RTX Real-time Operating System

Example Code Introduction for 32-bit NuMicro® Family

Information

Application	Demonstrate how multiple task program works in RTX(Real-Time eXecutive) real-time operating system
BSP Version	NUC230/240 Series BSP v3.01.002
Hardware	NuTiny–EVB–NUC240–LQFP100 V1.0

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

Keil RTX (Real-Time eXecutive) is a real-time operating system developed for ARM and Cortex M-series processors. It provides an environment in which user can build program software for simultaneous execution of multiple tasks, while also helping to make the program code more structured and easy to maintain. This sample code demonstrates how user creates tasks and performs various functions of real-time operating system in RTX environment.

1.2 Principle

There are several sample programs included in this project, mainly in the following categories:

- The creation, starting, execution, and termination of multiple tasks:
  Demonstrate how multiple tasks execute in RTX environment.
- The execution of the event:
  Demonstrate how multiple tasks synchronize and switch by event wait in RTX environment.
- The creation and execution of Semaphore:
  Demonstrate how multiple Tasks synchronize and switch by Semaphore mechanism in RTX environment.
- The creation and execution of Mutex:
  Demonstrate how multiple Tasks synchronize and switch by Mutex get/release in RTX environment.
- The creation and execution of Mailbox:
  Demonstrate how two tasks send and receive messages via Mailbox in RTX environment.
- The creation of Timer:
  Demonstrate the creation of Timer and confirm if Timer creation is successful.

1.3 Demo Result

Execute the RTX_Blinky project and display the execution results in the UART window. As shown in Figure 1-1:

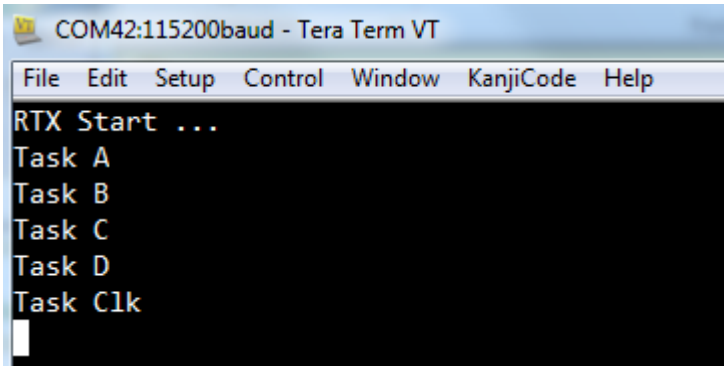


Figure 1-1 RTX_Blinky sample program execution result

In the implementation of the RTX_Blinky project, the corresponding LED on the NUC240 flashes. As shown in the yellow box in Figure 1-2:

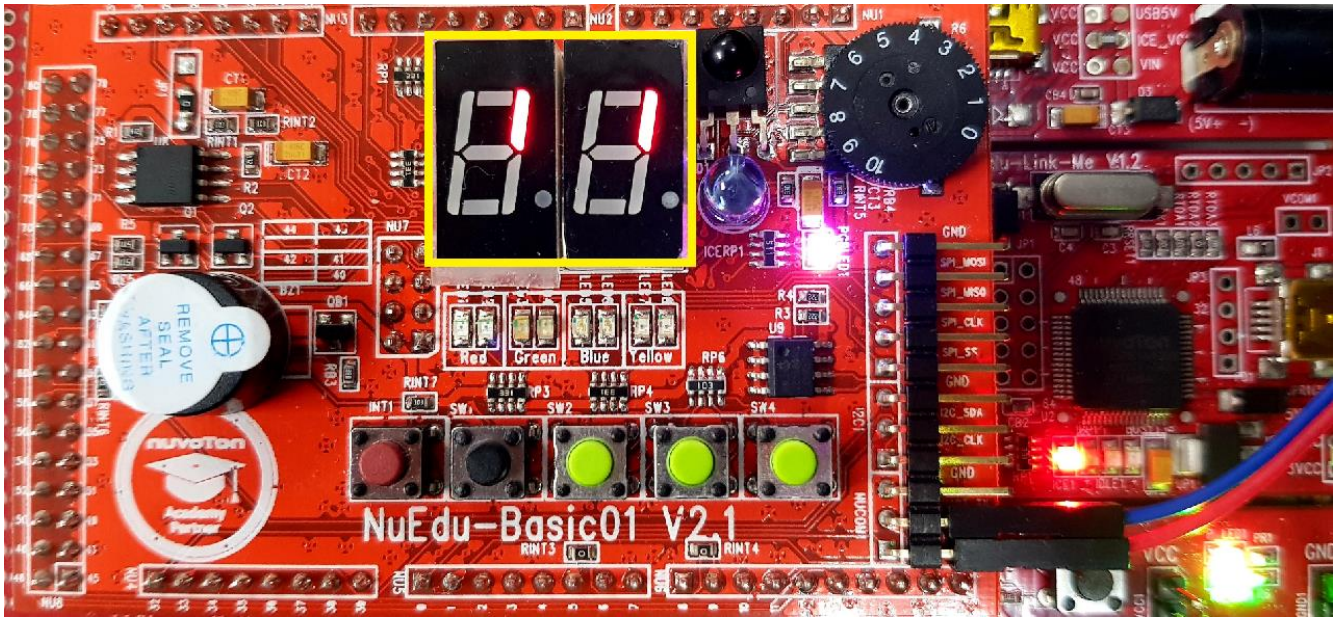


Figure 1-2 When the RTX_Blinky sample program is executed, the NUC240 LED flashes

