Embedded systems
Exercise session 3
Software architectures

Principles

Decision guidelines:

1. Make an inventory of all tasks.
2. Identify those that can be performed by interrupt routines.
3. Is preemption needed? (Yes $\rightarrow$ RTOS).
4. Are interrupts needed? (No $\rightarrow$ RR).
5. Is static CPU assignment acceptable? (Yes $\rightarrow$ RR/I, No $\rightarrow$ WQ).

Important points:

1. Only urgent and short operations in interrupt routines!
2. When peripherals are busy, the processor can do something else.

Problem 1

The onboard microcontroller of a homemade quadcopter has to receive and execute instructions sent by the pilot via a wireless link (such as taking off, landing, flying in a specified direction, . . .). In addition to processing those instructions, it constantly stabilizes the vehicle. In order to monitor and control stability, the quadcopter is equipped with two 3-axis accelerometers and one gyroscope. Moreover, it includes a camera whose signal, along with other monitoring data, has to be sent back to the pilot. The microcontroller has to perform the following tasks:

- Receiving and processing instructions: 10 times per second, execution time = 5 ms.
- Controlling the motors: 100 times per second, execution time = 2 ms.
- Acquiring data from sensors: period of 4 ms, execution time = 1 ms.
- Sending the video signal and other data: period of 40 ms, execution time = 20 ms.

1. What is the best software architecture for this system? Justify.
2. Using pseudocode, give a suitable global structure for this software.

List of tasks:

- τ_1: Receiving & processing instructions: period = 100 ms, exec. time = 5 ms.
- τ_2: Motors control: period = 10 ms, exec. time = 2 ms.
- τ_3: Sensor data acquisition: period = 4 ms, exec. time = 1 ms.
- τ_4: Sending data: period = 40 ms, exec. time = 20 ms.

Can some tasks be performed by interrupt routines?

No! (Execution times in *milliseconds*.)

Is preemption needed?

Yes! (τ_2 and τ_3 over τ_4.)

$\rightarrow$ RTOS.

Task priorities?

$P(\tau_2) > P(\tau_3) > P(\tau_1) > P(\tau_4)$.

Tasks communication:

- Global variables for the current mode of operation and last sensor data.
- Binary semaphore for controlling concurrent access to those variables.

```
#include <rtos.h>
#include <rtos-semaphores.h>
#include "datastruct.h"
#include "instruction.h"

static volatile mode  current_mode;
static volatile sensor_values  last_sensors_data;
static semaphore  mode_sem, sensors_sem;

void task1(void)  /* Receiving and processing instructions */
{
  instruction *in;
  mode          m;

  in = !! get data from wireless link
  if (in)
    {
      m = process_instruction(in);

      wait(mode_sem);
      current_mode = m;
      signal(mode_sem);
    }
}
```

```
void task2(void)  /* Motor control */
{
  mode            m;
  sensors_values v;

  wait(mode_sem);
  m = current_mode;
  signal(mode_sem);

  wait(sensors_sem);
  v = last_sensors_data;
  signal(sensors_sem);

  !! compute control variables according to m and v
  !! command motors
}

void task3(void)  /* Sensor data acquisition */
{
  sensors_values  v;

  v = !! get sensors data
```

```
  wait(sensors_sem)
  last_sensors_data = v;
  signal(sensors_data)
}

void task4(void)  /* Sending data */
{
  !! get camera data
  !! send camera data
}

void main(void)
{
  !! initialize OS
  !! initialize data structures

  create_periodic_task(task1, 100, 2);
  create_periodic_task(task2, 10, 4);
  create_periodic_task(task3, 4, 3);
  create_periodic_task(task4, 40, 1);
```

```
  mode_sem = create_binary_semaphore(1);
  sensors_sem = create_binary_semaphore(1);

  !! start tasks sequencing
}
```

