University of Salerno

Embedded System Project: Home Security System 19/20

Carmine Napolano ID: 0622701202

Emilio Sorrentino ID: 0622701199

Francesco Rosa ID: 0622701095

Summary

1. Problem Description

The goal of this work is to design and implement a *"Home Security System"* based on the following requirements:

- Two different kind of alarms: AREA and BARRIER, for detecting movements and obstacles;
- Drive a buzzer for notifications and alarms (different sound for different alarm);
- Allow the user to insert its own system configuration, that is composed by the following parameters:
    a. USER PIN, used for inserting commands, composed by 4 numbers;
    b. Alarms delay, each alarm can be configured with a different delay, that is the elapsed time between the signal detection and the effective emission of the alarm, a suitable value belongs to the range [0, 30] seconds;
    c. Alarm duration, the system has one alarm duration that represents the amount of time during which the alarm sound must be emitted, a suitable value belongs to the range [5, 60] seconds;
    d. Current date and time.

  The configuration can be inserted only at boot time, and in a time window of 30 seconds. If the user can't insert the custom configuration within 30 seconds the default configuration is loaded.
    - USER PIN, [0 0 0 0];
    - Sensors delay, 0 s;
    - Alarm duration, 5 s;
    - Current date and time;
- Allow the user to insert commands in order to:
    a. Activate/Deactivate the system

   b. Activate/Deactivate both or a single alarm
   c. Close an active alarm

- Send, periodically (each 10 seconds), a system log message, in order to share the system state in a particular moment, the message must be composed by:
   a. Current date time, in the format: [dd-mm-yyyy hh:mm:ss];
   b. Alarms states, in the format: Area [STATE] – Barrier [STATE], where STATE can be {Inactive, Active, Alarmed};
- Before the sensor gets alarmed the detected signal must be stable for at least 1 second;
- The sound is emitted when the system is alarmed, and any delays are elapsed;
- The user led must be on when the system boot and when the system isn't active;
- The user led must toggle when the system is active;

2. Problem Solution and Design choices

In this chapter we'll discuss about how the problems, presented in the previous chapter, have been solved. Based on the problem requirements we have:

A. The two kind of alarms required have been implemented by using:

- PIR sensor, for implementing the Area Alarm, since this module is able to detect motion inside an area;
- Laser combined with a photoresistor, for implementing the Barrier Alarm, the idea is that when an obstacle is between the laser and the photoresistor, the system should be able to detect a decreasing in brightness intensity by measuring voltage signal generated by the photoresistor and compare the read value with a threshold that should represent the value in normal condition, that is when the laser is directly pointed on the photoresistor;

B. Different alarm sounds

Since the given buzzer is an active one, so it's impossible to emit different ringtones,

in order to implement the different alarm sounds requirement, we have chosen to drive the buzzer with a PWM signal generated by a timer.

In this way, what we change is the duty cycle assigned to each alarm sound, so we change the time during which the buzzer is "*on*" and emits its buzz.

C. System Configuration

In order to give to the user, the possibility to insert its own configuration, we have decided to interface the board with the PC through the UART interface, in particular the user can communicate the configuration to insert into the system by using a software that implements different serial protocol, called Putty. The application protocol followed by the system and the user will be explained in detail in the next chapters.

The time window given to the user for inserting the configuration is about 30 seconds, if during this time, the configuration inserted by the user is correct, a *"System Configuration Loaded"* message will be sent, otherwise if the time elapsed or the user insert something wrong a *"System Configuration Rejected"* message will be sent and the default system configuration will be load.

If the user wishes to send its own configuration, he must reboot the system.

D. User commands

We decided to use a Keypad, to give to the user the possibility to send commands to the system.

The keypad module and the protocol used to send commands is discussed in the following chapters.

About commands, We have been taken the following decisions:

- A sensor cannot be activated if the system is not active;
- If the two sensors are both alarmed, the only command accepted is *"Deactivate both"* or *"Deactivate System"*.

- If the user sends a command that represents a state already met, the command is rejected.

The result of the command inserted is going to be sent over UART. The system will be sent different messages for different results. The messages are:

- *"WRONG USER PIN"*, if the user inserts the wrong pin;
- *"COMMAND ACCEPTED"*, if the command has been executed successfully;
- *"COMMAND REJECTED"*, if the user pin is correct but the command is wrong or doesn't match the actual system state (see the specification above).

The system confirms the operation through a short sound using the buzzer.

We decided to emit the short sound only when a command is accepted, and the buzzer is not busy to emit a sound.

E. System log message

To send each 10 seconds the system state to the user, it has been decided to use the UART interface, from the board to the PC, for sending the messages defined in the previous chapter.

The application protocol will be explained in the next chapters.

F. Check stability signal

Regarding this particular problem, we have to make different discussions for the two modules.

About the PIR module, the idea is that when a movement is detected, the system catches the transaction from 0 to 1 of the PIR's output signal, and it starts a timer for 1 second. If PIR's output signal goes from 1 to 0 (transaction caught by the system) before the time elapsed, the sensor is not alarmed otherwise it is.

Instead, for the barrier sensor, the argument is different, because this module is composed by two different components: the laser and the photoresistor. The idea is that the system reads periodically the analog value given by the photoresistor. This operation is performed by using what is called ADC (Analog to Digital Converter), that is a timed board's peripheral. In particular, we are talking about Sampling Frequency and Conversion Time, so, by knowing the time required for each new conversion, that is

$$t_c = Sampling\ time + Processing\ time$$

we can compute the number of conversions that are produced in a fixed time

$$\#conversions\ in\ fixed\ time = \frac{time}{t_c}$$

If we count $\#conversions\ in\ fixed\ time$ samples over the set threshold we know that the signal has been over the threshold for the given *time*, and the sensor can be alarmed. The threshold is set during the initialization phase, and it is equal to:

$$threshold = \frac{Dark\ Read + Light\ Read}{2}$$

where:

- Dark Read, is a reading of the sensor output when the laser is off;
- Light Read, is a reading of the sensor output when the laser is on, and it aims the photoresistor.

G. Alarm System

When a sensor is alarmed, then it communicates its state to the system, in this way based on the system state itself different actions are performed:

- If the system is active, and it receives a notification by an alarmed sensor, then the system checks the sensor delay. If it is different from 0, then system starts a timer for the sensor delay. When this time elapses, if the sensor is still alarmed (it has not been deactivated by the user) the system

becomes alarmed and the sound is emitted. Otherwise, if the sensor delay is 0, the system is immediately alarmed and the sound is emitted;

- If during the delay of an alarmed sensor, another sensor goes from Active to Alarmed, then the system waits for the delay of the first alarmed sensor and at the end of it, if both the alarms are alarmed then the sound emitted will be that one for both;
- While the system is emitting an alarm, another sensor becomes alarmed, then the sound emitted is immediately configured with that one for both and the duration timer is restarted.

H. System led, about this requirement we have decided to use the user led embedded on the board. The behaviour of this led is the following:

- When the system boot, it is turned on;
- When the user activates the system, the led starts blinking;
- When the system gets alarmed or inactive, the system led stops blinking and set its state to *"on"*.

3. Hardware Architecture

In this chapter the *"Hardware Architecture"* will be presented. In particular we'll talk about how the different components has been connected to the board and how the board's resources have been allocated.

Connection Scheme

We start from the final scheme, that is shown in *Figure 1*.

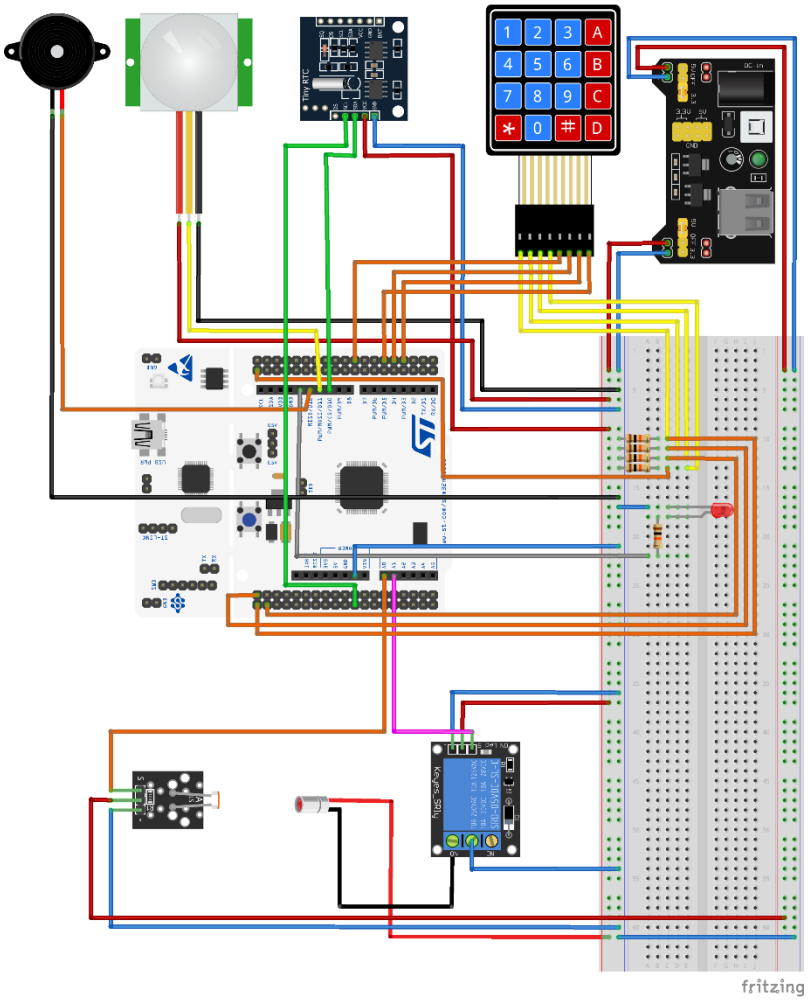

Figure 1: Hardware Scheme

Starting from the high left corner we can see the following components:

- **Active Buzzer**, this is a component that is able to emit a sound as soon as it is supplied by a voltage of 3.3-5V_{dc}, since it has an internal oscillator that makes possible the described behavior. As we can see the component has the signal pin connected to the

PA6 pin of the board, that is the same pin connected to the Channel 1 of the Timer 3, this means that, after a suitable configuration of both timer and channel, we can command the buzzer with a particular signal generated by the timer, that makes possible obtain “different kind of alarm sounds”, the detail of this implementation will be discussed after.

- **HC-SR501 Passive Infrared (PIR) Motion Sensor,** this is a particular kind of sensor that is able to detect movements by detecting variations in the surrounding infrared field. The component is passive because it doesn’t emit any infrared field, but it has a particular sensor that is sensible to these light signals, this means that when a subject, that emits infrared rays (like a human or an animal) passes in proximity of the sensor, it detects those rays and the module generates an output voltage signal with a value of 3.3V.

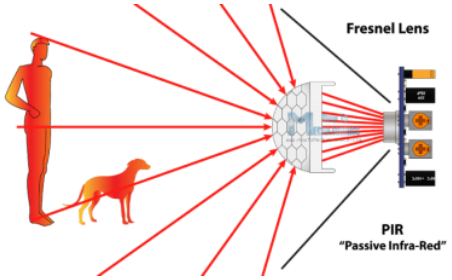

Figure 2: PIR Sensor

This output signal is given in input to the board, through the PA7 pin, that is configured as digital input able to raise interrupt when a rising or falling edge in the signal is detected (discussed in detail after).

The module allows the user to configure the following parameters:

  - Output timing, from 3 seconds to 5 minutes, the parameter represents the amount of time during which the output signal is high after a movement has been detected;
  - Sensitivity, from 3 meters to 5 meters, represents the spatial range in which a movement is detected;
  - Operation mode, that can be:
    - Single trigger mode, the signal is held high for the selected output timing,

then it goes down;

- Repeatable trigger mode, the signal is held high until a motion is detected.

The chosen configuration is:

- Time delay set to minimum value;
- Sensitivity set to middle value;
- Operation mode set to Repeatable trigger mode, in this way we are sure that the signal is high until a movement is detected and we can check the signal stability.

- **DS1307 RTC (Real Time Clock) module,** this is a particular component that is able to keep the date and time. The component allows the user to set a new date and time and to read the updated values. The reading and writing operation are performed according to the I2C serial communication protocol. At this scope, we have connected the SDA and the SCL module's pin to the corresponding pins on the board, in particular the STM32 board makes available three I2C peripherals, in our case we have chosen the I2C1, so we have connected the data and clock module's lines to the PB7 (SDA) and PB6 (SCL) of the board;
- **Keypad 4x4,** this component is simply a matrix of buttons. When a button is pressed, we short a particular row and column, so in order to understand which button is pressed the system must be able to detect the row and the column that are shorted.

  In our case We have the rows as input, connected to the board through a pull-down configuration. In this way when all the buttons are released we read a logic value of 0 (0 V), instead the columns are connected to GPIO configured as digital output, where the logic value is set to 1 (3.3 V).

  When a button is pressed, the row corresponding to the pressed button goes from 0 to

1, the rising edge is detected and the following procedure is executed:

  - All the columns are set to 0;
  - The columns are set to 1, one by one, until a change in the pressed row is detected;
  - When a change in the pressed row is detected, we know the row and the column of the pressed button, this means that we know the position of the button pressed in the button matrix and consequently its value.

- **External Power Supply,** this is a module that takes in input 12 V_{DC} and can generates two power lines of 5 V_{DC} and 3.3 V_{DC}. We have used this module to supply all the components, so the only duty of the board is to command and control them.
- **User LED**, it allows the user to know the system state. Its behaviour is the following:
  - When the system is off, also the LED is off;
  - When the system isn't active, it's inactive or alarmed, the LED is turn-off;
  - When the system is active, the LED toggles periodically, each 1 second.
- **LASER and PHOTORESISTOR,** these two components have been used for implementing the Barrier Sensor. The LASER status (on, off) is imposed by the board through the Relay, in this way we have decoupled the control system and the power unit.

  About the PHOTORESISTOR, it's an analogic sensor, in particular it doesn't give as output a sequence of bits, but it gives a voltage that is proportional to the brightness.

  The higher is the brightness, the lower is the voltage between the sensor pins.

  From the figure, we can see that the sensor isn't connected to the 5 V line, but it's connected to the 3.3 V line, in this way the sensor will generate a voltage in the range from 0 V (infinite brightness) to 3.3 V (completely dark), that is the range in which the ADC of the board works. How this ADC has been used, will be explained in the next

section.

Board Resources

Based on the requirements and the kind of component used, we have allocated on the board the following resources:

- **TIMERS,** about the timers, in this project we need to count different time in different situations and, moreover, in different modes.

  It is useful define the system clock frequency, that in our case is $f_{system} = 16\ MHz$

  Starting from the simplest ones, we have allocated:

  - TIM10, this timer is in charge to count the amount of time available for the user for inserting the configuration (30 seconds) and the time that periodically runs the system log procedure (10 seconds). In this case we have a single resource, used for two different scopes. We note that, the configuration procedure is executed only one time at boot time, so we do not need to recount this time after a configuration was loaded. To achieve this goal, we have set the TIM10, with the following parameters:
    - $Prescaler = 15999;$
    - $Counter\ period\ = 0.$

    When the system boots the *Counter period* is set to 29999, we obtain an update event with a frequency of

    $f_{update} = \frac{f_{system}}{(Prescaler+1)*(Counter\ period+1)} = \frac{1}{30} = 0.3\ Hz$, that means an update event any 30 seconds.

    After this time elapsed, the TIM10 is stopped, and when the system log procedure is started by the system, the *Counter period* is set to 9999, we obtain an update event with a frequency of:

$f_{update} = \frac{f_{system}}{(Prescaler+1)*(Counter\ period+1)} = \frac{1}{10} = 0.1\ Hz$, that means an update event each 10 seconds.

- TIM11, this timer is in charge of counting two kind of times. The first one is related to the sensor delay, instead, the second one is related to the duration of an alarm. This timer has been set with the following parameters:
  - $Prescaler = 15999;$
  - $Counter\ period\ = 0.$

  When a sensor becomes alarmed, it communicates its state to the system, that checks if the timer is already busy, if it isn't, the system sets the *Counter period* to $(Sensor\ delay * 1000) - 1.$

  When this time elapses and the sensor is still alarmed, the alarm sound must be emitted for a time equal to *Duration*, so in this situation, the system set the *Counter period* to $(Duration * 1000) - 1.$
- TIM1, this timer is a General-Purpose Time, that has been used for a simple scope. The duty of this timer is to count the stability signal time for the PIR sensor. Based on the requirements, a sensor, before to get alarmed, must keep its signal stable for 1 second, in order to avoid false activations. In this case, since this timer is fixed, we have set it with the following parameters:
  - $Prescaler = 15999;$
  - $Counter\ period\ = 999.$

  In this way, when the timer is started, it generates an update event after 1 second. We note that, the PIR allows the user to set an Output timing, that means how long time the output signal must be high after a movement has been detected.

  We **note** that the minimum output timing is equal to 3 seconds, this means that, based on the given sensor, the requirement for the signal stability is

automatically met, in a sense that the TIM1 may be useless. We have decided to keep it, and implements the requirement, because, in this way the software will be ready for a hardware update with a better sensor.

- TIM2, this is General-Purpose Time, that is in charge of timing the user led.

  It has been set with the following parameters:

  - $Prescaler = 15999;$
  - $Counter\ period\ = 999.$

  In this way, the timer generates an update event each 1 second after its starting. The timer is started when the user actives the system, and it is stopped when the user disables the system or an alarm is emitted.

- TIM3, this is a General-Purpose Time, that is in charge of timing the buzzer.

  In order to emit different sounds for different alarms, we have decided to drive the buzzer through a PWM signal. In particular, we have set the timer with the following parameters:

  - $Prescaler = 15999;$
  - $Counter\ period\ = 999;$
  - $Pulse = 0.$

  In this way, the PWM signal has a period of 1 s.

  The system changes the Pulse of the PWM signal based on which sensor raised the alarm, in this way what we change is the amount of time during which the buzzer is on and emits the sound.

  Moreover, the buzzer is used even when a command is accepted and executed. The behaviour of the buzzer is the same, but in this case, the buzzer has to perform only one buzz, so, in this case we handle the PulseFinishedCallback where we deactivate the buzzer after a pulse is performed.

- **Other peripherals**, about the other peripheral used, we only list them and perform a brief description about their roles, because the behaviour will be explained in detail in the next section.
    - **NVIC**, that is the peripheral that handles the interrupt events and communicates to the CPU which ISR run.
    - **DMA**, this peripheral has been used during the communication between the RTC and the board, in particular when the system requires the date and time to the RTC during the system log procedure, in order to free the CPU by the duty of transfer data between the peripheral and the local buffer.
    - **I2C**, this peripheral has been used in order to perform the communication between the board and the RTC, since the second one implements this serial communication protocol;
    - **UART**, this peripheral has been used in order to perform the communication between the board and the PC, when the configuration and the system log procedure must be performed.
    - **ADC**, this peripheral can perform an analog to digital conversion, useful to interface the MCU and the photoresistor. Thanks to this peripheral the MCU can read and elaborate the photoresistor level voltage, that gives information about the brightness level.

4. Software Architecture

In this chapter we'll discuss about the software architecture, starting from the general idea, then presenting all the different modules and what they do and how they communicate.

Figure 3: Event Driven Architecure

The paradigm followed during the software design phase was the event driven paradigm.

In this way the MCU can perform actions in response to external event, that are purely asynchronous with respect to the program execution. That is our case, because we don't have any control over when a sensor may raise an alarm and the system must respond immediately at such events.

Moreover, for the same reason, we don't know when a user can insert a command, so thinking to an infinite loop architecture, where we periodically scan the keypad state, can lead to losing some pressed buttons.

In this paradigm at each external event is associated an ISR (Interrupt Service Routine) that has the duty to implement the code that the CPU must execute in response to the event. The problems that this software architecture raises, such as concurrency problem and priority one, will be discussed after, when the implemented modules will be presented.

Now, we'll present the system modules.

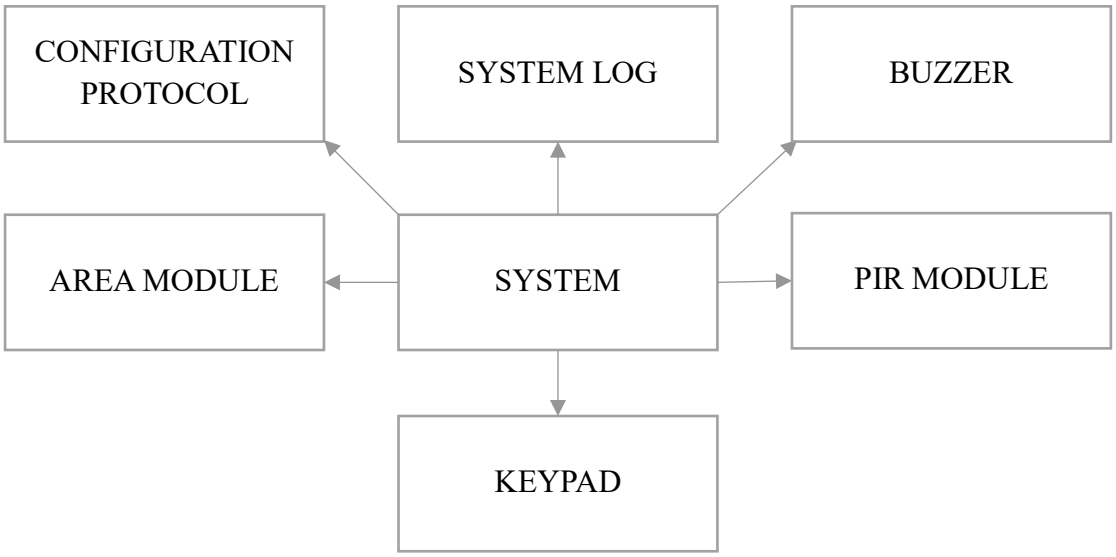

Figure 4: High Level System Description

From the Figure 4, we can see that the *System Module* is the core of the whole system.

It has several duties, from which:

- Initialize all components;
- Execute commands inserted by the user;
- Receive notification from the sensors and perform the suitable action based on the system state and configuration.

In general, the system behaviour can be resumed into two different phases:

- Initialization phase;
- Execution phase;

We can better explain the other system modules by analysing in detail those two phases.

Initialization phase

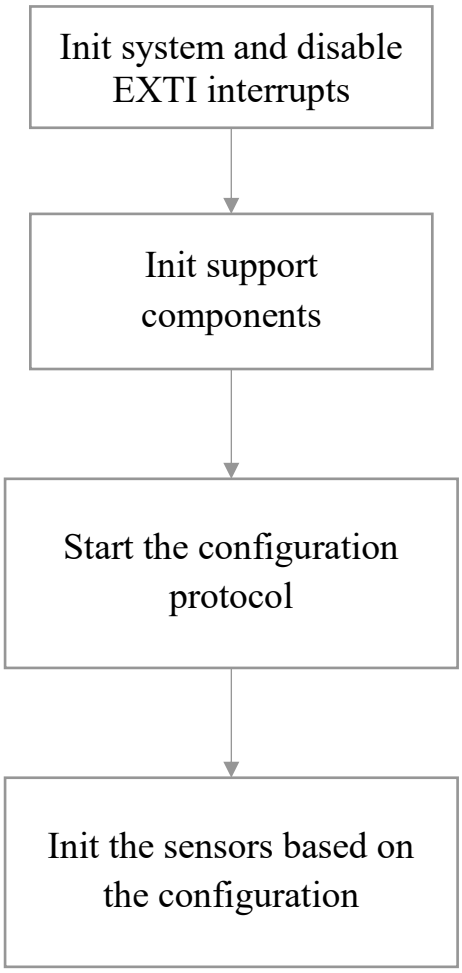

Figure 5: Initialization Phase

This is the phase that is executed when the system boots, and the initialization function is called.

This phase can be synthetized by the scheme in Figure 5.

The initialization phase is composed by 3 big steps:

1. **Init Support Module.** During this step, the system declares and initializes all the components needed for the *System Log module* and for the *Configuration Protocol module.*

In detail the system initializes:

- **UART_handler module**, by passing to it the UART peripheral reference that the system uses to communicate with the PC. This module wraps the HAL functions for writing and reading over the UART interface. In particular, it shows functions that allow

communication in both non-blocking and blocking mode. The non-blocking mode is performed through interrupt.

- **DS1307_RTC module,** by passing to it the I2C peripheral reference that the system uses for communicating with the RTC module. This module implements all the functions that are useful for the communication between the board and the RTC. In detail, this module contains a "private" structure, *Figure 6*, that represents the system date and time, and implements functions that can be used for update this structure by reading the current date and time from the RTC and for writing into the RTC memory the date time structure itself. The *Configuration Protocol module* and *System Log module* have a reference to this module, because they need to take the date and time stored by the RTC, moreover the first one has the need to write a custom date and time structure if the user gives his date and time during the configuration phase.

```
struct date_time_s
{
    uint8_t     seconds; // [0-59]
    uint8_t     minutes; // [0-59]
    uint8_t     hours; //[1-12] or [00-23]
    uint8_t     day; // [1-7]
    uint8_t     date; // [1-31]
    uint8_t     month; // [1-12]
    uint8_t    year; //[00-99]
};
```

Figure 6: Date_time structure

- **System log module**, by passing to it a reference to the RTC and UART_handler modules, defined before, and to the TIM10 peripheral handler. This module implements all the functions that are required for running the system log procedure. Moreover, it implements some funcionts, visible to other modules, that allow the caller to write or read messages through the UART peripheral used by the system, and to start/stop the system log procedure.

- **Configuration protocol module,** by passing to it a reference to the RTC module,to the TIM10 peripheral handler and to the system configuration structure. All the system configuration parameters are saved into this structure. This module implements the application protocol for the configuration, and shows to the external module a function that allows to start the protocol itself.

```
struct system_configuration_s{

    user_pin pin[PIN_SIZE];
    sensor_delay sensor_delay_1;
    sensor_delay sensor_delay_2;
    allarm_duration duration;

};
```

Figure 7: System Configuration Structure

2. **Start the configuration protocol,** the system after initialized the components, starts the configuration protocol, by calling the appropriate function implemented by the *Configuration Protocol Module*. This procedure is a blocking one, this means that the system doesn't perform any other action before the procedure will end.

   In this way when the control is given back to the system, it knows that the system configuration structure has been initialized with either custom configuration or default configuration, and can starts the third step.

3. **Init sensors based on the configuration,** during this phase all the sensors and the buzzer are initialized. In detail the system initializes:

   - **PIR Module,** by giving references to the digital pin sensor structure, that contains all the information about the port and the pin where the module is connected, to the TIM1 peripheral handler, used for counting the signal stability time.

     Moreover, the module requires the values of its initial state, set by the system to INACTIVE, the delay configured for the AREA sensor, the pulse value, that characterizes the Area alarm sound. The module implements all the function for set and get its state. The module does not implement any particular behaviour because, any change on its state produces an interrupt that is handled by the EXTI Callback redefined into the *System module.*

- **Laser module,** by giving it a reference to the GPIO_Typedef and the value of the pin, where it's connected, and its initial state. This module implements the functions for set/reset the laser state.
- **Photoresistor,** by giving it a reference to the ADC peripheral handler, that must be used to convert the analog value to a digital one. This module implements some functions that allow the caller to read a single value in polling mode or to read a sequence of value in interrupt mode and to stop the sequence itself.
- **Barrier module** by giving it a reference to the laser and the photoresistor, initialized before, and by giving the value of its initial state, set by the system to INACTIVE, the delay set for the Barrier sensor, the pulse related to the barrier alarm sound and the number of conversions that must be over the threshold in order to consider the signal stable.

  This module implements all the function for set e get its state, it handles the ADC Conversion Completed Callback and it implements functions visible to other modules that allow the caller to start and stop the module, in the sense that to start and stop the ADC sequential interrupt mode used for getting the sensor values.

  We note that when the barrier sensor is initialized an internal threshold value is set. This threshold is set based on the formula defined in the previous chapter.
- **Buzzer module**, by giving it the value of its initial state and the reference to the TIM3 peripheral handler. This module implements the buzzer activation and deactivation functions. The activation is performed by giving in input the value of the pulse that must be set into the TIM3 CCR1 register.

Operative Phase

Once the initialization phase is end, the system can run its operative phase.

This phase starts by calling a function called *run_system()*, that essentially performs the following actions:

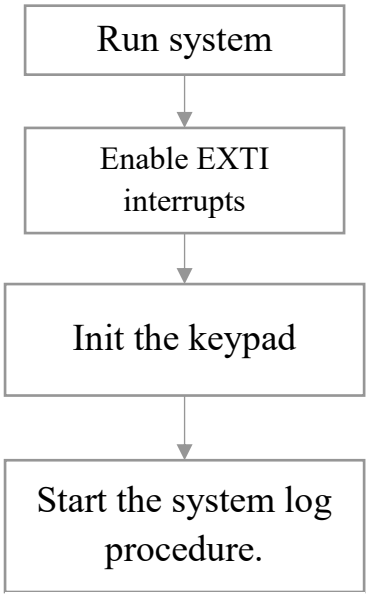

Figure 8: Run System Procedure

- **Enables EXTI interrupts**, previously disabled to avoid calls to the EXTI Callback during the configuration procedure.
- **Init the keypad,** we set the pin connected to the columns of the keypad to a high logic value, through the *Keypad module*, that in our case correspond to 3.3 V.

  The *Keypad module* implements the procedure, explained in the previous sections, for getting the value of the pressed button. Moreover, we have an additional module, called *Keypad handler*, whose scope is to implement functions related to the specific application, such as check the user pin.
- **Start the system log procedure**, this is the last "sequential" operation performed by the system, where the system commands the *system log module* to start the log procedure, so it starts the timer associated with the modules.

From this point the system is waiting for external events, in particular, the system is initialized as INACTIVE, so the first command that a user should send must be "*Activate System" command*, otherwise he isn't able to activate the sensors.

During this *"asynchronous phase"*, the system is waiting for receiving commands and notifications by the sensors.

In particular, the commands are handled and executed through a procedure that will be explained in the next section, now we can say that the *system module* redefines the EXTI Callback, that is called when a rising edge of the keypad rows is detected.

About the sensors, we can say that, they communicate their states in different modes:

- PIR, since it's a simple digital sensor, we have decided to set the input pin to raise an interrupt when a rising or falling edge is detected. When the EXTI Callback is called, the system checks the state of the PIR, so, if a rising edge has been detected, the system checks if the sensor is *Active* or not, in the first case, the system starts the timer related to the sensor, otherwise it doesn't perform any action. Instead, if a falling edge has been detected, the system checks if the sensor isn't *Alarmed* (TIM1 hasn't raised the update event), in this case the system stops the timer and the PIR state doesn't change.

  If the TIM1 raises its update event (this means that the PIR's output signal hasn't changed during the 1 second), then, in the Update Event Callback, the function *alarm_pir* is called. This function, implemented by the *system module,* has the duty to set the PIR state to *Alarmed*, and eventually set the timer delay.
- Barrier, in this case, a different procedure has been used, since the different nature of the sensor. In particular, the *Barrier module* redefines the ADC Conversion Complete Callback. When this function is called, means that a new raw value has been produced, in this case, the module reads the raw value and compare it with the threshold. We have two cases:
  - The raw value is higher than the threshold, in this case a counter is increased;
  - The raw value is less than the threshold, in this case the counter is reset if it's different from 0.

  If the counter is greater than $\#conversions\ in\ fixed\ time$ threshold, then the *alarm_barrier* function is called, and the system changes the barrier module state and sets the timer for counting the delay time associated with the sensor.

The system becomes *Alarmed*, when the function *alarm_system* is called. This function can be called in two different cases:

- The delay time, associated with a sensor, elapses, in this case the function is called by the Update Event Callback;
- The delay time, associated with a sensor, is equal to 0, in this case the system is immediately alarmed.

The function above, has the duty to set the system state to *Alarmed,* and to set:

- The timer that must count the *Duration time;*
- The buzzer with the pulse related to the alarmed sensor/s;
- Turn on the system led.

When the *Duration time* is elapsed or when a deactivation command is sent, then the following functions may be called:

- *dealarm_system*, called when *Duration time* is elapsed, this function deactivates the buzzer, and reset the system state to *Active,* by calling *activate_system*, and reset the state/s of the alarmed sensor/s to *Active;*
- *deactivate_module_pir*, set the PIR state to *Inactive* and deactivates the buzzer if the previous state was *Alarmed;*
- *deactivate_module_barrier,* set the Barrier state to *Inactive* and deactivates the buzzer if the previous state was *Alarmed;*
- *deactivate_both,* called when the *"Deactivate Both"* command is sent, it calls the two above functions.

At the end, when an activation command is sent, the following system functions may be called:

- *activate_system,* set the system state to *Active,* and starts the TIM1 that counts the blinking led period;
- *activate_module_pir,* checks if the system state is active, if it's, sets the PIR state to *Active;*
- *activate_module_barrier,* checks if the system state is active, if it's, sets the barrier state

to *Active;*

5. Software Protocols

In this section we will explain the different procedures that leads the system to a new possible state.

In this section we will answer to the following questions:

- How the system configuration is performed;
- How a command is sent;
- How the log message is sent each 10 seconds.

How the system configuration is performed.

The goal of this procedure is to allow the user to insert its own configuration within a time window of 30 seconds, otherwise a default configuration will be loaded.

The board and the PC communicate with each other by the UART interface and the user can insert the parameters through a program called Putty.

The procedure is started by the system, by calling the function proposed by the *Configuration Protocol,* called *configuration_protocol.*

After the *"System boot"* message has been sent, the timer is started and the steps in *Figure 9* are performed.

We note that each transmission and each reception is executed in interrupt mode, and the protocol status is handled by the module function called *protocol_callback_tx()* and *protocol_callback_rx*(), called by the default TX Completed Callback and RX Completed Callback.

About this protocol we have 2 different possible results:

1. The user inserts all the parameters within the 30 seconds, in this case the timer is stopped, and the inserted parameters are checked. The module controls if all the parameters are good. If they are the custom configuration is loaded and the *"System*

Configuration Accepted" message will be sent, otherwise the *"System Configuration Rejected"* message will be sent, and the custom configuration will be loaded;

2. The user doesn't insert all the parameters within the 30 seconds, in this case the TIM10 raises an update event, the Period Elapsed Callback is handled by the *timer_handler* module, that aborts all the UART communication, and sets the protocol state to *"End_Default"*. The configuration module loads the default configuration, and the *"System Configuration Rejected"* message will be sent.

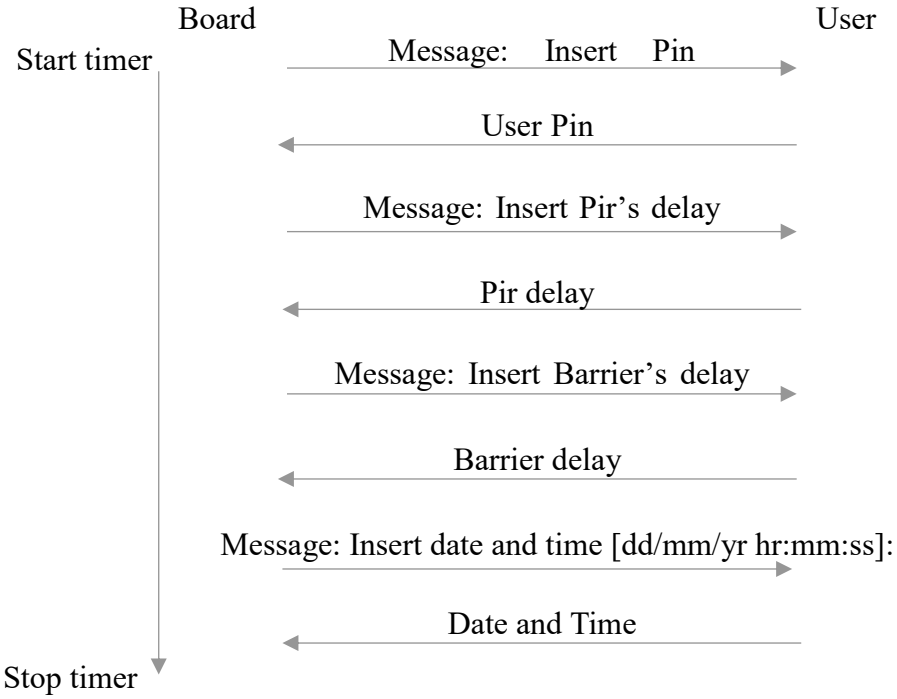

Figure 9: System Configuration Procedure

At the end of the protocol above, the RTC module is called. The way how the module is called changes based on the situation:

1. The module is loading the custom configuration, in this case, it writes the RTC's date_time structure with the given value, and performs a request to write these values into the peripheral;
2. The module is loading the default configuration, in this case, it requests to update the

internal RTC's date_time structure with the value stored into the peripheral.

How a command is sent

In the previous chapters we explained how a single button is handled.

We explained that the keypad has been handled in interrupt mode:

- The columns were connected to digital output pins;
- The rows were connected to digital input pins, with the following configuration:
    - Pull-down;
    - Rising edge interrupt mode.

When the user presses a button, the corresponding row level goes from 0 to 1, this rising edge is detected by the board and the ISR is executed. In particular, the *system module,* redefines the EXTI Callback, where the first step is to control which is the pressed button by user. This procedure is implemented by the Keypad module. It takes in input the pin of the GPIO that has raised the interrupt. The procedure involves the following steps:

- Check the signal stability;
- Set all the columns to 0, this leads the row level pressed to 0;
- Set each column to 1, one by one, until a change in the corresponding row is detected;
- When the row level goes from 0 to 1, due to a change in the column level, this means that the pressed button is located at index [pressed row, columns that generates a change]. If any change is detected the reading procedure returns *"No button pressed*".

The procedure followed by the system, about the commands, is the one represented in *Figure 9.*

