 检查重置状态标志 (Check reset status)

  检查系统重置状态以识别重置来源。如果此重置是由看门狗定时器触发的，则记录看门狗定时器重置及其执行地址。

- 记录开机/休眠事件 (Log Power-On/Power-Down)

  记录开机或休眠的总次数。

  开机事件包括所有的重置来源（包括 POR、nRESET、Chip Reset、DPD Wakeup 等），但不包括 WDT Reset。

- 记录看门狗定时器重置 (Log WDT Reset)

  看门狗定时器是一个定时器，旨在防止系统停止在未知状态下。因此，系统需要定期重置 WDT。如果在预定的超时持续时间内未重置 WDT，WDT 将触发 WDT 重置。

- 记录硬件错误 (Log hard fault)

  在程序执行期间由于错误操作或无效的内存访问引发的异常情况。当发生错误时，系统将进入错误异常处理程序，此时可以记录该事件的次数和执行地址。

1.3 记录事件发生次数

因为 Flash 写入单位为 32 位，而且每一位只允许写入两次 0。

为了利用 LDROM 空间，这个范例程序使用 16 位的值来记录事件计数且最大的记录数值为 1344（等于（0x2A8 - 0x8）* 2，参考图 1-1）。图 1-3 的范例显示有 7 次事件计数。

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

图 1-3 计数日志范例

1.4 记录事件发生执行地址

这个范例程序代码支持在硬件错误和看门狗定时器重置发生时记录应用程序的执行地址。用户可以检查这些地址来寻找应用程序中可能存在的问题。

图 1-4 显示地址 0x0010000C 是目前的写入标头，最后一次储存的执行地址为 0x0000022A，位于 0x00100008 位置。

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

图 1-4 地址日志记录范例

程序正常执行中用户并不预期发生硬件错误或看门狗定时器重置，因此我们可以在异常处理程序或中断处理程序中处理这些情况。

要在异常处理程序中撷取执行地址，用户需要从堆栈中取得返回地址。

返回地址是被中断程序中下一条指令的地址。这个值在异常返回时被还原到程序计数器 (PC) 上，以便中断的程序继续执行。

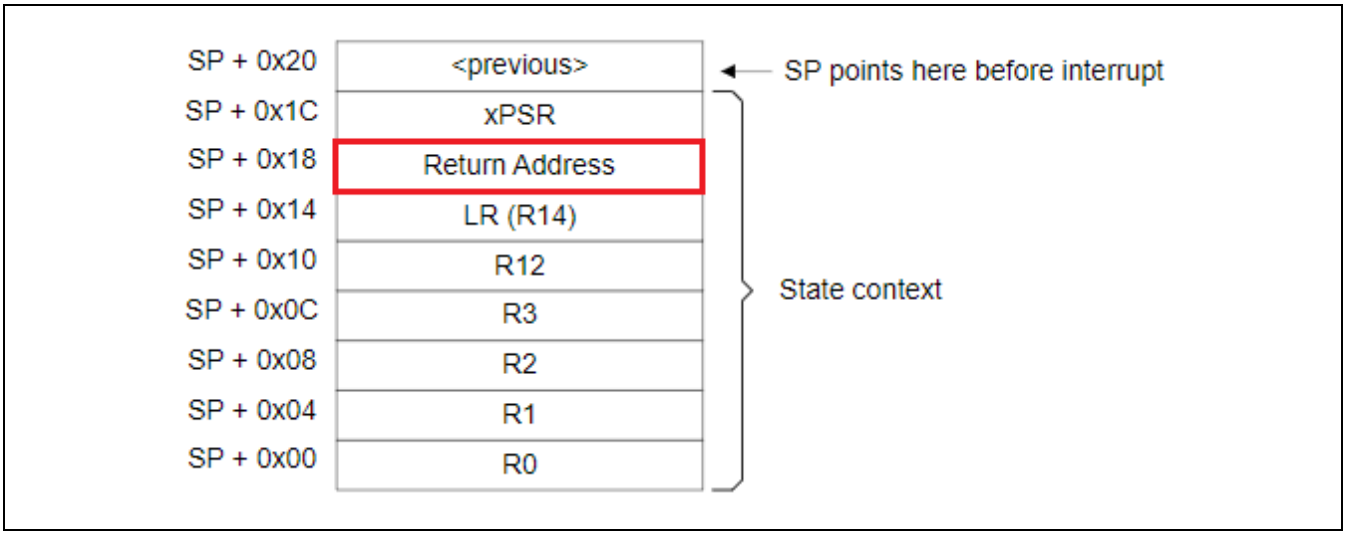

图 1-5 堆栈框架

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

基本的堆栈框架如图 1-5 所示，而返回地址位于 SP + 0x18 的位置。但是因为不同编译程序版本或参数可能会产生不同的堆栈框架操作，使用者必须检查编译程序输出的列表档案 (KEIL\Objects\M251_System_Event_Logger.txt) 中的堆栈操作，以计算正确的返回地址偏移量。

