Access EEPROM in Keil RTX by Multitasking

Example Code Introduction for 32-bit NuMicro® Family

Information

Application	This code demonstrates how to access EEPROM in Keil RTX program by multitasking.
BSP Version	NUC230_240 Series BSP CMSIS V3.01.001
Hardware	NuEdu-EVB-NUC240 V2.1 NuEdu-Basic01 V2.1

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 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

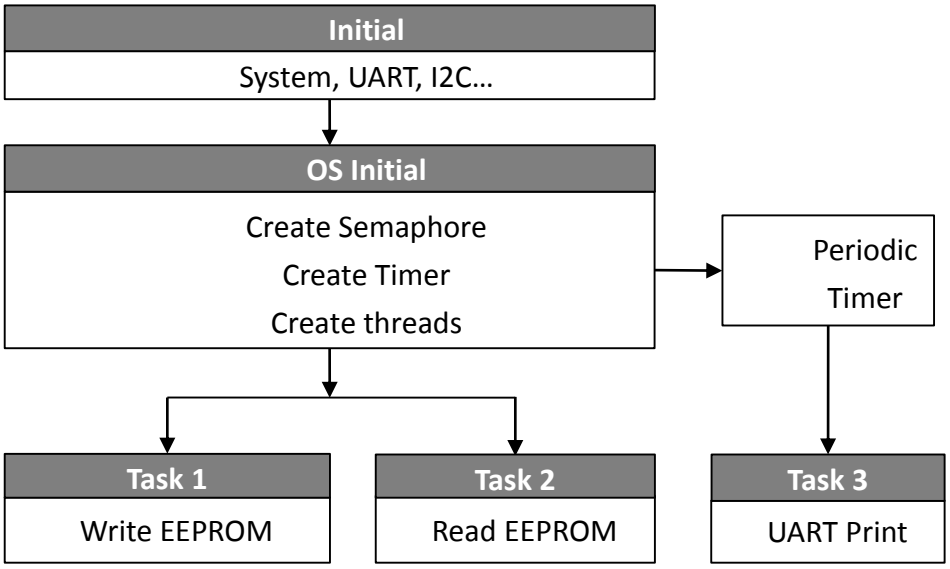
1 Function Description

1.1 Introduction

When a microcontroller application has to handle many tasks at the same time, we can use RTOS (real-time operating system) to help us manage these tasks. In RTOS system, CPU processing time is divided into a number of time slots. Each time slot processes different task and the switching between these tasks can be hundred times per second. It appears to that the application tasks are running simultaneously. When the application becomes more and more complex, we can use RTOS to help us managing these tasks.
CMSIS-RTOS in Keil MDK (Microcontroller Development Kit) provides a standard API for RTOS and application layer, includes the Keil free RTOS, RTX (Real-Time eXecutive) kernel. Once others RTOS can support CMSIS-RTOS, we can port our application to others RTOS platform easily. This sample code uses RTX kernel, which is provided by Keil CMSIS-MDK, to create a RTX kernel system, and then access EEPROM.

