Alarm Clock

July 14th, 2020

This is a graded exercise. For further information regarding the evaluation criteria read the document *gradedExercises.pdf* carefully. The submission deadline is 2020-07-23 17:00 (CEST). The interviews will be held on 2020-07-28.

Your solution has to be committed to your team folder in the Git repository by the specified time. We expect the following directories/files to be present in you root directory:

- `lib/ses/` (contains your current ses library source/header files, including updated scheduler and the rotary implementation in the case of the challenge)
- `task_6` (containing source/header files for task 6.3/6.4 (alarmClock) and the challenge)
- `fsm.<ext>` (a file containing a state chart of your alarm clock's state machine (see Task 6.2))

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 libraries!).

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

Task 6.1 : Creating a Clock

In this exercise you have to implement a simplistic alarm clock.

The first task is to create a 24-hour Clock with a resolution of 1 millisecond. For this task you have to extend your scheduler (*ses_scheduler.c*, *ses_scheduler.h*). Since the scheduler already uses a timer callback with a resolution of 1 millisecond, you can extend its ISR. Define a private time-keeping variable of type `systemTime_t` (which corresponds to an unsigned 32-bit integer, see below) for storing the current time and increment it every millisecond.

For orientation, calculate the time until the timer overflows. However, you do not have to handle this event.

For reading or writing the time-keeping value, you have to add these two functions:

```
typedef uint32_t systemTime_t;
systemTime_t scheduler_getTime();
void scheduler_setTime(systemTime_t time);
```

For easier handling in your alarm clock, you should write some wrapper functions calculating the time in a more human-friendly way, using the structure `time_t`:

```
struct time_t {
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
    uint16_t milli;
};
```

Task 6.2 : Finite-State Machine Model

The main task of this exercise sheet is the implementation of a simple but functional alarm clock. Before implementing the code, draw a **UML statechart** matching the description in the following paragraph. This drawing is part of the graded exercise and has to be present as digital file in your submission directory (.png, .pdf, .jpeg; you may also take a photo of a hand-drawn image as long as we can clearly read everything).

The following paragraph gives a description of the alarm clock in natural language:

When the microcontroller is powered up, the actual time is not known. The user has to set the time manually using the buttons. At this stage, the display shows an uninitialized clock using the format HH:MM. The second display line may show a request for the user to set time. First, the hour has to be set by repeatedly pressing the *Rotary Button*. After pressing the *Joystick Button*, the minutes have to be set via the *Rotary Button*. Pressing the *Joystick Button* again updates the system time and starts the clock. In this normal operation mode the time is shown in the format HH:MM:SS. In this state, the user can press the *Rotary Button* to enable the alarm or to disable the alarm. If the *Joystick Button* is pressed, the alarm time can be set. In this mode line 1 shows the alarm time instead of the current system time using the format HH:MM. If the alarm is enabled and the actual time matches the alarm time, the red LED shall toggle with 4 Hz. The alarm shall stop, if any button is pressed or 5 seconds have passed. The alarm must only be triggered if the clock is in its normal operating mode, i.e., it must not be triggered while the alarm time is being modified.

In addition, the LEDs are used for the following functionality:

- The green LED blinks synchronously with the counter of the seconds.
- The yellow LED is on, if and only if the alarm is enabled.
- The red LED is flashing with 4 Hz during alarm, and it is off otherwise.

The next task provides additional instructions on detail aspects of the state machine. You may want to read it first.

Task 6.3 : Implementing the Alarm Clock

Implement the described alarm clock using the state machine designed in Task 6.2! Implement the FSM as an exclusively **event-based** (i.e., not synchronous) one!

The FSM shall be implemented with pointers to functions. A state is represented by a pointer to an event handler (i.e., a function), which takes a pointer to the FSM itself and the event that occurred.

```
typedef struct fsm_s Fsm;     //< typedef for alarm clock state machine
typedef struct event_s Event; //< event type for alarm clock fsm

/** return values */
enum {
    RET_HANDLED,    //< event was handled
    RET_IGNORED,    //< event was ignored; not used in this implementation
    RET_TRANSITION  //< event was handled and a state transition occurred
};
typedef uint8_t fsmReturnStatus; //< typedef to be used with above enum

/** typedef for state event handler functions */
typedef fsmReturnStatus (*State)(Fsm *, const Event *);
```

The return value is used by the FSM “core” to decide about entry and exit actions, but more on that later.

We define the struct `Fsm` with additional attributes besides `state`. The member `isAlarmEnabled` should be used to store the current alarm state (on/off), and `timeSet` should be used to store the alarm-time or setup-time before normal clock operation.

```
struct fsm_s {
    State state;              //< current state, pointer to event handler
    bool isAlarmEnabled;      //< flag for the alarm status
    struct time_t timeSet;    //< multi-purpose var for system time and alarm time
};
```

The `Event` struct contains possible events that may trigger state changes. Using `signal`, the type of the event can be determined by the event handler (e.g., *Joystick Button* pressed).

```
struct event_s {
    uint8_t signal;     //< identifies the type of event
};
```

It is a probably a good idea to list the possible events within an enum.

