Development of a Tank water level & temperature monitoring solution based on the STM32L476RG Microcontroller

Carrasquel Gámez, Julio César[1]

[1]Università degli Studi di Roma La Sapienza
Dipartimento di Ingegneria Informatica, Automatica
e Gestionale Antonio Ruberti (DIAG)
Via Ariosto 25, 00185 Rome, ITALY
carrasquelgamez.1726154@studenti.uniroma1.it

Abstract.- STMicroelectronics provides the STM32L4 family of low-power microcontrollers based on the ARM Cortex M4 architecture. This project uses the STM32L476RG microcontroller as the core piece for the management of a tank water-level & temperature monitoring system. For the detecting the tank water-level is used the HCSR04 ultra-sonic ranging device whereas for the temperature is used the water-proof DS18B20 thermometer which goes immersed below the water. The system also includes an U-Blox NEO-6M GPS receiver which keeps track of the location where the system is operating. In order to carry out the development tasks it was used the STM32CubeMX framework and the System Workbench 4 IDE which provide an easy & professional environment. It was taken into advantage the several microcontroller capabilities such as the different clock sources, the UART interfaces and the management of different general input/output ports among others in order to make a correct system configuration. The first section of this work makes a description of the microcontroller & the project general structure. The second section describes the GPS module. The third section explains the temperature module. The fourth section addresses the water-level module, and finally the fifth section describes the implementation of a system terminal for interacting with the user.

1 Introduction

This introductory section of the project is organized in the following way: Section 1.1 gives a general introduction about the STM32L476RG microcontroller and the ARM Cortex M4 architecture in which is based the microcontroller. Section 1.2 introduces the software development tools used for the development of the system. Section 1.3 presents an schematic diagram showing the connection between the microcontroller board and the sensor devices. Section 1.4 describes the procedure done in the STM32CubeMX framework for initializing the microcontroller settings according to the required needs of the project (e.g. peripherals configuration, clock sources). Later on, Section 1.5 describes the project files structure. Finally, on Section 1.6 it is explained the main flow of the system program.

1.1 The Microcontroller

The STM32L476RG microcontroller [1] belongs to the STM32L4 family of low-power microcontrollers developed by STMicroelectronics. It is based on the ARM Cortex-M4 architecture. It has a 1 MB ROM FLASH Memory and a 128Kb SRAM data memory. It can work up to a clock frequency of 80 MHz. The microcontroller is embedded within the Nucleo-64 development board which provides a set of Arduino compatible pins plus a set of expansion pins, a 32 kHz crystal for low-power management, and a debugging unit controlled by the ST-link debugger chip provided with an USB interface for development & debugging purposes. The following figure introduces the development board with the microcontroller.

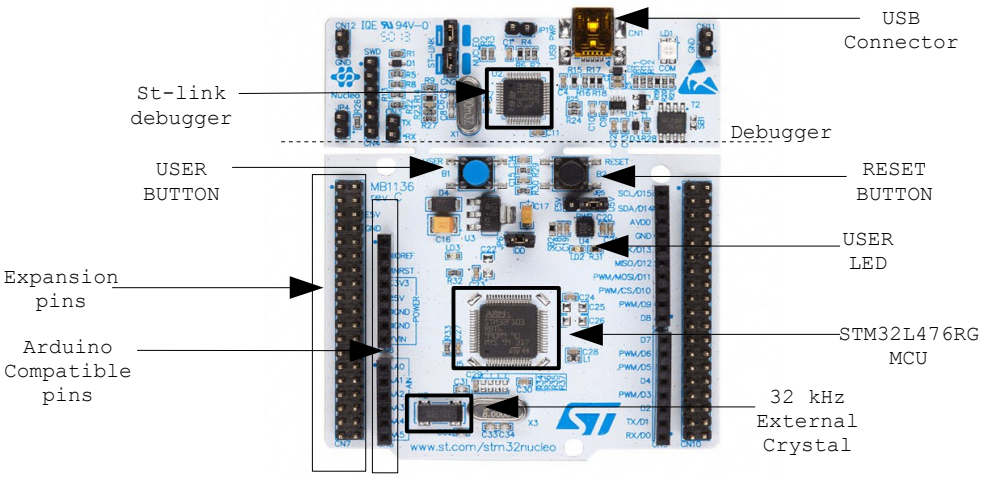

Figure 1. The Nucleo development board with the STM32L476RG MCU.

1.1.1 The ARM Cortex M-4 Architecture

ARM refers to a family of Reduced Instruction Set Computing (RISC) architectures. An ARM architecture is a set of specifications regarding the instruction set, the execution model, the memory organization and the instruction cycle among other features which describes precisely a machine implementing that architecture. Cortex-M is a range of scalable, energy efficient and ready to use processors designed for embedded technologies. They have been conceived for applications such as Internet of Things, automotive, industrial control systems, medical instruments, etc. They are 32-bits based Harvard architectures. In particular, the Cortex M-4 is described on the ARM architecture revision ARMv7-M. The architecture uses the Thumb-2 instruction set which is a mix of 16-bit and 32-bit instructions. Among its registers the Cortex M-4 has R0 – R12 general purpose registers, a stack pointer register SP, a link register LP -which is the return link from exception handling-, a program counter PC -loaded with reset handler start address on reset-, and a program status register APSR which handles several flags such as negative, carry bit, zero flag, etc.

What respects to the memory space it has an address space of 4 GB which in fact is standardized between all Cortex-M cores. This space is organized in several sub-regions with different logical functionalities. The following figure provides a map of the address memory space.

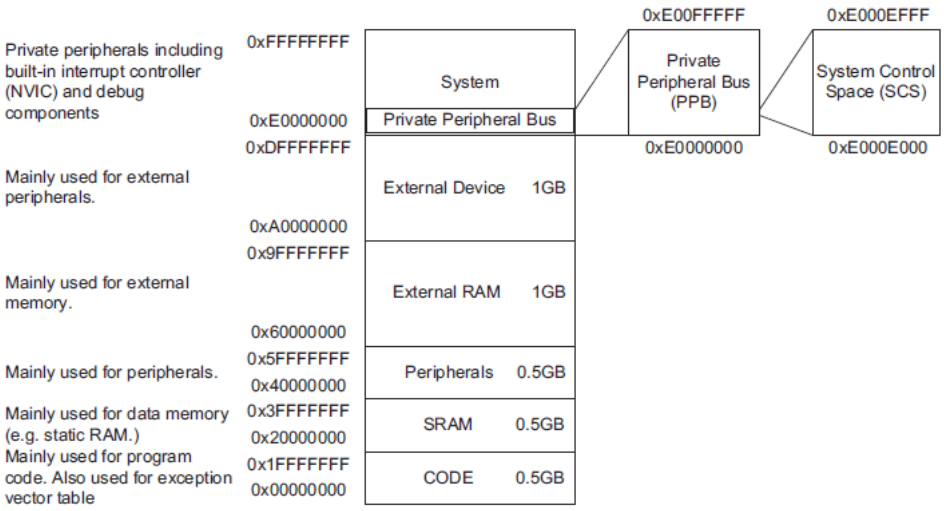

Figure 2. Cortex-M fixed address memory space.

The first 512MB are dedicated to the code area starting at the address 0x0000 0000 (32-bit addresses). This area also includes the pointer to the beginning of the stack and the vector table. It includes as well a system memory area which is a ROM region reserved to boot-loaders (used to load code from several peripherals, including USARTs, USB, etc). Other section which this area includes is the options bytes section region which contains a series of bit flags which can be used to configure several aspects of the MCU (flash read protection, boot mode, etc). The next region of the 4 GB space is the SRAM area which starts at address 0x2000 0000, and it is extended up to the effective amount of the internal SRAM of the microcontroller. Then, the next memory region of 512 MB is dedicated to the peripheral mapping (timers, USARTs, etc.), and this area is organized according to the microcontroller specific features. The next memory address region belongs to a 2 GB address conceived for external memory management. Finally, the last 512 MB memory portion corresponds to the internal Cortex processor peripherals plus a reserved area for future enhancements to Cortex processors.

A fundamental feature of the ARM architecture is the interruption & exception management -handling of asynchronous events that alter the program flow-. Whenever an interruption/exception happens, the current task is suspended and its context is saved -using the stack pointer-, and then it is executed a routine that can be either an exception handler or an interrupt service routine (ISR). This procedure is managed by the Nested Vectored Interrupt Controller (NVIC) which provides between its characteristics a flexible exception and interrupt management: it processes signals/requests coming from peripherals and exceptions coming from the processor core allowing to enable or disable them by software. In addition, the NVIC unit provides a sorting on the interrupts according to a programmable priority level based on the particular program needs. A particular feature of the NVIC is the system timer -also known as the SysTick-. It is a 24-bit decrement timer which is fed from a referenced on-chip clock source. It has its own exception handler. It is used in order to execute precise delays. In ST microcontrollers, the SysTick can be invoked at the software level through the use of the Hardware Abstraction Layer (HAL) library.

1.1.2 STM32L476RG microcontroller overview

Having introduced most of the fundamental characteristics of the ARM Cortex M-4 architecture which are implemented within the microcontroller, this part provides some additional characteristics that are specific to the microcontroller used in this project.

- *Clock sources:* The different clock sources that can be configured for the microcontroller unit are the listed ones below. For this project has been configured the internal multi-speed oscillator MSI adjusting it into a frequency of 48 MHz. The MSI provides a good compromise in terms of power-consumption, speed & wake-up time.
    - External 4 – 48 MHz crystal oscillator (HSE).
    - Internal 16 MHz factory-trimmed RC (HSI16).
    - Internal 32 Khz low power RC (LSI)
    - External 32 kHz crystal for RTC (LSE).
    - Internal multispeed 100 kHz – 48 MHz oscillator (MSI)
    - System PLL (uses HSE, HSI16 or MSI) up to 80Mhz.

- *Peripherals:* The following items presents some of the peripherals that are integrated within the microcontroller unit. A complete list of all available peripherals can be found on [2]:
    - 16 x Timers: Including low-power timers, generic ones, and the Systick timer.
    - 1 x RTC: A real-time clock unit providing time-of-day clock/calendar plus alarm interrupts.
    - 3 x I2C: Interfaces for handling the communication between the MCU and I2C buses.
    - 6 x USART (2 x Low-power): Universal synchronous/asynchronous interfaces handlers.
    - 114 x GPIO: General purpose configurable I/O ports.

1.2 Software development tools

The STM32L476RG microcontroller has been programmed using the following development tools: The STM32CubeMX framework which provides an environment for making an initial configuration of the MCU, and the System Workbench which is an Eclipse-based IDE for developing programs for the ST microcontrollers. The following parts of this section make an overall description for each of these used tools.

1.2.1 STM32CubeMX framework

The STM32CubeMX framework [3] is a development tool for making an initial configuration of the microcontroller features including those external elements which are embedded in the microcontroller board (e.g. external crystals, user leds, etc). The configuration procedure using this software consists in the following steps: First, select the microcontroller board that is going to be used. Secondly, it is proceeded to the microcontroller configuration step in which it can be done the MCU peripheral configuration (e.g. GPIO, USART), a pinout-conflict resolution, and the clock sources configuration among other settings. Finally, it is generated the C code for the selected microcontroller configuration. The generated code then can be used for programming on different several integrative environments (for instance, the System Workbench IDE 4).

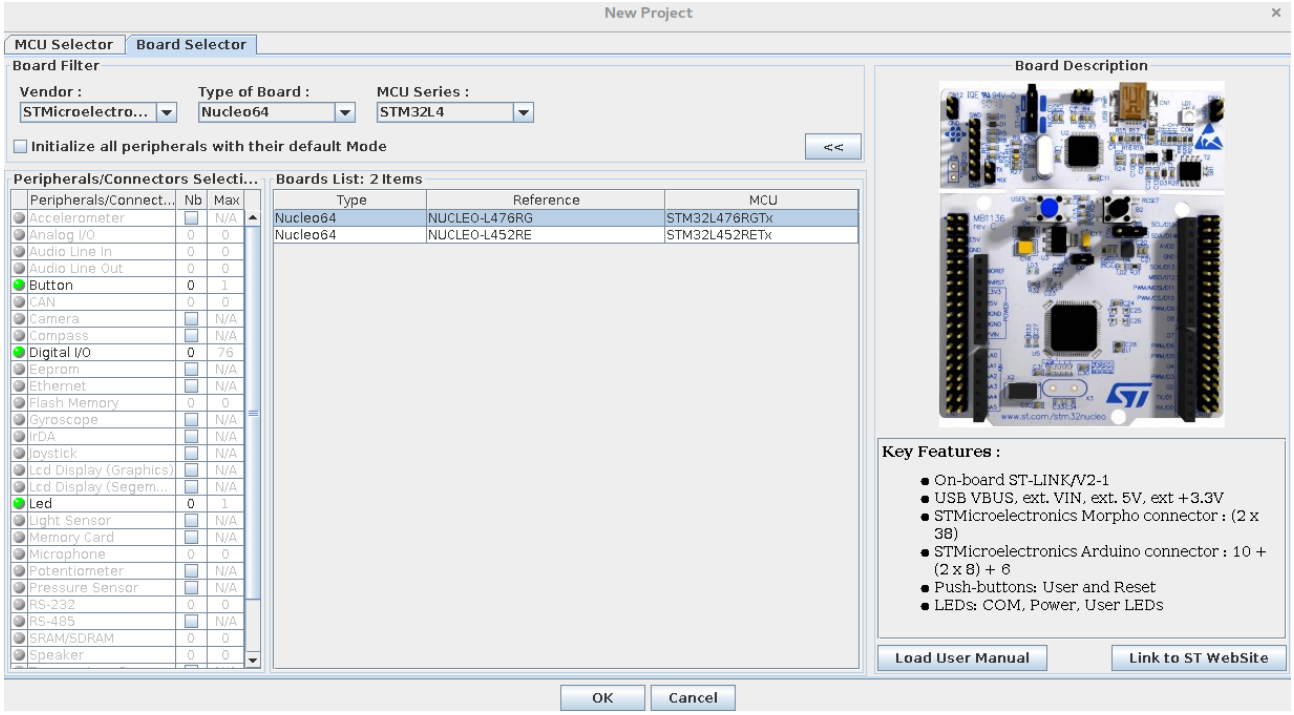

Figure 3. Microcontroller board selection on STM32CubeMX.

The figure presented above corresponds to the interface in which is selected the microcontroller board which is going to be used. For each different type of ST microcontroller that is used it is needed to download the necessary firmware through a wizard provided by the STM32CubeMX. Section 1.4 presents the following steps for the MCU general configuration and the C code generation applying them for case of this project.

1.2.2 System Workbench IDE 4

The System Workbench IDE 4 [4] is a software tool for developing programs for ST microcontrollers. It. The provides to the user a development environment based on Eclipse. Hence, it provides as well debugging capabilities which eases the development task. Whenever it is generated the project code from the STM32CubeMX with the desired MCU configuration it can be later imported as a project to the System Workbench in order to start working over it. Thus, the program code can be installed though the

microcontroller board through a USB connection between the microcontroller development board and the computer.

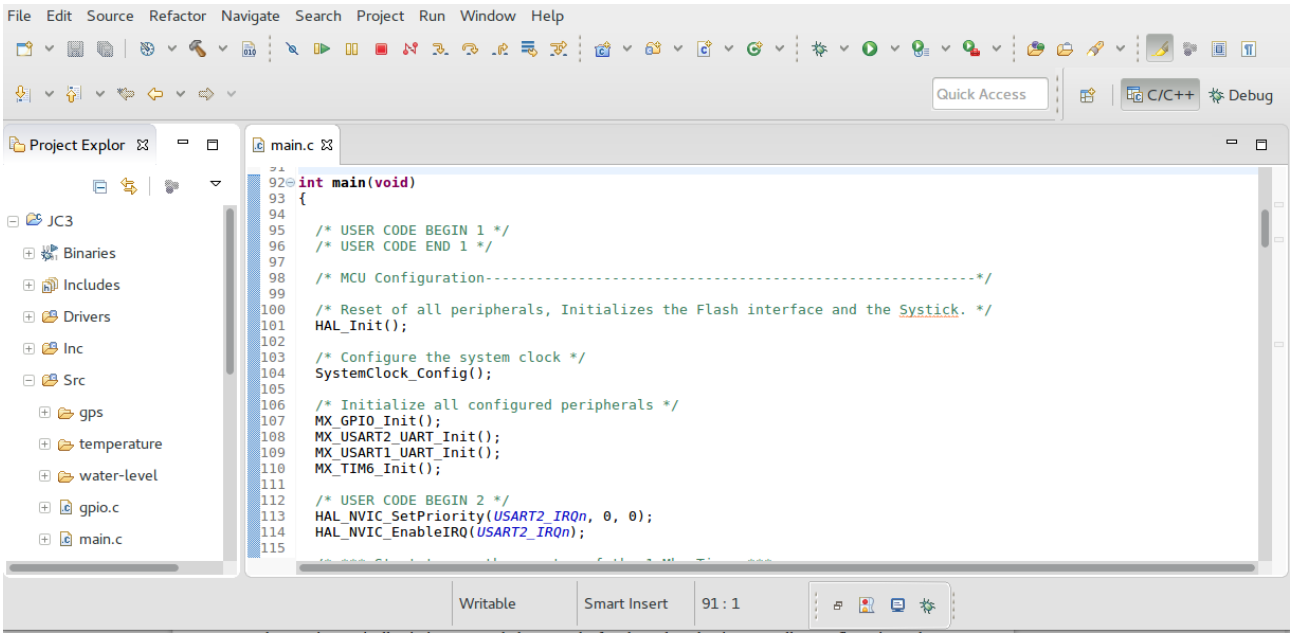

Figure 4. User interface of the System Workbench IDE 4.

1.3 System schematic diagram

This section presents a simplified schematic diagram of the hardware system configuration. It shows the wiring connection between the microcontroller board Nucleo-STM32L476RG and the different sensors included in the system. Not used pins were omitted in this figure in order to ease the diagram readability.

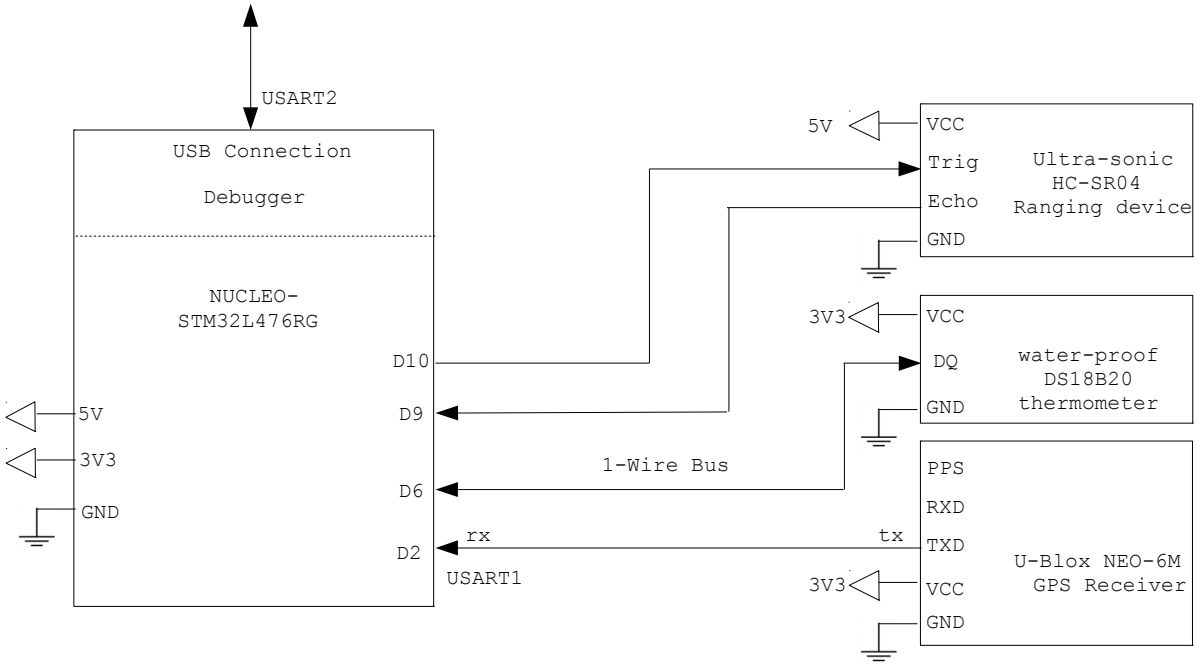

Figure 5. Connection between the MCU board and the different sensors used by the system.

The MCU port D2 is used for communicating with the U-Blox GPS Receiver. It is configured as an input pin since it receives the data from the GPS module. This line is managed by means of an UART interface -see Section 2-. The port D6 serves for connecting the microcontroller with the water-proof DS18B20 device in order to receive the temperature measurements. This line is managed by the 1-Wire protocol -see Section 3-. Finally, it is connected the MCU with the HC-SR04 device which calculates the distance between the top of the tank and the water surface. It is used the port D10 for sending a trigger signal to the device, and the port D9 for receiving an echo response pulse from the device. Later the microcontroller makes a calculation in order to get which is the current level of water -see Section 4-.

In the presented diagram, it is shown as well the debugging unit of the microcontroller board. Through this debugger is possible to interact with a computer using the USB interface. This is important for installing the system program and for debugging purposes. Through this channel the system can also be provided with the necessary power supply. Moreover, taking advantage of this channel it was developed a system console by which the user can interact in order to configure the system main variables and for printing the sensor measurements -see Section 5-.

1.4 Microcontroller settings configuration in STM32CubeMX

Having introduced the STM32CubeMX tool on Section 1.2, and having given a high-level view of the system through the schematic diagram, this part now proceeds to present the microcontroller settings configuration that was done through the STM32CubeMX tool. First of all, the first step of configuration required is to go to the pin-out tool section in which it can be managed the microcontroller peripherals. The following figure presents the resulted setting for the case of this project.

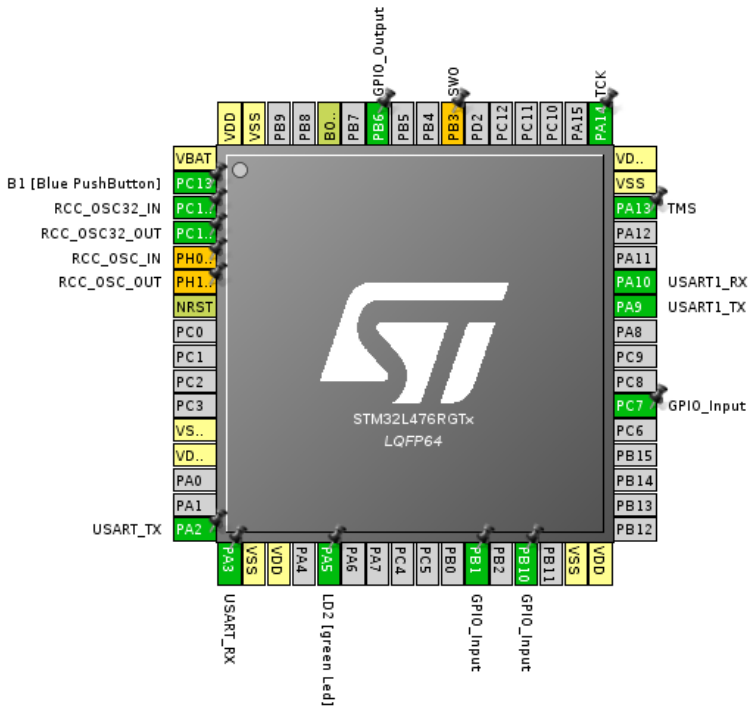

Figure 6. Microcontroller pinout configuration.

The presented view in the image refers to the microcontroller pin configuration. The most important assigned ports are the the following ones: The ports PA2 and PA3 that refer to the UART connection that goes towards to the computer through the USB cable for connecting it with the system terminal. The port PA5 is connected to the LED2 embedded in the Nucleo-STM32L476RG board. Ports PA9 and P10 are used by other UART interface which makes the connection with the GPS module. The port PB10 has been configured as a general purpose input/output port which will be used as the channel for connecting the MCU with the temperature

sensor. Ports PB6 and PC7 respectively serve for the connection of the Trigger and Echo lines for the ultra-sonic ranging device which is used for calculating the tank water level. There as well several Vss and Vdd ports on the microcontroller which respectively are the channels for the positive and negative voltage supply.

Having done the needed pin-out configuration the next step is to specify the system clock sources (through the clock tool tab) for this case the selected clock source is the internal multi-speed oscillator (MSI RC) configuring it into a frequency of 48000 kHz. As it can be seen in the next figure, it is selected this source in the system clock multiplexer. Hence, the microcontroller features such as timers or UART peripherals will operate at this frequency since it is also used as the peripheral clock.

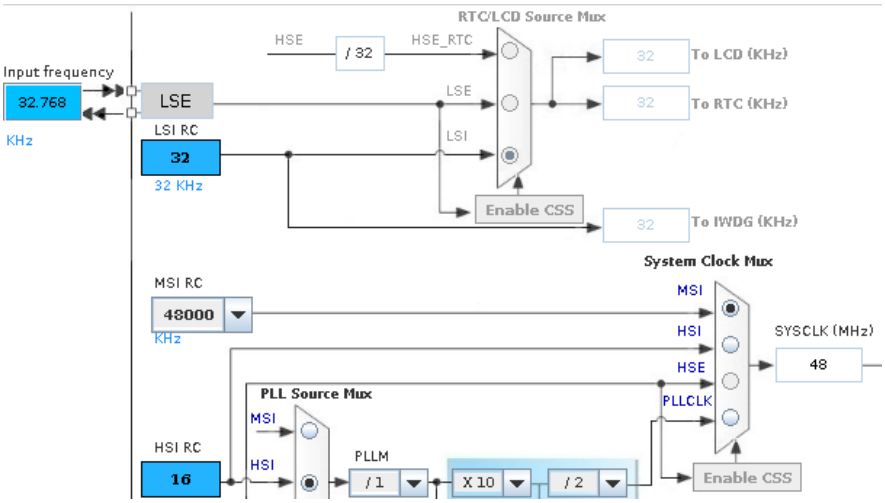

Figure 7. System clock configured to operate at 48 MHz.

Through this phase using the STM32CubeMX tool there is a core element that is configured which is the microcontroller timer TIM6. The STM32L476RG microcontroller provides several timers in order to carry out different tasks. For accomplishing some particular delay handling on the temperature and the water-level modules of this project -see Sections 3 and 4 respectively- is required a timer providing delays at a precision level of microseconds, and the system timer Systick only provides a resolution of milliseconds).

For achieving the required precision, it has been configured the MCU 16-bit timer TIM6 configuring it using the following strategy: Since the system clock source is using the MSI RC at 48 MHz which feeds this timer, then it is used a prescaler with a value of 48 which converts the clock signal source coming from the system clock into a frequency of 1 MHz which is 1 million of cycles per second. In this way, this timer can be used later in the project code to provide a delay function with a precision of microseconds. In particular, since this a 16 bit timer it will be able to count until a max value of 65535 microseconds, and then it will reset to 0. The counter mode has been configured to be count incrementally. The figure shown below presents the interface in which it was configured the timer.

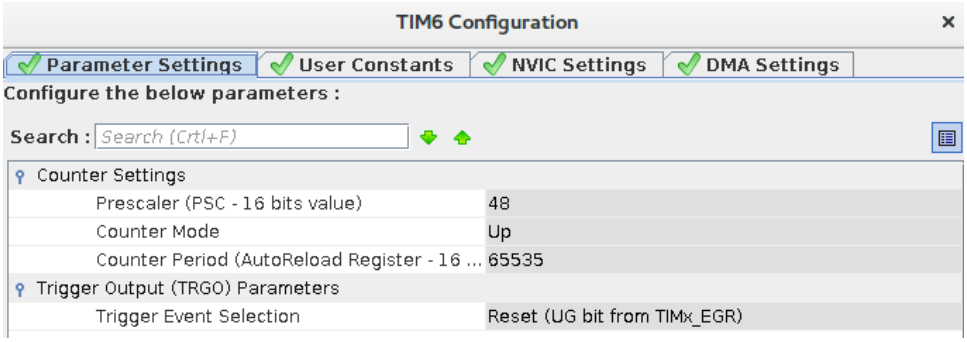

Figure 8. Configuration of a 16-bit timer.

At this point the microcontroller initial settings are completed. Thus, it is generated through the STM32CubeMX framework the associated C code which matches with the peripheral configuration done.

1.5 System modules and project files structure

The system project is abstractly organized in the following modules: A main procedure which has the program control, and it carries out the initialization of the system clock and the configured peripherals. It is in charge as well of the initializing the sensor handlers and to take the data from them. The next modules correspond to the GPS, the temperature, and the water-level handlers whose implementation are detailedly addressed in Sections 2, 3 and 4 respectively. The system comprises as well a system command line -see Section 5- for interacting with the user. All these modules are supported by the different features of the microcontroller such as the general peripherals for input/output, the UART interfaces, and the used clock sources. The following figure presents the described logical organization of the project modules.

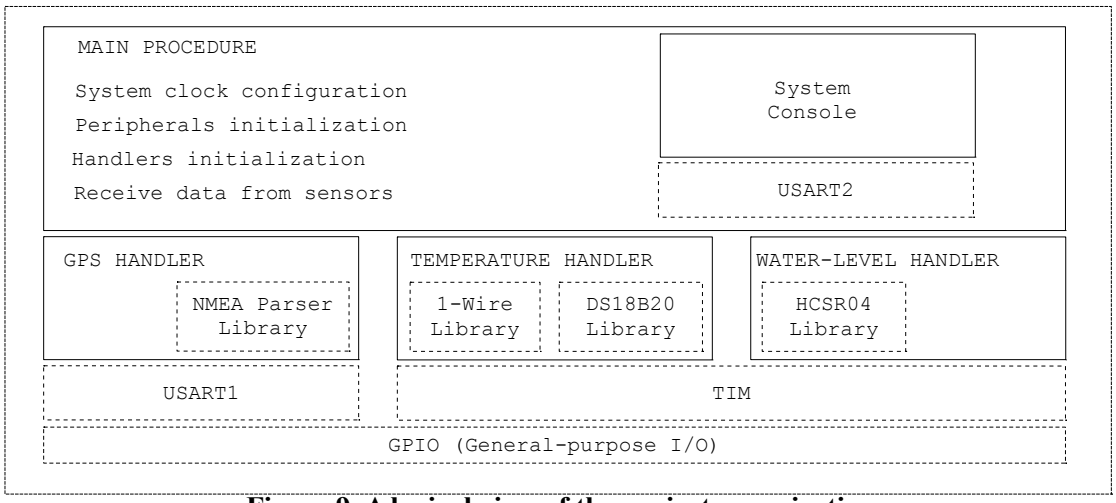

Figure 9. A logical view of the project organization.

Having presented the logical modules of the project, it is presented the project directory in which are described the most important files.

```
root
  ├ Drivers
  │ ├ CMSIS
  │ └ STM32L4xx_HAL_Driver
  │
  ├ Inc
  │ ├ gps
  │ │ ├ gps_handler.h
  │ │ └ minmea.h
  │ │
  │ ├ temperature
  │ │ ├ temperature_handler.h
  │ │ ├ tm_stm32f4_ds18b20.h
  │ │ └ tm_stm32f4_onewire.h
  │ │
  │ ├ water-level
  │ │ ├ waterlevel_handler.h
  │ │ └ tm_stm32f4_hcsr04.h
  │ │
  │ ├ gpio.h
  │ ├ main.h
  │ ├ sensor_variables.h
  │ ├ stm32l4x_hal_conf.h
  │ ├ stm32l4xx_it.h
```

```
| ├ syscalls.h
| ├ tim.h
| └ usart.h
|
├ Src
| ├ gps
| | ├ gps_handler.c
| | └ minmea.c
| |
| ├ temperature
| | ├ temperature_handler.c
| | ├ tm_stm32f4_ds18b20.c
| | └ tm_stm32f4_onewire.c
| |
| ├ water-level
| | ├ waterlevel_handler.c
| | └ tm_stm32f4_hcsr04.c
| |
| ├ gpio.c
| ├ main.c
| ├ stm32l4xx_hal_msp.c
| ├ stm32l4xx_it.c
| ├ system_stm32l4xx.c
| ├ tim.c
| └ usart.c
|
├ startup
| └ tm_stm32f4_onewire.c
├ Debug
└ STM32L476RGTx_FLASH.Id
```

The project directory includes the `Drivers` folder in which are placed two main subdirectories: First, it is located the `CMSIS` folder. Then, the second subdirectory is the `STM32L4xx_HAL_Driver` in which are located all the necessary files that make the the linking between the specified microcontroller model and the HAL (Hardware Abstraction Layer) library drivers procedures in order to encapsulate the particular microcontroller hardware structure for easing to the developer the coding task.

