Critical Systems Lab - MESCC
Water Pumping Automated System

Ricardo Mendes   Arthur Gerbelli

1201779   1220201

ISEP, January 2024, **Third Delivery**

Contents

1 Introduction

As this is the last report, we will try to summarize all the previous analyses, with the main focus being to achieve a conclusion and not only repeat the specificities of the last reports.

Although the following chapters are tightly defined, the report should be read as a whole.

2 Requirement Specifications

2.1 System Requirements

There was no change in the system requirements. Below, we listed all the identified system requirements and a comment about its implementation on the prototype.

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

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

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

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

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

.6: If the readings of the sensor are uneven, the system shall choose the worst case scenario, following the table below: `Implemented`

		sensor #1				
		0	**1**	**2**	**3**	**4**
	0	-	1	2	3	4
	1	1	1	1	1	4
sensor #2	**2**	2	1	2	2	4
	3	3	1	2	3	4
	4	4	4	4	4	4

1: below min; **2**: above min; **3**: above med; **4**: above max.

0: no connection to the sensor - if both sensors are unavailable, the alarm shall be triggered.

SR-2 .1: The status of all WPS shall be displayed on all RSS. `Partially Implemented:` the hadware available was not enough to add a second WPS and RSS.

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

.3: The RSS shall have an independent power supply from the WPS. `Not Implemented`

.4: The alarm on the RSS shall have an independent power supply from the RSS itself and from the WPS. `Not Implemented`

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

SR-4 .1: To improve the system´s communication reliability, a cluster of 2(two) MQTT brokers shall be deployed. `Not Implemented`

2.2 Hazard Analysis

We added one more entry to the hazard analysis, H-9. Below, we added some comments about the implementation of the identified mitigations in the prototype.

H-1: see SR-1.5

H-2:
- **Description:** Both pumps stopped working.
- **Mitigation:** Trigger alarm. `Implemented`

H-3:
- **Description:** A pump doesn't turn OFF when the water level in bellow minimum.
- **Mitigation:** Trigger alarm. `Not Implemented`

H-4: see SR-1.6

H-5: see SR-2.3 and SR-2.4

H-6:
- **Description:** RSS are not getting information from WPS.
- **Mitigation:** Implement a cluter of MQTT Brokers or remove this single point of failure by adopting DDS. Trigger alarm. `Partially Implemented: the alarm is triggered`

H-7:
- **Description:** RSS stops working.
- **Mitigation:** Have redundancy by having multiple RSS and each one displaying all statuses from all WPS. `Not Implemented`

H-8:
- **Description:** Control Unit stops working.
- **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. `Not Implemented`

H-9:
- **Description:** Rapid wear of the first water pump.
- **Mitigation:** Distribute evenly the work done by the pumps. This can be done, by choosing a random pump when one is need first. `Not Implemented`

2.3 System Structure and Traceability

This last chapter tries to map the implementation of the prototype to the model developed during the previous two deliveries.

During the implementation of the system, we noticed that the Web Server is a subsystem independent of the RSS.

Although, both are ways to visualize the WPS status, they have very different objectives and levels of criticality. The RSS must alert the maintenance team; the web

version is, in our understanding, only a nice thing to have. And most of all, we can remove one service without affecting the other. Because of this, we made some changes to the System Structure diagram, as it is possible to visualize below.
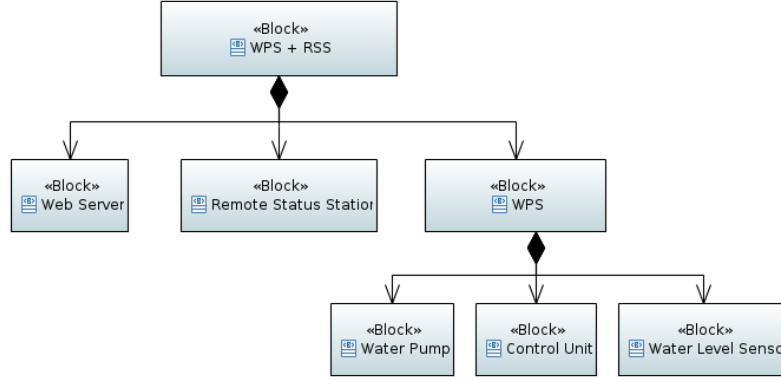

