Alarm Clock

June 25th, 2025

In this sheet you implement a user configurable alarm clock. First, you extend your scheduler to track the current time. Then, Task 6.2 asks you to draw a finite state machine (FSM) diagram representing the alarm clock's state according to user inputs. In Task 6.3, you implement the FSM. Finally, in the optional Task 6.4, you implement a driver for the rotary encoder to simplify user interaction.

Bonus Points

This exercise is graded. Please read the document *bonus_points.pdf* for detailed evaluation criteria.

- **Submission Deadline** 2025-07-13 23:59 (CET).
- **Interviews** during regular lab hours.
  - Group A: 2024-07-15
  - Group B: 2024-07-17
- Both team members must be present.

Commit your solution to the `main` branch of your team repository before the deadline. Ensure the following directories/files are present in the root directory:

- `lib/ses/` – contains your current ses library source/header files and includes the updated scheduler
- `task_6` – containing source/header files for Task 6.3/6.4 (alarm clock)
- `fsm.<ext>` – a file containing a state chart of your alarm clock's state machine (see Task 6.2)

Ignoring this directory structure results in point deductions. Ensure you include all **header** and **source** files necessary to build and link your solution. Keep the structure of those directories flat, i.e., do not use subdirectories there!

Task 6.1: Creating a Clock

Create a 24-hour clock with 1-millisecond resolution. Extend your scheduler's interface with two additional methods:

- `system_time_t scheduler_getTime(void)` returns the current system time represented as the number of milliseconds since midnight.
- `void scheduler_setTime(system_time_t time)` allows the user to set the current time.
- Create a type definition (`typedef`) for `system_time_t`. Calculate how long the timer can run with the selected type before overflowing.

Implement the new methods. Define a time-keeping variable of type `system_time_t` for storing the current time and increment it every millisecond, e. g., in `scheduler_update()`. The variable should only be accessible in the module scope.

To simplify working with the time in your alarm clock, write two wrapper functions that convert between `system_time_t` and a more human-readable time format using the structure `time_t` shown on the right.

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

Task 6.2 : Finite-State Machine Model

Draw a *UML statechart* for the alarm clock's functionality as described below. You can earn bonus points by submitting this statechart as a digital file (.png, .pdf, .jpeg). A clear photo of a hand-drawn image will also be accepted.

On power-up, the current time is uninitialized. 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 the time. The user must enter the time manually. First, increase the hour by repeatedly pressing the *Rotary Button* and confirm the setting with a *Push Button* press. Then, use the same mechanism to set the minutes. Once the minutes have been confirmed, the clock starts. In this normal operation mode, the display shows the time in the format `HH:MM:SS`.

The user can turn the alarm on or off by pressing the *Rotary Button*. The alarm time is set by pressing the *Push Button*. While setting the alarm time, line 1 of the display shows the alarm time instead of the current system time using the format `HH:MM`. Setting the alarm time works analogously to setting up the system time on start-up. The clock returns to normal operation mode after setting the alarm time. However, setting the alarm time does *not* activate the alarm.

If the alarm is enabled and the actual time matches the alarm time, toggle the red LED with 4 Hz. Any button press stops the alarm. If not stopped manually, the alarm automatically stops after 5 seconds. In either case, the alarm remains enabled. Ensure the alarm is only triggered if the clock is in its normal operating mode; i. e., it must not be triggered while the alarm time is modified.

In addition, use the LEDs as follows:

- The green LED blinks synchronously with the second counter.
- The yellow LED indicates if the alarm is enabled.
- The red LED is flashing with 4 Hz during an alarm, and it is off otherwise.

Task 6.3 : Implementing the Alarm Clock

In this task you implement the previously designed state machine. Make the FSM exclusively **event-based** (i.e., not synchronous)! Use function pointers to implement the FSM (as the lecture explains), where each state is represented by an *event handler* function. The function takes two arguments: a pointer to the FSM itself and the event that occurred. It returns a status code indicating whether the event was handled, ignored, or resulted in a state transition. Have a look at the lecture chapter *Finite-State Machines* for more details.

```c
typedef struct fsm_s fsm_t;     //< typedef for alarm clock state machine
typedef struct event_s event_t; //< event type for alarm clock fsm

/** return values */
enum return_values {
    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 enum return_values fsm_return_status_t; //< typedef return value

/** typedef for state event handler functions */
typedef fsm_return_status_t (*state_t)(fsm_t *, const event_t *);
```

The FSM uses the return value to decide about entry and exit actions.

Define the struct `fsm_s` (type `fsm_t`) with additional attributes besides `state`. Use the member `isAlarmEnabled` to store the current alarm status (on/off) and `timeSet` to temporarily store the alarm time or current time during setup.

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

The `event_s` struct (type `event_t`) contains events that may trigger state changes. Using the member `signal`, the event handler can determine the event type (e.g., *Push Button* pressed). List the possible events in an **`enum`**.

