Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Eletrónica e Telecomunicações e Computadores

Relatório de Progresso do Projeto Final de Curso

2º Semestre 2024/2025

22 de abril de 2025

Professor Orientador do Projeto: Pedro Sampaio (pedro.sampaio@isel.pt)

Aluno: Diogo Miguel Fonseca Santos (a49936@alunos.isel.pt)

Índice

Lista de Figuras

Lista de Tabelas

Lista de acrónimos

GPIO - General Purpose Input Output

SPI – Serial Peripheral Interface

UART – Universal Asynchronous Receiver/Transmitter

I2C – Inter-Integrated Circuit

NMEA - National Marine Electronics Association

IDE - Integrated Development Environment

BTT - Bicicleta Todo-o-Terreno

FreeRTOS - Free Real Time Operative System

RTOS - Real Time Operative System

BT - Bluetooth

API – Application Programming Interface

SDK – Software Development Kit

IDF – IoT Development Framework

IMU – Inertial Measurement Unit

MSL – Mean Sea Level

DAC – Digital-to-Analog Converter

ADC – Analog-to-Digital Converter

CPU – Central Processing Unit

1. Introdução

Este relatório dará continuidade ao relatório Inicial que delineou a ideia deste projeto final de curso e apresentou o seu enquadramento e descrição, assim como a discussão de eventuais problemas durante a realização do mesmo. Sendo assim, este documento irá abordar o progresso que o projeto teve ao longo deste mês e meio desde o relatório Inicial, assim como uma apresentação mais concreta da arquitetura do projeto, metodologias usadas, a pesquisa feita e eventuais alterações à calendarização do projeto.

O objetivo do projeto em si não se alterou: desenvolver o *hardware* e *software* para um sistema simples e acessível de navegação para bicicletas Todo-o-Terreno (BTT), que consiste na monitorização de variáveis como localização geográfica, direção e velocidade. O sistema de navegação será baseado num microcontrolador já escolhido (ESP32), num giroscópio, num acelerómetro, num módulo de GPS para o processamento, medição de variáveis e apresentação dos dados e num *rotary encoder button* para navegação num menu para visualização das diferentes variáveis. Para além disso, este sistema irá armazenar as trajetórias percorridas de forma não volátil para consulta posterior.

2. Escolha e seleção do *hardware*

De acordo com a calendarização que foi feita no relatório de Introdução, existiu um período alocado para a pesquisa de protocolos e componentes de *hardware*, com o objetivo de escolher os componentes de hardware certos para o projeto tendo em conta os problemas apresentados no relatório anterior. Foram estudados protocolos de comunicação, possíveis microcontroladores, filtros para dados sem processamento, e módulos/componentes a serem usados no projeto final de curso.

Ao longo do estudo de diferentes microcontroladores, foram pré-selecionadas três opções consideradas adequadas para a concretização deste projeto:

Figure 1 - LPC1769

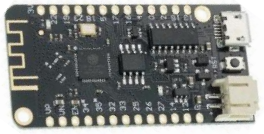

Figure 2 - ESP32

Figure 3 - STM32

Foram recolhidas informações sobre estes três microcontroladores para comparação de maneira a decidir qual a usar no projeto:

Critério	LPC1769	ESP32	STM32F4
Arquitetura	ARM Cortex-M3, 100 MHz	*Dual-core* Xtensa LX6, até 240 MHz (inclui 80MHz,160MHz e 240MHz)	ARM Cortex-M4 (até 180MHz)
Número de *cores*	1	2	1
Interface com sensores (SPI/I2C/UART)	Múltiplos canais	Múltiplos canais	Múltiplos canais
Comunicações sem fios (Wi-Fi/BT)	Não disponível nativamente	Integrado (Wi-Fi, BT, BLE)	Não disponível nativamente
Disponibilidade de bibliotecas	Boa, mas menos atualizadas	Com drivers prontos para sensores (Muito boa)	Tem HAL, LL, CMSIS
Multithreading	Sem RTOS integrado	Suporte nativo a FreeRTOS e *dual-core,* logo permite dividir tarefas	FreeRTOS integrado via CubeMX, com *overhead* por ser *single core*

Tabela 1 - Comparação de 3 microcontroladores

Com base nos critérios da tabela acima, foi decidido que se iria utilizar o **ESP32** para o projeto:

- O LPC1769 já é um microcontrolador conhecido do aluno por ter feito a unidade curricular de Sistemas Embebidos, e o aluno queria explorar um novo microcontrolador. Para além disso, este microcontrolador é *single-core* e comunicações sem fios não estão disponíveis nativamente. Por último, este microprocessador não tem um RTOS integrado.
- O STM32F4 é um microcontrolador que, apesar de ter FreeRTOS integrado, tem overhead típico de sistemas *single-core* e comunicações sem fios não estão disponíveis nativamente nele.
- Para além de ser ideal para *multithreading* e para sistemas embebidos, o ESP32 tem comunicações sem fios integradas, é dual-core (permite dividir tarefas – um core para sensores e outro para *display,* por exemplo) e tem suporte nativo a FreeRTOS.

Foram também estudados componentes possíveis para uso no projeto como sensores, módulos de GPS e *displays*.

Módulo Sensor	Acelerómetro	Giroscópio	Magnetómetro	Interface	Consumo de Energia
MPU9250	3 eixos ±2/4/8/16g	3 eixos ±250/500/1000/2000°/s	3 eixos AK8963 ±4800µT	I2C / SPI	3.7 mA
LSM9DS1	3 eixos ±2/4/8/16g	3 eixos ±245/500/2000°/s	3 eixos ±4/±8/±12/±16 gauss	I2C / SPI	4.3mA
BNO055	3 eixos ±2/4/8/16g	3 eixos ±125/250/500°/s	3 eixos ±1300µT (x-, y-*axis*) ±2500µT (z-*axis*)	I2C	12.3mA

Tabela 2 - Comparação de 3 módulos de sensores

Com base nos datasheets de cada sensor, foi construída a tabela acima para comparação entre eles, com o objetivo de selecionar um para utilizar no projeto. O critério usado aqui entre os três sensores foi o alcance do magnetómetro (permite saber a orientação absoluta), pelo que o componente escolhido foi o **MPU9250**.

Módulo GPS	Precisão	Taxa de atualização	Sensibilidade	Consumo de Energia
NEO-6M	*Horizontal position accuracy* GPS = 2.5 m	5 Hz	-161 dBm	39mA
NEO-M8N	*Velocity accuracy* = 0.05 m/s *Heading accuracy* = 0.3 degrees *Horizontal position accuracy* GPS = 2.5 m	Até 10 Hz	–166 dBm	27mA
Quectel L80	*Velocity accuracy* < 0.1 m/s *Acceleration Accuracy* = 0.1 m/s^2 *Horizontal position accuracy* < 2.5 m CEP	1Hz (*Default*), até 10Hz	-165dBm	20mA

Tabela 3 - Comparação de 3 módulos de GPS

Com base nos datasheets de cada módulo de GPS, foi construída a tabela acima para comparação entre eles, com o objetivo de selecionar um para utilizar no projeto. Uma vez que uma taxa de atualização de 5Hz já é bastante aceitável no contexto do projeto e que a precisão e sensibilidade não difere muito entre os módulos, optou-se pelo módulo com menor custo – o **NEO-6M**. Para a escolha do *ePaper display*, foi decidido que não seria necessário ter um *refresh rate* muito grande, dado que este *display* mostrará dados estáticos como velocidade, setas de direção, e a data do dia. Por isso, um *display ePaper* com pouco custo chega para o projeto, mesmo que com uma responsividade lenta. Optou-se assim por escolher o **WeAct Studio *Epaper* 2.9'' *module*** (aproximadamente 1 segundo de *full refresh*).

