Project Description Document

SUMMARY

The Assemblers 1
1 Document overview 2
2 Design choices 2
2.1 Hardware architecture 2
2.1.1 Hardware components 2
2.1.2 Hardware scheme 3
2.2 Software architecture 4
2.2.1 “Object-oriented” approach 4
2.2.2 Interrupts driven architecture 4
2.2.3 Controllers 4
2.2.4 Race conditions 4
2.3 Software interfaces 4
2.4 Software protocols 5
2.4.1 Blocking modules 5
2.4.2 Keypad reading procedure 5
2.4.3 Alarms delay procedure 7
2.4.4 Photoresistor ADC conversion 7
2.4.5 Alarm sound controller 8

THE ASSEMBLERS

Last name	**First name**	**ID number**
Cafaro	Davide	0622701062
Carratù	Carmine	0622701110
Cuomo	Luca	0622701121
D’Arminio	Emanuele	0622701059

1 DOCUMENT OVERVIEW

This document summarizes the work done for the Biomedical Embedded Systems class of the academic year 2019/2020 of the Computer Science Engineering Master's degree of University of Salerno.

The chosen project is the Home Security System. A description of the implementation is provided in the next sections of the document.

2 DESIGN CHOICES

The entire project, being an embedded system, can be decomposed in software and hardware parts.

Firstly, the overall hardware architecture is discussed, showing how the physical components were used and interconnected.

Next, an overview of the underlying software architecture is presented, with a detailed description of the software protocols used for the communication between the different software components.

2.1 HARDWARE ARCHITECTURE

2.1.1 Hardware components

The Home Security System makes use of several hardware components.

First of all, the STM32-F401RE microcontroller unit is used to host and execute the firmware and in order to communicate with and handle all the other pieces of the system.

The barrier alarm is implemented using two components: a laser emitter module which is aimed at a light-dependent resistor module. While the barrier alarm is active, the laser fires a beam at the photoresistor, which changes its voltage. In this way, when the beam is interrupted, the light variation is detected as a variation in the photoresistor voltage.

The area alarm is implemented using a HC-SR501 PIR motion sensor, signalling detected movements in a surveilled area of 120 degrees, ranging between 3 and 7 meters.

The user configuration is stored on an AT24C256 Eeprom using the I^2C serial protocol for the communication.

The LED that signals the system state is the onboard LED, connected to the MCU using the same GPIO interface as the other sensors and actuators.

The alarm sound is emitted using an active buzzer module: since the tone cannot be altered, the sound notifications are differentiated changing the intervals in which the buzzer is playing a sound or in which it is staying silent.

The different commands are accepted from the user with a 4x4 numeric keypad module. The keypad is connected to the MCU with 8 different GPIO pins which identify the matrix of buttons. The columns are configured as input pins, each with its pull-down resistor, whereas the rows are configured as output pins. More on why this is configured in such a way is discussed below in [2.4.2: Keypad reading procedure].

When a particular event is detected, as detailed in the project specification requirements document, the system logs it using the UART interface; in order to log the date and time in which the event occurred, a DS1307 RTC module is used and it is interfaced to the MCU using the I^{2}C serial protocol, using the same I^{2}C interface as the Eeprom.

All the STM32 board timers are used to perform operations which depend on time without requiring active waits, such as the management of the periods of buzzing, the periodic logging, debouncing mechanisms, and more.

