Critical Systems Lab - MESCC
Water Pumping Automated System

Ricardo Mendes 1201779

Arthur Gerbelli 1220201

ISEP, January 2024, **Second Delivery**

Contents

1 Introduction

This document is a follow-up of the previous work.

It takes into account the feedback received during the last presentation and also the current objectives of the exercise.

2 Requirement Specification

2.1 Problem Domain

2.1.1 [UPDATED] Stakeholder Needs

Although not mentioned on the assignment, we chose to add some changes to the Stakeholder Needs that are paramount to understand the next chapters.

- **SN-1.3** Every WPS will have two pumps and two water level sensors to achieve a certain level of redundancy and reliability on the system.
- **SN-1.4** To improve the system's performance, and given that we have one unused water pump, this pump should be used when the water level is above 2/3 of the well max capacity.
- **SN-1.5** In case that only one water pump is operational, the max capacity of the WPS shall be reduced.

SN-1.4 and **SN-1.5** are the result of the Stakeholder being able to describe the system's performance using the identified *Measure of Effectiveness*. The wet well capacity and the input flow can be greater if we use the second pump instead of leaving it on stand-by and only used during failure.

As described in the Stakeholder Needs, the second pump will work only if the water level is above 2/3 of the well capacity. In case that one of the pumps stops to be operational, an alarm will be triggered to alert the maintenance team, and the max. water level will be reduced.

2.1.2 [UPDATED] System Context and Use Cases

During the previous analysis of the System's Context and Use Cases, we decided to split the WPS system as a whole. The main objective is to split responsibilities and so, simplifying and clarifying the requirements. Another result, was the creation of two, very easy to grasp, systems: the RSS and the WPS.

By turning it simple, we can state that the system has obviously no bugs; on a complex one, we can only which to achieve a system with no obvious bugs.

The following changes were made in the Use Case diagrams:

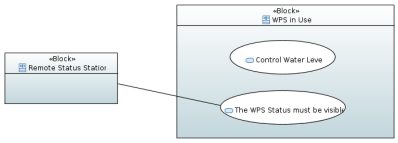

Figure 1: Use Case diagram - WPS

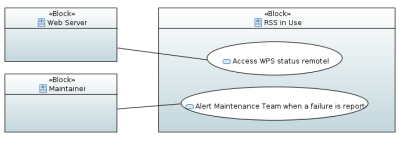

Figure 2: Use Case diagram - RSS

The activity diagram of the use case *"Control Water Level"*, also shows that WPS can be seen as an independent system regarding the RSS.

The interactions are between the sensors, the water pump as the main actuator, and the control unit.

So one of the externalities of the WPS, is to give visibility to its internal status.

