Seatbelt Reminder System

A small-scale embedded project using state machine approach

lkmuk

started from 2020-07-10
last modified on 2020-07-12

This documentation discusses the use of state machine approach in embedded development, in particular the state machine modeling framework and its implementation in C. This is exemplified and demonstrated by the Seatbelt Reminder System prototype discussed in [3]. The contributions here are some modifications to the original solution and supplementing that pseudo-C code with a full working prototype system. The problem description and development settings for this project can be found in *readme* file and are generally not reproduced in this documentation.

Contents

1 High-level event-based modeling of the system behavior using state chart

This section discusses how to model the desired behavior of our Seatbelt Reminder System (as described in *readme* file) in an intuitive state chart[1]. In the next section (§2), we will convert the UML state chart *"back to"* the classic FSM for our particular target setup which can then be readily implemented in C.

Before that, let's underline the major differences between UML state chart and the conventional Finite State Machine (FSM) where the later was employed in [3].

state chart vs FSM UML state chart provides substantial enhancements to FSM that ultimately lead to much simpler more manageable state machine model.[2] Below are some examples of the extensions:

- the notion of *entry action*, *exit action*, or *during action* of a state
  (the advantages of *exit* and *entry* action will be elaborated in §1.4)
- notion of events for triggering state transitions. For our example, I decided to simply update the system periodically so the default event is *an* ISR arrival. Another event will be timeout event for not fastening the seatbelt in time. Events will be part of the subject of §1.5.
- hierarchical and/or concurrent state behavior which are integral to tackling the *state explosion* issue for complex reactive systems and improving modularity, hence reusability (some discussion on that in §3.2)

Of course, our standalone seatbelt reminder system by no means qualifies as a complex system so this section will not attempt to cover the concept of state composition with hierarchy and concurrency. Yet, the first two concepts are quite relevant for our example and therefore applied. Consequently, the state machine model given at the end of this section (Fig. 1) looks slightly different (to be precise, clearer) than the (finite) state machine presented in [3].

1.1 Identifying system I/O

The FSM model in [3] has the following three inputs and output:

- *a sensor that detects whether a person sits on the seat,*
- *a sensor that detects whether the person has fastened his/her seatbelt,*

[1] The transition precedence or *sequence* mentioned in §1.5.3 is not specified in the UML standard but it is widely adopted since it exploits the sequential execution nature of the implementation.

[2] The price we pay is some framework support and/ or some additional (compile time) conversion to the classic FSM format, either manually or automatically. The conversion concept is addressed in §2.

- *a timer input and*
- *a buzzer output.*

Since the FSM has to be executed or *updated* regularly anyway, possibly at fixed (nominal) rate (with some acceptable jitter), we can simply infer the elasped time from sitting without fastening the seatbelt using the tricks presented in the following subsection (§1.2), thereby dispensing the timer input.[3]

Moreover, for the demo, I prefer a simple red led to a buzzer. All these lead to the following **revised *system* I/O specification**, as illustrated in Fig 4.

- seat occupancy sensor (one might use this for other tasks...)
- seatbelt sensor
- red LED as a warning signal to the occupant and/ or driver.

1.2 Enumerating states & extended state variable(s)

Perhaps, one of the most critical aspect of employing state-machine based programming paradigms is the enumeration of some meaningful states. In many case, to prevent *state explosion*, we might additionally *extend* the qualitative nature of the state with some internal (quantitative) state variable(s). This comes at the cost of more complicated testing and can become less intuitive to interpret the state chart. In many cases we might also have concurrent states which will not cover here.

Fortunately with temporal event notion and the simplistic nature of our system, we *can* specify the behavior entirely based on a qualitative finite set of states where each state is sufficient to capture previous system operation history. They are:

`IDLE`: the initial state when there is simply no one sitting on the seat.

`SEAT_NO_BELT`: when the seat *starts* to be occupied but the seatbelt is still not on.

`SEAT_WITH_BELT`: when both the seat is occupied and the seat belt is on.

`WARNING`: when the seat is occupied for "too long" (say exceeding 6 consecutive seconds) without the seatbelt fastened.

1.3 Identify possible state transitions

This step concerns simply the question ***whether it is possible*** to transit from one state to another state. From our previous discussion, our state machine has four states, so we can simply examine the possibility of all $4^2 - 4 = 12$ cases (excluding self-transition because self-transition will not be of interest until §1.5). An initial analysis of our Seatbelt Reminder System yields the one-step reachability table (Table 1).

One might be tempted to skip this step and directly consider ***how or under what conditions*** a state will transit to another (which is the step described in §1.5. From my experience, it is *better* not to skip this binary one-step reachability check. As a side note, the textbook solution omits or misses one of the transitions (from state 1 to 0) which under certain scenario can lead to unexpected behavior — longer time allowance for buckling up before triggering a warning signal.

[3] In fact, the aforementioned temporal logic of state charts assumes certain (in-system) support for timing behavior!

