Lab 6: Introduction to RTX Real-Time Operating System (RTOS)

EE-379 Embedded Systems and Applications
Electrical Engineering Department, University at Buffalo
Last update: Cristinel Ababei, April 2013

1. Objective

The objective of this lab is to learn how to write simple applications using RTX (ARM Keil's real time operating system, RTOS). These simple examples involve flashing LEDs, counting, and displaying on the board's LCD display a "simulation" of how to drive a step-motor. In addition, we'll look at FreeRTOS too.

Note: A lot of the discussion in this lab is based on ARM's online information.

2. RTOS

Simple embedded systems typically use a *Super-Loop* concept where the application executes each function in a fixed order. Interrupt Service Routines (ISR) are used for time-critical program portions. This approach is well suited for small systems but has limitations for more complex applications. These limitations include the following disadvantages:

- Time-critical operations must be processed within interrupts (ISR)
  - ISR functions become complex and require long execution times
  - ISR nesting may create unpredictable execution time and stack requirements
- Data exchange between *Super-Loop* and ISR is via global shared variables
  - Application programmer must ensure data consistency
- A *Super-Loop* can be easily synchronized with the System timer, but:
  - If a system requires several different cycle times, it is hard to implement
  - Split of time-consuming functions that exceed Super-Loop cycle
  - Creates software overhead and application program is hard to understand
- *Super-Loop* applications become complex and therefore hard to extend
  - A simple change may have unpredictable side effects; such side effects are time consuming to analyze.

These disadvantages of the *Super-Loop* concept are solved by using a **Real-Time Operating System (RTOS)**.

A RTOS separates the program functions into self-contained tasks and implements an on-demand scheduling of their execution. An advanced RTOS, such as the Keil RTX, delivers many benefits including:

- **Task scheduling** - tasks are called when needed ensuring better program flow and event response
- **Multitasking** - task scheduling gives the illusion of executing a number of tasks simultaneously
- **Deterministic behavior** - events and interrupts are handled within a defined time
- **Shorter ISRs** - enables more deterministic interrupt behavior
- **Inter-task communication** - manages the sharing of data, memory, and hardware resources among multiple tasks
- **Defined stack usage** - each task is allocated a defined stack space, enabling predictable memory usage
- **System management** - allows you to focus on application development rather than resource management (housekeeping)

3. RL-RTX

The Keil RTX (Real Time eXecutive) is a royalty-free deterministic RTOS designed for ARM and Cortex-M devices. It is one of the components of RL-ARM, the RealView Real-Time Library (RL-ARM). RTX and its source code are available in all MDK-ARM Editions [1].

RTX allows one to create programs that simultaneously perform multiple functions (or tasks, statically created processes) and helps to create applications which are better structured and more easily maintained. Tasks can be assigned execution priorities. The RTX kernel uses the execution priorities to select the next task to run (preemptive scheduling). It provides additional functions for inter-task communication, memory management and peripheral management.

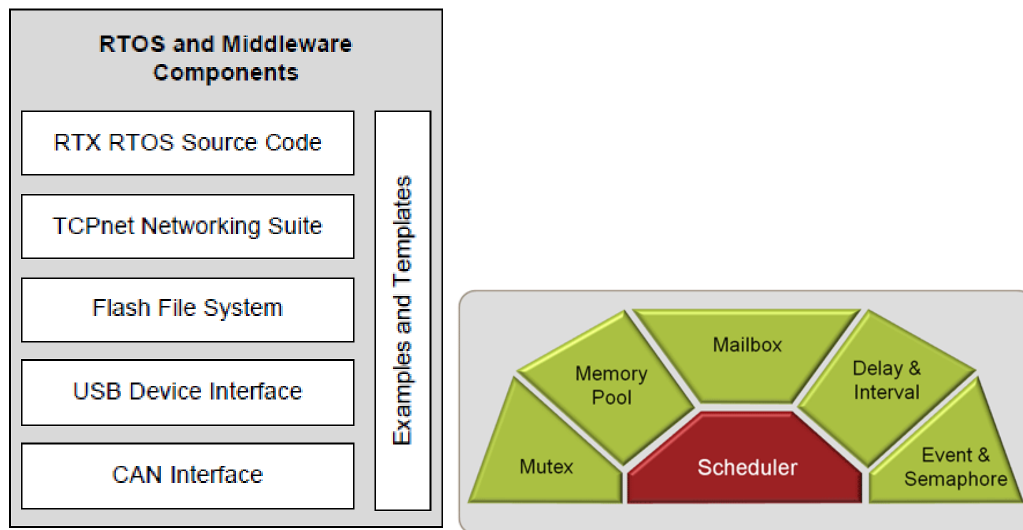


Figure 1 RL-ARM Real-Time Library and RTX Real-Time Operating System diagrams.