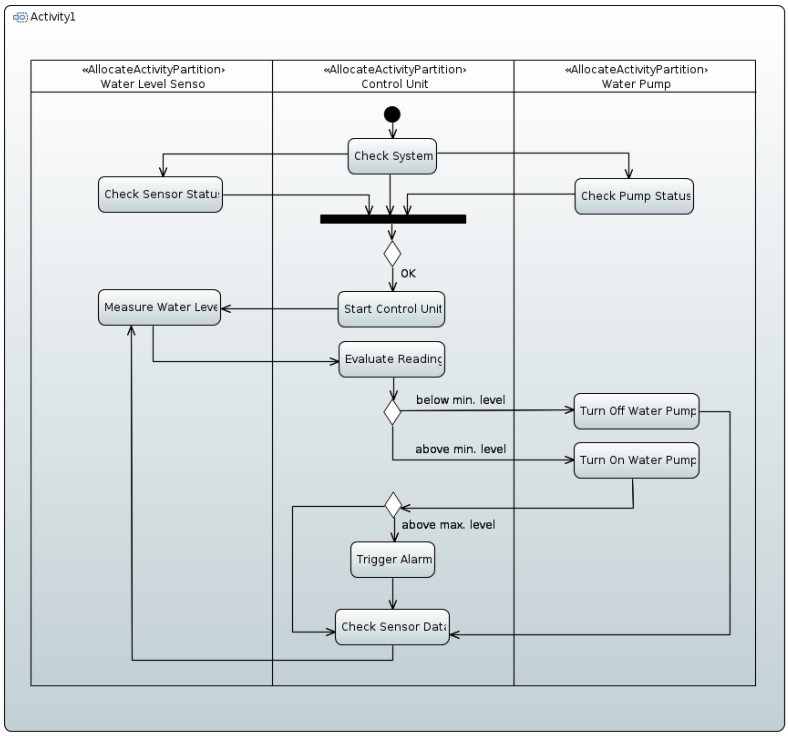

Figure 3: Control the Water Level inside the well

2.2 Solution Domain

2.2.1 Hazard Analysis

Given the critical nature of the system, we reintroduced here an updated analysis of its hazards. This list maps directly to the Stakeholder Need SN-1.3 but goes a little bit further.

H-1:
- **Description:** One of the pumps stops working.
- Cause: Mechanical problem.
- Effect: Lost of redundancy and reduction of system performance.
- **Mitigation:** Reduce the maximum water level to 2/3 and trigger alarm.

H-2:
- **Description:** Both pumps stopped working.
- Cause: Mechanical problem.
- Effect: Complete failure of the system.
- **Mitigation:** Trigger alarm.

H-3:
- **Description:** A pump doesn't turn OFF when the water level in bellow minimum.
- Cause: Mechanical problem.
- Effect: Pump overheating and complete failure.
- **Mitigation:** Trigger alarm.

H-4:
- **Description:** The two level sensors give contradictory readings, i.e. one above max and one below min.
- Cause: Sensor malfunction, connection issues.
- Effect: Inappropriate system behavior.
- **Mitigation:** If the reading of both sensor are too unequal, there must be a way to distinguish between the wrong and the correct data. There are three possible ways to deal with the issues: choose a master and a slave sensor, retain the previous input and compare it with the current one, or choose the worst case. Trigger the alarm if the system in unable to achieve a consensus.

H-5:
- **Description:** Power shortage.
- Cause: Multiple causes
- Effect: Complete failure of the system.
- **Mitigation:** RSS with independente power supply and trigger alarm.

H-6:
- **Description:** RSS are not getting information from WPS.
- Cause: Connection issues or Messagem broker stoped working.
- Effect: Unknown status of the system.
- **Mitigation:** Implement a cluter of MQTT Brokers or remove this single point of failure by adopting DDS.

H-7:
- **Description:** RSS stops working.
- Cause: Malfunction.
- Effect: Unknown WPS status.
- **Mitigation:** Have redundancy by having multiple RSS and each one displaying all statuses from all WPS.

H-8:
- **Description:** Control Unit stops working.
- Cause: Malfunction, bug.
- Effect: Total failure of the system.
- **Mitigation:** Implement redundancy by having a cluster of nodes running the Control Unit. If the number of nodes is 3 we can implement a voting system and run the same process with the same input in parallel. This would improve the system's fault tolerance.

The main output of this analysis was an updated System Requirements.

2.2.2 [UPDATED] System Requirements

SR-1 .1: While the water level is above the minimum level, WPS shall have a pump working.

.2: When the water level is below minimum level, WPS shall have all pumps stopped.

.3: If the water level is above the maximum level, then the WPS shall trigger an alarm at the Remote Status Station (RSS).

.4: A second pump shall be turned on only when the water level is above 2/3 the maximum water level.

.5: When only one pump is available, the maximum water level shall be reduced to 2/3.

.6: If the readings of the sensor are uneven to a level of 20cm, the system should choose the worst case scenario, following the table below:

sensor #2 \ sensor #1	**0**	**1**	**2**	**3**
0	0	0	0	3
1	0	1	1	3
2	0	1	2	3
3	3	3	3	3

0 = below min; **1** = above min; **2** = above med; **3** = above max.

SR-2 .1: The status of all WPS shall be displayed on all RSS.

.2: If the alarm is ON, the button in the RSS shall only disable it.

.3: The RSS shall have an independent power supply from the WPS.

.4: The alarm on the RSS shall have an independent power supply from the RSS itself and from the WPS.

SR-3 .1: The status of all WPS shall be visible on a web page.

SR-4 .1: To improve the system´s communication reliability, a cluster of 2(two) MQTT brokers shall be deployed.

2.2.3 [UPDATED] System Structure

As already stated during the System's Context analysis, we chose to split the system into two main system. This is mainly based on the principle of single responsibility.

