2IN70 - Track 2: Real-Time Architectures
Practical Training - Exercises 6 -

Today we will continue experimenting with the task synchronization primitives in the μC/OS-II real-time operating system running on the Freescale EVB9S12XF512E board.

6.1 Implementing precedence constraints

In this exercise you will gain insight into implementing precedence constraints between tasks. The CodeWarrior project for this exercise is in the directory `exercise6_1`. *Note that the RTI interrupt frequency is set to 2 kHz, meaning that the tick period is 0.5ms. An ATD conversion takes about 2.15ms. Keep these in mind when you analyze the timing of the tasks.*

1. The application is comprised of three tasks specified in `main.c`. Note that Task1 is not a periodic task. It is created using the standard μC/OS-II interface for creating task. While the task function argument to the `OSTaskCreatePeriodic()` is called repeatedly with the specified period, the task function argument to `OSTaskCreate()` is called only once. Task1 therefore implements its repetitive behavior using a while loop with synchronization in its body.

   Study the implementation and *before running the application on the board*, describe on a piece of paper how you expect the leds to behave.

2. Run your program on the board and verify your prediction.

3. **Identify the problem with the current implementation.**

4. **Propose two solutions. Motivate your answer by drawing a timeline for each solution, including when leds are toggled.**

   The phasing and period task parameters are assumed to be application requirements and cannot be changed.

5. *Before running the application on the board*, describe on a piece of paper how you expect the leds to behave after implementing any of the proposed solutions.

6. Assume that we extend the application with a lowest priority greedy task (representing e.g. video processing for the entertainment system, or route-planning for navigation). Saying that a task is greedy means that it consumes as much processing time as it can, e.g. in a `while (1)` {...} loop without any synchronization).

   Evaluate the two solutions you have proposed in Step 4 based on this new setup. Motivate your answer by drawing a timeline for each solution.

7. **Implement the better solution from the previous step.**

8. Run your program on the board and verify your prediction from Step 5.

6.2 Avoiding deadlock (optional)

In this exercise you will gain experience with avoiding deadlock. The CodeWarrior project for this exercise is in the directory `exercise6_2`.

1. **Implement an example application experiencing a deadlock.**

   Hint: Introduce several tasks sharing several mutexes. Use the `BusyWaiting()` function to simulate the task workload.

2. Verify that your program indeed suffers from deadlock by running it on the board or in the simulator.

3. **Propose and implement two methods for solving the deadlock.**

   Note: You can enable the SRP based implementation by setting the `ENABLE_SRP` flag inside the `os_srp_cfg.h` file. The SRP implementation introduces its own mutex type called `Resource`. If you chose to use SRP, you will therefore need to change your `OS_EVENT` mutex declarations to `Resource`.

4. Verify your proposed solutions by running your program on the board or in the simulator.