KTH Information and
Communication Technology

IL2206 EMBEDDED SYSTEMS

Laboratory 1: Introduction to Real-Time Operating Systems

1 Objectives

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 operating systems, you may be interested in the course *IL2212 Embedded Software* that is given in period 3.

2 Preparation Tasks

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

It is very important that students are well-prepared for the labs, since both the lab rooms and the assistants are expensive and limited resources. Thus these resources shall be used efficiently. Students, who are not prepared for the laboratory, 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, each student must be able to answer questions on the preparation tasks. All program code shall be well-structured and well-documented. The language used for documentation is English.*

2.1 Literature

Read chapter 16 and 17 of Labrosse's book on MicroC/OS-II [**?**] 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 [**?**].

2.1 completed ◯

You should also have a look on the project templates that are available when you create a new application in the Nios II IDE.

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 `OSTaskStackChk()`,
   - memory partition `OSMemQuery()`.

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

2.2 completed ◯

2.3 Accessing a Shared Resource: I/O

In the following program `TwoTasks.c` that is available on the home page two different tasks write messages to the standard output. Please observe also how you can get statistic information about the stack size.

```
0  // File: TwoTasks.c
1
2  #include <stdio.h>
3  #include "includes.h"
4  #include <string.h>
5
6  #define DEBUG 1
7
8  /* Definition of Task Stacks */
9  /* Stack grows from HIGH to LOW memory */
10 #define   TASK_STACKSIZE       2048
11 OS_STK    task1_stk[TASK_STACKSIZE];
12 OS_STK    task2_stk[TASK_STACKSIZE];
13 OS_STK    stat_stk[TASK_STACKSIZE];
14
15 /* Definition of Task Priorities */
16 #define TASK1_PRIORITY      6  // highest priority
17 #define TASK2_PRIORITY      7
18 #define TASK_STAT_PRIORITY 12  // lowest priority
19
20 void printStackSize(char* name, INT8U prio)
21 {
22     INT8U err;
23     OS_STK_DATA stk_data;
24
25     err = OSTaskStkChk(prio, &stk_data);
26     if (err == OS_NO_ERR) {
27         if (DEBUG == 1)
28             printf("%s (priority %d) - Used: %d; Free: %d\n",
29                     name, prio, stk_data.OSUsed, stk_data.OSFree);
30     }
31     else
32     {
33         if (DEBUG == 1)
34             printf("Stack Check Error!\n");
35     }
36 }
37
38 /* Prints a message and sleeps for given time interval */
39 void task1(void* pdata)
40 {
41   while (1)
42   {
43     char text1[] = "Hello from Task1\n";
44     int i;
45
46     for (i = 0; i < strlen(text1); i++)
47       putchar(text1[i]);
48     OSTimeDlyHMSM(0, 0, 0, 11); /* Context Switch to next task
49                                  * Task will go to the ready state
50                                  * after the specified delay
51                                  */
52   }
53 }
54
55 /* Prints a message and sleeps for given time interval */
56 void task2(void* pdata)
57 {
58   while (1)
59   {
60     char text2[] = "Hello from Task2\n";
61     int i;
62
63     for (i = 0; i < strlen(text2); i++)
64       putchar(text2[i]);
65     OSTimeDlyHMSM(0, 0, 0, 4);
66   }
67 }
68
69 /* Printing Statistics */
70 void statisticTask(void* pdata)
71 {
72     while(1)
```

```
73      {
74          printStackSize("Task1", TASK1_PRIORITY);
75          printStackSize("Task2", TASK2_PRIORITY);
76          printStackSize("StatisticTask", TASK_STAT_PRIORITY);
77      }
78  }
79
80  /* The main function creates two task and starts multi-tasking */
81  int main(void)
82  {
83    printf("Lab 3 - Two Tasks\n");
84
85    OSTaskCreateExt
86     (task1,                          // Pointer to task code
87      NULL,                           // Pointer to argument passed to task
88      &task1_stk[TASK_STACKSIZE-1],   // Pointer to top of task stack
89      TASK1_PRIORITY,                 // Desired Task priority
90      TASK1_PRIORITY,                 // Task ID
91      &task1_stk[0],                  // Pointer to bottom of task stack
92      TASK_STACKSIZE,                 // Stacksize
93      NULL,                           // Pointer to user supplied memory
94                                      // (not needed here)
95      OS_TASK_OPT_STK_CHK |           // Stack Checking enabled
96      OS_TASK_OPT_STK_CLR             // Stack Cleared
97      );
98
99    OSTaskCreateExt
100    (task2,                          // Pointer to task code
101     NULL,                           // Pointer to argument passed to task
102     &task2_stk[TASK_STACKSIZE-1],   // Pointer to top of task stack
103     TASK2_PRIORITY,                 // Desired Task priority
104     TASK2_PRIORITY,                 // Task ID
105     &task2_stk[0],                  // Pointer to bottom of task stack
106     TASK_STACKSIZE,                 // Stacksize
107     NULL,                           // Pointer to user supplied memory
108                                     // (not needed here)
109     OS_TASK_OPT_STK_CHK |           // Stack Checking enabled
110     OS_TASK_OPT_STK_CLR             // Stack Cleared
111     );
112
113   if (DEBUG == 1)
114   {
115     OSTaskCreateExt
116      (statisticTask,                   // Pointer to task code
117       NULL,                            // Pointer to argument passed to task
118       &stat_stk[TASK_STACKSIZE-1],     // Pointer to top of task stack
119       TASK_STAT_PRIORITY,              // Desired Task priority
120       TASK_STAT_PRIORITY,              // Task ID
121       &stat_stk[0],                    // Pointer to bottom of task stack
122       TASK_STACKSIZE,                  // Stacksize
123       NULL,                            // Pointer to user supplied memory
124                                        // (not needed here)
125       OS_TASK_OPT_STK_CHK |            // Stack Checking enabled
126       OS_TASK_OPT_STK_CLR              // Stack Cleared
127       );
128   }
129
130   OSStart();
131   return 0;
132 }
```

