LAB INSTRUCTIONS

Lab: The Elevator Control System

LAB DESCRIPTION

* Done in groups (2 people)
* Various aspects of developing an embedded system (elevator system): specification, design, implementation, testing
* Done using simulator, no real embedded hardware is used
* We will provide help with the lab on the following dates: (it is optional to come to those labs slots, attend and ask if you have trouble solving the assignments)
  + **Thursday November 30th**, 13:15 - 15:00, room 2315
  + **Thursday December 7th**, 13:15 - 15:00, room 2315
* Deadline Part I (Questions 1-3) **December 1st, 2017**
* Deadline Part II (Questions 4-5) **December 14th, 2017**
* Final deadline for revisions (all questions): **January 7th, 2018**

INTRODUCTION

In this lab you will practice how to implement control applications in C and FreeRTOS, how to access sensors and actuators, how to simulate the resulting system, how to specify safety requirements on a system, and how to create and automatically execute test-cases. The lab is (remotely) inspired by a similar assignment first made by Prover Technology AB, then adapted for previous versions of a similar course at Chalmers University by Carl Johan Lillieroth, K.V.S. Prasad, Mary Sheeran, Angela Wallenburg, and Jan-Willem Roorda; later adapted for a course at the University of Iowa by Cesare Tinelli. This version of the assignment is significantly extended, uses C/FreeRTOS instead of Lustre, and has a stronger focus on implementation and testing than on formal verification. As in previous exercises, we will consider the [**STM32F103 micro-controller**](http://www.keil.com/dd/chip/4794.htm) as platform.

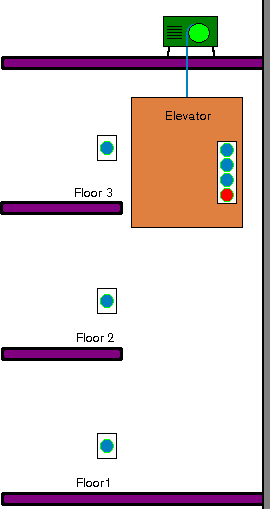
THE ELEVATOR

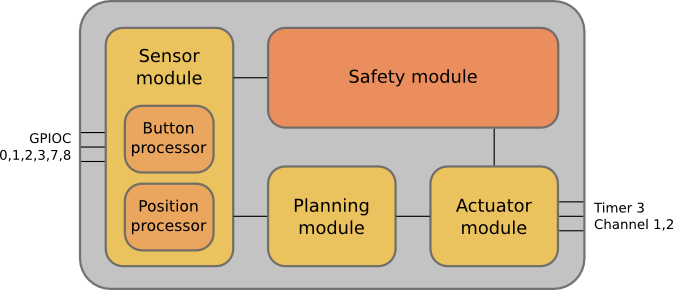
Consider a simple elevator, moving people up and down between three floors. A picture of the situation is shown to the right.

In the elevator car, there are four buttons: buttons 1, 2, 3 to go to the respective floor, and a stop button to stop the elevator. On each floor, there is a button to call the elevator. Furthermore, the elevator has two sensors, the first indicating if the elevator is on a floor or in between two floors, the second one indicating if the elevator's door is closed or not. There is also a linear position sensor for measuring the position of the elevator in the elevator shaft.

The elevator is moved up and down by a motor that is on the roof of the building, at a maximum speed of 50cm/s. The motor is controlled by two signals, one for moving the elevator up and one for moving it down; both signals are driven using pulse-width modulation (PWM) to enable smooth acceleration and deceleration of the elevator car. The distance between two floors is 400cm.

For sake of simplicity, the doors are in our case study controlled by some external unit, the elevator system is only able to sense whether the doors are open or closed. The goal of this lab is to implement and test the software controlling the elevator.





These different modules will run as concurrent tasks in the system, with communication between the modules partly realised using event queues (message passing), partly using shared variables protected by mutex semaphores. The priorities of the tasks are chosen as follows:

| **PRIORITY** | **TASK** |
| --- | --- |
| 4 | Position processor |
| 3 | Safety module |
| 2 | Actuator module |
| 1 | Button processor, planning |

Some parts (in particular, the actuator module) are already provided to you as a starting point (see the "Submission part 1" page in the Student portal). The missing implementations of the remaining modules are described in the following sections.

**NB:** If you get a message about a failing safety assertion right after unpacking, compiling, and starting the software, this is caused by the assumption 1 in question 3 below: "The doors can only be opened if the elevator is at a floor and the motor is not active". This can be prevented by telling the software that the elevator is initially at a floor (setting GPIOC pin 7 to 1 before starting the simulation); you can also comment the assumption in file "safety.c" for the beginning.