On the other hand, the `Inc` folder contains the project header files. Here it has been organized the sensors handlers modules (GPS, temperature, water-level) including the third-party libraries that they use in order to implement the necessary handling. On the other side, there as well the header files which the defines the methods for managing the microcontroller features that were previously configured in the STM32CubeMX such as the USART interfaces (`usart.h`), the timers (`tim.h`) and the initialized GPIO ports (`gpio.h`). This directory includes the header files for the system management such as the `stm32l4xx_it.h` which takes care of the interruptions handling. It is located as well the `main.h` header file which is the header definition for the main procedure in which are called the functions for initializing the system clock, peripherals and the sensors modules. Finally, the `sensors_variables.h` header file declares some variables which are used by the different modules. Those variables are initialized whenever the system starts, and they can be configured by the user through the configured system terminal -see Section 5-.

```
12  // *** SYSTEM SENSORS VARIABLES *********
13  uint8_t GPS_MODULE_ON;
14  uint8_t TEMPERATURE_MODULE_ON;
15  uint8_t WATERLEVEL_MODULE_ON;
16  uint16_t SAMPLING_PERIOD;
17  float_t WATERTANK_HEIGHT;
18  float_t TEMPERATURE_MAX_THRESHOLD;
19  float_t TEMPERATURE_MIN_THRESHOLD;
20  float_t WATERLEVEL_MAX_THRESHOLD;
21  float_t WATERLEVEL_MIN_THRESHOLD;
```

Figure 10. Extract from the sensor_variables.h header file.

The `Src` directory includes the files where are implemented most of the methods that are specified in the header files located in the `Inc` folder. For instance, all methods defined for the different sensors modules header files are implemented within this directory. In particular, it is implemented the main procedure file `main.c`. In this directory is located also the system_stm32l4xx.c file which implements the system initialization function for clock and vector table initialization plus other clock utilities. From the root directory it can be also found the `startup` folder which includes the `startup_stm32l476xx.s` file which is written in assembly, and it holds the reset handler (the first code to be executed) and the vector table. In addition, there is located the linker script where it can be found the program and data memory (RAM/RAM2, FLASH) with respect to the linear memory map of the microcontroller.

1.6 System program main flow

In this part of the section it is done a general description of the main flow in order to introduce the sequence of procedures that are executed in order to carry out the sensors monitoring. The following code corresponds to the main procedure in the `main.c` file which handles the system main control.

```
92  int main(void){
93     /* MCU Configuration----------------------*/
94     /* Reset peripherals, Initializes the Flash interface & the Systick.*/
95     HAL_Init();
96     /* Configure the system clock */
97     SystemClock_Config();
98     /* Initialize all configured peripherals */
99     MX_GPIO_Init();
100    MX_USART2_UART_Init();
101    MX_USART1_UART_Init();
102    MX_TIM6_Init();
103
104    /* USER CODE BEGIN 2 */
105    HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
106    HAL_NVIC_EnableIRQ(USART2_IRQn);
107    HAL_TIM_Base_Start(&htim6);
108    INITIALIZE_SENSOR_VARIABLES();
109    /* USER CODE END 2 */
110    /* Infinite loop */
111    /* USER CODE BEGIN WHILE */
112    while(1){
113       DISPLAY_MENU();
114       INITIALIZE_SENSOR_HANDLERS();
115       uint8_t esc = 0x00;
116       while(1){
117          HAL_UART_Receive_IT(&huart2, &esc, 1);
118          if(interrupt_flag){
119             interrupt_flag = RESET;
120             if(ESCAPE_CHARACTER(esc)){
121                HAL_UART_Abort_IT(&huart2);
122                break;
123             }
124          }
125          RECEIVE_DATA_FROM_SENSORS();
126       }
127    }
128    /* USER CODE END WHILE */
129 }
```

Figure 11. Main procedure in the main.c file.

In ll. 92 – 102 is carried out the MCU system initialization configuring the hardware abstraction layer library, the system clock, and the initialization of the chosen peripherals for using in this project (the UART interfaces, the initialization of the configured general input/output ports, and the initialization of the timer selected that will be used within the temperature and water-level modules). In ll. 105 – 106 is configured the interruption management of the USART2 interface.

Later, in l. 108 is called the function `INITIALIZE_SENSOR_VARIABLES()` which basically will set default values over the system sensors variables that are declared in the `sensors_variables.h` header file. By default, all modules (GPS, temperature, and water-level) will be turned on, the sampling period will have a default value of 1000 milliseconds, but the water-tank height and the thresholds will remain uninitialized. Later, inside the outer *while* statement it is executed the `DISPLAY_MENU()` routine on l. 113. This function will proceed to connect with the system terminal -see Section 5- through the UART channel for reading he system sensor variables that the system user may configure. Once that this routine finishes, and the sensor variables are configured according to those read values, it is called the `INITIALIZE_SENSOR_HANDLERS()` method. In that routine, it will be turned on or off the sensors handler modules in base to the options received by the previous function. If the methods are turned on they will be calibrated with the passed options (e.g. water-tank height, thresholds levels, etc).

Then, on the inner *while* statement at ll. 116 – 126 the system will start to receive the data from the sensors and it will forward this data towards the system terminal. This will be done according to the sampling period that was previously configured. On the other hand, whenever it may be received an interruption signal -by the means of a key pressed from the user- the system will stop to receive & print data from the sensors, and the program control will go back to the outer loop displaying again the console main menu.

Following sections of this report provide a detailed information for each of the corresponding sensor modules: A description of each device, and technical explanation about how the system carries out the signaling mechanism in order to retrieve the sensors measurements.

2 The GPS module

The system comprises a GPS receiver module that provides geo-location services to the monitoring system. This section provides information about the device used and the communication standard it uses. It is explained the configuration for connecting the device with the microcontroller based on an UART channel, and finally are presented the parser module and the logic structures used at the software level to store the relevant information of the GPS signals.

2.1 GPS receiver device

The GPS receiver module is a NEO-6M GPS device [5]. It belongs to a family of stand-alone GPS receivers produced by U-blox. It operates within a supply voltage range between 2.7 V – 3.6 V. It works with the NMEA 0183 data communication standard -explained in the following section-. The module provides as well one configurable UART interface for serial communication that it was the interface used in order to connect it with the microcontroller.

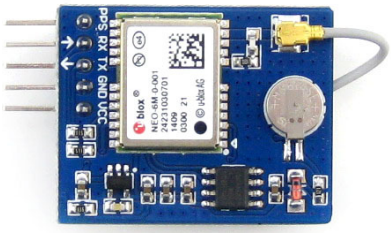

Figure 12. U-blox NEO 6-M GPS receiver module.

2.2 NMEA standard

The NMEA standard [6] is an specification which defines the communication between electronic devices (sonars, autopilots, GPS receivers, etc.). It has been developed by the National Marine Electronics Association -an US-based marine electronics trade organization-. In particular, the receiver used within this project uses the NMEA 0183 version which is used as well by other GPS receiver modules. Nevertheless, in marine applications this version has been slowly phased out in favor of the new version NMEA 2000.

The idea of the NMEA standard is to send a line of data which is called a sentence which it is independent from other sentences in terms of the localization information it provides. All of the NMEA sentences have a two letter prefix that defines the device which is transmitting the sentences (in our case, GP since we are working with the GPS receiver), and it is followed by a three letter sequence that defines the type of the sentences. The following table enumerates the sentences that are received by the NEO-6M GPS device.

NMEA sentence	Definition
GPGGA	Essential fix data which provides 3D location and accuracy.
GPGLL	Geographic latitude and longitude.
GPGSA	DOP and active satellites. It provides the being used satellites status data.
GPGSV	Satellites in view. It shows data about the satellites that the module might find.
GPRMC	Recommended minimum sentence. Essential position, velocity, and time.
GPTXT	Useful information of the receiver (power-up screen, software version, etc).

Table 1. NMEA sentences received from the NEO-6M GPS device.

Thereby, a set composed by several sentences -each of them belonging a particular type- conforms a NMEA message which is received and transmitted by the GPS receiver device. Each of the sentences are formatted using the ASCII representation. Each of them begins with the '$' character and concludes with the carriage return and the new line characters. The sizes of each sentence do not exceed 80 characters, and the data contained are separated by commas. In addition, each sentence provides a checksum -separated from the rest of the line by a '*' character- which is a two-digits hexadecimal number whose value represents the operation of applying a 8-bit exclusive OR between all characters (without including the initial character '$' and the '*' separator).

The example below is the structure of a GGA sentence plus the description of its fields. Particularly for the implemented system, this is one of the most important sentences that it is received within a NMEA message since it gives the coordinates which tracks the current position of the sensoring system.

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
Where:
     GGA                Global Positioning System Fix Data
     123519             Fix taken at 12:35:19 UTC
     4807.038,N         Latitude 48 deg 07.038' N
     01131.000,E        Longitude 11 deg 31.000' E
     1                  Fix quality: 0 = invalid, 1 = GPS fix (SPS), … , 7 =
                        Manual input mode, 8 = Simulation mode
     08                 Number of satellites being tracked
     0.9                Horizontal dilution of position
     545.4,M            Altitude, Meters, above mean sea level
     46.9,M             Height of geoid (mean sea level) above WGS84 ellipsoid
     (empty field)      time in seconds since last DGPS update
     (empty field)      DGPS station ID number
     *47                the checksum data, always begins with *
```

As it is explained in Section 2.4 of this report, the microcontroller program has a routine which is in charge of make the corresponding parsing of all sentences that arrive within a NMEA message, and after that it stores the sentences information within a suitable data structure.

2.3 UART-based communication handling

The GPS receiver provides an UART interface -composed by a pair of input & output ports- used in order to send the data from the device towards the microcontroller. Note that the input port is not used since there is no need of communication flowing from the microcontroller towards the GPS device. The UART channel between the microcontroller and the GPS device is handled by means of the USART1 interface which is managed in software by the `huart1` data structure. This UART channel is configured by connecting a transmission line between the GPS receiver output pin and the microcontroller PA10 pin for reception. The following code is an extract of the `usart.c` file, and it shows the initialization procedure for this channel. (In that file is also contained the code to initialize the PA10 pin).

```
49  void MX_USART1_UART_Init(void) {
50     huart1.Instance = USART1;
51     huart1.Init.BaudRate = 9600;
52     huart1.Init.WordLength = UART_WORDLENGTH_8B;
53     huart1.Init.StopBits = UART_STOPBITS_1;
54     huart1.Init.Parity = UART_PARITY_NONE;
55     huart1.Init.Mode = UART_MODE_RX;
56     huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
57     huart1.Init.OverSampling = UART_OVERSAMPLING_16;
58     huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
59     huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
60     if (HAL_UART_Init(&huart1) != HAL_OK) {
61        Error_Handler();
62     }
63  }
```

Figure 13. Initialization of the USART1 peripheral in the usart.c file.

This UART channel initialization is called within the file `main.c`. The USART1 is set on 9600 baud, a word-length of 8 bits -number of bits transmitted in an UART frame-, one stop bit -signaling the end of the frame-, and no parity bit. Hence, this initialization complies with the NMEA 0183 version and the GPS device capabilities. The interface is set in receiving mode. The `HwFlowCtl` attribute is set to none since this is only useful when there is a RS232 flow control, and the `OverSampling` attribute which is configured to `UART_OVERSAMPLING_16` means that the handler will perform 16 samples for each frame bit. The NMEA messages are sent through the channel each second in which it can be received up to 960 characters (given the configured baud). Since each sentence is composed at most by 82 characters, it can be received at most 11 different sentences in each message. The communication through the UART channel is managed by the GPS handler module `gps_handler` which is initialized and used in the file `main.c`. After the initialization of this handler, it is called a receive method (`GPS_HANDLER_Receive`) to get 1 byte sequentially until it has been received the full message; after that moment, the received information can be processed (e.g. print the system coordinates). The next flow chart shows a simplified view of the implemented usage of the GPS handler module.

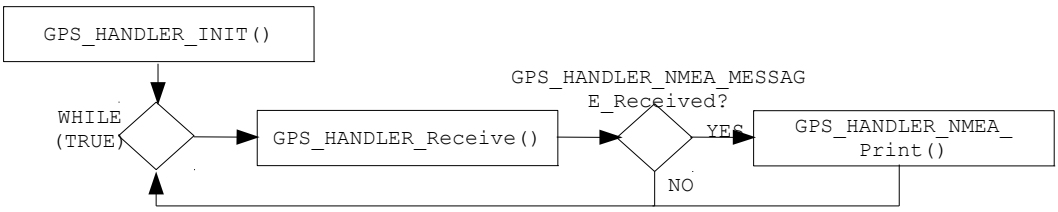

Figure 14. Usage of the GPS handler in the main procedure.

The following code illustrates the implemented reception procedure within the GPS software handler for the NMEA messages through the USART1 interface.

```
23  void GPS_HANDLER_Receive(){
24     HAL_StatusTypeDef status = HAL_UART_Receive(&huart1, &gps_c, 1, 100);
25     NMEA_MESSAGE_ARRIVED = false;
26     if(status != HAL_BUSY && status != HAL_ERROR){
27        if(status != HAL_TIMEOUT){
28           gps_buffrx[gps_p] = gps_c;
29           gps_p++;
30        }else{
31           if(gps_p > 0) {
32              GPS_HANDLER_Parse_Buffrx();
33           }
34           memset(gps_buffrx, 0, gps_p);
35           gps_p = 0;
36        }
37     }
38  }
```

Figure 15. GPS data reception procedure through the USART1 interface.

It is used a polling strategy in which it is waited for reading a character with a time-out of 100 milliseconds. Each received byte is stored in a reception buffer (the `gps_buffrx` variable). If when receiving a byte it is produced a time-out, then it means that it has finished to read a complete NMEA message, and it can proceed to parse the reception buffer. There might be the case in which the GPS handler starts to read in the middle of two NMEA messages, but if that happens the counter of bytes received `gps_p` would be zero since the reception buffer is empty, and therefore it will not go to the parsing method.

2.4 NMEA messages parsing and storage

Once a NMEA message is fully received by the microcontroller, it is handled by means of the `GPS_HANDLER_Parse_Buffrx()` routine. Within that routine, two important processes are carried out: The parsing of the NMEA sentences of the message, and their subsequent storage within suitable data structures.

For the task of the sentences parsing it is used the MINMEA library [7]. It is a light-weight GPS NMEA 0183 parser library written in pure C language -independent of any platform- which is intended for resource-constrained systems such as microcontrollers. For the second task it has been developed a pair of data structures in order to store the NMEA sentences. The following code shows the mentioned structures which are defined inside the `gps_handler.h` header file.

```
20  typedef struct {
21     struct minmea_sentence_gga gga;
22     struct minmea_sentence_gsa gsa;
23     struct minmea_sentence_gsv gsv;
24     struct minmea_sentence_gll gll;
25     struct minmea_sentence_rmc rmc;
26     struct minmea_sentence_vtg vtg;
27     struct minmea_sentence_gst gst;
28     } nmea_sentences;
29  nmea_sentences last_valid_nmea_sentences;
...
38  typedef struct {
39     struct minmea_time time;
40     struct minmea_float latitude;
41     struct minmea_float longitude;
```

```
42      struct minmea_float altitude;
43      char altitude_units;
44      bool valid;
45  } _nmea_preserved_data;
46  _nmea_preserved_data nmea_preserved_data;
```

Figure 16. Data structures for storing the NMEA sentences.

The structure type `nmea_sentences` is composed by a set of structures defined by the MINMEA library. Each structure stores the sentence illustrated by its mnemonic (e.g. all the data of GGA sentence is allocated in the `struct minmea_sentence_gga gga` structure). Thus, the complete data structure is instantiated into the variable `last_valid_nmea_sentences` whose purpose will be to store sentences that are received. However, it only stores valid sentences. To check for the validity of the sentences it is used a verification procedure which includes the use of their checksum. Thereby, whenever a NMEA message arrives, if the verification procedure finds an invalid sentence, that sentence will not be stored in `last_valid_nmea_sentences`.

The data structure `_nmea_preserved_data` stores the data that it is required to be always available. This data is the geographic coordinates (latitude, longitude and altitude) and its sampling time. When in a NMEA message this information arrives (which is searched within the GGA and GLL sentences), it is stored within the `nmea_preserved_data` record. Then, the record will be updated whenever the coordinates information arrives again in a NMEA message. The motivation of having this structure is to keep track of which was the last valid system position, such that this information will not be lost in future message receptions which do not provide the coordinates information.

The following flow chart describes the operations that are carried out inside the procedure `GPS_HANDLER_Parse_Buffrx()` in order to make the respective parsing (using the MINMEA library) and the posterior storage of the data (within the described structures).

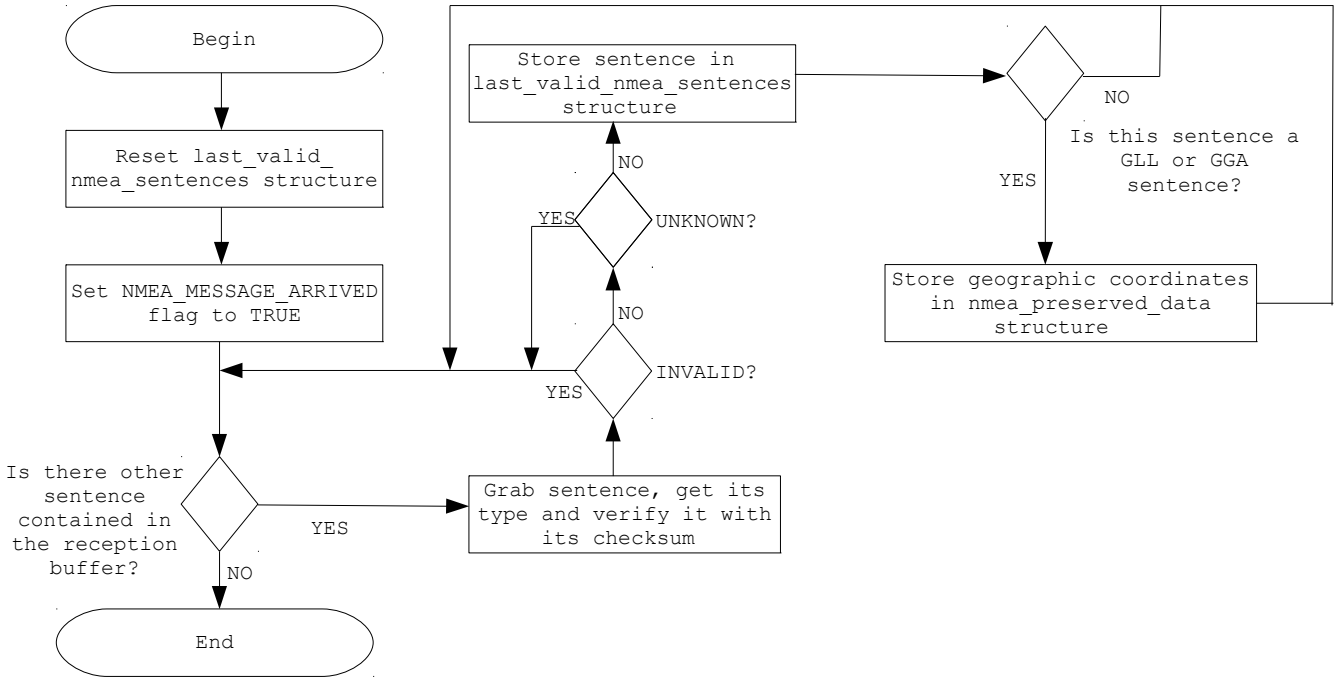

Figure 17. Sentences parsing and storage procedure.

2.5 A note about formatting the geographic coordinates for printing

The geographic coordinates arriving from the GLL or GGA sentences of a NMEA message come in a predefined decimal notation. Therefore, the GPS handler procedure `GPS_HANDLER_NMEA_Print()` makes a format conversion over the coordinates into a decimal degree notation before printing them on the standard output. For both cases, latitude and longitude, the conversion procedure goes as follows: given the latitude or longitude value separate the first (the most significant ones) two-digits of the integer part. Divide the rest of the number by 60, and then sum the division result to the first two digits that were initially separated. The following example illustrates the described procedure:

```
Latitude:     4807.038   → 48 + (07.038)/60  =  48 +  0.1173 → Latitude    = 48.1173
Longitude:    1131.000 → 11 + (31.000)/60 = 11 +   0.5167    → Longitude = 11.5167
```

3 Underwater temperature module

The temperature module aims for monitoring the tank water temperature. For accomplishing this task, it is used a water-proof DS18B20 temperature sensoring device which goes immersed below the water. This section makes an introduction about the device; it is introduced the 1-Wire standard which is the protocol used for communicating the microcontroller with the temperature sensor device. Then, it is described the connection between the microcontroller and the device having as a support the 1-Wire standard and some libraries which implement both the 1-Wire as well as the temperature device signaling mechanism.

3.1 The DS18B20 temperature sensor

The device used for measuring the water temperature is a water-proof DS18B20 temperature device [8]. The device is protected by a stainless steel probe head which makes it suitable for wet or harsh environments. It is able to measure temperatures between -55 °C up to +125 °C with an accuracy of ± 0.5 °C. It provides a programmable temperature sampling resolution from 9 to 12 bits.

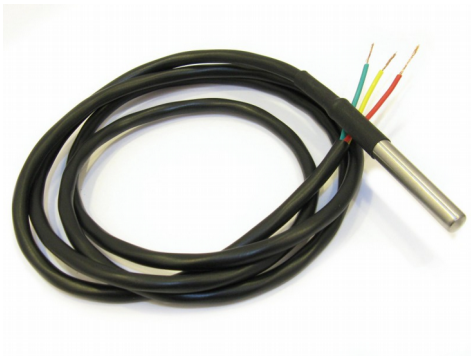

Figure 18. DS18B20 Water-proof temperature sensor

The device comprises three lines: The power supply voltage -working in a range of 3.0V up to 5.5V-, the ground line, and the data in/out transmission line. It is managed using the 1-Wire standard which works as a communication bus system by which many sensor devices can be connected. In each of the DS18B20 devices is stored an unique serial number, so this eases the task of recognizing which device is using the 1-Wire bus for sending data in a particular moment.

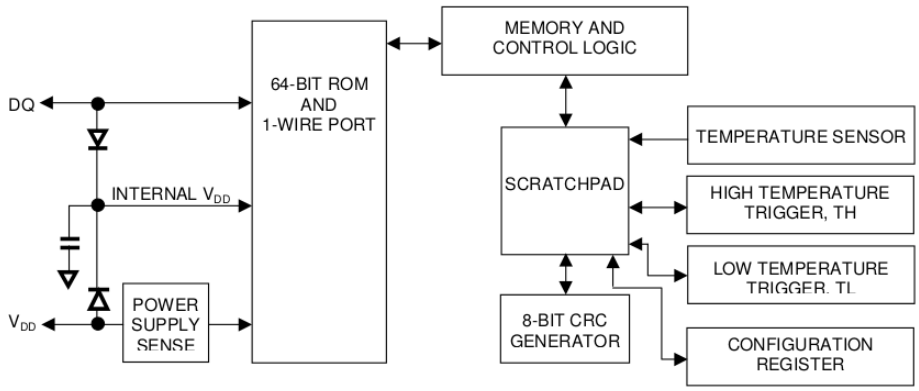

Figure 19. Internal DS18B20 components.

The figure showed above presents the organization of the device internal components. It has four main components: A 64-bit lasered ROM, the temperature sensor unit, non-volatile temperature alarm triggers (each of them is a 1-byte EEPROM), and a configuration register (used for setting temperature resolution). The 64-bit ROM stores an identifier which uniquely identifies each possible device. The first 8 bits are the device family code (DS18B20 is 0x28), and the next 48 bits are an unique code. The last byte is a CRC computed on the first 56 bits which can be used to determine if the ROM data has been received error-free.

Whenever is triggered the corresponding function for marking a temperature measurement, the result is placed in the scratch-pad memory module (within a 16-bits area). The result can have a resolution from 9 to 12 bits (programmable through the configuration register). The most significant bits indicate the sign value whereas the four least significant bits will store the fractional part of the value. Hence, having up to 4 bits for the fractional part we can have precision equating to resolutions of 0.5, 0.25, 0.125 and 0.0625.

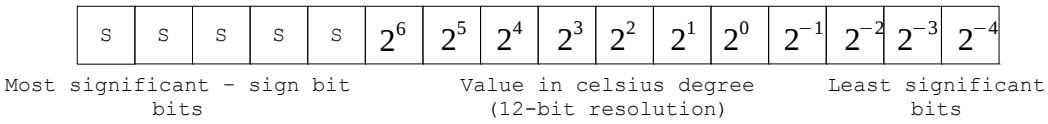

Figure 20. Temperature value bits stored in the scratch, and its representation.

3.2 A general view over the 1-Wire Protocol

The 1-Wire protocol is a communication standard designed by Dallas Semiconductors conceived to work under a bus system -a single conductor- which is shared by several devices. The aim of having developed a bus system approach is for providing the opportunity to manage different devices through a common channel (e.g. a group of temperature sensors over one bus connected towards the microcontroller). As it has been described in the previous section, it is the standard used to manage the communication between the microcontroller and the the DS18B20 device.

Within the communication procedure, there is one bus master which is the microcontroller and one or more slaves which are connected to the sensoring devices connected to the bus. The protocol for accessing the DS18B20 comprises the following phases: The initialization, the ROM function command, the memory function command, and the transaction/data.

The initialization phase consists in sending a reset pulse (of at least $480us$) transmitted by the bus master followed by the presence pulse transmitted by the slaves. Those presence pulses let the master know that the slaves devices are on the bus and ready to operate. Once the master has detected this slave device presence, it can yield a ROM function command such as searching the ROM of the devices, or for the case of having just

one slave in the bus to directly invoke a read ROM operation. After the ROM reading phase, it can be carried out some memory function commands (read or set conversion resolution, read or set temperature alarms, get temperature measurement, etc). A detailed view of the memory function commands can be found in [8].

3.3 Communicating with the sensor & temperature monitoring handling

Having introduced the DS18B20 device, and the 1-Wire protocol used for communicating it with the microcontroller, this section proceeds to present the implemented strategies for interacting with the temperature device by means of the correct microcontroller hardware configuration, and the corresponding management at the software level using as well some third-party handlers. At the hardware level, it is connected the sensor device transmission data line -see Figure 21- to the microcontroller pin PB10 (D6). On the other hand, the following figure presents the particular file section of *gpio.c* in which is initialized the microcontroller pin used.

```
54  /*Configure GPIO pin : PB10 */
55  GPIO_InitStruct.Pin = GPIO_PIN_10;
56  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
57  GPIO_InitStruct.Pull = GPIO_PULLUP;
58  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

Figure 21. Pin initialization for communication with the temperature sensor.

It is important to take into account that for this configuration it is needed to set the value `GPIO_PULLUP` for the `Pull` attribute. In the scope of electronics logic circuits, pull-up strategies aim to ensure that wires will be at a defined logic high level for the cases in which logic circuits are inactive. For the particular case of the 1-Wire protocol, it is required to have the control signal pulled high so the master device can pull it low to ask for data, and the slave device can pull it low to give the data. Moreover, this strategy is one of the features that allow the possibility of having multiple slaves sharing the same bus. The STM32L476 microcontroller has an internal programmable pull-up resistor giving the benefit of reducing the external circuitry required. Thereby, the instruction showed above of `GPIO_InitStruct.Pull = GPIO_PULLUP` indicates to active the described pull-up internal mechanism for the pin used.

Having done the correct hardware configuration between the microcontroller and the sensor device, and the corresponding initialization of the microcontroller pin, the following part of this section describes the procedure of handling the communication at the software level. For that purpose, following the same approach of the other modules, it has been developed the `temperature_handler` module. This module has been supported using a library for handling the 1-Wire protocol [9], and a second library which directly implements the signaling mechanism with the DS18B20 device [10]. The following figure presents a simplified view of the module header file in which are presented the main variables and methods.

```
11 #include "tm_stm32f4_onewire.h"
12 #include "tm_stm32f4_ds18b20.h"
13
14 uint8_t temperature_device[8];
15 float_t temperature_value;
16 TM_OneWire_t OneWire1;
17
18 uint8_t TEMPERATURE_ALARM_MIN_THRESHOLD_REACHED;
19 uint8_t TEMPERATURE_ALARM_MAX_THRESHOLD_REACHED;
20
21 void TEMPERATURE_HANDLER_INIT(void);
22 void TEMPERATURE_HANDLER_Receive(void);
23 void TEMPERATURE_HANDLER_Print(void);
```

Figure 22. Scheme of the temperature module header file -simplified view-.

In ll. 11 – 12 of the presented figure are included the libraries used for developing the module. Later, in ll. 14 – 16 are shown the variables used for the module: The `temperature_device` variable is used to store the 64-bits identifier of the DS18B20 device. The `temperature_value` serves as a reception buffer in order to allocate the last temperature measurement taken from the sensor which is taken as a single precision real number -following the IEEE754 standard-. The structure `OneWire1` is a data structure which manages several variables related to the communication between the DS18B20 and the microcontroller through the 1-Wire channel. The module permits to assign thresholds such that whenever a maximum or minimum temperature level has been reached, the system will dispatch an alarm indicating this event; for this purpose, in ll. 18 – 19 are declared two flags whose values are either true or false whenever the respective thresholds are passed. Hence, there are the `TEMPERATURE_MAX_THRESHOLD` and `TEMPERATURE_MIN_THRESHOLD` variables -located in the *sensor_variables* header file- which are the temperature thresholds levels that may be configured at the system start-up.

Finally, in ll. 21 – 23 of the presented code are declared the methods implemented for this module: the initialization routine `TEMPERATURE_HANDLER_INIT` which is in charge of initializing the `OneWire1` structure for starting the communication through the 1-Wire bus, getting the 64-bit identifier from the ROM memory of the DS18B20 device, and setting the device temperature measurement resolution to work which is of 12 bits. The `TEMPERATURE_HANDLER_Receive` procedure which is in charge of receiving a measurement from the sensor device, and for checking whether or not the current temperature value has passed any of the configured thresholds. If the latter happens, the corresponding thresholds alarm flags will be updated. The `TEMPERATURE_HANDLER_Print` prints to the standard output the last temperature value read from the sensor device, and dispatch an alarm if any of the thresholds flags are activated.

```
11    void TEMPERATURE_HANDLER_INIT(void){
12       TM_OneWire_Init(&OneWire1, GPIOB, GPIO_PIN_10);
13       TM_OneWire_First(&OneWire1);
14       TM_OneWire_GetFullROM(&OneWire1, temperature_device);
15       TM_DS18B20_SetResolution(&OneWire1,temperature_device,
16        TM_DS18B20_Resolution_12bits);
17
18       TEMPERATURE_ALARM_MIN_THRESHOLD_REACHED = RESET;
19       TEMPERATURE_ALARM_MAX_THRESHOLD_REACHED = RESET;
20       temperature_value = 0.0;
21    }
```

Figure 23. Temperature handler initialization routine.

As an example, the previous figure presents the `TEMPERATURE_HANDLER_INIT` procedure. It can be seen how the connection mechanisms that were explained in Section 3.2 (e.g. for accessing the sensor memory, for taking the ROM identifier, assigning the device precision, etc.) is encapsulated by means of the procedures implemented by the library: First, it makes an initialization of the `OneWire1` structure whose purpose is to serve as a medium for managing the 1-Wire channel. It is given as a parameter the identifiers of the GPIO port to be used (B10). Later, it is retrieved the 64-bits identifier of the temperature device to be used, and then it is configured to the precision which is going to work the temperature device that it is going to be of 12 bits. Finally, the thresholds flags and the `temperature_value` variable are initialized. Thus, after having call this procedure the system can use the method `TEMPERATURE_HANDLER_Receive` for starting to take the measurements done by the sensor device.

On the other hand, what respects to the third-party libraries used for managing the 1-Wire channel [9] and for the specific DS18B20 mechanism [10] few changes were done in order to adapt it to the STM32L476 microcontroller since those libraries were based for STM32F4 microcontroller models. The most significant change that was done in both libraries was to make them work with a system own timer whose counter works in the order of microseconds. This is needed since as it has been explained in Section 3.2 the 1-Wire bus

synchronization mechanisms work at the level of microseconds. In Section 1 it is presented a description of the configuration done for this timer which is used as well for the water-level monitoring module.

4 Water-level monitoring module

The water-level module is in charge of monitoring the level of water (measured in centimeters) of the tank by means of using ultra-sonic waves which are produced by the HC-SR04 device [11]. The sensor is positioned at the top of the tank, and sends the ultra-sonic waves towards the water surface where immediately are reflected back to the device. Thus, this mechanism can derive the distance between the device and the water surface. Therefore, knowing in advance the tank height it can be calculated which is the current tank water level making a subtraction between the tank height and the measured distance between the device and the water surface. This section provides a general description of the hardware device, the communication with the microcontroller, and the implemented handling at the software level.

4.1 Ultra-sonic ranging device

For calculating the distance between the top of the tank and the water surface it is used the HC-SR04 module (positioned at the top) which provides a non-contact distance measurement function which works between the range of 2 cm – 400 cm. Its range of accuracy is ± 3 mm. The following figure presents the device with its respective ports which are managed as follows: Vcc which is the 5V supply, the trigger line (*Trgh*) which is the pulse input which is received from the microcontroller, the the echo pulse (*Echo)* which instead is the pulse output that goes towards the microcontroller, and finally the 0V ground line.

Figure 24. HCSR04 device for ultra-sonic ranging measurement.

For measuring the distance with respect to an obstacle (for this particular case, the water surface), it is required to supply through the trigger input line a pulse of 10 microseconds. Then, it will start the ranging sending out an 8 cycle burst of ultrasound at 40 kHz. Whenever the sensor receives the signal back, it will set a high pulse through the echo output line whose pulse width will be proportional to the distance between the sensor and the obstacle. The following timing diagram illustrates the described procedure.

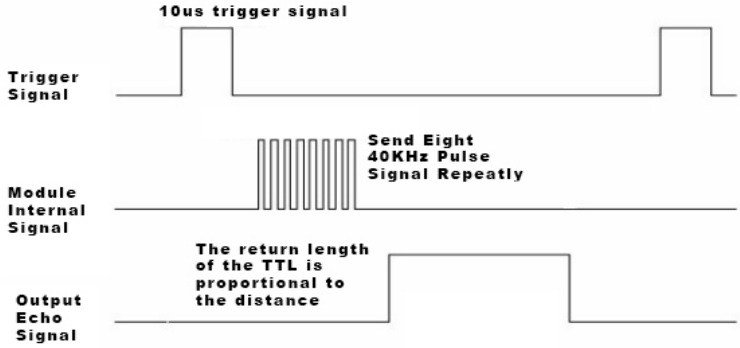

Figure 25. HCSR04 device timing diagram.

Finally, for calculating the distance between the module and the obstacle, the echo pulse width (in microseconds) is divided by 58 which giving the distance in order of centimeters.

4.2 Water-level variables configuration

For managing the device and retrieving the measurements, the project includes at the software level the `waterlevel_handler` module. It has been developed using the same strategy of the temperature module: The module is initialized having received the initial parameters, and then for retrieving the tank water level it is called the receive procedure according to the sampling period configured by the user.

Nonetheless, for this modules there are different features to be commented: as it was explained, the device measures the distance between the top and the water surface. Thus, it is needed to know before which is the tank height in order to make a subtraction between that height value `WATERTANK_HEIGHT` and the calculated distance *d* for getting which is the current water level .

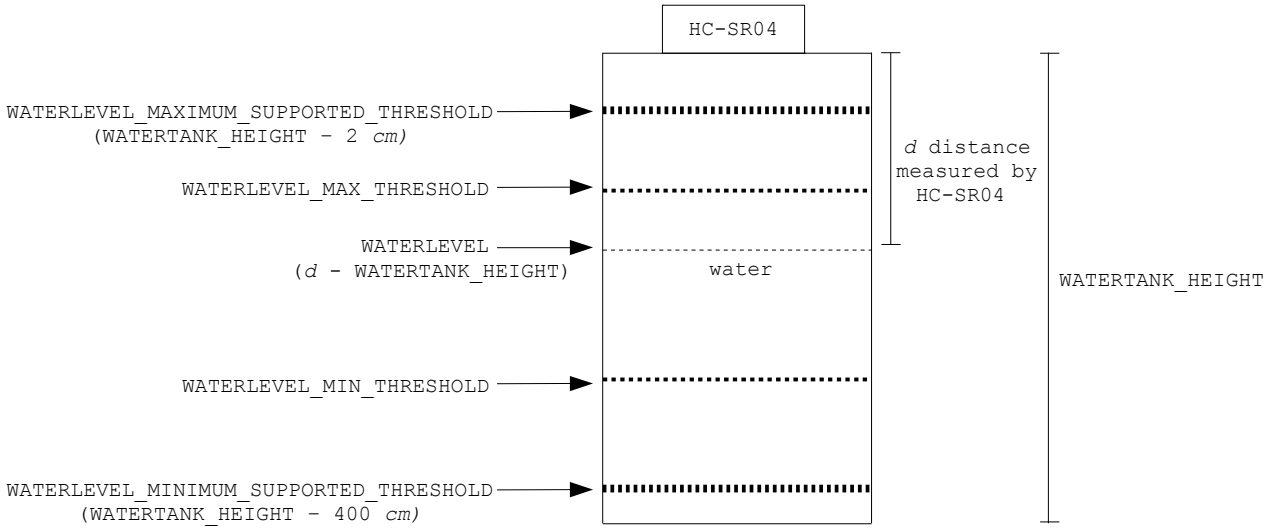

Figure 25. Water tank with its associated levels and thresholds.

In addition to the water tank variable, as it can be seen in the previous figure, other thresholds are managed such that the system will dispatch an alarm whenever these values are passed. These values are the max level of water that it is permitted in the tank `WATERLEVEL_MAX_THRESHOLD`, and the minimum level permitted `WATERLEVEL_MIN_THRESHOLD`. The max level threshold must be greater than the min level threshold.

Furthermore, both thresholds must be in a range measurable by the sensor in the following sense: Since the device HC-SR04 only works in a range between 2 cm and 400 cm, the device will not be able to deduce whenever water surface is within a distance lower than 2 cm. Analogously, it will not be able to deduce when the water surface is within a distance greater than 400 cm. Therefore, both thresholds must be situated in a range between `WATERTANK_HEIGHT - 2 cm` and `WATERTANK_HEIGHT - 400 cm`.

The configuration of the water tank height & the respective thresholds is done in the system initialization phase through the UART-based system console -see Section 5-. In that procedure those values are read by the user, and for the thresholds it is verified that are in the correct range according to the previous explanation. As it is commented in the Section 5, it is not mandatory to set a water tank height nor the thresholds. In that case the handler will work in a dummy mode in which it will be assumed that the tank has an “infinite” capacity, and it will not be retrieved the current water level but the brute distance between the sensor and the water surface.

4.3 Communicating with the sensor & water-level monitoring operation flow

Once the variables are configured through the initial calibration routine, it is called the initialization procedure `WATERLEVEL_HANDLER_INIT` of the `waterlevel_handler` module. Within this procedure are set the GPIO pins that are going to be connected at the hardware level with the sensor device. In this case it has been connected the microcontroller port B6 (D10) with the sensor Trigger port in order to send the pulse that activates the distance measuring function, and it has been connected the microcontroller port C7 (D9) with the sensor Echo port in order to receive the response signals.

To ease the development, the software handler is supported with a library which implements the signaling mechanism explained in Section 4.1. The library is available at [12], and it is based on the STM32F4 microcontroller models; as a consequence, minor changes were done in order to adapt it to the STM32L476 microcontroller. The most significant change that was done in the library was to make it work with a system own timer which is the same one used for the temperature module -See Section 3-. Through this timer, we can use a counter at the precision level of microseconds to comply the necessary delays for the sensor signaling procedure. The following code corresponds to the read function of the used library – using the own system configured timer-.

```
54  float TM_HCSR04_Read(TM_HCSR04_t* HCSR04) {
55  uint32_t time, timeout;
56  /* Trigger low */
57  HAL_GPIO_WritePin(HCSR04->TRIGGER_GPIOx, HCSR04->TRIGGER_GPIO_Pin, RESET);
58  /* Delay 2 us */
59  TIM6_MicrosecondsDelay(2);
60  /* Trigger high for 10us */
61  HAL_GPIO_WritePin(HCSR04->TRIGGER_GPIOx, HCSR04->TRIGGER_GPIO_Pin, SET);
62  /* Delay 10 us */
63  TIM6_MicrosecondsDelay(10);
64  /* Trigger low */
65  HAL_GPIO_WritePin(HCSR04->TRIGGER_GPIOx, HCSR04->TRIGGER_GPIO_Pin, RESET);
66  /* Give some time for response */
67  timeout = HCSR04_TIMEOUT;
68  while(! HCSR04_READPIN(HCSR04->ECHO_GPIOx, HCSR04->ECHO_GPIO_Pin)){
69     if (timeout-- == 0x00) {
70        return -1;
71     }
72  }
73  time = 0;
74  /* Wait till signal is low */
75  while(HCSR04_READPIN(HCSR04->ECHO_GPIOx, HCSR04->ECHO_GPIO_Pin)){
76     time++; /* Increase time */
77     TIM6_MicrosecondsDelay(1); /* Delay 1us */
78  }
79  /* Convert us to cm */
80  HCSR04->Distance =  (float)time * HCSR04_NUMBER;
81  /* Return distance */
82  return HCSR04->Distance;
83  }
```

Figure 26. Extract of the receive procedure for the HCSR04 Library.

So it can be seen how the procedure implements the required signaling algorithm contemplated in Section 4.1. After having set the trigger signal high (ll. 61 – 63) the HCSR04 device will activate and send the ultra-sonic waves towards the water surface. In the meantime, at the software level the procedure waits for a prudential time (ll. 66 – 72) before for starting to receive the sensor response through the Echo line. Whenever we start to receive the sensor response, it is calculated the echo pulse width (ll. 73 – 78). Finally, it

is done a conversion dividing the calculated echo pulse width (in milliseconds) by 58 to get the distance towards the water surface in centimeters.

The presented procedure is called within the `waterlevel_handler` module specifically on the `WATERLEVEL_HANDLER_RECEIVE` procedure, and as it used as follows: If the water tank height was not initialized then the variable `WATERLEVEL` is silly interpreted as the calculated distance. Otherwise, the `WATERLEVEL` variable is updated making the subtraction between the water tank height and the calculated distance. In addition, if the thresholds were initialized for this module, it is checked if the `WATERLEVEL` has passed either the `WATERLEVEL_MAX_THRESHOLD` or the `WATERLEVEL_MIN_THRESHOLD` bounds. For both cases, the module provides two flags: `WATERLEVEL_ALARM_MIN_THRESHOLD_REACHED` and `WATERLEVEL_ALARM_MAX_THRESHOLD_REACHED`. Each flag is set to high whenever the water level passes either the min threshold or the max threshold, according to the respective case.

Finally, the `waterlevel_handler` module provides the function `WATERLEVEL_HANDLER_Print` which is used to the put on the standard output which the current tank water level. In addition, the procedure prints a corresponding alarm if one of the two mentioned thresholds flags are set on.

5 System console

The monitoring system integrates a module embedded for managing a system terminal. The idea is to provide a command-line interface for interacting with the user connecting the microcontroller towards the computer through an UART channel which goes over the USB connection. Hence, through this interface the user can configure the system main variables (sampling period, turning on/off devices, water-tank height and the different thresholds), receive the sensor measurements and the corresponding alarms. The microcontroller program stores the necessary logic to carry out the module management whereas in the computer side is necessary to have a serial terminal client.

The console is supported having as a channel the microcontroller USART2 interface which is managed by the structure *huart2*. This UART channel is initialized in the *usart.c* project file. Hence, for printing data to the console it is called the method `HAL_UART_Transmit(&huart2, (uint8_t*) &buffer, strlen(buffer), 0xFFFF)` such that `buffer` stores the string to be printed in the console. As an alternative, it has been configured the C standard library function `printf(const char *format, …)` in such a way that whenever this function is called, the system will understand that the standard output in which to print the string will be in the UART channel. On the other hand, it was developed an scanning function for getting the system variables that are entered by the user through the system console. The next figure presents the `_scanf(uint8_t* buffp)` function which is the procedure developed for accomplishing this scanning task.

```
312 void _scanf(uint8_t* buffp){
313    memset(buffp, 0, SCANF_BUFFR);
314    uint8_t* p = buffp;
315    uint8_t c = 0x00;
316    while(1){
317       HAL_UART_Receive(&huart2, (uint8_t*) &c, 1, HAL_MAX_DELAY);
318       HAL_UART_Transmit(&huart2, (uint8_t*) &c, 1, HAL_MAX_DELAY);
319       *p++ = c == 0x0D ? 0x00 : c;
320       if(c == 0x0D){
321          break;
322       }
323    }
324    HAL_UART_Transmit(&huart2, (uint8_t*)"\r\n", 2, HAL_MAX_DELAY);
325 }
```

Figure 27. System console scanning function.

The variable `buffp` is the auxiliary variable which is used to read the complete string value inserted by the user. In ll. 316 – 322 of the presented code it is read one character (stored in the `c` variable) at a time inside the loop statement.

Each received character coming from the UART channel is immediately forward back to the UART interface printing it in the system console for letting the user see the characters he or she is typing. Each received character is sequentially stored in the `buffp` variable. Then, when the user presses the enter key -which is the carriage return character, 0x0D on the ASCII representation- and its received by the routine it will break the loop finishing the string scanning.

The system console initial view is shown just after the microcontroller does its corresponding initial configuration (system clock configuration, peripherals initialization, etc.). Internally, it is called the `DISPLAY_MENU()` routine (implemented on the *main.c* file) which prints the system main menu over the console. The options implemented in the system console are the following ones:

- `SETUP`: It provides to the user several options for initializing the system: choose the sensors to turn on/off, set the sensor devices sampling period, set the water-tank height, and configure the alarm maximum and minimum thresholds for both the temperature module and the water-level module.

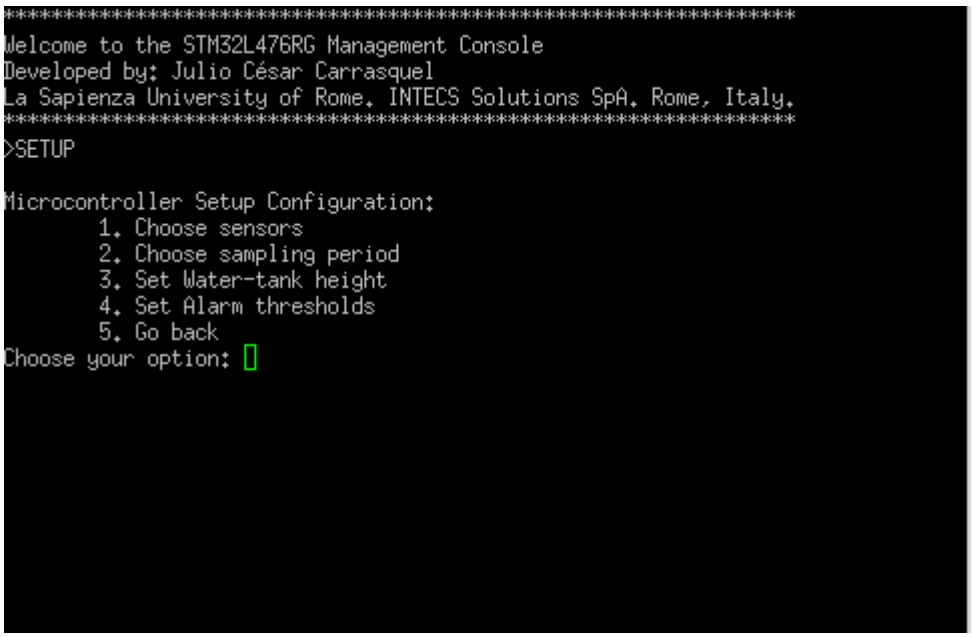

Figure 28. System console SETUP option.

- `START`: It initiates to read data from sensors according to the configuration previously done in the `SETUP` command -or with the default configuration-. The sensor measurements of the selected devices are shown in the console, and if any alarm flag is activated it is notified as well.

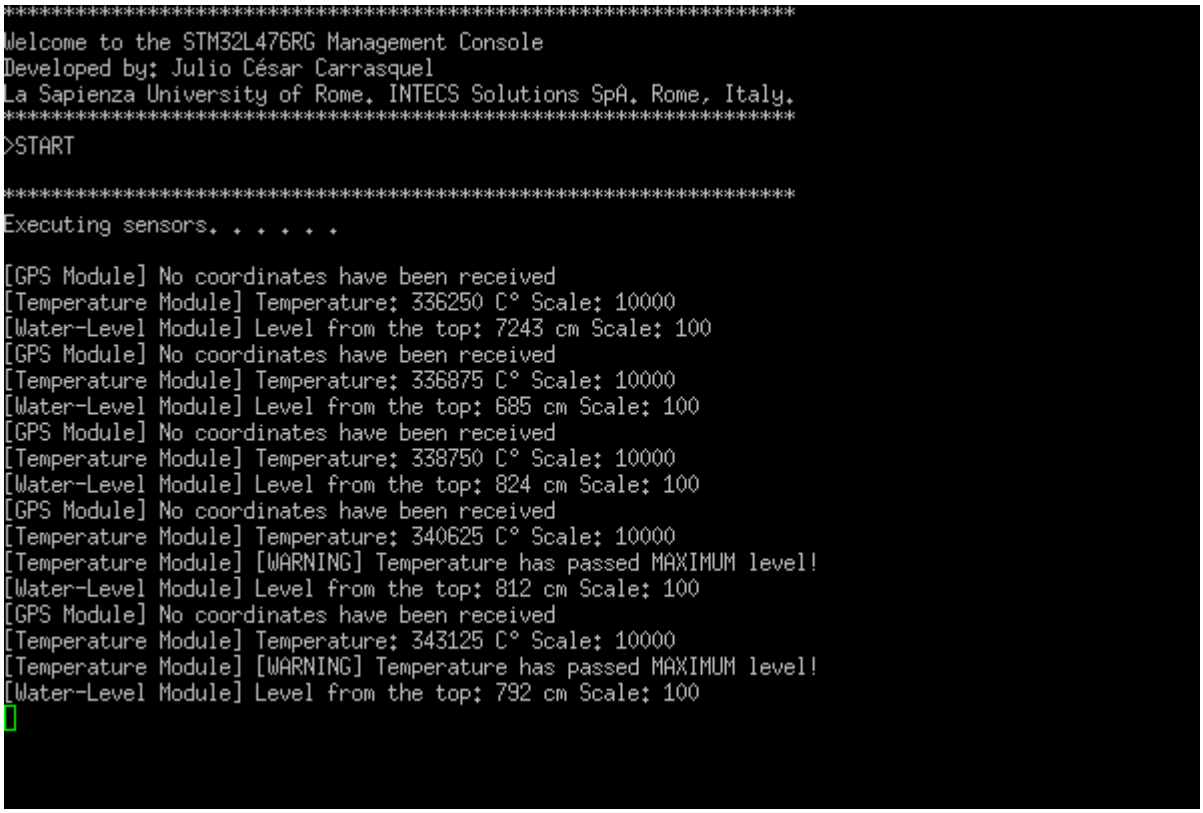

Figure 29. Starting to capture the sensor measurements through the START option.

- `CLEAN`: Clear the system console.

Having launched the `START` command the microcontroller has started to print the sensor measurements to the system console. In order to stop this execution and going back to the main menu it can be pressed the `Q` key. Internally this interruption management was done in the following way: whenever the reading the data from the sensor begins it is called the `HAL_UART_Receive_IT(&huart2, (uint8_t*) &esc, 0xFFFF)` command. Hence, whenever a key is pressed the system will be notified by means of a callback which will verify if the received character is indeed the `Q` character. If it is true then it will interrupt this sensor readings procedure, and the program control will go back to the main menu.

6 Download

The project source code is freely available for download at the following repository: https://github.com/juliocesarcarrasquel/waterlevel-monitoringsystem. It has been developed using the STM32CubeMX framework in order to configure the microcontroller features, and the System Workbench 4 IDE for the coding development phase.

7 References

[1] ST Microelectronics. STM32L476RG Microcontroller: Ultra-low-power with FPU ARM Cortex-M4 MCU 80 MHz. http://www.st.com/en/microcontrollers/stm32l476rg.html

[2] ST Microelectronics. STM32L476xx Datasheet: Ultra-low power ARM Cortex-M4 32-bit MCU. Retrieved from: http://www.st.com/resource/en/datasheet/stm32l476rg.pdf

[3] ST Microelectronics. STM32CubeMX – STM32Cube initialization code generator. http://www.st.com/en/development-tools/stm32cubemx.html

[4] OpenSTM32 Community Site. System Workbench for STM32. http://www.openstm32.org/System+Workbench+for+STM32

[5] NEO-6 U-Blox GPS modules Datasheet. Doc. Number: PS.G6-HW-09005-E. Rev. E. December 2011.

[6] NMEA Data. Retrieved from: http://www.gpsinformation.org/dale/nmea.htm. (April 2017).

[7] MINMEA: A lightweight GPS NMEA 0183 parser library in pure C. Available at: https://github.com/cloudyourcar/minmea

[8] Dallas Semiconductors Corp. Division of Maxim Integrated Products, Inc. DS18B20 Programmable Resolution 1-Wire Digital Thermometer Datasheet.

[9] STM32F4 Discovery. Library 12: 1-Wire Library for STM32F4. Available at: https://stm32f4-discovery.net/2014/05/library-12-onewire-library-for-stm43f4xx/

[10] STM32F4 Discovery. Library 13: Reading temperature with Dallas DS18B20 on STM32F4. Available at: https://stm32f4-discovery.net/2014/05/13-reading-temperature-with-dallas-ds18b20-on-stm32f429-discovery-board/

[11] ElecFreaks. Ultrasonic Ranging Module HC–SR04 Datasheet. Retrieved from: http://www.micropik.com/PDF/HCSR04.pdf (April 2017).

[12] STM32F4 Discovery. Library 30: Measure distance with HC-SR04 and STM32F4. Available at: https://stm32f4-discovery.net/2014/08/library-30-measure-distance-hc-sr04-stm32f4xx/

8 Acknowledgements

The presented solution in this report has been developed thanks to the support of INTECS Solutions SpA with headquarters at Rome, Italy. The company provided the facilities, the hardware equipment and the useful assistance throughout the development of the project. Special thanks to Ugo Maria Colesanti whose tutorship leaded to the achievement of the proposed goals within this work.