Assignment 1 Periodic Tasks in Zephyr RTOS (100 points)

As shown in the following diagram, periodic tasks in real-time embedded systems can be simply expressed as endless loops with time-based triggers. In the task body, specific computation should be done and locks must be acquired when entering any critical sections.

```
TASK periodic_task()
{
        < local variables >
        initialization() and wait_for_activation();
        while (condition) {
                <task body>
                wait_for_period();
        }
}
```

(from "*Ptask: an Educational C Library for Programming Real-Time Systems on Linux*" By Giorgio Buttazzo and Giuseppe Lipari, *EFTA 2013*)

In this assignment, you are asked to develop an application program that uses multiple threads to implement the periodic task models on Zephyr environment. The task body is a local computation and is simulate by a busy loop of *x* iterations in the assignment, i.e.

<compute_1> <lock_*m*> <compute_2> <unlock_*m* > <compute_3>

where "lock_*m*" and "unlock_*m*" are locking and unlocking operations on mutex *m*, and "compute_n" indicates a local computation. To simulate a local computation, we will use a busy loop of *n* iterations in the assignment, i.e.

```
volatile uint64_t n;

while(n > 0) n--;
```

The input to your program is a specification of a task set. The task parameters of each task is defined in *struct Tasks*:

```
struct task_s
{
        char t_name[32];        // task name
        int priority;           // priority of the task
        int period;             // period for periodic task in milliseconds
        int loop_iter[3];       // loop iterations for compute_1, compute_2 and compute_3
        int mutex_m;            // the mutex id to be locked and unlocked by the task
};
```

For instance, in the given *task_model.h*, a task set of 4 tasks are defined:

```
#define THREAD0 {"task00", 2, 50, {400000, 400000, 400000}, 1}
#define THREAD1 {"task11", 3, 160, {800000, 900000, 800000}, 0}
#define THREAD2 {"task22", 4, 220, {200000, 2000000, 400000}, 1}
#define THREAD3 {"task33", 5, 360, {200000, 2000000, 400000}, 2}

struct task_s threads[NUM_THREADS]={THREAD0, THREAD1, THREAD2, THREAD3};
```

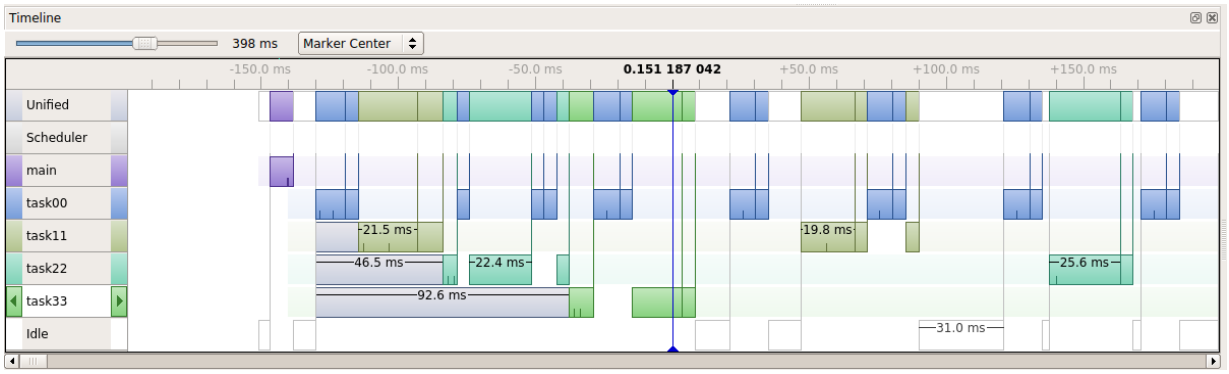
Your program should use the task set specification defined in *task_model.h* and create a thread for each task to perform task operations periodically. To verify the scheduling events of the tasks, you will check scheduling events displayed in Segger's SystemView Timeline windows which can be run in a host machine and connected to MIMXRT1050-EVKB via J-Link.

Additional requirements of the program are:

1. The task set specification is given in *task_model.h* which should be included in your program. You can modify task parameters and the numeric constants defined by other macros to test your program. However, please don't insert additional code in *task_model.h* since a *task_model.h* with different numeric values will be used to grade your program.
2. The numeric constant *TOTAL_TIME* in *task_model.h* indicates the total execution time in *ms* that your program should run. When the execution is completed, all threads should be terminated properly. Any waiting tasks (wait_for_period) should exit immediately. Any running or ready task should exit after completing its current iteration of task body.
3. The numeric constant *NUM_MUTEXES* in *task_model.h* indicates the maximal number of mutex locks that are needed for critical sections. For each mutex, the two inheritance ceiling configurations should be used to permit unlimited elevation and no inheritance.
4. For a periodic task, the execution deadline of task body is at the end of the period. Your implementation should put out an error message (you can use "printk") at the time that a deadline is missed. If a task misses its deadline, its next invocation should be started immediately as soon as the task finishes its current task body.
5. All tasks should be activated at the same time. A new shell root command "activate" should be implemented to trigger task activation once all periodic tasks complete their initialization.

Since the number of tasks in the task set is not fixed, it is difficult to hard code all tasks. One approach is to develop a generic task functions which take in the specification of task body, and period or event, to perform task execution. Note that this generic task functions must be reentrant.

Along with the program, you are required to submit a pdf file that contains screenshots from *SystemView* to verify the scheduling of the tasks and the list of task parameters that you use to generate the screenshots. Also, by setting CONFIG_PRIORITY_CEILING=0 or 10, you can produce two screenshots: one is with priority inheritance protocol enabled for all mutexes and the other one with no inheritance protocol. On the screenshot, please identify at least one occurrence of task blocking and preemption. An example screenshot is as follows.



Due Date

The due date is 11:59pm, Feb. 11.

What to Turn in for Grading

- Your submission is a zip archive, named RTES-LastName-FirstInitial_03.zip, that includes
  - A pdf report to include a brief description of your implementation and the SystemView screenshots of two runs of *trace_*app with CONFIG_PRIORITY_CEILING=0 and 10, respectively. The input task set of the two runs must be stated in the report. The screenshots should be chosen to illustrate the effect of priority inheritance. Don't forget to add your name and ASU id in the report.
  - An application folder, named *trace_app*, to include all your implementation of the assignment. The folder should contain *CMakeLists.txt, prj.conf, readme.txt*, and a source file directory *src.*
  - The readme text file describes all the commands you use to build and run your program.
- Note that any object code or temporary files should not be included in the submission. Submit the zip archive to the course Canvas by the due date and time.
- Please make sure that you comment the source files properly and the readme file includes a description about how to make and use your software. Don't forget to add your name and ASU id in the readme file.
- There will be 20 points penalty per day if the submission is late. Note that submissions are time stamped by Canvas. If you have multiple submissions, only the newest one will be graded. If needed, you can send an email to the instructor to drop a submission.
- The assignment must be done individually. No collaboration is allowed, except the open discussion in the forum on Canvas. The instructor reserves the right to ask any student to explain the work and adjust the grade accordingly.
- Failure to follow these instructions may cause deduction of points.
- Here are few general rule for deductions:
  - Cannot compile or compilation error -- 0 point for the assignment.
  - Must have "–Wall" flag for compilation -- 5-point deduction for each warning.
  - 10-point deduction if no compilation or execution instruction in README file.
  - Source programs are not commented properly -- 10-20-point deduction.
- ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed.