Licenciatura em Engenharia Eletrónica,
Telecomunicações e Computadores

Sistemas Embebidos para a
Internet das Coisas
2022/23

Sistema de gestão de iluminação com acesso à Ethernet

Aluno:
Alexandre Silva | 48195

Professor:
Pedro Sampaio

Julho de 2023

Conteúdo

Lista de Figuras

Introdução

O projeto da unidade curricular SEIoT visa a realização de um sistema autónomo que permite a gestão de iluminação baseado na deteção de movimento e na informação da luz. O sistema tem ainda a capacidade de envio dos dados (luz atual e registos da deteção de movimento) para um servidor na cloud, bem como a possibilidade de configuração remota. O sistema deve ser desenvolvido tendo como base o kernel FreeRTOS.

O sistema a desenvolver, cujo diagrama de blocos é apresentado na Figura x, será implementado tendo como base a placa de desenvolvimento LPCXpresso LPC1769 da NXP que inclui um microcontrolador LPC1769, um módulo detetor de movimento baseado no sensor AM312, um módulo com o sensor de luminosidade BH1750 com interface I2C, uma memória não volátil EEPROM de 128 Kb[1] com interface SPI e um módulo WIFI (ESP8266) para a comunicação com o servidor e configuração remota. O sistema disponibilizará interface local para o utilizador com um botão rotativo e de pressão e um mostrador LCD MC1602C baseado no controlador HD44780.

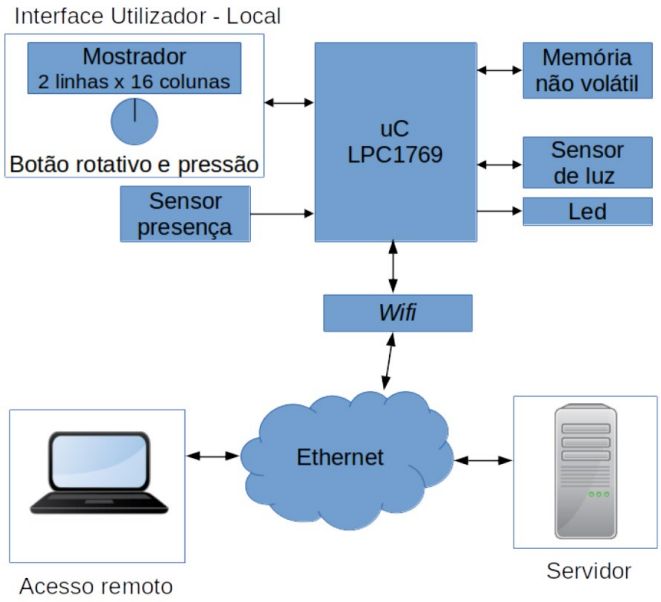


Figura 1 - Diagrama de blocos do sistema a desenvolver

Pretende-se que a aplicação a desenvolver, tendo como base o kernel FreeRTOS, apresente dois modos de funcionamento distintos (o modo normal e o modo de manutenção) e torne o sistema autónomo, ou seja, deve executar automaticamente após a ligação da energia elétrica.

No modo normal, o sistema deve acender a iluminação sempre que detetar a presença de movimento e o nível de luz ambiente for inferior ao estabelecido pela configuração.

[1] Funcionalidade não requerida por questões temporais.

Neste modo, sempre que for pressionado o botão, o sistema deverá afixar no mostrador LCD a informaçãorelativa ao calendário, relógio e o nível de luz durante 5 segundos. No restante tempo o sistema deve manter o mostrador LCD apagado.

O modo de manutenção permite definir o valor mínimo de luz que deve ser tido em conta para o acender da iluminação quando é detetada presença.

O sistema entra neste modo de funcionamento quando o botão for pressionado duas vezes repetidas (double click).

Durante definição do nível mínimo de luz, o botão rotativo promove o incremento ou o decremento do valor.

O botão de pressão, quando pressionado, confirma a seleção e realiza o retorno ao menu.

O sistema deve guardar na memória não volátil a configuração do nível mínimo de luz. O sistema deve utilizar o protocolo MQTT para, periodicamente, publicar no servidor utilizando o módulo WIFI) o valor da luz atual e o registo da deteção de movimento e subscrever um tópico[2] para a configuração remota do nível mínimo de luz. O sistema deve também possibilitar o acerto da data e a hora utilizando o protocolo NTP.

[2] Funcionalidade não requerida por questões temporais.

Esquema elétrico

A ligação dos vários periféricos anteriormente mencionados ao sistema de gestão de iluminação é visível na figura abaixo.

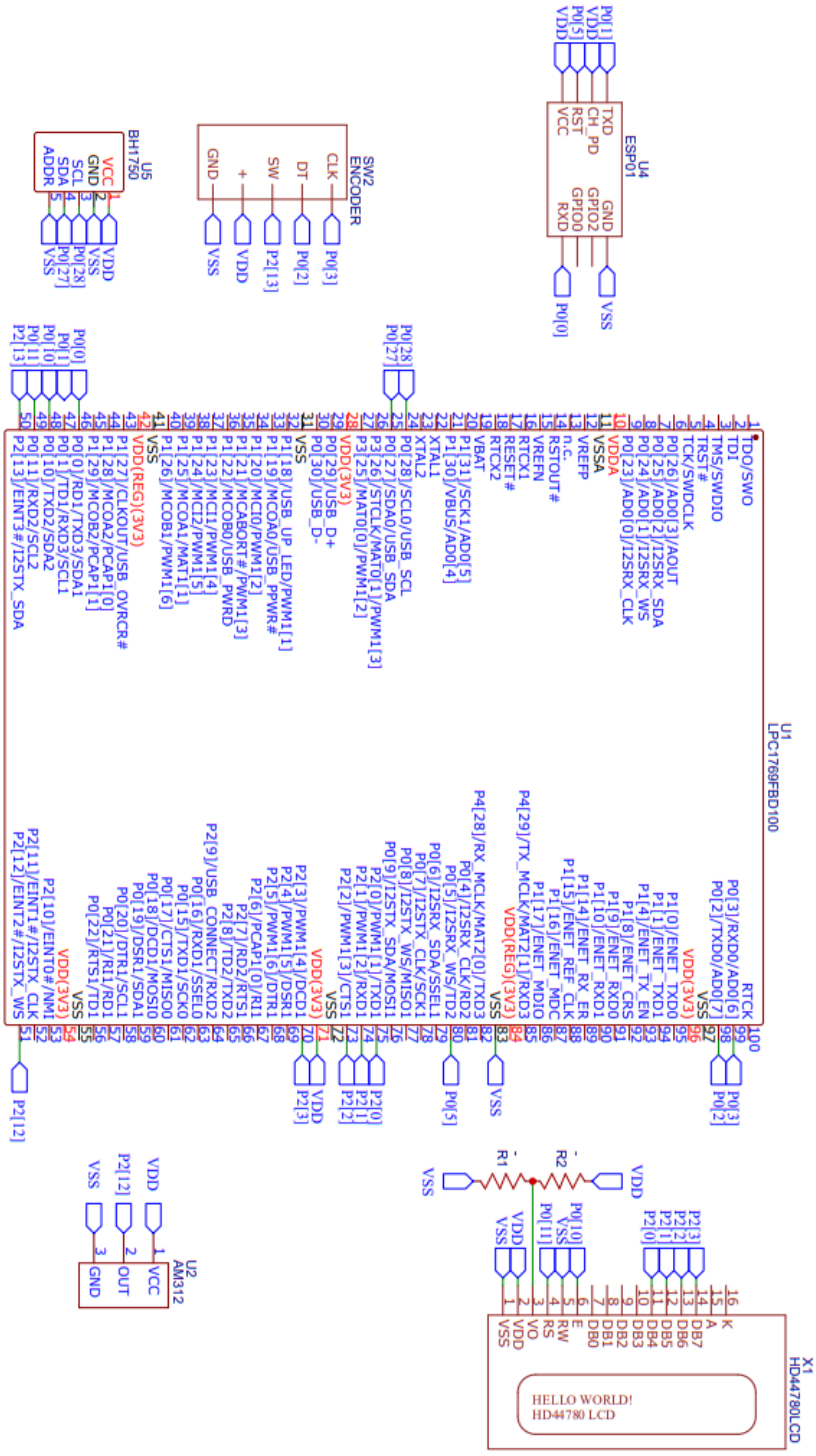


Figura 2 - Esquema elétrico do sistema de gestão de iluminação

Estrutura do relatório e diagrama de camadas

Este documento tem como objetivo expor o código desenvolvido no âmbito do projeto anteriormente enunciado, evidenciando as soluções encontradas por diagramas de estado; a sua divisão é feita em três capítulos essenciais, correspondentes a cada uma das camadas de abstração do código desenvolvido: FreeRTOS-Drivers, que desenvolve a camada DRIVERS, já fornecida, de modo a que a execução concorrente das mesmas funções seja realizada de forma segura entre os vários fios de execução; a um nível de abstração superior à camada FreeRTOS-Drivers, encontram-se as camadas UI (User Inteface) e Network que, desenvolvendo as funções já blindadas face à execução concorrente de código, permitem, no caso da primeira, a criação de uma interface de decisão com o utilizador através do display do codificador rotativo e no caso da segunda possibilitam a comunicação em rede através de diversos protocolos, como WiFi, TCP, UDP e MQTT; finalmente, a última camada fundamental trata-se da camada de aplicação, onde se procura implementar o sistema enunciado através das camadas de abstração inferiores; a figura evidencia o diagrama de camadas implementado na solução, incluindo além das três mencionadas, as restantes camadas existentes: a camada CMSIS-CORE, que providencia os vários registos de configuração do hardware do microcontrolador em registos facilmente acessíveis em C, a camada HAL que, através destes registos, implementa rotinas que fazem uso do hardware para implementar tipos de comunicação a nível eletrónico, como a entrada/saída de valores lógicos TTL, comunicação I2C, entre outros tipos de comunicação através dos quais são realizadas as funções da camada de abstração DRIVERS, que permite a comunicação entre o microcontrolador e os periféricos ligados ao mesmo.

Figura 3 - Diagrama de camadas da solução implementada

Os capítulos seguintes não dispensam, contudo, a prévia leitura do manual de referência das funções desenvolvidas, pois neste documento é apenas facultada a implementação das mesmas, sendo os manuais de referência mais adequados à apreensão da API como um todo coerente.