在本范例 HardFault_Handler 中，它的堆栈指针偏移量为 6（基本偏移量）+ 2（*PUSH {r7, lr}*）+ 4（*SUB sp, sp, #0x10*）= 12 如图 1-6。要在 C 程序代码中获取返回地址，请参考代码介绍中的 HardFault_Handler。

```
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]
        ...
```

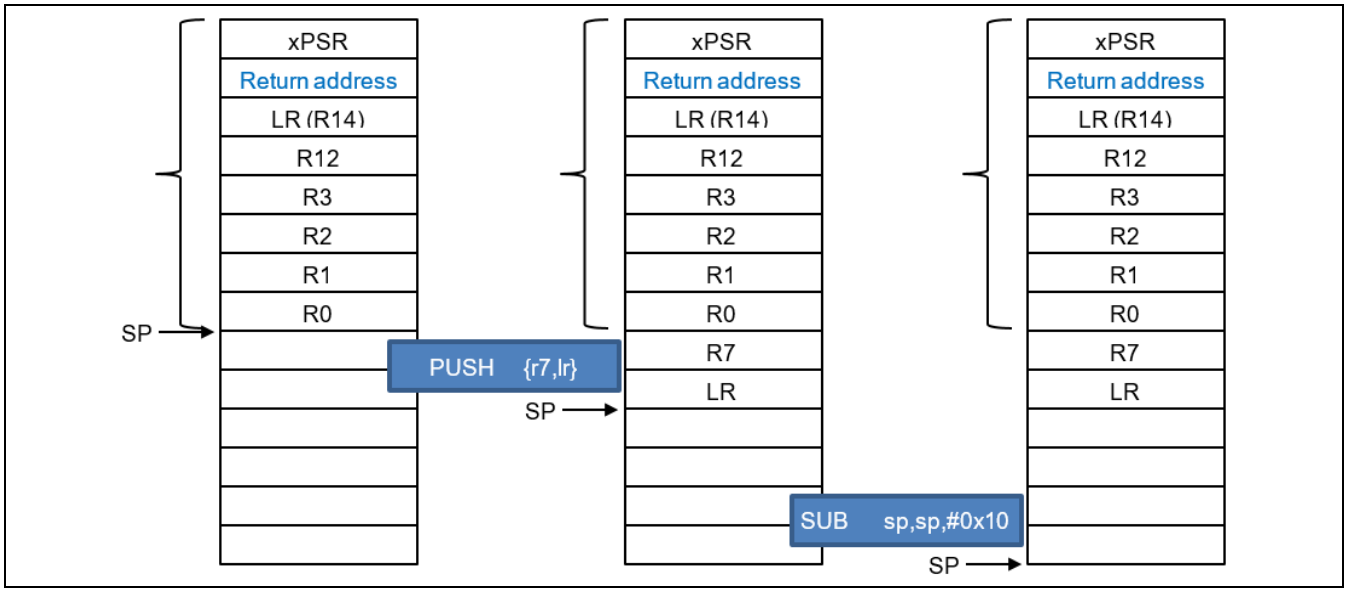

图 1-6 HardFault_Handler 堆栈框架操作

在本范例 WDT_IRQHandler 中，它的堆栈指针偏移量为 6（基本偏移量）+ 2（*SUB sp, sp, #8*）= 8 如图 1-7。要在 C 程序代码中获取返回地址，请参考代码介绍中的 WDT_IRQHandler。

```
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]
        ...
```

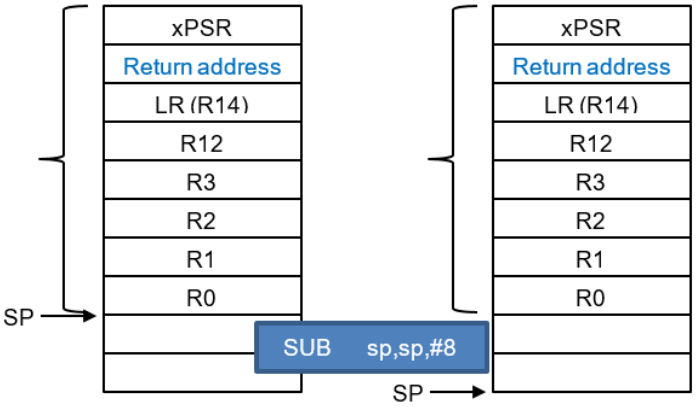

图 1-7 WDT_IRQHandler 堆栈框架操作

1.5 执行结果

使用 UART 透过终端软件连接并将其波特率设定为 115200。执行选单和结果将如下所示输出。

```
================================================
              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

  输出事件计数和所有事件发生的执行地址。

