Sant'Anna
Scuola Universitaria Superiore Pisa

Sonar implementation for STM Discovery Board

Project report

Gabriele Ara, Gabriele Serra

September, 2017

Prof. Marco Di Natale
Design of Embedded System
Scuola Superiore Sant'Anna

Contents

Chapter 1

Introduction

In this section we will provide a system description from the user perspective, while in next sections this description will be analyzed and rewritten in a more precise and rigorous definition.

1.1 What the system is supposed to do

We want to develop a system which will be able to detect the distance of objects within a short range and then show these objects positions on a screen. The system shall be able to detect objects in the half space in front of him, displaying both the angle of the obstacle's positions from the perspective of the system and their distance.

The system will have two operational modes:

Calibration Mode In this mode the user will move manually the mechanical arm of the system to its initial position.

Scanning Mode In this mode the system will scan all the possible angles in front of him, from $-90°$ to $+90°$, rotating his sensor(s) first in a direction, then in the opposite one. A symbol shall represent the system orientation at any time.

While the system is in Scanning Mode, it shall be possible for the system's user to switch between different views of the obstacles individuated by the system, changing the zoom from a minimum level to a greater one, until a maximum level is reached, then the minimum zoom level is shown again.

In that mode, the screen will show the current zoom level, the measurement unit in which distances are expressed in the current zoom level, a grid and a symbol for each detected object, printed on top of the grid, depending on the object distance and angle.

Chapter 2

User requirements

All terms used in this description are defined in the Data Dictionary contained in Table 2.1.

Following are then user requirements:

1. The system shall be able to detect *obstacles* within 600 cm.
2. Obstacles can be detected by the system if they are positioned within an *angle range* from $-90°$ to $+90°$ from the system's initial orientation.
3. The system shall show on a *screen* detected obstacles' *angle* and *distance*. The system shall show also his current *orientation* on that screen.
4. The user shall be able to change the screen *zoom level* pressing a *button*.
   - **4.1.** When the system has a certain current zoom level, only detected obstacles within the distance range of that zoom level shall be shown.
   - **4.2.** The distance range of the initial zoom level is equal to the system distance range, as specified in Table 2.2.
   - **4.3.** A list of all the zoom levels in the system is shown in Table 2.2. The table shows also the maximum distance range of each zoom level and the *unit* used to display it.
5. A *reset button* shall also be present, to allow the user to reset the system to its initial state.

ID	Term	Text Description	Data Type	Unit	Default	Min	Max	Resolution
SMD	SYSTEM_MAX_DIST	Maximum distance that can be detected	int	cm	600			
M_A	MAX_ANGLE	Maximum angle of vision	float	deg	90			
m_a	MIN_ANGLE	Minimum angle of vision	float	deg	-90			
OR	ORIENTATION	Angle of the arm supporting the two sensors	foat	deg	0	-90	90	2,8125
ZL	ZOOM_LEVEL	Current zoom level of the system view	int		0	0	5	1
CMD	CURRENT_MAX_DIST	Current maximum distance that can be detected with current zoom level	int	See Table 2.2				
CDU	CURRENT_DISPLAYED_UNIT	Distance unit displayed on the screen with current zoom level	string	See Table 2.2				

Table 2.1: User Requirements Data Dictionary

Zoom Level	Maximum Distance
0	6 m
1	4 m
2	2 m
3	100 cm
4	50 cm
5	20 cm

Table 2.2: Association between zoom level and maximum distance displayable. The measurement unit used by this table shall also be used by the screen to show this measurement.

Chapter 3

Functional requirements

All terms used in this description are defined in the Data Dictionary contained in Table 3.1.

Following we have functional requirements:

1. Two *sensors* shall be used to detect obstacles within given ranges.
   - **1.1.** Sensors shall be positioned on a mechanical *arm*, mounted on top of a *motor*, which will change arm's angular position.
   - **1.2.** Sensors will change *orientation* with the mechanical arm, along the angle range in one direction or in the opposite, scanning angles in that range in fixed steps.
   - **1.3.** More precisely, every 70 ms the arm angular position shall change by a *fixed angle*, depending on the system's *angular speed* and current direction.
   - **1.4.** For each position reached by the sensors, the system shall try to detect whether obstacle is present or not and if so, what is its distance.
   - **1.5.** When the system reaches one end of the angle range moving in a certain *direction*, it shall start moving in the opposite direction, until the other end is reached.
   - **1.6.** Distances of obstacles detected by each sensor shall be used to compute a triangulation of currently detected object.
2. For each detected obstacle, the system shall show a *marker* on the screen.
   - **2.1.** Markers will be positioned inside a *grid*, which will represent the space around the system. The grid will allow the user to perceive the obstacle's distance and angle.
   - **2.2.** When changing zoom level, all the markers shall be deleted and re-drawn on the screen, in order to correctly show obstacles' distance in the current zoom level scale. This shall however still be compliant with what expressed in point 4.1. of user requirements.

3.1 Working Modes

This section illustrates all working modes.

ID	Term	Text Description	Direction	DataType	Port	Pin
I1	B_RESET	Button used to reset the system	IN	boolean		
I2	B_ZOOM	Button used to change current zoom level or to start the system of in Calibration Mode	IN	boolean		
O1	TRIG_RX	Trigger signal for right sensor	OUT	boolean	GPIOA	7
O2	TRIG_LX	Trigger signal for left sensor	OUT	boolean	GPIOB	15
I3	ECHO_RX	Echo response of right sensor	IN	boolean	GPIOA	5
I4	ECHO_LX	Echo response of left sensor	IN	boolean	GPIOB	13
O3	PWM	PWM used to control motor position	OUT	boolean	GPIOA	1
O4	LCD_BUS	Bus connecting the board to LCD screen	OUT	boolean	CON3	

Table 3.1: Functional Requirements Data Dictionary

3.1.1 Calibration Mode

1. **Entering mode**
   System enters this mode as soon as it is turned on or at reset.
2. **While in mode**

   2.1.
   - **Pre-condition**: –
   - **Input sequence**: User presses B_ZOOM.
   - **Ouput sequence**: –
   - **Post-condition**: System is now in Scanning Mode.