3. Arquitetura do Sistema

A arquitetura do sistema desenvolvido neste projeto baseia-se na utilização do microcontrolador ESP32 (LOLIN32 Lite), com suporte ao sistema operativo em tempo real *FreeRTOS*, que facilita a escalabilidade, organização e manutenção do código, para além de atribuir uma vertente *threadsafe* á aplicação. O projeto encontra-se dividido em diferentes camadas (*HAL, DRIVERS*, *HAL_FreeRTOS*, *DRIVERS_FreeRTOS* e *APP*) e módulos (tarefas independentes e máquinas de estado finitas, que comunicam entre si através de *queues*).

No desenvolvimento deste projeto, foi adotada a abordagem de programação *baremetal*, ou seja, sem recorrer às APIs de alto nível fornecidas pelo SDK oficial da Espressif (ESP-IDF). Esta decisão foi tomada com dois objetivos principais:

1. Ao programar em *baremetal*, evita-se as abstrações das bibliotecas prontas e assim percebe-se com mais detalhe como o microcontrolador opera;
2. Em Sistemas Embebidos, o aluno trabalhou com um microcontrolador também em *baremetal* e por isso achou que fazia sentido aplicar a mesma metodologia neste projeto final e manter uma certa consistência com o que já tinha feito antes, tendo para além disso mais controlo sobre o que estava a desenvolver.

Cada periférico é gerido por uma tarefa dedicada que está encarregue de comunicar com o respetivo *hardware*, processar os dados e enviar eventos para a lógica principal da aplicação. Esta é estruturada em máquinas de estado, que decidem o comportamento global do sistema consoante os dados recebidos da IMU (*Inertial Measurement Unit*), do GPS e do *rotary encoder button.*

Em termos de tarefas, a ideia é a seguinte:

- A **tarefa principal** é aquela que contém a inicialização das tarefas e todas as máquinas de estado que forem necessárias para a aplicação do sistema;
- A **tarefa do *Rotary Encoder Button*** é a tarefa que faz *polling* dos sinais dos GPIOs (*General Purpose Input Output*) e os interpreta como rotação para a esquerda/direita ou um *click* do botão. Fará sentido ter esta tarefa dado que ela permite a navegação entre os diferentes modos de operação do sistema.
- A **tarefa do MPU9250/6050** é a tarefa que adquire e faz o tratamento preliminar dos dados provenientes do módulo MPU9250/6050. Esta irá fornecer ao sistema dados sobre a aceleração e rotação angular que serão usados posteriormente na fusão sensorial com o GPS.
- A **tarefa do GPS** é a tarefa que adquire e processa os dados de posição geográfica, velocidade e direção fornecidos pelo módulo GPS NEO-6M. Esta irá fornecer ao sistema dados como a latitude, longitude, *heading* estimado, *timestamp* da *sample* e velocidade.
- A **tarefa do *display*** é a tarefa que integra o *display* e-Paper no sistema e que garante a comunicação intuitiva entre o sistema e o ciclista. Esta tarefa irá conter o desenvolvimento da interface do *display* de modo que esta comunicação entre sistema e utilizador exista.
- A **tarefa de *logging*** é a tarefa que recebe do *Logger Timer* um valor que lê de uma variável global (a tarefa principal coloca lá o que deve de ser registado na *flash* e o *software timer* vai buscar repetidamente o valor com um período definido na tarefa principal quando o *timer* foi criado) e regista na flash os dados recebidos. Para além disso, também tem como função enviar as *logs* guardadas na flash para a tarefa principal, para serem disponibilizadas no *display*.

Na figura seguinte está ilustrada a descrição anterior da arquitetura do sistema que se pretende ter como projeto final de curso. Vale a pena notar que existe na figura um módulo "*Sensorial Fusion*" que terá código relativo a um filtro que será aplicado aos dados provenientes do módulo GPS com os dados anteriormente tratados provenientes do módulo MPU9250/6500 que será usado na tarefa principal para fazer esta segunda filtragem.

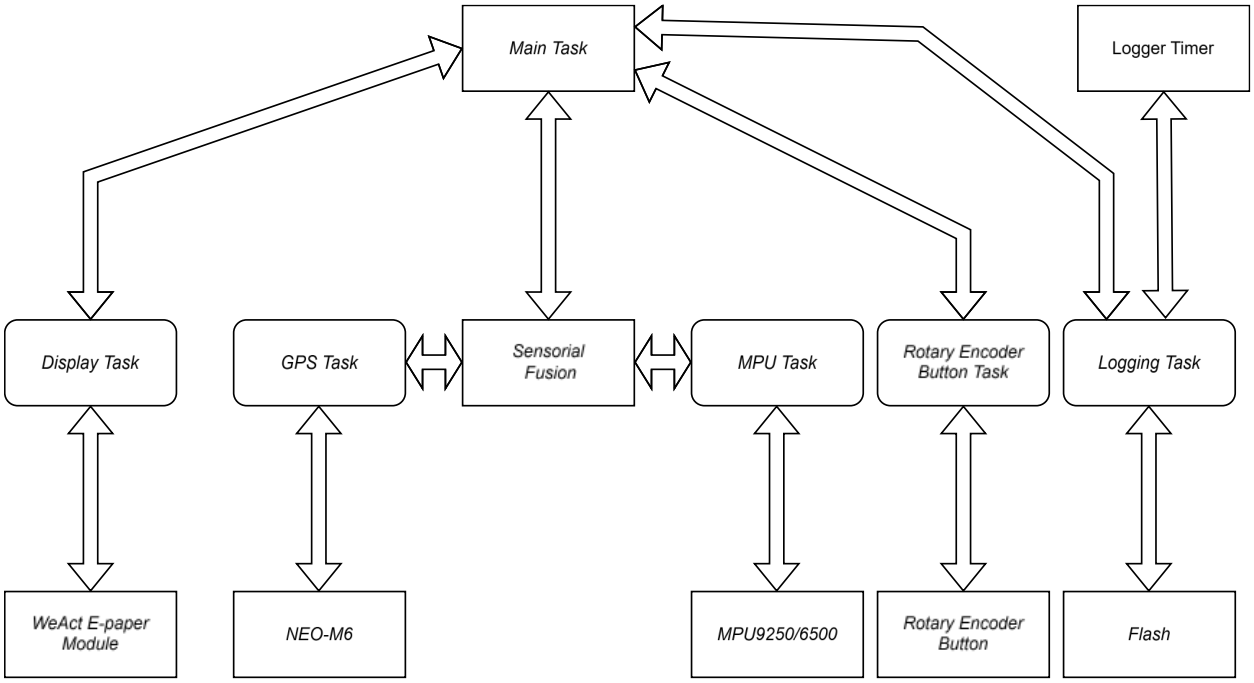

Figure 4 - Arquitetura da componente local do Sistema

O microcontrolador comunicará com os vários componentes através de dois protocolos:

- O módulo de GPS entrará em comunicação através do protocolo UART;
- O módulo do *Epaper display* entrará em comunicação através do protocolo SPI;
- Para a comunicação entre o microcontrolador e o módulo MPU9250/6500, é possível utilizar tanto o protocolo I2C como o protocolo SPI. Apesar da familiaridade prévia com o protocolo I2C, optou-se pela utilização do protocolo SPI neste projeto. Esta decisão teve como motivação principal a oportunidade de explorar e aprender um protocolo de comunicação digital diferente. Para além disso, o SPI permite atingir taxas de transferência superiores às do I2C ("A comunicação com todos os registos do dispositivo é realizada utilizando I2C a 400 kHz ou SPI a 1 MHz." [1]), proporciona uma comunicação *full-duplex* e tem uma maior imunidade ao ruído. Sendo este protocolo mais rápido, haverá uma menor latência na receção dos dados do sensor.
- No projeto, a comunicação entre o *rotary encoder button* e o ESP32 será feita através dos pinos digitais (GPIO) do microcontrolador (fazendo *polling* através dos *Timer Groups* do ESP32).

Estas tarefas comunicarão com a tarefa principal através de *message queues* de estruturas que serão depois recolhidas pela própria. Foi adotada esta estrutura modular baseada em tarefas para uma maior clareza e organização no desenvolvimento do projeto, assim como para ser mais fácil atualizar ou corrigir código de componentes do projeto sem afetar o restante do sistema.

[1] Frase tirada do *datasheet* do MPU9250

É de notar que esta arquitetura do sistema não é a arquitetura final, mas sim a componente local do projeto. A componente remota será realizada no futuro próximo, assim que a parte local estiver totalmente concretizada.

Existirão múltiplos modos de operação do sistema:

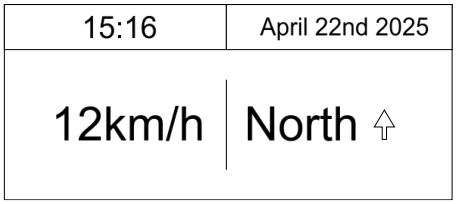

Figure 5 - Modo de Navegação

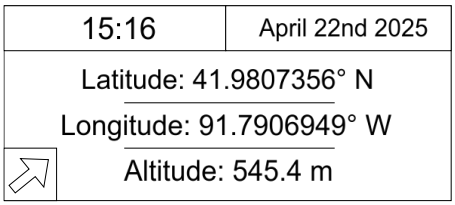

Figure 6 - Modo de Localização

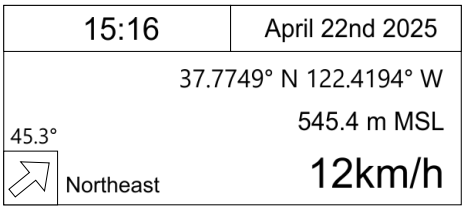

Figure 7 - Modo Informativo Completo

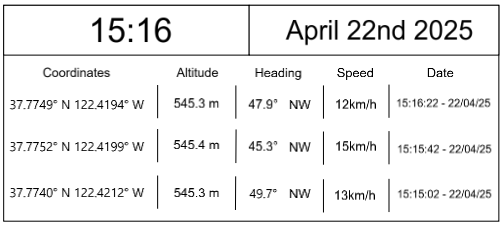

Figure 8 - Modo de Logging

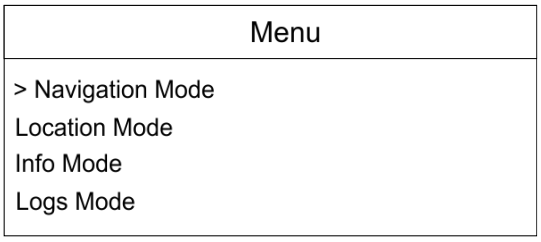

Figure 9 -Modo do Menu

Estes modos de operação estarão implementados em máquinas de estado. Para aceder a cada um dos modos, o utilizador utilizará o *rotary encoder button* para navegar pelo menu e achar o modo que quer selecionar.

4. Conhecimentos Adquiridos e Pesquisa Realizada

Este capítulo apresenta os principais conhecimentos teóricos e técnicos adquiridos até à data, que têm sido fundamentais para o avanço do Projeto Final de Curso. A teoria abordada aqui tem servido de base para a implementação prática dos módulos atualmente em desenvolvimento.

4.1. Protocolo *Serial Peripheral Interface*

O SPI (*Serial Peripheral Interface*) é um protocolo de comunicação amplamente utilizado entre microcontroladores e periféricos, como sensores, conversores analógico-digitais (ADC) e conversores digital-analógicos (DAC). Trata-se de um protocolo de comunicação síncrono, o que significa que todos os dispositivos envolvidos na comunicação partilham o mesmo sinal de relógio (*clock*).

Este protocolo é muitas vezes escolhido por permitir velocidades de comunicação bastante elevadas, compatíveis com frequências de relógio de CPU também altas. Além disso, o SPI é um protocolo de comunicação *full-duplex*, o que significa que é possível transmitir e receber dados em simultâneo. Este protocolo utiliza uma configuração do tipo *master-slave*, algo com que o aluno já está familiarizado. Nesta configuração, é possível ter múltiplos escravos ligados a um único *master*.

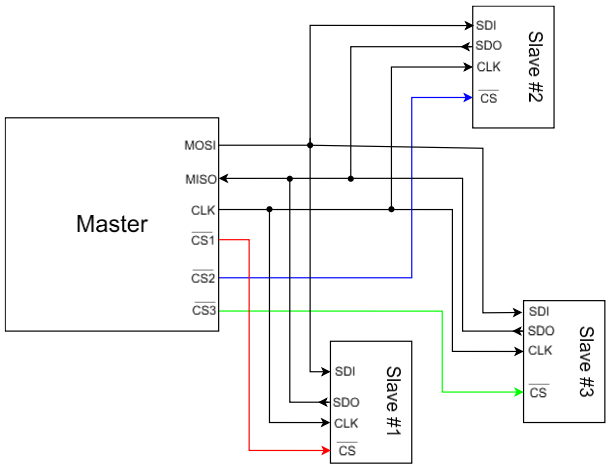

Figure 10 - Configuração master-slave

Do lado do *master*, os pinos utilizados são:

- MOSI (*Master Out Slave In*) – envia dados do *master* para o *slave*.
- MISO (*Master In Slave Out*) – envia dados do *slave* para o *master*;
- SCK ou CLK – fornece o sinal de relógio/*clock*;
- CS (*Chip Select*) – seleciona o *slave* com o qual o *master* quer comunicar;

A linha MOSI transporta os dados enviados pelo *master*, enquanto a linha MISO transporta os dados enviados pelo *slave*. O sinal de *clock* é fundamental para a sincronização da comunicação: o *master* gera este sinal e os *slaves* respondem de acordo com os seus ciclos. A linha CS (*Chip Select*) é geralmente ativa a nível lógico baixo. Ou seja, quando o *master* quer comunicar com um determinado *slave*, coloca a linha CS correspondente a 0 (GND), sinalizando que pretende iniciar a comunicação. Cada *slave* tem normalmente uma linha CS dedicada, o que pode ser visto como uma das limitações do SPI: o número de pinos digitais GPIOs disponíveis condiciona diretamente o número máximo de *slaves*.

Ao contrário do que acontece com protocolos como UART ou I2C, o SPI não utiliza um *bit frame* predefinido para a comunicação, o que simplifica o processo de envio e receção. O SPI possui quatro modos de operação, que permitem ajustar o comportamento do protocolo em função da polaridade (CPOL) e da fase (CPHA) do sinal de *clock*, como se pode ver na tabela seguinte.

	Polaridade do sinal de *clock*	Fase do sinal de *clock*	Estado inicial do sinal de *clock*	*Edge* ativa para envio de dados	*Edge* ativa para envio de dados
1ºModo	0	0	*Low*	*Rising edge*	*Falling edge*
2ºModo	0	1	*Low*	*Falling edge*	*Rising edge*
3ºModo	1	0	*High*	*Falling edge*	*Rising edge*
4ºModo	1	1	*High*	*Rising edge*	*Falling edge*

Tabela 4 - Modos de operação do SPI

No primeiro modo, não há *delay* para os impulsos de relógio (começa no estado baixo) e os dados são enviados imediatamente na transição ascendente do sinal SPI CLK – os dados de entrada são bloqueados na transição descendente do relógio.

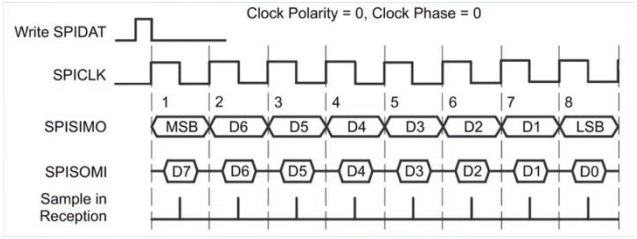

Figure 11 - Primeiro Modo de operação do SPI

No segundo modo, o SPI CLK inicia após algum *delay* (começa no nível lógico *low*) e os dados são enviados meio ciclo antes da primeira transição ascendente do SPI CLK. Os dados de entrada são captados na transição ascendente do SPI CLK e posteriormente enviados para as transições descendentes subsequentes do SPI CLK.

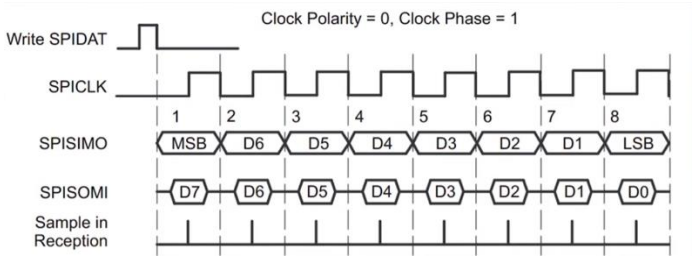

Figure 12 - Segundo Modo de operação do SPI

No terceiro modo, não há *delay* para os impulsos de *clock*, mas começa no estado lógico *high* e os dados são enviados imediatamente na transição descendente do sinal SPI CLK – os dados de entrada são bloqueados na transição ascendente do relógio.

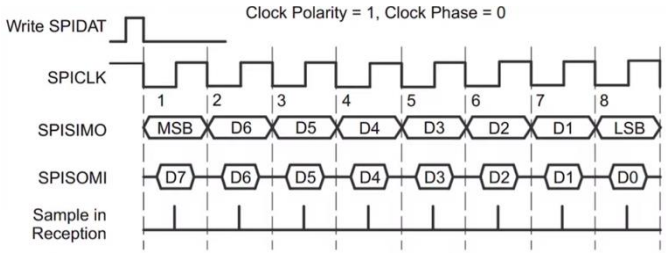

Figure 13 - Terceiro Modo de operação do SPI

No quarto modo, o SPI CLK inicia após algum *delay* (inicia no estado lógico *high*) e os dados são enviados meio ciclo antes da primeira transição descendente do SPI CLK – os dados de entrada são bloqueados na transição descendente do SPI CLK, mas depois enviados nas transições ascendentes subsequentes do SPI CLK.

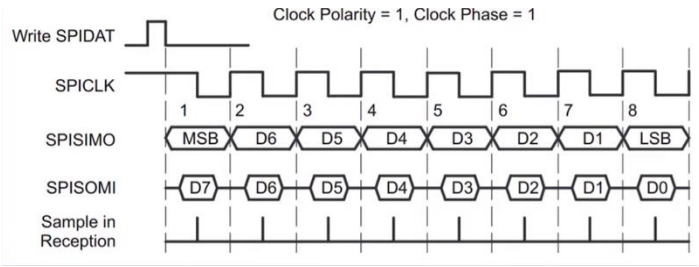

Figure 14 - Quarto Modo de operação do SPI

Esta flexibilidade permite adaptar a comunicação SPI aos requisitos específicos de cada dispositivo com que se pretende comunicar.

Ao observar a estrutura interna dos dispositivos mestre e escravo numa comunicação SPI, é possível identificar que ambos possuem *shift registers*. Estes registos estão ligados de tal forma que formam um *buffer* circular entre dispositivos - isto é, a comunicação ocorre de forma contínua entre os dois.

Os *shift registers* operam no modo SISO (*Serial In Serial Out*), o que significa que:

- A saída do *shift register* do *master* está ligada à entrada do registo do *slave* — esta ligação é a linha MOSI (*Master Out Slave In*).
- A saída do *shift register* do *slave* está ligada à entrada do registo do *master* — esta ligação é a linha MISO (*Master In Slave Out*).

A comunicação SPI é síncrona, o que quer dizer que existe um sinal de relógio (clock) partilhado, que é sempre gerado pelo *master*. Este sinal de *clock* é o que garante a sincronização dos dados entre ambos os dispositivos.

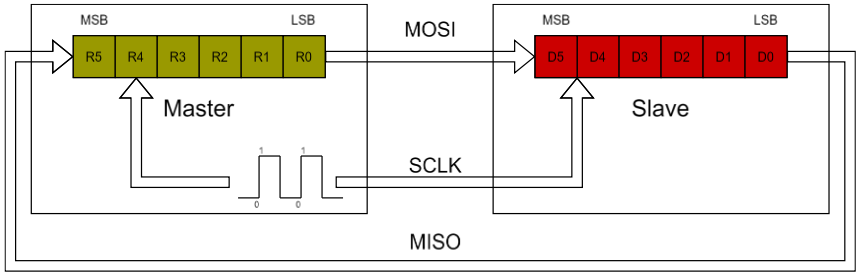

Figure 15 - Estrutura interna dos registos na configuração master-slave

A comunicação SPI inicia-se quando o *master* gera um pulso de *clock*. Assim que o pulso começa, os registos de deslocamento dos dois dispositivos deslocam o primeiro bit (normalmente o bit menos significativo – LSB) para a direita:

- O *master* desloca o seu bit R0;
- O *slave* desloca o seu bit D0;

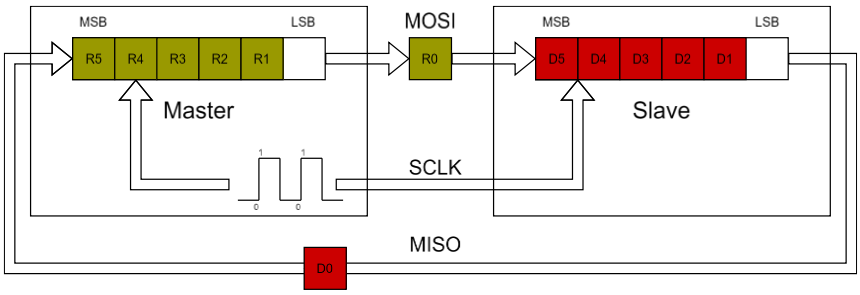

Figure 16 - Shift de bits de um registo para outro

Como os dispositivos estão ligados em anel (via MOSI e MISO), o bit LSB que "sai" do *master* (R0) entra na posição MSB (bit mais significativo) do registo do *slave*. O mesmo acontece com o bit do *slave* (D0), que entra na posição MSB do registo do *master*.