Listing 1: `TwoTasks.c`

- 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 we take away `OSTimeDlyHMSM()` statements? Why?

Semaphores can help you to get the desired program behavior. In MicroC/OS-II a semaphore has first to be declared as global variable of type `OS_EVENT` and must the be created in the main program by `OSTaskCreate`.

```
OS_EVENT  *aSemaphore;
...
int main(void)
{
   aSemaphore =  OSSemCreate(1); // binary semaphore (1 key)
   ...
   // Task Creation
}
```

The semaphore can be used to protect a critical section by the functions `OSSemPend` and `OSSemPost` as shown below. It is not necessary that the same task that locked the semaphore is the one that releases it. Other tasks can also release the semaphore.

```
void mytask(void *pdata)
{
  INT8U err;
  ...
  OSSemPend(aSemaphore, 0, &err); // Trying to access the key
  // Critical section
  OSSemPost(aSemaphore);  // Releasing the key
  ...
}
```

The command `OSSemPost()` increments the semaphore value. If tasks are waiting on the semaphore, `OSSemPost()` removes the highest priority task waiting for the semaphore and makes it ready to run. The scheduler is then called to determine if the awakened task is then the highest priority task ready to run.

The command `OSSemPend()` tries to access the semaphore key. If successful, the semaphore value is decremented and the task continues execution. If unsuccessful the calling task is put into the waiting list for that semaphore and control is returned to the scheduler.

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

2.3 completed ◯

2.4 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`.

2.4 completed ◯

2.5 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!

2.5 completed ◯

2.6 Cruise Control Application

If activated, the cruise control system shall maintain the speed of the 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-board:

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.

