Task Scheduler

May 31st, 2016

This is a graded exercise. For further information regarding the evaluation criteria read the document *GradedExercises.pdf* carefully. The submission deadline is 2016-06-12 23:59 (CEST). The interviews will be held on Tuesday after submission, on 2016-06-14.

Your solution has to be committed to your team folder in the SVN repository by the specified time. We expect the following directories/files to be present in your root directory at https://svn.ti5.tu-harburg.de/courses/ses/2016/teamX/ where X is your team's number.

- `ses` (dir; contains your current ses library's source/header files)
- `task_5_2` (dir; containing scheduler test code source/header files)
- `task_5_3` (dir; optional; if you decide to do the challenge)
- `readme.txt` (file; contains <*Name1*> <*Matr. number 1*>, <*Name2*> <*Matr. number*>)

Ignoring this directory structure will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present (you do NOT need to check in the compiled libraries!).

Keep the structure of those directories flat, i.e., do not use subdirectories there!

Task 5.1 : Task Scheduler

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;
```

So far you have implemented a hardware timer in the previous exercise sheet. If we need more timers, we could use more hardware timers. But the number of hardware timers are limited to six for the ATmega128RFA1. Moreover, when using a time-consuming function inside an interrupt service routine, other interrupts are blocked. This should be avoided! To overcome the mentioned problems, a task scheduler can be used. We need only one hardware timer for the scheduler, which allows the execution of tasks[1] after a given time period in a synchronous context (not in 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.

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

Put the files *ses_scheduler.h*, *ses_scheduler.c* (from the provided ZIP file) into your ses library project. In this task, you have to 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 function is scheduled for execution after a fixed time period by `taskDescriptor.expire`, which is also used by the scheduler as an individual counter, counting down the time (in milliseconds) till execution. 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 scheduler and are repeated depending on their period. To schedule a task, those parameters have to be set, and a pointer to the `taskDescriptor` has to be provided to `scheduler_add()`. 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()`.

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

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

Clients of the task scheduler have to provide the memory to store the `taskDescriptor`. Thereby, the scheduler is not restricted to a certain number of tasks.

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 period of periodic tasks should be reset here.

The function `scheduler_run` executes the scheduler in a superloop (`while(1)`) and is commonly 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 (a crash is possible!) if shared variables are accessed. 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, it can lead to unwanted behavior to schedule the same `taskDescriptor` twice. Make sure your code prevents this situation!

Task 5.2 : Using the Scheduler

In this exercise you have to use your scheduler and implement a program which runs different tasks in parallel.

Create a new project with a file *scheduler_test.c* containing the `main` function. Then implement the following functionality:

- Toggle the green led with a frequency of $0.5\,s^{-1}$.
- Instead of using timer 1, use the scheduler for debouncing the buttons by calling `button_checkState` as task every 5 ms.
- When pressing the joystick button, turn on 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 `taskDescriptors` 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()` for waiting multiple milliseconds!

Task 5.3 : Preemptive Multitasking (Challenge)

In this task you have to 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, which have to run infinitely, should be scheduled using round-robin 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 following listing shows the intended usage of the preemptive multitasking scheduler. Provide the needed library for running this program:

```
#include "pscheduler.h"

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

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

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

```
task_t taskList[] = {taskA, taskB, taskC};
int main(void) {
  pscheduler_run(taskList,3);
  return 0;
}
```

The following hints may help you:

- Put the implementation of the preemptive scheduler into a separate project `task_5_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 actually contains everything you have to know. Also have a look into the lecture slides again, they also contain some useful hits.
- After compiling your code, open the **.lss* file which contains the assembler code. Use release mode for compiling since this makes the code more readable. Also check whether the compiler adds unintentional code. The *.lss* file is also a good place to identify potential problems, because the debugging possibilities are rather restricted.
- The source file *challenge.c* (in exercise ZIP file) might contain some useful code snippets.