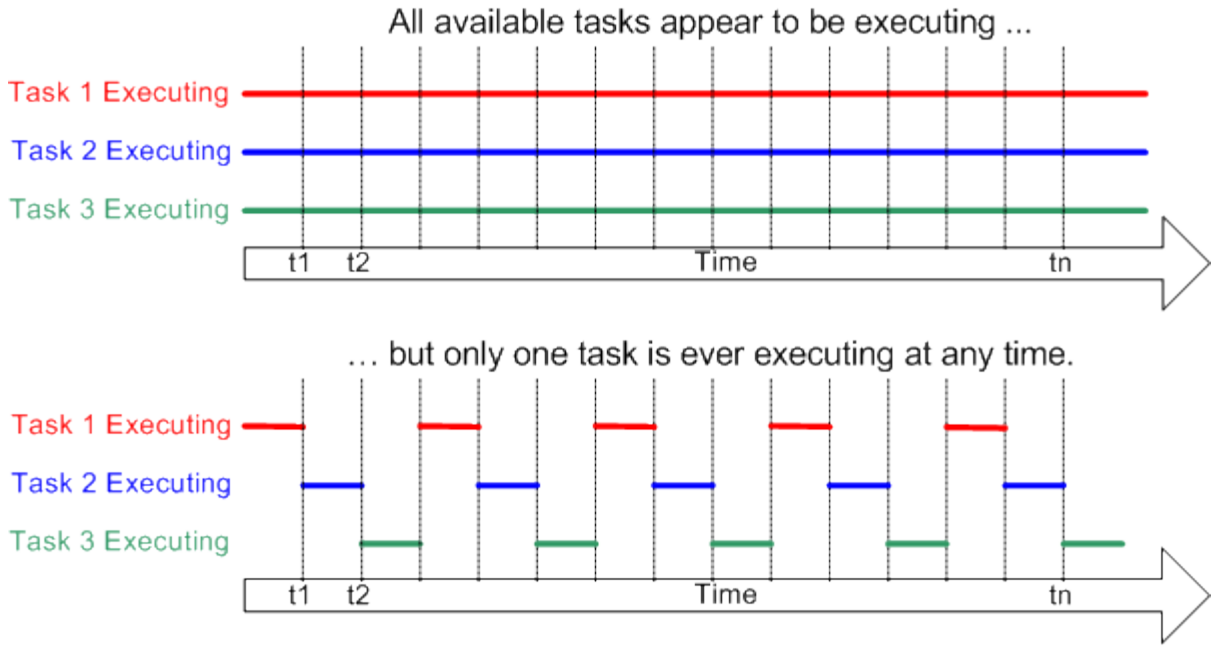
سیستم عامل بلادرنگ RTX

تهیه کننده : رامین جعفرپورفرد

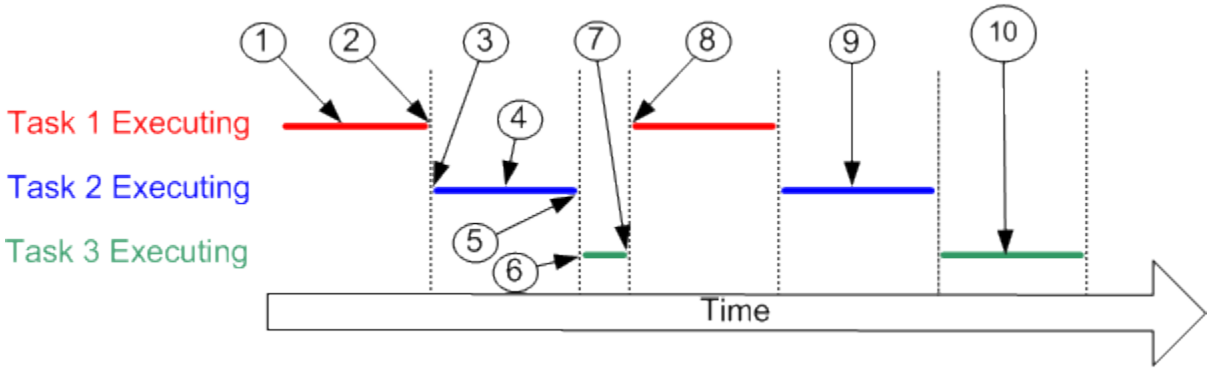
فرض کنید در یک پروژه تعداد زیادی سنسور و محرک وجود دارد و باید به طور مداوم مقادیر سنسورها خوانده شده و تغییرات لازم در محرک ها اعمال شود. بعضی سنسورها و محرک ها اولویت بالایی دارند و باید در دوره های سریعتری بررسی شوند یا عمکرد بعضی خروجی ها به هم وابسته هستند. نوشتن کل برنامه در یک حلقه کنترلی برای چنین پروژه ای کار خیلی سختی هست و تضمین لازم برای اجرای به موقع بخش های مختلف وجود ندارد.این جا است که اهمیت سیستم عامل های بلادرنگ مشخص میشود.RTOS این امکان را فراهم میکند که حلقه کنترلی به قسمتهای کوچکتر (Task) تقسیم شود و همه این تسک ها با اولویت بندی که انجام میشود به صورت همزمان و بلادرنگ اجرا شوند.واژه بلادرنگ در یک سیستم عامل به این منظور میباشد که آن سیستم عامل تضمین لازم برای اجرای به موقع تسک های با اولویت بالاتر را ارائه میدهد.مثلا ممکن است نیاز باشد یک تسک دقیقا در دوره های زمانی 50ms اجرا شود.