Table 1: One-step reachability between states for our Seatbelt Reminder System

↓ current state \ next state →	0	1	2	3
0 (IDLE)	-	y	(y)	
1 (SEAT_NO_BELT)	y	-	y	y
2 (SEAT_WITH_BELT)	(y)	y	-	
3 (WARNING)	y		y	-

where "y" denotes a very plausible one-step transition;
"(y)" denotes a *Just-in-case*† backup transition;
blank entries correspond to *absolutely* impossible or unreasonable one-step transitions.
† *JIC in the sense that it might defy common sense (assuming integrity of the input data), e.g. leaving seat before releasing seatbelt as in one-step state transition: 2 to 0*

1.4 Identify state transition actions

In many case, a state can be *characterized* by what need(s) to be done as the (sub)system leaves from this state (regardless of next state) or enters into this state (regardless of previous state). These use cases lead naturally to the notion of *exit* and *entry actions* respectively. This notion allows us to **avoid repetition** for the said use cases, hence **improved maintainability** [2].

Our example turns out to fit into this category of use cases very well. Indeed, we can model all transition actions in our case as either entry or exit actions!

Entry and exit action for state *WARNING* It is obvious that we should turn ON the warning lamp as we enter the *WARNING* state whereas we should turn it OFF as we leave from it.

Entry action for state *SEAT_NO_BELT*??? We might consider start of timing as an entry action but this is actually unnecessary *at this stage*. We can leverage on the notion of timing events to emphasize what behavior we actually want, instead of implying how to implement this timing behavior.[4]

1.5 Specify transition conditions & priorities

1.5.1 Events and Guards

In UML state chart notation, a state transition *can*[5] only take place when the corresponding *event (class)* is triggered and the *guard condition(s)* are meet. On a state chart, these information is encoded by the following syntax:

Event name [the condition(s)] / state transition action(s);

Note that a event in state chart needs not be a clock event. A meaningful update event for a transition to be taken can really be anything, say when the *seat sensor* registers a rising edge. Whenever an event is missing on a transition arch, it just means ANY event can trigger such transition. Similarly, the guard condition is optional.

[4]Possible implementation schemes are HW counter (i.e. HW), or SW counter. The later is exploited here because of the rather tolerant timing requirements in our case and the simplicity.
[5]see §1.5.3 for the reason why it is not an adamant "will".

1.5.2 Temporal Logic

In our example, the timing event of primary interest is the timeout event that *should* trigger a warning signal: The trigger of state transition from `SEAT_NO_BELT` to `SEAT_WITH_BELT` can be modeled as `after(`α `sec)` which says if the system is "stuck" at state `SEAT_NO_BELT` for α second transit to state `SEAT_WITH_BELT`.

For the sack of completeness, it is worthwhile to note that one should refrain from modeling state chart with self-looping typical of many FSM *implementations* because this is simply unnecessary[6] for the modeling stage and can lead to abstruse state machine model[7].

1.5.3 Exploiting the sequential execution nature

A FSM at any given state MUST have **ONE deterministic** state transition, be it self-looping or transition to any other state. Consequently, one always has to ensure each state has a set of exit transition (including self-looping) that covers the entire set of possible scenarios for each state. In the worst case, assuming we have n binary inputs and no extended state variables, there are 2^n possible scenarios for each state. Furthermore, we will very likely end up with "Spaghetti code" — fraught with if-else constructs and/or AND, OR operations to be evaluated as inside the if-condition test.

Fortunately, state charts (geared towards sequential execution) permit us to considerably simplify the matter:

Default transition If there is no events that can trigger a transition, it is not unreasonable to deem the (sub)system to remain at this state. Note that this is NOT self-looping.

Consequently, we are relieved from checking if *each* state in the system has a "complete" set of transition conditions.

Execution sequence or priority Mainstream computers invariably execute code sequentially. As far as the execution of the update code (here, we assume there is only one for a state machine) is concerned, whether several transitions are triggered at the same time is irrelevant because **once** a (the first) valid trigger is found, **that** transition is taken.

As a result, any transition from a given state does NOT have to be mutually exclusive which helps eliminating spaghetti coding. Since this implies the possibility that multiple state transition triggers become valid at the same time, we can simply assert the execution order or priority of the outgoing arches for each state to ensure *determinism*. In fact, this might corresponds what we might intuitively have in mind! For our example, given the system is at state `SEAT_NO_BELT` which from previous discussion (Table 1) has three possible transitions, it is logical to assign the highest priority to the transition to state `WARNING`, i.e. we wish to start timing with minimal latency. The same argument applies if the system is at state `SEAT_WITH_BELT` so we prioritize the transition to state `SEAT_NO_BELT`.

Although it is not always straight forward to rank the relative importance, I decided to assign the priorities to all states that have multiple possible transitions. By doing so, we might on the one hand compromise some design freedom. On the other hand, such loss is minute or even non-existent if the transition conditions do not overlap. Above all, it is just a matter of *who* decides it, either you or the tool. Furthermore, it is very straight forward to modify those priorities at a later point of time, shall there be such need.

[6] use *during* actions instead

[7] If the self-loop is taken, it is deemed as leaving and **re**-entering that state, i.e. not counted as "stuck" at this state!

1.6 The outcome of this stage: the state chart model

The principles outlined above yield the following visual formalism which can be served as the formal requirements specification of our Seatbelt Reminder System. It is remarkable that such an intuitive graphical representation is actually expressive enough to capture all of the desired behavior.

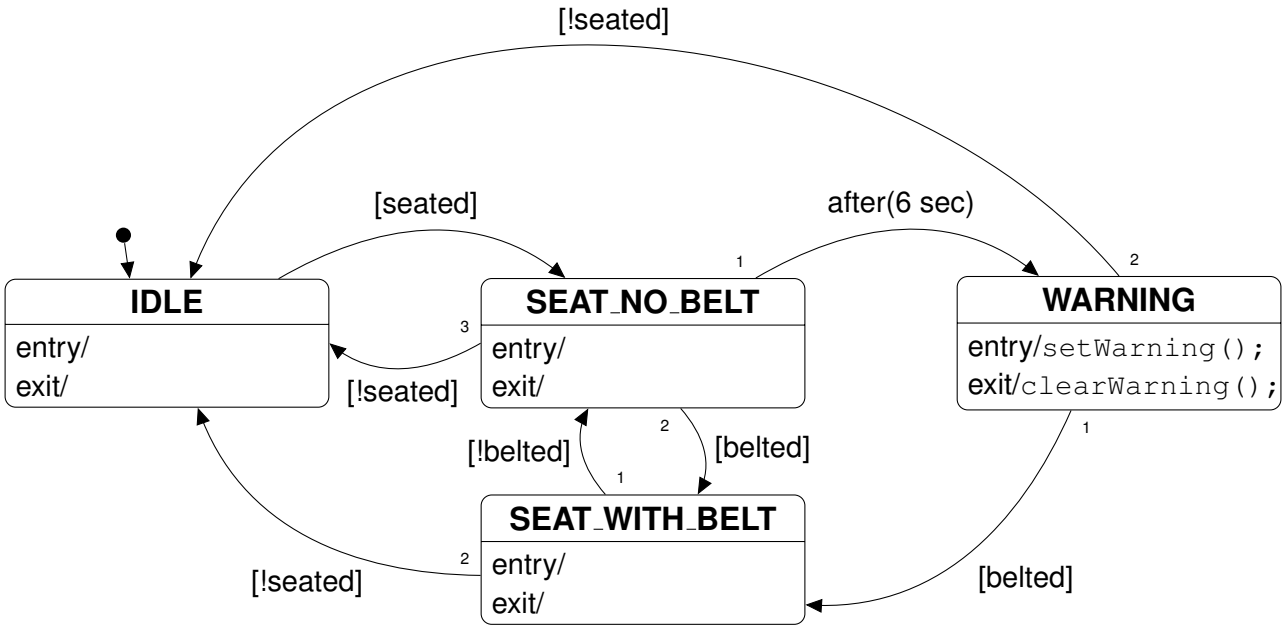

Figure 1: The resultant state chart of the Seatbelt Reminder System

2 Implementing the state machine in C

Implementing the state machine as depicted by the state chart of Fig. 1 often require some transformation into a conventional FSM which can be readily converted into a C-implementation.

This transformation augments the state chart with more implementation specifications. In our example, these design specifications are mainly:

- A periodic polling implementation, i.e. the system gathers the input values at regular time interval of 0.5 ms, and updates the output according to the control logic;
- An additional software counter that track the elapsed time being at state `SEAT_NO_BELT`. Note that this approach exploits the 0.5-ms periodic occurrence of the update event. If this update event is aperiodic, we might need to resort to an independent HW counter (i.e. timer). In fact, this counter regardless of how it is implemented becomes an extended state variable. Its initialization and update will inevitably creates more transition actions. Furthermore, we have to convert the 6-sec allowance into appropriate threshold values to which the counter value compares.

The transformation also includes cascading state *entry* and *exit* actions into the relevant transition arches. The end result is summarized by an implementation specific FSM in §2.3.

2.1 Implementing Temporal logic with SW counter

Implementing the 0.5 ms update cycle For the target hardware I used, the simplest way to implement the periodically updated FSM is via the *SysTick Handler* in which the update code is written into the SysTick Interrupt Service Routine (ISR).

The target clock tree is configured in such a way that the SysTick Interrupt Request (IRQ) arrives every 0.5 ms with acceptable precision. Fig 2 shows part of the details which ultimately decides the SysTick timer input frequency.