We assume that our STM32F103 micro-controller is connected in the following way to the buttons, sensors, and actuators of the elevator system:

| **PIN OF THE STM32F103** | **COMPONENT** |
| --- | --- |
| GPIOC 0 | Button to call elevator to floor 1 (1: button is pressed, 0: button is released) |
| GPIOC 1 | Button to call elevator to floor 2 |
| GPIOC 2 | Button to call elevator to floor 3 |
| GPIOC 3 | Stop button |
| GPIOC 7 | At-floor sensor (1: elevator is at a floor, 0: elevator is in between floors) |
| GPIOC 8 | Door sensor (1: doors are closed, 0: doors are open) |
| GPIOC 9 | Position sensor |
| Timer 3, channel 1 | Motor signal for moving elevator upwards |
| Timer 3, channel 2 | Motor signal for moving elevator downwards |

LAB QUESTION 1: THE BUTTON PROCESSOR

Buttons can directly be connected to the pins of general-purpose I/O ports of a micro-controller; in the elevator system, this is done as specified in the table above. The value of a button (whether the button is pressed or not) can be read by calling the corresponding firmware function (or by directly reading from the right special-purpose register), e.g., "GPIO\_ReadInputDataBit(GPIOC, GPIO\_Pin\_2);" In order to deal with bouncing buttons, a button should only be considered as pressed when the corresponding pin bit is set for some amount of time (say, 20ms). Also, we are only interested in the *rising* or *falling* edges of the pin signal: even a button that is pressed for a long time should only generate one "key-press" and one "key-release" event.

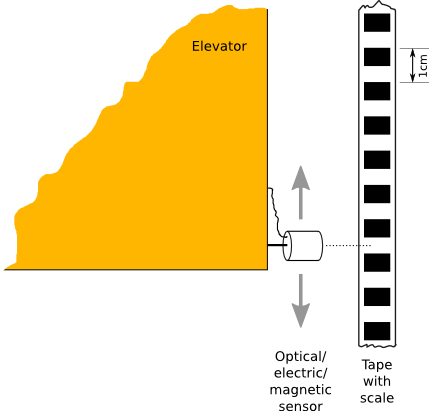
In this question, you have to implement a task that periodically polls the values of the pins GPIOC 0, 1, 2, 3, 7, 8, and generates "press" and "release" events as specified in "main.c". For simplicity, we consider also the at-floor and door sensors as buttons in this context. Add your implementation to the file "pin\_listener.c", where skeletons of the functions to be implemented already exist.

**NB:** when implementing de-bouncing, it is not possible to use a delay statement in the function "pollPin" in file "pin\_listener.c", since this function is supposed to be called repeatedly in each iteration for all pins. You have to make sure that "pollPin" returns without any delays.

Test your implementation using the uVision simulator. To visualise the generated events, you can add some "printf" statements to the "plannerTask" function in file "planner.c" (this is the place where events will be consumed in the final system). To simulate pressed buttons, open the dialogue **Peripherals - General purpose I/O - GPIOC**, where you can manually toggle the buttons in the row "Pins."

LAB QUESTION 2: THE POSITION PROCESSOR

One of the simplest methods to measure the vertical position of an elevator is a linear position sensor, reading from a vertical tape that is installed in the elevator shaft.



When the elevator moves in up- or downward direction, the sensor will generate a sequence of (rectangular) pulses corresponding to the number of markers of the tape that are passing by. By counting pulses, the movement and position of the elevator can be determined. In this lab, we assume that there is one marker per 1cm on the tape; this means that if the sensor generates pulses at a sequence of 1Hz, the elevator is moving at a speed of 1cm/s. Since the maximum speed of our elevator is 50cm/s, the maximum frequency of pulses will be 50Hz, and the minimum length of each pulse is 10ms (half the length of the pulse period).

Counting of pulses can be done directly using a micro-controller, either in software or in hardware. The latter is done by setting up one of the timers of the controller in the right way, and can handle also very high frequencies. For simplicity, in this lab we do the counting in software, which works more or less like the recognition of key presses in the previous question.

In this question, you have to implement a task that is polling the status of the position sensor (connected to GPIOC9) every 3ms, and that is updating the variable "position" in "position\_tracker.h" accordingly. We need to determine the position of the elevator with high reliability (to prevent the elevator from crashing into the floor or ceiling of the shaft), so make sure that your implementation can count pulses at up to 50Hz without missing any! Add your implementation to the file "position\_tracker.c". This task also needs to know about the current direction of the elevator (which cannot be determined directly from the pulses), which is provided by the actuator module through the function "setDirection". You can assume that the elevator is at floor 1 when the system is started up.