Problem 2

A portable audio player is equipped with a microcontroller that has to react to user actions, display messages on a small screen, and send audio data to an MP3 decoder. The function of the MP3 decoder is to convert audio data into analog signals. This decoder is connected to a dedicated digital input of the microcontroller, on which it signals whether it is ready or not to receive data. The microcontroller needs to run the following tasks:

- A task τ_1 checking the state of the keyboard 25 times per second. Its execution time is negligible.
- A task τ_2 sending a fixed number of characters to the screen, at most 25 times per second. This task requires 3 ms to complete.
- A task τ_3 sending audio data to the MP3 decoder whenever it is ready to receive it. Sending data can take up to 80 ms. In the worst case, this operation might be requested 10 times per second. The microcontroller is the master of the transaction; it can pause the data transfer in order to perform other tasks, provided that the duration of this pause does not exceed 10 ms.

Note: The input of the microcontroller connected to the decoder can be configured to trigger interrupts.

List of tasks:

- τ_1: Checking keyboard: period = 40 ms, exec. time = ε.
- τ_2: Updating screen: period $\geq$ 40 ms, exec. time = 3 ms.
- τ_3: Sending audio data: period: $\geq$ 100 ms, exec. time = 80 ms.
- τ_4: React to ready-to-receive signal: $\geq$ 10 ms, exec. time = ε.

Notes:

- τ_1 and τ_2 can be grouped into a single task with a period of 40 ms.
- τ_4 implemented by an interrupt routine.

Is preemption needed?

Yes! ($\tau_{1/2}$ over τ_3.)

$\rightarrow$ RTOS.

Task priorities?

$P(\tau_{1/2}) > P(\tau_3)$. Task τ_4 moved to an interrupt routine.

Tasks communication:

- Global variable for the current status of the device.
- Binary semaphore for controlling access to this variable.
- Integer semaphore for signaling that the MP3 decoder is ready to receive new data.

```
#include <rtos.h>
#include <rtos-semaphores.h>
#include "datastruct.h"

static volatile status  current_status;
static semaphore  status_sem, data_sem;

void task1_2(void)  /* Keyboard and screen management */
{
  keyboard_state k;
  status         s;

  k = !! read_keyboard_state;

  !! compute new status s from current_status and k

  wait(status_sem);
  current_status = s;
  signal(status_sem);

  !! update screen
}
```

```
void task3(void)  /* Audio data sending */
{
  status s;

  for (;;)
    {
      wait(status_sem);
      s = current_status;
      signal(status_sem);

      !! compute data (using value of s)

      wait(data_sem);

      !! send data to MP3 decoder
    }
}

interrupt void task4(void)  /* Decoder is ready to receive */
{
  signal(data_sem);
}
```

```
void main(void)
{
  !! initialize OS
  !! initialize data structures

  create_periodic_task(task1_2, 40, 2);
  create_one_shot_task(task3, 1);

  status_sem = create_binary_semaphore(1);
  data_sem = create_int_semaphore(0);

  enable();  /* User-programmed interrupts */

  !! start tasks sequencing
}
```

Problem 3

An homemade digital oscilloscope has two analog input channels, four digital logic input channels, a LCD screen for displaying signals, a serial connection for sending data to a computer, and some control buttons. There is one button for switching between digital and analog modes, and another one for choosing between displaying the current signals on the screen, or sending these signals to the computer via the serial connection. By pressing two additional buttons, one can also modify the current voltage and time scales.

- The microcontroller is only able to perform a single A/D conversion at a given time. (In order to acquire both analog input channels, it is thus necessary to sample them one after the other.) Such a conversion takes at least 12 μs, and triggers an interrupt upon completion.
- In order to sample the digital inputs, the microcontroller reads the values on the corresponding pins every 10 μs.
- Processing acquired data before displaying it needs at most 4 ms of CPU time.
- The screen contents have to be refreshed at least 20 times per second, and each refresh takes 3 ms.
- Sending acquired data via the serial connection takes 5 ms and must be done 20 times per second.
- Buttons are checked at least 25 times per second.