```
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

  只输出事件计数和最后一次事件发生的执行地址。

```
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

  清除整个日志区域并重新初始化。

- [3] Trigger hard fault

  这个范例展示三种引发硬件错误的方法。

  输入 0～2 来测试记录不同的硬件错误执行地址。

```
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

  触发看门狗定时器重置，无信息输出。

- [5] Trigger power down

  进入深度省电模式，将 PC.0 从 0 变为 1 以退出省电模式。

```
Select: 5
Enter deep power down mode
  Set PC.0 from 0 to 1 to exit power down
```

- [6] Reset chip

  重置系统，无信息输出。

2. 代码介绍

这个范例程序代码提供了事件日志操作的功能函数，代码位于 System_Event_Logger.c 和 System_Event_Logger.h。

- 常数定义

```
/* 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)
```

- 枚举定义

```
/* 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;
```

- 数据结构定义

```
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;
```

- 全局变量

s_sDataLogger 是用于记录日志状态的内部数据结构。g_astrDataType 是每块数据区域的常数字符串数组，用于日志输出。

```
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"
};
```

- 初始化日志区域

清除整块日志区域并将起始和结束卷标写入每块数据区域。

```
/**
  * @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
  *             eSEL_ERRCODE_INVALID_BASE_ADDR
  *                 Invalid base address
  *             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;
}
```

- 加载系统事件日志

```
/**
  * @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;
}
```

- 输出系统事件记录

使用者可修改此函数产生自定义的输出结果。输出结果可以参考章节 1.5 [0] Dump all system events 或[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 == FLASE, 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;
}
```

- 记录计数

  调用此函数以记录指定数据事件的计数。

```
/**
  * @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;
}
```

- 记录地址

  调用此函数以记录指定数据事件的发生地址。

```
/**
  * @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

在硬件错误中断处理函数取得错误地址并调用 SEL_LogCount 和 SEL_LogAddr 记录次数和地址。

```
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();
```

参考章节 1.4 了解如何从堆栈中取得返回地址

```
    /* 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 是一个专门的定时器，可用于监视应用程序的执行情况，负责在系统进入未知状态时执行 WDT 重置。

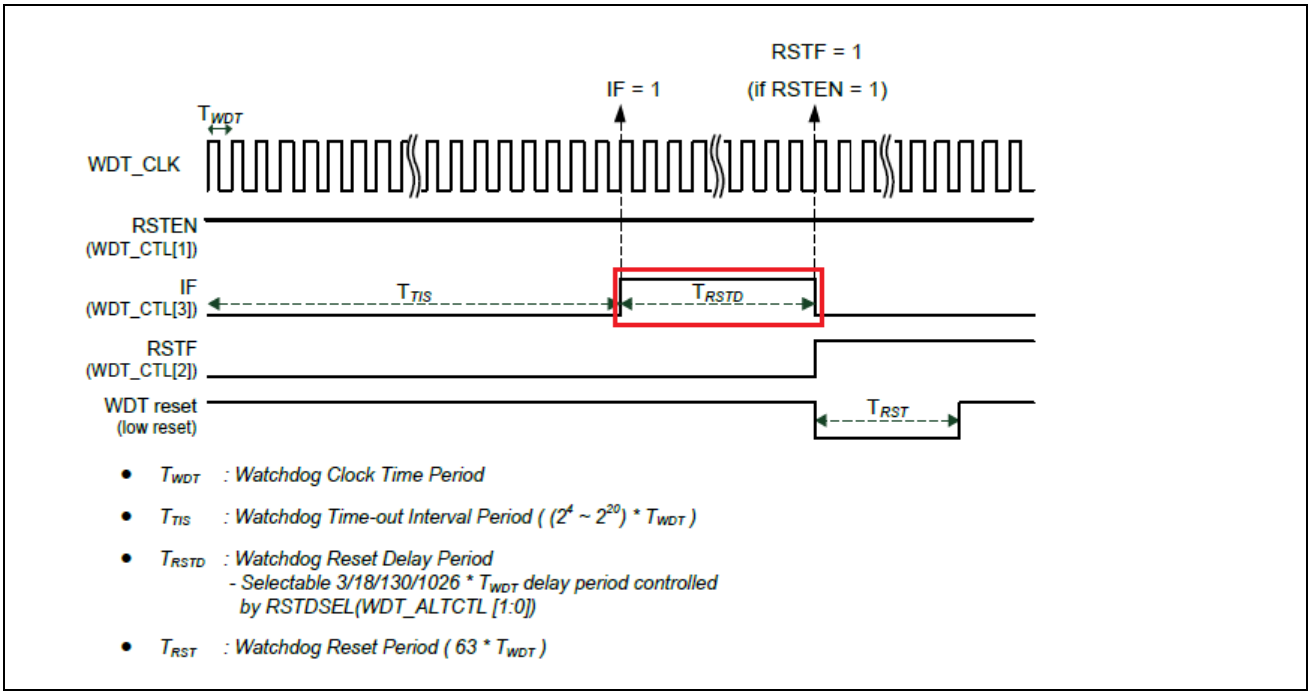

图 2-1 WDT 重置时间时序

在图 2-1 中：当 T_{TIS}（看门狗定时器超时）发生时，WDT 将触发超时中断（IF = 1），应用程序必须在 T_{RSTD} 周期内重置 WDT 计数器，以防止 WDT 重置系统。如果系统因某些错误进入未知状态或停止，系统无法及时重置 WDT 计数器，在 T_{RSTD} 超时后，WDT 立即触发系统重置。

此时可能没有足够的时间将计数或地址记录到 LDROM。执行地址会先保存在配置于.bss.noinit 区段的临时变量 g_u32WDTResetPC 中，在系统重置后进行记录。

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

在某些情况下，执行地址可能无法被正确保存。例如，系统被阻塞在其他具有比 WDT_IRQHandler 更高优先级的异常处理程序或 IRQ 处理程序中。