Testing the position processor using environment simulation

To test the implementation of the position processor, we need to generate pulses at the right frequency, which can hardly be done manually. A much more systematic method is to test the controller using a *simulation* of the environment, in this case, of the elevator and motor hardware (**this is called "software-in-the-loop" and will be discussed in more detail in the lectures**). Such simulations are frequently written in high-level languages like Matlab/Simulink; in uVision, simulations can be written as [**"Debug Functions"**](http://www.keil.com/support/man/docs/uv4/uv4_debug_functions.htm) in a subset of C. Debug functions are user-specified programs that are executed during simulation in parallel with the actual controller, and which can access and manipulate I/O ports of the controller to simulate the environment.

For the lab, we provide a simple simulator of the elevator in the file "elevator\_simulator.ini". To use this simulator, change into the uVision debugging view, choose **Debug - Function Editor**, where you load "elevator\_simulator.ini", then press the button **compile**. The simulator is now available and can be started by typing "simulateCarMotor();" in the debugging Command window. If you now start the simulation, a continuous stream of values "0.0000" will be printed in the Command window, which represents the current position of the elevator. In order to move the elevator, add a call like "setCarTargetPosition(50);" to one of the tasks in the system, which will cause the actuator module to start the motor, and finally the environment simulation to generate pulses. You can see those pulses when you open the dialog **Peripherals - General purpose I/O - GPIOC**.

LAB QUESTION 3: THE SAFETY MODULE

An elevator is a safety-critical system, which means that we carefully have to formulate safety requirements, assumptions made about the environment, and test the system properly. The safety module represents a *runtime monitor* that is observing system and environment during execution, permanently checking whether any of the requirements and assumptions are violated. In this case, the safety module is able to override the planning module and stop the elevator. The runtime monitor is sampling-based and assesses the state of the system every 10ms; for this, the safety module directly reads values from the GPIO ports of the micro-controller, but also uses data produced by the input module (in particular the position of the elevator). The safety module will also be important when testing the control system, since it can detect potentially dangerous behaviour already during simulation.

Environment assumptions

When implementing the control system, various assumptions are made about the environment. If any violations of the assumptions are detected during operation, this either means that the assumptions were incorrect, that a hardware defect has occurred, or that some of the sensors of the control system do not work correctly. In all cases, the elevator has to be stopped to prevent more serious consequences.

1. The doors can only be opened if the elevator is at a floor and the motor is not active
2. The elevator moves at a maximum speed of 50cm/s
3. If the ground floor is put at 0cm in an absolute coordinate system, the second floor is at 400cm and the third floor at 800cm (the at-floor sensor reports a floor with a threshold of +-0.5cm)

Add one further assumption of your own (that is not already captured by the assumptions above). Translate all assumptions into C code and add them to the file "safety.c" (the first assumption is already present in the file).

**Justify how you translated from natural language to C code: explain why the C code that you have written corresponds to the assumptions given above, and the one found by yourself; in particular be careful with (logical) implications!**

Safety requirements

Similarly to environment assumptions, we also formulate requirements on the admissible behaviour of our control system:

1. If the stop button is pressed, the motor is stopped within 1s
2. The motor signals for upwards and downwards movement are not active at the same time
3. The elevator may not pass the end positions, that is, go through the roof or the floor
4. A moving elevator halts only if the stop button is pressed or the elevator has arrived at a floor
5. Once the elevator has stopped at a floor, it will wait for at least 1s before it continues to another floor

Add one further requirement of your own. Translate all assumptions into C code and add them to the file "safety.c" (the two first requirements are already present in the file). **As for the environment assumptions, justify how you translated from natural language to C code.**

**NB:** you might wonder what the precise difference between environment assumptions and safety requirements is. A short answer: environment assumptions talk about the **inputs** of the control system (in our case, GPIOC 0,1,2,3,7,8,9), while the requirements talk about the **outputs** of the system (here, the channels of timer 3). In other words, environment assumptions are properties that the control system has to rely on, while the requirements concern properties that are controlled and should be ensured by the system.

LAB QUESTION 4: THE PLANNING MODULE

The planning module implements the high-level control logic of the elevator and is responsible for consuming key events and position information generated by the input module, and for deciding where the elevator should go. The latter is done by calling the function "setCarTargetPosition" in file "global.h". If the stop button is pressed, the planner also has to make sure that the elevator is stopped through the function "setCarMotorStopped".

The planner should never forget about key presses and make sure that the elevator eventually goes to every floor to which it was called and stops there for some amount of time (at least long enough to open the doors); passengers should not suffer from starvation. Vice versa, the elevator should not stop at floors to which it was not called. The planner also has to be careful with changing the direction of the elevator too abruptly. E.g., if the elevator is moving from floor 1 to 3, an incoming call to floor 2 has to be postponed if the elevator is already too close to floor 2 to stop.