2.1.2 Hardware scheme

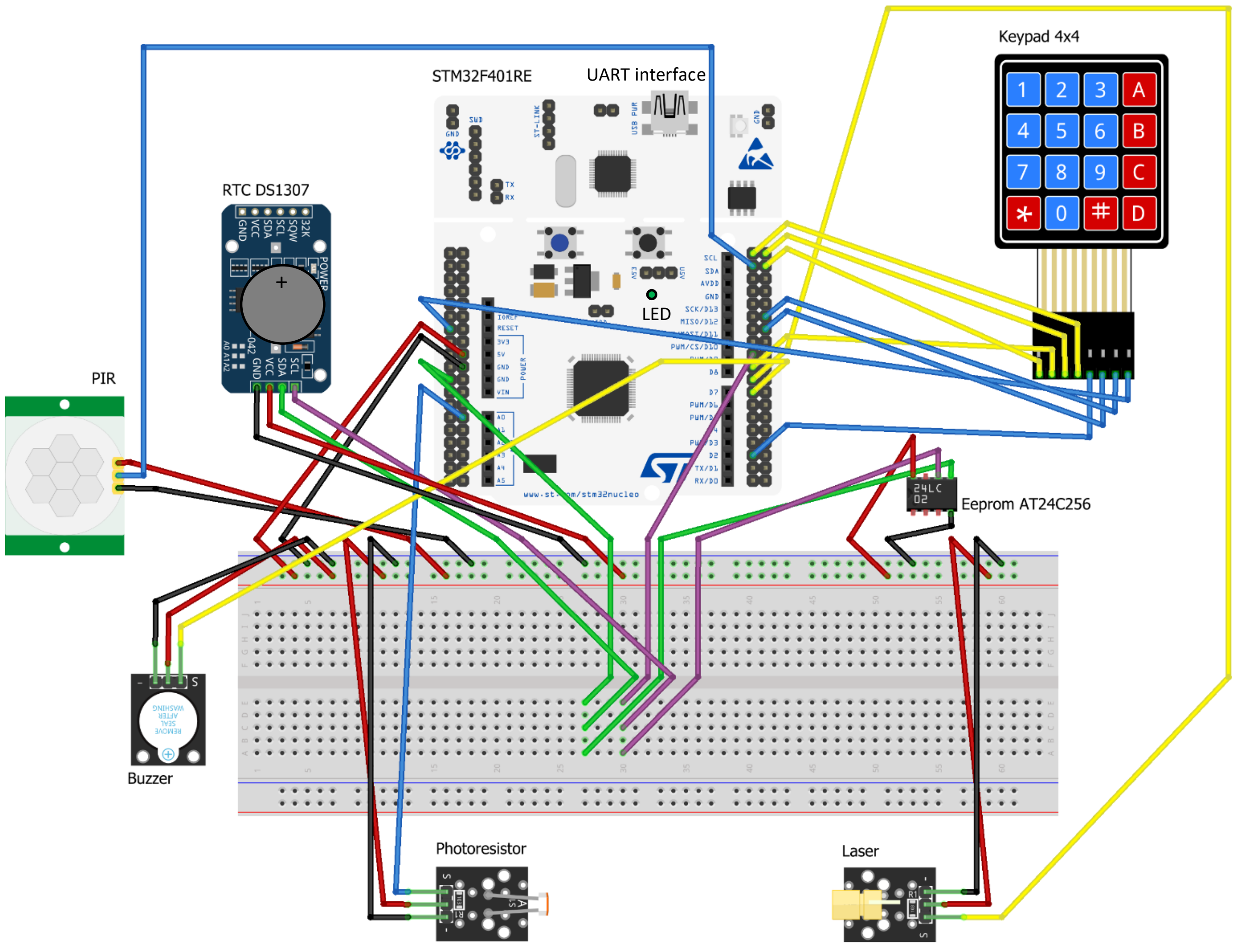

Figure 1 - Hardware architecture scheme

Red	VCC
Black	GND
Blue	GPIO Input
Yellow	GPIO Output
Purple	I^{2}C SCL
Green	I^{2}C SDA

2.2 SOFTWARE ARCHITECTURE

2.2.1 “Object-oriented” approach

The overall software structure is organized in order to separate as much as possible the logically different parts of the project, obtaining an advantage in readability, maintainability, reusability of the single modules, and a reduction in complexity and coupling of the resulting code.

An approach much inspired to what the object-oriented programming paradigm achieves is used: each software module exists if it represents something, such as a sensor or an actuator.

Each module is responsible for its state – statically defined inside the module, has functions to be initialized and destroyed, and, if necessary, getter functions for information to be requested from other modules. It is also responsible for the resources it uses, such as interrupt lines or timers.

2.2.2 Interrupts driven architecture

The whole project is configured to react to the environment using timers and GPIO interrupts: when an interrupt which is owned by a certain module is called, it can access the module state because both the interrupt or timer handler function and the module state are statically defined inside the same file, restricting the scope of the resources. Since the sensors and actuators modules only modify their state, they are completely oblivious and unaware of the other modules.

2.2.3 Controllers

Controller modules are therefore used in order to glue together the structure of the code. For instance, the keypad module reacts asynchronously through interrupts to the user pressing the buttons, and at the end of the button reading passes the character to a callback (more precisely, a function pointer) which then consumes the character. The callback is passed to the keypad module in the initialization phase. In this way, the keypad does not know who uses the characters and in which way. The *command_controller* module consumes characters read from the keypad module, parses the command, and modifies the whole system state according to the received instructions and to the command protocol.