استفاده از سیستم عامل مزایای دیگری همچون ساده شدن برنامه نویسی ، امکان تقسیم ساده وظایف در پروژه و انجام پروژه بصورت گروهی، ساده شدن عملیات خطایابی با تست تسک ها و امکان گسترش سیستم در در پروژه های بعدی را دارد.

MultiTasking یا عملکرد چند وظیفگی این امکان را برای سیستم عامل فراهم میکند که چند برنامه متفاوت به صورت موازی با هم اجرا شوند.این کار با جابجایی سریع تسک در حال اجرا توسط سیستم عامل انجام میشود.



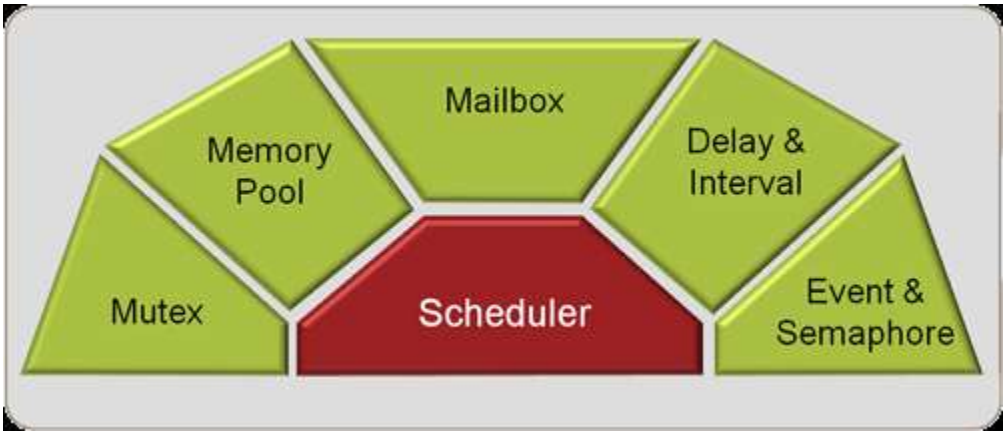
Scheluder یا زمانبند قسمتی از کرنل سیستم عامل میباشد که تصمیم میگیرد در هر لحظه کدام تسک باید پردازش شود.در یک سیستم عامل معمولی (غیر بلادرنگ) الگوریتم سیستم عامل برای زمان بندی به این صورت است که زمان پردازش باید به صورت عادلانه بین تسک ها تقیسم شود.به طور مثال در الگوریتم Round-Robin زمان پردازش به تکه های کوچک مساوی (Tick) تقسیم میشود و به هر تسک این مقدار زمان پردازش داده میشود.اگر اجرای تسک تمام نشود تسک متوقف شده و تسک بعدی با همان زمان پردازش میشود.اگر هم تسکی زودتر تمام شود یا منتظر یک رویداد باشد تسک دیگر اجرا خواهد شد.



