ECE 254/MTE 241 Lab3 Draft
RL-RTX Kernel Programming
Last updated: 2012/11/05

Objective

This lab is to learn about, and gain practical experience in ARM RL-RTX kernel programming. In particular, you will add three functions to ARM RL-RTX library.

After this lab, students will have a good understanding of

- how to program a function to read kernel task control block related data structure;
- how to use SVC as the gateway to program OS functions;
- how to block and unblock a task by using context switching related kernel functions; and
- how real-time operating system manages fixed-size-block memory pool.

Starter files

Download the `lab3_example.zip` file from the lab website. It contains the following:

- A simple project that finds the starting address of free memory region at runtime.

Pre-lab Preparation

1. Read Sections 2.1, 2.2, 34.1 and 34.3 in [1]
2. Read the `rt_TypeDef.h` file in `lab3_example.zip` file and answer the following questions.
   - What are the purpose of `p_lnk, p_rlnk, p_dlnk,` and `p_blnk` variables in `struct OS_TCB`?
   - What is the purpose of `ret_val` in `struct OS_TCB`?
3. Read the `rt_Task.c, RTX_lib.c` files in `lab3_example.zip` file and answer the following questions.
   - What are the purposes of `os_active_TCB` array?
   - Is the os idle daemon TCB an element of `os_active_TCB`?
   - What is the purpose of variables `mp_tcb` and `mp_stack`?
4. Read the `HAL_CM3.c` file in the `lab3_example.zip` file and answer the following questions.
   - What registers are saved on the task stack? (Hint: check `init_stack` function)
   - How to determine the start and end address of a task stack?
   - How to determine the current stack pointer of a task?
5. Read the `rt_Membox.c` file in the `lab3_example.zip` file and answer the following questions.
   - What is the data structure used to keep track of free memory boxes in a memory pool?
6. Read the `rt_Mailbox.c` file in the `lab3_example.zip` file and answer the following questions.
   - When a process is blocked with `WAIT_MBX` state, it will be resumed once a message appears in the mailbox (assuming timeout value is set to 0xFFFF). What is the return code of `os_mbx_wait()` after the task is resumed?

- Inside the `rt_mbx_wait()` function, there are three return statements. The first one returns `OS_R_OK`. The last two return `OS_R_TMO`. Does this mean the answer to the question above is either `OS_R_OK` or `OS_R_TMO`? Why or why not?

7. The `os_dly` appears in multiple kernel files. It is an ordered list. What is the purpose of variable `os_dly`? What criteria are used to order the items in this list? Can you use `os_dly` list to enqueue TCBs that are waiting for memory blocks in Part B? Why or why not?

Lab3 Tutorial
Introduction to Keil LPC1768 Hardware and Programmers Model (on-line slides)

Lab3 Assignment

There are two parts of this assignment. They are Part A and Part B.

Part A
To get familiar with kernel source code, a good start is to write a function to retrieve a kernel data structure. In this assignment, you are to implement a primitive to obtain the task status information from the RTX at runtime given the task id. The function description is as follows.

- `OS_RESULT os_tsk_get (OS_TID task_id, RL_RASK_INFO *buffer)`

  The primitive returns information about a task. The system call returns a `rl_task_info` structure , which contains the following fields:

```
typedef struct rl_task_info {
  U8    state;        /* Task state      */
  U8    prio;         /* Execution priority     */
  U8    task_id;      /* Task ID value for optimized TCB access */
  U8    stack_usage;  /* Stack usage percent value. eg.=58 if 58% */
  void  (*ptask)();   /* Task entry address     */
} RL_TASK_INFO;
```

  The `state` field describes the state of this task and is one of:

  `INACTIVE`

  Tasks which have not been started or tasks which have been deleted are in `INACTIVE` state.

  `READY`

  Tasks which are ready to run are in the `READY` state.

  `RUNNING`

  The task that is currently running is in the `RUNNING` state. Only one task at a time can be in this state.

  `WAIT_DLY`

  Tasks which are waiting for a delay to expire are in the `WAIT_DLY` state. The task is switched to the `READY` state once the delay has expired.

  `WAIT_SEM`

  Tasks which are waiting for a semaphore are in the `WAIT_SEM` state. When the token is obtained from the semaphore, the task is switched to the `READY` state.