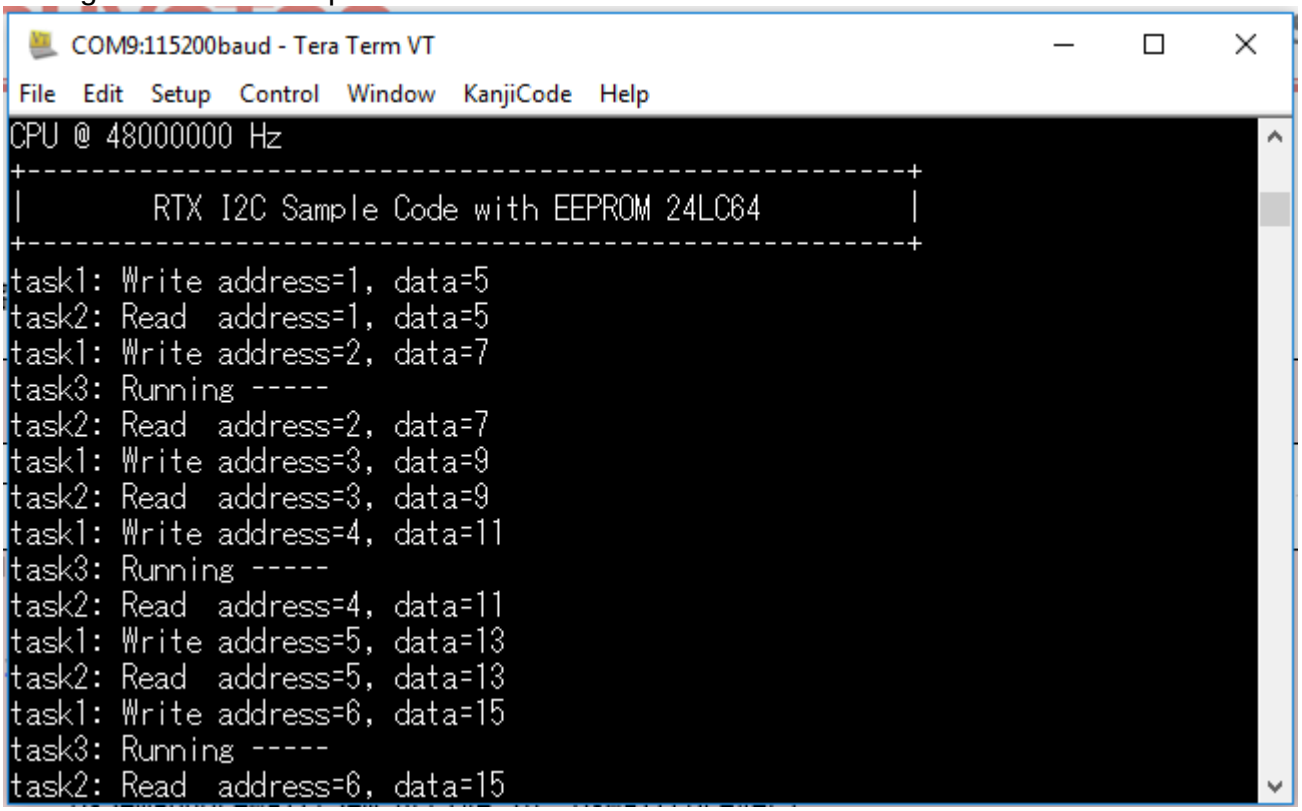
1.2 Principle

Initialize UART for print message and I2C for accessing EEPROM in the beginning of program. Create a semaphore, one timer and two threads. The two threads are reading and writing to EEPROM. One thread writes data and then releases one semaphore token. Another thread waits the semaphore token and then reads EEPROM data. The periodic timer is configured to trigger UART to print message.