   2.2.
   - **Pre-condition**: –
   - **Input sequence**: User presses B_RESET.
   - **Ouput sequence**: –
   - **Post-condition**: System shuts down and resets itself.

3.1.2 Scanning Mode

1. **Entering mode**
   System enters this mode as soon as user presses B_ZOOM during Calibration Mode.
2. **While in mode**

   2.1.
   - **Pre-condition**: Current zoom level is different from the maximum one – which is 5.
   - **Input sequence**: user presses the B_ZOOM.
   - **Ouput sequence**: –
   - **Post-condition**: Current zoom level is equal to the one before this action + 1.

   2.2.
   - **Pre-condition**: Current zoom level is equal to the maximum one – which is 5.
   - **Input sequence**: User presses B_ZOOM.
   - **Ouput sequence**: –
   - **Post-condition**: Current zoom level is equal to the initial one – which is 0.

   2.3.
   - **Pre-condition**: –
   - **Input sequence**: User presses B_RESET.

- **Ouput sequence**: –
- **Post-condition**: System shuts down and resets itself.

Chapter 4

System Hardware Architecture

First we'll illustrate system's architecture from a global point of view, then we will show more in detail system's hardware and software organization.

4.1 Global overview

System global overview is shown in Figure 4.1, using SysML diagrams. *Block Definition Diagram* highlights separation between software modules and how hardware components are controlled by different software modules, while *Internal Block Diagram* is used to show data flows between the various hardware components.

4.2 Hardware components

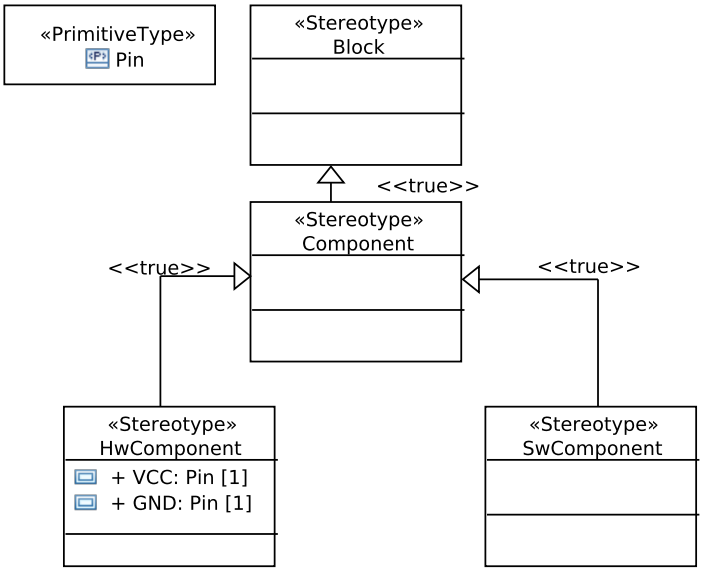
Hardware components choice was driven by parts availability and their price per unit. We will first illustrate external hardware components, before moving on describing briefly the used microcontroller.

4.2.1 Ultrasonic sensors

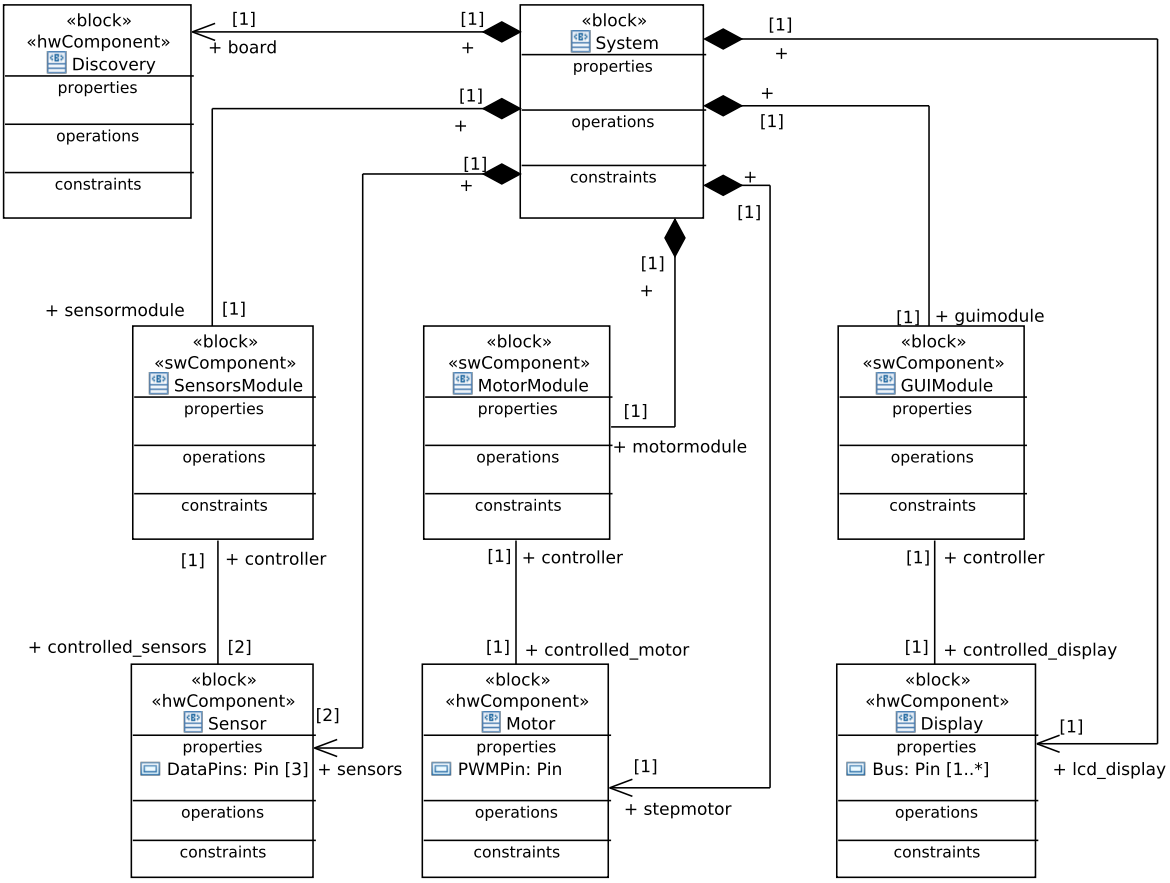
To detect obstacles in front of the system two two ultrasonic distance sensors have been used. The choice fell on two *HY-SRF05* sensors, shown in Figure 4.2. Ultrasonic sensors overcome many of the weaknesses of IR sensors, the latter affected easily by color of obstacles and lighting of the environment. On the contrary, ultrasonic sensors provide precise distance measurement regardless of these conditions, because they use ultrasonic sound waves.