Add your implementation of the planner to the file "planner.c". While developing this module, it might be reasonable to work with a reduced floor distance (say, 20cm instead of 400cm) to speed up simulation. To make this change, you have to modify some of the constants in "safety.c" and "motor.c". **But be sure to change your system back to 400cm in the end!**

LAB QUESTION 5: TESTING AND SIMULATION

Once all parts of the system have been implemented, you should be able to simulate both the control system and the elevator as described in question 2. To give commands to the system, open the dialogue **Peripherals - General purpose I/O - GPIOC** where you can use pins 0, 1, 2, 3, 8 to take over the role of passengers. Test whether the system behaves as it is expected from an elevator, and whether you can violate any of the assumptions and requirements formulated in question 3; in this case, either the system contains bugs that have to be fixed, or the formalisation of the assumptions and requirements might have to be refined.

This method of *interactive, online* testing has the advantage that it is easy to assess whether system behaviour is correct (manually), but the disadvantage that the testing process is not automated. When developing complex system, usually a lot of time can be saved (and the quality of the resulting implementation significantly improved) by writing test cases (or *test scripts*) that can be executed automatically. This way the tests can be rerun with little effort each time the control system has been modified.

In uVision, test cases can again be written as debug functions. We provide two examples for this in the files "testcase0.ini" and "testcase1.ini":

* "testcase0.ini": tests the scenario that the elevator is called from floor 1 to floor 2.
* "testcase1.ini": starting from floor 1, it is simulated that the elevator is called to floor 2. While the elevator is in between floors 1 and 2, further calls to the floors 1 and 3 are received. The test script then checks that the elevator will eventually (after some reasonable amount of time) have visited both floors 1 and 3, without enforcing a particular order in which floors 1 and 3 are visited (since no order is specified for the planning module)

To run the test cases, you have to change to the debugging mode, where you first load and compile "elevator\_simulator.ini", and then, say, the file "testcase0.ini". You can then start the debug functions "testCase0();" and "simulateCarMotor();" in parallel in the debugging Command window, by simply typing those commands after each other. Then you can start the simulation and lean back.

**Simulate your implementation using both "testcase0.ini" and "testcase1.ini", and fix any bugs that you observe in your implementation. Consider the scripts "testcase0.ini" and "testcase1.ini" as a customer specification that you are not allowed to change.**

Then, have a look at how "testcase0.ini" and "testcase1.ini" are implemented. Write a further test script "testcase2.ini" of your own, exercising the following scenario:

* Starting from floor 1, simulate that the elevator is called to floor 3. When the elevator is just about to pass floor 2, a call to floor 2 is received. Check in your test case that the elevator will first visit 3, and only afterwards floor 2. The planning module needs to implement this order, since it is not possible to stop the elevator right away when it is already very close to floor 2.

**Again, simulate your implementation using the new script "testcase2.ini", and fix any bugs that you observe.**

Random testing

Random testing is a popular and often very effective technique to fully automatically exercise systems. Rather than writing a fixed test script by hand, we can provide a debug function that will randomly simulate calls and button presses to the elevator. The assessment whether the system behaves correctly is in this situation entirely done by the safety module. The test script can in principle run forever (say, over night) to test also for very unlikely and pathological use of the elevator.

Develop such an automatic test script that randomly generates events for the GPIOC Pins 0, 1, 2, 3, 8. You have to make sure that the script never violates the environment assumption made in question 3 (e.g., by opening the doors while the elevator is in between floors). Your script should simulate *realistic* behaviour of users, in particular with respect to the (randomly chosen) time that the script waits in between events. Generating events too quickly will just fill up the "call queue" of the planning module, while too few events will make the elevator stay in the same place most of the time. **Justify the choices that you made in your report.**

**Please fix any bugs that you find in your implementation by means of testing!**

SUBMISSION

This lab should be done in group of two people; you can sign up for groups on the Studentportal course page.

**NB:** we expect submitted code to be well-written, properly formatted, and documented using comments where appropriate. Unreadable code will be rejected right away.

**Submission is done via the student portal. Also feedback will be provided via the portal.**If you go to the group page on the student portal, you will find a file area for your group where you can upload your solution. The deadlines are split into two parts:

* Part I (Questions 1-3) are due on**December 1st, 2017**
* Part II (Questions 4-5)are due on **December 14st, 2017**

You will have to submit

* a complete, compiling and working implementation of the elevator control system
* a short report documenting your solutions to questions 3 and 5
* the two test scripts to be written in question 5.

**NB:**after assessing your solution, we might request that you revise your implementation or the report to fix bugs, or to make sure that your system actually corresponds to the specification provided here. In this case we will provide feedback as soon as possible. The final deadline for revisions is **January 7th, 2018**.

**NB:** we will also test your implementation using the scripts "testcase0.ini", "testcase1.ini", and our own version of "testcase2.ini". Implementations failing any of those tests will be rejected right away.