تفاوت سیستم عامل بلادرنگ با سیستم عامل های معمولی در این قسمت در این است که در سیستم عاملهای بلادرنگ تسک با اولویت بالاتر تا زمانیکه نیاز داشته باشد میتواند از زمان پردازشی استفاده کند ،مگر آنکه تسک دیگری با اولویت برابر وجود داشته باشد.به عبارتی در سیستم عامل بلادرنگ الگوریتم های سوئیچ تسک تنها برای تسک های با اولویت برابر اجرا میشود.

سیستم عامل RL-RTX :

RTX یک سیستم عامل بلادرنگ (Real Time OS) میباشد که به صورت رایگان در کیت توسعه MDK-ARM نرم افزار Keil ارائه میشود.این سیستم عامل برای میکروکنترلرهای با هسته ARM طراحی شده است و امکان ایجاد برنامه هایی با وظایف چندگانه به طور همزمان را فراهم میکند.



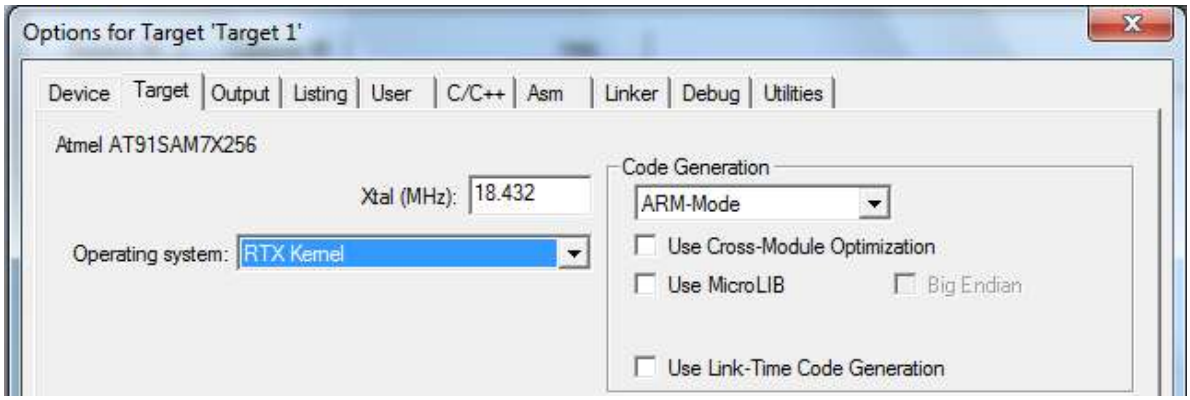
ویژگی ها :

- سیستم عامل بلادرنگ با حق امتیاز رایگان و سورس کد
- زمانبندی قابل انعطاف : round-robin ، pre-emptive و collaborative
- بلادرنگ پر سرعت با زمان تاخیر اینتراپت کم
- سورس کم حجم برای سیستم های با منابع محدود
- تعداد بی نهایت تسک با 254 سطح اولویتی
- تعداد بینهایت mailboxes ، semaphores و تایمر
- پشتیبانی از کارکردهای چند نخی امن
- امکان عیب یابی کرنل در شبیه ساز MDK_ARM
- پشتیبانی از ویزارد برای پیکربندی
- پشتیبانی از کتابخانه های FlashFS ، TCPnet ، CAN و USB

با توجه به اینکه میکروپروسسورهای سری Cortex-M مجهز به چند قابلیت سخت افزاری برای پشتیبانی بهتر از سیستم عامل هستند دو ورژن مختلف از RTX برای سری ARM7/ARM9 و Cortex-M وجود دارد.در بخش های بعدی از میکروکنترلر AT91SAM7X256 استفاده شده است.