The result can be seen below:

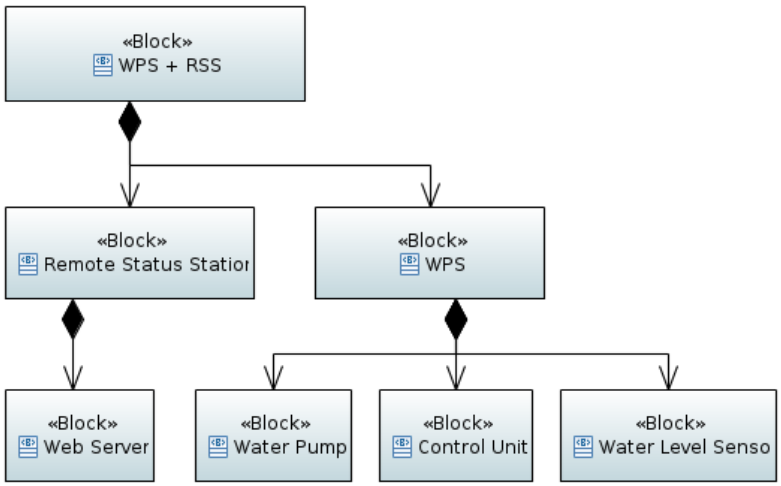

Figure 4: System Structure Diagram

2.2.4 Traceability

This chapter should clarify that the results from the Problem Domain and the analysis of the Solution Domain map to the final system implementation.

The Stakeholder Needs are being answered by the Requirements that we documented:

	SN-1.1 Pump	SN-1.2 S...	SN-1.3 Cr...	SN-1.4 Pe...	SN-1.5 F...	SN-2.1 WP...	SN-2.2 D...	SN-2.3 A...
SR-1 WPS	☐	☐	☐	☐	☐	☐	☐	☐
SR-1.1 Water Level Min	☑	☑	☐	☐	☐	☐	☐	☐
SR-1.2 Water Level Low	☑	☑	☐	☐	☐	☐	☐	☐
SR-1.3 Water Level Max	☑	☑	☐	☐	☐	☐	☐	☐
SR-1.4 Water Level 2/3	☑	☑	☐	☑	☐	☐	☐	☐
SR-1.5 Pump Redundancy	☑	☑	☑	☐	☑	☐	☐	☐
SN-1.6 Sensor Data	☐	☐	☐	☐	☐	☐	☐	☐
SR-2 Remote Status Station	☐	☐	☐	☐	☐	☐	☐	☐
SR-2.1 Display WPS Status	☐	☐	☐	☐	☐	☑	☑	☑
SR-2.2 Disable Alarm	☐	☐	☐	☐	☐	☑	☐	☐
SR-2.3 RSS Power Supply	☐	☐	☑	☐	☐	☐	☐	☐
SR-2.4 Alarm Power Supply	☐	☐	☑	☐	☐	☐	☐	☐
SR-3 Web Server	☐	☐	☐	☐	☐	☐	☑	☐
SR-4 MQTT Communication	☐	☐	☑	☐	☐	☐	☐	☐

Figure 5: Traceability to Stakeholder Needs

The diagram, developed during the analysis of the System Structure, is being used as the main foundation of the deployment diagram (please see the diagram on the *Implementation* chapter.

Most important, the Hazard analysis gave raise to multiple new requirements, that although not obvious in the Stakeholder Needs, are here handled critical for the description of the system.

3 Implementation

3.1 Concorrency and Real Time Scheduling

3.1.1 Introduction

To give a realistic analysis of the processes that we are proposing to run, we first need to describe the hardware where they will be running.

The microcontroller ESP32 has FreeRTOS integrated into it as a component; however, in our case, there is a slight difference:

The original FreeRTOS (...) is a compact and efficient real-time operating system supported on numerous single-core MCUs and SoCs. However, to support dual-core ESP targets, such as ESP32, ESP32-S3, and ESP32-P4, ESP-IDF provides a unique implementation of FreeRTOS with dual-core symmetric multiprocessing (SMP) capabilities (hereinafter referred to as IDF FreeRTOS).[1]

So we are dealing with a **fixed priority preemptive scheduler with time slicing**, meaning:

- *Each task is given a constant priority upon creation.*
- *The scheduler can switch execution to another task without the cooperation of the currently running task.*
- *The scheduler periodically switches execution between ready-state tasks of the same priority in a round-robin fashion.* [1]

From now on, we will focus our attention on the **Control Unit**, undoubtedly the most critical subsystem.

