

Keil ARM Real-Time Library (RL-ARM) Real-Time Executive (RTX) Kernel

Reference: Keil Help Files

Using RTX in a project

- ▶ Project > Options for Target:
 - ▶ Select: **Operating System > RTX Kernel**
- ▶ Copy **RTX_Config.c** from **\Keil\ARM\Startup** for Philips LPC21xx family
 - ▶ Modify RTX options as desired (next slide) to configure kernel
- ▶ Modify startup.s
 - ▶ Comment out **SWI_Handler B SWI_Handler**
 - ▶ Add: **IMPORT SWI_Handler**
- ▶ RTX kernel needs **RTL.h** include file
- ▶ Examples in **\Keil\ARM\RL\RTX\Examples**



RTX_Config.c – Kernel Configuration

Copy to project and set the following

- ▶ # concurrent running tasks (in any state)
- ▶ # tasks with user-provided stack
- ▶ Stack size for each task
- ▶ Enable/disable status checking (of stack)
- ▶ Specify CPU timer to be used as system tick timer
- ▶ Specify timer input clock frequency
- ▶ Specify timer tick interval
- ▶ Enable/disable round-robin task switching
- ▶ Specify time slice for round-robin task switching
- ▶ Define idle task operations
- ▶ Specify # of user timers (from on-chip timers)



Starting the RTX kernel

```
#include <rt1.h>
void main () {
    ...
    os_sys_init(task-name);
}
```

▶ Function `os_sys_init(task-name)`

1. Initialize & start the RTX kernel.
2. Create task “task-name” with priority = 1
3. Run “task-name”

▶ Must call from main C function.

▶ This function does not return

▶ `os_sys_init_prio(taskname, p);` //same, but assign priority p



RTX Tasks

- ▶ Scheduling unit is the “task”,
- ▶ Define with `__task` keyword:
 - ▶ `void func-name (void) __task {...}`
- ▶ First task started when RTX initialized
- ▶ Other tasks “created” by `os_task_create()`
 - ▶ Creates task, adds to ready queue, assigns task ID
 - ▶ Task ID used for all subsequent activities
 - `OS_TID id1,id2;`
 - `id2 = os_task_create(task1,p); //create task1, priority p`
 - `id1 = os_task_self(); //return ID of current task`
- ▶ Can also pass an argument to a created task
- `id2 = os_task_create_ex(task1,p,&var);`
- `... void task1(int *argv) __task {}`



Task priorities

- ▶ Priority = integer 0-255 (0 and 255 are reserved)
 - ▶ Higher value = higher priority
 - ▶ Priority 0 reserved for idle task (lowest priority)
 - ▶ Attempt to set priority = 0 results in priority = 1
- ▶ Task priority set when task created:
 - ▶ `os_sys_init(task);` //first task assigned priority 1
 - ▶ `os_sys_init_prio(task,p);` //first task assigned priority p
 - ▶ `os_task_create(task, p);` //new task assigned priority p
- ▶ Change priorities:
 - ▶ `os_task_prio(tid, p);` //tid = task id, new priority p
 - ▶ `os_task_prio_self(p);` //current task new priority p



Task states

- ▶ **RUNNING** – currently running
- ▶ **READY** – ready to run, RTX chooses highest-priority
- ▶ **WAIT_DLY** – waiting for a delay to expire
- ▶ **WAIT_ITV** – waiting for an interval to expire
- ▶ **WAIT_OR** – waiting for at least one event
- ▶ **WAIT_AND** – waiting for all of a set of events
- ▶ **WAIT_SEM** – waiting for a semaphore
- ▶ **WAIT_MUT** – waiting for a mutual exclusion release
- ▶ **WAIT_MBX** – waiting for a mailbox message
- ▶ **INACTIVE** – task not started or deleted



Preemptive multitasking

- ▶ RTX suspends a running task if a higher priority task (HPT) becomes ready to run
- ▶ Task scheduler executes at system tick timer interrupt.
- ▶ Task switch occurs when:
 - ▶ **Event** set for a HPT by the running task or by an interrupt service routine
 - ▶ Token returned to a **semaphore** for which HPT is waiting
 - ▶ **Mutex** released for which HPT is waiting
 - ▶ Message posted to a **mailbox** for which HPT is waiting
 - ▶ Message removed from a full mailbox, with HPT waiting to send another message to that mailbox
 - ▶ Priority of the current task reduced and a HPT is ready to run



Round-Robin Multitasking

- ▶ RTX gives time slice to each task
- ▶ Task executes for duration of time slice, unless it voluntarily stops (via a system “wait” function)
- ▶ RTX changes to next ready task with same priority
 - ▶ if none – resume current task
- ▶ Enable/disable in `RTX_Config.c`
 - ▶ also define time slice