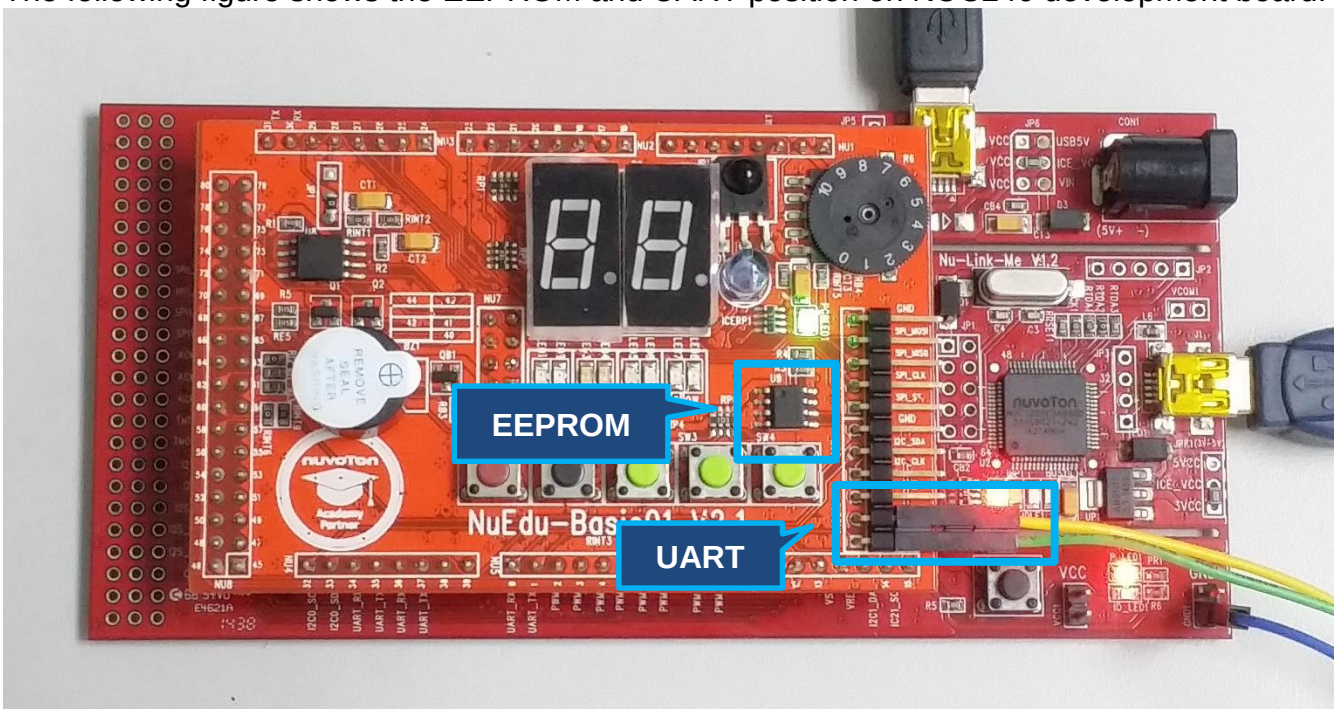


1.3 Demo Result

Connect NUC240 development board to computer through NuBridge. The UART printing message shows on computer terminal.



The following figure shows the EEPROM and UART position on NUC240 development board.



2 Code Description

2.1 RTX

Thread

This code has two parallel tasks. Declare function prototype and ID for each task (t_task1 and t_task2). Use osThreadDef to declare every tasks as threads. It can be used to configure task priority, too.

```
/* Thread IDs */
osThreadId t_task1;     // Declare a thread ID for task 1 (EEPROM Write)
osThreadId t_task2;     // Declare a thread ID for task 2 (EEPROM Read)

/* Function Declaration */
void task1_EEPROMwrite(void const *argument);
void task2_EEPROMread(void const *argument);

/* Thread Definition with function, priority, and stack requirements */
osThreadDef(task1_EEPROMwrite, osPriorityNormal, 1, 0);
osThreadDef(task2_EEPROMread, osPriorityNormal, 1, 0);
```

In main function, use osThreadCreate function to create new thread for each task and assign return value to their IDs.

```
    /* Create task and assign thread IDs */
    t_task1 = osThreadCreate(osThread(task1_EEPROMwrite), NULL);
    t_task2 = osThreadCreate(osThread(task2_EEPROMread), NULL);
```

Semaphore

Semaphore is used for internal communication between tasks. Declare a semaphore named Sem_Arrive and declare its ID variable named Sem_Arrive_id.

```
/* Semaphore ID */
osSemaphoreId Sem_Arrive_id;

/* Semaphore Declaration */
osSemaphoreDef(Sem_Arrive);
```

Call osSemaphoreCreate function to create new Semaphore in main function and assign the return value to its ID. The available token number is 1.

```
    /* Create Semaphore and configure token number */
    Sem_Arrive_id = osSemaphoreCreate(osSemaphore(Sem_Arrive), 1);
```

Timer

Declare a timer ID named Timer_Delay_id. Define this timer name as Timer_Delay. The task3_UARTprint function is triggered when the timer is time-out.

```
/* Timer ID */
osTimerId Timer_Delay_id;

/* osTimer Declaration */
osTimerDef(Timer_Delay, task3_UARTprint);
```

In main function, Use osTimerCreate function to create a new timer. Configure timer operation in periodic mode. There is no input parameter and assign the return value to the timer ID. Call osTimerStart function to start the timer counting. The time is 50ms. Timer will trigger the execution of task3_UARTprint function.

```
    /* Create Timer in periodic mode and without argument */
    Timer_Delay_id = osTimerCreate(osTimer(Timer_Delay), osTimerPeriodic, (void *)NULL);

    /* Start Timer with period 50ms */
    osTimerStart(Timer_Delay_id, 50);
```

In task3_UARTprint function, UART prints the message about execution.

```
void task3_UARTprint(void const *argument)
{
    /* UART print message */
    printf("task3: Running -----\n");
}
```

OS Kernel Initialization and Execution

Call osKernelInitialize fuction to start OS Kernel Initialization.

```
    /* Initialize RTOS */
    osKernelInitialize();
```

Call osKernelStart function to start OS Kernel execution and begin thread switching.

```
    /* Start the RTOS */
    osKernelStart();
```

2.2 EEPROM

Enable I2C, configure multiple function pin, clock, baud rate and interrupt.

```
    /* Initial I2C */
    I2C_EEPROM_Init();
```

The first task waits for available semaphore toke, calls I2C_EEPROM_Write function, writes data and releases one semaphore token.

```
void task1_EEPROMwrite(void const *argument)
{
    while(1) {

        /* Wait for semaphore token */
        osSemaphoreWait(Sem_Arrive_id, osWaitForever);

        /* Delay 10 msec */
        osDelay(10);

        /* Access EEPROM address from 0~10 */
        u32Addr++;
        if (u32Addr>10)
            u32Addr=0;

        /* Write data */
         printf("task1: Write address=%d, data=%d \n\r",u32Addr,(u32Addr*2+3));
         I2C_EEPROM_Write(u32Addr,(u32Addr*2+3));

        /* Release semaphore token */
        osSemaphoreRelease(Sem_Arrive_id);
    }
}
```

