Task Scheduler

May 21st, 2025

Grading

This is a graded exercise. Read the document *bonus_points.pdf* carefully to learn about the evaluation criteria. The submission deadline is 2025-06-15 23:59 (CET); we will only consider solutions uploaded to the main branch of your team's Git repository. We hold the interviews during the regular lab hours in the week following the submission; i. e., on 2025-06-17 (group A) or 2025-06-19 (group B). **Both team members must be present**.
We expect the following directories/files to be in your repository's root directory.

- `lib/ses` — directory containing 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` — directory containing scheduler test code source/header files.

All functions declared in the header files within the *lib* folder must be implemented in the corresponding source file. Don't implement those functions elsewhere (e. g., in *main.c*). Ensure you adhere to the C programming rules in the *ppi-handout* slide deck. Coding style and documentation are 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 in Gitlab**.
Exercise 4.3 is optional and is not graded. However, it is an excellent opportunity to deepen your understanding of multitasking on a single CPU.

Task 4.1: Task Scheduler

In exercise 3 you learned how to use a hardware timer. While multiple hardware timers are available to serve different tasks, there are several reasons to prefer a software timer built on top of a single hardware timer:

- The number of hardware timers is limited and might not be enough for your application(e. g., there are only four timers available on the ATmega32U4).
- Executing a processing-intensive function inside an interrupt service routine is disadvantageous since it blocks other interrupts. Hence, this should be avoided!
- Setting up and similarly using several hardware timers likely leads to redundant code.

A task scheduler can overcome these problems requiring only a single hardware timer. At the same time, it allows the execution of many (periodic) tasks[1] in a synchronous context (not inside of the interrupt service routine). To provide a synchronous execution of tasks, the scheduler periodically polls for executable tasks inside its function `scheduler_run()`.

[1] In this context, a task is a function, which terminates/returns eventually by itself.(Cooperative multitasking)

Put the files *ses_scheduler.h* and *ses_scheduler.c* (from the provided ZIP file) into your SES library folder. The following sections guide you through the implementation of the provided function skeletons.

The function `scheduler_init()` initializes the scheduler and Timer0 (*ses_timer.h*, *ses_timer.c*). The scheduler keeps track of all tasks via the data structure `task_descriptor_t`:

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

/** Task data structure */
typedef struct task_descriptor_s task_descriptor_t;
struct task_descriptor_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 reserved:7;       // reserved
  task_descriptor_t * next; // pointer to next task, internal use
};
```

- A task is marked for execution (execute = 1) after the time stored in `expire` has expired. All times are in milliseconds.
- Tasks can be scheduled for single (*one-shot* or *non-periodic*) execution (`period==0` or periodic execution (`period`>0). One-shot tasks need to be removed after execution; periodic tasks remain in the task list but their expiration time is reset to the period (`expire = period`).

To schedule a task, you have to set the related fields in the task descriptor and call the function `scheduler_add` subsequently. Provide a pointer to `task_descriptor_t` as parameter. Please note that the function to be executed takes a parameter of type `void *`, which enables passing arbitrary parameters to a task (`param` in `task_descriptor_t`). Existing tasks are removed from the scheduler with the function `scheduler_remove()`. This concept of task handling has been introduced already in the lecture.

The scheduler should organize the task descriptors in a singly linked list, which is initially empty:

```c
static task_descriptor_t * taskList = NULL;
```

Dynamic vs. Static Memory Allocation

Remember that the use of dynamic memory allocation (i. e., `malloc()`) is discouraged in low-resource embedded systems. The module using the scheduler—your main module—has to statically allocate the memory to store the `task_descriptor_t` and pass a pointer to this memory to the `scheduler_add()` function. Thereby, the scheduler can handle an arbitrary number of tasks.

The callback of `Timer0` is used to update the scheduler every 1 ms by decreasing the expiration time of all tasks by 1 ms and mark expired tasks for execution. Additionally, the expiration time of periodic tasks should be reset to the period at this point.

The function `scheduler_run()` executes the scheduler in a infinite loop (`while(1)`) and is usually called from the `main()` function. If a task is marked for execution, it executes it and resets its execute flag. Non-periodic tasks are removed from the list. Be careful to handle task execution, adding, removing, and resetting in the correct order. Interrupts should be blocked for a minimal duration. The comments in the provided templates give additional hints on the expected implementation.

Asynchronity and Race Conditions

Be aware that during the execution of the scheduler functions, they might be interrupted by ISRs that again call scheduler functions. This can lead to memory inconsistencies—if shared variables are accessed—which can cause unexpected behavior. Therefore, access to `taskList` has to be restricted by disabling the interrupts in critical sections (mutual exclusion). There are 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
}
```