Capítulo 2

FreeRTOS-Drivers

A implementação das funções da camada FreeRTOS-Drivers teve como princípios fundamentais a garantia de que a execução das funções da camada abaixo, DRIVERS, ocorre sempre em secções críticas, isto é, onde apenas uma thread executa um troço de código. O mecanismo de sincronismo em questão pode ser implementado à custa de semáforos/mutex's ou queue's, sendo este último o mais apropriado quando existe a transmissão/receção de argumentos/retornos de funções dos DRIVERS. Na figura abaixo, está representado o diagrama de estados da solução implementada na API rtosLCD; esta implementação tem como base o facto que as funções da biblioteca da camada abaixo não possuem qualquer valor de retorno: desta forma, apenas é necessário garantir a exclusão mútua entre tarefas executantes desta função assim como o envio de dados para a tarefa encarregue de executar as funções do display.

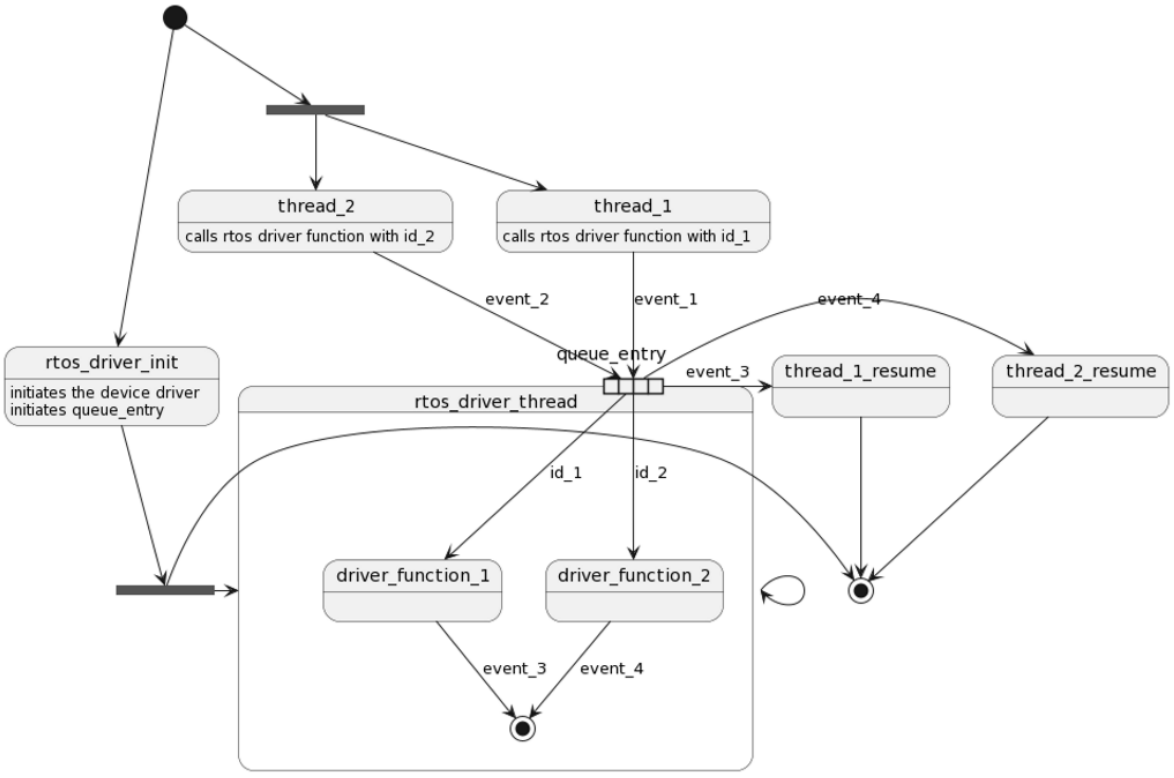


Figura 4 - Implementação de sincronismo através de uma única fila de espera

Primeiramente, como representado na figura acima, é executada a função de inicialização da API antes de qualquer outra chamada de função da mesma: feito este procedimento, são duas as threads que concorrentemente chamam funções da API. Supondo a sequência tick a tick, se primeiro ocorre a chamada da função com id1 pela thread1 seguida da chamada da função de id2 pela thread2, a thread1 será a primeira a enviar dados para a queue: depois disto, a thread2 enviará dados para queue ao mesmo tempo que a tarefa encarregue da execução das funções da camada abaixo decide a função a executar com base no id recebido, ao mesmo tempo que a thread1 resume o seu fio de execução.

Relativamente às API's onde existe retorno, a solução implementada, representada na figura abaixo, bastante semelhante à anteriormente mencionada, consiste numa queue extra destinada a transmitir à função chamante o retorno obtido pela função da camada abaixo.

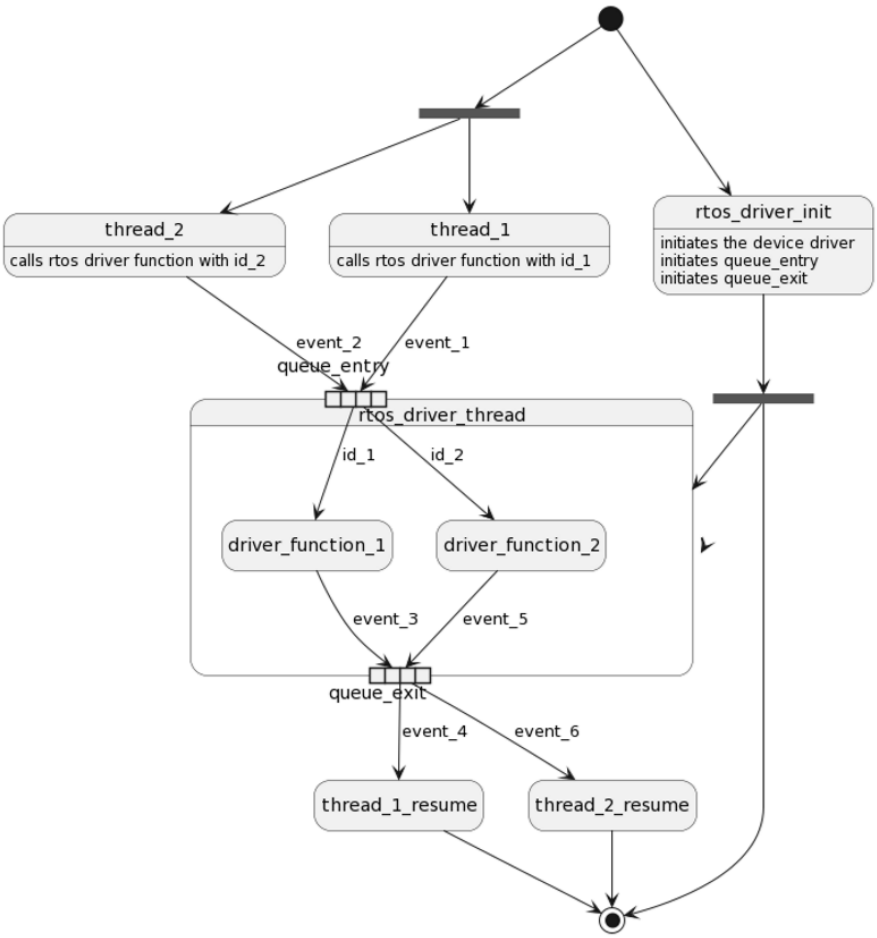
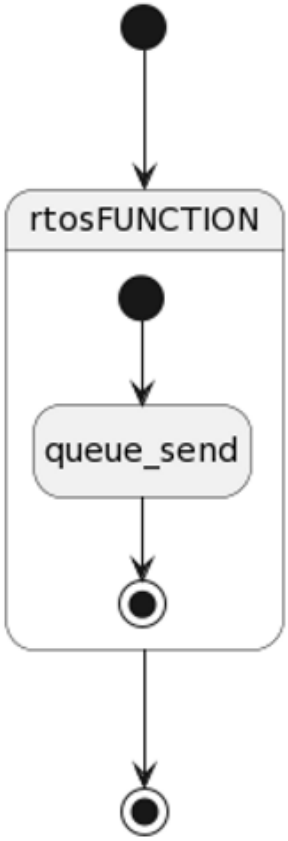
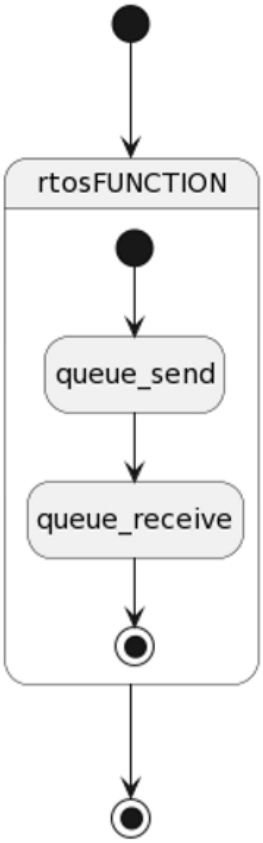


Figura 5 - Implementação de sincronismo através de uma dupla fila de espera

Nesta implementação, assumindo também a execução tick a tick, a primeira chamada à função da camada FreeRTOS entrará na queue_entry que por sua vez é lida pela tarefa encarregue de executar as funções da camada abaixo: desta vez, ao invés do prosseguimento do fio de

execução pela tarefa que chamou a função pela primeira vez, ocorrerá o seu bloqueamento(da tarefa) até que sejam recebidos pela mesma os dados retornados na denominada queue_exit. Estes procedimentos são visíveis nas figuras abaixo, onde são evidenciados os estados de execução das funções da camada FreeRTOS-Drivers cuja implementação coincide com aquelas anteriormente abordadas.





Figuras 6 e 7 - Implementação de fila única e dupla das funções da camada FreeRTOS-Drivers

A implementação mais atípica de uma API da camada abordada no presente capítulo é a referente ao PIR: neste caso, ao invés de optar por uma queue, atendeu-se à ausência de necessidade de implementação de uma queue de envio de dados, pois não existe qualquer argumento para a função de deteção de presença, como também à ausência de necessidade de secções críticas, dada a natureza leitora da mesma função: assim, uma rotina de interrupção através do FreeRTOS foi utilizada para verificar de forma periódica o valor lógico de presença, guardando-o numa variável.

1.1 rtosLOCK

Mecanismos como um semáforo ou uma queue surgem como recursos disponibilizados pelo sistema operativo em tempo real para colmatar problemas como uma race condition: é possível realizar as funções de uma API de forma thread-safe, garantindo que se duas threads executarem funções relacionadas com um recurso, como o LCD, a primeira thread executará uma vez uma função, seguida da segunda thread que por sua vez executará outra função, seguida da primeira thread que por sua vez executará outra função: existem, contudo, cenários nos quais uma thread possa necessitar de executar duas funções, de forma concorrente, sem passar o fio de execução para a outra thread depois da execução da primeira função.