ایجاد اولین پروژه با RTX در Keil :

1. ایجاد پروژه جدید از قسمت Project -> New uVision Project ...
2. تعیین سیستم عامل از قسمت Project -> Option for Target -> Target -> Operation System



3. کپی کردن فایل پیکربندی RTX در پوشه پروژه و اضافه کردن آن به پروژه

مسیر فایل پیکربندی برای سری SAM7X : ...Keil/ARM/Startup/Atmel/ RTX_Conf_SAM7X.c

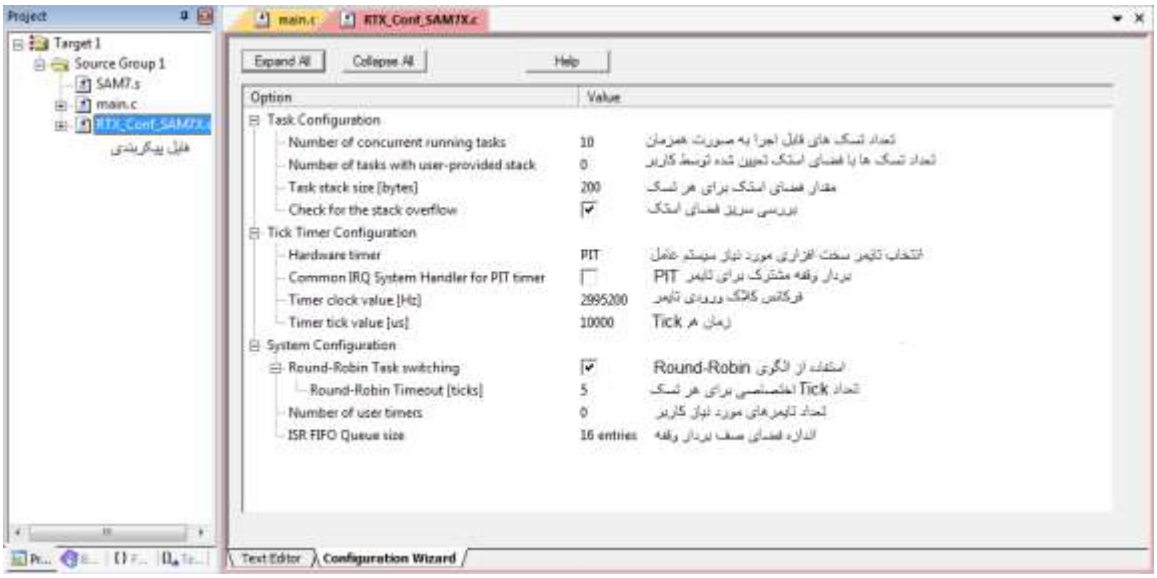
4. تغییر فایل اسمبلی استارت آپ SAM7.S

تغییر کد SWI_Handler B SWI_Handler با کد IMPORT SWI_Handler

5. الحاق فایل RTL.H به سورس اصلی برنامه #include <rtl.h>

اگر مراحل بالا درست انجام شود پروژه بدون مشکل کامپایل میشود.

فایل RTX_Conf_SAM7X.c که به پروژه اضافه شده است مربوط به پیکربندی RTX مباشد و میتوان با باز کردن آن توسط Wizard تنظیمات بخش های مختلف آنرا به راحتی تغییر داد.