Execute the RTX_Delay project and display the execution result in the UART window. As shown in Figure 1-3:

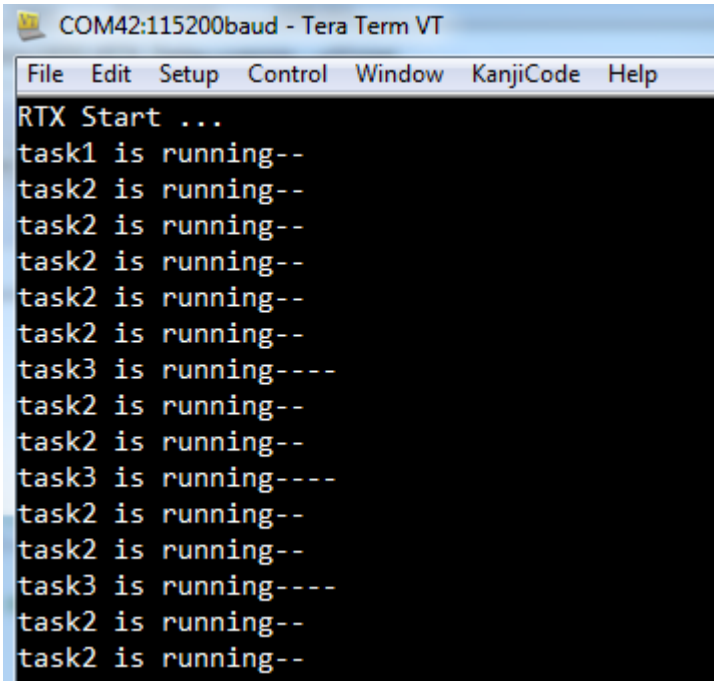


Figure 1-3 RTX_Delay sample program execution result

Execute the RTX_Event_Interrupt project and display the execution result in the UART window. As shown in Figure 1-4:

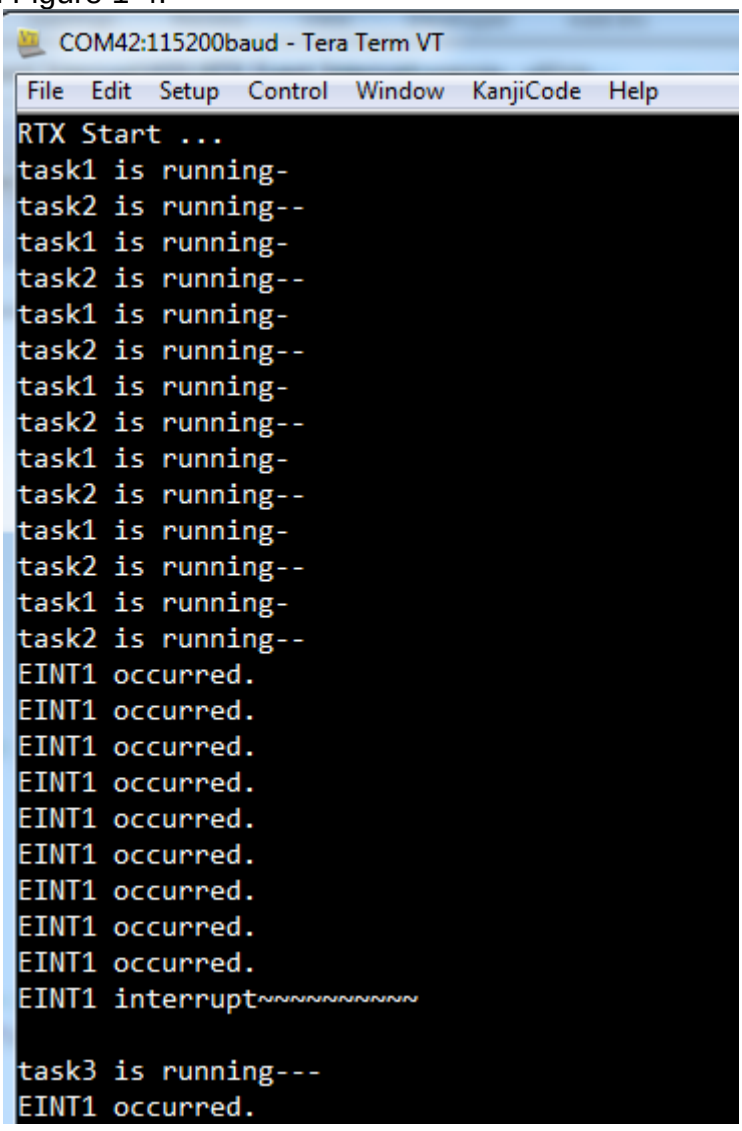


Figure 1-4 RTX_Event_Interrupt sample program execution result

Execute the RTX_Event project and display the execution result in the UART window. As shown in Figure 1-5:

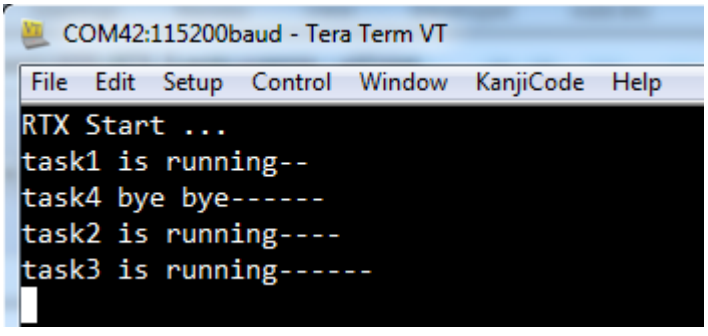


Figure 1-5 RTX_Event sample program execution result

Execute the RTX_Idle project and display the execution result in the UART window. As shown in Figure 1-6:

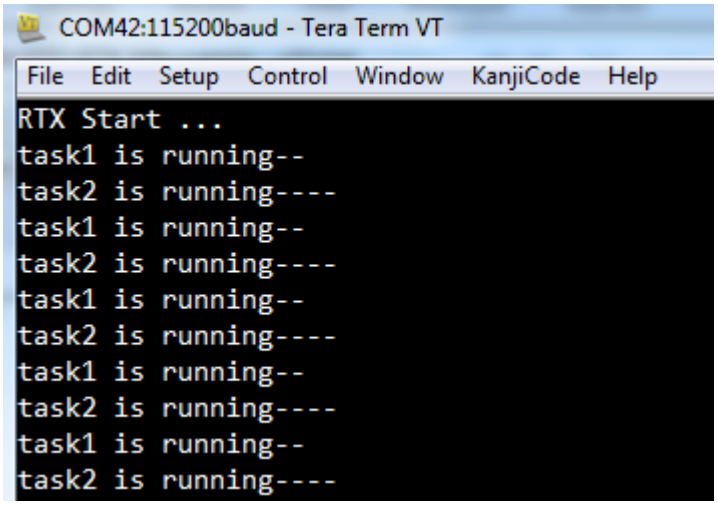


Figure 1-6 RTX_Idle sample program execution result

Execute the RTX_Mailbox project and display the execution results in the UART window. As shown in Figure 1-7:

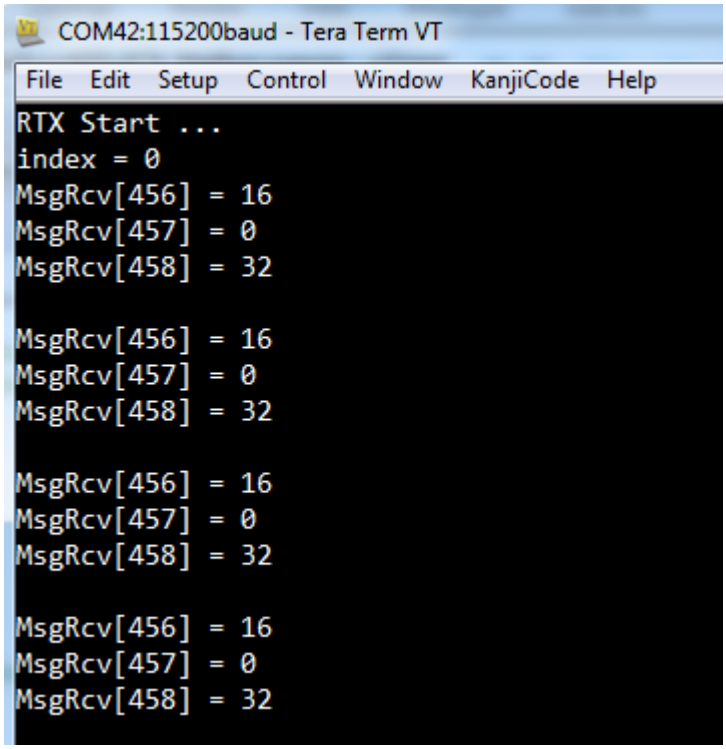


Figure 1-7 RTX_Mailbox sample program execution result

Execute the RTX_Multiple_Instance project and display the execution result in the UART window. As shown in Figure 1-8:

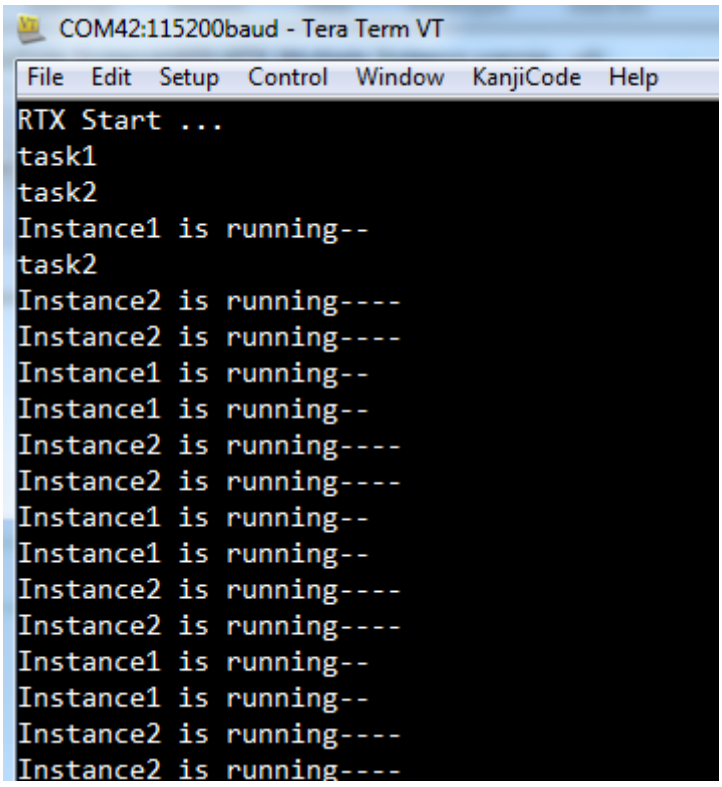


Figure 1-8 RTX_Multiple_Instance sample program execution result

Execute the RTX_Mutex project and display the execution results in the UART window. As shown in Figure 1-9:

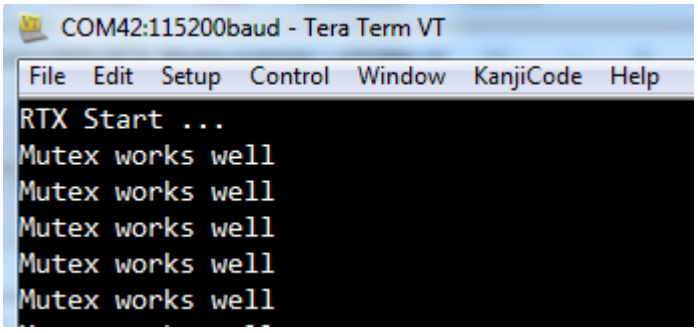


Figure 1-9 RTX_Mutex sample program execution result

Execute the RTX_Priority_Inv project and display the execution result in the UART window. As shown in Figure 1-10:

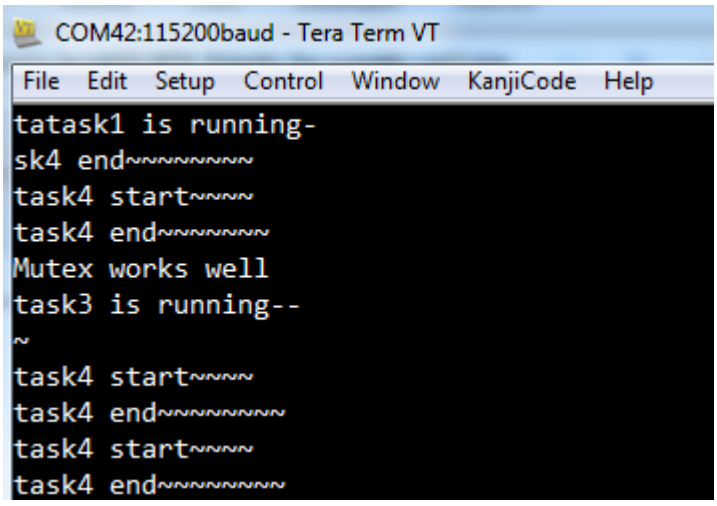


Figure 1-10 RTX_Priority_Inv sample program execution result

Execute the RTX_Semaphore_Barrier project and display the execution result in the UART window. As shown in Figure 1-11:

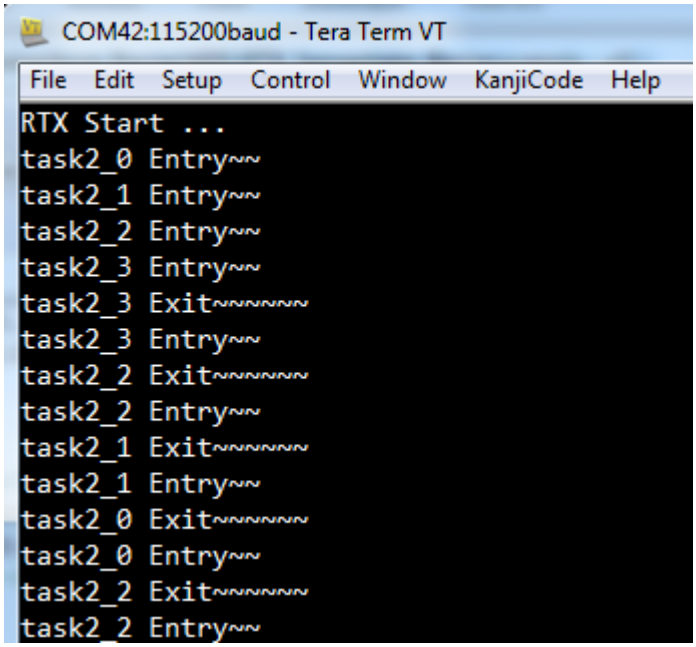


Figure 1-11 RTX_Semaphore_Barrier sample program execution result

Execute the RTX_Semaphore_Multiplex project and display the execution result in the UART window. As shown in Figure 1-12:

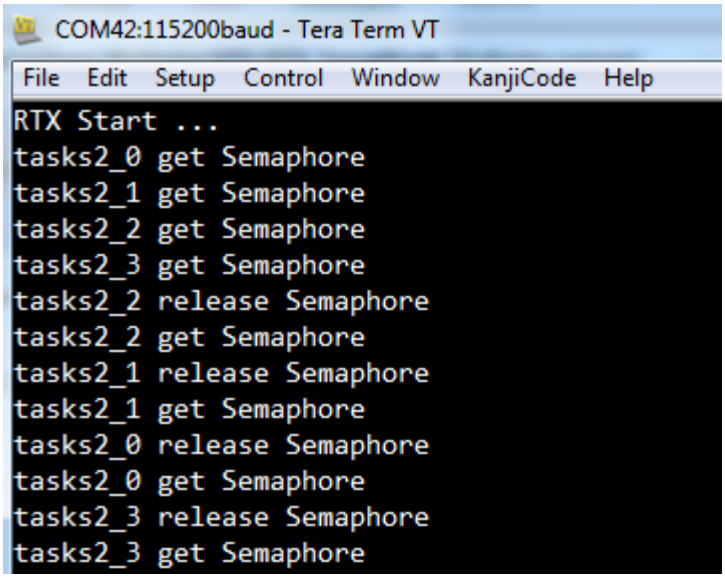


Figure 1-12 RTX_Semaphore_Multiplex sample program execution result

Execute the RTX_Semaphore_Rendezvous project and display the execution result in the UART window. As shown in Figure 1-13:

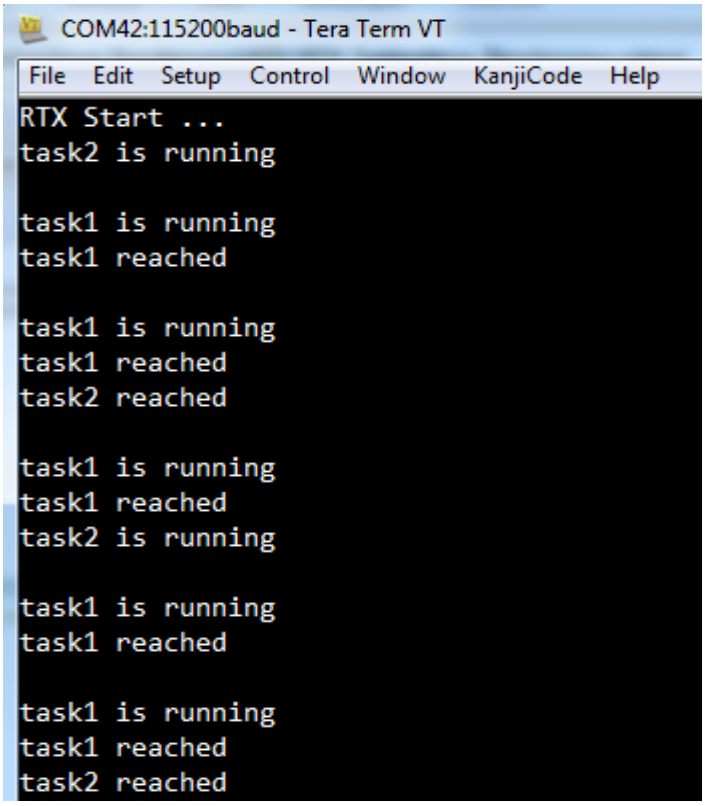


Figure 1-13 RTX_Semaphore_Rendezvous sample program execution result

Execute the RTX_Semaphore_Synchronization project and display the execution result in the UART window. As shown in Figure 1-14:

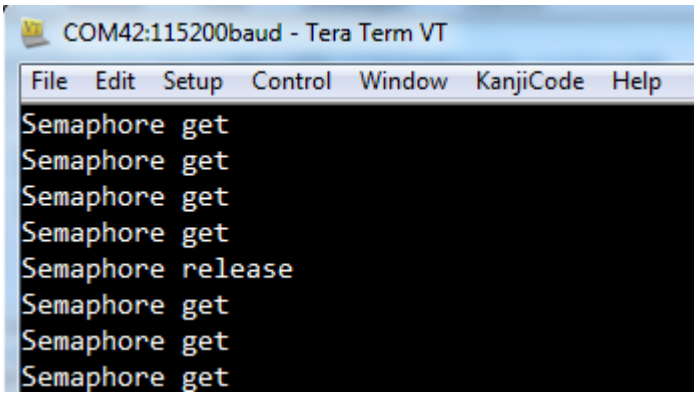


Figure 1-14 RTX_Semaphore_Synchronization sample program execution result

Execute the RTX_Timer project and display the execution result in the UART window. As shown in Figure 1-15:

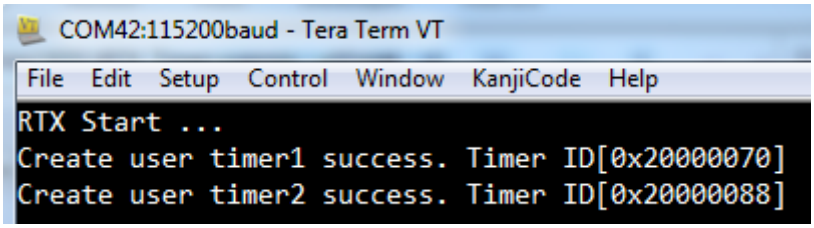


Figure 1-15 RTX_Timer sample program execution result

2 Code Description

The following is the process of creating and starting the task:

```
osThreadId t_taskA;                         /* assigned task id of task: TaskA */
osThreadId t_taskB;                         /* assigned task id of task: TaskB */
osThreadId t_taskC;                         /* assigned task id of task: TaskC */
osThreadId t_taskD;                         /* assigned task id of task: TaskD */
osThreadId t_taskClk;                       /* assigned task id of task: TaskClk */

void TaskA(void const *argument);
void TaskB(void const *argument);
void TaskC(void const *argument);
void TaskD(void const *argument);
void TaskClk(void const *argument);

/* define task as thread function */
osThreadDef(TaskA, osPriorityNormal, 1, 0);
osThreadDef(TaskB, osPriorityNormal, 1, 0);
osThreadDef(TaskC, osPriorityNormal, 1, 0);
osThreadDef(TaskD, osPriorityNormal, 1, 0);
osThreadDef(TaskClk, osPriorityNormal, 1, 0);

int main(void)
{

    /* Create task and assign thread ID to t_task */
    t_taskA = osThreadCreate(osThread(TaskA), NULL);
    t_taskB = osThreadCreate(osThread(TaskB), NULL);
    t_taskC = osThreadCreate(osThread(TaskC), NULL);
    t_taskD = osThreadCreate(osThread(TaskD), NULL);
    t_taskClk  = osThreadCreate (osThread(TaskClk), NULL);
    osSignalSet (t_taskA, 0x0001);

    /* Start the OS kernel and begin thread switching */
    osKernelStart();
}

/*----------------------------------------------------------------------------
 *        Task 1 'TaskA': Phase A output
 *---------------------------------------------------------------------------*/
```

```
void TaskA (void const *argument) {
    printf("Task A\n");
    for (;;) {
        osSignalWait (0x0001, 0xffff);    /* wait for an event flag 0x0001    */
        LED_On (LED_A);
        signal_func (t_taskB);            /* call common signal function      */
        LED_Off(LED_A);
    }
}

/*----------------------------------------------------------------------------
 *        Task 2 'TaskB': Phase B output
 *---------------------------------------------------------------------------*/
void TaskB (void const *argument) {
    printf("Task B\n");
    for (;;) {
        osSignalWait (0x0001, 0xffff);    /* wait for an event flag 0x0001    */
        LED_On (LED_B);
        signal_func (t_taskC);            /* call common signal function      */
        LED_Off(LED_B);
    }
}

/*----------------------------------------------------------------------------
 *        Task 3 'TaskC': Phase C output
 *---------------------------------------------------------------------------*/
void TaskC (void const *argument) {
    printf("Task C\n");
    for (;;) {
        osSignalWait (0x0001, 0xffff);    /* wait for an event flag 0x0001    */
        LED_On (LED_C);
        signal_func (t_taskD);            /* call common signal function      */
        LED_Off(LED_C);
    }
}

/*----------------------------------------------------------------------------
 *        Task 4 'TaskD': Phase D output
 *---------------------------------------------------------------------------*/
void TaskD (void const *argument) {
    printf("Task D\n");
```

```
    for (;;) {
        osSignalWait (0x0001, 0xffff);    /* wait for an event flag 0x0001    */
        LED_On (LED_D);
        signal_func (t_taskA);            /* call common signal function      */
        LED_Off(LED_D);
    }
}

/*----------------------------------------------------------------------------
 *        Task 5 'TaskClk': Signal TaskClk
 *---------------------------------------------------------------------------*/
void TaskClk (void const *argument) {
    printf("Task Clk \n");
    for (;;) {
        osSignalWait (0x0100, 0xffff);    /* wait for an event flag 0x0100    */
        LED_On (LED_CLK);
        osDelay (8);                      /* delay 8 TaskClk ticks            */
        LED_Off(LED_CLK);
    }
}
```

The following is the process of event controlling:

```
void task1 (void const *argument)
{
    while(1)
    {
        osSignalWait(0x0005,0xffff);   /* Wait for an event to be set, no time out */

        printf("task1 is running--\n");
        osDelay(50);

        osSignalSet(t_task2, 0x0001);   /* Set an event for task 2. Each task has 16 event
flags */
        osSignalSet(t_task3, 0x0001);   /* Set an event in task 3 */
    }
}

void task2 (void const *argument)
{
    while(1)
```

```
    {
        osSignalWait(0x0001,0xffff);            /* Wait for an event */

        printf("task2 is running----\n");
        osDelay(50);

        osSignalSet(t_task3, 0x0002);           /* Set an event in Task 3 */
    }
}

void task3 (void const *argument)
{
    while(1)
    {
        osSignalWait(0x0003,0xffff);            /* Wait for two event flags to be set */

        printf("task3 is running------\n");
        osDelay(10);

        osSignalSet(t_task1, 0x0001);           /* Trigger task 1 to start over */
    }
}
```

The following is the process of creating and controlling semaphore:

```
osSemaphoreDef_t *SemMultiplex;
osSemaphoreId SemMultiplexId;

uint32_t g_au32Param[6] = {0, 1, 2, 3, 4, 5};

void task1 (void const *argument)
{
    /* Create semaphore with semaphore num = 6 */
    SemMultiplexId = osSemaphoreCreate(SemMultiplex, 6);

    /* Create 6 sub-tasks of task 2 */
    t_task2_0 = osThreadCreate(osThread(task2), &g_au32Param[0]);
    t_task2_1 = osThreadCreate(osThread(task2), &g_au32Param[1]);
    t_task2_2 = osThreadCreate(osThread(task2), &g_au32Param[2]);
    t_task2_3 = osThreadCreate(osThread(task2), &g_au32Param[3]);
    t_task2_4 = osThreadCreate(osThread(task2), &g_au32Param[4]);
    t_task2_5 = osThreadCreate(osThread(task2), &g_au32Param[5]);
```

```
    /* Delete task1 itself */
    osThreadTerminate(t_task1);

}

void task2 (uint32_t *argument)
{
    while(1)
    {
        /* Wait here until you aquire a semaphore    */
        osSemaphoreWait(SemMultiplexId, 0xffff);
        printf("tasks2_%d get Semaphore\n", *argument);
        osDelay(100);
        osSemaphoreRelease(SemMultiplexId);
        printf("tasks2_%d release Semaphore\n", *argument);
    }
}
```

The following is the process of creating and controlling mutex:

```
osMutexDef_t mutex;
osMutexId mutexId;
uint32_t g_u32CriticalRes1, g_u32CriticalRes2;

void task1 (void const *argument)
{
    /* Initilise the Mutex */
    mutexId = osMutexCreate(&mutex);

    g_u32CriticalRes1 = 0;
    g_u32CriticalRes2 = 0;

    while(1)
    {
        /* Wait here until the mutex is aquired */
        osMutexWait(mutexId, 0xffff);

        g_u32CriticalRes1++;
        /* Before Mutex release, the procees can't be interrupted */
```

```
        /* The value of g_u32CriticalRes1 and g_u32CriticalRes2 should be the same*/
        osDelay(50);
        g_u32CriticalRes2++;

        osMutexRelease(mutexId);
    }
}

void task2 (void const *argument)
{
    while(1)
    {
        /* Wait here until the mutex is aquired */
        osMutexWait(mutexId, 0xffff);

        if(g_u32CriticalRes1 != g_u32CriticalRes2)
        {
            printf("Mutex error\n");
            while(1);
        }
        else
        {
            printf("Mutex works well\n");
        }

        g_u32CriticalRes1++;
        g_u32CriticalRes2++;

        osMutexRelease(mutexId);
    }
}
```