Defensive Programming

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

Task 4.2 : Using the Scheduler

This task requires you to use your scheduler for implementing a program that runs multiple periodic tasks with different periods.

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

- Implement a task that toggles an LED and uses 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.
- Use the scheduler instead of Timer1 for debouncing the buttons. Create a task that executes the function `button_checkState()` every 5 ms. You have to add the `button_checkState()` prototype to *ses_button.h*, so that you can access this function from *main.c*.
- When pressing the push button, turn on the yellow LED. Turn it off when pressing the push button again, or after 5 s, whatever comes first.
- Implement a stopwatch, which starts and stops when pressing the rotary button. Use the display to show the current stopwatch time in seconds and tenths of seconds. You do not need to implement a reset of the stopwatch (you may use the reset button for this purpose, however).

Code Reuse

Make use of the libraries created in previous exercises (timer, display, LED, and buttons). Redundant code will lead to point deductions.

Variable Scope and Busy Waiting

Ensure that the task descriptors are not local variables that may run out of scope after adding them to the scheduler. 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 (Optional Challenge)

In this task, you implement a *preemptive* multitasking execution model. Several predefined tasks (functions) are passed to the scheduler at the beginning; they are running infinitely, and they should be scheduled using a round-robin strategy with a time slot length of 1 ms. On a timer interrupt (every 1 ms), the currently running task is preempted, and the next task is dispatched in the ISR (interrupt service routine). This implies that you have to store the context of the current task on its own stack and restore the context of the next task. The task context comprises

- the program counter (PC, which is automatically pushed onto the stack when entering the timer ISR),
- the general purpose registers (R0–R31),
- the status register (SREG), and
- the stack pointer (SP).

Stack Pointer

The lecture shows that the stack pointer cannot be stored on the stack. It has to be stored in a separate (pointer) variable.

The function `scheduler_run()` creates a task context (type `context_t`) for every task and initializes these contexts. For this purpose, you must put a task's initial context onto its stack (including the program counter), and set the stack pointer correctly. The function then starts the timer, picks the first task, and restores its context.

The following example shows an exemplary use of the preemptive multitasking scheduler. Your task is to provide the library required to run this program:

```c
#include "pscheduler.h"
// pscheduler.h defines:
// - typedef void (* task_t)(void);
// - void pscheduler_run(const task_t * taskArray, uint8_t len);

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

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

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

// Attention: task_t for the preemptive scheduler is different from
// task_t from the previous exercises.
static const task_t TASKS[] = {taskA, taskB, taskC};
#define NUM_TASKS (sizeof(TASKS)/sizeof(TASKS[0]))

int main(void) {
  ...
  pscheduler_run(TASKS, NUM_TASKS);
  return 0;
}
```

The following hints may help you:

- Put the implementation of the preemptive scheduler into a separate project `task_4-3`; it will not be re-used in coming lab classes.
- Make yourself familiar with preemptive multitasking, round-robin scheduling, context switches, and how the MCU executes a program (stack, stack pointer, status register, etc.).
- The context switch must be done using inline assembly; familiarize yourself with passing C variables to assembler programs.
- Read the file *Multitasking-on-an-AVR.pdf*; essentially, it contains everything you have to know. In addition, review the lecture slides, as they contain useful hints.
- After compiling your code, inspect the created assembly code as described in sheet 1 (task 2). Check whether the compiler adds undesirable code. Inspecting the assembly code is an excellent way to identify potential problems.
- The source file *challenge.c* (in exercise ZIP file) contains helpful code snippets.