This subsystem is responsible for coordinating the water pumps, depending on the input from the water level sensors and the current state of those pumps. Additionally, it must publish the current state of the whole WPS system to a MQTT broker. Also, for this purpose, the **Control Unit** will be responsible for asking for the water level status given by the sensor. As already mentioned, the idea is to query the sensor at regular intervals. A timeout or an agreed-upon response can signal to our system that something is wrong with the sensor.

In conclusion, the **Control Unit** will run three tasks:

- T1: Retrieve data from sensors (TCP Client);
- T2: Process data and give instructions to pumps;
- T3: Publish WPS status to MQTT broker (MQTT client).

One last note: we cannot ensure a 100% real time system given the multiple libraries from Arduino that we are using, especially for communication purpose.

3.1.2 Concorrency

T1, T2 and T3 tasks share the same resource: a data structure with the WPS status, pump status, water level status and alert status.

When T2 is updating this data, T3 should wait until this processing is finished. Only afterwards does T3 have permission to read the data and publish it. The same happens with the relation between T1 and T2.

This will be accomplished using a mutex. Currently, this feature is not yet implemented but will be used in the final presentation.

3.1.3 Real Time Scheduling

Given the two last chapters, we can, with some degree of confidence, make a realistic predition of our system's tasks scheduling.

As described by the documentation [1], the algorithms are preemptive and priority-driven, meaning that a task with a priority higher than the one currently in the CPU will interrupt it. The priority will be defined by the task period - hence simulating a **Rate Monotonic** (RM) scheduler. In RM the priority is inversely proportional to the period: the shorter the period, the higher the priority.

Using the hardware specification for an ESP-WROOM-32:

- Clock rate: 80 to 240 MHz
- Clock cycle per instruction (CPI): 2 clock cycles — for instructions following ALU and branch instructions. 4 clock cycles — in other cases.

Ruf estimation for **processing data** by disassembling the compilation output and using the worst hardware specifications:

```
$ avr-objdump -m avr -S control-unit.ino.cpp.o | wc -l
```

output –> 448 [1].

$$CPU\ Time = \frac{Instruction\ count * CPI}{Clock\ Rate} \tag{1}$$

$$CPU\ Time = \frac{448 * 4}{80000} = 0,0224\ seconds \tag{2}$$

We reached the following numbers in milliseconds, being Ci the worst-case execution time (WCET) and Ti the periodicity:

Task	Ci	Ti
1	60	100
2	25	200
3	35	250

[1] Includes instructions added by the compiler for debugging purpose

The blocking periods given by shared resources are not being taken into account.

As proved in *Liu and Layland* article[2] , the schedulability of a set of tasks can be determined by its CPU utilization.

The schedulability can be confirmed by calculating the Least Upper Bound (LUB) and comparing it with the utilization factor (U).

$$U(n) = \Sigma_{i=1}^{n} \frac{Ci}{Ti} \leq n(2^{\frac{1}{n}} - 1) \tag{3}$$

where C is the WCET and T the period.

The formula mentioned above represents the comparison made by the sum of each task's utilization time and the LUB for a n number of tasks.

It is worth noting that if the U surpasses the value of 1 - the CPU utilization exceeds 100% - the set is not schedulable.

Yet, in the case of an intermediate value of U,

$$1 \geq U(n) \geq LUB \tag{4}$$

the schedulability is undetermined.

In those cases, we resort to an iterative technique that will diverge or converge from a worst-case response time in a finite number of steps:

$$Rwci(0) = \Sigma_{k \in hp(i)} Ck + Ci \tag{5}$$

$$Rwci(m+1) = \Sigma_{k \in hp(i)} \left\lceil \frac{Rwci(m)}{Tk} \right\rceil * Ck + Ci \tag{6}$$

where *Rwc* is the worst case response time and *hp* is the highest priority task.

The equation 6 shows us some of the characteristics of the *critical instant* when dealing with preemptive and fixed priorities. *Synchronous release*, where all tasks are launched at the exact same moment, will mean a worst response time for the tasks, specially for the task with the least priority, usually more prone to miss their deadline.

