## **Index**

# Project Introduction

The main objective of the next epigraphs will be to expose our work carried out in the last quarter. The embedded microcontroller that has allowed us to achieve the desired functionality is the ARM processor LPC1768, with a complete set of peripherals (To be exhibited soon) and USB communication. Finally, its programming and debugging have been carried out through the integrated development environment Keil Uvision, specifically with the fifth version available.

Once our main work tool used is known, it is interesting to expose first what is the final functionality that we have pursued throughout the whole project development process. The final implementation consists of an ultrasound capable of detecting the distance between him and objects that are around him. This will be done in a horizontal plane with a range of 180 degrees, and its movement will have two operational modes: one manual mode manipulated by the user thanks to a series of buttons on the microcontroller, and a second mode fully automatic with a program that will be displayed through the LCD screen of the Mini-DK.

In addition, the system will be set with a detection threshold corresponding to a maximum detection distance. The system will detect when there is an obstacle and emit a fixed tone through a small speaker connected to the board. Simultaneously accompanying the process, the frequency of the said sound will be proportional to the distance.

To achieve the general functionality, we will work element by element in the next pages developing all the necessary content for its implementation and understanding of how each operation takes place.

First of all, to have a general and cohesive vision of the project, we proceed to schematize all the connections made among the microprocessor and its peripherals.

# Description of the hardware

As we have seen, the entire operation of the system will be governed by the 2 main working modes: manual and automatic.

In manual mode buttons Key 1 and Key 2 will control the position of the servomotor increasing and decreasing 10 degrees each time it is pressed, specifically Key 1 will cause the rotation to the left, and Key 2 the rotation to the right. The turning point between modes comes when pulsing ISP. If ISP is activated in manual mode: as the servomotor makes the rotation movement, the ultrasound will be constantly measuring the distance with close objects, and the result updated periodically will be shown through the display and refresh every 0.5 seconds. If ISP is activated again after the previous event, the measurements will stop, and pressing again will restart the measuring process.

However, when the system is in automatic mode the operation is slightly different. In this case the sonar will be continuously sweeping through the 180 degrees stablished range, and at the same time the system will be showing the measurements made by the ultrasound sensor. Also, the system will change the movement direction automatically at the limit positions and, in this mode, ISP will only stop and resume the entire sweep movement.

Once we have understood how each mode works, we can stablish the relationship between them. At the moment the system is in auto-sweep mode and the only way of changing to manual mode is doing a reset of the CPU while pressing KEY 1 at the same time.

Regarding to the ultrasound module, the obstacle detection has a limit range of 30cm-150cm distance. The sound that accompanies this measuring process has a standard frequency of 500Hz, however we have also implemented an extension of the program that, taking into account our threshold of reliable distances, its frequency varies with the distance.

Finally, we have to mention our last peripheral that is going to participate in the project, the serial interface UART0. The main function of our last extension is to show a menu of options, where the possibility of change the resolution in degrees and set a period for each servomotor movement is shown within different already stablished options.

Therefore, to finalize the understanding and general vision of the project, we think is necessary to introduce the flow diagram that at the end will govern the algorithm of the entire program.

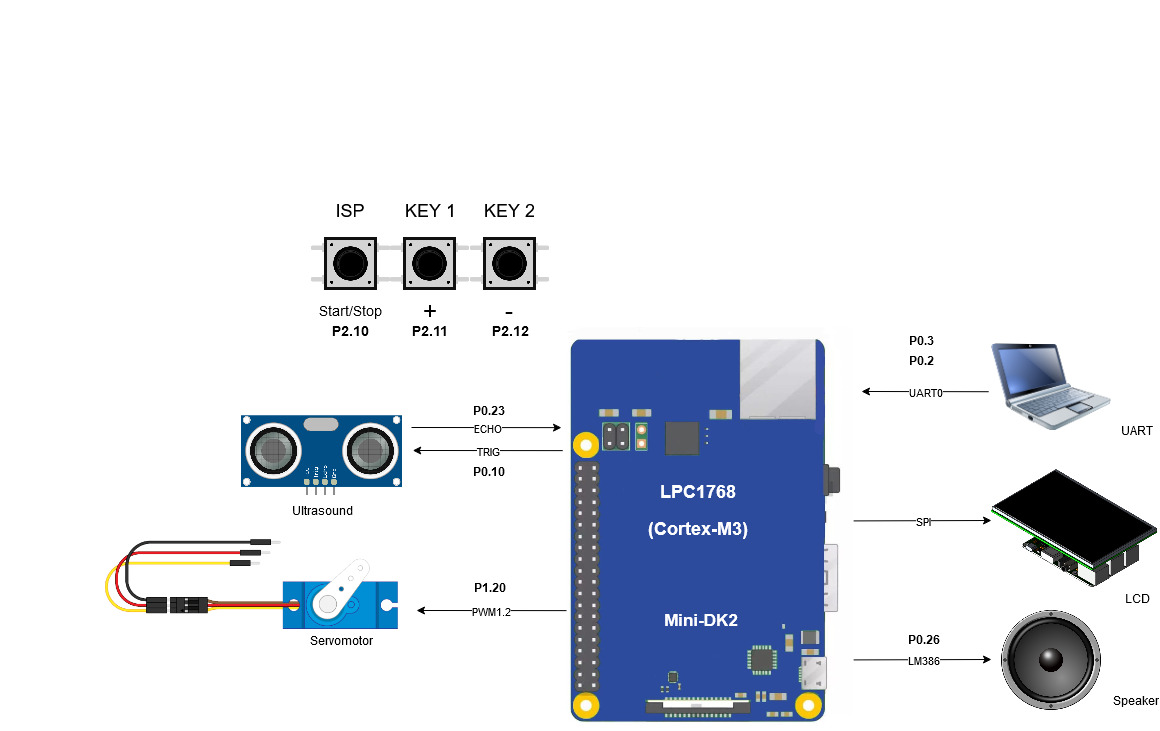


Figure 1. Block diagram of the project

# Description of the software

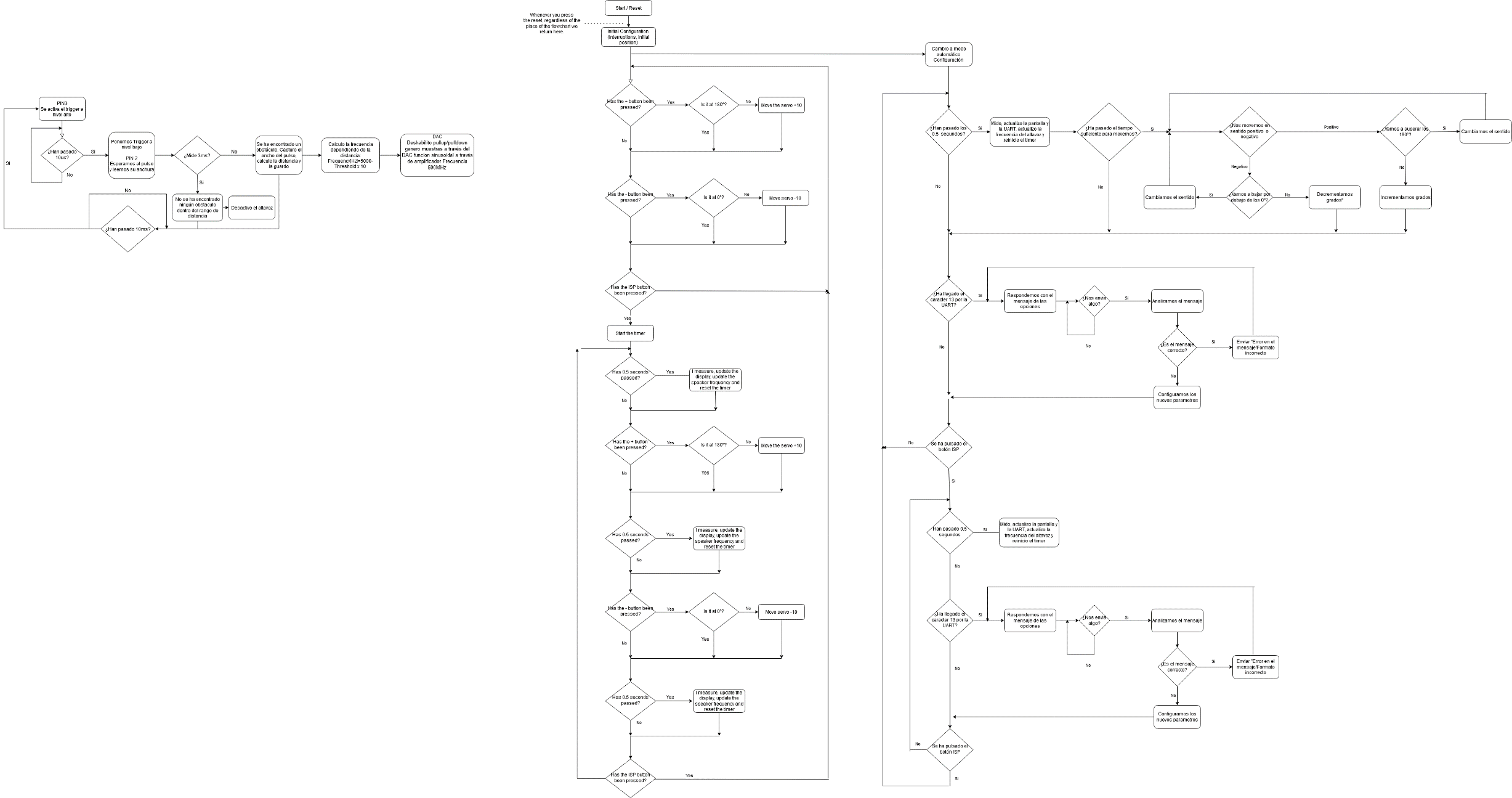


Figure 2. Flowchart of manual-auto sweep mode, servo and UART implementation

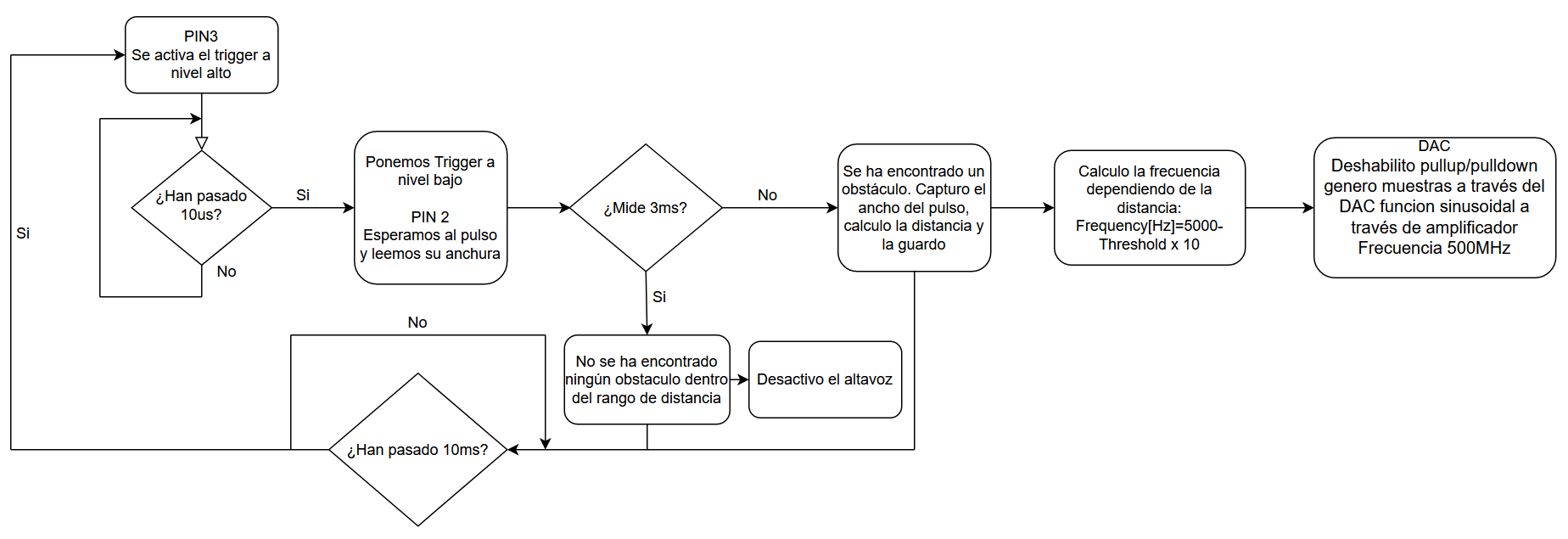
(\* comentar flowchart para más tarde \*)

Figure 3. Flowchart of the ultrasound sensor, speaker and DAC

# Structure of the project and techniques applied

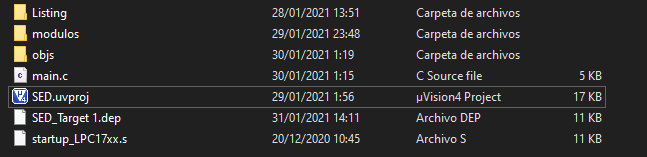
Due to the wide extension of the project, we have applied some techniques like the use of libraries per hardware functionality, the use of a global structure and some basic compilation options in Keiluvision program. Also, we have applied some elemental optimizations techniques like the preferably use of switch cases rather than if/else conditions or the use of smaller type of variables.

Our main structure will store important information of the sonar system that will be globally used and modified by the whole program, occasionally we will pass this structure as a pointer reference to a function. The structure consists of:

* Three parameters that indicate the current state of the sonar: The actual mode (state), the position of the servo and the last distance measured encoded in a float.
* Two configurable parameters via UART: the servo period and the servo motion resolution.
* Four flags: The f\_block\_keys that prevent the Eints handlers being executed multiple times when the button is pressed. The f\_block\_move disables the move of the servo, the f\_block\_measure that allows the UTS to measure and the f\_block\_transmision that disables the transmission from the board.

Texto

Descripción generada automáticamente

Regarding the compiler options, by default KeilUvision generate objs in the same folder where the project is housed, this is uncomfortable for wide projects like this one. So, we have adjusted the target compiler options to store all objs and list items needed in a separated folder improving the handling of the files as the next image shows.

We have organized the project in libraries by hardware functionalities, each block will be explained individually in next sections. The layout that will be used in the header “\*.h” files will be the following:

Texto

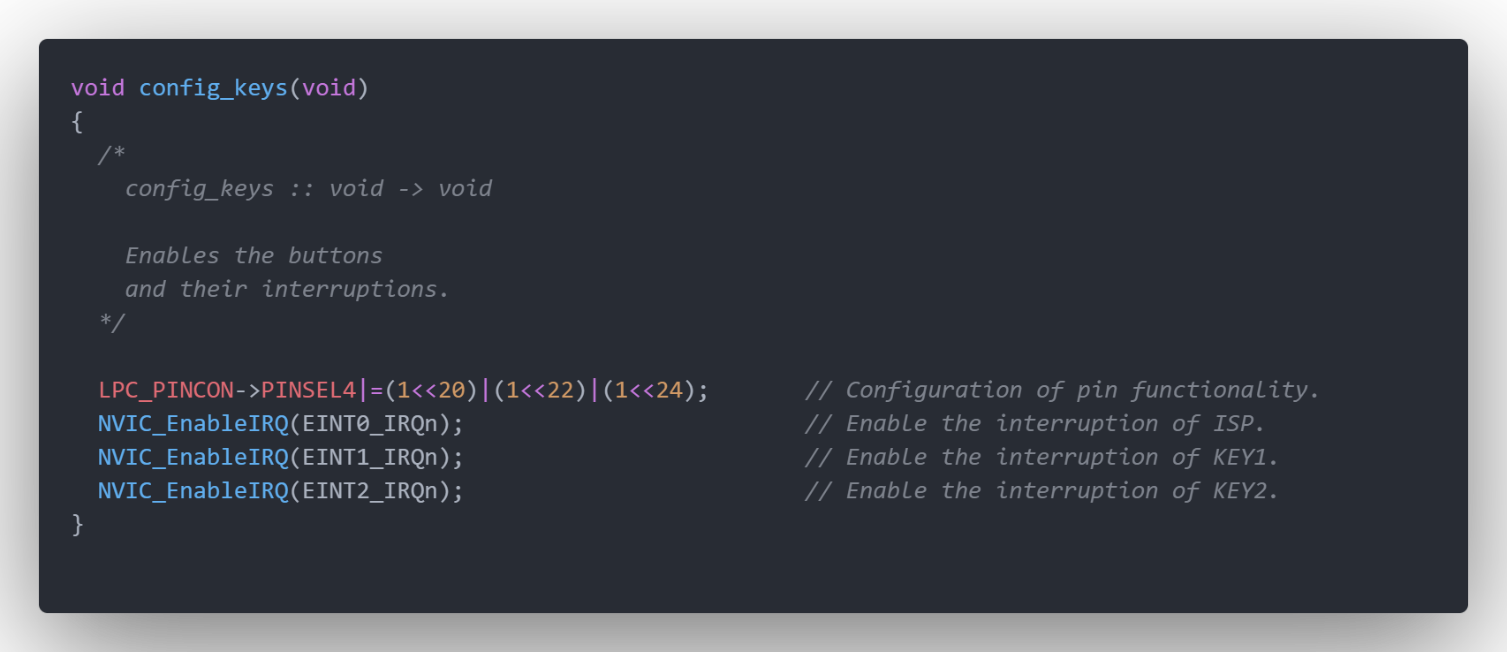
Descripción generada automáticamente

The different modules are related as the following representation. In this representation, we will assign to each of the libraries a box, in this box we can find the functions, structures, and “defines” instructions that are available if the libraries are included. Each of the arrows indicates the libraries that each file contains.

Diagrama, Dibujo de ingeniería

Descripción generada automáticamente

Case module

The first section to be developed consists on a state machine that will be in charge of managing the interruptions by the buttons, in order to select the movement in manual mode, and to show or not the distance measurements.