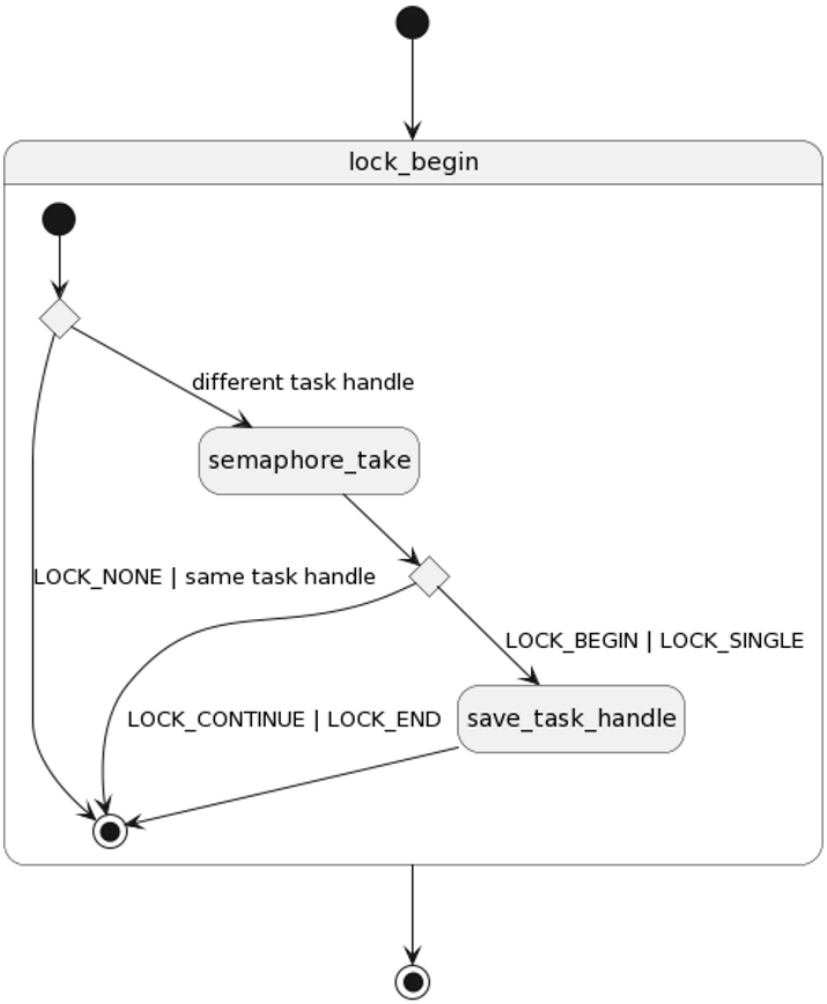


Figura 7 - Implementação da função rtosLOCK_Begin

O mecanismo de mutex de tarefa, desenvolvido como rtosLOCK, garante, por um lado, a execução de um número ilimitado de diferentes funções de uma API numa secção crítica consoante a vontade do programador, como também permitem às API's das camadas superiores gozarem da mesma funcionalidade sem o surgimento de qualquer erro de operação.

A função rtosLOCK_Begin inicia assim, como evidenciado na figura acima, um mutex que bloqueará tarefas com TaskHandle diferente da thread que inicie o mecanismo.

Na figura abaixo, que representa o diagrama de estados da função rtosLOCK_End, responsável pelo fim da região crítica previamente iniciada pela mesma tarefa, é realizado um delay de um tick após o liberamento do mutex associado à mesma tarefa, permitindo assim a outra tarefa, no momento bloqueada na função rtosLOCK_Begin, associar o TaskHandle do mecanismo LOCK à sua tarefa e relegar as demais threads a ficarem bloqueadas no mesmo mutex.

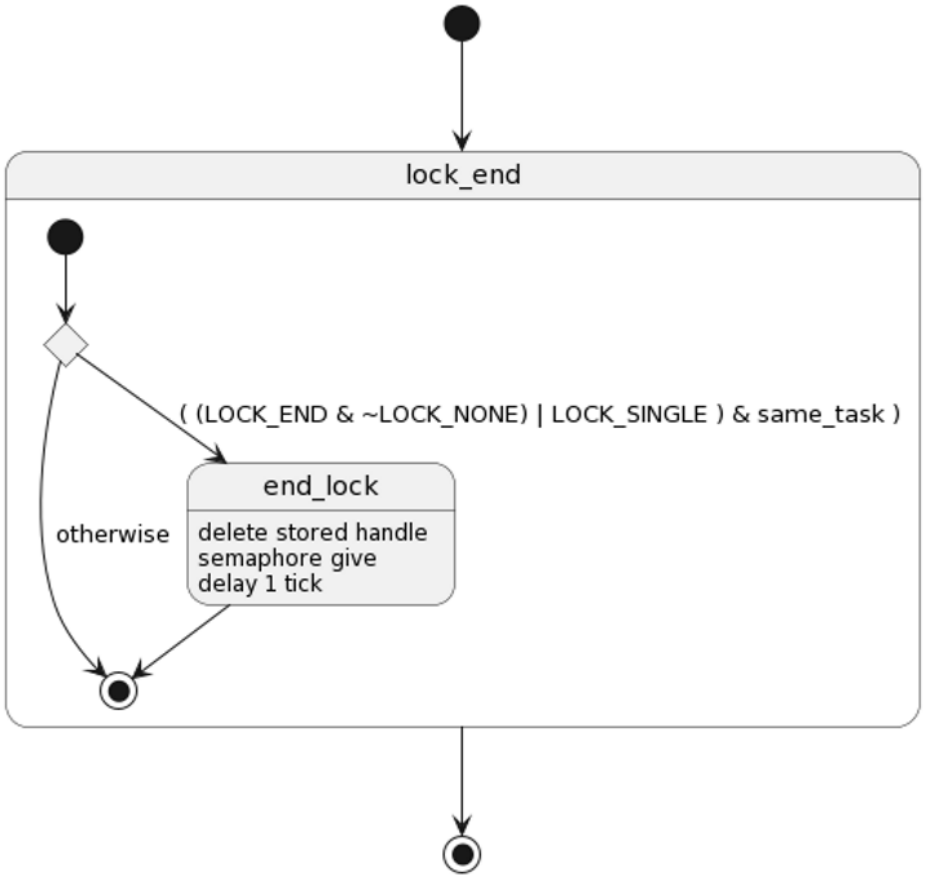


Figura 8 - Implementação da função rtosLOCK_End

1.1 ESP8266

O módulo ESP8266, utilizado para estabelecer comunicação através dos protocolos WiFi e da camada de transporte, funciona através da receção de comandos AT, seguida da transmissão de comandos de resposta que permitem saber o resultado do comando previamente enviado.

Utilizando as funções fornecidas, nomeadamente ESPSERIAL_Send e ESPSERIAL_Recv, foi desenvolvida uma API que permite a interação com o módulo, disponível no manual de referência da biblioteca FreeRTOS-Drivers.

A solução implementada consiste na leitura dos dados enviados de forma assíncrona pelo módulo ESP8266 para um buffer, de tamanho limitado, através da função rtosESPSERIAL_Refresh, que espera a receção de dados durante um período fixo cada vez que é chamada, como evidenciado na figura abaixo.

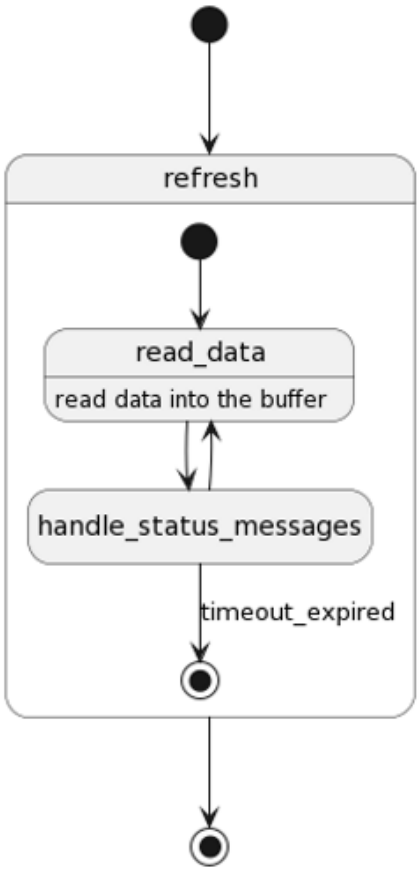


Figura 9 - Implementação do algoritmo da função rtosESPSERIAL_Refresh

A transmissão de mensagens por parte do módulo não é, contudo, limitada apenas às respostas a comandos enviados: o módulo ESP8266 envia também mensagens relativas ao estado de ligações correntes que podem, quando não são propriamente retiradas, interromper uma mensagem de resposta pretendida. Por forma a colmatar esta ocorrencia, estas mensagens são procuradas cada vez que a função rtosESPSERIAL_Refresh é chamada, através da função rtosESPSERIAL_xHandleStatus, cuja implementação é visível na figura abaixo.

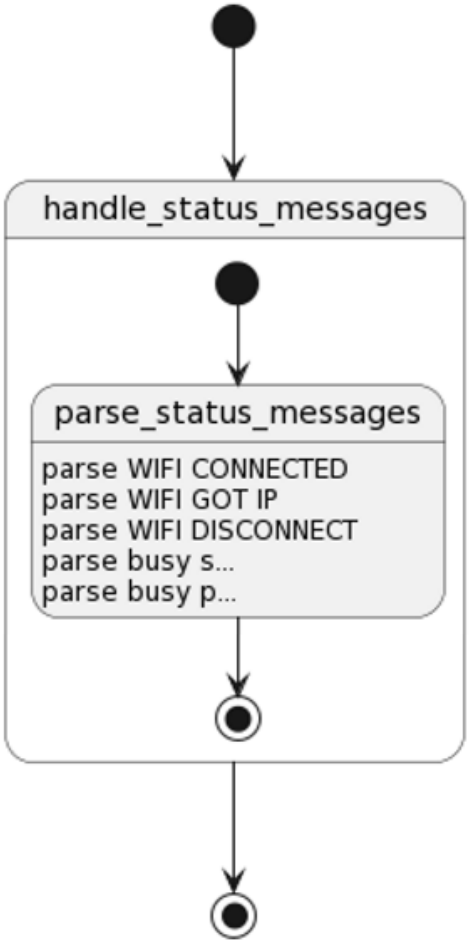


Figura 10 - Implementação do algoritmo rtosESPSERIAL_xHandleStatus

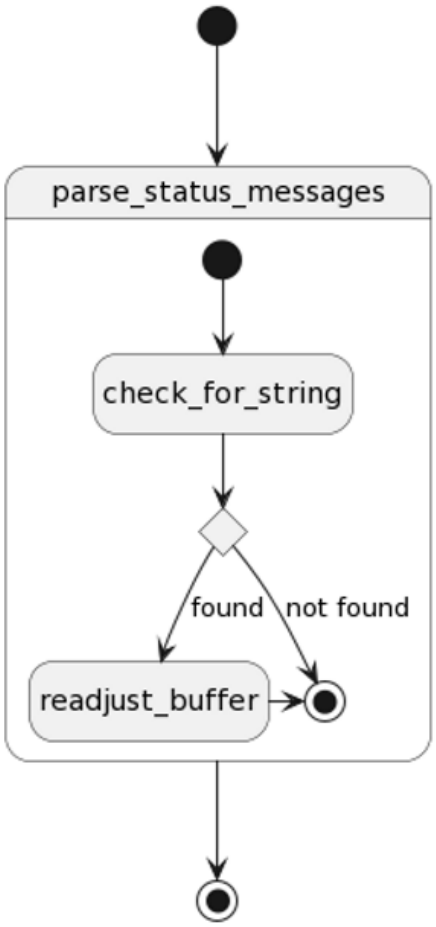


Figura 11 - Implementação do algoritmo rtosESPSERIAL_xParseStatus

No que toca à transmissão de dados, é importante garantir que nenhuma resposta não recebida seja deixada para trás: neste caso, a função encarregue pela verificação da resposta poderia induzir o sistema em erro. Assim, torna-se imperativo que o buffer seja limpo antes da transmissão de dados, como evidenciado na figura abaixo.