The following is the process of mailbox setup and execution:

```
osPoolId poo1Id;
osMailQId mailId;

msg_t message[2];
uint8_t g_u8Idx;
```

```
osMailQDef_t mailbox[16];
osPoolDef_t  mpool[16];

void task1 (void const *argument)
{
    msg_t *mptr;                                /* Pointer to the mailbox send buffer */
    uint32_t i;


    mailId = osMailCreate (mailbox, t_task1);
    poo1Id = osPoolCreate(mpool);

    for(g_u8Idx = 0; g_u8Idx < 2; g_u8Idx++)
    {
        message[g_u8Idx].msg_id = g_u8Idx;
        for(i = 0; i < 3; i++)
        {
            message[g_u8Idx].msg[i] = g_u8Idx*3 + i;
        }
    }

    g_u8Idx = 0;
    while(1)
    {
        g_u8Idx %= 2;

        osPoolAlloc(poo1Id);                    /* Set the pointer to a mailbox buffer */
        mptr = (msg_t *)mpool;
        *mptr = message[g_u8Idx];              /* Write data to the mailbox buffer    */
        osMailPut(mailId, mptr);                /* Send a message through mailbox      */

        printf("index = %d\n", g_u8Idx++);
        osDelay(50);
    }
}

void task2 (void const *argument)
{
    msg_t MsgRcv,*rptr;                         /* Pointer to the mailbox receive buffer */
    uint32_t i;
```

```
    while(1)
    {
        osMailGet(mailId, 0xffff);           /* Wait for a new message in mailbox */
        MsgRcv = *rptr;
        osMailFree(mailId, rptr);            /* Release the mailbox receive buffer */

        for(i = 0; i < 3; i++)
        {
            printf("MsgRcv[%d] = %d\n", MsgRcv.msg_id*3 + i, MsgRcv.msg[i]);
        }
        printf("\n");
    }
}
```

The following is the process of creating timer:

```
osTimerId Timer1_Id;
osTimerId Timer2_Id;

void Timer1_Callback (void const *arg)
{

}

void Timer2_Callback (void const *arg)
{

}
osTimerDef (Timer1, Timer1_Callback);
osTimerDef (Timer2, Timer1_Callback);

void task1 (void const *argument)
{
    /* Create Timer 1*/
    Timer1_Id = osTimerCreate (osTimer(Timer1), osTimerPeriodic, NULL);
    /* Check if Timer 1 create success*/
    if (Timer1_Id == NULL)
    {
        printf ("Failed to create user timer1.\n");
    }
    else
    {
        printf ("Create user timer1 success. Timer ID[0x%x]\n", (uint32_t)Timer1_Id);
```

```
    }
    /* Create Timer 1*/
    Timer2_Id = osTimerCreate (osTimer(Timer2), osTimerPeriodic, NULL);
    /* Check if Timer 1 create success*/
    if (Timer2_Id  == NULL)
    {
        printf ("Failed to create user timer2.\n");
    }
    else
    {
        printf ("Create user timer2 success. Timer ID[0x%x]\n", (uint32_t)Timer2_Id);
    }

    while(1)
    {
        osDelay(20);
    }

}
```

3 Software and Hardware Environment

Software Environment

- BSP version
  - NUC230/240 Series BSP CMSIS v3.01.002
- IDE version
  - Keil uVersion 5.28

Hardware Environment

- Circuit components
  - NuTiny–EVB–NUC240–LQFP100 V1.0
- Diagram
  - Connect the UART TX (PB.1) pin to the PC UART RX to display the execution result of the sample code on the PC.

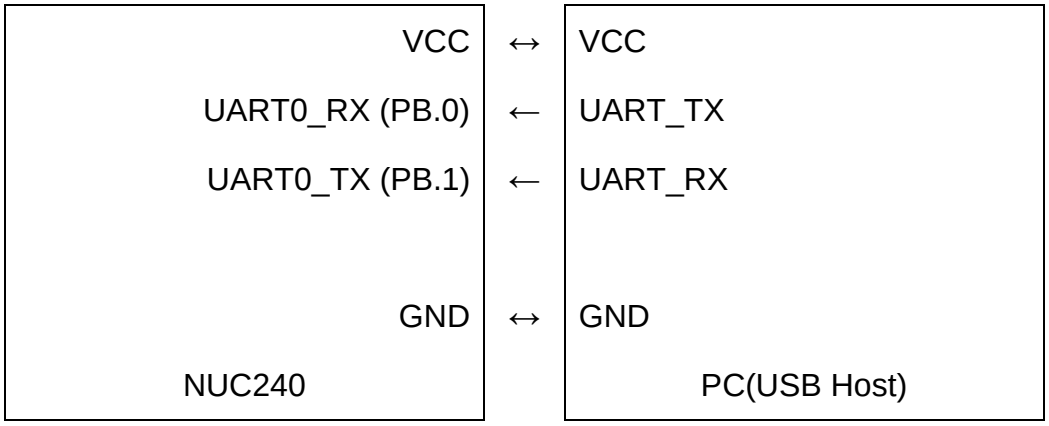


Figure 3-1 Connection of NUC240 and PC

4 Directory Information

- EC_NUC240_RTX_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

5 How to Execute Example Code

1. Browsing into sample code folder by Directory Information (section 4) and double click RTX_xxx.uvproj.
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. 18, 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.