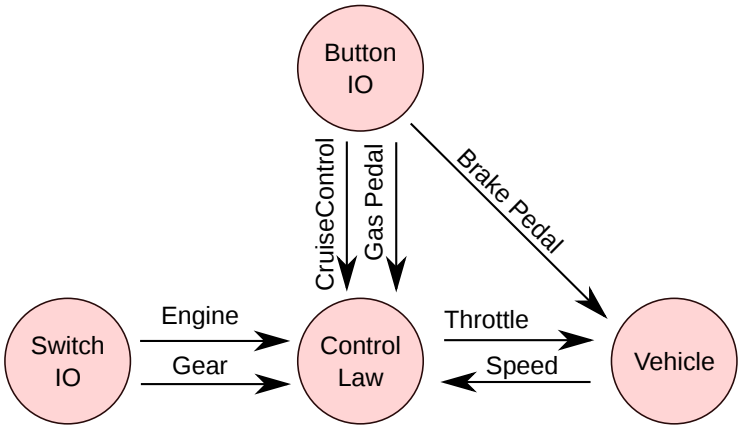

Figure 1: Tasks in the Cruise Control System

The car will travel on an oval track of the length 2400m, which has the profile as illustrated in Figure 2.

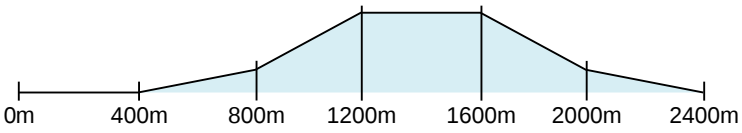

Figure 2: Profile of the oval track, which has a length of 2400m

2.6.1 Understanding the Initial Program

Available from the course web page is an executable skeleton program `cruise_skeleton.c`, which implements the 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 2.6.2).

Study the initial program carefully in order to have a clear understanding of the program. Execute it on the DE2-board.

◯ 2.6.1 completed

2.6.2 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'.

◯ 2.6.2 completed

2.6.3 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.

◯ 2.6.3 completed

2.6.4 Control Law

Implement the control law in the `ControlTask` so that it fulfills the specification from Section 2.6. 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 segement display (HEX5 and HEX4). The display shall be reset to 0 when the cruise control is deactivated. ◯ 2.6.4 completed

2.6.5 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. In case the watchdog task does not receive the signal during a specified interval, it shall 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.

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 a system overload occurs under different circumstances (e.g. cruise control activated, car standing still, pushing gas pedal, ...).

Pay attention to choose the priorities for the tasks so that any system overload is actually detected.

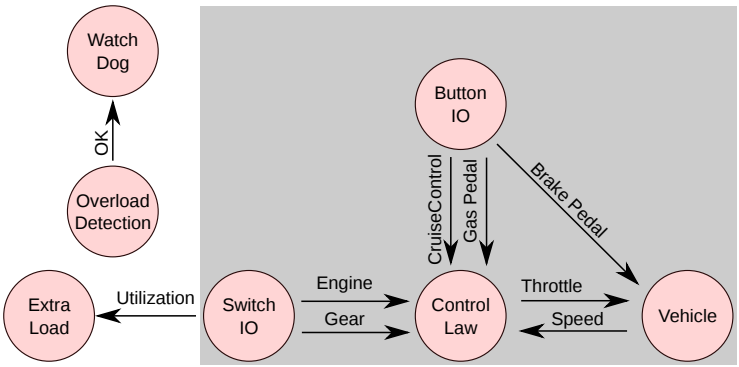

Figure 3: Cruise Control System with Overload Detection

◯ 2.6.5 completed

2.6.6 (Optional) Optimization

Check the size of your program, using the command `nios2-elf-size` 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 and determine to what extent the size has decreased. Test again, at which utilization a system overload occurs. Did the optimization for size decrease the performance of the system?

3 Laboratory Tasks

3.1 Demonstration of Preparation Tasks

Demonstrate the programs that you have developed in the preparation tasks for the laboratory staff. Be prepared to explain your programs in detail.

3.2 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!

4 Examination

In order to pass the laboratory the student must

- have completed the corresponding preparation tasks of Section 2 before the lab sessions
- have demonstrated the preparation tasks for the laboratory staff and completed the surprise task (Section 3).

References