HazMapper: Robô de Mapeamento de Áreas Perigosas

Projeto Integrador 2, 3º ano, Licenciatura em Engenharia Eletrónica e Computadores
Universidade do Minho, Escola de Engenharia

Autores
Álvaro Silva, A104948
Ana Cruz, A104950
André Martins, A104944
Mariana Martins, A104961

Orientador
Prof. Adriano Tavares

***Abstract*—A crescente complexidade e os riscos inerentes a diversos ambientes industriais e de desastre sublinham a necessidade premente de soluções autónomas e seguras para a monitorização e mapeamento de áreas perigosas. A exposição humana a atmosferas tóxicas, temperaturas extremas ou áreas de difícil acesso representa um desafio significativo para a segurança ocupacional e a resposta a emergências. Neste contexto, a robótica móvel surge como uma tecnologia habilitadora, oferecendo a capacidade de coletar dados críticos sem colocar operadores em risco direto.**

Este relatório apresenta o HazMapper, um robô autónomo concebido para a avaliação de ambientes de risco. O HazMapper integra um conjunto de sensores para medir parâmetros ambientais cruciais, como a concentração de dióxido de carbono (CO_2) e a temperatura, transmitindo os dados em tempo real para uma estação de controlo remota. Para garantir a navegação segura, o robô dispõe também de sensores de deteção de obstáculos, que impedem o avanço em direções bloqueadas, permitindo a movimentação apenas nas áreas desimpedidas. No caso de não ser possível visualizar o ambiente diretamente, a presença de uma câmara *onboard* possibilita o acompanhamento visual remoto do que está a acontecer. A arquitetura do sistema baseia-se num microcontrolador `STM32H755ZIQ` para processamento central e controlo de movimento, com locomoção omnidirecional garantida por quatro rodas e motores independentes. A comunicação sem fios é assegurada por um módulo `ESP`, que estabelece uma ligação Bluetooth Low Energy (BLE) com uma aplicação Android desenvolvida especificamente para controlo intuitivo via joystick virtual. Adicionalmente, o robô incorpora um sistema de medição contínua da tensão da bateria, essencial para a gestão da autonomia e fiabilidade operacional. Foi implementado um RTOS no sistema de modo a o tornar mais eficiente e rapidamente responsivo.

AGRADECIMENTOS

Agradecemos ao André Costa e à professora Sofia Paiva por todo o apoio e orientação ao longo da realização do projeto. Agradecemos também ao professor Vítor Silva pela confiança no nosso potencial. Por fim — e não menos importante — agradecemos ao professor Adriano Tavares pela confiança depositada e por ter aceite o nosso pedido para ser nosso orientador.

I. ANÁLISE

A. Diagrama de Use-Case

Anexo X-A Figura 20

B. Diagrama de Blocos

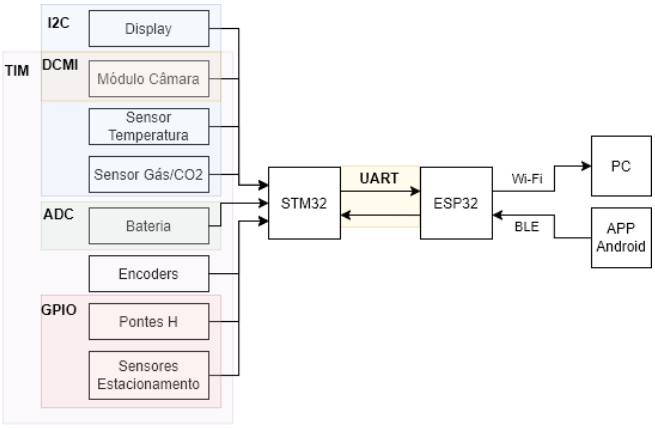

Fig. 1. Diagrama de Blocos Do Sistema

II. ARQUITETURA DE CONTROLO

A. Sistema de Movimentação

O HazMapper está equipado com rodas omnidirecionais. Estas possuem a capacidade de se movimentar livremente em qualquer direção no plano horizontal, sem a necessidade de mudar a orientação do movimento. Este tipo de rodas oferece uma grande flexibilidade e agilidade, tornando-o muito útil em aplicações que requerem movimentos precisos e rápidos.

No nosso sistema, o movimento do robô é assegurado por quatro motores, cada um acoplado a uma roda omnidirecional. O controlo individual de cada motor permite combinar os seus movimentos para produzir deslocamentos complexos, incluindo avanços, recuos e deslocamentos diagonais.

Cada motor é gerido de forma independente por uma estrutura (`motors_id`) que agrega um conjunto de variáveis

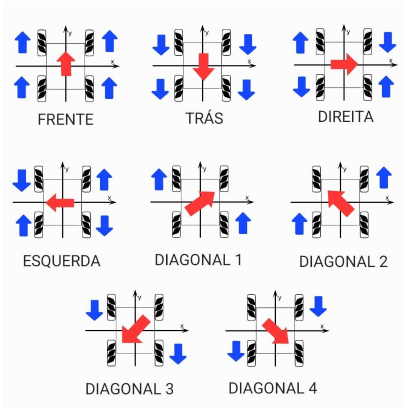

Fig. 2. Direções possíveis do robô

e estruturas necessários para o seu controlo. As principais características são:

- **Estado:** Indica se o motor está parado (OFF), a andar para a frente (FORWARD) ou para trás (REVERSE).
- **Pinos de Direção:** Dois pinos digitais (`FORWARD_DRIVE` e `REVERSE_DRIVE`) controlam o sentido de rotação.
- **PID Individual:** Cada motor possui os seus próprios parâmetros PID (`Kp`, `Ki`, `Kd`), permitindo afinação independente.
- **Variáveis de Execução:** Incluem posição e velocidade (em rad/s e rpm), valor de PWM (modulação por largura de pulso) aplicado, e variáveis internas do controlo PID (erro anterior, integral, etc.).
- **Timer de PWM:** Um `TIM` dedicado por motor é utilizado para gerar o sinal PWM que regula a velocidade, que é aplicado ao pino ENABLE da ponte-H.
- **Timer para Encoder:** Cada motor utiliza um `TIM` em modo encoder (canais A e B) para leitura da posição e velocidade.

Esta estrutura modular permite o controlo preciso e independente de múltiplos motores, facilitando a escalabilidade e manutenção do sistema.

A direção de cada motor pode ser alterada ativando as entradas apropriadas do controlo da ponte H associada, como podemos ver nos Anexos X-B, permitindo movimento para a frente ou para trás, ou a paragem total. O verdadeiro desafio no controlo de um robô com rodas omnidirecionais reside na coordenação dos quatro motores para alcançar o movimento global pretendido. Assim, o movimento do robô é gerido com base numa LUT (Look-Up Table), onde estão definidos todos os movimentos possíveis (Figura 2) e a combinação correspondente de estados dos quatro motores, informação que foi retirada a partir da Figura 3. Esta LUT é consultada sempre que é enviado um comando de movimento, permitindo uma transição rápida e organizada entre diferentes direções

Quando um determinado comando é ativado, o sistema atualiza os estados individuais dos motores de acordo com a entrada correspondente na LUT e posteriormente o PWM a

```
commands cmd_list [] = {
//      COMMAND     MOTOR 1     MOTOR 2     MOTOR 3     MOTOR 4
        {FW,        FORWARD,    FORWARD,    FORWARD,    FORWARD },
        {FW_RIGHT,  OFF,        FORWARD,    FORWARD,    OFF     },
        {RIGHT,     REVERSE,    FORWARD,    FORWARD,    REVERSE },
        {BW_RIGHT,  REVERSE,    OFF,        OFF,        REVERSE },
        {BW,        REVERSE,    REVERSE,    REVERSE,    REVERSE },
        {BW_LEFT,   OFF,        REVERSE,    REVERSE,    OFF     },
        {LEFT,      FORWARD,    REVERSE,    REVERSE,    FORWARD },
        {FW_LEFT,   FORWARD,    OFF,        OFF,        FORWARD },
        {STOP,      OFF,        OFF,        OFF,        OFF}
};
```

Fig. 3. LUT dos movimentos possíveis

eles aplicado. Estes estados são traduzidos em sinais digitais que controlam os pinos de ativação das pontes H responsáveis por alimentar os motores.

O sistema é ainda capaz de ligar e desligar o robô através de um pino digital dedicado, garantindo que, quando desligado, todos os motores ficam inativos por segurança.

1) Algoritmo de Controlo de Velocidade – Controlo PID: Para além do controlo da direção, é também essencial que todas as rodas girem à mesma velocidade, a fim de evitar desvios na trajetória do robô e assegurar uma locomoção estável e controlada.Para garantir o controlo preciso da velocidade dos motores do robô, foi implementado um controlador PID (Proporcional-Integral-Derivativo) em malha fechada. Este tipo de controlo baseia-se na medição contínua da velocidade atual de cada motor e na comparação com um valor de referência, designado por *setpoint*. O objetivo do controlador é minimizar o erro entre a velocidade medida e a desejada, ajustando dinamicamente o sinal de controlo aplicado a cada motor.

A velocidade atual de cada motor é determinada utilizando timers individuais dedicados a cada motor, configurados em modo encoder com suporte a codificação em quadratura. Cada motor possui um encoder incremental que gera dois sinais digitais, desfasados de 90°, permitindo contar pulsos e identificar a direção de rotação. Estes sinais são ligados às entradas de dois canais do timer. Esta funcionalidade permite que o contador interno do timer incremente ou decremente automaticamente em função da do sentido de rotação e do número de transições dos sinais, sem necessidade de intervenção do processador, atualizando o valor do contador (`TIMx_CNT`) conforme o movimento do eixo, o que é crucial para o controlo bidirecional dos motores.

A variação de posição entre duas amostragens sucessivas é calculada subtraindo o valor anterior da posição ao valor atual.

