IL2206 EMBEDDED SYSTEMS

Laboratory 2: Introduction to Real-Time Operating Systems

Version 1.0

1 Objectives

This laboratory will help you to

- acquire hands-on experience with real-time system concepts;
- develop a real-time application on a typical real-time operating systems; and
- explore real-time application's performance and memory footprint trade-offs.

2 Preliminary Work

This laboratory shall introduce the students to Real-Time Operating Systems (RTOS). The topic of this laboratory is the synchronization and communication between tasks in a real-time operating system applications. An RTOS offers several objects for this purpose, like semaphores, mailboxes, message queues and events. These objects are also used to connect interrupts to tasks. In this laboratory, students shall deepen their understanding and practice the use of these components by implementing a multi-task application.

If you want to learn more about real-time systems, you may be interested in the course *IL2212 Embedded Software* that is given in period 3.

The students are strongly encouraged to interact with other students through the course's online platform (Canvas) in case technical issues are encountered. The discussion forum will be monitored by the course assistants, who will take actions in case of larger problems. It is very important that students are well-prepared for the labs. Students who are not properly prepared shall not get help from the assistants during the lab sessions!

NOTE: *The preparation tasks shall be documented and shall be shown to the laboratory assistants at the beginning of the lab session! Students can work together with their lab partner. However, both students in a group need to demonstrate individual work and proficiency in all tasks. All program code shall be well-structured and well-documented. The language used for documentation is English.*

Read this laboratory manual in detail and complete the preparation tasks before your lab session in order to be allowed to be examined.

2.1 Literature

Read chapter 16 and 17 of Labrosse's book on MicroC/OS-II [Lab02] with focus on mechanisms for synchronization and communication. The book chapters are available via the course web page. It is important that you get a good understanding how these mechanisms work in order to use them efficiently. Go through the MicroC/OS-II documentation and check the available mechanisms and their associated functions. Read [Alt].

You should also have a look on the project templates provided in the lab repository, for example `hello_ucosii` or the ones generated by the Eclipse-Nios2 IDE.

literature read ◯

2.2 Questions

1. For this and the following questions you may need to again consult chapter 16, which describes the functions in MicroC/OS-II in detail. When dealing with message queues does MicroC/OS-II provide a
   (a) non-blocking write?
   (b) blocking write with timeout?
   (c) blocking write?
   (d) non-blocking read?
   (e) blocking read with timeout?
   (f) blocking read?
2. Which memory management functions can you use to allocate or free memory in MicroC/OS-II? How do they work and what is the advantage of these functions compared to usual C-functions as *malloc*?
3. The function `OSQPend` returns a pointer `void*`.
   (a) What is a `void*`-pointer and how can it be used?
   (b) How can you regenerate the message that has been send using the `void*`-pointer.
4. The last step of the laboratory is to optimize the code size. Think already now of possible optimizations, and prepare the optimized code. Write down how you plan to optimize the code! How can you measure the code size?

5. MicroC/OS-II offers functions to measure the

    - task stack `OSTaskStkChk()`,
    - memory partition `OSMemQuery()`.

    Understand how to use the functions in order to be able to optimize the program for code size.

2.2 completed ◯

3 RTOS Tasks

This section is an introduction to RTOS tasks their communication and synchronization mechanisms.

3.1 Accessing a Shared Resource: I/O

In the program `TwoTasks.c` found under `il2206-lab/app/lab2-rtos` in the course repository, two different tasks write messages to the standard output. Please observe also how you can get statistic information about the stack size.

- What is the expected output of this program in the first glance?
- The program might not show the desired behavior. Explain why this might happen.
- What will happen if you take away `OSTimeDlyHMSM()` statements? Why?
- Semaphores can help you to get the desired program behavior. What are semaphores? How can one declare and create a semaphore in MicroC/OS-II?
- How can a semaphore protect a critical section? Give a code example!
- Who is allowed to lock and release semaphores? Can a task release a semaphore that was locked by another task?
- Explain the mechanisms behind the command `OSSemPost()` and `OSSemPend()`!

Modify the program so that it shows the expected behavior and save it as a file `TwoTasksImproved.c`.