اولین برنامه :

```
#include <AT91SAM7X256.h>
#include <rtl.h>

__task void task1(void);
__task void task2(void);

int main()
{

        os_sys_init(task1);
         while(1){}
        return 0;

}
```

```
__task void task1(void)
{
        os_tsk_create(task2, 1);
        while(1){  //Process of Task1      };
}
__task void task1(void)
{
        while(1){  //Process of Task2      };
}
```

در برنامه بالا دو تسک ایجاد شده اند. در هر کدام از تسک ها یک حلقه بینهایت وجود دارد که دستورات داخل این حلقه ها طبق الگوریتم Round-Robin همواره اجرا خواهند شد.الگوی ایجاد یک تسک بصورت زیر میباشد :

```
__task void task (void)
{
  //initialization
  while(1){
    //task process
  }
}
```

دستورات اصلی تسک ها باید همیشه داخل حلقه بینهایت قرار بگیرند. خارج شدن از داخل این حلقه باعث Crash در سیستم عامل خواهد شد.

تابع os_sys_init() برای مقدار دهی اولیه و راه اندازی سیستم عامل استفاده میشود و پس از آن اولین تسک که توسط این تابع گرفته شده است فعال میشود.معمولا تسک اول نیز برای راه اندازی اولیه مقادیر مورد نیاز کاربر و سخت افزارهای جانبی و ایجاد تسک های اصلی برنامه استفاده میشود.

ایجاد تسک توسط تابع os_tsk_create() انجام میشود.الگوی این تابع به صورت زیر میباشد :

```
OS_TID os_tsk_create (
  void (*task)(void),   /* Task to create */
  U8    priority );     /* Task priority (1-254) */
```

این تابع تسک و سطح اولویتی آنرا میگیرد و تسک مربوطه را فعال میکند.همچنین برای هر تسک یک ID اختصاصی ایجاد شده و توسط تابع برگشت داده میشود.OS_TID برگشت داده شده کاربردهای زیادی دارد که کم کم در مورد آنها بحث خواهیم کرد.

برای غیر فعال کردن یک تسک از داخل خود تسک از تابع os_tsk_delete_self() استفاده میشود. فراخوانی این تابع در داخل هر تسک باعث میشود که آن تسک به حالت غیر فعال رفته ، فضای استک آن حذف شود و تا زمانیکه تسک دوباره ایجاد نشود زمانی برای پردازش آن در نظر گرفته نمیشود.اگر نیاز باشد از داخل تسکی ، تسک دیگری حذف شود باید TID آن تسک در دسترس باشد .این کار توسط تابع os_tsk_delete(OS_TID TID) انجام میشود.

برنامه چشمک زن :

```
#include <AT91SAM7X256.h>
#include <rtl.h>
__task void task_init(void);
__task void task1(void);
__task void task2(void);
int main()
{
        os_sys_init(task_init);
        while(1){}
        return 0;
}
__task void task_init(void)
{
        *AT91C_PIOB_PER = (1 << 0) | (1 << 1);
         *AT91C_PIOB_OER = (1 << 0) | (1 << 1);
        *AT91C_PIOB_OWER = (1 << 0) | (1 << 1);
        os_tsk_create(task1, 1);
        os_tsk_create(task2, 1);
        os_tsk_delete_self();
}
__task void task1(void)
{
  int i = 0;
        while(1){
                for (i = 0; i < 0x7ffff; i++);//process
                *AT91C_PIOB_ODSR ^= (1 << 0);   //Toggle of LED1
```

```
        }
}

__task void task2(void)
{
  int i = 0;
        while(1){
                for (i = 0; i < 0x8ffff; i++);//process
                *AT91C_PIOB_ODSR ^= (1 << 1);   //Toggle of LED2
        }
}
```

عملکرد برنامه بدین صورت است که ابتدا در تابع Main سیستم عامل توسط تابع os_sys_init راه اندازی شده و تسک task_init به عنوان اولین تسک فعال میشود.این تسک دو پین استفاده شده در برنامه را خروجی کرده و دو تسک دیگر را فعال میکند.سپس توسط تابع os_tsk_delete_self پایان عملکرد خود را به سیستم عامل اطلاع میدهد.دو تسک ایجاد شده هر کدام وظیفه دارند هر بار پس از انجام پروس اصلی خود وضعیت خروجی یکی از پین های میکروکنترلر را معکوس کنند.به جای پروسس اصلی در تسک ها از حلقه for استفاده شده است.