The second task waits for available semaphore token, calls I2C_EEPROM_Read to read data, checks data value and releases one semaphore token.

```
void task2_EEPROMread(void const *argument)
{
    uint32_t u32Data;

    while(1)
    {
        /* Wait for semaphore token */
        osSemaphoreWait(Sem_Arrive_id, osWaitForever);
```

```
        /* Read data */
        u32Data = I2C_EEPROM_Read(u32Addr);
        printf("task2: Read  address=%d, data=%d \n\r",u32Addr,u32Data);

        /* Check data is correct or not */
         if(u32Data!=(u32Addr*2+3)){
             printf("I2C Byte Write/Read Failed, Data 0x%x\n", u32Data);
             while(1);
         }

        /* Release semaphore token */
        osSemaphoreRelease(Sem_Arrive_id);

    }
}
```

3 Software and Hardware Environment

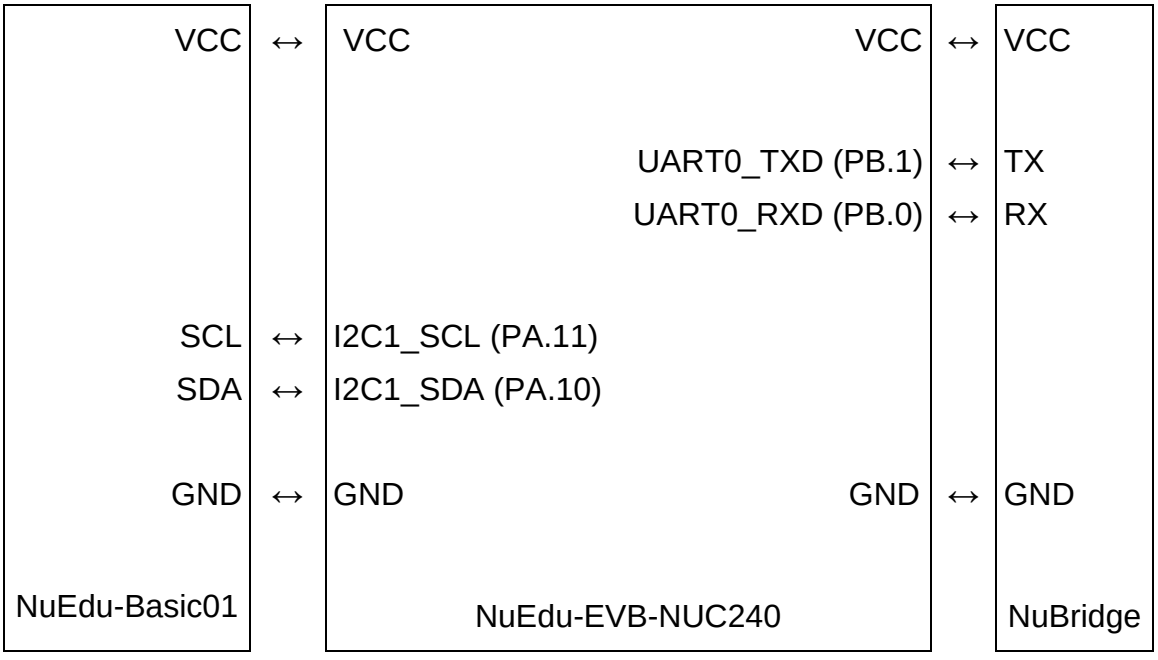
Software Environment

- BSP version
  - NUC230_240 Series BSP CMSIS V3.01.001
- IDE version
  - Keil uVersion 5.28

Hardware Environment

- Circuit components
  - NuEdu-EVB-NUC240 V2.1
  - NuEdu-Basic01 V2.1
  - NuBridge
- Diagram

Connect NuEdu-EVB-NUC240 development board with NuEdu-Basic01 board and NuBridge.



4 Directory Information

- EC_NUC240_RTX_EEPROM_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
    - NuEdu — Library for NuEdu-SDK-NUC240 board
    - StdDriver — All peripheral driver header and source files
  - SampleCode
    - ExampleCode — Source file of example code

5 How to Execute Example Code

1. Browsing into sample code folder by Directory Information (section 4) and double click NUC240_RTX_EEPROM.uvprojx.
2. Enter Keil compile mode
   a. Build
   b. Download
   c. Start/Stop debug session
3. Enter debug mode
   a. Run

6 Revision History

Date	Revision	Description
Sep. 30, 2019	1.00	1. Initially issued.

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.