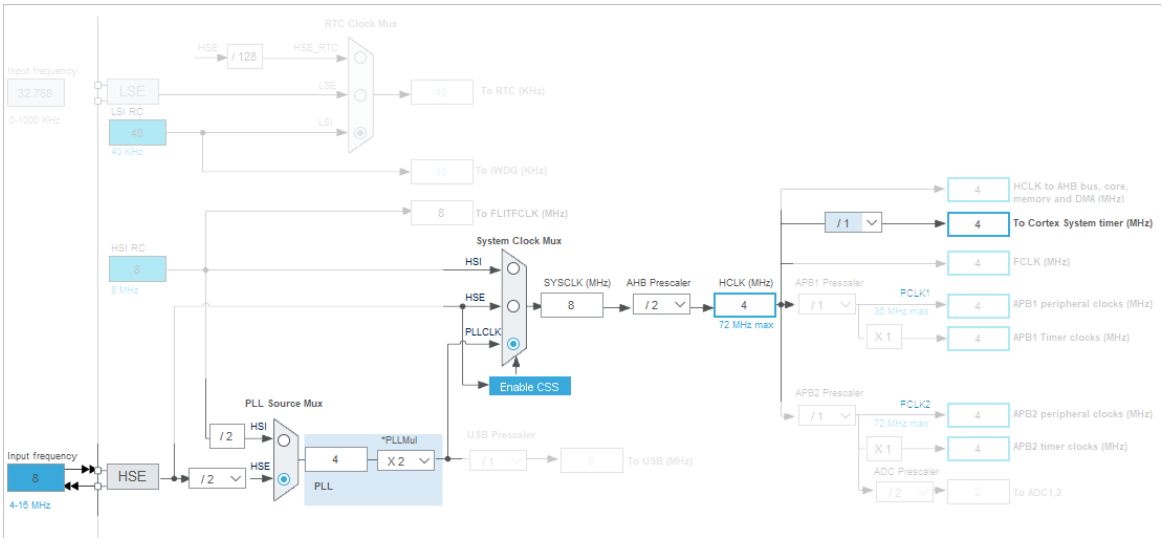

Figure 2: The clock tree configuration used in this project.
The corresponding code is attached in §B.2.

The rest lies on the calculation of the required auto reload value for triggering a SysTick IRQ. This can be calculated by 0.5 ms $\times$ 4 MHz $- 1 = 1999$. Note that the CMSIS `SysTick_Config` function used in the code (see the `main` function in B.1) does us a favor: the ugly "minus 1" due to the fact the timer counts from 0 is abstracted away.

The SW counter and additional transition actions The following implementation decisions are made

- The counter is initialized as 1 every time the system enters state `SEAT_NO_BELT`. Note that it is advantageous to classify this as an entry action.
- Moreover, the counter is increment by 1 every time the system re-loops into this state.

The implementation-specific transition condition Since the threshold for triggering a transition from state `SEAT_NO_BELT` to `WARNING` should *correspond* to the duration of

$$\frac{6 \text{ sec}}{0.5 \text{ ms per update events}} = 12000 \text{ update events.}$$

With the counter initialization and update actions defined as above, the threshold value should also be 12000. In our implementation, this is simply to replace the *after(6 sec)* in Fig. 1 with a conventional condition: [counter >12000] .[8]

[8] As mentioned earlier, it can be a bit tricky to inspect the counters: the corresponding threshold actually depends on how or when we initialize them, how they are incremented/ decremented as well as whether you use $>$ comparison or $\geq$.

2.2 Cascading the transition actions

Now, it is the right time to cascade all the elegant *entry* and *exit* actions into a single transition action for each transition arch.

For our case (a non-hierarchical state machine), the cascading involves 3 parts:

Suppose the system is leaving from state X into state Y in which this transition has an additional transition action. The cascading is as simple as just combining these actions in the following order:

1. exit action of state X, followed by...
2. (arch-specific) transition action from state X to Y, and then
3. entry action of state Y.