A posição atual do eixo do motor é obtida através da leitura do valor do contador do timer, utilizando a macro `__HAL_TIM_GET_COUNTER()`, e corresponde à posição absoluta acumulada desde o início do movimento. Com base na posição acumulada, calcula-se a posição angular em graus, utilizando a seguinte expressão:

$$\theta_{\text{graus}} = \frac{\text{posição} \times 360}{\text{PULSOS} \times 4 \times \text{SAMPLING TIME}} \qquad (1)$$

A velocidade angular é calculada a partir da variação

do número de pulsos entre duas medições consecutivas. Assume-se um período de amostragem constante, e os fatores numéricos utilizados nas expressões abaixo incorporam esse intervalo de tempo. A fórmula utilizada para o cálculo da velocidade em rad/s é:

$$\omega = \frac{\Delta\text{pulsos} \times 2\pi}{\text{PULSOS} \times 4 \times \text{SAMPLING TIME}} \quad \text{(rad/s)} \quad (2)$$

Cada transição nos sinais A e B (subida e descida) é contabilizada, resultando numa multiplicação por quatro da resolução efetiva — daí a utilização de um fator de 4 nos cálculos que envolvem o número de pulsos por volta (*pulses per revolution*, ou PPR). Estes cálculos permitem obter, com elevada precisão, a posição e a velocidade angular do motor, tirando partido da contagem automática dos pulsos em modo quadratura.

O erro de controlo é calculado como a diferença entre o *setpoint* e a velocidade medida y:

$$\text{erro} = \text{setpoint} - y \quad (3)$$

O controlador PID calcula o sinal de controlo como uma combinação de três componentes:

- **Componente Proporcional (P):** responde proporcionalmente ao erro atual, fornecendo uma correção imediata;
- **Componente Integral (I):** acumula o erro ao longo do tempo, eliminando desvios persistentes;
- **Componente Derivativo (D):** atua sobre a taxa de variação do erro, ajudando a reduzir oscilações e antecipar tendências.

Matematicamente, a saída do controlador é dada por:

$$\text{output}[k] = K_p e[k] + K_i \sum_{i=1}^{k} \frac{e[i]+e[i-1]}{2} h + K_d \frac{e[k]-e[k-1]}{h} \quad (4)$$

Na implementação prática, este cálculo é discretizado e realizado a cada período de amostragem. A saída do controlador é utilizada para ajustar o *duty cycle* do sinal PWM aplicado a cada motor, controlando assim a sua velocidade. Para garantir a segurança do sistema e respeitar os limites físicos, esta saída é limitada entre 0 e a tensão máxima fornecida pela bateria. No sistema implementado, a única forma de controlar a velociade é através da definição do *setpoint*.

A eficácia deste tipo de controlo depende fortemente da **sintonização adequada dos parâmetros do PID**, nomeadamente os ganhos proporcional (K_p), integral (K_i) e derivativo (K_d). Um ganho proporcional demasiado elevado pode induzir oscilações; um ganho integral incorretamente calibrado pode provocar acumulação de erro e instabilidade; e um ganho derivativo excessivo pode amplificar o ruído proveniente da medição da velocidade.

Uma análise mais detalhada do processo de determinação dos parâmetros do PID será apresentada na Subsecção VI, onde se descrevem os testes realizados e os critérios utilizados para alcançar uma resposta estável e eficaz.

Este controlo em malha fechada (Figura 21) permite uma regulação robusta e eficiente da locomoção do robô, mesmo perante perturbações externas, variações de carga ou assimetrias entre motores.

B. Interface de Comando via Aplicação Móvel

A interação com o robô HazMapper é realizada através de uma aplicação móvel () desenvolvida para a plataforma Android, utilizando a linguagem Kotlin. Esta aplicação serve como a interface principal de controlo e futura monitorização, oferecendo uma experiência de utilizador intuitiva para a operação remota do robô.

1) Protocolo de Comunicação e Otimização BLE: A comunicação com o módulo `ESP` do robô é realizada via **Bluetooth Low Energy (BLE)**, utilizando um **protocolo binário de dois bytes**. O **primeiro byte** define o tipo de comando (ex: `0x00` para ON/OFF, `0x01` para Direção, `0x02` para Velocidade), e o **segundo byte** especifica o valor associado. A estrutura completa desta gramática de comunicação é detalhada na Figura 4. Para otimizar o tráfego BLE, os comandos de movimento são enviados apenas quando os seus valores atuais diferem dos últimos transmitidos.

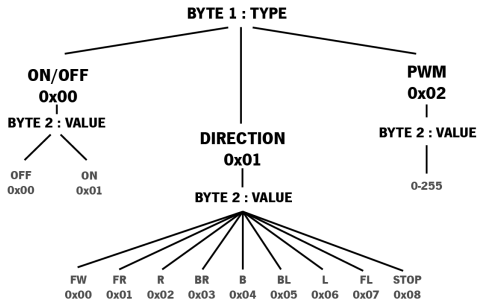

Fig. 4. Gramática - comandos

2) Funcionalidades da Interface: A interface de utilizador inclui um **Switch ON/OFF** e um **Joystick Virtual**:

- O **Switch ON/OFF** controla o estado operacional do robô. Ao ligar, envia um comando (`0x00, 0x01`). Ao desligar, além do comando (`0x00, 0x00`), a aplicação envia comandos de *STOP* (`0x01, 0x08`) e velocidade zero (`0x02, 0x00`), garantindo a imobilização imediata.
- O **Joystick Virtual** (`JoystickView`) permite controlo omnidirecional, traduzindo a posição do "handle" em comandos de direção e velocidade, enviados a cada 500ms.
  - **Direção:** A posição angular é quantificada em 8 direções discretas (ex: `FW`, `LEFT`) ou `STOP` (na *deadzone* central), enviadas como tipo `0x01`.
  - **Velocidade:** A distância do "handle" ao centro é normalizada de 0 (parado) a 255 (máximo), transmitida como tipo `0x02`. Uma histerese de 20 unidades é aplicada para estabilidade.

3) Conectividade: A aplicação gere as permissões Bluetooth necessárias (incluindo as específicas do Android 12+) e utiliza uma classe auxiliar (`BLEManager`) para a inicialização e gestão da conexão BLE, assegurando uma comunicação fiável com o robô.

Fig. 5. App - Android

C. Comunicação Bluetooth via ESP

Neste contexto, a `ESP` atua como um intermediário entre a aplicação Android e a `STM32H755` utilizada, fazendo *buffering* dos dados recebidos via Bluetooth, transmitindo-os à `STM` através da UART.

Uma vez que é importante ter em consideração os recursos do microcontrolador, foi implementado o Bluetooth Low Energy em vez do Bluetooth Classic, embora sejam introduzidas restrições a nível do *throughput*. A implementação do BLE foi feita recorrendo ao NimBLE, uma stack Bluetooth 5.4 *open-source* da Apache. Na `ESP` implementou-se um servidor **Generic Attribute Profile** (GATT) com característica de escrita, permitindo ao cliente GATT (aplicação Android) escrever nesse tópico. Entre os dois dispositivos existe *pairing*, sem *bonding*. De modo a garantir que não há perda de dados foram realizadas as seguinte considerações.

- **Máximo payload (ATT)** com Data Length Extension (DLE): 251B
- Isso equivale a: $\frac{251}{2} = 125$ **frames de 2B por pacote BLE**
- Considerando um máximo de 6 pacotes por evento de conexão (limite da pilha NimBLE):

$$6 \times 125 = 750 \text{ frames por evento}$$

- Com eventos a cada 15ms, temos aproximadamente:

$$\frac{1\,\text{s}}{15\,\text{ms}} = 66{,}66 \approx 66 \text{ eventos por segundo}$$

- Frames por segundo:

$$66 \times 750 = 49500 \text{ frames por segundo}$$

- Cada frame com 2 bytes:

$$49500 \times 2 = 99000\,\text{bytes/segundo} = 792000\,\text{bps}$$

- Portanto, para suportar essa taxa via UART, a ***baud rate mínima*** precisa de ser:

$$\boxed{\geq 792000bps}$$

Deste modo, a UART foi implementada com uma *baud rate* de 1 Mbps e com um *buffer* de 1024 bytes.

III. Sistema de Sensorização para Deteção de Riscos

A. Sensores de deteção de obstáculos

Com o intuito de tornar o robô sensível ao meio que o rodeia implementamos sensores de ultrassom para deteção de obstáculos. Tendo em perspetiva a realização do mapeamento do meio pretendíamos ter sensores que conseguiam reconhecer obstáculos existentes a uma distância considerável (superior a 0.5m). Como tal, implementamos sensores de estacionamento para automóveis no robô.

Deste modo, foi necessário compreender o funcionamento dos sensores e como eles reconheciam e interpretavam as distâncias detetadas. Estes são acoplados a um componente central que suporta até 8 sensores e que envia dados para um pequeno display. Após uma primeira observação, obteve-se a onda da Fig.6.

Fig. 6. Onda Total dos sensores de ultrassom

Após alguns testes, apercebemo-nos que a onda enviada se divide um 3 partes: Start Bit, Sync Bit e 8 bytes de informação (1 byte por sensor). Para além disso, reparamos, também, que o valor enviado nesses bytes era diretamente proporcional à distância detetada em centímetros, com uma razão de proporcionalidade de 10, tal como apresenta a tabela I.

Distância Detetada	**Valor Enviado (decimal)**
$d < 0.3\,\text{m}$	0
$0.3\,\text{m} \leq d < 0.4\,\text{m}$	3
$0.4\,\text{m} \leq d < 0.5\,\text{m}$	4
$0.5\,\text{m} \leq d < 0.6\,\text{m}$	5
$0.6\,\text{m} \leq d < 0.7\,\text{m}$	6
$0.7\,\text{m} \leq d < 0.8\,\text{m}$	7
$0.8\,\text{m} \leq d < 0.9\,\text{m}$	8
$0.9\,\text{m} \leq d < 1\,\text{m}$	9
$1.0\,\text{m} \leq d < 1.1\,\text{m}$	10
$1.1\,\text{m} \leq d < 1.2\,\text{m}$	11
$1.2\,\text{m} \leq d < 1.3\,\text{m}$	12
$1.3\,\text{m} \leq d < 1.4\,\text{m}$	13
$1.4\,\text{m} \leq d < 1.5\,\text{m}$	14
$d \geq 1.5\,\text{m}$	255

TABLE I
Distância detetada e respetivo valor enviado

Uma análise mais detalhada ao funcionamento das ondas dos sensores encontra-se no anexo **??**.