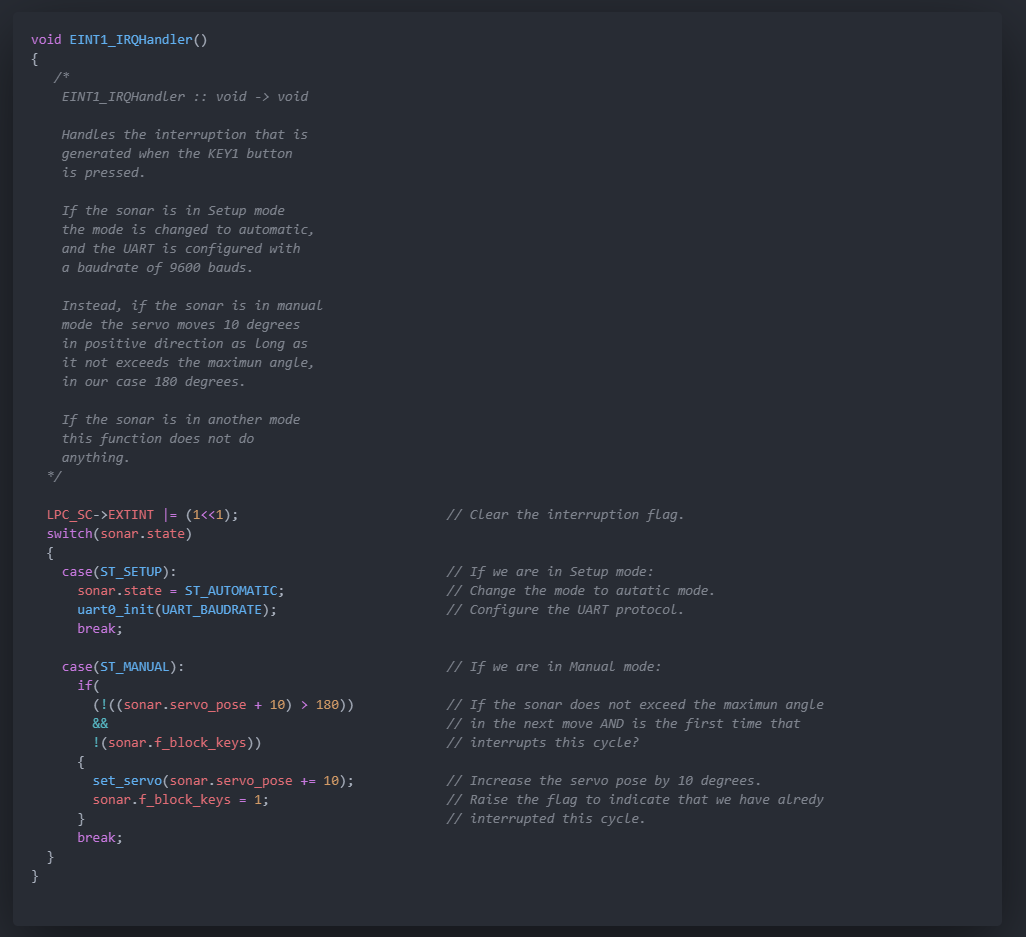
We start by configuring the buttons, enabling their interrupts when activated, and attributing their functionality to the pins 2.10, 2.11 and 2.12.

A header has been written for each function, and it will show what input and output variables the function needs (in this case it does not need any inputs or outputs since it is not necessary to carry out the internal sentences), as well as its content.

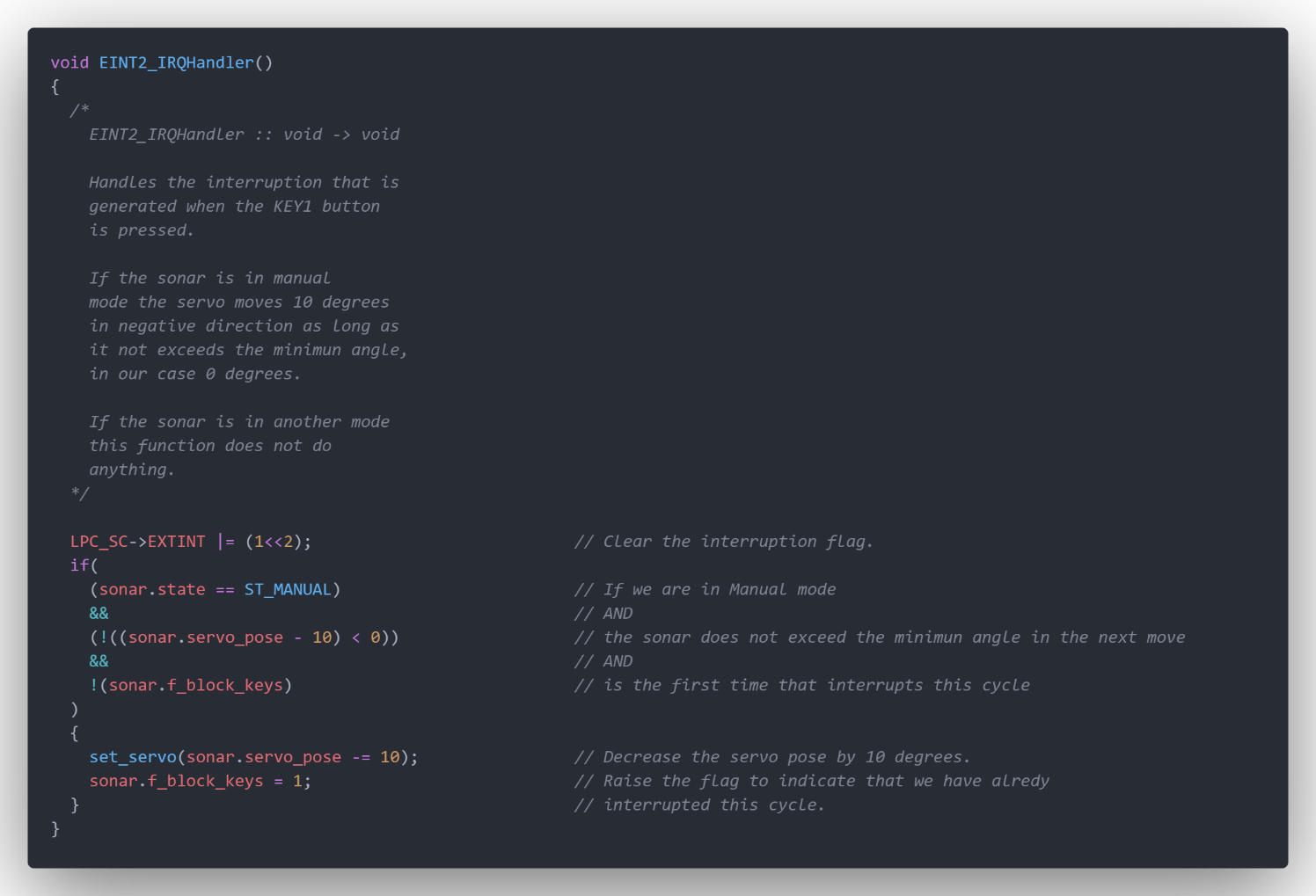


Once the pins are configured, the Handlers for each button interrupt are displayed. A handler is nothing more than the routine that is executed when the interruption that we have established is activated. The functionality of this particular handler, belonging to ISP on pin 2.10 can be read in the header itself, however two elements of the loop should be highlighted.

In first place, the flag called f\_block\_keys has the functionality of being rised in case of interruption, this flag is lowered each time the main is returned, therefore it is used so that when the user presses a button more than once by accident, the servo will just move once until the program returns to the main.

Also, to carry out the toggle, which consists of setting the flag low if it is high, and vice versa, a bitwise XOR gate has been used. This is the specific implementation that we have chosen to switch between modes.

This is the handler belonging to the Key 1 button, on pin P2.11. In case we are in Setup mode, the mode that we have established for when it has not yet been entered, neither has manual mode or automatic mode. If KEY 1 is pressed at the beginning, we change to automatic mode and set the baudrate of the UART. The baudrate stablishes its working speed, in this case 9600 bits per second, however all the functions related to the UART will be developed under its own heading.

In case of being in manual mode, this handler is used to move 10 degrees in a positive direction, always setting a maximum limit of 180 degrees.

Finally, the KEY 2 button at pin 2.12 is disclosed. We have used it to move it in negative direction 10 degrees, as long as we do not exceed the maximum of 180 degrees stablished. This handler does not have a switch case since it will not be used in automatic mode.

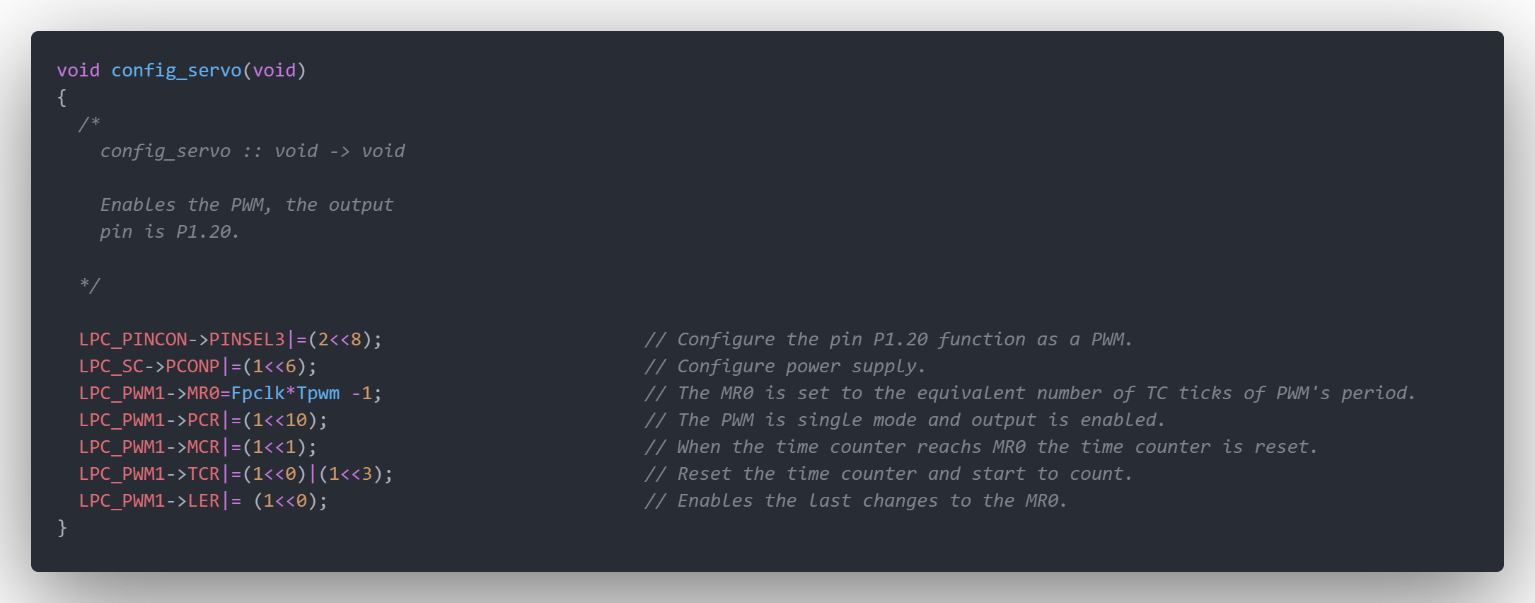
To close the block, we display the statements belonging to the .h file to link the library to the program.

As we can see, we have introduced the other necessary libraries, the global variables belonging to its own structure, and the headers of all the functions inside it. We will always do this in the same way.

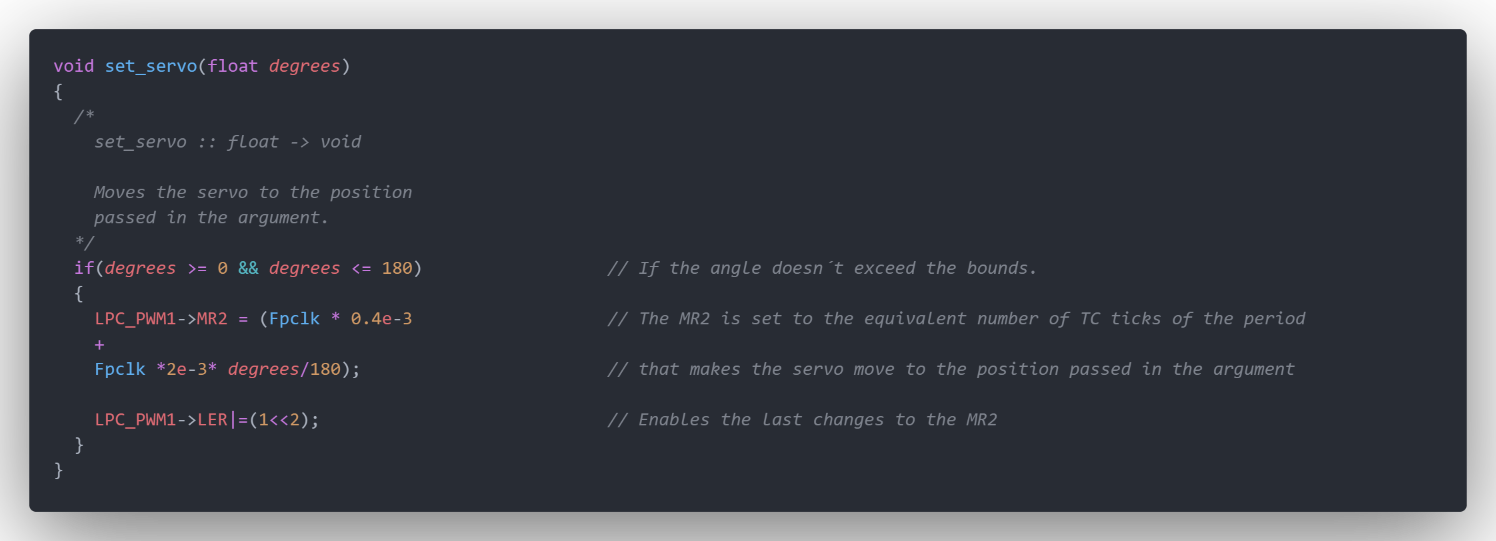
The first 2 statements are used for the compiler to only include the library once.

Servomotor – 9G 90 Towerpro

We start with the hardware peripherals, specifically the servo motor. For this porpose, it will be necessary to use a PWM. A PWM signal is a square signal that can only be in 2 states, high level and low level. The interesting feature about this signal is that all the information it provides depends on the width of the pulses, or in other words in the duty cycle. Therefore, our objective with this peripheral is to design pulses to get the servo to move exactly as we want.

In our concrete case, the position increases as the duty cycle is higher.

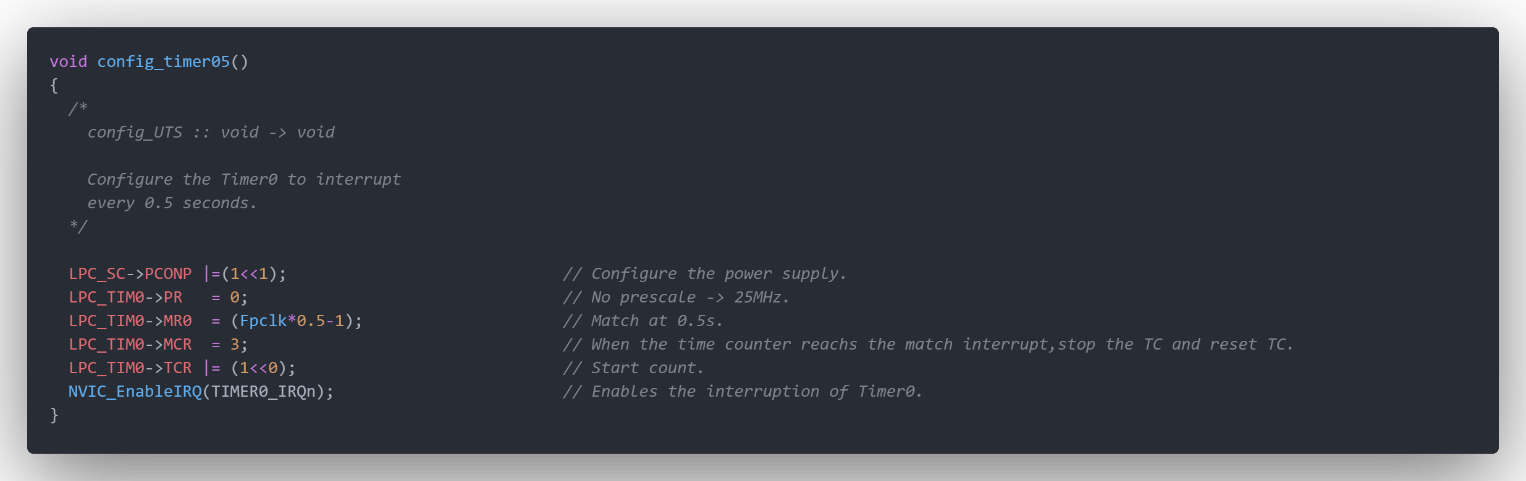
Therefore, the configuration of the servo will be equivalent to configuring the PWM that will govern its movement. To do this we enable the PWM output in P1.20, we feed it, and we establish the period of the signal that we will use by means of the Match Register 0. We also have configured it in single edge, which means that the signal will automatically be set high each period change. Therefore, the appropriate match we need will set the signal to zero only one time per period. We will reset the timer counter each time it ends cyclically, and we will enable the counter and the Match changes.



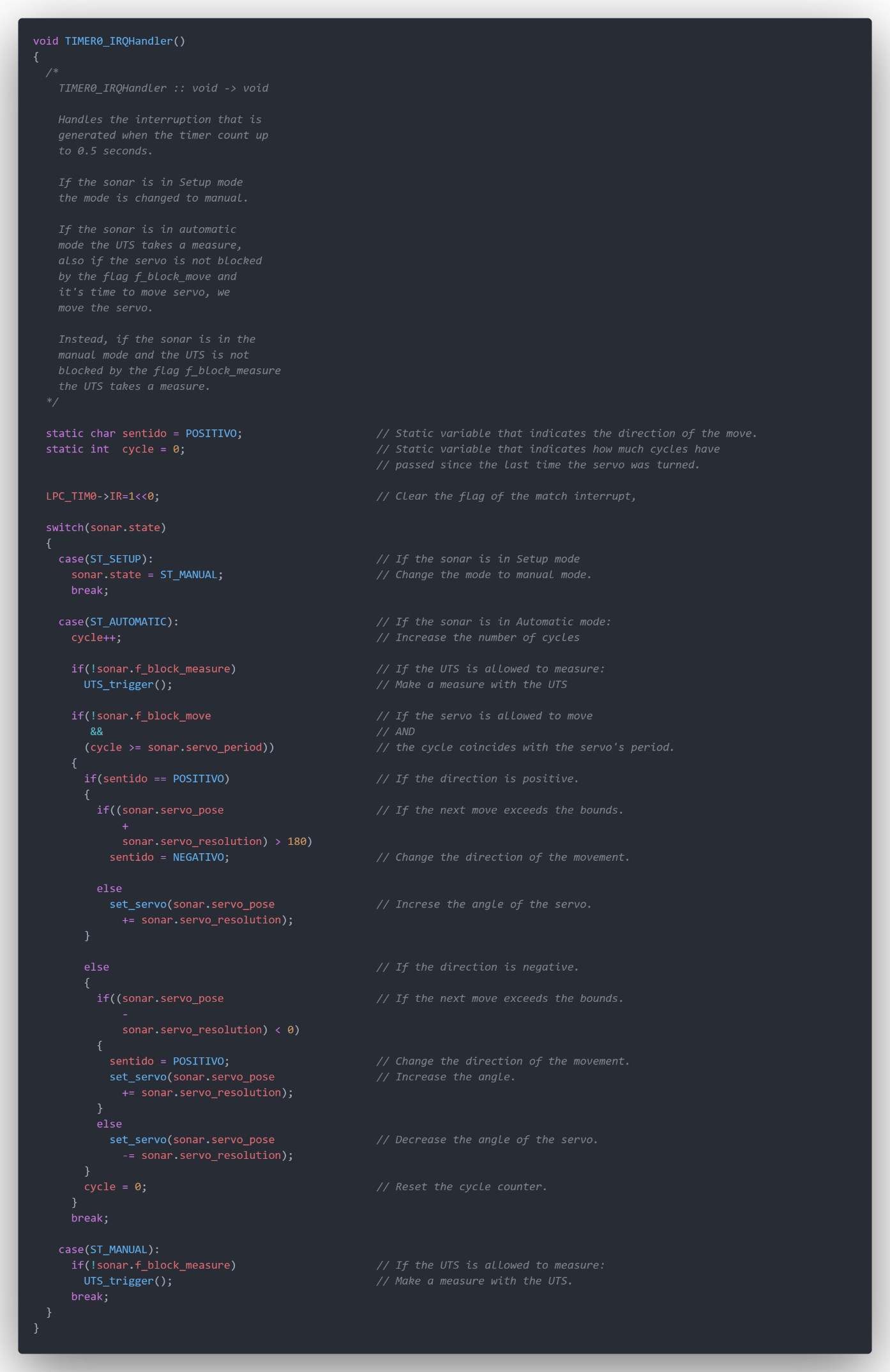
Finally, we see the function that moves the servo once the degrees are granted. For this, is necessary to satisfy the condition of an angle within the range of degrees between 0 and 180. If it is fulfilled, we proceed to move the servo. To do this we establish a match to set the PWM at low level, depending on the variable degrees, and moving the servo.

Hardware wise, the servo is made up of a box with a series of gears and a control card inside. The connections are quite simple, since it only requires a ground wire, a 5V wire, and finally the connector to pin 1.20 of the microprocessors.

Timer05

We have considered it appropriate to reserve an entire section to Timer 0, since it will be the main responsible for every action that will take place every servo movement, activating all the different peripherals that will be developed in the next pages.

First of all we feed the timer, set its frequency to 1MHz as usually, and activate match mode every 0.5 seconds. Every match, the timer will interrupt and reset the value of its timer counter. In order to make it run, we enable its interruption and start the count.



Now we proceed to explain the Timer 0 Handler that will control how the different peripherals interrupt and take actions. As we can see at first, positive direction of the servo and non-cycles are set as initial state in every interruption.

After clearing the flag of the timers interruption, in order to let it enter again every 0.5 seconds, the state machine takes place.

In total we have 3 different states, or “cases”: set up mode, automatic mode, and manual mode. The state machine will work as long as the timer interrupts:

· If sonar.state == ST\_SETUP: When sonar.state is 0, the program is in set up mode. If we don’t press ISP (which would activate its interruption and consequently the change to automatic mode), the program will automatically change to manual mode.

· If sonar.state == AUTOMATIC: When sonar.state is 1, the program is in automatic mode. When this happens we will increase the cycle, every interruption of the timer (0.5 seconds). If the ultrasound sensor interruption flag allows us (is not already taking place) we will activate the trigger emission. Then, if the servo is able to move (the UART will block it in order to make changes) and the number of cycles is equal to the servo set period (an easy way to adjust the servo period of movement), we proceed to move the servo. If we are in positive direction, the position of the servo is increased 10 degrees, and if the displacement exceed the 180 degrees range, the direction changes to negative. The process is analogous if we are in the negative sense.

·If sonar.state == MANUAL: In this case the movement has been already programmed by the KEY interruptions, so in this loop we will only send the trigger every 0.5 seconds.

Ultrasonic sensor – SRF04

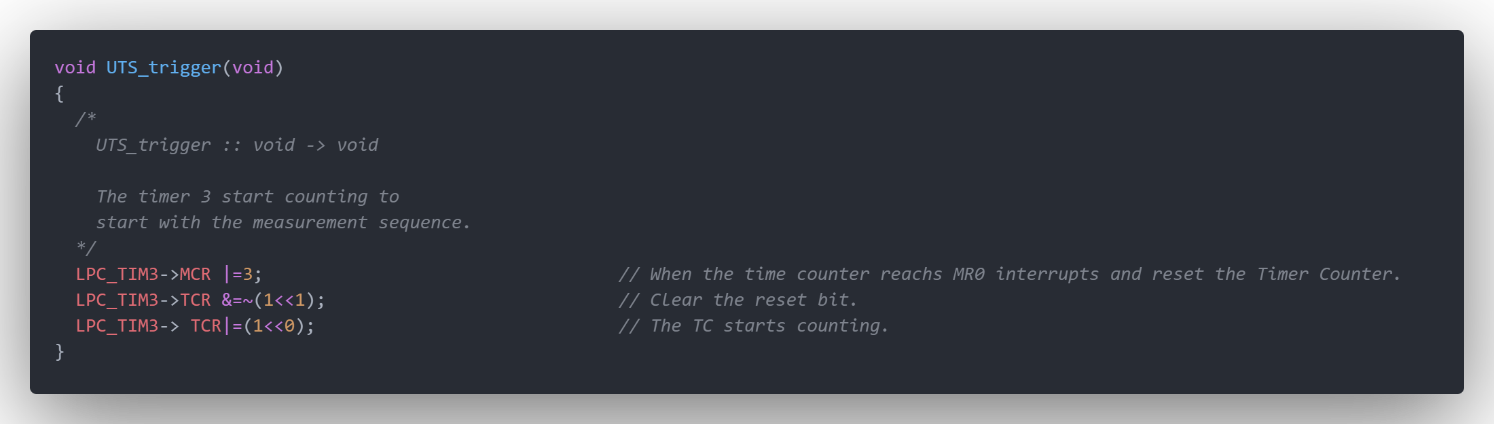
The operation of the ultrasound is probably the most complex one of the entire project. It consists of a device that uses sound waves to produce photographs in order to detect objects. However, it must be considered that it has a limited detection range, in our case between 3 and 300 cm. A routine of its physical routine proceeds as follows: The ultrasound sends a signal at a high level, which we will call Trigger, with a duration of 10 microseconds. After sending it, a burst of pulses made up of 8 cycles are automatically emitted.

The key to obtaining the measurements that we need will be found in the time between the end of the burst and the return time of the echo. This time is interpreted as a pulse width by the ultrasound, proportional to the distance to the object. As we are going to see below, it will be necessary to divide that time in half, since it measures both the outward and return journey made by the signal. Finally, we will consider that if no object is found (and therefore the ultrasound does not receive any type of response echo), the sensor will emit a width of 36ms by internal configuration.

Regarding the connections, the device has the power and the ground pins, in addition to a pin for the trigger and another for the echo, connected to the board by the pins P0.10 and P0.23 respectively.

We are going to explain the code that makes this operation possible.

For this, the first thing we do is carry out its configuration. We want to generate a PWM, but at the same time measure the width of the pulse that the device returns to us. Therefore, in this section, it will be necessary to switch from Match mode to Capture mode. Regarding the configuration, after feeding the timer and set pin 0.23 in capture mode (where echo will be connected) and pin 0.10 in match mode (where the trigger will be connected) the match is programmed to toggle every 10 microseconds. Since we want this pulse to be emitted only once, the interruption of the match itself will deactivate it by passing to capture mode in order to measure the distance. As to capture mode, it is configured to interrupt on the rising edge of the echo signal.



Therefore, every time we want to start a measurement, we call the UTS\_trigger function and put the timer in match mode. After deleting the timer reset functionality, we start counting by enabling the timer counter.

Our Timer 3 can interrupt for two reasons: by match or by capture. When it is interrupted by a match, it means that we have emitted the trigger pulse, starting the measurement sequence. For achieving that, we delete the interrupt flag and deactivate the interruption by match, since the next routine that we want to happen is the echo capture routine, and we do not want to continue issuing trigger pulses.

If we enter the handler because a capture has been made, after clearing its flag by interruption, we save the value in the start variable and reconfigure the capture mode, so it happens on the falling edge. Remember that we want to obtain the width of this pulse.

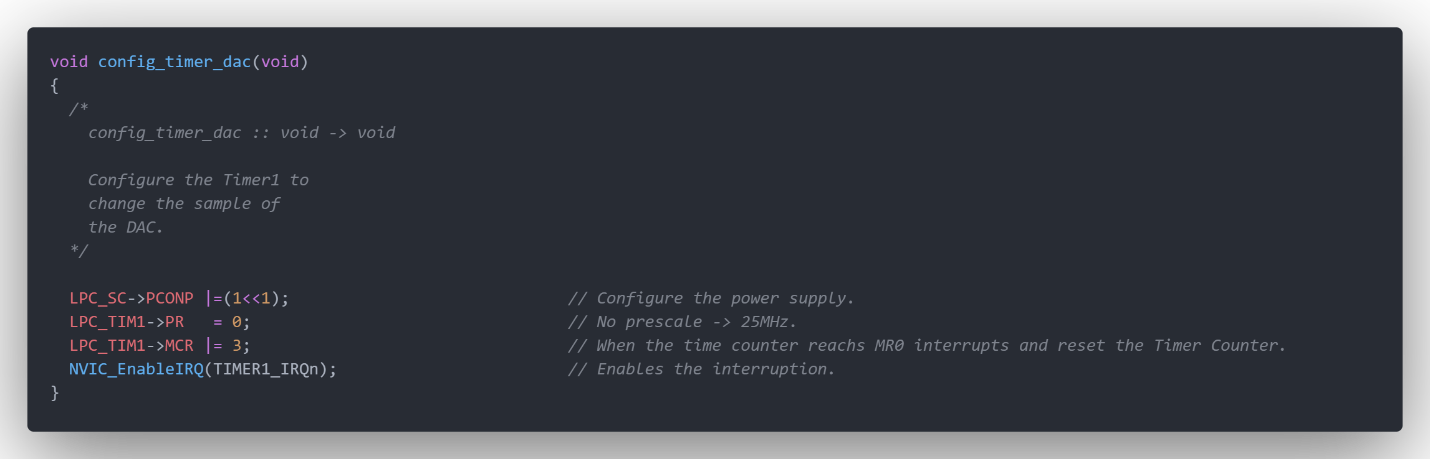
After the next interruption, which will be on the falling edge, we enter the else routine. Now we can calculate the distance since we have enough information:

As we can see the distance is divided by 2, since the echo will bounce the object and return, multiplied by 340 in reference to sound speed, and by 100 in order to get our result in centimetres. Once obtained it, we have to make the necessary configuration to leave the ultrasound ready for another measurement routine. To do this, we will stop the counter, reset the timer, and set the capture mode in rising edge mode. Also reset the variable that stores the value of the rising edge of the echo will be necessary.

As mentioned before, a DAC is to be implemented to reproduce a distance-proportional frequency signal through a speaker with the aid of an amplifier. Although this functionality will be explained below, the “if loop” in the Timer 3 handler is intended to calculate each frequency for each distance measurement, in case it is below the frequency of our Threshold.

Speaker with amplifier – LM386 with PAM8302

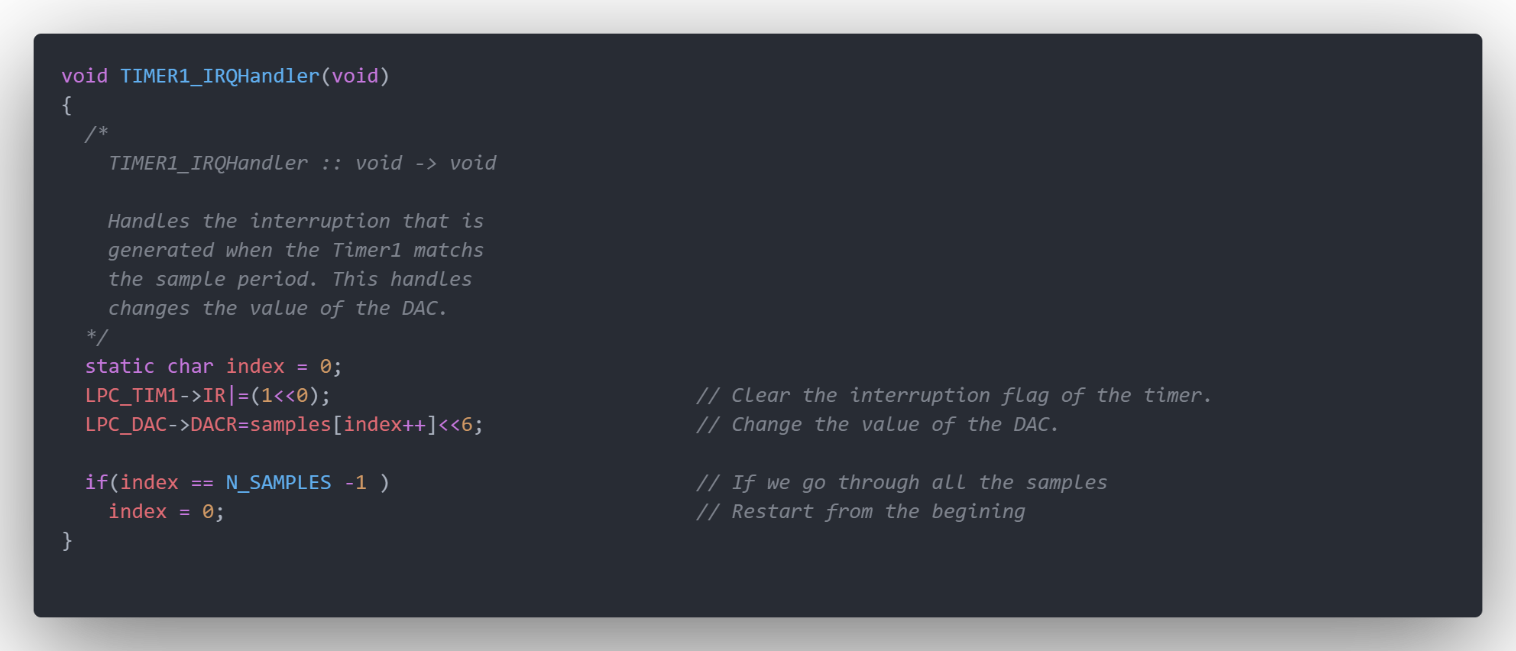
To emit the sound of variable frequency depending on the distance, it is necessary to use a DAC to generate the sinusoidal signal, ours will be 10 bits. This signal will be output by the AOUT pin of our microcontroller, specifically the P0.26 pin.

As we do with all modules, we start by configuring the DAC. We select pin P0.26 as the DAC output, deactivate the pull-up and pull-down resistors, and finally deactivate the DMA mode.

To sample the output of the analog signal, a timer is necessary, we have used Timer 1. In order to get its configuration we feed it, set the standard frequency to 1MHz, configure it to interrupt and reset in each match, and finally enable its interruption.



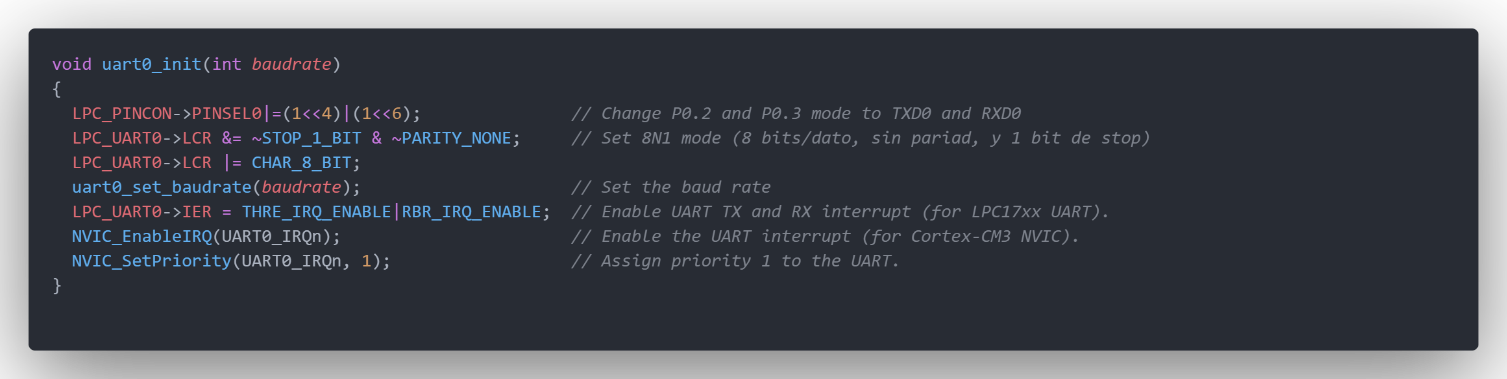
The function generate samples deals with extracting samples from a sinusoidal function. In our case, given that the DAC is 10 bits, the highest value it can reach is 1023. Therefore, we generate a sine with peak-to-peak amplitude 1023, and an offset of 511 since we do not want the signal to acquire negative values.



Finally, we arrive to the Handler of Timer 1. After deleting the input flag to its interruption to be able to enter after the match, it is responsible for changing the value of the DAC. The DACR register takes each sample and outputs it on the selected pin. In case of having finished the sampling process, we reset the index of the register to emit a new value.

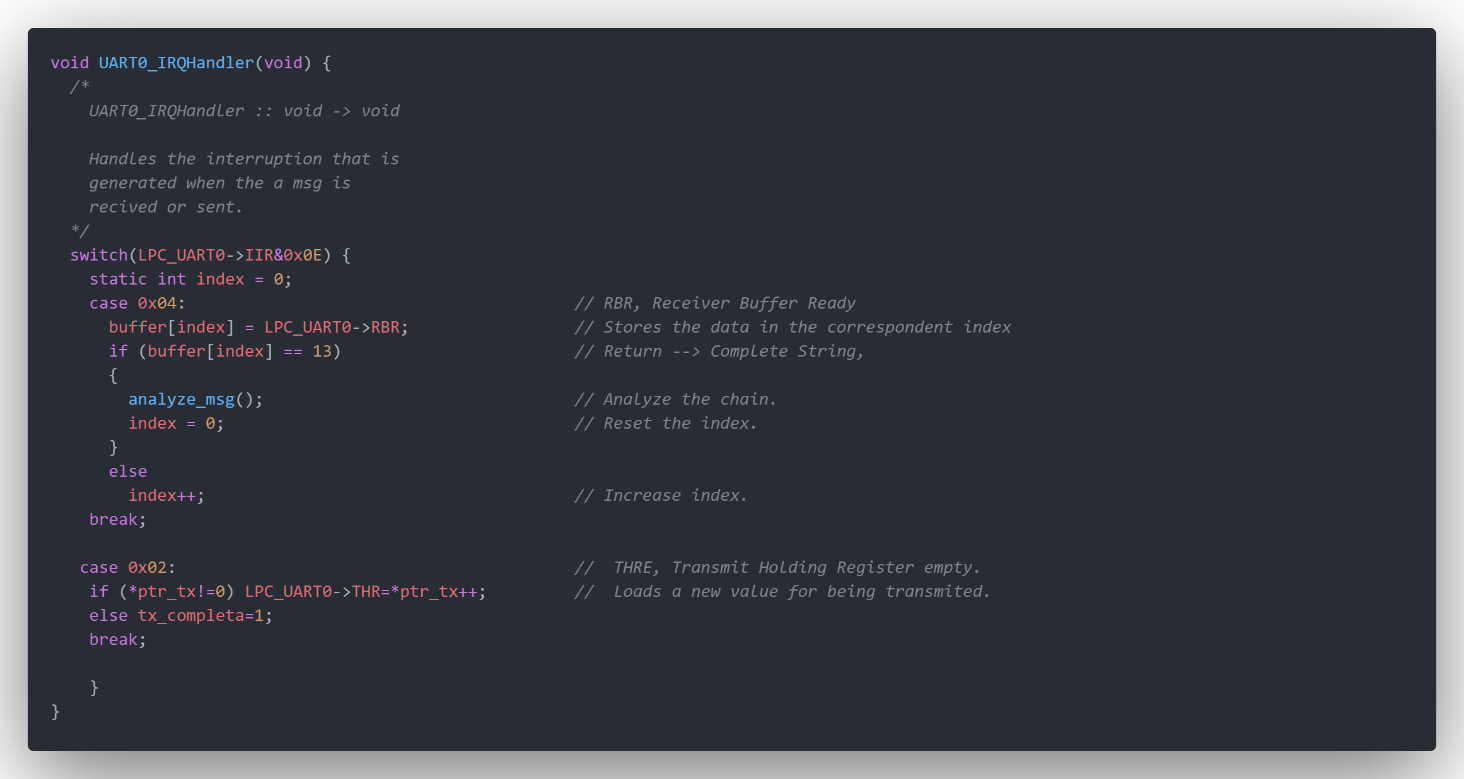
This is where the if loop of the Timer 3 interrupt acquires relevance. This is important as we really want a measurement to take place every 0.5 seconds. Therefore, the program must make sure that the object is within the assigned threshold and to activate the sound if necessary.

UART

The UART uses the online mode and transmits the data at a speed that we can configure. It is a simple but not so optimal means of communication, such as I2C. Our processor has 4 UARTs available, we have used UART0 for the functionality of our program.

First, as we have done in all the modules, we are going to carry out the configuration of the UART0. This function has a particularity, it needs the baundrate as an input. The P0.2 and P0.3 are set as the transmission and reception communication channels. The data will be sent and received in sets of 8 bits, without considering parity and the stop bit enabled. So every time we transmit data, it will start with the start bit, then the data bits will be sent (8 bits in this case) and the string will end with the stop bit. The start and stop bits in this case will be insert in the string automatically by the software.

The baud rate is set to 9600 bits per second. Its configuration has been made calling the function uart0\_setbaudrate, this function is already defined in a library provided by the university and included in our code. Finally we enable the UART to interrupt, and assign its priority. The priority order will be exposed entirely and configure at the main set priority section.



In the UART handler we can discern two exceptions: one for receiving and the other one for transmitting. The receive exception saves the data in buffer index, and in case a 13 character is detected in the string (RETURN character), we proceed to analyse the message. In the second case we are sending data, if during the process we find the NULL character, the string has been read correctly and the data is transmitted entirely. In case NULL character is not found through the string, next char will be sent and the process continues until the null character is found or the string ends.

The first time that we receive a message, the green message of the image is sent through the UART. At this moment is when we block the servo movement, the distance measurement with the ultrasound sensor peripheral, and the transmission of information.

Next, we have programmed the different possible answers of the system to the different replies of the user:

·If the user inserts an ‘h’ character, the message will be shown again.

·If the user inserts an ‘m’ character, we invert the flags using a XOR operator. This will allow the user to stop/start the servo and measure or not. This character was not in the project indications, but we made the decision of include it in order to give the user the possibility of starting the servo movement again after the stop order.

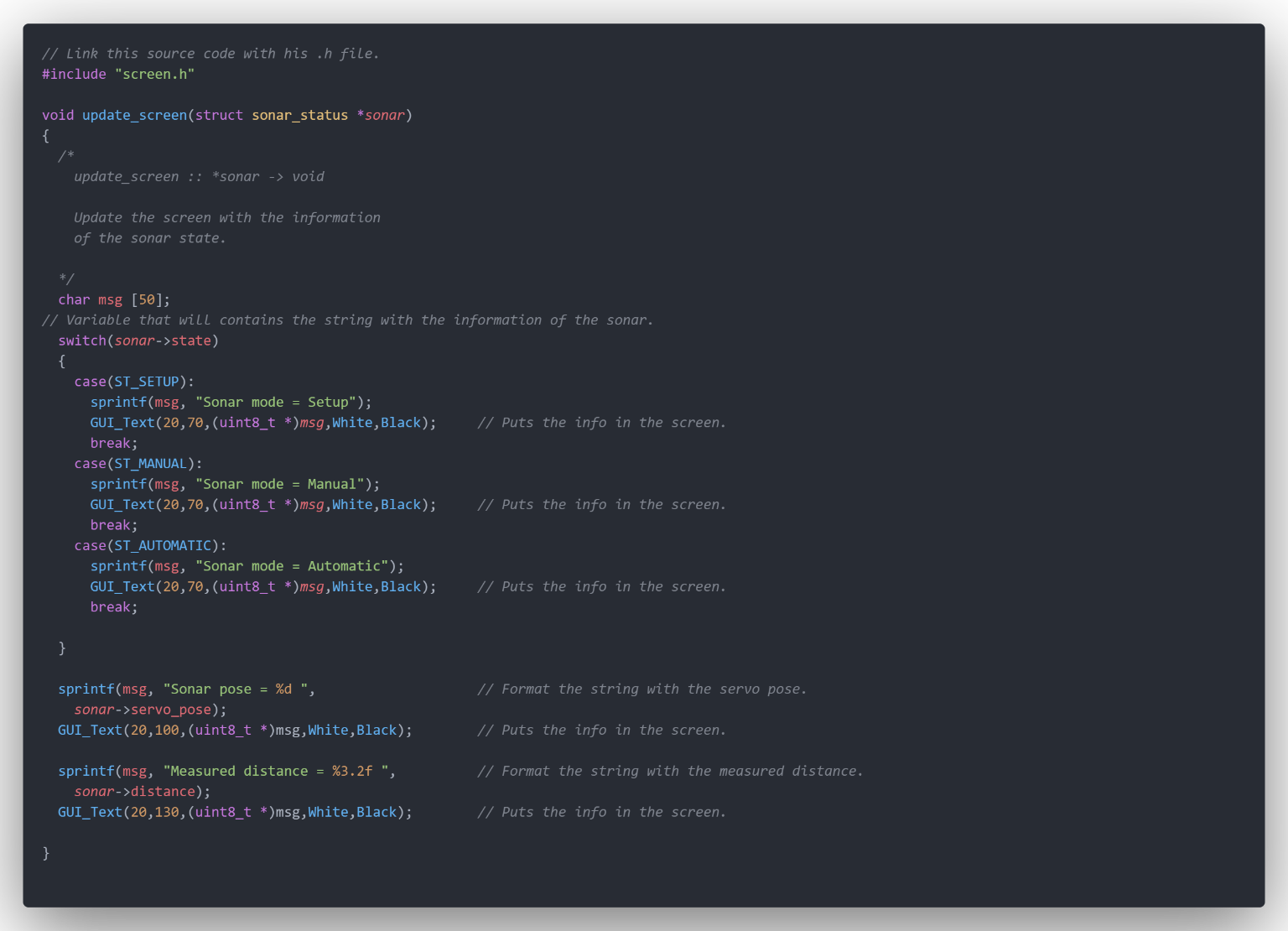
·If the user inserts an ‘s’ character, the user will be able to change the servo period between three possible options: pulse 1 for 0.5 second per period, pulse 2 for 1 second per period, and finally 3 for 2 seconds per period. Once we have checked the answer is in a valid range using the ASCII values, we will store the values calculating the difference between the value and the ASCII value of zero.