Figure 1: System Structure diagram

Looking at the prototype's deployment diagram 2, we can clearly see that, the output of the modeling, traces nicely to its implementation. The grayed parts of the diagram don't make up the prototype.
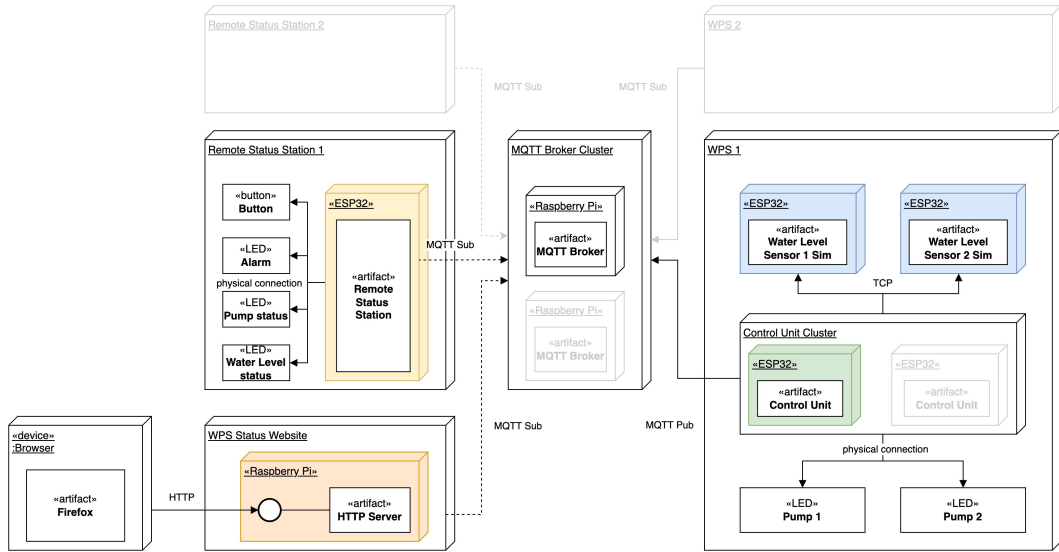

Figure 2: Prototype's deployment diagram

3   Implementation

3.1   CCSYA - Assembly

The following code snippet in implemented on the Sensor Sim, as it was in the previous
delivery. The only change was done on one of the comments that had a typo.

```
1   .global getMask
2
3   getMask:    entry a1, 48
4               movi a5, 0          # init increment
5               mov a6, a2          # save inputed water level
6               movi a2, 1          # init return value
7
8   loop:
9               addi a5, a5, 1      # increment by one
10              beq a5, a6, end     # branch to 'end' if equal
11              slli a2, a2, 1      # left shif by 1, i.e. 0001 -> 0010
12              j loop              # jump to loop
13
14  end:
15              retw.n
```

As an input, the code receives the position of the LED that needs to turn ON. So
to ensure that only one of the LEDs is turned ON, we start with the value 1 (see line
6). This value in a byte is represented by '0000 0001'. From now on, we only shift
the leftmost bit, one position at a time, until we reach the selected position.

The output is an 8-bit integer. If we have as output the decimal number '16', and
convert it to binary ´0000 1000´, we can see that we want to turn ON the LED in the
4th position. The following code implements this logic.

```
// more code

  mask = getMask(level); // ASM code
  implementMask(mask);

// more code

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;
  }
}
```

3.2 RTAES - Concorrency and Real Time Scheduling

3.2.1 Implementation of the Tasks and Concorrency control

As a remainder, the following tasks are being run in a ESP-WROOM-32 on a single core:

- **Task 1**: Retrieve data from sensors (TCP Client);

- **Task 2**: Process data and give instructions to pumps;

- **Task 3**: Publish WPS status to MQTT broker (MQTT client).

The creation of the tasks was achieved by using the **xTaskCreatePinnedToCore(...)** [1] function from the FreeRTOS API. The function **xTaskCreate(...)** was also available, but we wanted to ensure that the proposed tasks would run on the same core.

```
xTaskCreatePinnedToCore(
                requestSensorData,   /* Task function. */
                "Task1",      /* name of task. */
                10000,        /* Stack size of task */
                NULL,         /* parameter of the task */
                1,            /* priority of the task */
                &Task1,       /* Task handle to keep track of created task */
                0);           /* pin task to core 0 */
```

The access control to the critical section was done thanks to the use of a single semaphore: `xSemaphoreHandle xMutex;`.

