Migrate RTX to CMSIS-RTOS

AN264, May 2014, V 1.0

ARM KEIL Microcontroller Tools

Abstract

This application note demonstrates how to migrate your existing RTX based application to the new CMSIS-RTOS layer.

Introduction

The CMSIS-RTOS API is a generic RTOS interface for Cortex-M processor-based devices. CMSIS-RTOS provides a standardized API for software components that require RTOS functionality and therefore gives serious benefits to the users and the software industry.

- CMSIS-RTOS provides basic features that are required in many applications or technologies such as UML or Java (JVM).
- The unified feature set of the CMSIS-RTOS API simplifies sharing of software components and reduces learning efforts.
- Middleware components that use the CMSIS-RTOS API are RTOS agnostic. CMSIS-RTOS compliant middleware is easier to adapt.

The ARM reference implementation of CMSIS-RTOS (CMSIS-RTOS RTX) is based on the RTX kernel and is part of the CMSIS software pack.

Prerequisites

MDK-ARM Version 5.00 or above (www.keil.com/mdk5).

A Device Family Pack (DFP) for your microcontroller device. Availability can be checked in the MDK5 pack installer or on www.keil.com/dd2.

Contents

Differences

The following table provides an overview of general differences between RTX and CMSIS-RTOS RTX.

	RTX	CMSIS-RTOS RTX
Time slice	RTX is using kernel ticks as the standard unit for measuring periods	CMSIS-RTOS fully relies on milliseconds as this provides a fully qualified timing base without knowing the configuration of a tick
Response time	*-no change-*	
Memory requirements	Code < 4KBytes / User Timer: 8 Bytes	Code < 5KBytes / User Timer: 20 Bytes
Terminology	Tasks / Events	Threads / Signals
Delivery	Integrated in MDK4 releases	Part of CMSIS software pack
System Tasks	Init task created by user	Main runs as first system task / 1 dedicated task for user timers
Priorities	Up to 250 numerical task priorities	Up to 7 named thread priorities
License	MDK-ARM EULA	BSD 3-clause license

Integrating CMSIS-RTOS in your Project for Migration

The following section lists the required steps for migrating an existing project from RTX to CMSIS-RTOS RTX.

1. Create a backup of your project.
2. Replace RTX_Conf.c with RTX_Conf_CM.c.
3. The new configuration file RTX_Conf_CM.c shows up in the project tree under CMSIS. Open it and transfer the following settings from your old configuration to your new configuration.
4. Change the API function calls as explained in the individual functional chapters following.

This chart helps to map the configuration settings from the old to the new configuration file:

RTX / RTX_Conf.c	CMSIS-RTOS RTX / RTX_Conf_CM.c
Number of concurrent running tasks	Number of concurrent running threads
Default Task stack size	Default Thread stack size
(See Note 1)	Main Thread stack size
Number of tasks with user-provided stack	Number of threads with user-provided stack size
(See Note 2)	Total stack size for threads with user-provided stack
Check for stack overflow	Check for stack overflow
Processor mode for thread execution	Run in privileged mode
Timer clock value [Hz]	Timer clock value [Hz]
Timer tick value [us]	Timer tick value [us]
Round-Robin Task switching	Round-Robin Task switching
Round-Robin Task Timeout [ticks]	Round-Robin Task Timeout [ticks]
Number of user timers	Timers Callback Queue size **(See Note 3)**
ISR FIFO Queue size	ISR FIFO Queue size

Notes:

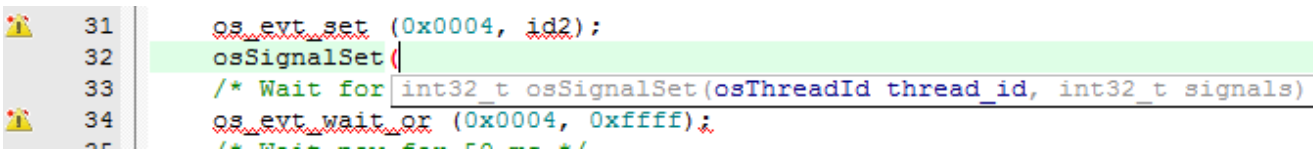
1) Set the Main Thread stack size to the RTX Task stack size or the user stack size used in the (now obsolete) os_sys_init_user call (see Kernel Management).
2) Remove all the user stack objects allocated in your project for os_tsk_create_user. The total size of all those objects is the value required for Total stack size for threads with user-provided stack.
3) Also enable the User Timers and allocate a sufficient Timer Thread stack size according to worst case stack usage of all timer callback functions.