To check if an object is in the area covered by the ultrasonic sensor, it is needed to supply a pulse at least 10 µs long to the *trigger* input. The HY-SRF05 will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo line to high value. It then listens for an echo, and as soon as the sensor detects the echo it will lower down echo line value again.

The echo line is therefore a pulse whose width is exactly the time needed by the ultrasonic wave to hit an obstacle and come back to the sensor. Dividing this time by 2 times the speed of sound will give us the distance of the object. If nothing is detected then the HY-SRF05 will lower its echo line anyway after about 30 ms.

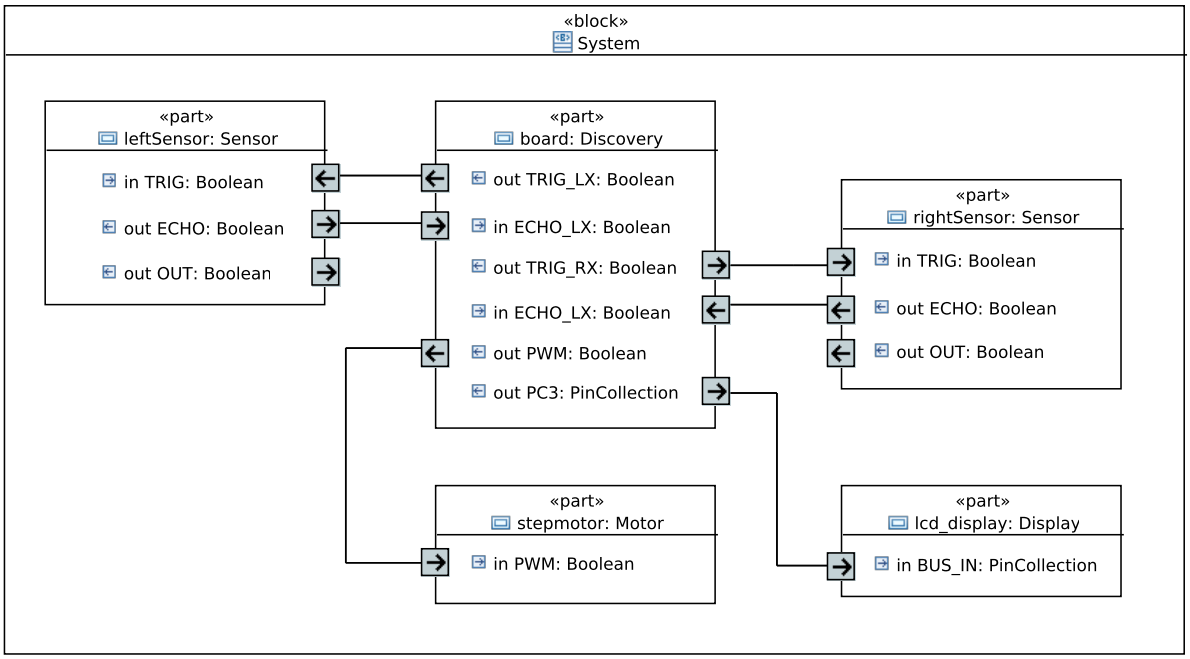


(a) System stereotypes, defined in a SysML profile



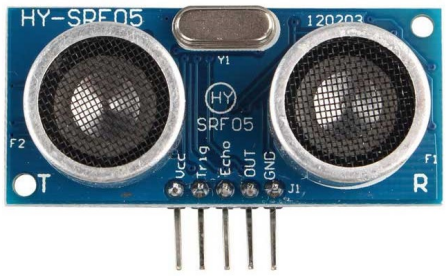
(b) System Block Definition Diagram

Figure 4.1: System architecture shown using SysML diagrams.

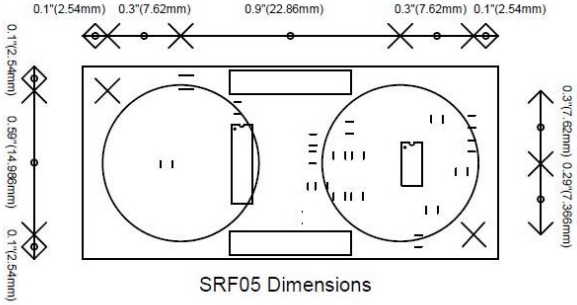


(c) A bounded feasible region

Figure 4.1: System architecture shown using SysML diagrams. (cont.)