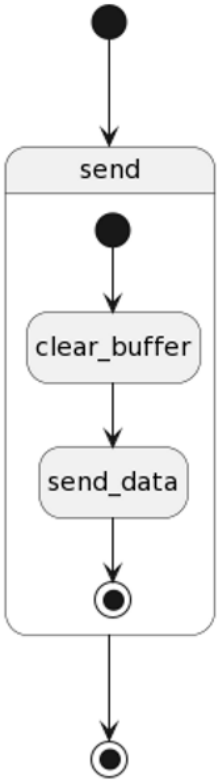


Figura 12 - Implementação do algorismo da função rtosESPSERIAL_Send

A espera de respostas do módulo ESP8266 bloquearia o fio de execução durante um período tempo indefinido caso não existisse um período de espera máximo: assim, nas funções que realizam a espera de respostas do módulo, foi implementada uma sequência de estados representativos do estado atual do módulo: o estado START passado a este tipo de funções inicia a espera a uma determinada resposta, ficando no estado WAIT até que a mensagem desejada seja encontrada, passando deste modo para o estado SUCCESS; quando uma mensagem de erro pré-definida é encontrada, o estado muda imediatamente para FAILURE, e quando o tempo de execução máximo da função é expirado, é também para failure que o estado transita.

A implementação das funções de espera desta API permite que a função chamante possa realizar outras tarefas enquanto o estado não transita de WAIT para FAILURE ou SUCCESS,

dado que a espera ativa é realizada pela tarefa gerente do módulo ESP8266. A implementação da função rtosESPSERIAL_WaitResp encontra-se evidenciada na figura abaixo.

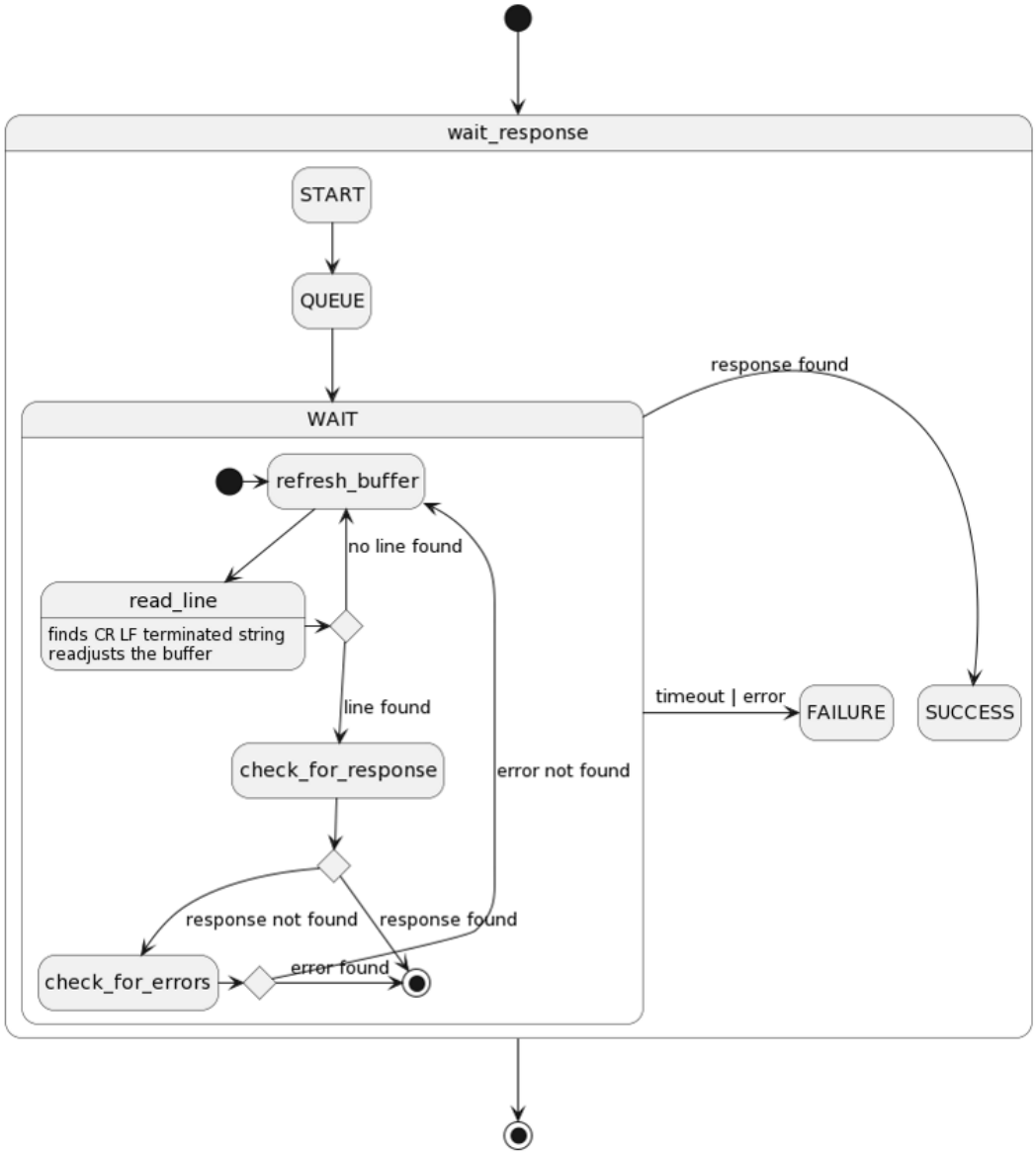


Figura 13 - Implementação do algoritmo da função rtosESPSERIAL_WaitResp

A função rtosESPSERIAL_WaitStr é bastante semelhante à função anteriormente descrita, com a exçeção de que a anterior verifica respostas linha-a-linha, sendo esta função útil em situações onde a resposta pretendida não acaba em CR LF.

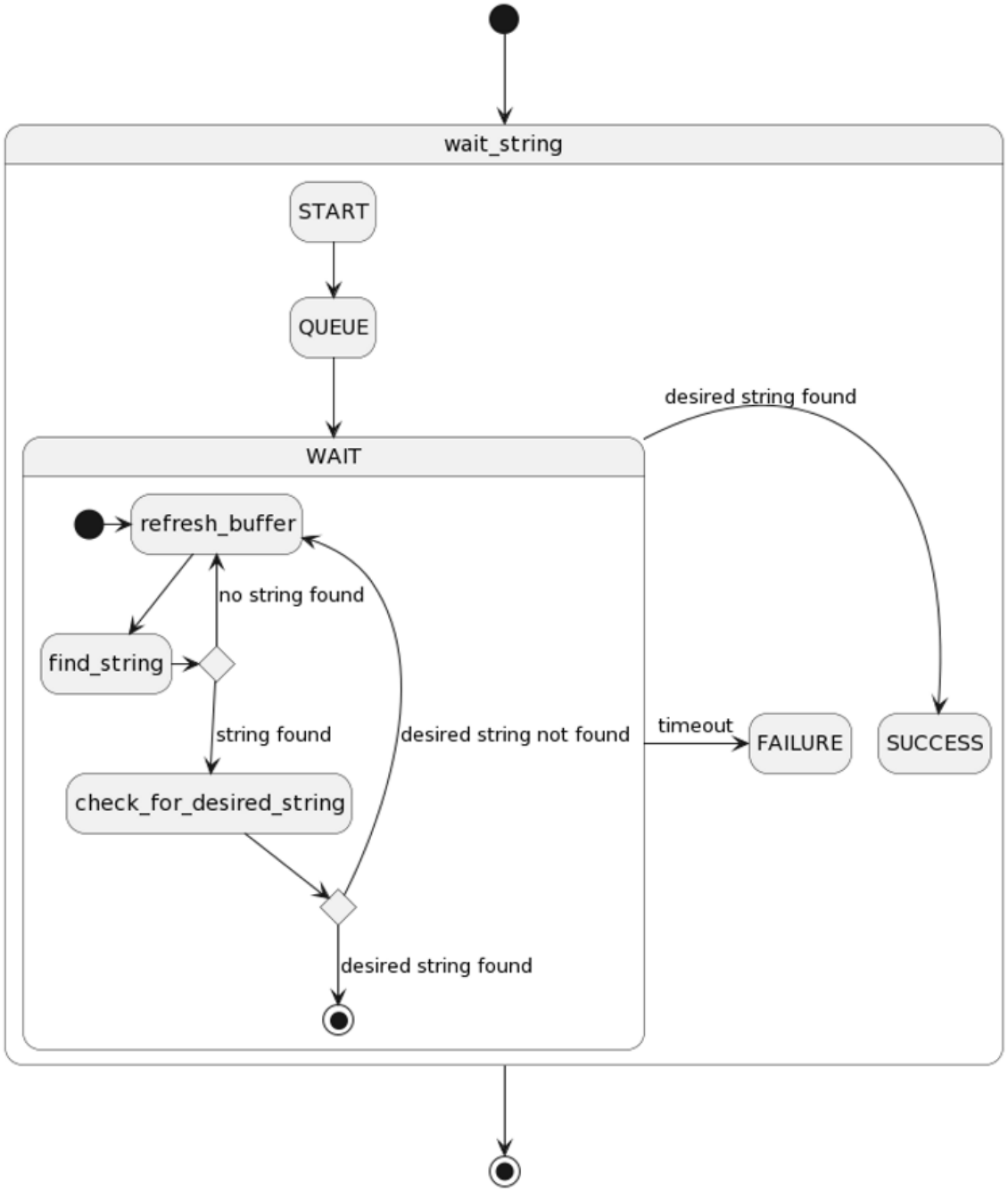


Figura 14 - Implementação do algoritmo da função rtosESPSERIAL_WaitStr

A função rtosESPSERIAL_SendAT faz uso de grande parte das funções anteriormente descritas, acrescentando CR LF a um comando passado como argumento e ordenando à tarefa encarregue da execução das funções da camada abaixo, do módulo ESP8266, uma espera ativa, alterando o valor do estado cujo endereço é passado como argumento no momento em que seja determinado sucesso ou fracasso do comando.