2.2.4 Race conditions

Another important factor in the software architecture is that, since there are concurrent read and writes to shared resources between controller, sensor and actuator modules, race conditions arise. This was accounted for in a very simple way: each read and write is performed during an Interrupt Service Routine, and these were configured to have the same priority of execution. In this way, different interrupts can never stop the execution of another one, effectively achieving complete read and write atomicity and solving all concurrency problems.

2.3 SOFTWARE INTERFACES

Each module exposes an interface which is generally presented in this way: a *module_init* which initializes the internal and private state of the module passing all the resources the module will be considered responsible for, a *module_destroy* which de-initializes a module and makes a new initialization possible (each module is therefore an implementation of the Singleton design pattern). Each module also exposes the callbacks for the handling of its resources which are going to be called in the Hardware Abstraction Layer callbacks. For a more precise description of the interfaces of the individual module, each header file documents its interface thoroughly.

2.4 SOFTWARE PROTOCOLS

2.4.1 Blocking modules

While almost all the modules exposed API is non-blocking, some modules only have blocking functions, because they don’t return immediately to the caller, but rather wait for a particular operation to end. In particular, the logger module – that is, the software part which handles communication via the UART interface in order to log events – exposes a blocking *logger_print* function, waiting for the I/O operation to be concluded.

The function is based on the blocking HAL function *HAL_UART_Transmit* rather than on the corresponding interrupt-based function. The problem with using the non-blocking version lies in guaranteeing that the memory whose pointer passed to the function as an argument is not deallocated or tampered with until the transmission ends.

Two different designs were discussed to guarantee this behaviour: dynamic memory allocation and a global, big enough static data buffer.

Both options would have complicated the system because of needing to handle memory ownership and synchronization between different calls to the transmission primitive. The blocking option is very simple to implement, and empirical evidence shows that the system is capable enough of handling these data transfers without hampering with the functioning of the system.

The RTC module also exposes blocking functions for getting and setting the date on the component: this was made because these two functions are respectively only called in the logger module and in the initialization phase. In the first case, the logger module is already blocking per se, and in the second one, performance is not relevant at all because of the long booting sequence.

2.4.2 Keypad reading procedure

Earlier in the document it is specified that the pins identifying the columns are configured as input pins reacting to rising-edge interrupts and the row pins are configured as output pins.

