Task Scheduler

June 9th, 2020

This is a graded exercise. For further information regarding the evaluation criteria read the document *GradedExercises.pdf* carefully. The submission deadline is 2020-06-18 17:00 (CEST). The interviews will be held on Tuesday after submission, on 2020-06-23, during the exercise session. Make sure you join the exercise session you are assigned to in StudIP. Both team members must be present with camera and microphone.

Your solution has to be committed and pushed *to the master branch* of your team's Git repository by the specified time. We expect the following directories/files to be present in your repository root directory.

- `lib/ses` (dir; contains your current ses library's source/header files. We will grade all libraries written so far (adc, button, led, scheduler and timer))
- `task_4-2` (dir; containing scheduler test code source/header files)
- `task_4-3` (dir; optional; if you decide to do the challenge)

All functions declared in the header files within the *lib* folder should be implemented in the corresponding source file. Don't implement those functions in the main file. Make sure you adhere to the C programming rules specified in the appendix of the lecture. Coding style and documentation is as important as functionality.

Ignoring the directory structure or the coding style rules will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present.

If you want to earn the additional three bonus points for the challenge, you only have to do the challenge on this exercise sheet. The challenge tasks from previous sheets are not graded.

Task 4.1 : Task Scheduler

In the previous exercise sheet you have implemented a hardware timer. If we needed more timers, we could use more hardware timers. However, the number of hardware timers is limited, i.e., there are only six timers available in the ATmega128RFA1. Moreover, executing a time-consuming function inside an interrupt service routine is disadvantageous, since it blocks other interrupts. Hence, this should be avoided! A task scheduler can be used to overcome these problems. A scheduler needs only one hardware timer, but allows the execution of many tasks[1] after a given time period in a synchronous context (not inside of the interrupt service routine). In order to provide a synchronous execution of tasks, the scheduler periodically polls for executable tasks inside its `scheduler_run()` function.

Put the files *ses_scheduler.h*, *ses_scheduler.c* (from the provided ZIP file) into your ses lib folder. In this task, you will implement the function skeletons. The function `scheduler_init()` initializes the scheduler and Timer2 (*ses_timer.h*, *ses_timer.c*). All tasks are represented by a data structure as shown in Listing 1.

Each task is described by a function pointer to the function to execute (`taskDescriptor.task`). A task is scheduled for execution after a fixed time period given by `taskDescriptor.expire`. The scheduler also uses this variable to count down the milliseconds until the task should be executed. Tasks can be scheduled for single execution (`taskDescriptor.period`==0) or periodic execution (`taskDescriptor.period`>0). "Single execution tasks" are removed after execution; periodic tasks remain in the task list and are repeated depending on their period.

[1] In this context, a task is a function, which terminates/returns after some time by itself

To schedule a task, you have to set these parameters in the task descriptor and call the `scheduler_add()` function. Provide a pointer to the `taskDescriptor` as a call parameter. Additionally, the function to be executed takes a `void *` parameter to enable the passing of parameters to a task (`taskDescriptor.param`). Added tasks can later be removed from the scheduler with the function `scheduler_remove()`.

Listing 1: `taskDescriptor` data structure

```
/** type of function pointer for tasks */
typedef void ( * task_t)(void *);

/** Task data structure */
typedef struct taskDescriptor_s {
    task_t task;          ///< function pointer to call
    void * param;        ///< pointer, which is passed to task when executed
    uint16_t expire;      ///< time offset in ms, after which to call the task
    uint16_t period;      ///< period of the timer after firing; 0 means exec once
    uint8_t execute:1;    ///< for internal use
    uint8_t unused:7;     ///< unused
    struct taskDescriptor_s * next; ///< pointer to next taskDescriptor, internal use
} taskDescriptor;
```

Within the scheduler, the scheduled tasks should be organized as a singly linked list, which is initially empty:

```
static taskDescriptor* taskList = NULL;
```

The scheduler does not reserve memory for the task descriptors. Instead, the module using the scheduler has to provide the memory to store the `taskDescriptor`. Thereby, the scheduler is not restricted to a fixed number of tasks. Hence, you need to declare the task descriptor variable in your main program module and pass a pointer to this variable to the `scheduler_add()` function.