(a) Sensor's appearance



(b) Sensor's dimension

Figure 4.2: HY-SRF05 Ultrasonic Distance Sensor.

Figure 4.3: HS-645MG servo motor

4.2.2 Servo motor

In order to change sensors' position a servo motor is used. A typical servo consists of a small electric motor driving a train of reduction gears. In particular for this work, the choice fell on HS-645MG from Hitec, shown in Figure 4.3.

The servo motor will constantly check whether the position of the arm is correct with respect to the desired position, expressed as PWM signal, generating a correction torque, either clockwise or counterclockwise, in case we want to move to another position. This way the mechanical arm mounted on top of motor's shaft will move from one desired orientation angle to another.

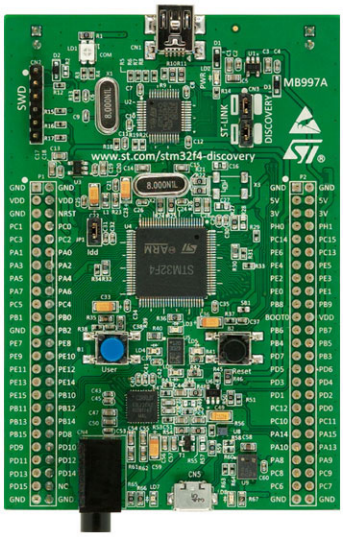
The PWM signal used to control this servo motor expresses the desired position in terms of duration of the pulse sent at a constant frequency. To keep the motor in the same position, the same pulse must be repeated at the working frequency. Working frequency of a HS-645MG is 50 Hz, while minimum and maximum positions are expressed respectively by sending pulses 700 and 2300 µs long.

4.2.3 Development board and LCD

These devices are controlled by a STM32F4 Discovery board, which runs both controller and applicative code using ERIKA OS as real-time operating system. The Discovery Board is shown in Figure 4.4a. Power can be provided to the board using a standard USB Mini Type-B connector, connected to a 5 V power supply[1]; board includes also a voltage regulator and two buttons, respectively a reset button and a user button.

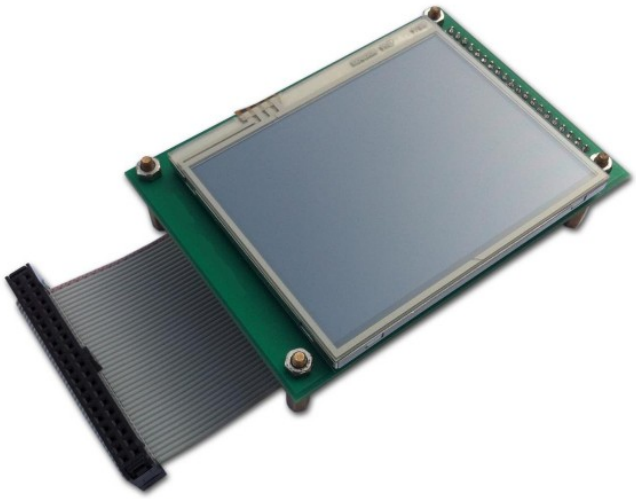
Since the system requires a screen on which to show detected obstacles' positions, an Expansion kit was needed to connect the Discovery board to his LCD screen (STM32F4DIS-LCD). This kit includes both the LCD screen and an Expansion board (STM32F4DIS-EXT), which expands basic functionalities of the Discovery board providing additional connectors (like

[1]Power supply must not supply more than 100 mA.



(a) STM32F4 Discovery board

(b) Expansion board for STM32F4 Discovery board



(c) LCD Screen included with Expansion board kit

Figure 4.4: STM32F4 Discovery kit

Ethernet and serial ones), a MicroSD card slot and connectors for LCD or camera peripherals. Expansion board and its LCD screen are both shown in Figure 4.4.

4.3 Connections and wiring

External peripherals have been connected to the board using a small circuit obtained from a small perforated board. We will illustrate each peripheral pin and wiring scheme in following sections, while full wiring scheme can be found in Figure 4.5.

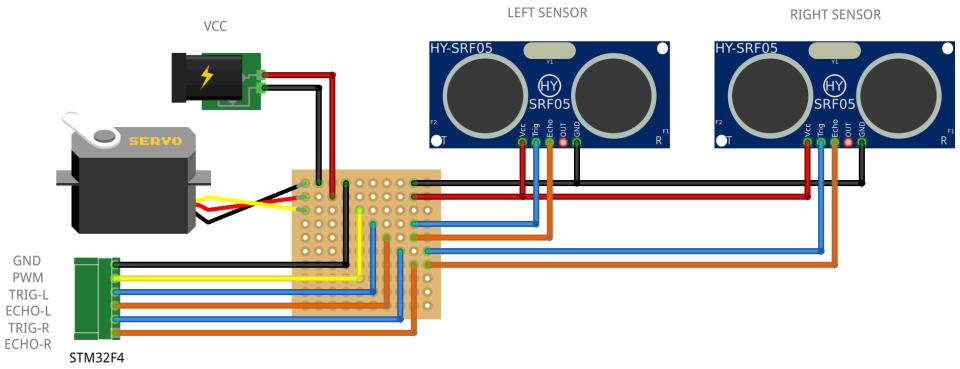


Figure 4.5: Full wiring scheme between external peripherals and the Discovery board

4.3.1 Servo motor connection

Hitec servomotors come with a universal connector called “S” connector. Meanings of individual wires in this connector are shown in Table 4.1. The Discovery board has been connected to appropriate right connectors adopting the wiring scheme shown in Figure 4.6. The PWM signal has been provided using the port and pin assigned to PWM output, as declared in Table 3.1.

4.3.2 Ultrasonic sensors connection

HY-SRF05 sensors have 5 pins, whose meanings are shown in Table 4.2. These sensors can be used in two different configurations, wither using one pin both for trigger/echo or 2 separate pins. The 2 pins mode is the default one and can be selected by simply leaving the OUT pin unconnected. The Discovery board has been connected to right pins adopting the wiring scheme shown in Figure 4.7. Each sensors’ pin has been connected to the relative board one, as declared in Table 3.1.

Name	Cable color	Description
SIGNAL	Yellow	PWM signal to the motor
VCC	Red	Electricity supply
GND	Black	Ground

Table 4.1: Servo motor “S” connector pin meanings

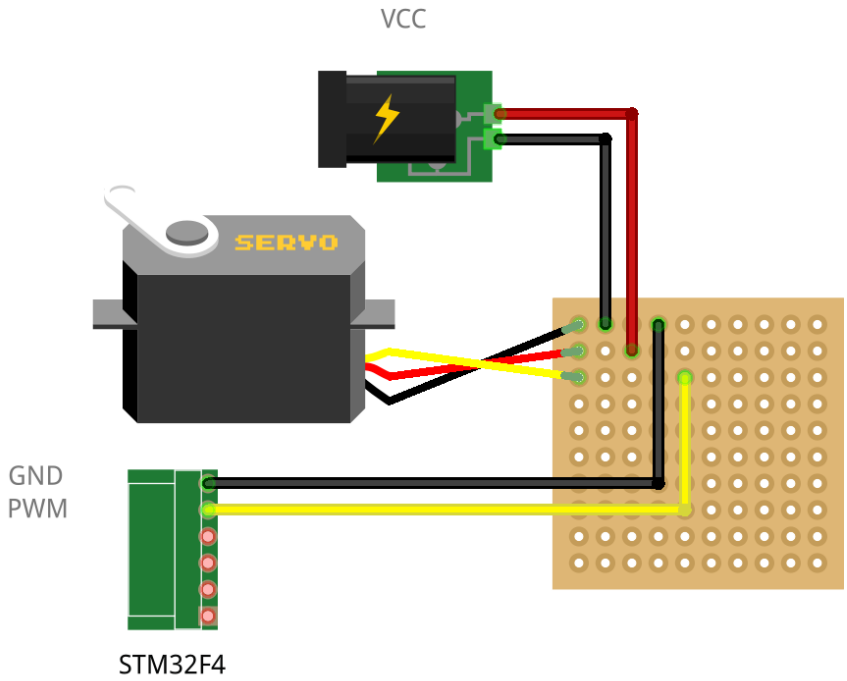


Figure 4.6: Servo motor wiring scheme

Pin	Description
VCC	Power supply
TRIG	10 µs pulse trigger input
ECHO	Sensor response
OUT	Change sensor mode
GND	Ground

Table 4.2: Ultrasonic sensors connection

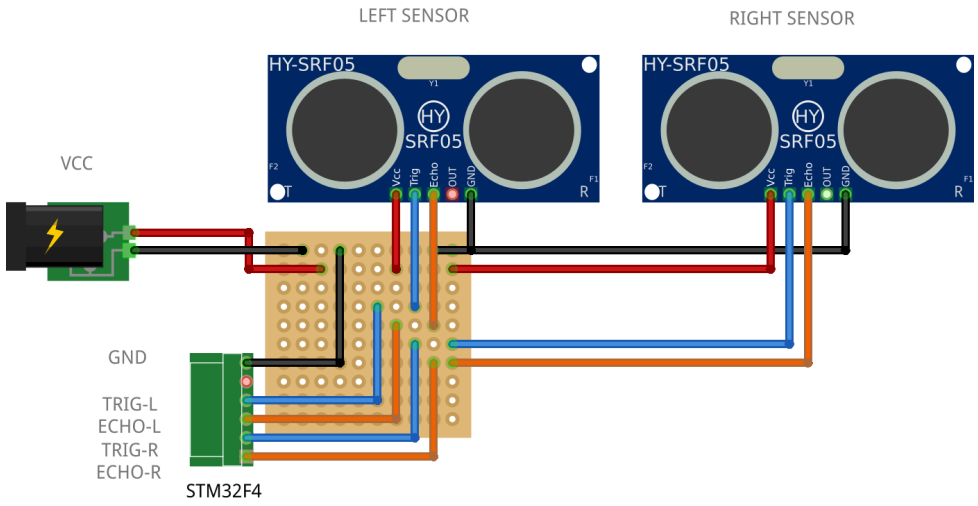


Figure 4.7: Ultrasonic sensor wiring scheme

Figure 4.8: Appearance of the final system.

Chapter 5

Software implementation

We will now show briefly software organization, highlighting major features provided by each of its components.

5.1 RTOS and API

Since this system has to deal with real-time requirements, a real-time operating system (RTOS) is needed to ensure the fulfillment of these requirements. To this, we used Erika Enterprise RTOS, which provides support to fixed and dynamic priority scheduling, priority ceiling and so on.

Erika OS is a modular and incremental operating system, OSEK/VDX standard compatible, which can be compiled in such a way it provides only needed resources and primitives, hence reducing his already small flash memory footprint.

System requirements could be accomplished by simply including only fixed priority scheduling support, hence we used the smallest-footprint version available to implement system's controller, including only needed ST libraries. In addition to ST provided libraries, we included some other higher-level libraries, to manage respectively servo motor and input/output pins.

5.2 Software components

In this section we will illustrate application and control software organization.

5.2.1 Main module and tasks

Main module is in charge of coordinating all other modules. It handles *SysTick* timer interrupts and manages the three concurrent real-time tasks that compose the system:

GUI Task executed with a period of 90 ms, it checks whether the user has pressed B_ZOOM and refreshes the LCD screen.

Step Task executed with a period of 70 ms, it sends the trigger pulse to both sensors and reads previously detected distance, moving then the motor to the next desired position.

Stop Trigger Task executed with the same period of Step Task, but with a different phase, it simply stops sending the trigger signal to both sensors.

5.2.2 Motor module

Motor module is in charge of handling all servo motor routines and control, hence taking care of PWM generation, motor state and so on. This is all transparent to other modules, which can only see the following interface. In the code, we refer to user-domain positions as the positions as they are perceived by the user. The library converts automatically all provided positions in user-domain in motor-domain positions (i.e. width of PWM pulse).

```
/*
 * Returns the user range domain motor position.
 * in:  void
 * ret: int_t, motor position in user range domain
 */
extern int_t motor_get_pos();

/*
 * Initializes motor with initial parameters.
 * in:  initial motor position
 *      initial motor direction%*      default motor increment % What?
 * ret: void
 */
extern void motor_init(int_t init_pos, direction_t init_dir);

/*
 * Sets position of the motor and moves the motor to chosen position
 * in:  new position of the motor
 * ret: void
 */
extern void motor_set_pos(int_t position);

/*
 * Sets a new value for the direction of the motor movement
 * in:  new value for the direction
 * ret: void
 */
extern void motor_set_dir(direction_t direction);

/*
 * Moves the motor following the current increment and direction
 * in:  void
 * ret: void
 */
extern void motor_step();
```

5.2.3 GUI module

LCD screen handling and refresh is taken care by GUI module. This module uses a custom defined library to refresh "widgets" content on the screen. Screen content changes between different modes:

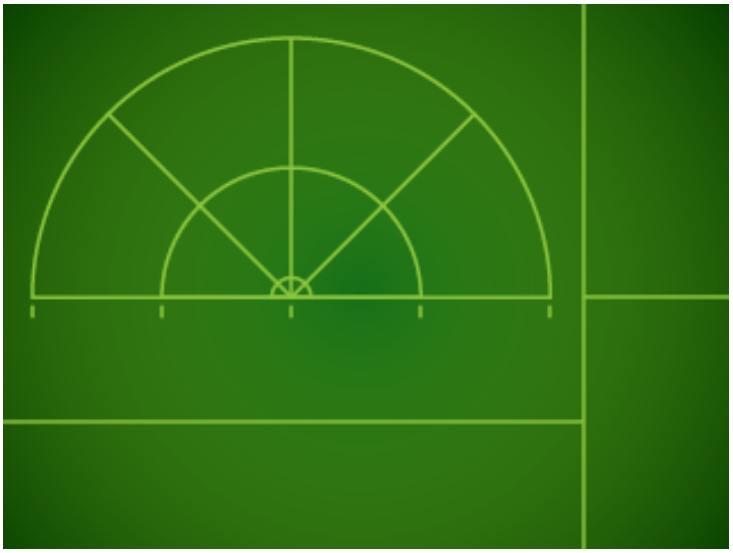


Figure 5.1: LCD main background

Calibration Mode In this mode a message is printed on the screen telling the user to move the mechanical arm in the desired orientation.

Scanning Mode In this mode it will be shown background in Figure 5.1, over which current zoom level, maximum distance and obstacles positions will be drawn, according to what has been specified in *User Requirements*.

The GUI module interface is the following:

```
/*
 * Changes zoom level to the next one, starting back from zero if
 * maximum zoom level is reached.
 * in:  void
 * ret: void
 */
extern void gui_change_zoom_level();

/*
 * Sets current position and distance that has been measured.
 * in:  void
 * ret: void
 */
extern void gui_set_position(int_t pos, int_t distance);

/*
 * Shows on the screen the calibration message.
 * in:  void
 * ret: void
 */
extern void gui_show_calibration_message();
```

```
/*
 * Initializes the interface for Scanning mode with all static
 * texts and background.
 * in:  void
 * ret: void
 */
extern void gui_interface_init();

/*
 * Refreshes LCD screen, to be called only in Scanning mode after
 * gui_interface_init call.
 * in:  void
 * ret: void
 */
extern void gui_refresh();
```

5.2.4 Sensor module

Sensor module handles all sensor pins and states, requiring main module to call some methods at specified intervals. In particular, this module expects a file called `constants.h` in which it is defined a `SYST_PERIOD` constant, expressing *SysTick* period in microseconds. This because `sensor_read` function shall be called every `SYST_PERIOD` µs for distances to be returned correctly by this module.

Module interface is the following:

```
/*
 * Initializes all sensors' pins and ports
 * in:  void
 * ret: void
 */
extern void sensors_init();

/*
 * Reads both sensors value, to be called each SYST_PERIOD microseconds
 * to obtain valid measured distances in centimeters.
 * in:  void
 * ret: void
 */
extern void sensors_read();

/*
 * Sends trigger signal, raising trigger line of both sensors.
 * At the moment of sending the trigger pulse it also updates the
 * global calculated distance at the previous step.
 * in:  void
 * ret: void
 */
extern void sensors_send_trigger();
```

```
/*
 * Stops trigger signal, lowering down both lines.
 * in:  void
 * ret: void
 */
extern void sensors_stop_trigger();

/*
 * Returns the last calculated distance.
 * in:  void
 * ret: int_t, the last calculated distance
 */
extern int_t sensors_get_last_distance();
```

Chapter 6

Testing

We will now show a subset of all tests that have been performed on the system.

Through all phases of system development, several tests have been performed, either on hardware or software components, which involved both prototypes and final system implementation. Main tests on the system involved real use case scenarios, e.g. observing system's reactions to obstacles in sonar's field of vision at different distances and angles, providing hence the system a various set of different inputs.