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

In addition to the state machine functions presented in the lecture (slide: *FSM Function Pointer Implementation*), the dispatch function provides additional *entry and exit* actions, which are helpful for the alarm clock implementation. In this context, *dispatching* means to send an event to the state machine for processing. The following snippet shows the implementation of `fsm_dispatch()` and demonstrates how the event handler's return values are used to dispatch entry and exit actions.

```c
/* dispatches events to state machine, called in application*/
inline static void fsm_dispatch(fsm_t * fsm, const event_t * event) {
    const event_t entryEvent = {.signal = ENTRY};
    const event_t exitEvent = {.signal = EXIT};
    state_t s = fsm->state;
    fsm_return_status_t 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
    }
}
```

The following function initializes the state machine.

```c
/* sets and calls initial state of state machine */
inline static void fsm_init(fsm_t * fsm, state_t init) {
    //... other initialization
    const event_t entryEvent = {.signal = ENTRY};
    fsm->state = init;
    fsm->state(fsm, &entryEvent);
}
```

The dispatch function notifies the state machine of an occurred event. The following code snippet illustrates how to handle a *Push Button* press. The function `state_normalOperation(...)` represents one state of the FSM, which defines all actions on incoming events in this particular state.

```c
static void pushbuttonCallback(void * param) {
    const event_t e = {.signal = PUSHBUTTON_PRESSED};
    fsm_dispatch(&theFsm, &e);
}

fsm_return_status_t state_normalOperation(fsm_t * fsm, const event_t * event) {
    switch(event->signal) {
        //... handling of other events
        case PUSHBUTTON_PRESSED:
            fsm->state = setHourAlarm;
            return RET_TRANSITION;
        default:
            return RET_IGNORED;
    }
}
```

Race Conditions

Operations regarding the state machine need to follow a *run-to-completion* semantic. This means that once an FSM operation starts, it must not be interrupted by another FSM operation. While atomic blocks can ensure this, the alarm clock involves several long-lasting display-related operations that make this approach impractical.
To handle this, avoid running FSM operations in an interrupt context. Instead, use the scheduler to defer calls to the dispatch function from interrupt context to task context. This ensures that FSM operations can be completed successfully without interruption.

The presented snippets can be extended to a generic event processor, implementing arbitrary state machines. It is heavily inspired by the material presented in “Practical UML Statecharts — Event-Driven Programming for Embedded Systems” by Miro Samek, which contains much more practical information! Usually, event processors use an event queue to store and process incoming events in the correct order. Instead, we use the scheduler to queue the events for our alarm clock, which is not the cleanest, but—within the scope of the exercise—a reasonable solution.

Task 6.4 : Rotary Encoder (Optional)

The current implementation of the time-setting with two buttons is tedious. The rotary encoder enables the user to increase or decrease the time by turning the knob of the rotary button. In this exercise, you implement a driver for the rotary encoder and extend the FSM to use the rotary encoder. Ensure you preserve the previous behavior: Pressing the rotary or turning it clockwise increases the value, while turning it counterclockwise decreases the value.

The rotary encoder has two output lines: A (connected to `PORTB7`) and B (`PORTD6`). Each line can either be *high* or *low* (open/closed switch to GND=C). Figure 1 displays the encoding and shows the mechanical detent positions. Extend the module `ses_button` to support the rotary encoder with callbacks for a left or right turn following the interface below. Utilize these callbacks to increment or decrement the current hour or minute in the state machine.

```c
    void button_setRotaryCWCallback(pButtonCallback callback);
    void button_setRotaryCCWCallback(pButtonCallback callback);
```

Debouncing

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

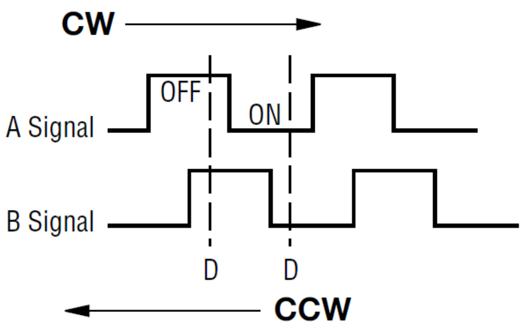

Figure 1: Sequence of rotary inputs for clockwise (CW) and counterclockwise (CCW) turning.
Credit: Bourns, Inc., Datasheet PEC11H Series - 11 mm Encoder w/High Detent Force

Understanding the Rotary Encoder

To better understand the rotary encoder, you can use the LEDs. Consider the provided function below to visualize the two inputs A and B. Do not forget to initialize the inputs and LEDs! If you rotate the encoder slowly, the LEDs show a different pattern for clockwise and counterclockwise rotations.

```c
int main(void) {
    // TODO: Init inputs and LEDs

    while(1) {
        if (BUTTON_ROTARY_A_PIN & (1 << BUTTON_ROTARY_A_BIT)){
            led_redOn();
        } else {
            led_redOff();
        }
        if (BUTTON_ROTARY_B_PIN & (1 << BUTTON_ROTARY_B_BIT)){
            led_greenOn();
        } else {
            led_greenOff();
        }
    }
    return 0;
}
```