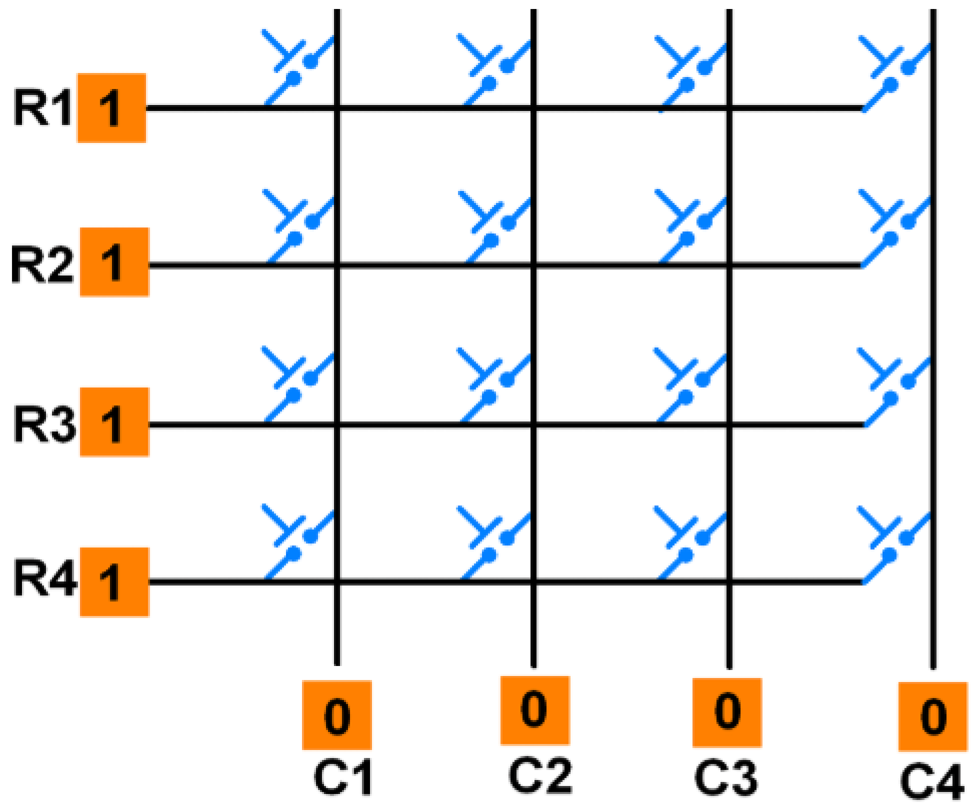

Figure 2 - Internal keypad connections scheme

In the initialization phase, all the row pins are set to a high value; in the meanwhile, the columns read as a low value because they have a floating value but are pulled down by the board resistors.

When a button is pressed, it closes the connection (the blue circuit in Figure 2) among the corresponding row and column. In the current configuration, this makes the whole column emit a high signal level, triggering the interrupt. This is the way a button press is detected on the whole keypad.

Once the button press is detected, it is still needed to identify which row the button belonged to. Before trying to discern the actual pressed row, a simple debouncing technique is applied: a timer is initialized in order to skip the initial instability of the signal, and the interrupt returns.

After a brief interval of time, the timer handler starts the button identification process and the signal is assumed to be stabilized.

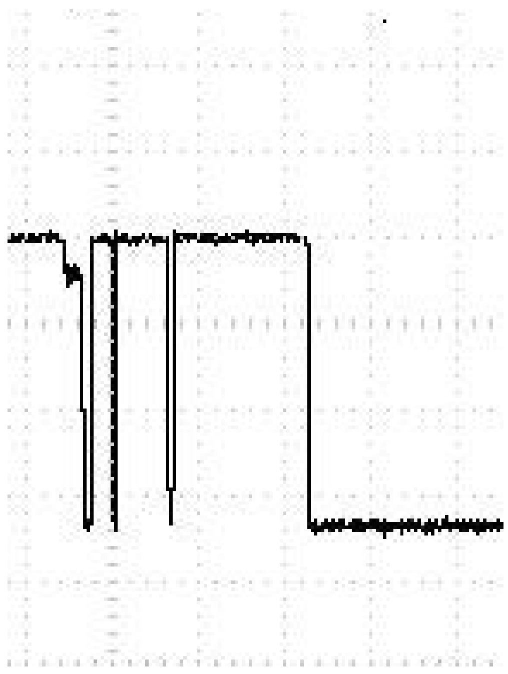

Figure 3 - An example in which the initial instability of the signal is evident.

The timer handler now puts to low all the rows but one and checks if the column which was triggered before is still high. If it is, the row of the pressed button is the current one, and the character is parsed and sent to the *consume_character* callback discussed in [2.2.3: Controllers]. If it is not, the process is iterated over the next row. If no row makes the columns high again, the initial triggering is assumed to be a noisy variation in the signal of the columns.

2.4.3 Alarms delay procedure

When a sensor detects a high event, it starts a timer which triggers after a number of seconds equal to the delay specified in the configuration.

In the meanwhile, if the sensor detects that the high condition disappears, it stops the timer and resets its counter register.

Therefore, if the timer manages to trigger an update event then the signal has been stable for the delay interval. The sensor is then set to an alarmed state and the same timer as before is configured to trigger after a number of seconds equal to the alarm duration specified in the configuration. The state of the sensor is not changed (therefore new alarm events are inhibited) until the timer fires this new interrupt or the user manually disables the alarm through the keypad interface.

2.4.4 Photoresistor ADC conversion

The default ADC interrupt scheme in the STM32 environment can only be set to polling frequencies which are too high for the purposes of this project. To circumvent this problem, a timer is used to manually specify when the ADC conversion must start, guaranteeing a fine control over the conversion polling rate.

The ADC conversion itself is non-blocking: the timer only requests the ADC to start the conversion, while the end of the procedure is handled in a separate interrupt.

2.4.5 Alarm sound controller

The alarm sound that the system must emit is different depending on which combination of states the sensors are in; therefore, a controller module polls the state using a timer and determines which sound must be played by reading the states.

Since the buzzer module is configured to ignore concurrent calls to the *buzzer_play* API, it is not relevant if the controller polls and issues commands to the buzzer too frequently. This makes the period and sound choices simple.

This means that if the sound to be played changes while the buzzer is already playing, the maximum time of inhibition is the duration of the currently played sound. It is therefore advised to use short sounds: they will be alternated or repeated by the controller in function of the alarm states.