List of tasks:

- τ_1: A/D conversion on one channel: period $\geq$ 12 μs.
- τ_2: Digital input sampling: period = 10 μs, exec. time = ε.
- τ_3: Processing data: same rate as τ_4, exec. time = 4 ms.
- τ_4: Screen refresh: period $\leq$ 50 ms, exec. time = 3 ms.
- τ_5: Serial communication: period = 50 ms, exec. time = 5 ms.
- τ_6: Checking buttons: period $\leq$ 40 ms, exec. time = ε.

Notes:

- τ_3, τ_4 and τ_5 can be grouped into a single task run every 50 ms.
- At any time, only one of τ_1 or τ_2 should be operating.
- τ_1 is performed by a peripheral, and managed by an interrupt routine.
- τ_2 and τ_6 can be implemented by interrupt routines.

Is preemption needed?

No.

Are interrupts needed?

Yes.

Is static processor assignment OK?

Yes!

$\rightarrow$ Round-Robin with Interrupts.

Task/task and task/interrupt routine communication:

Global variables:

- Flags for digital/analog and screen/remote modes.
- Flag for 50 ms timer expiration.
- Structure for raw and processed data, with separate fields for digital and analog samples.
- Structure for buttons input.

```
#include "types.h"
#include "datastruct.h"

volatile bool  mode_is_digital = 1, mode_is_remote = 0;
volatile scales  scaling_info;
volatile bool  timer1_ready = 0;
volatile samples_data  raw_data;
         samples_data  processed_data;
volatile buttons_state  buttons;

interrupt void timer1(void) /* 50 ms timer */
{
  timer1_ready = 1;
}

interrupt void timer2(void)  /* Read buttons */
{
  !! Read keyboard status and update global variable buttons
}

interrupt void task1(void)  /* End of A/D conversion */
{
  !! Fetch conversion result and append it to raw_data.analog
  !! Start next A/D conversion
}
```

```
interrupt void task2(void) /* Digital input sampling */
{
  digital_data  d;

  d = read_digital_inputs();

  data_append(raw_data.digital, d);
}

void task3(void)  /* Data processing */
{
  samples_data  d;

  disable();
  d = raw_data;
  enable();

  !! Process d and store the result in processed_data
}
```

```
void task4(void)  /* Screen refresh */
{
  !! Display the contents of processed_data on the screen,
  !! taking current value of mode_is_remote and
  !! scaling_info into account.
}

void task5(void)  /* Serial communication */
{
  !! Sends the contents of processed_data over the serial
  !! line.
}

void task6(void)  /* React to buttons */
{
  buttons_state s;

  disable();
  s = buttons;
  buttons.new_keypress = KEY_NONE;
  enable();
```

```
  switch (s.new_keypress)
    {
    case KEY_DIGITAL_OR_ANALOG:
      mode_is_digital = !mode_is_digital;
      break;

    case KEY_SCREEN_OR_REMOTE:
      mode_is_remote = !mode_is_remote;
      break;

    case KEY_SCALING:
      !! Modify scaling_info
      break;

    case KEY_NONE:
    default:
      break;
    }
}
```

```
 void main(void)
{
  !! Initialize global data structures

  !! Configure A/D: 12 µs sampling period, end-of-conversion
     interrupt (which calls task1())

  !! Configure timer1 interrupt (which calls timer1()),
     with period : 50 ms

  !! Configure timer2 interrupt (which calls timer2()),
     with period = 40 ms

  !! Configure timer3 interrupt (which calls task2()),
     with period = 10 µs and high priority.

  enable();  /* Global interrupts */
```

```
for (;;)
  {
    if (mode_is_digital)
      {
        if (!! A/D enabled)
          !! Disable A/D conversion

        !! Enable timer3 interrupt
      }
    else
      {
        if (!! A/D disabled or inactive)
          !! Enable and start A/D conversion

       !! Disable timer3 interrupt
      }
```

```
    if (timer1_ready)
      {
        timer1_ready = 0;
        task3();
        task4();

        if (mode_is_remote)
          task_5();
      }

    task_6();
  }
}
```