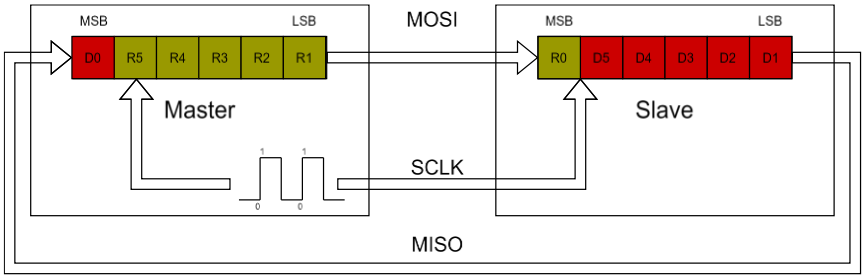

Figure 17 - Bits deslocados para o outro registo

Este processo repete-se a cada ciclo de *clock*, permitindo que os bits sejam transferidos simultaneamente em ambas as direções — é este o princípio do funcionamento *full-duplex* do protocolo SPI.

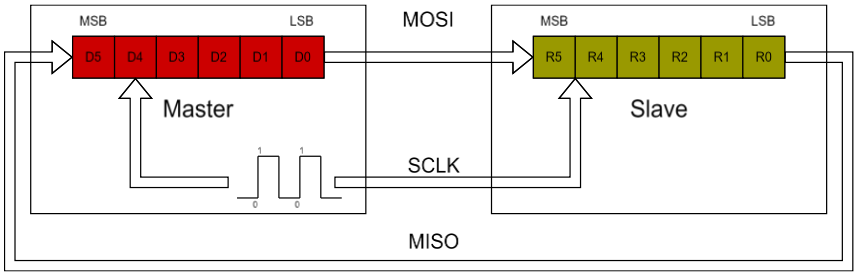

Figure 18 -Bits transferidos na sua totalidade

4.2. Protocolo UART

O protocolo UART (*Universal Asynchronous Reciever/Transmitter*) é um protocolo de comunicação serial e assíncrona, e é ideal para comunicações simples. Assíncrono significa que os dispositivos não partilham um sinal de *clock* – a sincronização é feita com ase na configuração do baud rate, que tem de ser igual nos dois dispositivos. Este protocolo opera segundo uma topologia *ad hoc*, também conhecida como *peer-to-peer*. Isto significa que a comunicação é feita diretamente entre dois dispositivos (não existem conceitos de *master* ou *slave* neste tipo de topologia como no SPI), sem necessidade de um barramento partilhado ou controlador central. Cada ligação UART é exclusiva entre um emissor e um recetor e nela existem pelo menos duas linhas: uma para transmissão e outra para receção. O pino de transmissão de um dispositivo é ligado ao pino de receção do outro e vice-versa

(a ligação ao GND de ambos os dispositivos precisa de ser comum). Este é um protocolo *character oriented*, ou seja, transmite os dados um *byte* (caractere) de cada vez.

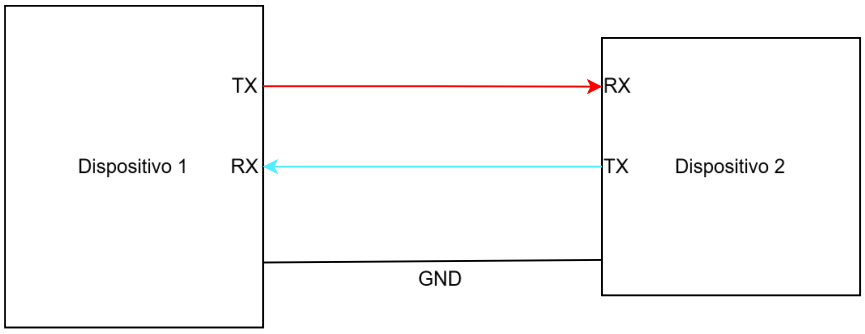

Figure 19 - Configuração UART

Quando se fala de um protocolo de comunicação, tem-se em conta o facto de que cada tipo tem o seu próprio *bit frame* único de comunicação. Neste caso em concreto, o *bit frame* do protocolo UART é a sequência de *bits* que encapsula cada caractere/*byte* transmitido, incluindo bits de controlo que indicam ao recetor o início e o fim desse caractere.

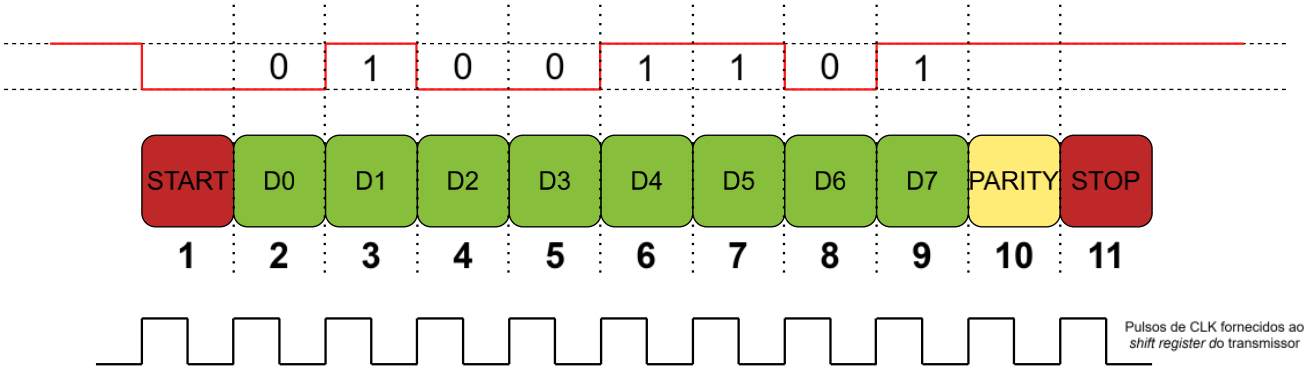

Figure 20 - Exemplo de um frame UART

No protocolo UART, a transmissão de um carácter é realizada através de um *frame* composto por vários *bits*. Este *frame* inicia-se com um *bit* de início (*start bit*), que coloca a linha de transmissão no estado lógico *low*, sinalizando ao recetor o início da comunicação. De seguida, são transmitidos oito *bits* de dados, correspondentes ao carácter que se pretende enviar. Opcionalmente, pode ser incluído um *bit* de paridade, utilizado para a deteção simples de erros. O frame termina com um ou mais *bits* de paragem (*stop bits*), que colocam novamente a linha no estado lógico *high*, indicando o fim da transmissão desse carácter. A utilização de mais do que um *bit* de paragem permite acomodar dispositivos mais lentos, garantindo tempo adicional antes do envio de um novo *frame*.