For large sets, the LUB to processor utilization factor in RM is around 70%. This means that in the worst case, the CPU will be idle 30% of the time.

The article also points out that there are strong suggestions that make U=1 not feasible.[2].

For our task set, we will first calculate the utilization factor:

$$U = \frac{60}{100} + \frac{25}{200} + \frac{35}{250} + \frac{2}{7} = 0,865$$

$$LUB = 3(2^{\frac{1}{3}} - 1) \approx 0,78$$

$$1 \geq 0,865 \geq 0,78$$

$$CPU\ utilization\ doesn't\ exceed\ 100\%\ but\ cannot\ confirm\ that\ it\ is\ schedulable! \tag{7}$$

[2]In the real world, there is always some bandwith of the CPUs time that is used by processes, i.e. OS daemons

For T3

$$\begin{aligned} Rwc3(0) &= 60 + 25 + 35 = 145 \\ Rwc3(1) &= \left\lceil \frac{145}{100} \right\rceil * 60 + \left\lceil \frac{145}{200} \right\rceil * 25 + 35 = 120 \\ Rwc3(2) &= \left\lceil \frac{120}{100} \right\rceil * 60 + \left\lceil \frac{120}{200} \right\rceil * 25 + 50 = 120 \leq 250 \end{aligned} \tag{8}$$

For T2

$$\begin{aligned} Rwc2(0) &= 60 + 25 = 85 \\ Rwc2(1) &= \left\lceil \frac{85}{100} \right\rceil * 60 + 25 = 85 \leq 200 \end{aligned} \tag{9}$$

The conclusion is that the **task are schedulable**.

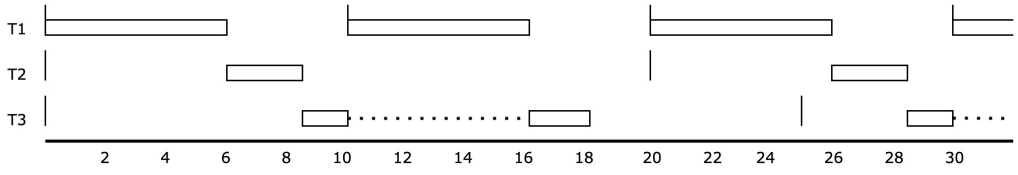

Figure 6: Gaant Chart

3.2 Communication Infrastructure

The following text and diagram describes the prototype that we propose to implement.

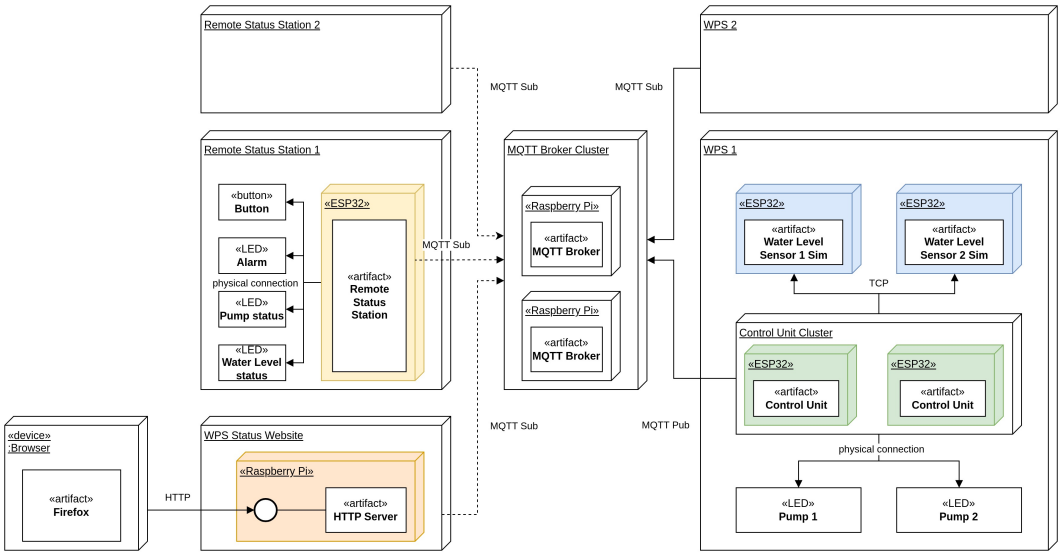

Figure 7: Deployment diagram

Communication Software Infrastructure:

- The Web Server will be implemented using the Apache HTTP Server and will run on a Raspberry Pi 4.
- The MQTT Brokers will be implemented using Mosquitto MQTT and will run on a Raspberry Pi 4.
- The Sensors Sim and Control Unit will run on Arduino ESP32 and will be implemented using 'C' with the help of some Arduino Libraries.