هر تسک در هر لحظه در یکی از وضعیتهای در حال اجرا،آماده برای پردازش،غیرفعال و انتظار قرار میگیرد. زمانیکه تسک 1 در حال اجرا میباشد تسک 2 در حالت آماده برای پردازش قرار میگیرد و بلعکس.

در برنامه نویسی بر پایه سیستم عامل توابع تاخیر معمولی به ندرت استفاده میشوند.چون استفاده از توابع تاخیر موجب هدر رفتن زمان پردازشی سیستم و کاهش کارایی سیستم عامل میشود.در عوض خود سیستم عامل توابعی را محیی میکند که با استفاده از آنها میتوان بدون هدر دادن زمان پردازشی تاخیرهای دقیق ایجاد کرد.تابع os_dly_wait(int tick_num) برای ایجاد تاخیر تعریف شده است و با هر فراخوانی در داخل هر تسک به تعداد Tick دریافتی آن تسک را در حالت انتظار قرار میدهد.

در برنامه بالا چنانچه بخواهیم LED1 با تاخیر 1 ثانیه ای روشن، خاموش شود تسک مربوطه به صورت زیر عوض میشود.

```
__task void task1(void)
{
  int i = 0;
  while(1){
    os_dly_wait(100);
    for (i = 0; i < 0x9ffff; i++);   //process
    *AT91C_PIOB_ODSR ^= (1 << 0);   //Blink
```

```
  }
}
```

با توجه به اینکه زمان پیشفرض هر Tick ، 10 میلی ثانیه میباشد دستور os_dly_wait(100) به اندازه 1 ثانیه تاخیر ایجاد میکند. ولی در عمل چون از میزان تاخیر ایجاد شده توسط فرایند اصلی(حلقه for) یا تاخیر ایجاد شده توسط تسک های با اولویت برابر اطلاعی در دست نیست از روش دیگری که در زیر آمده استفاده میشود :

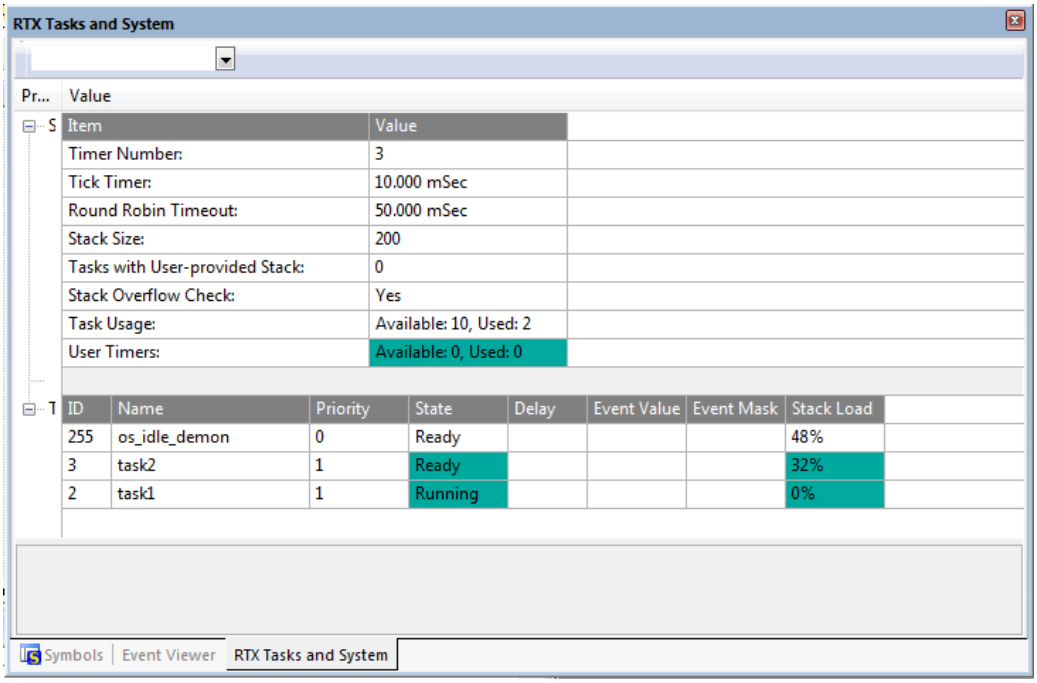
```
__task void task1(void)
{
  int i = 0;

os_itv_set(100);
  while(1){

    os_itv_wait();
    for (i = 0; i < 0x9ffff; i++);   //process
   *AT91C_PIOB_ODSR ^= (1 << 0);   //Blink
  }
}
```