Inicialmente, tanto a linha de transmissão como a de receção mantêm-se no estado *high*, indicando que a linha está inativa (*idle*). Quando o transmissor pretende iniciar a comunicação, força a linha para o estado *low* durante um pulso de *clock*, sinalizando o início do *frame*. Após o envio do *byte* de dados, seguido, se aplicável, pelo bit de paridade, são enviados os *bits* de paragem, retornando a linha ao estado *high* durante o(s) respetivo(s) pulso(s) de *clock*.

Desta forma, um *frame* UART é constituído, no mínimo, por 10 *bits* (1 *bit* de início, 8 *bits* de dados e 1 *bit* de paragem) ou 11 *bits* quando inclui um *bit* de paridade.

Existem dois métodos que se usam para programar o protocolo UART: *bit banging* e o recurso a periféricos UART integrados (presentes na maioria dos microcontroladores atuais).

O método de *bit banging* consiste em implementar, por software, o controlo direto dos pinos de transmissão e receção, gerindo manualmente os tempos e os estados lógicos de cada bit transmitido e recebido. Este método exige um controlo rigoroso dos tempos, sendo altamente dependente da velocidade de execução do microcontrolador e da precisão das rotinas de temporização implementadas. Apesar de permitir alguma flexibilidade, o *bit banging* consome muitos recursos do processador, já que este tem de estar continuamente dedicado à gestão da comunicação, o que o torna pouco eficiente para aplicações mais exigentes.

O outro método é o uso de periféricos UART integrados nos microcontroladores. Estes periféricos são módulos dedicados que gerem, de forma autónoma, toda a comunicação UART, incluindo a geração dos bits de início e de paragem, o envio e receção dos bits de dados, e, quando aplicável, a verificação da paridade.

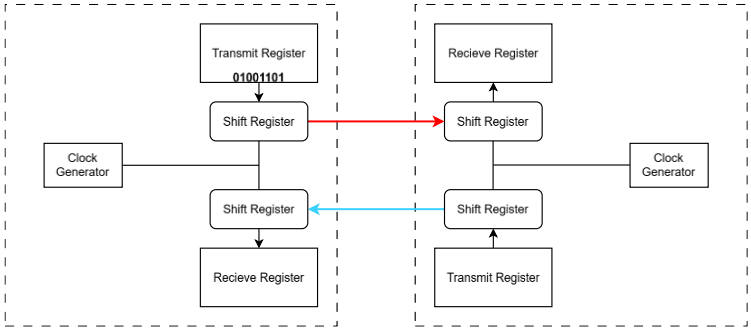

Figure 21- Diagrama de Comunicação UART com Registos de Transmissão e Receção

Em cada dispositivo existem blocos específicos, nomeadamente um *transmit register*, ligado a um *shift register*, e um *receive register*, ligado a outro *shift register*. Ambos estes shift registers estão sincronizados com o sinal de *clock* de cada dispositivo. Estes mesmos blocos estão presentes em ambos os dispositivos de comunicação. O *transmit register* envia o conjunto completo de dados para o *shift register* de uma só vez, e este recebe os dados em paralelo e envia-os, *bit* a bit, para o dispositivo recetor - processo designado por *Parellel In Serial Out*. Do lado do recetor, ocorre o processo inverso. O *shift register* recebe todos os *bits* sequencialmente, um a um e, após receber o conjunto completo, envia todos esses dados em simultâneo para o *receive register* - operação conhecida como *Serial In Parallel Out*.

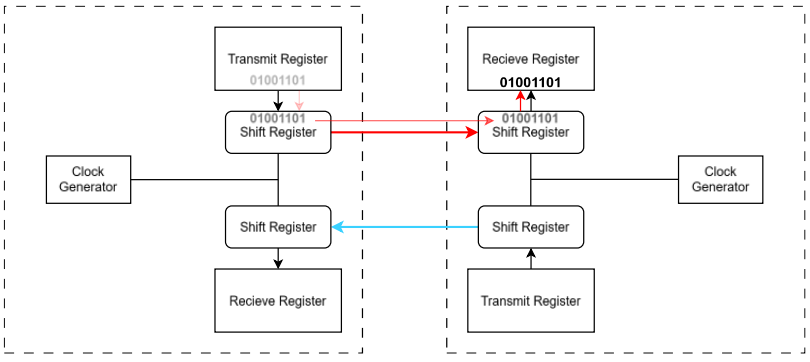

Figure 22 - Fluxo de dados na comunicação UART

Quando o dispositivo transmissor pretende enviar um *byte* de dados para o dispositivo recetor, esse *byte* é primeiramente carregado para o *transmit register*. Em seguida, é transferido para o *shift register*, onde os *bits* são organizados nas posições D0, D1, D2, e assim sucessivamente. O *shift register* desloca então cada *bit*, um de cada vez, sincronizando com o pulso de *clock*, transmitindo-os pela linha de comunicação para o dispositivo recetor.

No dispositivo recetor, o processo é semelhante, mas no sentido inverso. Os *bits* chegam sequencialmente ao *shift register* do recetor, que os desloca de forma síncrona com o *clock*, até que o *byte* completo seja reconstruído. De seguida, o *byte* é transferido para o *receive register*, ficando disponível para posterior utilização. O sinal de *clock* desempenha um papel fundamental neste processo, garantindo que a *baud rate* (taxa de transmissão) está devidamente sincronizada entre ambos os dispositivos. A cada pulso de *clock*, as linhas de transmissão e de receção trocam dados de forma coordenada, assegurando a integridade da comunicação.

A principal vantagem deste método é a liberdade que o processador ganha. O processador apenas necessita de configurar o periférico UART e gerir a troca de dados através de interrupções ou da leitura e escrita de registos específicos, sem ter de controlar manualmente os tempos ou os estados das linhas de comunicação. Esta abordagem permite uma comunicação mais fiável, especialmente em sistemas que exigem multitarefa.

Para configurar a comunicação UART com periféricos, seguem-se as seguintes etapas:

1. Configurar o sinal de *clock* do periférico UART, ajustando-o de acordo com a *baud rate* pretendida.
2. Carregar os dados no *transmit register*.
3. Ativar o *timer* responsável pela deslocação dos dados, permitindo que o *shift register* do dispositivo transmissor envie os *bits* sequencialmente pela linha de transmissão. O dispositivo recetor irá receber os dados de acordo com a sua taxa de amostragem (*sampling rate*).

Após iniciada a transmissão, é necessário monitorizar o estado da comunicação, o que pode ser feito de duas formas:

- Método de *Looping* (*Polling*): Durante o envio, o processador verifica ciclicamente o *bit* de *status* que indica se os 8 *bits* foram enviados. No recetor, procede-se de igual forma, verificando se os dados foram corretamente recebidos.
- Método com Interrupções: Quando os dados são transmitidos ou recebidos com sucesso, o periférico UART gera automaticamente uma interrupção. Esta interrupção é tratada por uma rotina de atendimento (*interrupt service routine* — ISR), que gere o evento sem necessidade de monitorização contínua. Este método é mais eficiente e adequado para sistemas com multitarefa, dado que liberta o processador para outras operações.

4.3. *Frames* NMEA

O NMEA é um protocolo padrão de comunicação definido pela National Marine Electronics Association. Foi criado para permitir a interoperabilidade entre equipamentos de navegação (GPS, bússolas digitais, sonares, etc.).