·If the user inserts an ‘g’ character, the user will be able to change the resolution between three possible options: pulse 05g for 5 degrees per period, pulse 10g for 10 degrees per period, pulse 15g for 15 degrees per period and finally 20g for 20 degrees per period.

·In case the user does not introduce any of this possible answers, the message “unexpected message” will be shown on the display.

We have decided to transmit data every 15 cycles of the servo, so if the transmission takes place and the sonar is able to measure, the UART will send the message and wait until the message is sent. We have also included, although it was not necessary, a series of messages at the display that will show the servo position and distance at every moment, so it is easier to see what is happening and to debug if it is necessary.

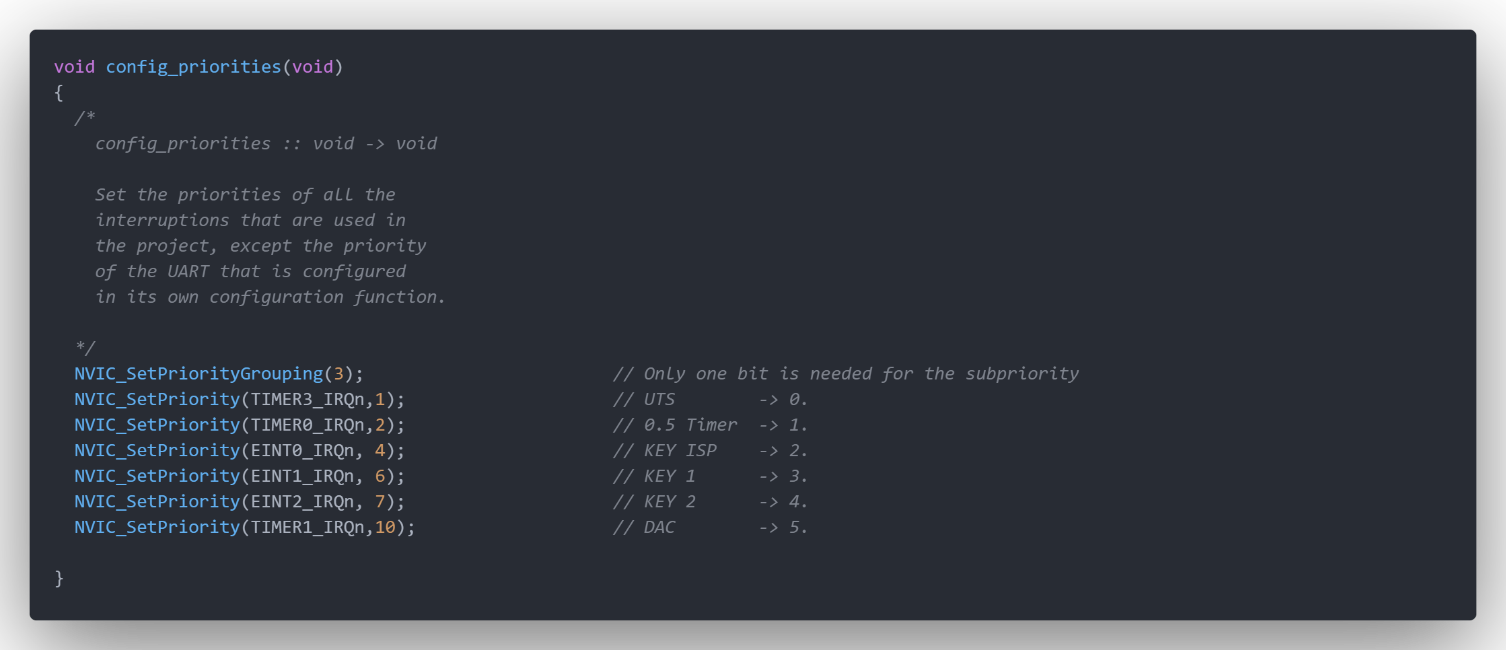
LCD Display

Finally, as the last module, we arrive to the LCD display, the screen where we will show all that information that has been transmitted through the channels described in the previous section with the UART.

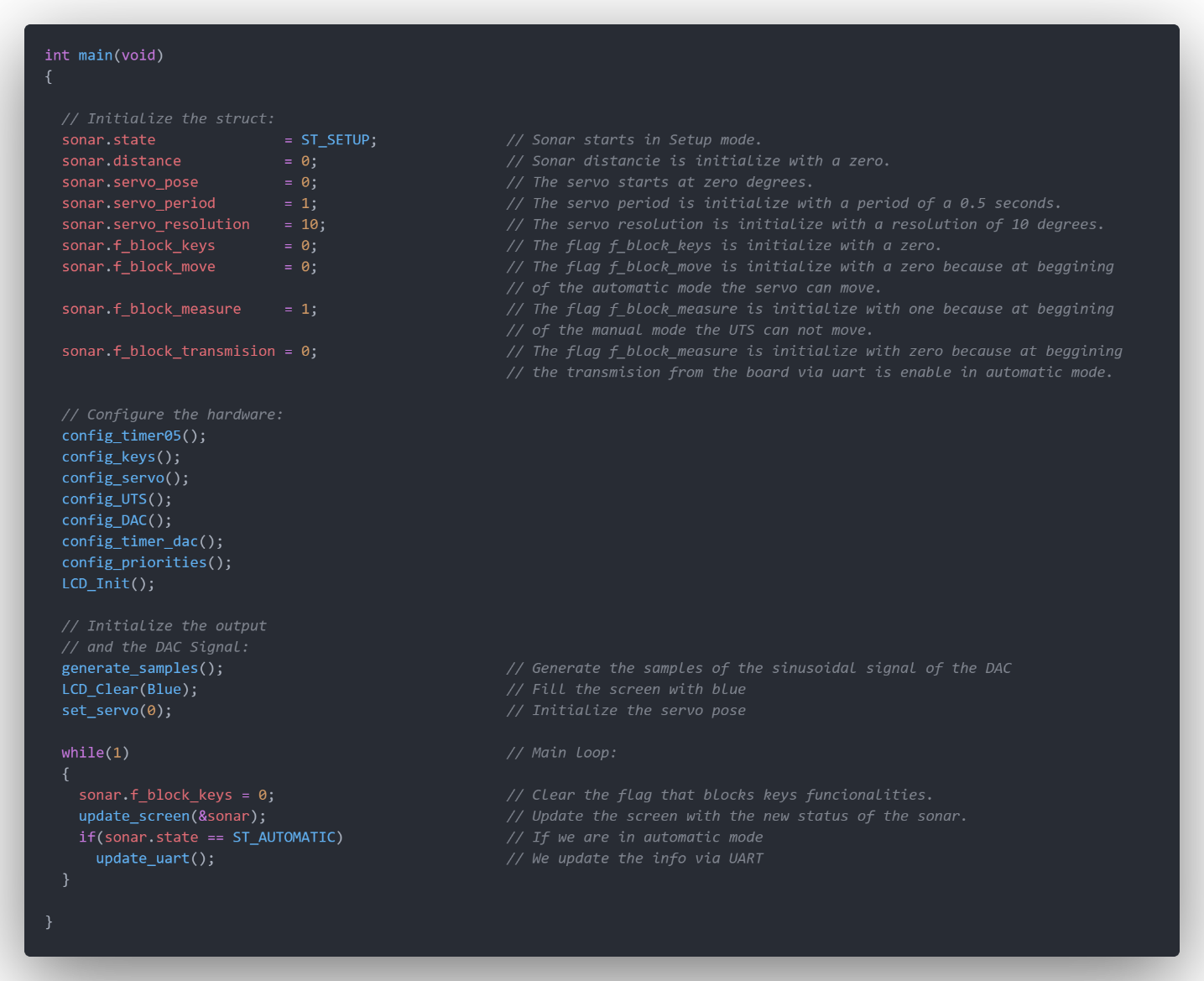
After initializing the variable that contains the sonar information string, depending on the mode in which we are working, we will proceed to show the appropriate message on the screen. A way to easily print or stablish the display format, we have included some libraries. Using the libraries functions, its easier to choose the screen colour, the text position and colour or its size. The GUI\_Text inputs are horizontal position, vertical position string be printed, string colour and background colour. Finally, we use sprintf which is a function from string.h to format the string with the correspondent value.

Main

Finally we come to the main routine, from where we will call all the functions responsible for making the configurations and setting the priorities, and from where the code will jump to the different interrupts when the necessary conditions are met.

The priority configuration has been set as:

We have decided to use 1 bit to subpriorities and 4 for pre-empting, considering more priority the UART (configured at its module already) and the Timer 3. This is necessary to do not interrupt its pulses width and duration, and do not lost any information send through the UART which could affect to the correct functioning of the distance measuring and screen shown information. Then the rest of the modules have been ordered in a way where they can work without compromising each other, being DAC and KEY interruption the less prioritize ones where they can work without compromising each other, being DAC and KEY interruption the less prioritize ones.



Finally we come back to the main, where the initial values of each element of the main structure of variables are established, the configuration functions of each of the modules are called, and finally cyclically we erase the flag of the buttons to allow their functionality when pressed again as soon as possible, and the information sent through the UART arranged on the screen is constantly updated.