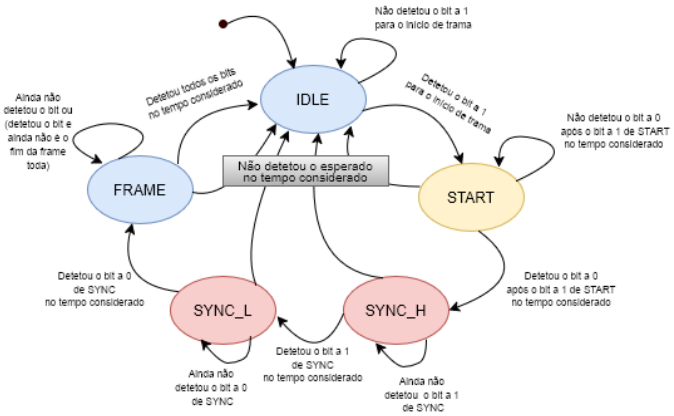

Fig. 7. Máquina de Estados implementada para os Sensores de Obstáculos

Conhecendo o princípio de funcionamento dos sensores, implementámos um método de descodificação baseado na amostragem do sinal enviado, com um período de aquisição de 10 μs. O processo de descodificação e tratamento da trama recebida é implementado através de uma máquina de estados finita (FSM), conforme ilustrado na Figura 7. Este sistema realiza a leitura sequencial dos bits recebidos, a validação por comparação com os padrões esperados para cada secção da trama e o tratamento adequado conforme o estado atual do processo. A cada condição da FSM acrescenta o a verificação de se foram lidos valores ou não. Caso não tenham sido, volta novamente para o estado onde se encontra.

B. Câmara de Monitorização para Zonas de Risco

De forma a tornar o robô aplicável à situação apresentada, além de dispor de controlo remoto para monitorizar zonas perigosas, foi essencial dotá-lo com a capacidade de transmissão de vídeo em tempo real, sendo assim implementado o módulo OV7670, uma câmara que utiliza o protocolo SCCB, compatível com I2C, para comunicar com a `STM32`. Esta comunicação é realizada de forma assíncrona, através de um barramento diferente dos outros sensores, pois neste módulo não se encontram integradas as resistências *pull-up* características do barramento I2C.

A `STM32` configura inicialmente a câmara para capturar imagens com resolução QVGA (320×240 pixels), em modo RGB565 e vídeo progressivo. O funcionamento da câmara assenta em sinais como o MCLK (clock mestre fornecido pela `STM32` à câmara- 16 MHz), PCLK (clock de pixel que sincroniza a saída de dados), HSYNC (sincronização horizontal que indica o início e fim de cada linha) e VSYNC (sincronização vertical que indica o início e fim de cada frame).

Cada frame contém 76 800 pixels, e com 16 bits por pixel, totaliza 153 600 bytes por frame. Para captar estes dados, o microcontrolador `STM32` recorre ao periférico DCMI (Digital Camera Memory Interface), que recebe os dados da câmara em modo de 8 bits por ciclo de PCLK. O DCMI está configurado para trabalhar em conjunto com o DMA, que transfere os dados recebidos diretamente para um buffer na memória RAM, evitando a intervenção contínua do CPU. Após a captura de um frame completo (`HAL_DCMI_FrameEventCallback`), o conteúdo do buffer é transmitido pela UART3 configurada a uma *baud rate* elevada (2 Mbps), para permitir uma transferência rápida dos dados para o computador. A transmissão é realizada também por DMA, assegurando que o microcontrolador não fica bloqueado e pode continuar a capturar novos frames ou executar outras tarefas.

No computador, um script em Pyhton recolhe os dados enviados pela UART, reconstrói os frames a partir da sequência de bytes e exibe o vídeo capturado em tempo real, permitindo a visualização do ambiente envolvente.

Devido às limitações da aplicação embebida utilizada no que diz respeito ao suporte de periféricos multimédia, não foi possível obter um *streaming* de vídeo contínuo. Como alternativa, conseguimos apenas visualizar imagens de forma intervalada, ainda que com um atraso reduzido. A limitação da plataforma `STM32`, especialmente no processamento e transmissão eficiente de dados de imagem, impede a implementação de uma câmara verdadeiramente *on board*, obrigando à sua ligação ao computador via cabo USB.

Apesar da adoção de mecanismos para melhorar o desempenho, como o aumento da *baud rate* e a utilização de *DMA* (Direct Memory Access), o sistema continua sem capacidade suficiente para suportar uma transmissão fluida. Considerámos ainda a implementação de compressão JPEG para comprimir os dados transferidos, mas essa abordagem, embora beneficie a largura de banda, não resolve por completo o problema da velocidade de aquisição e envio.

Adicionalmente, o uso do modo `RGB565`, embora eficiente em termos de profundidade de cor e compatibilidade com a maioria dos sensores, requer dois ciclos de *clock* para a transmissão de cada pixel, o que contribui ainda mais para a latência do sistema e agrava a dificuldade de alcançar uma taxa de atualização próxima do tempo real.

C. Monitorização da Temperatura

A monitorização da temperatura do meio ambiente é crítico para a avaliação de riscos do HazMapper. Para isso, utilizamos um sensor de temperatura digital (HDC1080) com interface I2C. A comunicação é gerenciada pela `STM32` através de uma Máquina de Estados Finita (FSM - representada na Fig.8) assíncrona, garantindo que as operações de leitura do sensor não bloqueiem outras funcionalidades do robô.

A FSM é composta por quatro estados principais que coordenam a aquisição de dados de temperatura de forma não-bloqueante:

- **SENSOR_STATE_IDLE**: Estado inicial, onde o sistema aguarda um sinal para iniciar uma nova medição.
- **SENSOR_STATE_TEMP_TX**: Transmite-se o comando de leitura de temperatura ao HDC1080 via I2C.A transição para o próximo estado ocorre após a conclusão da transmissão.

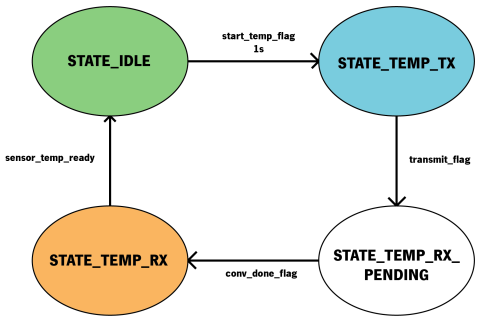

Fig. 8. Máquina de Estados - leitura do sensor HDC1080

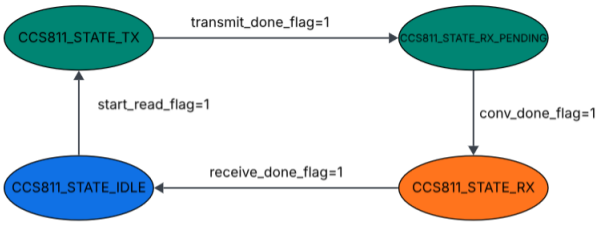

Fig. 9. Máquina de estados- leitura do sensor CCS811

- **SENSOR_STATE_TEMP_RX_PENDING**: Após a transmissão, o sistema aguarda o tempo de conversão interna do sensor (15ms para o HDC1080).
- **SENSOR_STATE_TEMP_RX**: Os 2 bytes de dados de temperatura são lidos do HDC1080 via I2C Uma vez recebidos os dados brutos são processados e convertidos para o valor em Celsius utilizando a fórmula: $T_{celsius} = ((raw/65536.0f) * 165.0) - 40.0$.

D. Monitorização dos níveis de gás

A monitorização dos gases em redor do HazMapper é uma componente essencial para garantir uma vigilância eficaz das áreas de risco. Para tal, foi integrado o sensor digital CCS811, responsável por medir a concentração de dióxido de carbono equivalente (eCO) e compostos orgânicos voláteis totais (TVOC) através de comunicação I2C.

A estrutura de programação deste sensor segue o mesmo princípio aplicado ao sensor de temperatura: foi implementada uma máquina de estados assíncrona.

Num primeiro momento, o sensor é configurado através da função `ccs811_init` na qual configuramos o pino WAKE de modo a que esteja a low para aceitar comandos I2C e configuramos a transição do boot mode para o application mode. Para isso, é enviado um comando especial, o byte 0xF4, designado pelo fabricante como `APP_START`. Este comando não corresponde à escrita num registo específico, mas sim a uma instrução direta que ordena ao CCS811 que inicialize o firmware interno de medição de gases, ficando pronto para fornecer dados de qualidade do ar. Após esta inicialização, o registo de modo de funcionamento `MEAS_MODE` é também configurado para definir a frequência de amostragem e eventuais interrupções. A partir daí, toda a leitura de dados é gerida pela máquina de estados, que coordena a sequência "pedido de leitura → espera de conversão → receção dos dados", garantindo que a comunicação I2C decorre de forma fiável e sem conflitos.

Após esta inicialização, aproveitou-se a configuração do temporizador já utilizada para o sensor de temperatura, aplicando uma estrutura semelhante para que o CCS811 também realizasse medições com uma frequência de 1 segundo.

Através da função `start_read`, é enviado o endereço do registo `0x02`, que corresponde ao bloco de dados de medição. Em seguida, o receive armazena o conteúdo num array de 8 bytes, onde os primeiros dois bytes contêm o valor de CO equivalente (eCO), o terceiro e quarto bytes contêm o valor de TVOC e os restantes incluem informação adicional, como o estado do registo STATUS.

A máquina de estados é composta por quatro estados principais:

- **CCS811_STATE_IDLE**: Estado inicial, no qual o sistema permanece em espera e, ciclicamente, verifica se é necessário iniciar a leitura do registo `0x02` do sensor.
- **CCS811_STATE_TX**: Estado de transmissão, onde o endereço do registo `0x02` é enviado ao sensor. Após esta transmissão, é introduzido um tempo de espera de aproximadamente 5 ms para garantir a disponibilidade dos dados.
- **CCS811_STATE_RX_PENDING**: Decorrido o tempo de espera, o sistema inicia a receção dos 8 bytes de dados provenientes do registo `0x02`. A informação recebida inclui, em particular, os valores de CO2 e TVOC.
- **CCS811_STATE_RX**: Estado final da leitura, em que os 4 bytes de interesse (do CO2 e TVOC) já foram processados. Neste ponto, é ativada uma flag para indicar que a receção terminou com sucesso.

Após esta etapa, foi implementado um código para controlar um display LCD 1602A, com o objetivo de apresentar, em tempo real, os valores medidos dos gases: o CO em ppm e o TVOC em ppb. Desta forma, tornou-se possível visualizar diretamente, no display, os dados adquiridos pelo sensor CCS811.

E. Monitorização da bateria

A constante avaliação da tensão da bateria é crucial para a autonomia e gestão de energia do robô, permitindo o ajuste do controlo de velocidade dos motores e alertas sobre níveis críticos. Este processo é implementado utilizando o **Conversor Analógico-Digital (ADC)** da própria `STM32`.