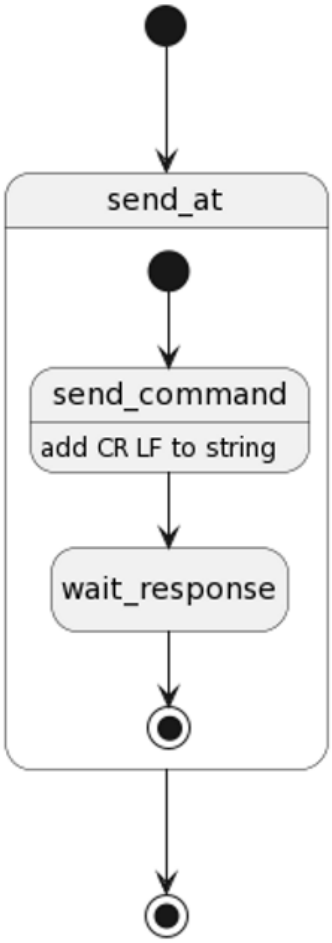


Figura 15 - Implementação do algoritmo da função rtosESPSERIAL_SendAT

Por fim, presente implementação, foi realizada a função rtosESPSERIAL_Setup, visível na figura abaixo, que envia uma série de comandos AT pré-estabelecidos ao módulo antes de permitir que qualquer outra tarefa faça uso das suas funcionalidades: esta implementação tem como limitação imediata, contudo, a incapacidade de lidar com falhas de ligação do módulo que podem induzir a um reinício, levando a que futuros comandos sejam comprometidos, como é o caso de CWLAP, por exemplo, que requer o prévio envio do comando CWMODE=1.

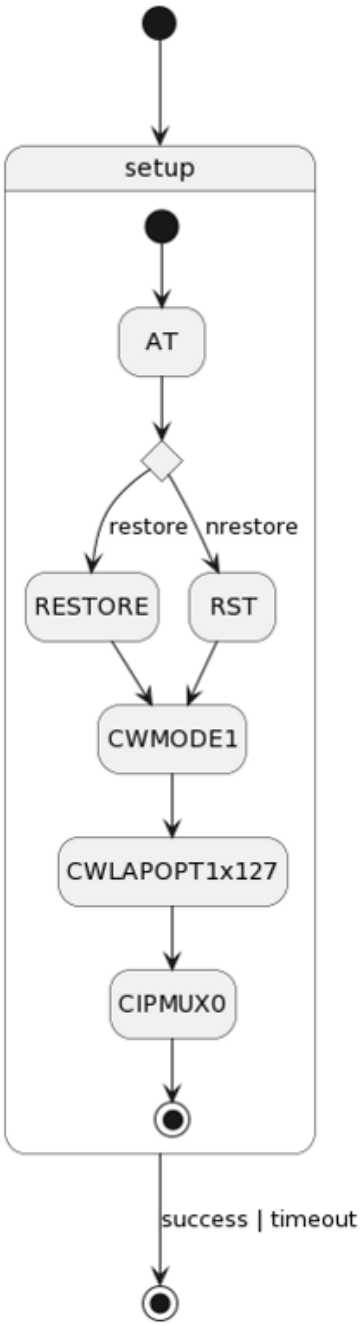


Figura 16 - Implementação da função rtosESPSERIAL_ Setup

Capítulo 2

FreeRTOS-Network-UI

As funções desenvolvidas para a camada FreeRTOS-Network-UI tem como objetivo a interface com protocolos comunicação através do módulo ESP8266: deste modo, foram desenvolvidas API's para WIFI, camada de transporte (TCP-IP e UDP), NTP e MQTT, correspondentes à parte "Network", bem como para a interface de menus através do display e do codificador rotativo, correspondente à parte restante, "UI".

Para o estabelecimento de uma ligação WIFI, foi desenvolvida a função netWIFI_Connect, que envia o comando CWJAP em conjunto com o SSID e password do AP pretendido, disponibilizando ainda a opção de tornar a rede como predefenida, isto é, quando o módulo for reiniciado, conectar-se-à a este AP automaticamente. Esta função realiza assim uma espera ativa até que ocorra sucesso ou falha, em resposta ao comando enviado.

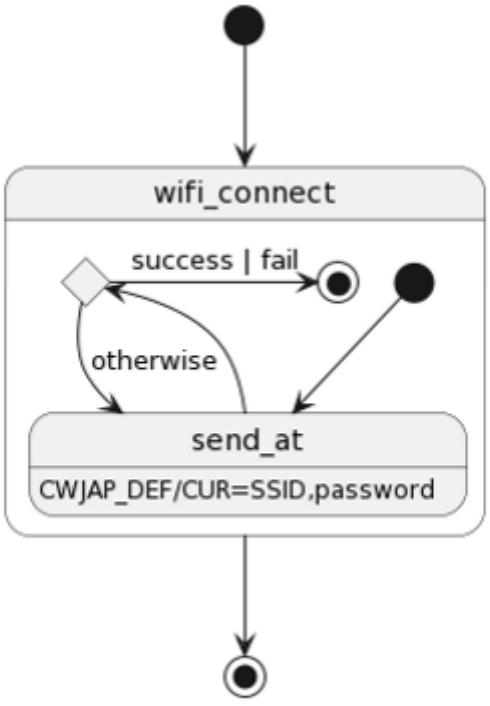


Figura 17 - Implementação da função netWIFI_Connect

A função netWIFI_Disconnect disconecta o módulo da rede à qual se encontra conectada no momento da chamada, através do comando CWQAP.

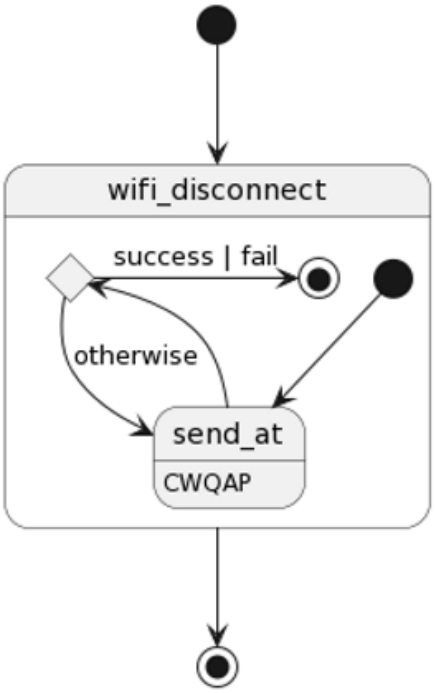


Figura 18 - Implementação da função netWIFI_Disconnect

O estado corrente da ligação WiFi é verificável através do comando AT+CIPSTATUS, através do qual é possível saber se a ligação se encontra com IP atríbuido, que é aliás o objetivo da função netWIFI_Check, através da resposta STATUS:2. Em certos momentos, o encerramento de uma ligação de TCP-IP, por exemplo, pode levar a que o estado retornado seja diferente de STATUS:2, fazendo com que nesta implementação o resultado dependa apenas de STATUS:5, que representa a inexistência de qualquer ligação.

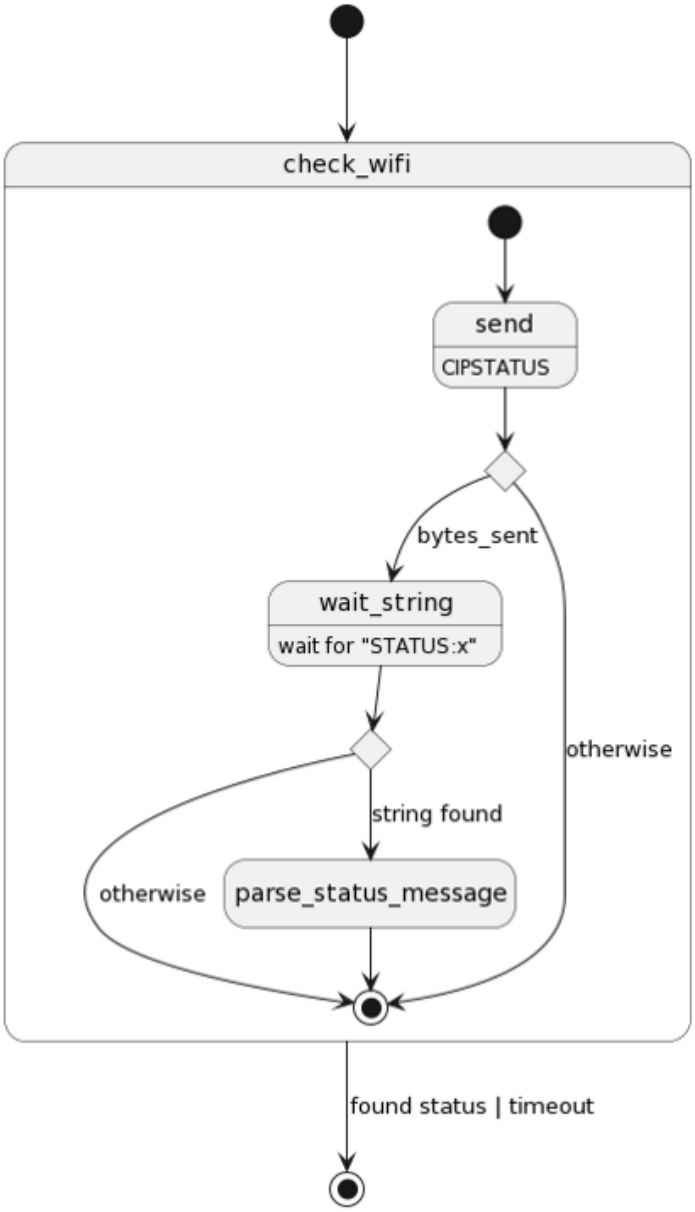


Figura 19 - Implementação da função netWIFI_Check

O conjunto de instruções relacionadas aos comandos AT disponibiliza ainda o comando CWLAP, que permite a listagem dos AP's disponíveis encontrados pelo módulo. A função netWIFI_Scan implementa esta funcionalidade, realizando o parsing dos vários atributos retornados, entre os quais o SSID, endereço MAC e RSSI, entre outros, que são guardados em memória, prontos a serem obtidos através da função netWIFI_ScannedSSID, que retorna um objeto que possui todos estes parâmetros associados a cada AP.