For this exercise, we use the functions presented in the lecture, but extend `fsm_dispatch()` to provide entry and exit actions as well, because these are useful for the alarm clock. Here, you can also see how the return status of the event handler is used to dispatch entry and exit actions.

```
/* dispatches events to state machine, called in application*/
inline static void fsm_dispatch(Fsm * fsm, const Event * event) {
    static Event entryEvent = {.signal = ENTRY};
    static Event exitEvent = {.signal = EXIT};
    State s = fsm->state;
    fsmReturnStatus r = fsm->state(fsm, event);
    if (r == RET_TRANSITION) {
        s(fsm, &exitEvent);           //< call exit action of last state
        fsm->state(fsm, &entryEvent); //< call entry action of new state
    }
}
/* sets and calls initial state of state machine */
inline static void fsm_init(Fsm * fsm, State init) {
    //... other initialization
    Event entryEvent = {.signal = ENTRY};
    fsm->state = init;
    fsm->state(fsm, &entryEvent);
}
```

These functions can be used to control the state machine, e.g., to dispatch an event when a button was pressed. The following code snippet illustrates the intended usage:

```
#define TRANSITION(newState) (fsm->state = newState, RET_TRANSITION)

static void joystickPressedDispatch(void * param) {
    Event e = {.signal = JOYSTICK_PRESSED};
    fsm_dispatch(&theFsm, &e);
}

fsmReturnStatus running(Fsm * fsm, const Event * event) {
    switch(event->signal) {
        //... handling of other events
        case JOYSTICK_PRESSED:
            return TRANSITION(setHourAlarm);
        default:
            return RET_IGNORED;
    }
}
```

The shown `TRANSITION` macro can be used exclusively to change states to make sure that state changes are always the last operations within an event handler.

Be aware that operations regarding the state machine need a “run-to-completion” semantic, i.e., must not be interrupted by another operation on the FSM. One way to accomplish this, is to run the FSM completely within task context. Another would be to use atomic blocks; however, considering that the alarm clock involves several LCD-related operations, this approach should be discarded.

With the given information, it should be possible to implement the alarm clock in a very concise and elegant way. Below you will find additional information (and, of course, the challenge task).

The shown code snippets can easily be extended to a more generic event processor, which could be used to implement arbitrary state machines. It is inspired heavily by the material presented in “Practical UML Statecharts — Event-Driven Programming for Embedded Systems” by Miro Samek. If you are interested, you can find much more information there! Note also that usually such an event processor would use an event-queue to store incoming events. With our recommendation to execute the FSM in task context, we instead use the scheduler to queue the events for our alarm clock which is probably not the most clean, but – within the scope of the exercise – a very practical and functional solution.

Task 6.4 : Rotary Encoder (Challenge)

To improve the usability of setting the time, the rotary encoder shall be used in addition to the push button; however, the previous behavior is to be preserved: Pressing the rotary or turning it clockwise increases the value, while turning it counterclockwise decreases the value. The rotary encoder has two inputs A (connected to `PORTB5`) and B (`PORTG2`), which can be *high* or *low* (open/closed switch to GND=C). The encoding is displayed in Fig. 1, also showing the mechanical detent positions. Implement the functionality to detect a left or right turn of the incremental rotary encoder within a new library module `ses_rotary`. Use callbacks to increment or decrement the current hour or minute. Consider the given interface.

```
typedef void (*pTypeRotaryCallback)();
void rotary_init();
void rotary_setClockwiseCallback(pTypeRotaryCallback);
void rotary_setCounterClockwiseCallback(pTypeRotaryCallback);
```

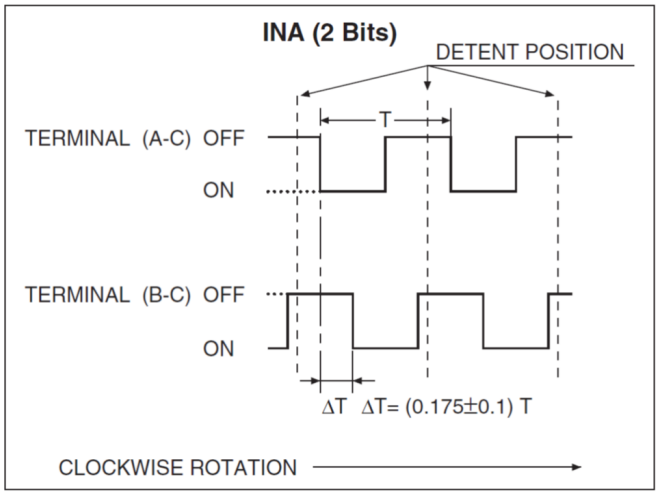

Figure 1: Sequence of rotary inputs for clockwise turning.
Credit: Noble Electronics Co., Ltd., Datasheet Rotary Encoders RE Series.

To get better understanding of the rotary button, plot the samples of both inputs on the LCD. You may use the given function below, which plots the current samples once the rotary is turned until the display is full. Do not forget to initialize the inputs!

The rotary requires careful debouncing. Use a sampling approach similar to the button debouncing. Yet, occasionally skipping dents at fast turning is acceptable.

```
/*
* Callback-procedure to plot the samples of the rotary pins on the LCD after first change
*/
void check_rotary() {
    static uint8_t p = 0;
    static bool sampling = false;

    bool a = PIN_REGISTER(A_ROTARY_PORT) & (1 << A_ROTARY_PIN);
    bool b = PIN_REGISTER(B_ROTARY_PORT) & (1 << B_ROTARY_PIN);

    if (a != b) {
        sampling = true;
    }

    if (sampling && p < 122) {
        lcd_setPixel( (a) ? 0 : 1, p, true );
        lcd_setPixel( (b) ? 4 : 5, p, true );
        p++;
    }
}
```