Draw the new application as a block diagram containing processes, semaphores and shared resources. *Use the graphical notation which has been used in the lectures and exercises for this purpose, and included in Figure 4 from Appendix A.*

3.1 completed ◯

3.2 Communication by Handshakes

Modify the `TwoTasks.c` in such a way that implements two tasks which communicate with each other via a handshaking procedure. Both tasks have two states 0 and 1. In each state the tasks shall print a message to indicate the status of the active task, e.g. “Task 0 - State 0”, if task 0 is in state 0. The program shall then show the following execution pattern

```
Task 0 - State 0
Task 1 - State 0
Task 1 - State 1
Task 0 - State 1
Task 0 - State 0
Task 1 - State 0
...
```

independent of the task periods. Use semaphores to solve the problem! *Before you write the program draw state diagrams including semaphores for task 0 and task 1.* Save the new program as `Handshake.c`.

3.2 completed ◯

3.3 Shared Memory Communication

Modify your program `Handshake.c` in such a way that task 0 sends integer numbers (starting from 1) to task 1. Task 1 shall multiply the numbers with -1 and send them back to task 0. Task 0 shall then print these numbers to the console. For the communication between task 0 and task 1 a single memory location `sharedAddress` shall be used, i.e. both task 0 and task 1 read and write to/from this location! Save the file as `SharedMemory.c`.

The execution of the program shall give the following output.

```
Sending   : 1
Receiving : -1
Sending   : 2
Receiving : -2
...
```

Draw a block diagram containing processes, semaphores and shared resources!

3.3 completed ◯

3.4 Context Switch

MicroC/OS-II provides also functions to set (`OSTimeSet`) and measure (`OSTimeGet`) time. The function prototypes are shown below.

```
INT32U OSTimeGet(void);
void OSTimeSet(INT32U ticks);
```

However, these functions are far too slow to measure the context switch time, since the real time clock frequency is usually between 10 and 100 Hz. Thus in order to measure the context switch overhead time, you have to use a timer or performance counter for measurement in the same way as in the previous laboratory.

Modify your program `Handshake.c` to measure the time that is needed for a context switch. Save this program as `ContextSwitch.c`. Please think carefully how to measure the context switch time as accurate as possible. You are needed to make an average on at least 10 measured values - excluding the values which differ so much to the average. 3.4 completed ◯

4 Cruise Control Application

In the next part of this lab you will have to develop a toy cruise control application using the DE2/DE-115 board. If activated, a cruise control system shall maintain the speed of a car at a constant value that has been set by the driver.

The system has the following inputs:

- **Engine (ON/OFF).** The engine is turned on, in case the signal `ENGINE` is active. The engine can only be turned off, if the speed of the car is 0 m/s.
- **Cruise Control (ON/OFF).** The cruise control is turned on, if the signal `CRUISE_CONTROL` is activated and if the car is in top gear (`TOP_GEAR` is active) and if the velocity is at least $20\frac{m}{s}$ and the signals `GAS_PEDAL` and `BRAKE_PEDAL` are inactive.
- **Gas Pedal (ON/OFF).** The car shall accelerate, if the signal `GAS_PEDAL` is active. The cruise control shall be deactivated, if `GAS_PEDAL` is active.
- **Brake (ON/OFF).** The car shall brake, when the signal `BRAKE` is active. Also the cruise control shall be deactivated, if the signal `BRAKE` is activated.
- **Gear (HIGH/LOW).** The car has two different gear positions (high, low) indicated by the signal `TOP_GEAR`. If `TOP_GEAR` is active the the gear position is high, otherwise low. The cruise control is deactivated, when the gear position is moved to low.

The inputs are connected to the following IO-units on the DE2/DE2-115 boards according to Table 1.

Signal	Pin	LED
`ENGINE`	SW0	LEDR0
`TOP_GEAR`	SW1	LEDR1
`CRUISE_CONTROL`	KEY1	LEDG2
`BRAKE_PEDAL`	KEY2	LEDG4
`GAS_PEDAL`	KEY3	LEDG6

Table 1: Connection of Signals to IO-Pins on the DE2-Board

The system consists of four periodic tasks as illustrated in Figure 1. The car will travel on an oval track of the length 2400m, which has the profile as illustrated in Figure 2.