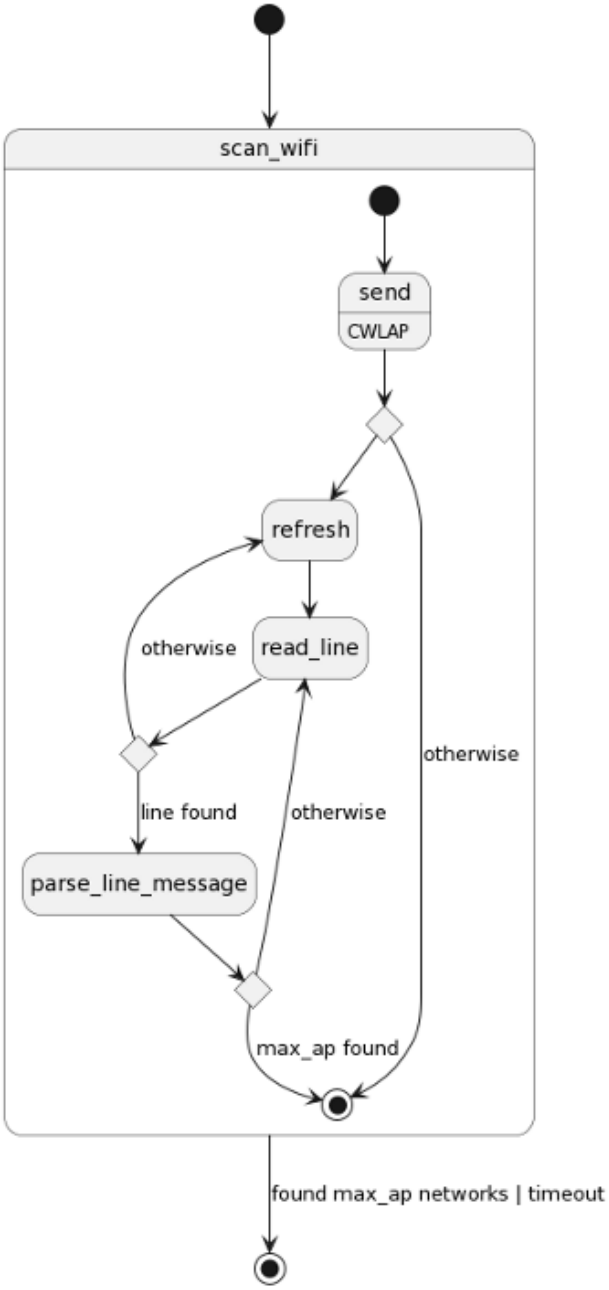
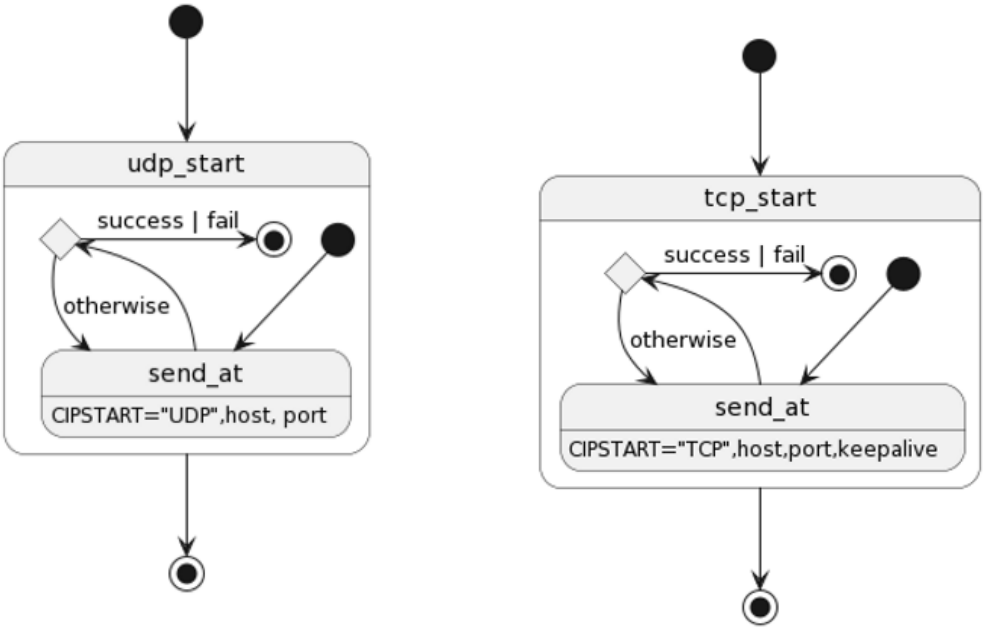


Figura 20 - Implementação da função netWIFI_Scan

Relacionadas à camada de transporte, as funções netUDP_Start e netTCP_Start permitem a iniciação de uma ligação UDP ou TCP-IP, respetivamente: a sua implementação, evidenciada nas figuras abaixo, é apenas diferente nos parâmetros a introduzir no comando CIPSTART.



Figuras 21 e 23 - Implementação das funções netUDP_Start e netTCP_Start

Na solução encontrada, cada vez que uma ligação da camada de transporte é iniciada, o recurso do módulo ESP8266 é bloqueado pela função rtosLOCK_Begin, sendo apenas desbloqueado pelas funções netTCP_Close/netUDP_Close, cuja implementação é igual.

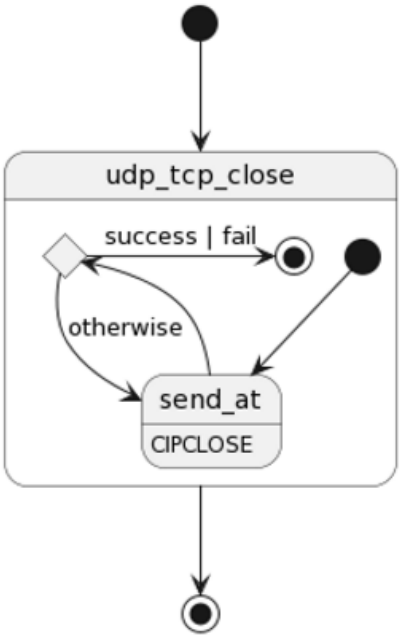


Figura 22 - Implementação das funções netUDP/TCP_Close

Relativamente às funções de transferência de dados, a função netTRANSPORT_Recv espera a resposta +IPD por parte do módulo, lendo o número de bytes totais a receber e permitindo a leitura de uma parte parcial dos dados a receber, alterando deste modo o estado inicial a executar da próxima vez que é feita uma chamada à mesma função; este número de bytes a receber é guardado, sendo esquecido apenas no caso de timeout da função.

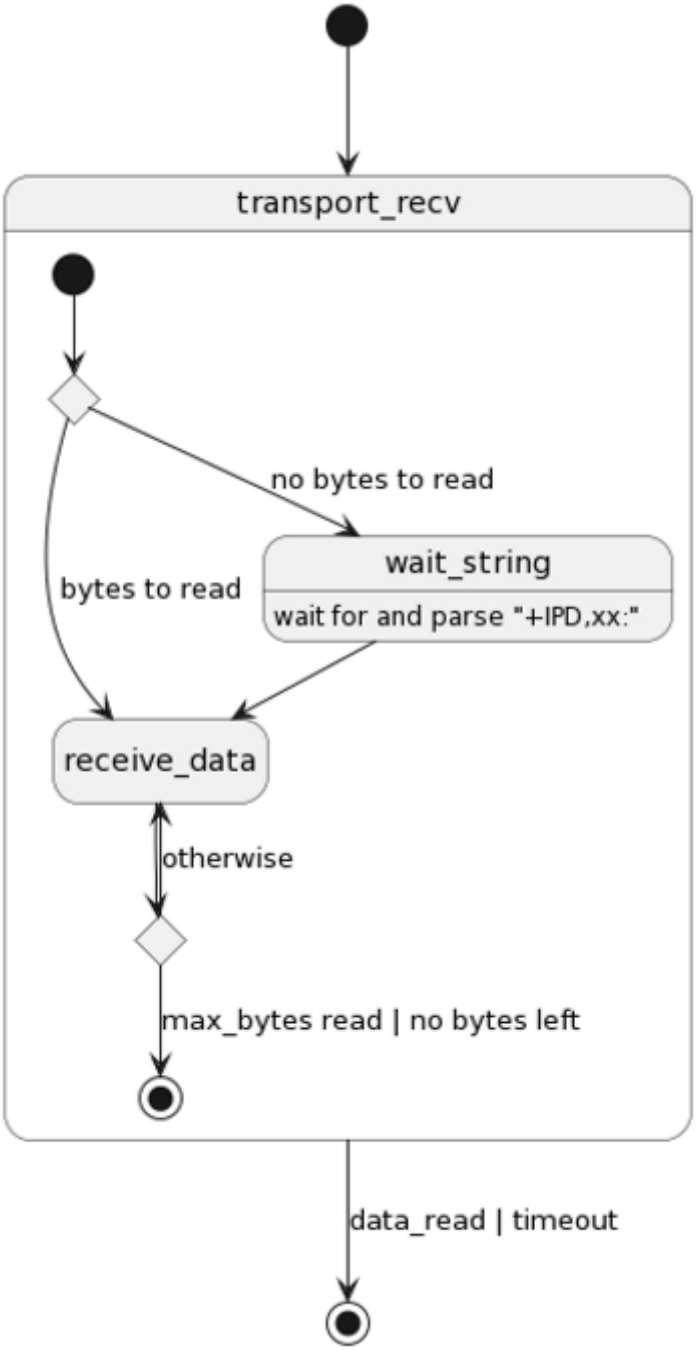


Figura 23 - Implementação da função netTRANSPORT_Recv

A função netTRANSPORT_Send realiza o envio de dados para um determinado endereço, através do envio do comando CIPSEND, que determina o nº de bytes a enviar, seguido do prompt de “>”, que demarca o início da transferência destes dados, ao final da qual é recebido o comando “SEND OK”.