The following chapters cover individual functional blocks and discuss differences.

Detailed descriptions of types, functions and arguments are available in the CMSIS-RTOS RTX documentation.

Hint:

Enable the "Text Completion" options in **Edit -> Settings**. Once you included the cmsis_os.h in any module the code completion and function parameter display will be of great help when integrating the new CMSIS-RTX functions.



Kernel Management

RTX	CMSIS-RTOS RTX
os_sys_init	*-obsolete-*
os_sys_init_user	*-obsolete-*
os_sys_init_prio	*-obsolete-*

CMSIS-RTOS RTX's first thread is **main**. You can move all code from the initialization task to **main** function replacing any **os_sys_init*** calls.

For compatibility reasons, call **osKernelInitialize()** and **osKernelStart()** in your main function. This is not required in the current releases of CMSIS-RTOS RTX, but can cause malfunction in future releases or when using a different CMSIS-RTOS compliant RTOS.

For example, see the following implementation of main:

```
int main (void) {
  osKernelInitialize ();                    // initialize CMSIS-RTOS

  // initialize peripherals here

  // create 'thread' functions that start executing,
  // example: tid_name = osThreadCreate (osThread(name), NULL);

  osKernelStart ();                         // start thread execution
}
```

After system initialization **main** can be used as a fully functional thread or it can be deleted (see **Task Management**). If you return from main, as in the above example, the thread will be properly terminated and removed from the thread list. Main cannot be restarted without a system reset after that.

NVIC Priority Group Settings

The initialization of CMSIS-RTOS RTX reads out the NVIC Priority Group configuration and sets the system handlers to the correct priority. If custom settings of groups are made after the startup code executes, the system may be miconfigured.

Make sure that you set the PendSV to the lowest level available after configuration of your custom interrupt priorities and groupings.

```
NVIC_SetPriority(PendSV_IRQn, <new priority>);
```

In the next chapter you will learn how to transfer your task creation to the new thread management of CMSIS RTOS.

Task Management

RTX	CMSIS-RTOS RTX
os_tsk_create	osThreadCreate
os_tsk_create_ex	osThreadCreate
os_tsk_create_user	osThreadCreate
os_tsk_create_user_ex	osThreadCreate
os_tsk_delete	osThreadTerminate
os_tsk_delete_self	osThreadTerminate(osThreadGetId())
os_tsk_pass	osThreadYield
os_tsk_prio	osThreadSetPriority
os_tsk_prio_self	osThreadSetPriority(osThreadGetId()…
os_tsk_self	osThreadGetId
isr_tsk_get	*-not available-*
OS_TID	osThreadId

osThreadCreate replaces all variants of os_tsk_create_* in a single function. A major difference is the way tasks are defined and declared. The prototype now carries an optional argument that can be passed by osThreadCreate. The macro osThreadDef statically declares the initial priority level and user stack size for the thread.

```
//RTX task definition:
__task void taskname (void)
{

}

//RTX task create
os_tsk_create(taskname);

//CMSIS-RTOS thread definition and declaration
void ThreadName (void const *arg);
osThreadDef (ThreadName, osPriorityNormal, 1, 0);

void ThreadName (void const* arg)
{

}

//CMSIS thread create
osThreadCreate(osThread(ThreadName), NULL);
```

Besides defining and creating RTX tasks vs. CMSIS-RTOS threads, all other thread management tasks can be migrated using the above table.

Hint:

The task prototype of CMSIS-RTOS now has an optional parameter. This can be freely used to pass any arguments to the task. This might be useful to configure multiple instances of a task to e.g. use different hardware resources, etc…

Time Management

RTX	CMSIS-RTOS RTX
os_dly_wait	osDelay
os_itv_set	*-not available-*
os_itv_wait	osDelay *or User Timers*
os_time_get	osKernelSysTick

The osDelay function is the only timed wait function in CMSIS-RTOS RTX.

os_itv_wait is no longer available but you can recreate this functionality using periodic user timers (see the User Timer section).