Apart from these tests, individual software components have also been tested using *CUnit* suite, performing functional and conformance testing. According to what is required by the committee, only these latter tests are illustrated in following sections.

6.1 Conformance Tests

This section illustrates tests performed by the first suite included in our automatic testing program. This suite performs conformance testing of a state machine that is included inside Sensor module used to manage erroneous states the system can get into after partial or total hardware failures. These erroneous states are all identified depending on each sensor's echo response width, as shown in Figure 6.1. In the Figure, arrows represent two consecutive activations of Step Task.

Starting from this, possible states in which a system can transition to are:

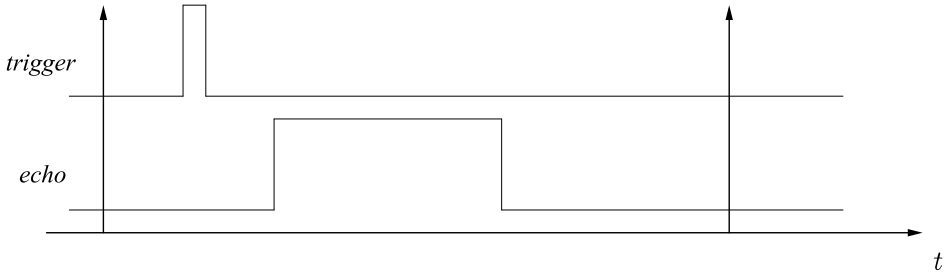
State OK if echo correctly started and finished within Step Task period;

State Lost if echo did not start at all within current period;

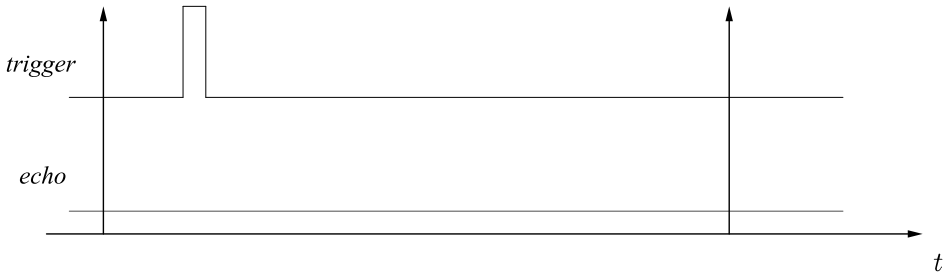
State Long if echo started within current period, but did not finished;

State Next OK if echo started within (one or more) previous period(s), but finished during current period.

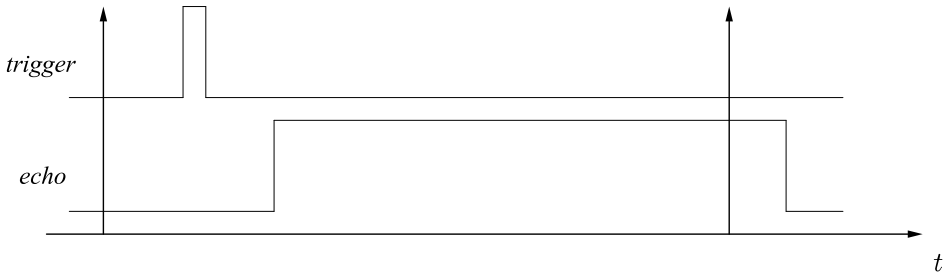
When no reply or a too long reply are detected by the system the detected distance is not valid and cannot be shown to the user. Probably a hardware failure made the line getting stuck at either 0 or 1 value. Otherwise, the detected distance is valid.



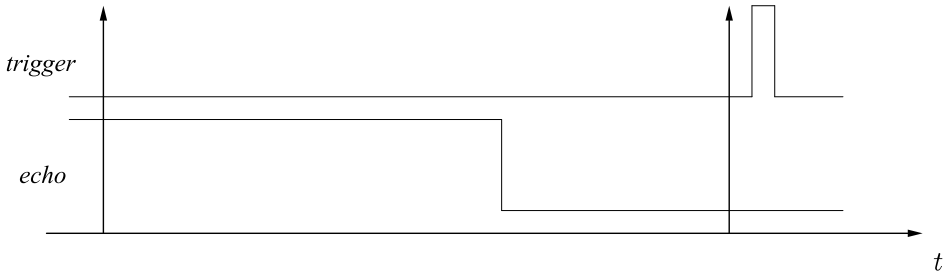
(a) Correct sensor behavior, next state will be State OK and trigger signal can be sent again at next activation



(b) Stuck at zero echo behavior, next state will be State Lost and trigger signal can be sent again at next activation



(c) Stuck at one echo behavior, next state will be State Long and trigger signal will not be sent at next activation



(d) Termination of a stuck at one behavior, next state will be State Next OK and trigger signal can be sent again at next activation

Figure 6.1: Various conditions that can be identified by listening to a sensor's echo signal.