2.3 The resultant FSM

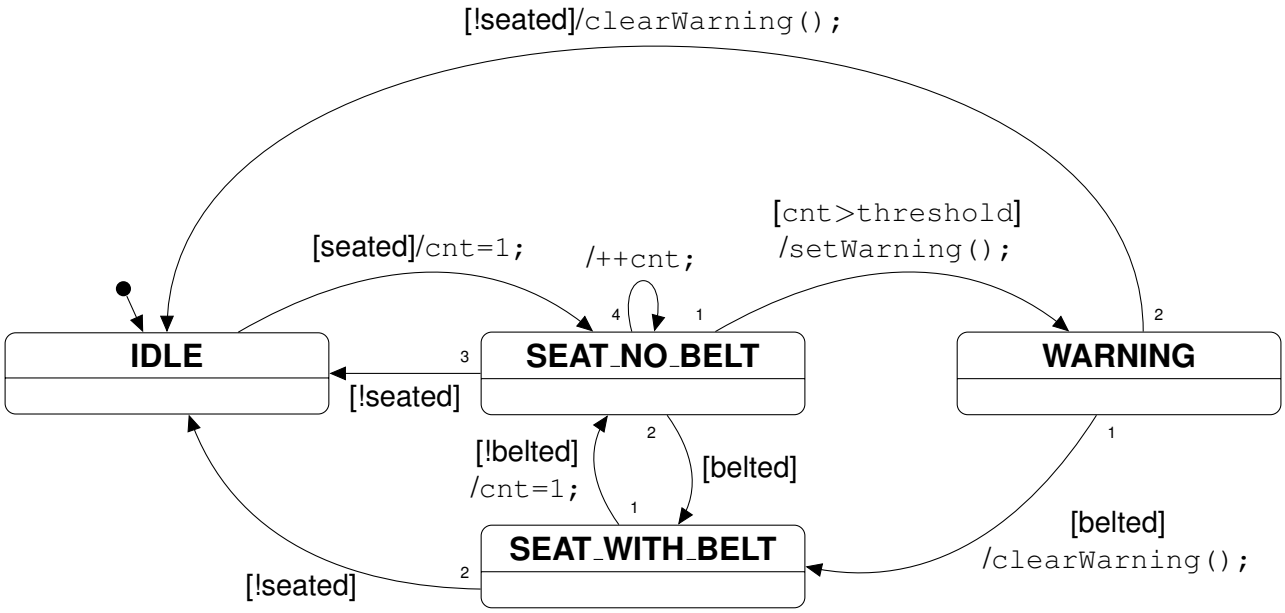

Figure 3: The resultant FSM of the Seatbelt Reminder System. Apparently, it is more implementation oriented and less intuitive than Fig. 1 because of both the cascading of state actions and the extended state variable. In this implementation, the extended state variable is `cnt`. To *correctly* implement the specified behavior in Fig. 1, it is not only necessary to convert the 6-sec into the appropriate `threshold`, but also to calculate the proper initial value!

2.4 Update the state value

Actually, the cascading in §2.2 misses a critical part: *updating the system state* so that in the next update cycle the system is on the "right track" to decide where to transit to. It is appropriate to *integrate* the cascading steps above with a memory update — overwriting the (global) state variable.

To minimize **latency in updating our output** (which is ideally the first action of the cascaded transition action sequence), it is advantageous to execute the state update last — i.e. *appending* the cascading steps.

Therefore, the cascading in any implementation indeed looks like:

1. exit action of state X, followed by...
2. (arch-specific) transition action from state X to Y, and then
3. entry action of state Y, and finally
4. update the state variable (named `stateSeat1` in my code).

2.5 The resultant C-code Implementation

The FSM in Fig. 3 can be **easily and efficiently** implemented using a *switch*-construct as follows:

Snippet of the ISR implementing the FSM in Fig. 3 written in C

```
switch (stateSeat1){
  case IDLE: // aka state 0
  if (isSeated){
    ACTION_WHEN_ENTERING_SEATED_NO_BELT;
    stateSeat1 = SEAT_NO_BELT;     // 0-->1
  }
  break;
  case SEAT_NO_BELT: // aka state 1
  if (elapsedTime_until_BucklingUp > delayBeforeWarning) {
    ACTION_WHEN_ENTERING_WARNING;
    stateSeat1 = WARNING;      // 1-->3
    break;
  }
  if (isBelted){
    stateSeat1 = SEAT_WITH_BELT;  // 1-->2
    break;
  }
  if (!isSeated){
    stateSeat1 = IDLE;     // 1-->0
  }
  elapsedTime_until_BucklingUp++;    // 1-->1
  break;
  case SEAT_WITH_BELT: // aka state 2
  if (!isBelted){
    ACTION_WHEN_ENTERING_SEATED_NO_BELT;
    stateSeat1 = SEAT_NO_BELT;       // 2-->1
    break;
  }
  if (!isSeated){ // Just in case
    stateSeat1 = IDLE;               // 2-->0
    break;
  }
  break;
  case WARNING: // aka state 3
  if (isBelted){
    ACTION_WHEN_LEAVING_WARNING;
    stateSeat1 = SEAT_WITH_BELT;     // 3-->2
    break;
  }
  if (!isSeated){
    ACTION_WHEN_LEAVING_WARNING;
    stateSeat1 = IDLE;               // 3-->0
    break;
  }
```

```
45    break;
}
```

fsm–c.txt

The complete application source file (with more inline comments, variable declarations and overhead of polling input values) is attached in B.1. The notation in the code is more verbose compared to Fig. 3. The correspondences should be clear.

We might replace all the `break;` keyword with `return;` to further reduce program size and the runtime. This is possible here because the SysTick ISR has no other code to execute after this FSM.

3 The bigger picture

3.1 Testing and Tool support

Verification of the implementation At this stage, we want to prove the *correctness* of our implementation **with respect to the state chart model**. I plan to post a video to show some black-box test cases.

Validation of the state chart This project is all about prototype demonstration (i.e. a hobby project). We did not really systematically validate the model specification at the first place. What we did was pretty much thought experiments! If it turned out we had misunderstood the requirements, it would cost some time to correct our implementation.