The main features of RTX include:

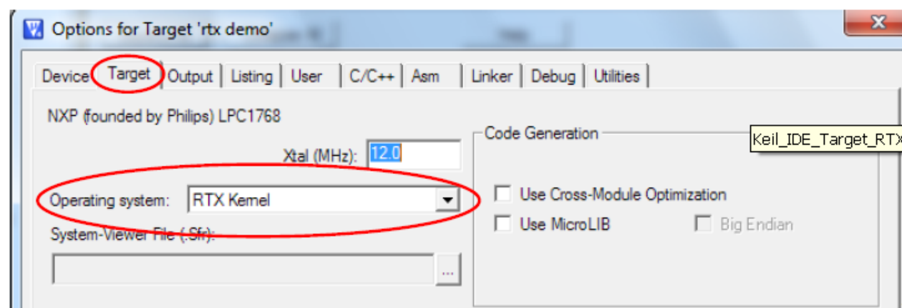
- Royalty-free, deterministic RTOS with source code
- Flexible Scheduling: round-robin, pre-emptive, and collaborative
- High-Speed real-time operation with low interrupt latency
- Small footprint for resource constrained systems
- Unlimited number of tasks each with 254 priority levels
- Unlimited number of mailboxes, semaphores, mutex, and timers
- Support for multithreading and thread-safe operation
- Kernel aware debug support in MDK-ARM
- Dialog-based setup using µVision Configuration Wizard

4. Example 1: Creating an RL-RTX Application

RTX programs are written using standard C constructs and compiled with the RealView Compiler. The header file `RTL.h` defines the RTX functions and macros that allow you to easily declare tasks and access all RTOS features.

The following are the basic steps (in the context of this course) to create an application that uses RL-RTX library:

1. Create a new project and use NXP LPC1768 as the target processor.
2. Copy the `RTX_Conf_CM.c` (C:\Keil\ARM\RL\RTX\Config) to the project folder and add it to your project.
   Note: Examples that come with the Keil's uVision installation contain in their own project folders local copies of the `RTX_Conf_CM.c`, which may be different from the one in C:\Keil\ARM\RL\RTX\Config due to their different time-stamps…
3. Open the `RTX_Conf_CM.c` under the Configuration Wizard and set the CPU to 100000000. You need to set the total number of tasks the system can manage. The default value is 6 and normally is sufficient.
4. Open up the Target Option window and activate the "Target" tab. Choose "RTX Kernel" as the operating system (default is set to None). Click Use MicroLIB in the code- generation window.



5. Now you can start to develop an RTX application. Start with the `os_tsk_*()` function reference which is within the µVision Help File.

Let's now create a simple example project called **lab6_ex1_4tasks** and place it in a new folder called **lab6_ex1_4tasks/**. The entire uVision project folder is included in the downloadable archive for this lab. The **main_4tasks.c** is also listed in Appendix A of this document. This is a simple RTX application that lights up the first LED when it starts and then displays the "Hello World!" string on the LCD screen. Then the system creates an initialization task named `init` which spawns four tasks `task1, task2, task3`, and `task4`. `task1` counts and displays digits 0-9 in a round robin fashion. `task2` makes a LED blinks every one second. `task3` keeps incrementing a global counter. `task4` keeps decrementing a global counter.

All these tasks are created with the same priority. Tasks of equal priority will be run in *Round Robin* when they enter the ready state. Higher priority tasks will preempt (or interrupt) a running task. Lower priority tasks will not run. Note that, it might have been more intuitive if Round Robin Timeout had been called Round Robin Task Switching Time. If no tasks are ready to run then RTX will execute its *idle demon*. This is located in the file **RTX_Conf_CM.c**. One can insert user code to run during this idle time.

Configure the project to use "RTX Kernel" as the operating system. Also, remember to configure the CPU speed to 100 MHz by means of the `RTX_Conf_CM.c` configuration wizard. Compile, and download to the board. Observe operation and comment.

Now, run a simulation debug session and use the following debug features of uVision:

- **RTX Tasks and System Window**
  Click Debug -> OS Support -> RTX Tasks and Systems. This window updates as the RTX runs. You can watch task switching and various system and task related information in this window.
- **RTX Event Viewer**
  Click Debug -> OS Support -> Event Viewer. Event Viewer provides a graphical representation of how long and when individual tasks run.