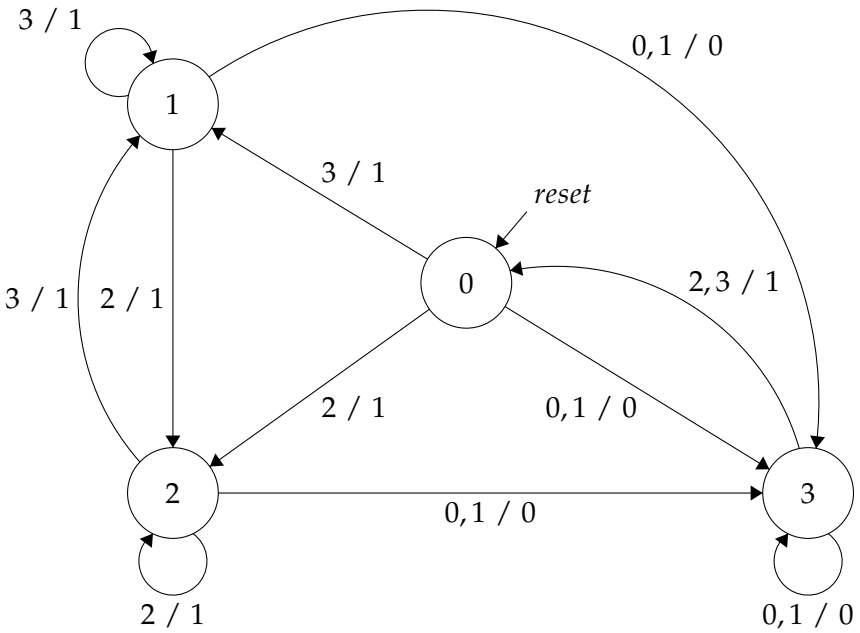


Figure 6.2: Erroneous states' state machine

State Id	State Label
State NEXT OK	0
State OK	1
State LOST	2
State LONG	3

(a) States translation table

Send Trigger	Output Label
No	0
Yes	1

(b) State machine output values

Echo finished	Echo started	Input Label
No	Yes	0
No	No	1
Yes	No	2
Yes	Yes	3

(c) State machine input values

Table 6.1: State machine translation tables. Input values are evaluated at Step Task activation.

Obviously, Step Task period must be set in such a way valid distances are not wrongly interpreted as too long ones. To do so, various testings have been made, obtaining the optimal trade-off.

Notice that in the case a sensor's echo line is stuck at a high value, no trigger signal is sent to the sensor, to avoid overlapping of multiple responses altogether. Trigger signals will start to be sent again after the line changes its value again.

The resulting state machine is the one shown in Figure 6.2, where for the sake of clarity of the scheme states, inputs and outputs have been translated into integer values. Translation table can be found in Table 6.1.

		Test Count	Active?
Suite	**FSM Conformance Test**	4	Yes
Test	State NEXT_OK		Yes
Test	State OK		Yes
Test	State LOST		Yes
Test	State LONG		Yes

(a) Conformance test suite tests

Running Suite FSM Conformance Test	Outcome
Running test State NEXT_OK	Passed
Running test State OK	Passed
Running test State LOST	Passed
Running test State LONG	Passed

(b) Tests outcomes

Table 6.2: FSM conformance test suite and results.

Given this specification, we need to check whether our software implementation of this state machine is compliant or not with it. To so so, we designed a test suite with complete state and transition coverage usint *CUnit*, taking advantage of the fact that we can directly access current state information and set current state directly acting on state machine implementation. Hence, we don't need a *transition tour* or more complicated procedures, like *PW method*.

Table 6.2 resumes the obtained results, which clearly indicate how our state machine implementation is compliant with the given specification.

6.2 Functional Tests

The second suite that we defined performs functional testing on the implementation of the triangulation function, which is used to determine actual distance of a detected obstacle based on the distances measured by the two distinct sensors. To do so, we considered the function as a black box and determined test values by simply looking at expected input variables ranges and types.

Table 6.3 resumes the obtained results, from which we can clearly detect that there was an error handling some out-of-range values of input variables, since robustness tests failed. Thanks to the test, we could detected easily the error and fix it. Test results on the fixed software can be found in Table 6.4.

Table 6.5 shows cumulative results for both illustrated test suites.

6.3 Tests coverage

We also proved actual goodness of these test suites calculating also coverage of automatic testing code. To do so, we used *GCov* suite in combination with *CUnit* to obtain code lines execution statistics. In order to summarize these stats, we used *LCov* suite, which takes *GCov* results and creates with them HTML pages containing both cumulative results and individual lines stats.

		Test Count	Active?
Suite	**Triangulation Functional Testing**	6	Yes
Test	Nominal Testing		Yes
Test	Boundary Testing		Yes
Test	Robustness Testing		Yes
Test	Worst-Case Testing		Yes
Test	Worst-Case Robustness Testing		Yes
Test	Random Testing		Yes

(a) Triangulation functional test suite tests

Running Suite Triangulation Functional Testing	**Outcome**
Running test Nominal Testing	Passed
Running test Boundary Testing	Passed
Running test Robustness Testing	**Failed**
Running test Worst-Case Testing	Passed
Running test Worst-Case Robustness Testing	**Failed**
Running test Random Testing	Passed

(b) Tests outcomes

Table 6.3: Triangulation functional test suite and initial results.

Running Suite Fixed Triangulation Functional Testing	**Outcome**
Running test Nominal Testing	Passed
Running test Boundary Testing	Passed
Running test Robustness Testing	Passed
Running test Worst-Case Testing	Passed
Running test Worst-Case Robustness Testing	Passed
Running test Random Testing	Passed

Table 6.4: Fixed triangulation functional test suite outcomes.

Cumulative Summary for Run

Type	**Total**	**Run**	**Succeeded**	**Failed**	**Inactive**
Suites	2	2	- NA -	0	0
Test Cases	10	10	10	0	0
Assertions	83	83	83	0	N/A

Table 6.5: Cumulative outcomes summary for the all test suites.

Thanks to these tools, we proved that our tests cover 100% of tested code, hence proving their goodness. This result is shown in Table 6.6.

Filename	**Lines Coverage**		**Functions**	
main.c	100.0%	168/168	100.0%	18/18
sensor.c	100.0%	35/35	100.0%	3/3

Table 6.6: Cumulative stats obtained by checking coverage of given test suites.