3.2.2 Real Time Scheduling

It must be openly said that this part was the trickiest one.

To check in pratical terms, the results of the schedulability calculation done in the previous report, some changes needed to be added to the code.

Although the execution times for Task 2 look to be very close to the ones achieved in the calculations, the same doesn't apply for Task 1 and Task 3. This is partly the fault of communication latency. Both tasks need to request or send information using a wireless connection and the TCP/IP protocol.

Another problem was the method implemented to track the task's execution time. The granularity wasn't enough, especially for Task 2.

As a practical solution, we chose to update the values from the last report in a proportionally way.

Table 1: Theoretical values (left) and implemented values (right) in ms:

Task	Ci	Ti		Task	Ci	Ti
1	60	100		1	1200	2000
2	25	200		2	500	4000
3	35	250		3	700	5000

The values on the right side were achieved by running the code in different situations. A closer look, shows that we incresed all values **x 20**.

To track the execution time we added the following code to each task:

```
for (;;) {
    unsigned long start_time = millis();
    unsigned long finish_time;
    unsigned long duration;

    // Code -------------------

    finish_time = millis();
    duration = finish_time - start_time;
    Serial.print("TASK N - Execution Time[ms]: ");
    Serial.println(duration);
}
```

And here are some of the outputs:

```
TASK 3 - Duration[ms]: 40
TASK 1 - Duration[ms]: 917
TASK 2 - Duration[ms]: 475
TASK 1 - Duration[ms]: 658
TASK 1 - Duration[ms]: 754
TASK 3 - Duration[ms]: 770  < above WCET
TASK 2 - Duration[ms]: 476
TASK 1 - Duration[ms]: 649
TASK 1 - Duration[ms]: 799
TASK 3 - Duration[ms]: 29
TASK 2 - Duration[ms]: 477
TASK 1 - Duration[ms]: 176
TASK 1 - Duration[ms]: 997
TASK 2 - Duration[ms]: 481
TASK 1 - Duration[ms]: 187
TASK 3 - Duration[ms]: 157
TASK 1 - Duration[ms]: 990
TASK 2 - Duration[ms]: 475
TASK 1 - Duration[ms]: 1036
```

It is important to state, that given the short execution time of Task 2, we needed to add a sleep function to increase it and so bring it closer to the values on the right side table 1.

Also, to control that the tasks didn't extended beyond its deadline, we added the following code:

```
long next_release = 5000 - duration; // 5 seconds deadline for Task 3
if (next_release > 0) {
    vTaskDelay(next_release / portTICK_PERIOD_MS);
} else {
    Serial.println("TRASK 3 - FAILED Deadline !!!");
}
```

In some cases, Task 1 failed the deadline by a couple of milliseconds. This was because of the defined timeout value of 1 second for each TCP request (one request per sensor). When the two requests reached a timeout, we already had a 2-second execution time. Unfortunately, the library used for the TCP connection has a lower limit of 1 second for the timeout.

We consciously chose not to fix this issue, mostly because a fix would mean a more extended increase of the values, beyond the x20, and when we have at least one sensor responding, the execution time are very close to the ones in the table 1.

```
TASK 1 - Connection failed: 172.20.10.13
TASK 1 - Connection failed: 172.20.10.5
TASK 1 - Duration[ms]: 2009
TASK 1 - FAILED Deadline !!!
```

3.2.3 Concorrency

In terms of concurrency and, as proposed in the last report, we identified as a critical section the read and write of a data structure that encapsulates the WPS status.

```
typedef struct{
  volatile bool alert;
  volatile uint8_t id;
  volatile uint8_t curr_water_level;
  volatile uint8_t curr_pump1_status;
  volatile uint8_t curr_pump2_status;
  volatile uint8_t curr_pump1_state;
  volatile uint8_t curr_pump2_state;
} WPS;
```

The changes to the data structure WPS were made by accessing functions instead of accessing the values directly.

```
void set_curr_pump2_status(uint8_t value) {
  xSemaphoreTake(xMutex, portMAX_DELAY);
  _wps.curr_pump2_status = value;
  xSemaphoreGive(xMutex);
}
```