```
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();
```

参考章节 1.4 了解如何从堆栈中取得返回地址

```
    /* 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();
    }
}
```

- 测试选单

```
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_u32TriggerWDTReset = 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;
    }
}
```

- 初始化 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);
}
```

- 主程序

参考图 1-2 描述的基本工作流程

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

    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");
```

如果该次系统重置是由看门狗定时器触发的，则记录计数和地址。否则将会视为正常的开机事件记录开机次数。

```
    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_u32WDTResetPC);
        g_u32WDTResetPC = 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. 软件与硬件需求

3.1 软件需求

- BSP 版本
  - M251_M252_M254_M256_M258_Series_BSP_CMSIS_V3.02.004
- IDE 版本
  - Keil uVersion 5.37

3.2 硬件需求

- 电路组件
  - NuMaker-M258KG V1.1
- 接线示意图
  - 将 UART0 TX（PB.13）脚连接到 PC UART RX，以显示此范例程序代码的选单和执行结果；将 UART0 RX（PB.12）脚连接到 PC UART TX，以选择测试项目。
  - 将 Wake Up PIN0（PC.0）脚连接到 GND。

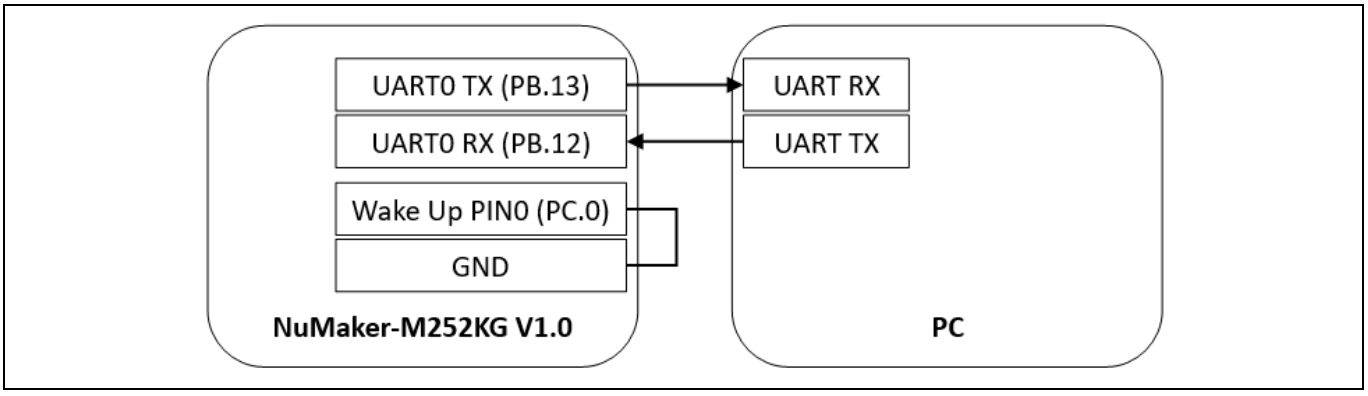

图 3-1 接线示意图

4. 目录信息

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

图 4-1 目录信息

5. 范例程序执行

1. 根据目录信息章节进入 ExampleCode 路径中的 KEIL 文件夹，双击 *M251_System_Event_Logger.uvprojx*。
2. 进入编译模式接口
   - 编译
   - 下载代码至内存
   - 进入 / 离开除错模式
3. 进入除错模式接口
   - 执行代码

6. 修订纪录

Date	Revision	Description
2023.07.15	1.00	初始发布。

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.