Este protocolo é baseado em frases ASCII, em que cada frase começa com um '$', seguido de um identificador de 5 caracteres, com os dados separados por virgulas juntos de seguida. Cada frase pode ter um *checksum* opcional (ser precedida por um '*'), terminando com '<CR><LF>'.

Existem muitas frases neste protocolo, mas as mais comuns e que tem algum interesse são:

Tipo	Nome	Dados da frase
GPGGA	*Global Positioning System Fix Data*	Data em UTC, Latitude, Longitude, Indicador de Qualidade do GPS, Nº de satélites, *Horizontal delution of position*, Altitude
GPRMC	*Recommended Minimum Navigation Information*	Data em UTC, Status, Latitude, N/S, Longitude, E/W, *Speed over ground*, Variação Magnética, *Track made good*
GPVTG	*Track made good and Ground speed*	*Course over ground, Speed over ground*
GPGLL	*Geographic Position – Latitude/Longitude*	Latitude, N/S, Longitude, E/W, Data em UTC, *Status*
GPGSA	*GPS DOP and active satellites*	Modo do *fix*, Ids dos satélites

Tabela 5 – Frases NMEA mais comuns e com algum interesse

4.4. Filtros de fusão sensorial

Quando se utilizam sensores inerciais, como acelerómetros e giroscópios, é comum surgirem problemas como ruído, deriva e leituras instáveis. Para melhorar a fiabilidade das medições, recorre-se a filtros de fusão sensorial.

O **filtro complementar** é uma solução simples para a fusão de dados. Ele assume que os erros dos sensores são complementares. Como exemplo:

- O acelerómetro fornece uma boa estimativa de inclinação em baixa frequência, mas é sensível a vibrações e acelerações falsas;
- O giroscópio fornece boa estimativa em alta frequência, mas sofre de deriva ao longo do tempo.

A ideia do filtro complementar é combinar os dois sinais com diferentes pesos:

$$\hat{a}nguloEstimado = \alpha \times (\hat{a}nguloEstimado + velocidadeAnguGyro \times \Delta t) + (1 - \alpha) \times \hat{a}nguloAcel$$

Onde α é um fator de ponderação entre 0 e 1. Um valor típico é $\alpha \approx 0.98$, o que dá maior peso ao giroscópio (curto prazo) e corrige gradualmente com o acelerómetro (longo prazo).

O **filtro de Kalman** é uma técnica mais avançada e baseia-se num modelo matemático do sistema e dos sensores, considerando:

- O estado do sistema, como por exemplo, a orientação e a velocidade;
- O modelo de evolução temporal do estado;
- Os erros estatísticos das medições, como a variância e o ruído;

Este filtro atualiza a estimativa do estado com base numa previsão (modelo do sistema) e numa correção (medição do sensor), atribuindo pesos automaticamente conforme a confiabilidade dos dados em cada instante. Este filtro é mais complexo, exigente e requer maior capacidade de processamento, apesar de ser mais preciso do que o filtro complementar.

5. Progresso do Projeto

Neste capítulo é demonstrado o progresso do projeto até á data e comparado o que se fez com aquilo que estava planeado ser feito até á data de entrega do relatório de Progresso.

O aluno começou por estudar o hardware a ser utilizado e o IDE que utiliza para programar. De seguida, ele começou a testar código simples com o microcontrolador ESP32. O aluno fez o LED interno do ESP32 piscar como teste inicial e depois resolveu criar código simples com a framework da Espressif para fazer um LED

externo piscar através de um pino GPIO. Por último, para teste do FreeRTOS nativamente integrado na *framework*, o aluno criou código para fazer duas tarefas diferentes ligarem ou desligarem o LED. Inicialmente, o aluno compilou o código e correu o código criado através do *serial terminal* dado pela Espressif (programa ESP-IDF *terminal*), mas depois migrou para o Espressif IDE para entender como fazer o setup do próprio. Questões como o *debugging* foram exploradas, mas com o IDE em questão não havia muito a fazer.

Exisitam 3 métodos para fazer debug disponíveis: GDBStub Debugging, Core Dump Debugging (ambos post-mortem) e OpenOCD Debugging.

Após estar mais confortável com o ambiente de trabalho, o aluno pesquisou cobre como faria o *layering* do projeto, descobrindo que no IDE da *Expressif* a maneira de o fazer é criar ou importar componentes. Em relação a componentes importados, consegue-se importar componentes tanto da API deles como do repositório de componentes da *framework* deles. Componentes são para ser pensados como bibliotecas com comandos de *wrapper* de CMake. Após investigar múltiplos componentes *open-source* online e do repositório da Espressif, o aluno percebeu que havia liberdade na maneira como os componentes podiam ser construídos e que sendo assim, iria planear a estrutura do projeto com base no que aprendeu na unidade curricular de Sistemas Embebidos. Como o LOLIN32 Lite (microcontrolador ESP32 que o aluno utilizou) também não suporta depuração JTAG via USB, a depuração OpenOCD estava fora de questão, pois precisaria de um adaptador externo ligado a determinados pinos da placa. Como resultado, o aluno decidiu que se fizesse debugging, utilizaria a depuração básica do GDBStub.

O aluno investigou maneiras de mudar a frequência do CPU e descobriu que acessando um ficheiro sdkconfig no próprio IDE ou indo ao terminal ESP-IDF e executando o comando idf.py menuconfig, conseguia ter acesso a configurações do microcontrolador. Acessando a opção Component Config – ESP System Settings – CPU frequency, o aluno conseguiu mudar a frequência do CPU.

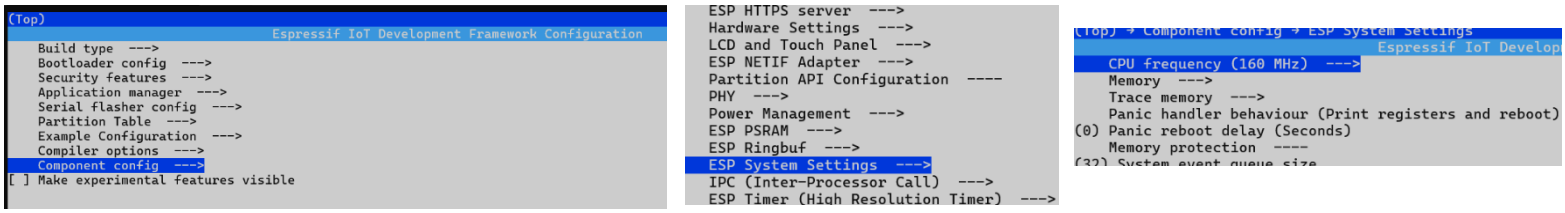

Figure 23 -Passos para mudar a frequência do CPU

Á medida que o aluno começou a ler o manual técnico de referência do ESP32, ele começou a construir a estrutura do projeto, começando com a parte local do sistema. O aluno começou por mapear em estruturas os registos relacionados com a GPIO Matrix e com o IO Multiplexer em *header files* criados numa pasta dentro da pasta *components* chamada MyHWMAP.