  `WAIT_MUT`

Tasks which are waiting for a free mutex are in the `WAIT_MUT` state. When a mutex is released, the task acquires the mutex and switches to the `READY` state.

`WAIT_MBX`

Tasks which are waiting for a mailbox message are in the `WAIT_MBX` state. Once the message has arrived, the task is switched to the `READY` state. Tasks waiting to send a message when the mailbox is full are also put into the `WAIT_MBX` state. When the message is read out from the mailbox, the task is switched to the `READY` state.

`WAIT_MEM`

Tasks which are waiting for memory are in the `WAIT_MEM` state. Once the memory is available, the task is switched to `READY` state. The `os_mem_alloc()` function is used to place a task in `WAIT_MEM` state.

These states are described in details in the RL-ARM Real-Time Library User's Guide → Theory of Operation→ Task Management section except for `WAIT_MEM` state. Read Lab3 Assignment Part B regarding how tasks are blocked upon calling `os_mem_alloc()` function.

The `prio` field describes the priority of this task.

The `task_id` field describes the id of task assigned by the OS.

The `stack_usage` describes how much stack space is used by this task. The value is the percent value. For example, if 58% of this task stack is used, `stack_usage` is set to 58.

The `ptask` field describes the entry address of this task function.

The function returns `OS_R_OK` on success and `OS_R_NOK` otherwise.

Part B

A memory pool allows fixed-size memory block allocation. It is a commonly seen memory management scheme used in real-time operating systems. One way of implementing the memory pool is to use compile time array. This is the approach taken by the stocked ARM RL-RTX library. Another way of implementing the memory pool is to pre-allocate memory blocks in the free memory region where the image does not reside. We can start the memory pool from the end address of the image and pre-allocate blocks of memory in the free memory region. We will take the latter approach in this lab.

You are to add two dynamic memory management RTX functions into the ARM RTX-RL library to manage a memory pool. The memory pool starts at the end address of the image and ends at an address smaller than 0x10008000. The dynamic memory can be used by the requesting task for storing local variables. When a task does not need the requested memory block anymore, the memory should be returned to the kernel (i.e. free the memory). The function descriptions are as follows.

- ```
  void *os_mem_alloc (unsigned char flag)
  ```

  The primitive allocates a fixed-size of memory to the calling task and returns a pointer to the allocated memory. Valid values for `flag` are:

  `MEM_NOWAIT`

  The primitive returns a `NULL` pointer if there is no memory available

  `MEM_WAIT`

  When there is not enough memory available, the calling task is blocked until enough memory becomes available. If several tasks are waiting for memory and memory becomes available, the highest priority waiting task will get the memory.
  ```

Both `MEM_NOWAIT` and `MEM_WAIT` are macros you define yourself. One valid example can be:

```
#define MEM_NOWAIT 0
#define MEM_WAIT 1
```

- `OS_RESULT os_mem_free (void *ptr)`

  The primitive frees the memory allocation pointed by `ptr`. It returns `OS_R_OK` on success and `OS_R_NOK` otherwise.

Requirements

1. Implement the functions in Part A and Part B aforedescribed.
2. Create a set of testing tasks to demonstrate that you have successfully implemented the required functions. Your test tasks should do the following tests.
   - A task periodically prints task status of each task in the system.
   - A task can allocate a fixed size of memory
   - A task will get blocked if there is no memory available when `os_mem_alloc()` is called with flag `MEM_WAIT`.
   - A blocked on memory task will be resumed once enough memory is available in the system.
   - Create a situation that multiple tasks with different priorities are all blocked waiting for memory. When memory becomes available, test whether it is the highest priority task that gets the memory first.

Demonstration
This lab requires a demo of your system by using the simulator in debug mode of μVision IDE to a lab TA. Demo reservation will be done through course book system. Each demo is 20-30 minutes. Both group members are required to attend the demo. You will get **0** mark on this lab if you miss the demo without 48 hour notice.

Post-Lab Deliverables
Submit a .zip file with name `lab3.zip` that contains the following item(s) to the course book system before the deadline. Unless you notify the lab TAs and the lab instructor by email, we will mark the latest submission when there are multiple submissions presented.

- Entire lab3 μVision project folder

A Note on Grace Days
The source code cannot be submit late due to the scheduled demo.

References

[1] LPC17xx User Manual, Rev2.0, 2010.