4.1 Understanding the Initial Program

In the course repository, under `il2206-lab/app/lab2-cruise` there is an executable skeleton program `cruise_skeleton.c`, which implements the

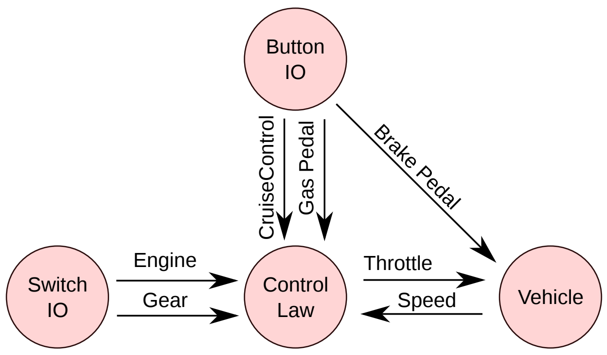

Figure 1: Tasks in the Cruise Control System

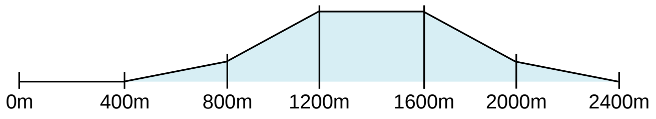

Figure 2: Profile of the oval track, which has a length of 2400m

vehicle task and an initial skeleton for the control task. In the skeleton program, the control task uses a constant throttle of 40. The task `VehicleTask` implements the behavior of the car and its functionality shall not be changed during this laboratory. The only permitted code modification in `VehicleTask` is the replacement of the timer (see Task 4.2).

Study the initial program carefully in order to have a clear understanding of the program. Execute it on the board.

4.1 completed ◯

4.2 Use Soft Timers to Implement Periodic Tasks

The skeleton program uses the statement `OSTimeDlyHMSM` to implement periodic tasks, which will not give an exact period. Use instead soft timers for this purpose and connect them with semaphores to your tasks. Use the same period for the soft timers as in the original skeleton program.

See: `OSTmrCreate` in μC/OS-II Reference Manual, Chapter 16.

NOTE: Do not forget to integrate the C-code for the soft timer by selecting the option 'Enable code for Timers' below 'RTOS-Options' in 'System Library Properties'.

4.2 completed ◯

4.3 I/O-Tasks

Create the tasks `ButtonIO` and `SwitchIO`, which read the buttons and switches on the DE2-board periodically. The task `SwitchIO` creates the signals `ENGINE` and `TOP_GEAR`, while the task `ButtonIO` creates the signals `CRUISE_CONTROL`, `GAS_PEDAL` and `BRAKE_PEDAL`. Use the red LEDs to indicate that a switch is active and the green LEDs to indicate that a button is active, as specified in Table 1.

4.3 completed ◯

4.4 Control Law

Implement the control law in the `ControlTask` so that it fulfills the specification from Section 4. Note that the braking functionality is implemented inside the `VehicleTask`, whereas the `ControlTask` sets the throttle.

The control law shall react according to the state of the buttons and switches. When the cruise control is activated, the current velocity shall be maintained with a maximum deviation of $\pm 2\frac{m}{s}$ for velocities of at least $25\frac{m}{s}$. Use the green LED LEDG0 to indicate that the cruise control is active.

Implement the dummy functions `show_position` and `show_target_velocity`. `show_position` shall indicate the current position of the vehicle on the track with the six leftmost red LEDs as specified in Table 2.

LED	Position
LEDR17	[0m, 400m)
LEDR16	[400m, 800m)
LEDR15	[800m, 1200m)
LEDR14	[1200m, 1600m)
LEDR13	[1600m, 2000m)
LEDR12	[2000m, 2400m]

Table 2: LED assignment to show the position of the vehicle

`show_target_velocity` shall display the target velocity, which the cruise control is trying to maintain, on the seven segment display (HEX5 and HEX4). The display shall be reset to 0 when the cruise control is deactivated.

4.4 completed ○

4.5 Watchdog

To allow for the detection of an overloaded system, add a watchdog task and an overload detection task to the system. The overload detection task shall report to the watchdog with an 'OK' signal, when there is no overload condition, i.e. the system's utilisation is less than 100%. In case the watchdog task does not receive the 'OK' signal during a specified interval, the watchdog should infer that the system is overloaded, i.e. reached 100% utilisation, and it shall consequently issue an overload warning message.

Add another task to impose an extra load on the system. It shall be possible to dynamically adjust the amount of processing time that the task utilizes. To set the utilization, the switches SW4 to SW9 shall be used. The switch pattern shall be interpreted as a binary number (with SW4 as the lowest bit), i.e., 2^6 values can be represented. The utilization shall be adjustable in 2% steps. Everything higher than 100% (i.e., all numbers above 50) shall be considered as 100% utilization. Hint: think about the system's hyperperiod while interpreting the utilisation ratio 'x%'.