A tensão da bateria é, primeiramente, reduzida por um **divisor de resistivo** ($R_1 = 50k\Omega$, $R_2 = 10k\Omega$) para ser compatível com a faixa de entrada do ADC (suporta até 3.3V). O valor digital de 12 bits (raw_value) obtido pelo ADC é então convertido para Volts (V) através da seguinte fórmula:

$$V_{\text{BAT}} = \text{raw\_value} \times \frac{V_{\text{REF}}}{ADC_{\text{MAX}}} \times \frac{R_1 + R_2}{R_2} \tag{5}$$

Onde $V_{\text{REF}} = 2.0\,\text{V}$ e $ADC_{\text{MAX}} = 4095$.

A aquisição dos dados do ADC é realizada de forma periódica a cada 1ms, com a conversão iniciada via **Direct Memory Access (DMA)** (`HAL_ADC_Start_DMA`), minimizando a carga no CPU. A função `Battery_process()` interpreta a leitura da tensão (`vbat`) e, com base num limiar de 11.0V, ativa LEDs indicadores (laranja para bateria OK, vermelho para bateria crítica), alertando o utilizador sobre o estado de carga.

F. Transmissão de Dados de Sensores via Wi-Fi com `ESP`

A transmissão dos dados via Wi-Fi com a `ESP` visa conferir uma monitorização cuidada do sistema, bem como do ambiente que o rodeia.

Para tal, configurou-se a `ESP` como Access Point (AP). Um dispositivo (o computador, por exemplo) conecta-se a essa rede Wi-Fi e, após estar corretamente configurado. pode receber/enviar dados a partir dessa conexão, quer através de um servidor HTTP (Modelo OSI: HTPP - camada de Aplicação, baseado em TCP/IP - camada de transporte), quer através de sockets UDP. Como o envio das dados é frequente, a perda de algum dado não é necessariamente crítica, e, como tal, recorremos ao envio de sockets UDP para os dados dos sensores. Para além disso, por não requerer confirmação, o UDP introduz menor *overhead*.

IV. Integração do RTOS

A implementação de um Sistema Operativo de Tempo Real (RTOS) no sistema visou otimizar a eficiência computacional e o desempenho global, assegurando:

- Comportamento **determinístico**;
- Baixa **latência**;
- *Jitter* controlado.

Para tal, adotou-se o **FreeRTOS**, aproveitando os seus mecanismos avançados de:

- Comunicação entre processos (IPC);
- Escalonamento de tarefas por prioridades;
- Gestão de memória estática.

V. Estrutura de Hardware Integrado em PCB

Para este projeto, decidimos desenvolver uma PCB personalizada com o objetivo de minimizar a utilização de fios e reduzir interferências eletromagnéticas. A PCB foi projetada em duas secções principais: uma de potência e outra de sinais digitais.

Entre estas duas zonas, foi criada uma área de isolamento, onde se encontram os isoladores capacitivos, como podemos ver na Figura 10. Estes são responsáveis por garantir o isolamento galvânico entre os sinais de potência e os sinais digitais, especialmente nos casos em que sinais de potência precisam ser lidos pela `STM32` ou vice-versa. Para o canal ADC, foi utilizado um isolador analógico juntamente com um díodo Zener de 3,3V para proteger o pino da `STM32` contra sobretensões.

De forma a garantir o isolamento total entre os domínios de potência e digital, também foi utilizado um conversor DC-DC isolado e regulado para alimentar a `STM32`. Certificámo-nos

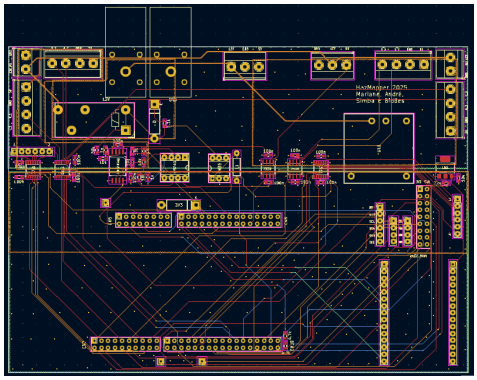

Fig. 10. Layout da PCB

ainda de que todos os sinais partilhados entre os dois domínios estavam devidamente isolados. Como os pinos da `STM32` fornecem, em média, apenas 25 mA de corrente, decidiu-se utilizar um regulador *LDO* (de 5 V para 3,3 V) dedicado. Esta solução garante que os sensores e demais componentes que necessitam de alimentação a 3,3 V não sejam alimentados diretamente pela `STM32`, a qual não conseguiria fornecer corrente suficiente de forma segura e estável.

Foi também implementado um mecanismo de proteção no circuito de alimentação. Assim que o botão de ON/OFF é pressionado, um relé é ativado para ligar a alimentação de potência. O sinal de controlo enviado pela `STM32` para o relé passa por um optoacoplador, garantindo o isolamento elétrico entre o microcontrolador e a carga de potência. Adicionalmente, foi colocado um díodo de flyback (freewheeling) paralelo à bobina do relé para proteger o circuito contra picos de tensão gerados pela comutação do relé, como podemos ver na Figura 24.

Na Figura 10 é possível observar os suportes dedicados para a `STM32`, que ficará montada de forma invertida, conforme mostrado Figura 11. Também existem suportes para a `ESP`, entradas para os encoders dos motores, ligações para as alimentações externas e conexões para os sensores.

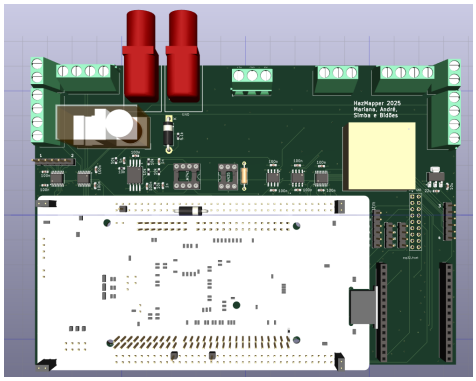

Fig. 11. Visualização 3D da PCB

A PCB foi construída com quatro layers na seguinte disposição: sinal, GND, sinal, GND. Esta configuração ajuda a garantir uma boa integridade dos sinais e minimizar interferências. Foi também realizado stitching de vias ao longo

da placa, especialmente em zonas críticas e onde circulam sinais de alta frequência (na ordem dos MHz), como o MCLK e o PCLK da câmara. Nessas regiões, foi adicionado extra stitching para melhorar o plano de retorno e reduzir ruído. Os sinais digitais de comunicação foram organizados na primeira camada da PCB, e foram criados planos dedicados para otimizar o roteamento e reduzir interferências entre os circuitos.

A conceção desta PCB permitiu integrar de forma segura e eficiente os circuitos de potência e controlo, garantindo isolamento adequado, proteção contra ruídos e picos de tensão, e um layout otimizado para desempenho eletromagnético. O uso de componentes SMD e uma estrutura de quatro layers contribuiu para a miniaturização e estabilidade do sistema.

VI. Testes e Validação dos Módulos

A. Controlo via Teclado para Direção e PWM

Inicialmente, para testar se a estrutura mencionada no Capítulo II-A funcionava corretamente — ou seja, se as direções e a velocidade realmente alteravam conforme esperado — decidimos implementar o controlo por meio do teclado do computador, via UART3.

Para isso, definimos dois arrays:

```
char* valid_movements[NUM_MOVES] = {"W",
"E", "D", "C", "X", "Z", "A", "Q", "S"};

const float valid_pwm[] = {0, 10, 20,
30, 40, 50, 60, 70, 80, 90, 100};
```

Cada letra correspondia a um comando de movimento específico, fazendo com que as rodas girassem nas direções previstas. Já os números pressionados no teclado determinavam o duty cycle aplicado pelo controlador PWM, usando o valor correspondente no array `valid_pwm`, que era passado para a função `update_pwm`.

Dessa forma, pudemos comprovar que o robô alterava o seu comportamento de movimento de acordo com os comandos enviados, validando uma primeira fase do funcionamento do sistema de controlo.

B. Encoders

Para testar se os encoders estavam a funcionar corretamente, observámos os sinais A e B num osciloscópio. Confirmámos que os sinais estavam desfasados em 90° (Figura 12), característica essencial da codificação em quadratura.

Além disso, para determinar o sentido positivo da velocidade, analisámos a relação entre as transições dos sinais. No nosso caso, o robô desloca-se para a frente quando o motor roda no sentido anti-horário. Assim, quando o sinal B faz a sua transição, se o sinal A estiver em nível lógico alto (1), o motor está a rodar no sentido anti-horário (movimento para a frente). Por outro lado, se o sinal A estiver em nível lógico baixo (0) nesse momento, o motor está a rodar no sentido horário (movimento para trás).

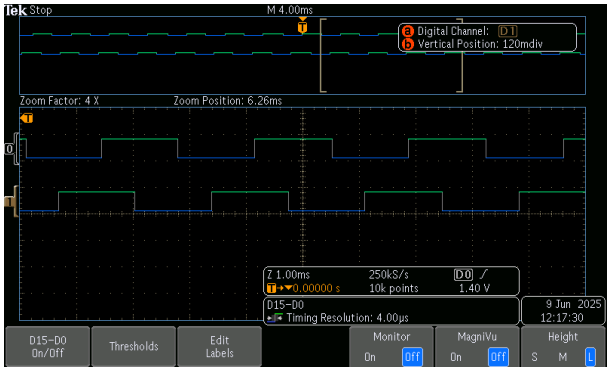

Fig. 12. Sinais A (1) e B(0) do Encoder em Quadratura

C. Cálculo e ajuste dos parâmetros PID

Para implementar o controlo PID, primeiro foi necessário determinar os parâmetros de cada motor, seguindo o procedimento igual ao realizado no Guia 2 de PI2. Utilizámos a mesma equação de estados (Equação 8) e realizámos os testes sob as mesmas condições para todos os motores: aplicámos uma tensão constante ao motor, com ua $= 9{,}2\,V$ e PWM $= 76{,}7\%$. Assim, medimos a corrente e a velocidade de rotação correspondentes, como podemos observar na Tabela II.

Os outros parâmetros do modelo foram calculados com base nas expressões fornecidas no Guia2, mas em específico para os nossos motores, nomeadamente: o momento de inércia do sistema mecânico (Equação 9), o coeficiente de fricção linear (Equação 11), a constante eletromecânica (Equação 10) e a resistência elétrica do motor. Os resultados encontram-se na Tabela II.

De seguida, procedemos à definição do período de amostragem. Para tal, utilizámos um script em MATLAB que calcula o período de amostragem ideal. Este deve ser suficientemente pequeno para que o controlador consiga detetar, de forma significativa, a presença da dinâmica mais rápida do sistema, associada à função de transferência relacionada com o polo dominante.

Este polo é aquele com maior módulo, correspondendo à maior frequência de corte na resposta em frequência do sistema. Conforme ilustrado na Figura 13, adotámos a seguinte relação para o período de amostragem:

$$T_s = \frac{1}{8 \times |polo|} \tag{6}$$

Como todos os polos dos motores analisados apresentaram valores muito próximos, e para simplificar o desenvolvimento do código, optámos por utilizar o mesmo período de amostragem para todos os motores.

Com o período de amostragem definido, procedemos ao cálculo dos parâmetros do controlador PID usando o "PID Tuner". A Figura 14 exemplifica como a implementação do controlo PID permite ao sistema atingir uma resposta mais rápida e estável.

Após calcular os parâmetros do controlador PID para cada motor (Tabela II), realizámos testes práticos. Para melhorar

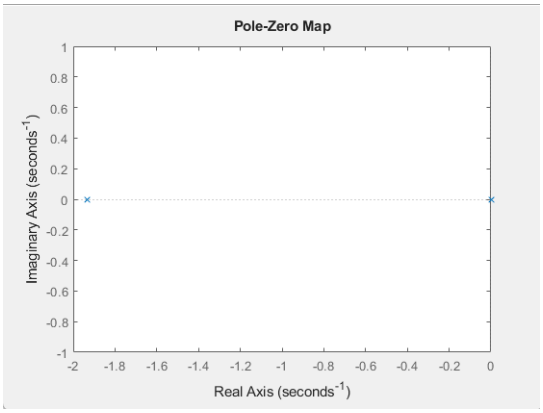

Fig. 13. Pólos do sistema

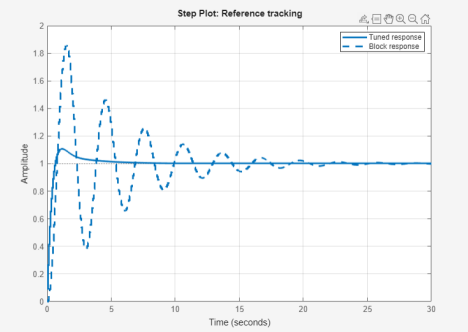

Fig. 14. Sistema com controlo vs sem controlo PID

a qualidade da resposta, foi necessário adicionar um filtro para eliminar ruídos e perturbações. No entanto, mesmo assim, a resposta apresentava um overshoot considerável. Para corrigir este comportamento, fomos ajustando manualmente os parâmetros do PID, com destaque para o ganho integral (K_I), que foi progressivamente reduzido para diminuir o overshoot.

Em alguns casos, aumentámos ligeiramente o ganho derivativo (K_D) para melhorar a resposta transitória e aumentar a estabilidade do sistema, pois este ganho ajuda a antecipar as variações da resposta e a amortecer oscilações.

De forma geral, o ajuste dos parâmetros PID foi feito seguindo estes passos:

1) Ajustar K_P para obter uma resposta rápida, mas sem causar instabilidade.
2) Reduzir K_I para evitar excesso de overshoot e reduzir o erro e o tempo de estabilização.
3) Incrementar K_D para suavizar a resposta e minimizar oscilações na fase transitória.

Este processo foi iterativo e necessário para garantir um controlo eficiente e robusto para cada motor testado.

D. ADC

Para testar o ADC, começámos por utilizar um potenciómetro, que apenas permitia uma tensão máxima de 3,3 V, garantindo assim a segurança do pino da STM32 e confirmando o funcionamento correto do conversor analógico-digital. Após esta verificação inicial, avançámos para o dimensionamento do circuito de medição da bateria.

	Motor 1	**Motor 2**	**Motor 3**	**Motor 4**
ia (A)	0.16	0.15	0.14	0.21
w (rad/s)	25.9	27.2	27.2	26.3
J (kg/m^2)	0.07	0.07	0.07	0.07
Bf (N/(m/s))	0.0019	0.0016	0.0015	0.0025
Km (N/A)	0.306	0.293	0.292	0.317
R (ohm)	8.06	8.13	9.03	4.15
Kp (teórico)	1.8	1.67	1.51	0.9
Ki (teórico)	0.65	0.51	0.41	0.3
Kd (teórico)	0.72	0.63	0.63	0.57
Filtro	0.67	0.67	0.67	0.67
Sampling time (ms)	65	65	65	65

TABLE II
PARÂMETROS DOS MOTORES E CONTROLADORES

E. Módulo OV7670- Câmara

Para validar o correto funcionamento da câmara OV7670, começámos por testar a comunicação I2C. O datasheet fornecia os endereços de escrita (0x42) e leitura (0x43) do dispositivo, assim como os endereços internos dos registos. Como se pode observar na Figura 25 capturada no osciloscópio, foi possível realizar com sucesso uma escrita, seguida de uma leitura do registo 0x0B, correspondente ao identificador do dispositivo. O valor lido foi 0x73, o que confirma que a comunicação está a funcionar corretamente e que a câmara foi devidamente identificada.

Na Figura 26, é possível observar o sinal **PCLK**, cuja frequência está aproximadamente nos 16, correspondendo à frequência do sinal **MCLK** fornecido à câmara. O **PCLK** define o ritmo a que os 8bits de dados de cada pixel são disponibilizados nas linhas de dados (`D[7:0]`), sendo fundamental para a sincronização da leitura da imagem.

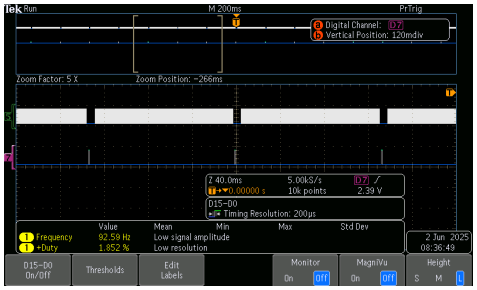

Fig. 15. Sinais de sincronização da OV7670 - HSYNC (5) e VSYNC (7)

Na Figura 15, é possível identificar os sinais **HSYNC** e **VSYNC**. O **VSYNC** ocorre com uma frequência significativamente menor, pois indica o fim de cada *frame* completo, enquanto o **HSYNC** sinaliza o início de cada linha da imagem. Devido à escala da captura, o **HSYNC** não é claramente visível, já que ocorre a cada 240 linhas (em resolução 320 x 240), tornando-se demasiado frequente para ser distinguido nessa visualização. Assim, comprovou-se o correto funcionamento da câmara, validando a comunicação

e os sinais essenciais. Com esta etapa concluída, foi então possível avançar para a implementação da recolha de dados de imagem.

F. BLE ESP

Para validar o funcionamento do BLE na ESP recorremos à aplicação nRF Connect. Aqui conseguimos verificar se a ESP está detetável (Fig.16) e se o tópico GATT com o UUID personalizado, designado para serviço de escrita aparece disponível (Fig. 17).

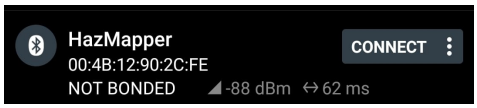

Fig. 16. ESP com BLE detetável)

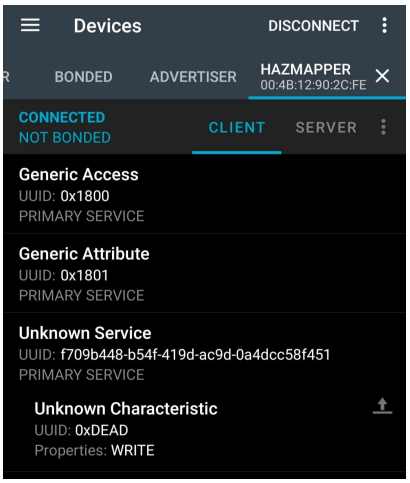

Fig. 17. Tópico GATT para Esrita

G. Wi-Fi ESP

Com o intuito de verificar se a conexão Wi-Fi estava funcional, verificamos no computador se a rede criada pela ESP como AP aparecia (Fig.18).

Fig. 18. Rede Wi-Fi criada pela ESP como AP

De seguida, conectamo-nos a essa rede e verificamos, através de um script *python*, direcionado para a porta UDP configurada na ESP, se estávamos a receber dados de teste enviados pela ESP através de um socket UDP, tal como demonstra a figura 19.

VII. Desvios da Ideia Inicial

Inicialmente, um dos grandes objetivos era ter implementado um *digital-twin* que replicava os movimentos do robô em ROS2. Contudo, devido a restrições de tempo, não foi possível tê-lo totalmente implementado à data da entrega do relatório, apesar de ter iniciado algum trabalho (Anexo X-J).

```
C:\Users\aamar>cd Desktop

C:\Users\aamar\Desktop>python udp_receiver.py
Recebido de ('192.168.4.1', 64101) : Hello from ESP32!
Recebido de ('192.168.4.1', 64101) : Hello from ESP32!
Recebido de ('192.168.4.1', 64101) : Hello from ESP32!
Recebido de ('192.168.4.1', 64101) : Hello from ESP32!
```

Fig. 19. Receção Mensagens UDP

VIII. Trabalho Futuro

- Conseguir replicar os movimentos do robô utilizando o ROS2 no computador e μROS na STM (Anexo X-J;
- Implementação de ROS2 e μROS para conseguir produzir um *digital-twin* no robô no computador;
- Implementação de um servidor HTTP MJPEG através do Wi-Fi ESP com o intuito de reproduzir a imagem enviada pela enviada (Anexo X-I);
- Implementação de módulo de GPS para analisar a localização exata onde foram registados os dados;
- Implementação de módulos de comunicação por radiofrequência a longa distância (LoRa);
- Implementação de microcontrolador com interface multimédia com maior capacidade (Raspberry Pi).

IX. Conclusão

O desenvolvimento deste robô teve como principal objetivo criar uma solução capaz de se deslocar em ambientes potencialmente perigosos, recolhendo informação relevante através de sensores e comunicando-a de forma remota. Ao longo do projeto, foi possível integrar várias tecnologias de forma funcional: o ESP32 garantiu não só o controlo via Bluetooth BLE, facilitando a interação direta com o utilizador, como também a recolha e envio de dados através de Wi-Fi. A STM32 ficou responsável por toda a interface direta com o robô e os módulos integrados.

Foram adicionados sensores de temperatura, gás e estacionamento, que ajudam o robô a perceber melhor o ambiente e aumentam a segurança durante a navegação. A inclusão de rodas omnidirecionais revelou-se bastante útil, pois permite uma movimentação precisa, algo que é muito vantajoso em espaços mais apertados. Também foi feita a ligação a um módulo de câmara, com vista a permitir, no futuro, a criação de mapas visuais das áreas percorridas. A monitorização do estado da bateria garantiu maior fiabilidade ao sistema.

No geral, o projeto atingiu a maioria dos objetivos propostos, demonstrando que é possível combinar diferentes módulos e microcontroladores para criar uma plataforma móvel versátil, estando sempre cientes da suas limitações. Apesar disso, e como já referido, ficaram identificadas algumas melhorias possíveis para implementação futura.

X. Anexos

A. Análise- Diagrama de Use-Case

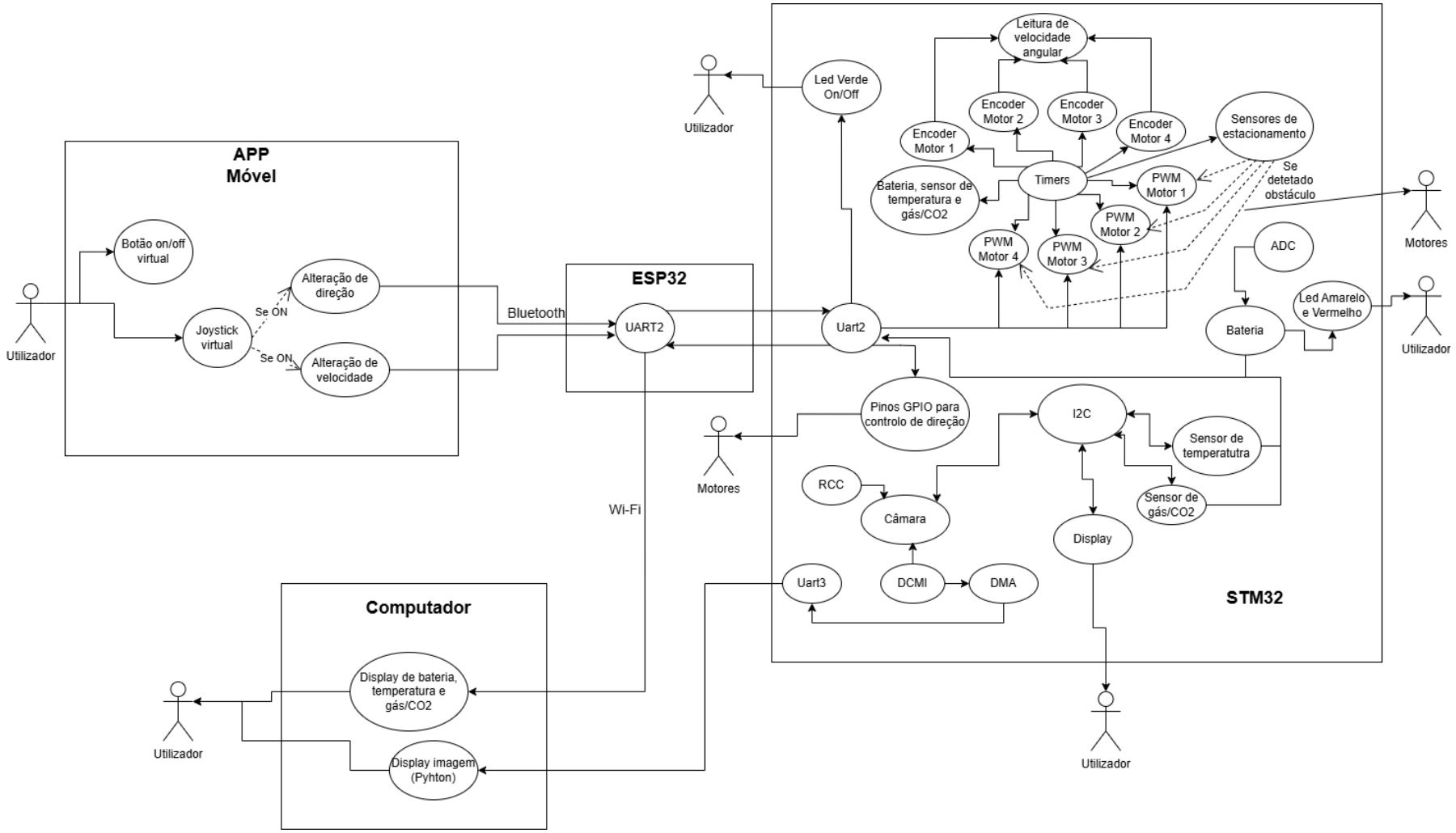

Fig. 20. Diagrama de *Use-Case*

B. Controlo da Ponte-H

Para o controlo dos motores de corrente contínua utilizados no robô HazMapper, recorreu-se à utilização de pontes H do tipo L298N. Este circuito integrado permite comandar motores DC em ambos os sentidos de rotação, através do controlo de dois sinais digitais de entrada por motor. O L298N possui dois canais, o que permite controlar dois motores com um único módulo.

No sistema desenvolvido, foram utilizadas duas placas L298N, uma para cada par de motores (frontal e traseiro), garantindo assim controlo independente de cada roda. A direção de rotação de cada motor é determinada pela combinação lógica dos sinais de entrada, conforme ilustrado nas Tabelas III e IV.

Além do controlo da direção, a velocidade dos motores é ajustada através da modulação por largura de impulso (PWM), aplicada ao terminal de ativação do L298 (ENA ou ENB), permitindo um controlo suave e preciso da locomoção do robô.

In1	**In2**	$V_{\text{Out1}} - V_{\text{Out2}}$
1	1	0
0	0	0
0	1	REVERSE
1	0	FORWARD

TABLE III
SENTIDOS DE ROTAÇÃO PARA OS VALORES DE IN1 E IN2

In3	**In4**	$V_{\text{Out4}} - V_{\text{Out3}}$
1	1	0
0	0	0
0	1	REVERSE
1	0	FORWARD

TABLE IV
SENTIDOS DE ROTAÇÃO PARA OS VALORES DE IN3 E IN4

C. Controlo em malha fechada

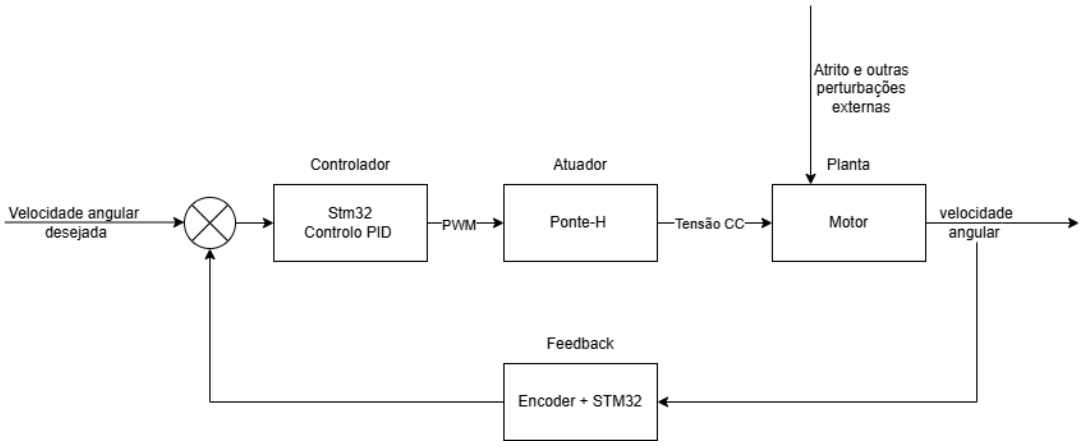

Fig. 21. Esquemático do sistema em malha fechada

D. Sensores de Ultrassom

O módulo da Figura 22 é o módulo que vem com os sensores quando são comprados e onde se conecta a alimentação dos sensores, (conector JST vermelho), os dados recebidos e os vários sensores.

Através da análise da Figura 23 conseguimos ver a informação enviada pelo módulo dos sensores no momento em que um dos sensores está a detetar obstáculos. Vemos o Start Bit, o Sync Bit (identificado como "*nd*") e reparamos que os primeiros 8 bits enviados a seguir ao bit de sincronização não têm todos o mesmo *duty-cycle*. Após análise em várias frames, concluimos que bits com *duty-cycle* menor ($\approx 30\%$) equivalem a um bit com valor lógico 0 e bits com *duty-cycle maior* ($\approx 60\%$) equivalem a um bit com valor lógico 1. Portanto, o valor recebido nesta frame, para o primeiro sensor foi 3, em decimal, o que indica a deteção de um obstáculo a uma distãncia compreendida entre 0.3m e 0.4m na direção do sensor.