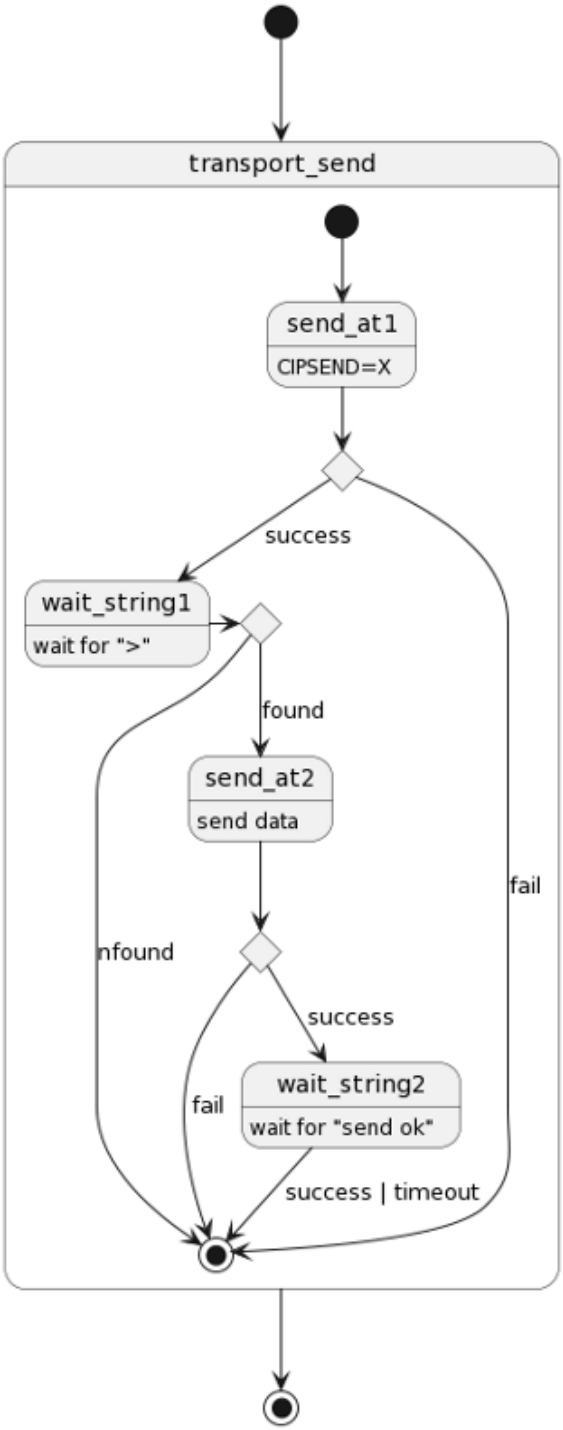


Figura 24 - Implementação da função netTRANSPORT_Send

O protocolo NTP permite a receção do tempo atual, através da comunicação UDP com um servidor remoto. Primeiramente, é enviado um header, seguido de transferência e receção dos dados pretendidos, que estão localizados nos bytes 40-44 recebidos em resposta.

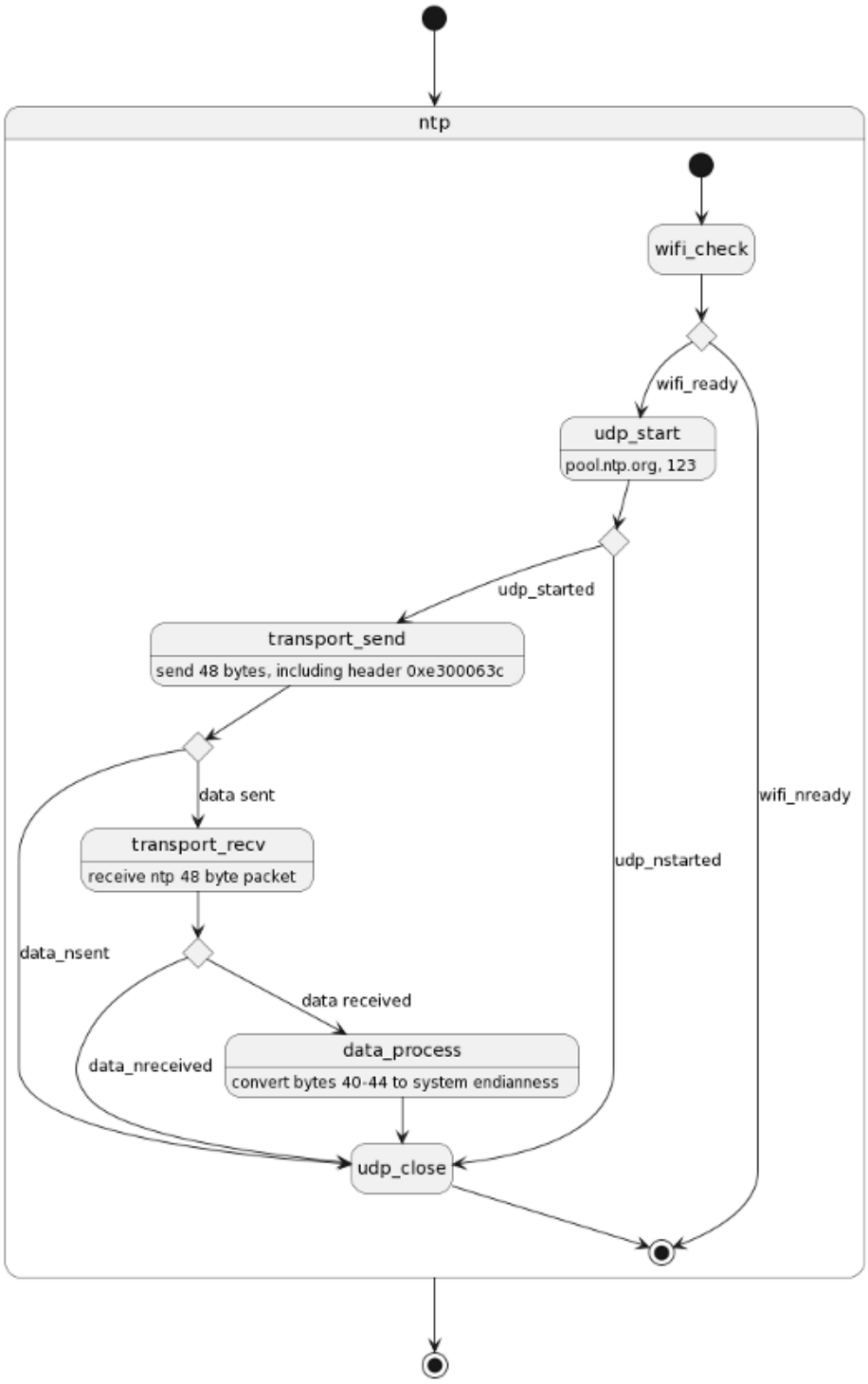


Figura 25 - Implementação da função netNTP_GetTime

No que toca ao protocolo MQTT, a sua implementação, evidenciada na figura abaixo, é realizada com base em dois estados fundamentais, ready e idle, a partir dos quais é possível suspender e retomar, respetivamente, a ligação com o broker pretendido. A publicação de dados é realizada através de uma queue, cujos dados são recebidos no estado ready.

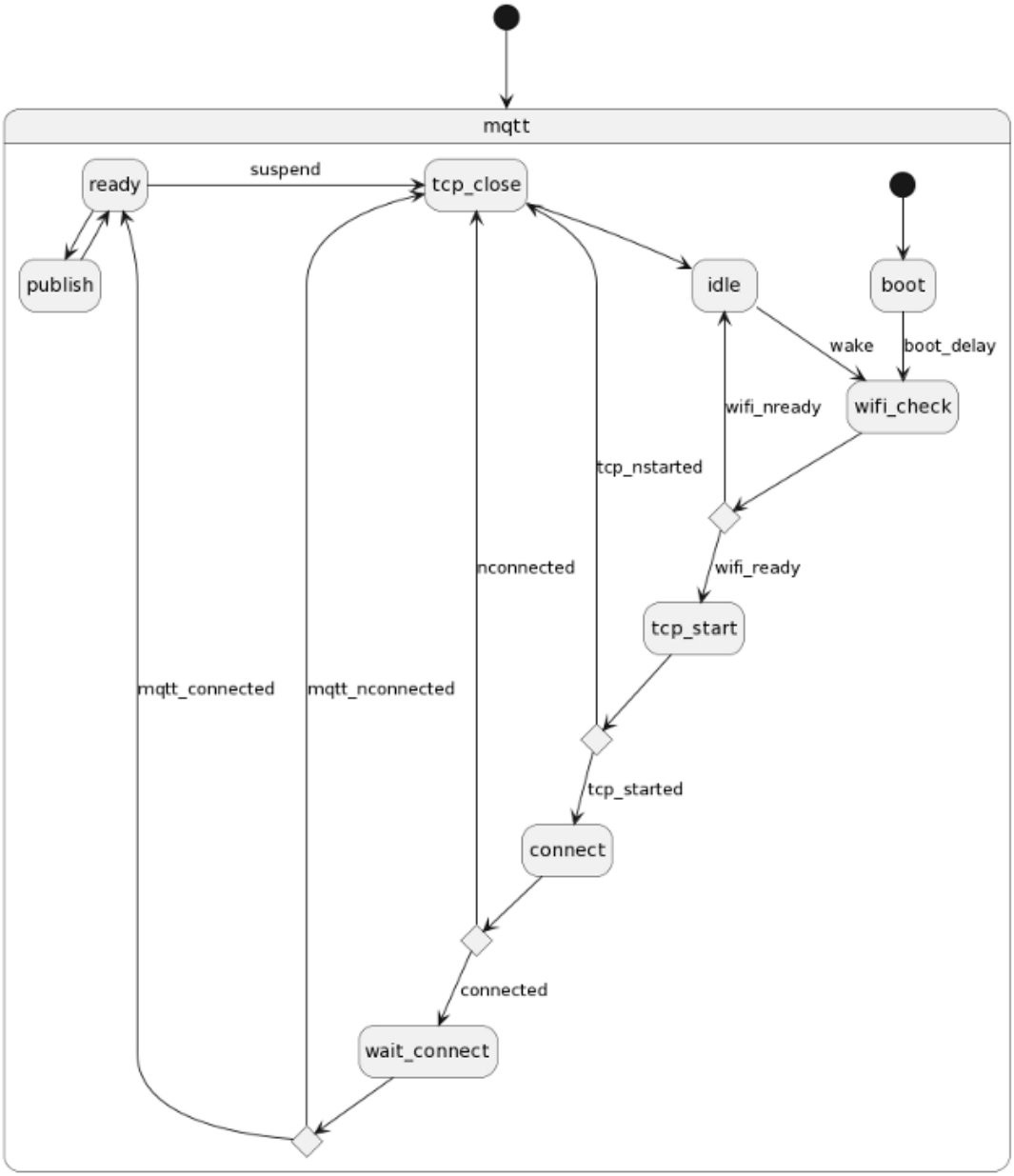


Figura 26 - Implementação da tarefa netMQTT_Task

No que toca às funções desenvolvidas para gerar uma interface de menu através do display e do codificador rotativo, a função uiMENU_Execute permite que sejam passados, dentro de um objeto previamente gerado pela função uiMENU_Generate, como argumentos o título do menu, os nomes dos vários itens que o compõeme e os handlers respetivos a cada item.