Figure 3 illustrates the resulting system with overload detection. The grey box contains the original system. Note that there are *no connections* from the original system to the overload detection system, i.e., watchdog task and overload detection task.

Test at which utilization, in '%', a system overload occurs under different circumstances (e.g. cruise control activated, car standing still, pushing gas pedal, ...).

Pay attention to the priority design choice for the tasks, i.e. watchdog, extra-load task, overload detection task, so that any system overload is effectively detected.

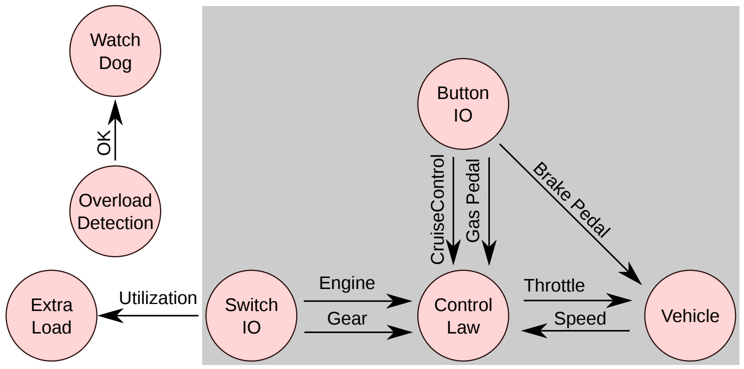

Figure 3: Cruise Control System with Overload Detection

4.5 completed ◯

4.6 Optimization

Check the size of your program and write down the numbers.

In the skeleton program, the stack size has been set to 2048. Find out how much stack each task uses at most and reduce the stack sizes accordingly.

Optimize the system for code size using different techniques and determine to what extent the size has decreased. You will need to reduce the footprint of your application to such extent that you will be able to fit *the code* (i.e. program) inside the `onchip_memory`, and the rest (i.e. data) on `sram`.

Test your new program using the two configurations in Table 3, where *config. 1* assumes data on SDRAM and program on SRAM and *config. 2* assumes data on SRAM and program on the on-chip memory.

Configuration	Size	Overload at
non-optimized, config. 1		
optimized, config. 1		
optimized, config. 2		

Table 3: Comparison between program versions

At which utilization, in '%', a system overload occurs? Did the optimization for size decrease the performance of the system (considering only *config.1*)? Was it worth it, in terms of performance, to optimize for size so that it fits inside the on-chip memory?

4.6 completed ◯

5 Surprise Task

You will be given a 'surprise task', which you need to conduct during the laboratory session. In order to be able to solve the surprise task you need to have a very good understanding of the preparation tasks!

References

[Alt] Altera. *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial.* Version TU-NIOSII-MCRC/OS-II-1.2.

[Lab02] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel.* CMP Books, 2002.

A RTOS graphical primitives

In order that the lab assistants understand your applications, draw them as graphics using the symbols in Figure 4.

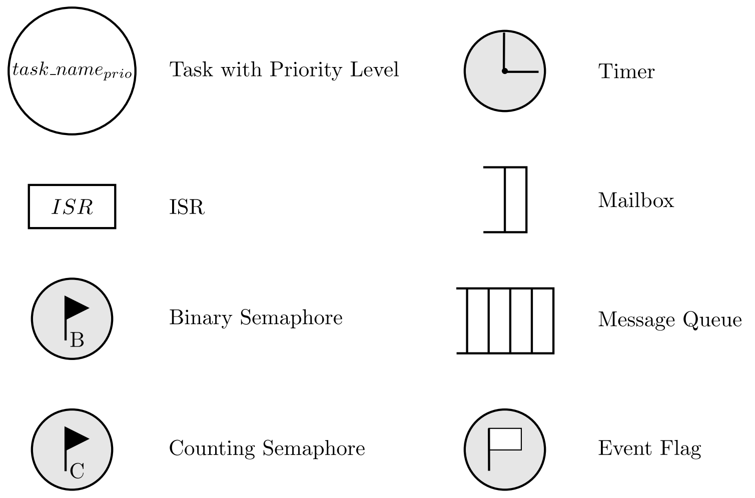

Figure 4: Graphical primitives for describing RTOS applications