Depois o aluno começou a criar a HAL na pasta MyHAL e os DRIVERS na pasta MyDRIVERS, com o objetivo de construir uma base para GPIO estável, criando o os seguintes ficheiros:

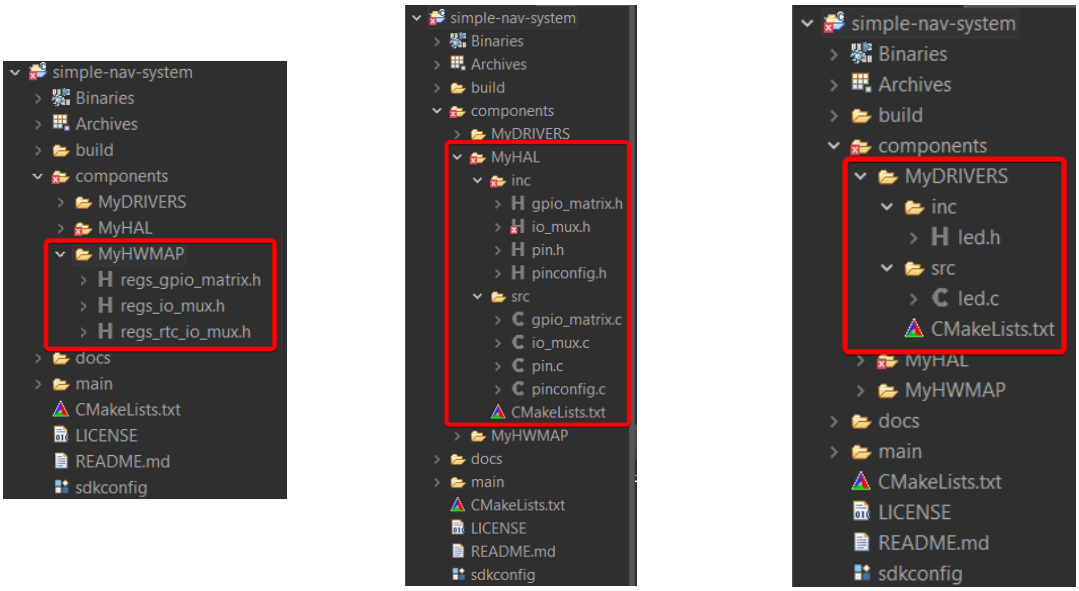

Figure 24 - Estrutura atual do sistema

A descrição dos ficheiros é a seguinte:

Pasta components/MyHAL

- **gpio_matrix.h**
  Define funções e enums para configurar e manipular o GPIO Matrix, incluindo seleção de sinais de entrada/saída, configuração de interrupções e modos de operação.

- **gpio_matrix.c**
  Implementa as funções declaradas em gpio_matrix.h, permitindo configurar sinais de entrada/saída, inversão de sinais e controlo de seleção de periféricos.

- **io_mux.h**
  Declara funções e enums para configurar o IO_MUX, incluindo seleção de funções, força de drive, resistências pull-up/pull-down e modos de sono.

- **io_mux.c**
  Implementa as funções de configuração do IO_MUX, como seleção de funções, configuração de força de drive e modos de sono para pinos.

- **pin.h**
  Declara funções e enums para manipulação de pinos GPIO, como definir direção, ativar/desativar saídas e inicializar pinos.

- **pin.c**
  Implementa as funções de manipulação de pinos GPIO, incluindo validação de IDs de pinos e configuração de resistências e funções.

- **pinconfig.h**
  Declara funções para configurar modos de resistências, entrada/saída e modos open-drain para pinos GPIO.

- **pinconfig.c**
  Implementa as funções de configuração de pinos definidas em pinconfig.h.

Pasta components/MyDRIVERS

- **led.h**
  Declara as funções e tipos necessários para controlar um LED.
- **led.c**
  Implementa funções para controlar um LED. Ele utiliza abstrações de manipulação de pinos definidas em outros ficheiros (pin.h e pinconfig.h) para configurar e alterar o estado do LED. O LED é configurado como *Active-Low*, ou seja, ele acende quando o pino está em nível lógico baixo.

Pasta components/MyHWMAP

- **regs_gpio_matrix.h**
  Define os registos do GPIO Matrix, incluindo configurações de seleção de funções para pinos GPIO. Contém mapeamentos detalhados para registos de entrada e saída.
- **regs_io_mux.h**
  Define os registos do IO Multiplexer e é responsável por configurar as funções e características dos pinos GPIO, como resistências pull-up/pull-down, direção, força de drive, e seleção de funções alternativas

6. Adaptação do Projeto e Reajuste Temporal

O aluno está bastante atrasado em relação ao que tinha planeado. Neste momento, tem uma estrutura sólida para os drivers de GPIO, mas falta-lhe implementar o resto do sistema local. Neste momento, o objetivo é ter a IMU a funcionar numa demo para demonstração no dia 30 de abril de 2025, e depois ter até dia 9 de maio a estrutura local do sistema completa e minimamente funcional.

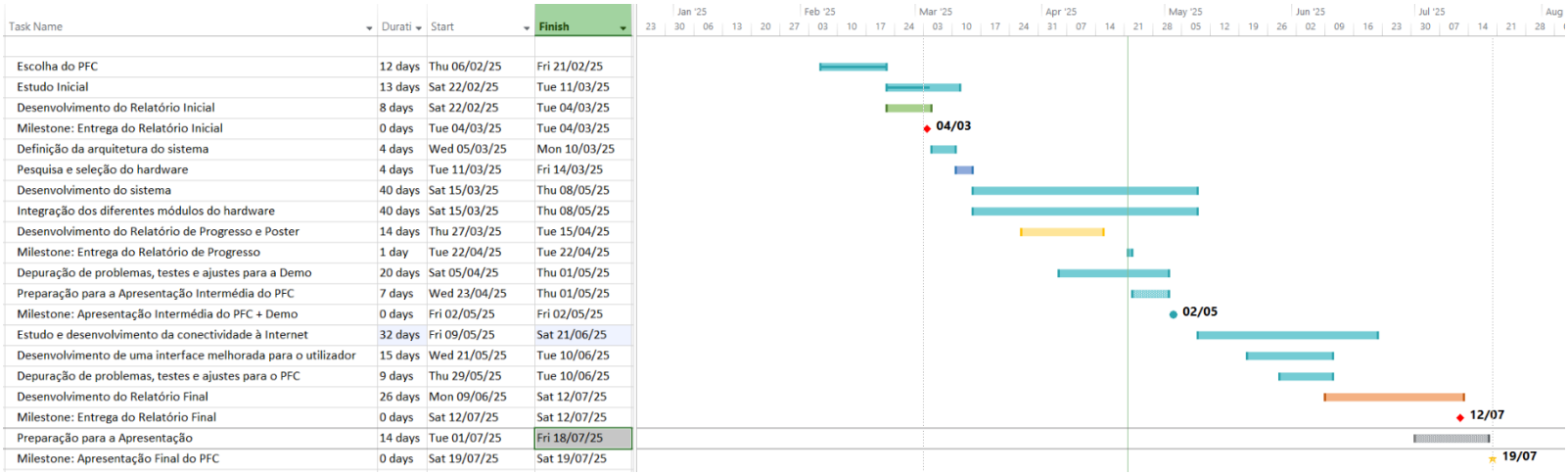

Figure 25 – Ajuste na calendarização das tarefas para o Projeto Final de Curso

7. Bibliografia

Datasheet do MPU9250

Datasheet do LSM9DS1

Datasheet do BNO055

Datasheet do NEO-6M

Datasheet do NEO-M8N

Datasheet do Quectel L80

ESP32 ESP-IDF Programming Guide

LOLIN32 Lite Schematics

ESP32 Technical Reference Manual

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/index.html