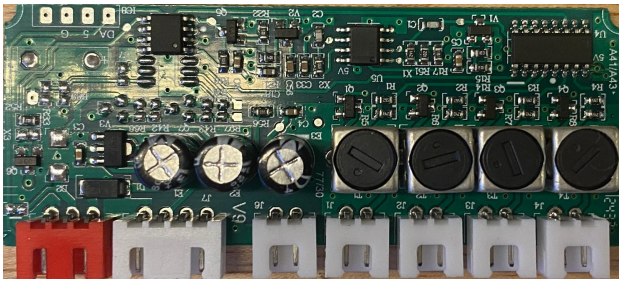

Fig. 22. Módulo dos Sensores de Ultrassom

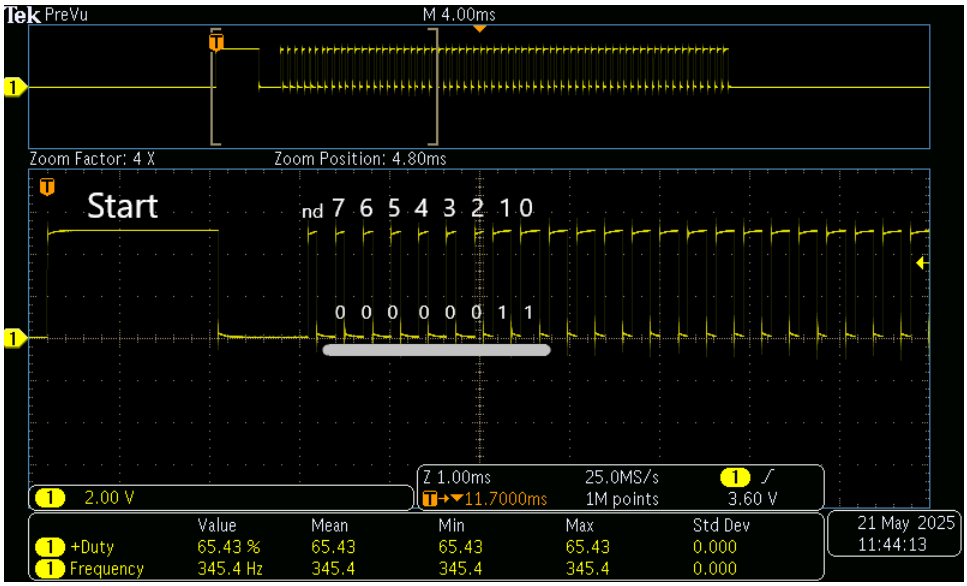

Fig. 23. Análise da Onda Enviada pelo módulo dos Sensores

E. Esquemático da PCB

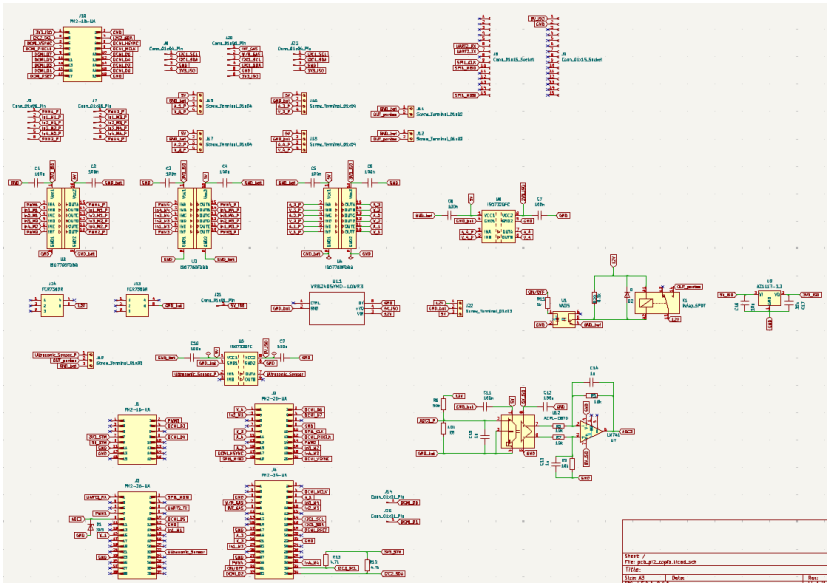

Fig. 24. Esquemático da PCB

F. Imagens relevantes para o Módulo da Câmara

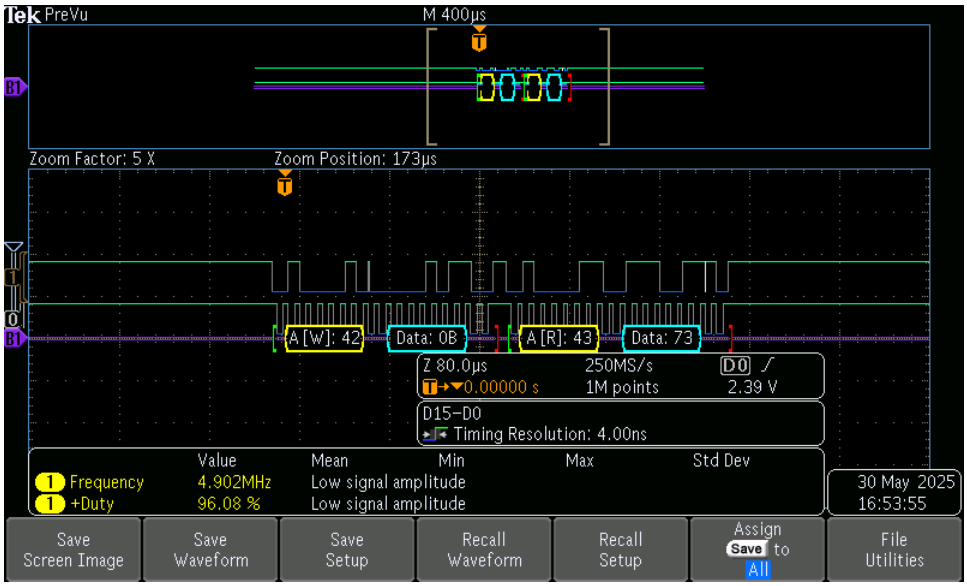

Fig. 25. Comunicação I2C com a câmara OV7670

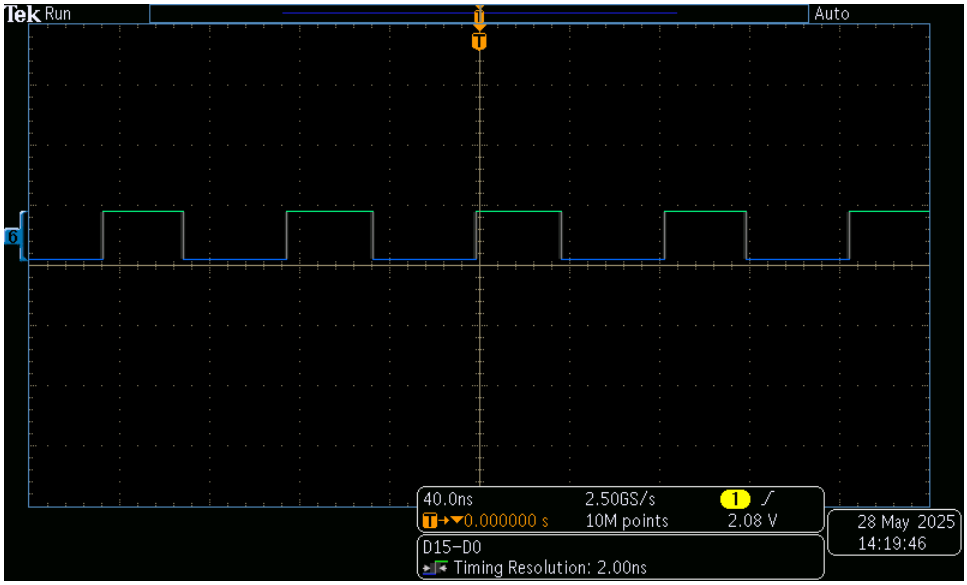

Fig. 26. Sinal PCLK da OV7670

G. Equações para o cálculo dos parâmetros do Motor

$$\begin{bmatrix} \frac{d\omega(t)}{dt} \\ \frac{d\theta(t)}{dt} \end{bmatrix} = \begin{bmatrix} -\frac{B}{J} & -\frac{K_m^2}{RJ} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \omega(t) \\ \theta(t) \end{bmatrix} + \begin{bmatrix} \frac{K_m K_w}{RJ} & \frac{1}{J} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u(t) \\ m_d(t) \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} \omega(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \omega(t) \\ \theta(t) \end{bmatrix} \tag{8}$$