For more details on the above, please take some time to read entirely the “Keil RTX RTOS: The Easy Way” tutorial [2]. Please read especially section 7, which describes other ways to switch tasks aside from the aforementioned *Round Robin*.

5. Example 2: Simulating a stepper-motor driver

a) ***Semaphores: MUTEX***

There are several types of semaphores (the basic idea behind each type is the same):

- Binary
- Counting
- Mutex

Semaphores are typically used in one of two ways:

1) To control access to a shared device between tasks. A printer is a good example. You don't want 2 tasks sending to the printer at once, so you create a binary semaphore to control printer access. When a device wishes to print, it attempts to "take" the semaphore. If the semaphore is available, the task gets to print. If the semaphore is not available, the task will have to wait for the printer.
2) Task synchronization. By tasks taking and giving the same semaphore, you can force them to perform operations in a desired order.

Counting semaphores are used when you might have multiple devices (like 3 printers or multiple memory buffers).
Binary semaphores are used to gain exclusive access to a single resource (like the serial port, a non-reentrant library routine, or a hard disk drive). A counting semaphore that has a maximum value of 1 is equivalent to a binary semaphore (because the semaphore's value can only be 0 or 1).
Mutex semaphores are optimized for use in controlling mutually exclusive access to a resource. There are several implementations of this type of semaphore.

Mutex stands for “**Mutual Exclusion**” and is a specialized version of a semaphore. Like a semaphore, a Mutex is a container for tokens. The difference is that a Mutex is initialized with one token. Additional Mutex tokens cannot be created by tasks. The main use of a Mutex is to control access to a chip resource such as a peripheral. For this reason, a Mutex token is binary and bounded. Apart from this, it really works in the same way as a semaphore. First, we must declare the Mutex container and initialize the Mutex:

```
os_mut_init (OS_ID mutex);
```

Then any task needing to access the peripheral must first acquire the Mutex token:

```
os_mut_wait (OS_ID mutex, U16 timeout);
```

Finally, when we are finished with the peripheral, the Mutex must be released:

```
os_mut_release (OS_ID mutex);
```

Mutex use is much more rigid than semaphore use, but is a much safer mechanism when controlling absolute access to underlying chip registers.

Take some time to read more on:

- Mutex management routines in RTX:
  http://www.keil.com/support/man/docs/rlarm/rlarm_ar_mut_mgmt_funcs.htm
- Building Applications with RL-ARM, Getting Started (included also in the lab files):
  http://www.keil.com/product/brochures/rl-arm_gs.pdf

b) RTX_Blinky Example

This is the RTX_Blinky example that comes bundled with the Keil uVision software installation. You can find it in **C:\Keil\ARM\Boards\Keil\MCB1700\RTX_Blinky** or inside the downloadable archive with the files of this lab.

In this example, four LEDs are blinking simulating the activation of the four output driver stages phase A,B,C,D. This is also displayed on the LCD display of the MCB1700 board. Compile, and download to the board. Observe operation and comment.

Take some time and read the source code from **blinky.c** in order to get a good understanding of what's happening. See how a mutex is utilized to control the access to the LCD display. If you want to see how some of the `os_*` functions are implemented, read the RTX source code from **C:\Keil\ARM\RL\RTX/**.

There is also an abundance of information on ARM's website. For example, here is the description of `os_mut_wait`:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0062a/rlarm_os_mut_wait.htm

6. Lab assignment

*This is optional and **available only to undergraduates** in this course. If done correctly, you may get up to 3% of the final grade.*

Create a new uVision project and write a program using the Super-Loop approach discussed in this lab to implement Example 2. You should not use RTX at all but your program should achieve the same behavior as RTX_Blinky example. Develop your program using only the programming techniques that we used earlier in this course. This exercise is to outline the differences between RTX and Super-Loop embedded programming approaches.

7. Credits and references

[0] Building Applications with RL-ARM, Getting Started (included also in the lab files):
http://www.keil.com/product/brochures/rl-arm_gs.pdf
[1] RTX Real-Time Operating System;
-- http://www.keil.com/arm/selector.asp
-- http://www.keil.com/rtos/
-- http://www.keil.com/rl-arm/rtx_rtosadv.asp
-- http://www.keil.com/rl-arm/kernel.asp
-- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0062a/rlarm_ar_your_first_app.htm
-- http://www.keil.com/support/man/docs/rlarm/rlarm_ar_svc_func.htm

[2] Robert Boys, Keil RTX RTOS: The Easy Way V0.4. 2012;
http://www.sase.com.ar/2012/files/2012/09/RLarmSteps.pdf
[3] Irene Huang, ECE-254 Labs; https://ece.uwaterloo.ca/~yqhuang/labs/ece254/document.html
[4] On mutexes;
-- RL-ARM, Getting Started (Chapter 2): http://www.keil.com/product/brochures/rl-arm_gs.pdf
-- Daniel Robbins, Common threads: POSIX threads explained - The little things called mutexes;
http://www.ibm.com/developerworks/library/l-posix2/
-- A.D. Mashall, Programming in C, UNIX System Calls and Subroutines using C, 2005 (Chapter on Mutual Exclusion Locks);
http://www.cs.cf.ac.uk/Dave/C/node31.html#SECTION003110000000000000000
-- M. Mitchell, J. Oldham, and A. Samuel. Advanced linux programming, 2001 (Chapter 4);
http://advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf
-- Multithreaded Programming (POSIX pthreads Tutorial); http://randu.org/tutorials/threads/
-- MSDN Introduction to Mutex Objects; http://msdn.microsoft.com/en-us/library/windows/hardware/ff548097(v=vs.85).aspx
-- Lock-free programming; http://preshing.com/20120612/an-introduction-to-lock-free-programming

APPENDIX A: Listing of main_4tasks.c of Example 1 project

```
// simple RL-RTX application to blink an LED and
// to display 0-9 in a round robin fashion on LCD
// display of MCB1700 board
// this is meant to be a "hello world" example for
// RTX application development;

#include <stdio.h>
#include <LPC17xx.h>
#include <RTL.h>
#include "GLCD.h"
#include "LED.h"

#define __FI 1 // Use font index 16x24

// global counters will count 60 seconds up and down;
int g_counter1 = 0, g_counter2 = 60;
char text_buffer[8];

// displays 0-9 in a round robin fashion
__task void task1(void)
{
    int i = 0;
    GLCD_DisplayString(3, 0, 1, "Task 1:");

    for (;;i++) {
        GLCD_DisplayChar(3, 7, 1, i+'0');
        os_dly_wait(100);
        // Note1: The Delay function pauses the calling task by the amount
        // of ticks passed as the argument. Control will switch to the
        // next task ready else passes to the idle demon. After the
        // specified number of ticks has expired, the calling task will
        // be placed in the ready state. The delay does not use up
```

```
        // processing time with a loop.
        if (i == 9) {
            i = -1;
        }
    }
}

// toggles LED #7 at P2.6 every second
__task void task2(void)
{
    GLCD_DisplayString(4, 0, 1, "Task 2:LED");
    for (;;) {
        LED_on(7);
        os_dly_wait(60);
        LED_off(7);
        os_dly_wait(40);
    }
}

// task that keeps incrementing a global counter
__task void task3(void)
{
    GLCD_DisplayString(5, 0, 1, "Task 3:");
    for (;;) {
        g_counter1++;
        if (g_counter1 == 60) g_counter1 = 0; // reset;
        os_dly_wait(100);
        sprintf(text_buffer, "%d", g_counter1);
        GLCD_DisplayString(5, 7, __FI, (uint8_t*)text_buffer);
    }
}

// task that keeps decrementing a global counter
__task void task4(void)
{
    GLCD_DisplayString(6, 0, 1, "Task 4:");
    for (;;) {
        g_counter2--;
        if (g_counter2 == 0) g_counter2 = 60; // reset;
        os_dly_wait(100);
        sprintf(text_buffer, "%d", g_counter2);
        GLCD_DisplayString(6, 7, __FI, (uint8_t*)text_buffer);
    }
}

// initialization task that spawns all other tasks
__task void init(void)
{
    os_tsk_create(task1, 1);  // task 1 at priority 1
    os_tsk_create(task2, 1);  // task 2 at priority 1
    os_tsk_create(task3, 1);  // task 3 at priority 1
    os_tsk_create(task4, 1);  // task 4 at priority 1
    os_tsk_delete_self();     // task must delete itself before exiting
}
```

```
int main(void)
{
    // (1) initialize the LPC17xx MCU;
    SystemInit();

    // (2) initialize GLCD and LED;
    LED_init();
    GLCD_Init();
    LED_on(0); // turn on LED #0 at P1.28

    GLCD_Clear(Yellow);
    GLCD_DisplayString(0, 0, 1, "RTX Hello World! :-)");

    // (3) initilize the OS and start the first task
    os_sys_init( init);
}
```