Interactions between elements:

The main communication protocol is MQTT. The only publisher is the Control Unit in the WPS. The Web Server and the RSS are the subscribers.

The communication made between Control Unit and Water Sensor will be TCP/IP through wireless. Given that it is a critical system, the Control Unit will query the sensor in regular intervals and asking for the data.

A time-out or incorrect data sent by the sensor can trigger an alert to the RSS. Essentially, it is a pull strategy from the point of view of the Control Unit.

The Web Client will interact with the HTTP Server using the HTTP protocol.

All other connection shown in the diagram area physical connection inside a breadboard.

During the demonstration, all those systems will be connected wirelessly to a hotspot and share the same network.

3.3 Assembly

The piece of functionality that we chose to write in assembly is located in the sensor simulator.

The sensor simulator has four push buttons. Each button triggers a different change in the water level. The current water level can be seen by the LED that is turned on.

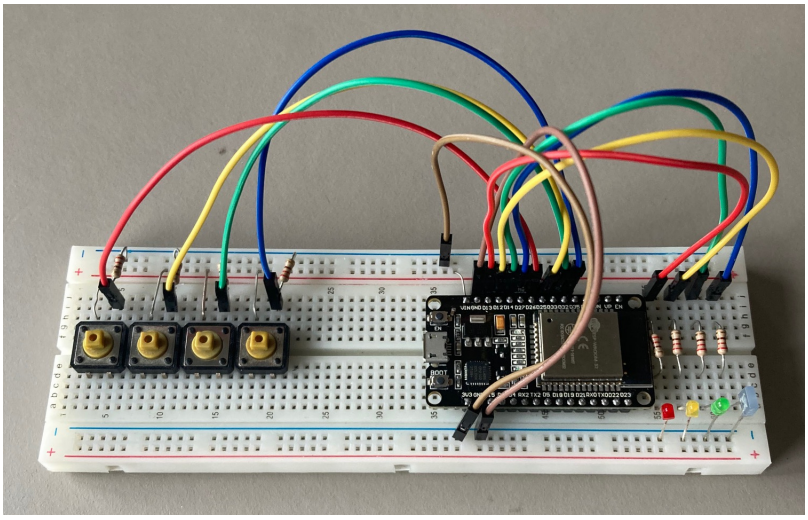

Figure 8: Sensor Sim prototype

Because only one water level state is possible, we chose to use assembly to create a mask to be implemented on the LEDs.

```
.global getMask

getMask:    entry a1, 48
            movi a5, 0          # init increment
            mov a6, a2          # save inputed water level
            movi a2, 1          # init return value

loop:
            addi a5, a5, 1      # increment by one
            beq a5, a6, end     # branch to 'end' if not equal
            slli a2, a2, 1      # left shif by 1, i.e. 0001 -> 0010
            j loop              # jump to loop

end:
            retw.n
```

For example, if we push BUTTON_MIN (the green one), this will trigger the code above and output the following number **2** that corresponds to **0000 0010** in a 1 byte binary.

The code below then implements the mask:

```
void implementMask(int8_t mask)
{
  int8_t n_bit = 0;
  while (n_bit < MAX_LEVELS) {
    if (mask & 0x01) {
      digitalWrite(STATES[n_bit], HIGH);
    }
    else {
      digitalWrite(STATES[n_bit], LOW);
    }
    n_bit++;
    mask = mask >> 1;
  }
}
```

References

[1] Espressif documentation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos_idf.html#id23/

[2] C.L.Liu, J.Layland: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the Association of Computing Machinery, Vol.20, Nº.1, (1973)

[3] ESP-WROOM-32 Specs: https://www.mouser.com/datasheet/2/891/esp-wroom-32_datasheet_en-1223836.pdf