If a periodic interval task should run on the same as priority the user timer task is configured to, transform that task to a user timer callback.

If multiple periodic interval tasks at different priorities are used, it is recommended that the callback function only uses **osSignalSet** to wake up the respective thread based on the priorities defined.

Example:

```
osTimerId Timer1_Id;

void Timer1_Callback  (void const *arg) {

  /* Periodic tasks that formerly where in the interval task
     or wake up task with
     osSignalSet(id_of_interval_task, 0x0001); */


}
osTimerDef (Timer1, Timer1_Callback);

…

Timer1_Id = osTimerCreate (osTimer(Timer1), osTimerPeriodic, NULL);
osTimerStart  (Timer1_Id, 10);
```

Event Flag Management

RTX	CMSIS-RTOS RTX
os_evt_clr	osSignalClear
os_evt_get	osSignalClear
os_evt_set	osSignalSet
os_evt_wait_and	-not available-
os_evt_wait_or	osSignalWait
isr_evt_set	osSignalSet

osSignalClear combines the functions os_evt_clr and os_evt_get. Thus it is not possible to functionally replace the os_evt_wait_and function from RTX.

A special isr_evt_set function is no longer required. osSignalSet is safe to be used on interrupt level.

Mailbox Management

RTX	CMSIS-RTOS RTX
os_mbx_check	osMailAlloc(…,**0**)
os_mbx_declare	osMailQDef
os_mbx_init	osMailCreate
os_mbx_send	osMailPut
os_mbx_wait	osMailGet
isr_mbx_check	osMailAlloc(…, **0**)
isr_mbx_receive	osMailGet(…, **0**)
isr_mbx_send	osMailPut

Memory Allocation Functions

RTX	CMSIS-RTOS RTX
_declare_box*	osPoolDef
_init_box*	osPoolCreate
_alloc_box	osPoolAlloc
_calloc_box	osPoolCAlloc
_free_box	osPoolFree

Mutex Management

RTX	CMSIS-RTOS RTX
os_mut_init	osMutexCreate(osMutex())
os_mut_release	osMutexRelease
os_mut_wait	osMutexWait
-not available-	osMutexDelete
OS_ID	osMutexId
OS_MUT	osMutexDef()

Example of Declaration:

```
osMutexDef (MutexIsr);
osMutexId mutex_id;

mutex_id = osMutexCreate  (osMutex (MutexIsr));
```

Example of Usage:

```
osMutexWait    (mutex_id, 0);  //Wait until a Mutex becomes available.
...
osMutexRelease(mutex_id);      //Release a Mutex. Threads that wait for the same
                                 mutex will go into READY state.
```

Semaphore Management

RTX	CMSIS-RTOS RTX
os_sem_init	osSemaphoreCreate(osSemaphore())
os_sem_send	osSemaphoreRelease
os_sem_wait	osSemaphoreWait
isr_sem_send	osSemaphoreWait
-not available-	osSemaphoreDelete
OS_ID	osSemaphoreId
OS_SEM	osSemaphoreDef()

Example of Declaration:

```
osSemaphoreId semaphore;                        // Semaphore ID
osSemaphoreDef(semaphore);                      // Semaphore definition

semaphore = osSemaphoreCreate(osSemaphore(semaphore), 1);
```

Example of Usage:

```
val = osSemaphoreWait (semaphore, 1); // Wait 1ms for the free semaphore
if (val > 0) {
                                      // No time-out / semaphore was acquired
                                      // The interface is free now
  osSemaphoreRelease (semaphore);     // Return a token back to a semaphore
}
```

System Functions

RTX	CMSIS-RTOS RTX
os_resume	*-not available-*
os_suspend	*-not available-*
tsk_lock	*-not available-*
tsk_unlock	*-not available-*

CMSIS-RTOS does not provide System Functions that lock the Kernel like RTX does. This can be considered poor practice. If you want to lock the scheduler for a specific operation, it is recommended that you implement a Supervisor Call (SVC) function. See CMSIS-RTOS RTX User Guide.

User Timer Management

RTX	CMSIS-RTOS RTX
os_tmr_create	osTimerCreate
os_tmr_kill	osTimerDelete
os_tmr_call	*-not available-*
-not available-	osTimerStart
-not available-	osTimerStop

Revision History

- May 2014: Initial Version