Tool support On top of that, the manual conversion from the state chart model to implementation code can be error-prone.

Notwithstanding these shortcomings in our demo project, the model-based programming (or cyber-physical system design, in general) approach *can* actually streamline the entire development process. It allows us to validate our requirements as early as possible through *simulation* while the model-to-implementation conversion process if automated (i.e. *code generation*) *can* ensure *correct* implementation. All we need are some sort of development tool support, possibly with some also some runtime support.

In the future, I might also update this project to incorporate more tool support.

3.2 Deploying concurrent state machines into a single target

What if we want to deploy several *instances* of the Seatbelt Reminder System into the same MCU? A conventional way is a *superloop* that is more like round-robin scheduling but this scheduling is fixed at compile time. Obviously, this is not necessarily the most efficient way to handle such physical parallelism.

With event-driven paradigm, we can *model* each instance as a *concurrent* state machine. Thereby, we can readily *reuse* the artifacts in this project. [2] provides great details regarding this aspect.

References

[1] Niklas Gürtler. *ARM-ASM-Tutorial*. URL: https://www.mikrocontroller.net/articles/ARM-ASM-Tutorial.

[2] Miro Samek. *Practical UML Statecharts in C/C++*. 2nd ed. Newnes, 2009.

[3] Marilyn Wolf. "State machines". In: *Computers as Components: principles of embedded computing system design*. 4th ed. Morgan Kaufmann, 2017, pp. 222–224.

A Schematics of the System

Please check Figure 4. You might also want to see §1.1 which defines the system input/ output.

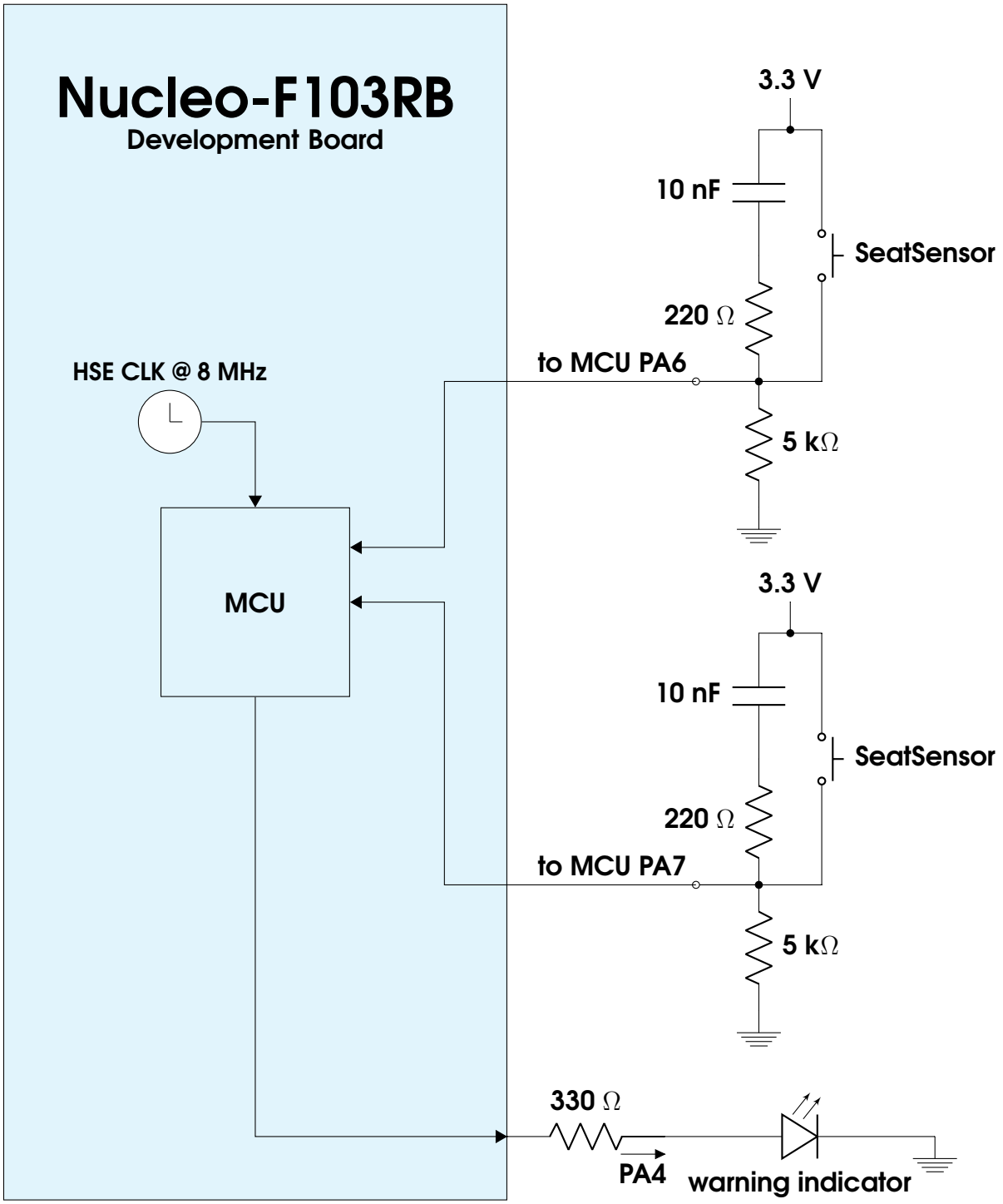

Figure 4: A schematics showing the pin assignment of the MCU and its external connections