The callback for the Timer2 interrupt is used to update the scheduler every 1 ms by decreasing the expiry time of all tasks by 1 ms and mark expired tasks for execution. Additionally, the expiration time of periodic tasks should be reset here to the period.

The function `scheduler_run()` executes the scheduler in a superloop (`while(1)`) and is usually called from the `main()` function. It executes the next task marked for execution (if any), resets its execute flag and/or removes it from the list, if it is non-periodic. Be careful to handle task execution, adding, removing and resetting correctly!

Note that any interrupt service routine that calls a scheduler function may interleave with the execution of a scheduler function from the main-loop, i.e., from a task. This can lead to an inconsistency of memory if shared variables are accessed, which can cause unexpected behavior. Therefore, the access to the list of `taskDescriptor`s has to be restricted by disabling the interrupts in critical sections. There are some macros defined in `util/atomic.h` of the AVR libraries which can be used, e.g.:

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    // this block is executed atomically
}
```

✎ Read the comments in the source code carefully.

⚠ With the given structure, adding or removing the same `taskDescriptor` twice may lead to unwanted behavior. Make sure your code prevents this situation!

Task 4.2 : Using the Scheduler

In this task, you will use your scheduler and implement a program which runs different tasks in parallel.

Create a new project `task_4-2` in your workspace as described in task 2.2. Then implement the following functionality in the `main.c` file by **adding** or **removing** tasks to the scheduler:

- Implement a task to toggle an LED and use the parameter to select the color, e.g. by an `enum`. Use it to toggle the green led with a frequency of 0.5 Hz.
- Instead of using timer 1, use the scheduler for debouncing the buttons by calling `button_checkState` as task every 5 ms. Don't change the library code for the button.
- When pressing the joystick button, turn on the yellow LED. Turn it off when pressing the joystick button again or after 5 seconds, whatever comes first.
- Implement a stop watch, which starts and stops when pressing the rotary button. Use the LCD to show the current stop watch time in seconds and tenths of seconds. You do not need to implement a reset of the stop watch (you may use the reset button for this purpose).

Make use of the library created in previous exercises (LCD, LED, and Buttons).

Make sure, that the `taskDescriptor`s you use are not local variables, which run out of scope after scheduling – this is a sure means of producing undefined behavior!

Do not use `_delay_ms()` (or **any** other busy waiting) for waiting multiple milliseconds!

Task 4.3 : Preemptive Multitasking (Challenge)

In this task you will implement a preemptive multitasking execution model. The idea is that a number of predefined tasks are passed to the scheduler at the beginning. These tasks are running infinitely, and should be scheduled using a round-robin strategy with a time slot length of 1 ms. At the beginning, you have to initialize a separate stack for each task and run the first task. Note that you can change the stack pointer of the AVR MCU. On a timer interrupt (each 1 ms), you have to store the context of the current task in its own stack and to restore the context of the next task.

The example in Listing 2 shows the intended use of the preemptive multitasking scheduler. Your task is to provide the library required to run this program:

The following hints may help you:

- Put the implementation of the preemptive scheduler into a separate project `task_4-3`, it won't be re-used in coming lab classes.
- Make yourself familiar with preemptive multitasking, round-robin scheduling, context switches, and the way the MCU executes a program (stack, stack pointer, status register, etc.).
- The context switch has to be done using inline assembler, make yourself familiar with the concept of passing C variables to assembler programs.
- Read the file *Multitasking_on_an_AVR.pdf* (StudIP), this essentially contains everything you have to know. Also have a look into the lecture slides again, they also contain useful hits.
- After compiling your code, take a look at the assembler code as described in sheet 1 (task 2). Also check whether the compiler adds unintentional code. Inspecting the assembler code is also a good way to identify potential problems.
- The source file *challenge.c* (in exercise ZIP file) contains useful code snippets.

Listing 2: Intended use of the preemptive scheduler

```c
#include "pscheduler.h"

void taskA (void) {
    ...
    while (1) { ... }
}

void taskB (void) {
    ...
    while (1) { ... }
}

void taskC (void) {
    ...
    while (1) { ... }
}

task_t taskList[] = {taskA, taskB, taskC};
uint8_t numTasks = 3;

int main(void) {
  pscheduler_run(taskList, numTasks);
  return 0;
}
```