The status reads, were specially controlled during the publication of the WPS status to the MQTT broker.

```
String getWpsStatus(){
    xSemaphoreTake(xMutex, portMAX_DELAY);
    String message = String(_wps.alert) + "," + String(_wps.id) + "," + String(_wps.curr
    xSemaphoreGive(xMutex);
    return message;
}
```

3.3 COMCS - Communication

- TCP pull strategy
    - Arthur

3.4 Prototype
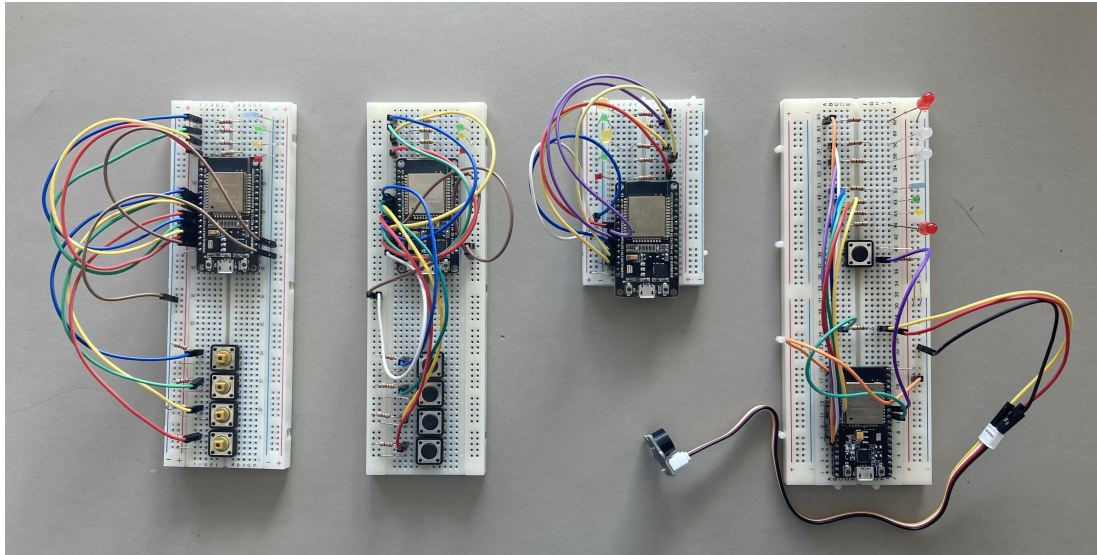
3.4.1 Overview

Figure 3: from left to right: Sensor 1, Sensor 2, Control Unit + Pumps, RSS
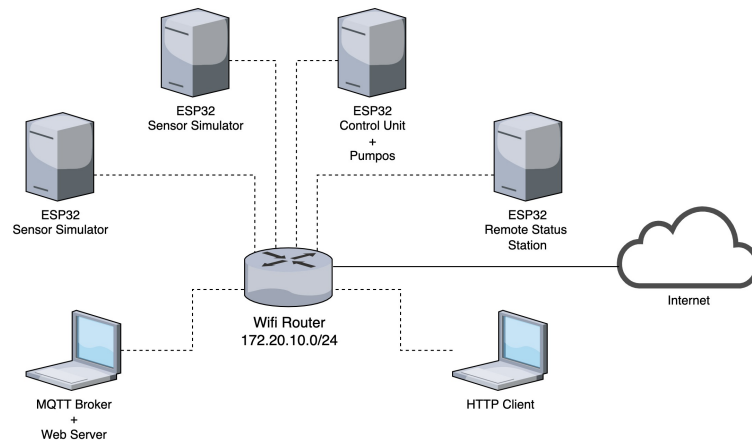
- diagram if implementation

Figure 4: Network diagram

3.4.2   WPS

- explicar a tabela dos inputs dos sensores e levantar alarm caso (A tabela agora é diferente da ultima vez) - strategy for cluster of control unit nodes

    - dois timeouts dos sensores equivale a max level.... nao devia estar assim

3.4.3   MQTT Broker

publish every 5 sec

    subscriber checks every 15sec if message - one topic per WPS

3.4.4   RSS

- reads broker every 1 sec - sends alert after 15 sec without message - subscribes to each WPS topics (iter) - log who clicked the button and maintain LED on... because the system heals it self

3.4.5   Web Server

4   Team Work

References

[1] Espressif FreeRTOS API: `https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-reference/system/freertos.html`