B The source files

B.1 application C-code

```
#include "include/stm32f103xb.h"

// manual timing calculation at comile time
    /*
     * assumptions (these are configured in the startup.S and explained in
    startup-Clock-setup-info.txt):
     *   (not repeated here)
     * specifications:
     *  1.  FSM update Interval = 0.5 ms (i.e. IRQ period)
     *  2.  allowed time elapsed from being seated to finish buckling the
    seatbeat
     *        WITHOUT triggering the warning LED or buzzer = 6 sec
     *
     * the corresponding constant literals to evaluate:
     *  1.  IRQ period in unit time of SysTick Input Clock
     *  2.  maximum delay before triggering warning (i.e. in unit time of ISR
    Period!!!)
     *
     * make sure they are ALL integers AND within their respective expected
    range!!!
     */
    #define SysTimerInterval_ticks 2000 // see item 1 above, range: 1... 2^24
    #define delayBeforeWarning 12000 //12000    // see item 2 above, expect 1...
     2^32

// MCU Digital I/O usage (in pin number, all in GPIO Port A)
    // FSM inputs (all in GPIO Port A)
    #define INPUT_SEAT_OCCUPIED 6
    #define INPUT_BELT_ON  7

   // FSM outputs (also in GPIO Port A)
    #define OUTPUT_WARNING_SIGNAL 4 // could be a warning LED or buzzer or ...

// some helpful macros concerning state transition actions
// (this facilitates examining when such action is executed during code analysis
    )
    #define ACTION_WHEN_ENTERING_WARNING  \
            GPIOA->BSRR = (1<<OUTPUT_WARNING_SIGNAL)
        // turn on the warning output!!!
    #define ACTION_WHEN_LEAVING_WARNING  \
            GPIOA->BSRR = (1<<(OUTPUT_WARNING_SIGNAL+16))
        // turn off the warning signal
    #define ACTION_WHEN_ENTERING_SEATED_NO_BELT \
            elapsedTime_until_BucklingUp = 1
// more macros
    #define getState(GPIOx,pinNum) (((GPIOx->IDR)>>pinNum)& 1U)==1U

    enum SeatBeltReminderFsmStates{IDLE,SEAT_NO_BELT, SEAT_WITH_BELT, WARNING};

// global variable (assuming only modified by the SysTick Handler)
    static enum SeatBeltReminderFsmStates stateSeat1 = IDLE; // i.e. initial
    state
      // note: benefit of initial state being numerically 0: save (some) flash
    space
```

```
    static uint32_t elapsedTime_until_BucklingUp;  //actually
        /* time in state SEAT_NO_BELT  [in unit time of SysTick IRQ period]
         * (for visualization, we could keep counting it in State 3*/

// prototypes
    int main(void); // only for startup configuration, empty while-loop
    void configIO(void);
    void SysTick_Handler(void);

// definitions
int main(void){
    configIO();
    (void) SysTick_Config(SysTimerInterval_ticks); // macro defined in "core_cm3
    .h"
    while (1);/*better alternative: inline with wfi */
    return 0;
}

void SysTick_Handler (void){  //ISR, see STM32F1_vecTable.S
// exit the ISR if the IRQ was not REALLY caused by the SysTick
    if (((SysTick->CTRL)&SysTick_CTRL_COUNTFLAG_Msk) !=
    SysTick_CTRL_COUNTFLAG_Msk) {
        return ;
    }


// the Time-triggered Seat Belt Reminder System FSM

  // fetching physical (push button) inputs
  volatile uint32_t isSeated = getState(GPIOA,INPUT_SEAT_OCCUPIED);
  volatile uint32_t isBelted = getState(GPIOA,INPUT_BELT_ON);

  // performing the FSM logic
  switch (stateSeat1){
    case IDLE: // aka state 0
            if (isSeated){
                ACTION_WHEN_ENTERING_SEATED_NO_BELT;
                stateSeat1 = SEAT_NO_BELT;      // state transition 0-->1 (even
    if belted)
            }
            // omitting the direct state transition: 0-->2
            //  if seated && belted && currently in IDLE state: 0-->1-->2
    instead
            break;
        case SEAT_NO_BELT: // aka state 1
            if (elapsedTime_until_BucklingUp > delayBeforeWarning) {
                ACTION_WHEN_ENTERING_WARNING;    // note: this is the only way
    to state 3
                stateSeat1 = WARNING;           // state transition 1-->3
                break;
            }
            if (isBelted){
                stateSeat1 = SEAT_WITH_BELT;    // state transition 1-->2
                break;
            }
            if (!isSeated){
                stateSeat1 = IDLE;              // state transition 1-->0
            }
            elapsedTime_until_BucklingUp++;
            break;
```

```
        case SEAT_WITH_BELT: // aka state 2
            if (!isBelted){
                ACTION_WHEN_ENTERING_SEATED_NO_BELT;
                stateSeat1 = SEAT_NO_BELT;      // state transition 2-->1
                break;
            }
            if (!isSeated){ // Just in case
                stateSeat1 = IDLE;              // state transition 2-->0
                break;
            }
            break;
        case WARNING: // aka state 3
            if (isBelted){
                ACTION_WHEN_LEAVING_WARNING;
                stateSeat1 = SEAT_WITH_BELT;    // state transition 3-->2
                break;
            }
            if (!isSeated){ // and still belted...!?
                ACTION_WHEN_LEAVING_WARNING;
                stateSeat1 = IDLE;              // state transition 3-->0
                break;
            }
            break;
    }
}

void configIO(void){ // change me with HAL if you want to!
    // RCC control
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN; // enable GPIO Port A

    // configure the pins for the seat and belt sensor inputs AND the
   OutputSensor
        // clear the value for the 3 pins involved
        GPIOA->CRL &= ~(GPIO_CRL_CNF7 | GPIO_CRL_MODE7|
                        GPIO_CRL_CNF6 | GPIO_CRL_MODE6|
                        GPIO_CRL_CNF4 | GPIO_CRL_MODE4);
        // set the corresponding bits according to our Pin Assignment
         GPIOA->CRL |= (
          GPIO_CRL_MODE4_1 |  // PA4 output
            GPIO_CRL_CNF6_0|  // PA6 (seat detector input as floating input,
   externally pulled down)
            GPIO_CRL_CNF7_0  // PA7 (belt detector input as floating input,
   externally pulled down)
         );
}
```