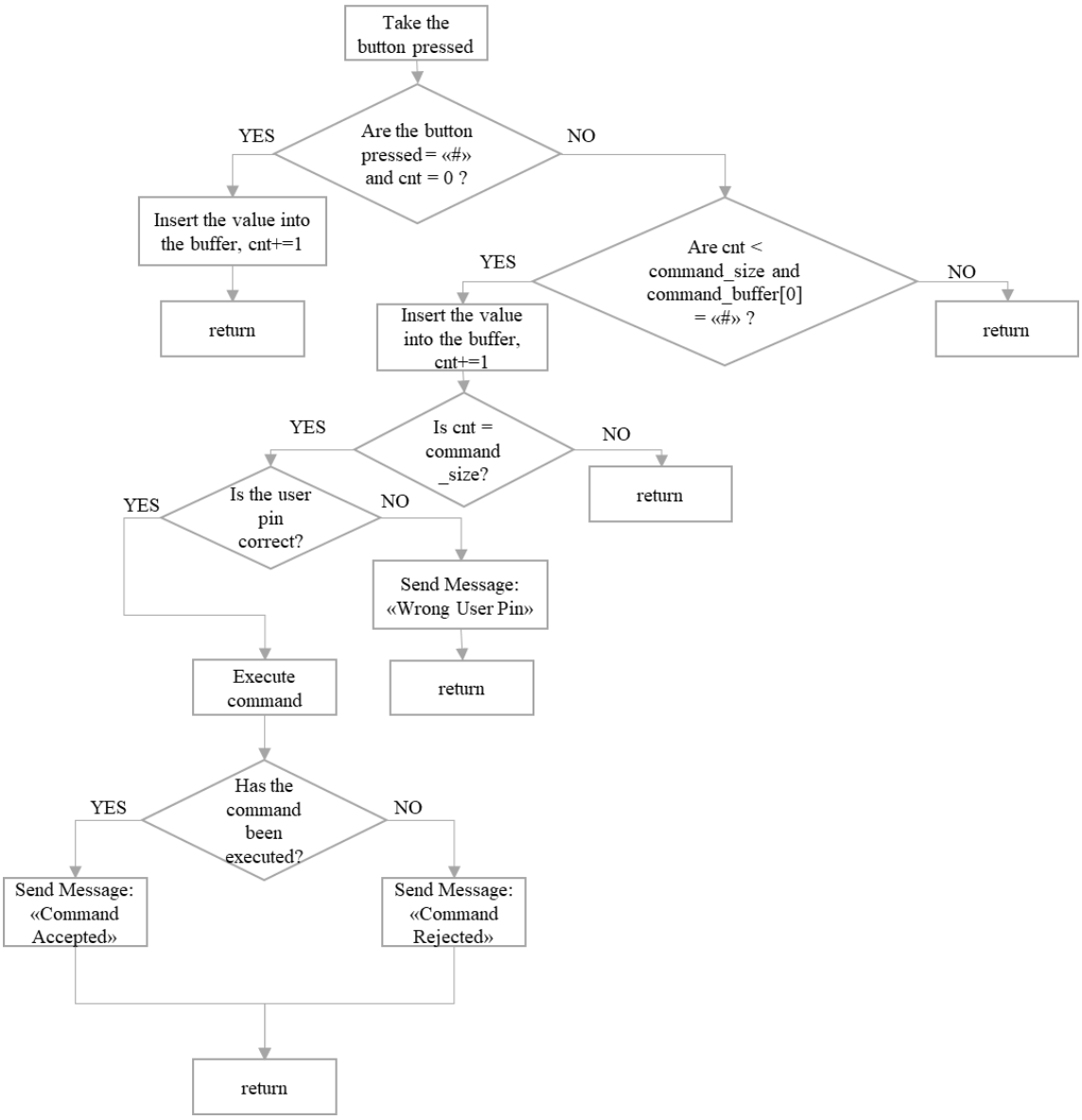

Figure 10: Command Procedure

How the log message is sent each 10 seconds

Previously, we said that the system log procedure is taken over the UART interface. The goal of this procedure is to communicate the system state by sending a message with a specific format each 10 seconds.

The message format is: [dd-mm-yyyy hr:mm:ss] Area [Area State]– Barrier [Barrier State].

To send this message periodically, we have used a timer, in particular we have used the TIM10,

with the configuration explained previously.

With that configuration, the TIM10 raises an update event, each 10 seconds, this interrupt is handled by the Period Elapsed Callback, that the *timer_handler* module redefined.

When this interrupt is handled, the *start_send_log_message* function is called.

This function performs the first protocol steps, that involves in making a request for update the RTC date and time structure in DMA mode.

When the data transfer between the I2C peripheral and the buffer is completed, the *RTC module* converts the data received and updates the date_time structure with those converted data. At the end, the *log_callback_tx* function is called.

This function implements the other procedure steps, that can be reassumed by the *Figure 11*.

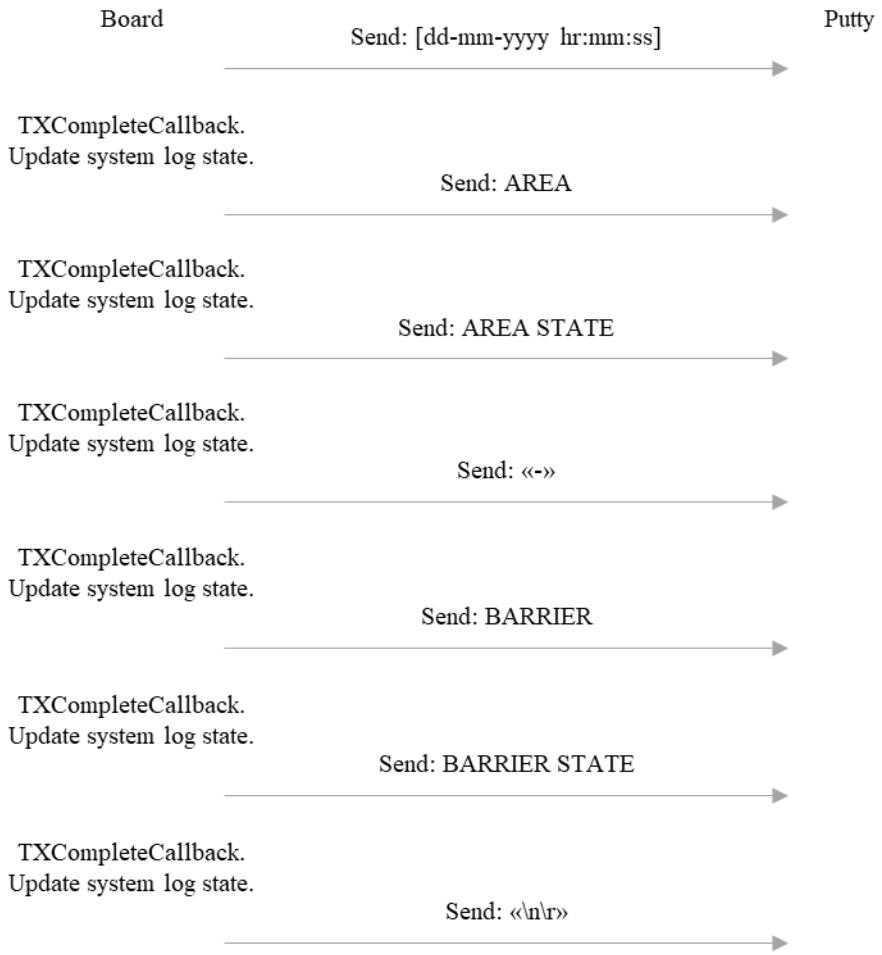

Figure 11: System log procedure