$$J = \frac{1}{2} m r^2 \tag{9}$$

$$u_a(t) = R \cdot i_a(t) + K_m \cdot \omega(t) \tag{10}$$

$$B = \frac{m_m}{\omega_{ss}} \tag{11}$$

H. Organização do Sistema implementado com FreeRTOS

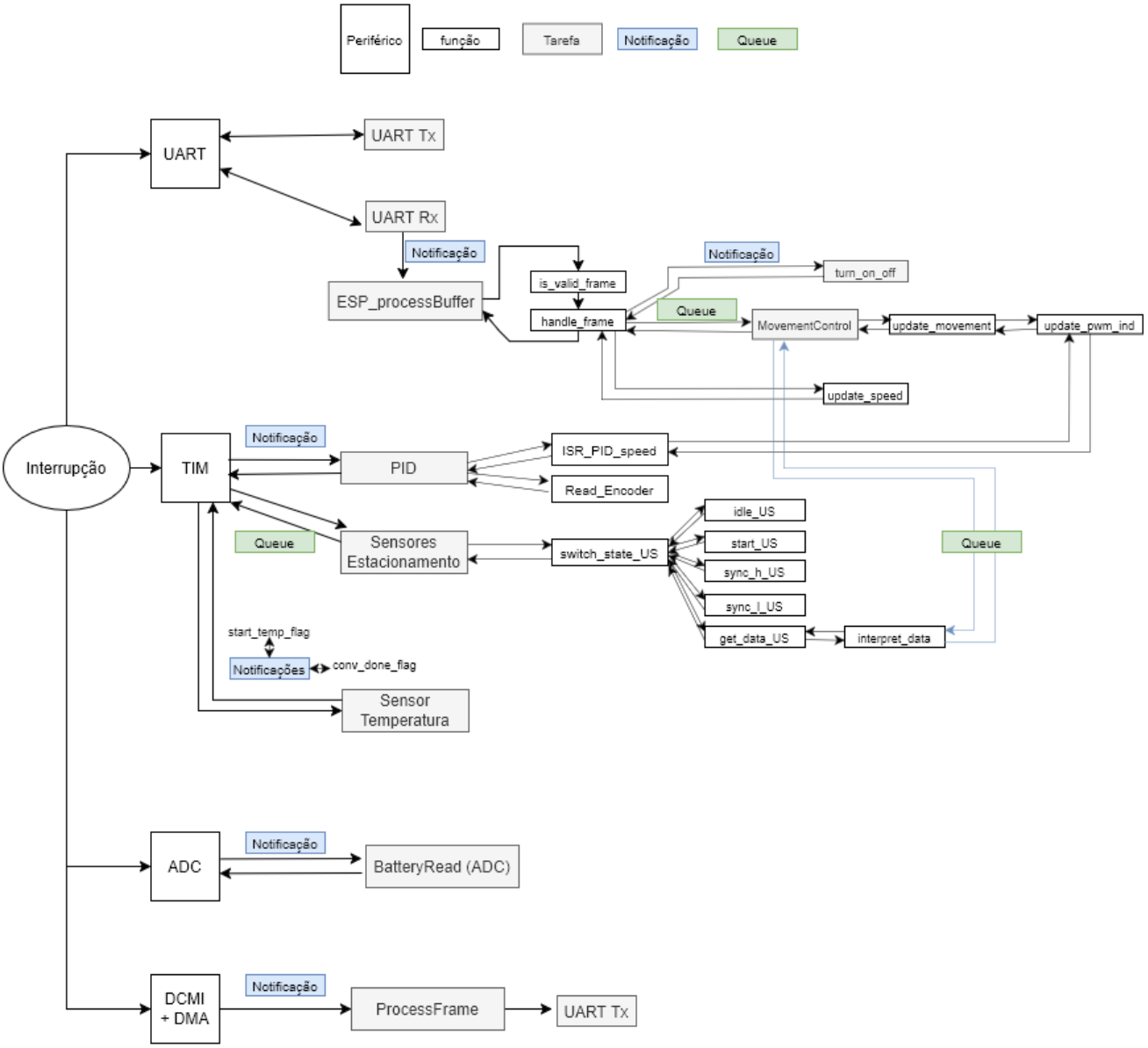

Fig. 27. Organização do Sistema implementado com FreeRTOS

I. Servidor HTTP MJPEG

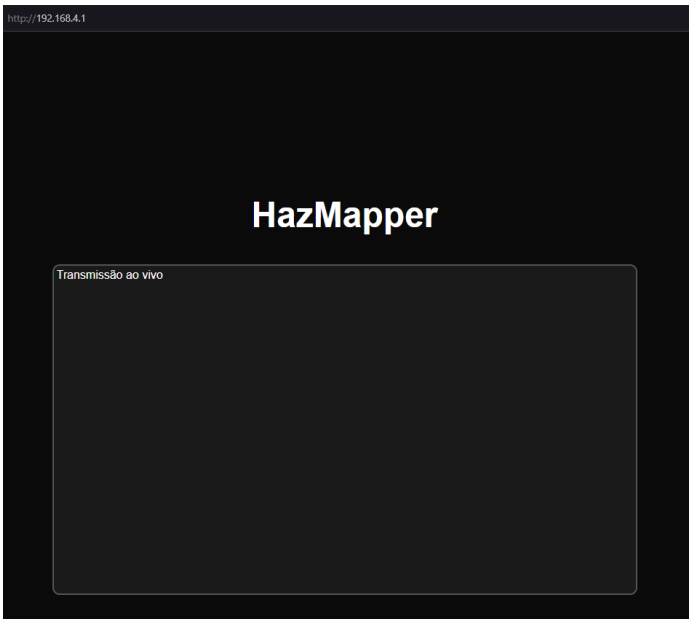

Fig. 28. Servidor HTTP criado

Foi criado um servidor HTTP MJPEG para transmissão em tempo real de imagens JPEG enviadas pela `STM` e recebidas pela `ESP`. O intuito principal seria receber as imagens no formato RGB565 na `ESP` e convertê-las para JPEG. Esta imagem seria posteriormente transmitida, frame a frame, através de um servidor HTTP MJPEG designado para o efeito. À data de entrega, o servidor estava montado (Figura 28) no endereço 192.168.4.1 (endereço Wi-Fi da `ESP`), contudo não foi possível realizar o teste. Os resultados previstos seriam uma transmissão com alguma latência relativamente à realidade devido às limitações do *hardware*, contudo, com uma implementação do tipo *header* e *footer* para envio e receção de imagem, seria possível garantir um *jitter* constante e relativamente baixo.

J. Integração do micro-ROS com STM32H755 para simulação de Digital Twin

O sistema desenvolvido tem como objetivo controlar os movimentos da tartaruga no simulador **turtlesim**, utilizando os dados de direção gerados pelo robô físico. Para tal, foi utilizada a tecnologia **ROS2 (Robot Operating System 2)**, que permite comunicação modular entre diferentes componentes através de *nós*, *tópicos* e *mensagens*.

O **ROS2** é um middleware amplamente utilizado em sistemas robóticos modernos, permitindo a interligação entre módulos de forma distribuída e eficiente. Um dos simuladores frequentemente utilizados para testes e aprendizagem é o **turtlesim**, que simula o movimento de uma tartaruga em duas dimensões.

O movimento da tartaruga é controlado através do tópico `/turtle1/cmd_vel`, que recebe mensagens do tipo `geometry_msgs/msg/Twist`. Estas mensagens contêm dois vetores: `linear` e `angular`, ambos com três componentes (x, y, z). Para este projeto, apenas o vetor `linear` será utilizado, especificamente os eixos `x` e `y`, para definir a direção da tartaruga no plano 2D.

Como o núcleo `M4` do microcontrolador `STM32H755` não possui capacidades de comunicação por Wi-Fi, foi necessário utilizar o **micro-ROS**, uma versão compacta do ROS2 desenhada para correr em microcontroladores com recursos limitados. O micro-ROS é responsável por criar o nó que publica os dados no tópico `/turtle1/cmd_vel`.

Contudo, o micro-ROS não comunica diretamente com o ROS2. Para isso, é necessário utilizar um `micro-ROS Agent`, que atua como intermediário entre o microcontrolador e a rede ROS2. Foi utilizada a `ESP32`, anteriormente mencionada, como dispositivo intermediário entre o *core* M4 (onde corre o micro-ROS, em cima do freeRTOS) e o ROS2. A `ESP32` corre o `micro-ROS Agent`, que recebe os dados do micro-ROS via UART e os transmite para a rede Wi-Fi, ligando-se ao ROS2 que corre num computador host.

A arquitetura de comunicação pode ser representada da seguinte forma:

Para garantir uma comunicação eficiente entre os dois núcleos (Cortex-M7 e Cortex-M4) da `STM32`, optou-se pela utilização do periférico IPCC (Inter-Processor Communication Controller). Este módulo permite a troca de sinais e mensagens entre os núcleos de forma rápida e sincronizada, sem risco de acesso concorrente à memória.

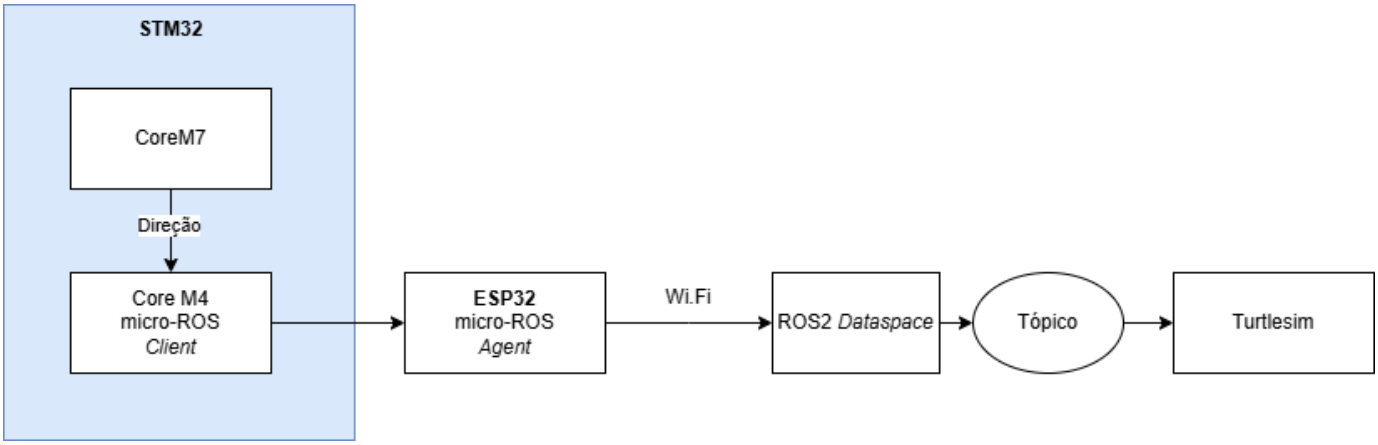

Fig. 29. Sistema com micro-ROS e Turtlesim

Neste projeto, o núcleo M7 recebe comandos de direção (por exemplo, da ESP32) e atualiza uma variável partilhada numa região comum de memória (SRAM4). Após atualizar a direção, o M7 utiliza o IPCC para notificar o núcleo M4 de que há uma nova direção disponível. O M4, que corre o micro-ROS, é ativado por esta interrupção e lê a nova direção, convertendo-a numa mensagem do tipo geometry$_msgs/msg/Twistqueéentãopublicadanotópico/turtle1/cmd_vel, controlandoassimomovimentodatartaruganosimulador.$

Os comandos enviados seguem a estrutura da mensagem `Twist`, para cada movimento respetivo:

- Frente: {`linear: {x: 0.0, y: 1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Trás: {`linear: {x: 0.0, y: -1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Direita: {`linear: {x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Esquerda: {`linear: {x: -1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Diagonal Frente-Esquerda: {`linear: {x: 1.0, y: 1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Diagonal Frente-Direita: {`linear: {x: 1.0, y: -1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Diagonal Trás-Esquerda: {`linear: {x: -1.0, y: 1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}
- Diagonal Trás-Direita: {`linear: {x: -1.0, y: -1.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}`}

Com base na direção recebida, a tartaruga move-se no plano, assumindo que está inicialmente orientada com a cabeça para cima (eixo positivo do eixo Y). No lado do ROS2, é possível implementar um nó em C++ que subscreve aos dados enviados pela ESP32. Este nó recebe os comandos publicados, processa-os e apresenta-os no terminal para verificação. Posteriormente, esta funcionalidade pode ser expandida para incluir ações mais complexas, como controlar outros dispositivos, realizar processamento adicional ou interagir com outros nós do sistema.

XI. Referências

Datasheet STM32H755, STMicroelectronics.
Reference Manual STM32H755, STMicroelectronics.
User Manual STM32H755, STMicroelectronics.
Application Notes para STM32H755, STMicroelectronics.
Datasheet ESP32 Series, Espressif.
Technical Reference Manual ESP32, Espressif.
Espressif Github: https://github.com/espressif.
Guia 2, PI2, LEEIC UM, 3º ano.
Datasheet L298, STMicroelectronics.
Datasheet PMODHYGRO, Digilent.
Datasheet PMODAQS, Digilent.
Datasheet OV7670, OmniVision.
Datasheet Display LCD com interface I2C, CYPRESS.

Kotlin and Android, Android Developers
μROS: https://micro.ros.org/
ROS2: https://docs.ros.org/en/humble/Tutorials.html