../app.c

B.2 Startup file

The startup code and clock configuration function were adapted from [1] in which the clock configuration function is adapted according to the specifications in §2.1.

```
.syntax unified
.cpu cortex-m3
.thumb

.include "include/STM32F103xx.inc"

.text
```

```
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    ldr r0, =_DataStart
    ldr r1, =_DataEnd
    ldr r2, =_DataLoad

    b 2f
1:  ldr r3, [r2], #4
    str r3, [r0], #4
2:  cmp r0, r1
    blo 1b

    ldr r0, =_BssStart
    ldr r1, =_BssEnd
    ldr r2, =0

    b 2f
1:  str r2, [r0], #4
2:  cmp r0, r1
    blo 1b

    bl configSysClock
    bl main
1:  bkpt
    b 1b
    .ltorg

.type configSysClock, %function
configSysClock:
    @ Turn on HSE
    ldr r0, =RCC
    ldr r1, =((1 << RCC_CR_HSION)| (1 << RCC_CR_HSEON))
    str r1, [r0, #RCC_CR]

    @ Pre-configure PLL (see startup-Clock-setup-info.txt)
    ldr r2, =((0<<RCC_CFGR_PLLMUL)|(1<<RCC_CFGR_PLLXTPRE)|(1 << RCC_CFGR_PLLSRC)
    | (8<<RCC_CFGR_HPRE))
    str r2, [r0, #RCC_CFGR]

    @ Pre-Calculate value for RCC_CR
    orr r1, #(1 << RCC_CR_PLLON)
    @ Wait for HSE ready
1:  ldr r3, [r0, #RCC_CR]
    ands r3, #(1 << RCC_CR_HSERDY)
    beq 1b
    @ Turn on PLL
    str r1, [r0, #RCC_CR]

    @ Pre-Calculate value for RCC_CFGR
    orr r2, #(2 << RCC_CFGR_SW)
    @ Wait for PLL ready
1:  ldr r3, [r0, #RCC_CR]
    ands r3, #(1 << RCC_CR_PLLRDY)
    beq 1b

    @ in my case, Sys Clock only 4 MHz, not exceeding 24 MHz,
    @ so it is not necessary to add wait state to the flash
```

```
    @ Switch system clock to PLL
    str r2, [r0, #RCC_CFGR]

    @ Pre-Calculate value for RCC_CR
    @ (while waiting PLL to become selected by RCC as SysClock source)
    bic r1, #(1 << RCC_CR_HSION)
    @ Wait for switch to PLL
1:  ldr r3, [r0, #RCC_CFGR]
    and r3, #(3 << RCC_CFGR_SWS)
    cmp r3, #(2 << RCC_CFGR_SWS)
    bne 1b

    @ Turn off HSI to save power
    str r1, [r0, #RCC_CR]

    bx lr
    .ltorg
```

../startup.S