در این روش ابتدا توسط دستور os_itv_set زمان تاخیر مشخص میشود و با هر بار فراخوانی تابع os_itv_wait زمان هدر رفته توسط سیستم عامل محاسبه شده و از تاخیر اصلی کم میشود.

برای تست برنامه های فوق میتوان از شبیه ساز Keil استفاده کرد و روشن، خاموش شدن پین های PB0 و PB1 را مشاهده کرد.همچنین شبیه ساز Keil قابلیت عیب یابی RTX را دارد. در موقع شبیه سازی میتوان این قابلیت را از مسیر Debug -> OS Support فعال کرد.



رویدادها :

در برنامه نویسی با استفاده از سیستم عامل رویداد ها نقش اساسی در کنترل روند اجرای تسک ها دارند.فرض کنید داده های مورد نیاز برای یک تسک توسط چند تسک دیگر تولید میشوند و تسک موبوطه تا زمان دریافت تمامی داده ها نمیتواند روال پردازشی خود را ادامه دهد.در چنین حالتی روش عادی بدین صورت است که این تسک باید مرتبا آماده بودن داده های مورد نیاز را چک کند.ولی این کار موجب هدر رفتن زمان پردازشی میشود.برای حل این مشکل راه حلی به نام رویدادها ایجاد شده است.هر تسک در RTX یک پرچم 16 بیتی دارد که میتواند برای ایجاد 16 رویداد مختلف استفاده شود.

در مشکل فوق تسک اصلی به سیستم عامل اطلاع میدهد که تا زمانیکه تعداد مشخصی از رویداد ها اتفاق نیفتد در وضعیت انتظار باقی خواهد ماند.با آماده شدن داده های مورد نیاز پرچم ها یک به یک فعال خواهند شد و زمانیکه تمام داده ها آماده شدند تسک مورد نظر به کار خود ادامه خواهد داد.

دستورات os_evt_wait_or و os_evt_wait_and برای منتظر شدن برای اتفاق افتادن رویداد ها تعریف شده است.در حالت AND باید تمام رویداد های مشخص شده اتفاق بیفتند ولی درحالت OR رخ دادن تنها یکی از رویداد ها کافی است. الگوی تعریف این توابع به صورت زیر است :

```
OS_RESULT os_evt_wait_or (
   U16 wait_flags,   /* Bit pattern of events to wait for */
        U16 timeout );    /* Length of time to wait for event */
```

متغیر timeout حداکثر زمانی که تسک میتواند منتظر رخ دادن رویداد بماند و wait_flags پرچم هایی که تسک منتظر آنها است را مشخص میکند.مقدار بازگشتی این تابع اگر برابر OS_R_EVT باشد نشان میدهد که رخ دادن رویداد باعث اتمام انتظار شده است و اگر برابر OS_R_TMO باشد نشان میدهد زمان Timeout تمام شده است.برای تشخیص اینکه کدام رویداد اتفاق افتاده است از تابع os_evt_get() استفاده میشود.

برای ایجاد یک رویداد از تابع os_evt_set استفاده میشود که به صورت زیر تعریف شده است:

```
void os_evt_set (
    U16    event_flags,     /* Bit pattern of event flags to set */
    OS_TID task );          /* The task that the events apply to */
```

event_flags به پرچمی که باید فعال شود و OS_TID به ID تسک موبوطه اشاره میکند.