Time management

- ▶ Pause and restart tasks using a timer
- ▶ `os_dly_wait(T);` //Suspend calling task for time T
 - ▶ T units = “system ticks” (1 to 0xFFFFE)
 - ▶ T = 0xFFFF indicates “infinite” time
 - ▶ Same rules for T in other functions
- ▶ `os_itv_set(T);` //Set timer interval T for task wakeup
- ▶ `os_itv_wait();` //Pause task until wakeup interval expires
 - ▶ Interval must have been set with `os_itv_set()`
 - ▶ Cannot mix `os_itv_wait` and `os_dly_wait` in one task



RTX Interprocess Communication

- ▶ Event flags – for task synchronization
 - ▶ Each task has 16 EFs.
 - ▶ A task can wait for EFs to be set by tasks or interrupts.
- ▶ Sempahores – control access to common resource
 - ▶ Semaphore object contains **tokens** (“counting” semaphore)
 - ▶ Task can request a token (put to sleep if none available)
- ▶ Mutexes – mutual exclusion locks
 - ▶ Can “lock” a common resource and unlock when done
 - ▶ Kernel blocks other tasks until unlocked
- ▶ Mailboxes – support message passing
 - ▶ Message is a pointer to a message frame
 - ▶ Task put to sleep if it requests a message that’s not available



RTX Semaphores

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

```
OS_SEM s1; //declare semaphore name s1
```

```
os_sem_init(s1,t); //set up s1; set initial tokens = t
```

```
os_sem_send(s1); //increment #tokens in s1
```

```
os_sem_wait(s1,timeout); //request token from s1
```

- ▶ **#tokens>0 (return OS_R_OK)**

- ▶ Return token and decrement #tokens

- ▶ Continue task, or go to ready list if higher priority task waiting

- ▶ **#tokens=0**

- ▶ Put task to sleep until token available or timeout period expires

- Timeout = 0 to 0xFFFE (0xFFFF for infinite period)

- ▶ Return OS_R_TMO if timeout before token available

- ▶ Return OS_R_SEM if token returned after some wait



RTX Event Flags

- ▶ `os_evt_clr(bp, tsk)` – clear one or more EFs of task `tsk`
 - ▶ `bp` = 16-bit mask, with 1 bits indicating EF #s
 - ▶ Example: `bp=0x8002` => EF #15 and EF #0
- ▶ `os_evt_set(bp, tsk)`
 - ▶ Set one or more EFs of `tsk`
- ▶ `os_evt_wait_and(bp, timeout)`
 - ▶ Wait for one or more EFs to set, or until timeout
 - ▶ Return `OS_R_EVT` if flags set, or `OS_R_TMO` if timeout
- ▶ `os_evt_wait_or(bp, timeout)`
 - ▶ Wait for any EF to set, or until timeout
- ▶ `N = os_evt_get();`
 - ▶ Retrieve EF that caused `os_evt_wait_or` to complete



RTX Mutual Exclusion (MUTEX)

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

```
OS_MUT m1; //declare MUTEX object m1
```

```
os_mut_init(m1); //initialize object m1
```

```
os_mut_wait(m1,timeout);
```

- ▶ Try to acquire mutex m1 and return one of:
 - ▶ OS_R_OK = acquired immediately (object was not locked)
 - ▶ OS_R_MUT = acquired after a wait (object was locked)
 - ▶ OS_R_TMO = not acquired after timeout
- ▶ os_mut_release(m1);
 - ▶ Decrement counter to release object



RTX Mailboxes

- ▶ `os_mbx_declare(name,cnt);`
 - ▶ Create mailbox (MB) object “name”
 - ▶ `cnt = # messages (msg pointers)` that can be entered
 - ▶ This is a “macro” – invoke outside of any C functions
- ▶ `os_mbx_init(name,size); //initialize MB for use`
 - ▶ `size = #bytes (#msgs defined by os_mbx_declare)`
- ▶ `os_mbx_send(name,msgptr,timeout);`
 - ▶ Put msgptr into mailbox “name”, if not full
 - ▶ Put task to sleep if mailbox full (timeout = max wait time)
- ▶ `os_mbx_wait(name,&msgptr,timeout)`
 - ▶ Get next msg pointer from mailbox “name”
 - ▶ If empty, put task to sleep until msg sent or timeout
- ▶ `os_mbx_check` – return # msgs that can be added to MB



Mailbox example

```
/* C module 1 */
#include <rtl.h>
os_mbox_declare (mbox1, 20); //

void task1 (void) __task {
    void *msg;
    os_mbox_init(mbox1, sizeof(mbox1));
    msg = alloc();
    /* create message content here */
    os_mbox_send(mbox1, msg);
}

/* C module 2 references mailbox from module 1*/
extern OS_MBX mbox1;
void task2 (void) __task {
    void *msg;
    os_mbox_wait (mbox1, &msg, 0xffff);
    /* process message contents */
    free(msg);
}
```