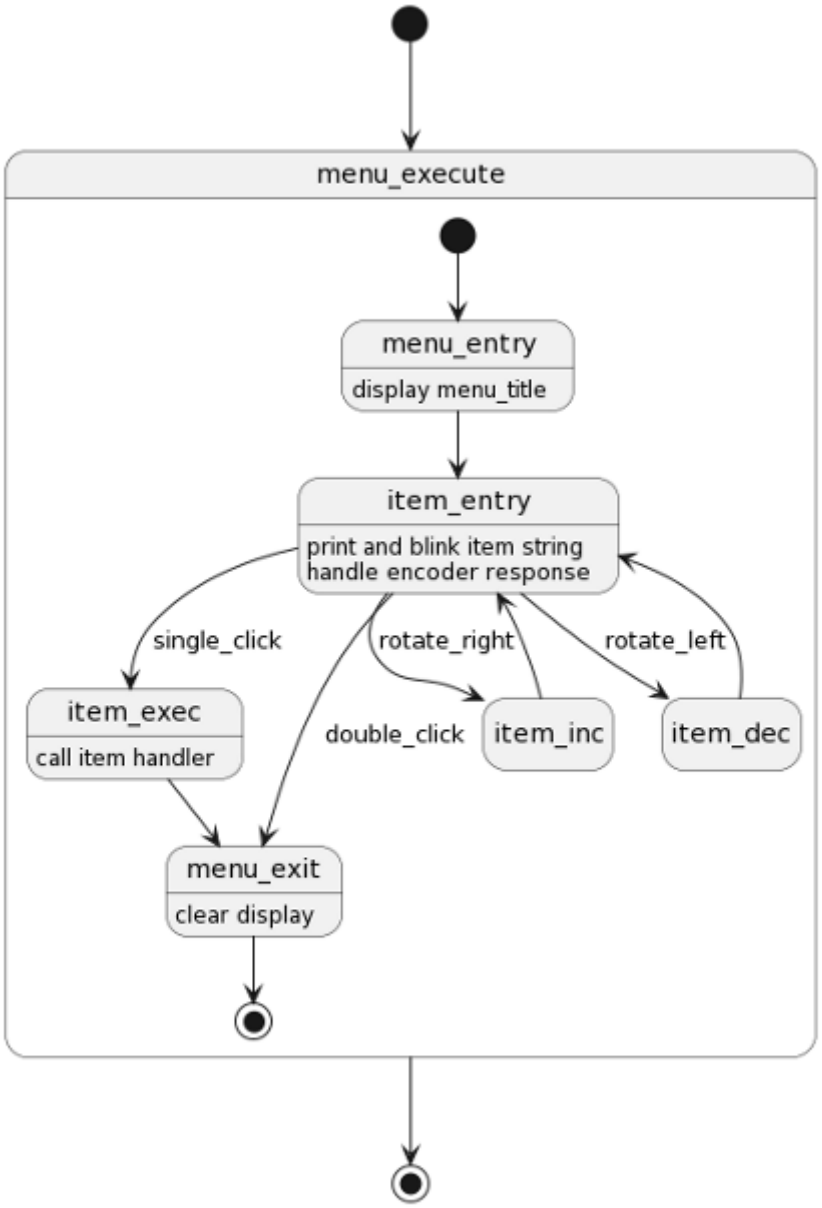


Figura 27 - Implementação da função uiMENU_Execute

Outra função importante para a interface com os menus é a função uiMENU_InputData, que permite a entrada/modificação de dados através de strings e conjuntos de carateres suscetíveis de serem utilizados para a modificação de cada string, como por exemplo números ou números e letras.

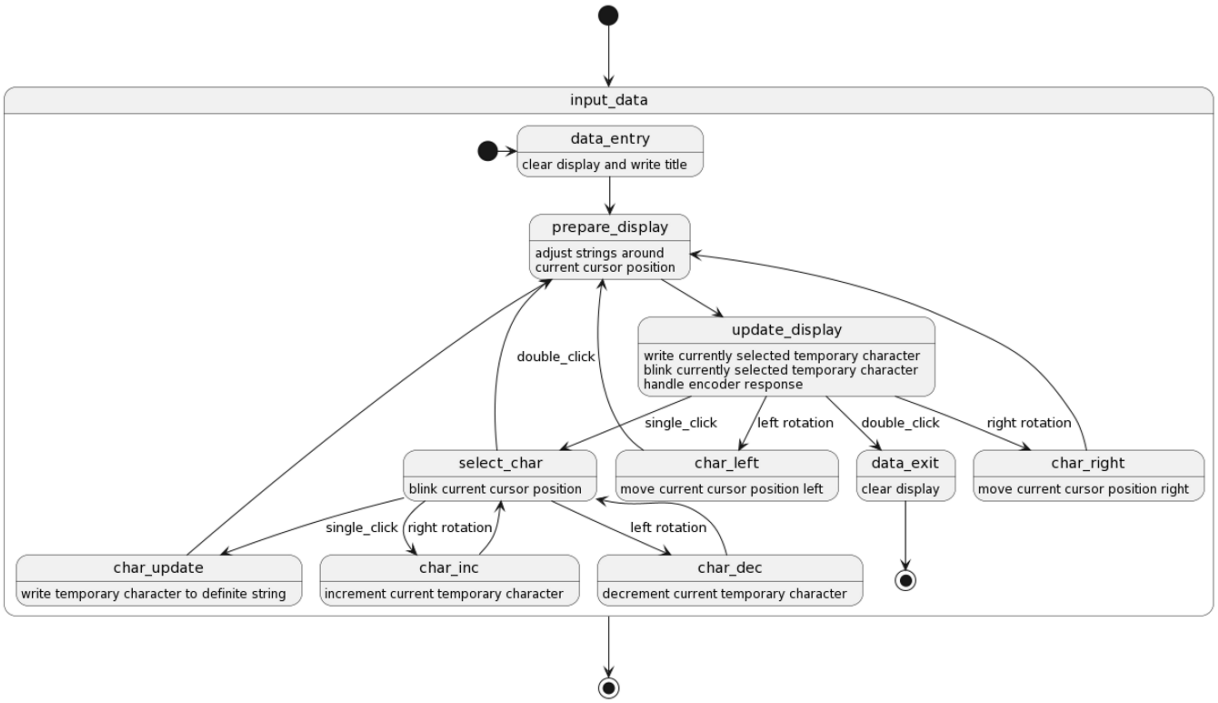


Figura 28 - Implementação da função uiMENU_InputData

Capítulo 3

Aplicação

No que toca à camada de aplicação, a solução é evidenciada pela figura abaixo: existem duas tarefas, cada uma responsável pelo seu modo, que comunicam entre si através de notificações de tarefas, alternando o fio de execução da aplicação através dos mesmos.

No modo normal, a presença e o acendimento da luz consoante o nível de luminosidade detetado pelo sensor BH1750 são geridos pela rotina de interrupção ManageLightISR, que é executada periodicamente; relativamente à publicação de dados relativos à luminosidade e à presença, esta tarefa é realizada por outra rotina de interrupção, que inicia o serviço MQTT sempre que este se encontre indisponível. Ambas estas rotinas são paradas quando o modo de manutenção é executado, e retomadas no modo normal Já no modo de manutenção foram definidos vários menus, relativos a diversas funcionalidades como o ajuste de hora e data, a sincronização do relógio do sistema por NTP, o ajuste do nível mínimo de luz, entre outros.

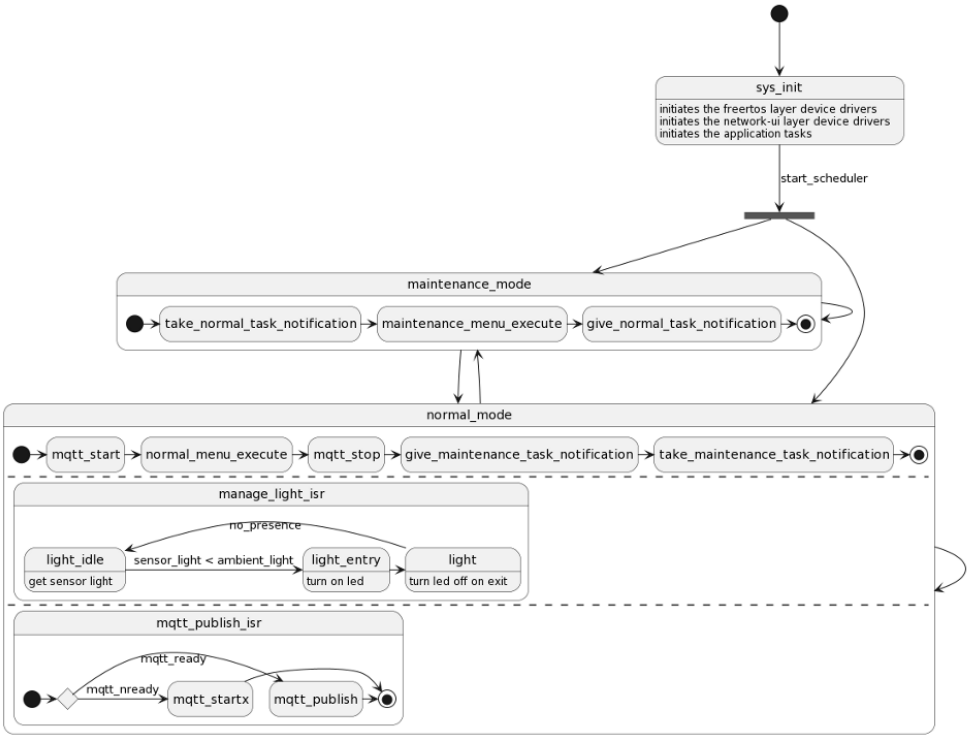


Figura 29 - Implementação da aplicação

Conclusões

A implementação do sistema enunciado no projeto da unidade curricular terminou com o desenvolvimento de bibliotecas das várias funcionalidades do sistema que passam a testes unitários, mas que utilizadas em conjunto gastam cerca de 90% da heap alocada (16Kb) para a utilização com o FreeRTOS; desta forma, quando certas funcionalidades do menu foram executadas, o sistema tende a cair em hardfaults surgidas na função pvPortMalloc, o que indica insuficiência de memória. Relativamente à utilização dos protocolos estudados, o sistema publica periodicamente com sucesso num servidor remoto através do protocolo MQTT e atualiza com sucesso a hora corrente do sistema através do protocolo NTP. Adicionalmente, as interfaces de menu desenvolvidas permitem a existência de uma grande quantidade de funcionalidades com densidade de código baixa relativamente a implementações com recurso a máquinas de estado mais extensas.

Bibliografia

- FreeRTOS Reference Manual
- Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide
- ESP8266: AT Instruction Set
- Miro Samek, Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, Newnes (2^{a} edição), 2008.