UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DAINF — DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

GUSTAVO LUIZ ANDRADE CORRÊA

SISTEMA DE ATUALIZAÇÃO DE FIRMWARE OVER-THE-AIR PARA DISPOSITIVOS DE IOT BASEADO NAS BIBLIOTECAS LWIP, MBED TLS E FATFS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2021

GUSTAVO LUIZ ANDRADE CORRÊA

SISTEMA DE ATUALIZAÇÃO DE FIRMWARE OVER-THE-AIR PARA DISPOSITIVOS DE IOT BASEADO NAS BIBLIOTECAS LWIP, MBED TLS E FATFS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de engenheiro de computação.

Orientador: Prof. Dr. Gustavo Weber Denardin
Departamento Acadêmico De Elétrica

PATO BRANCO
2021

Ministério da Educação
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEP. ACADEMICO DE INFORMATICA DAINF-PB

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO - TCC

SISTEMA DE ATUALIZAÇÃO DE FIRMWARE OVER-THE-AIR PARA DISPOSITIVOS DE IOT BASEADO NAS BIBLIOTECAS LWIP, MBED TLS E FATFS

Por

Gustavo Luiz Andrade Correa

Monografia apresentada às 13 horas 50 min. do dia 12 de agosto de 2021 como requisito parcial, para conclusão do Curso de Engenharia da Computação da Universidade Tecnológica Federal do Paraná, Campus Pato Branco. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação e conferidas, bem como achadas conforme, as alterações indicadas pela Banca Examinadora, o trabalho de conclusão de curso foi considerado APROVADO.

Banca examinadora:

Prof. Dr. Fabio Luiz Bertotti	Membro
Prof. Dr. Giovanni Alfredo Guarneri	Membro
Prof. Dr. Gustavo Weber Denardin	Orientador
Profa. Dra. Viviane Dal Molin de Souza	Professor(a) responsável TCCII

AGRADECIMENTOS

Agradeço ao meu orientador Gustavo Weber Denardin, por me inspirar a realizar um projeto desse porte, por todo o suporte e apoio que me foi dado durante a elaboração desse trabalho e pelas suas aulas que me apresentaram a carreira à qual desejo seguir.

À todos os meus professores, que me ajudaram a chegar até a conclusão desse trabalho, da escola até a graduação, que me instigaram a ter esse amor pelo conhecimento e estudo.

À toda a minha família, que colaborou direta e indiretamente no meu caminho, aos meus pais, irmão e primos que me ajudaram nessa minha longa jornada pela UTFPR.

Aos vários colegas que fiz durante o curso, em especial à aqueles que me acolheram em momentos que precisei e faziam meus dias mais alegres, como o André, Felipe e Natanael, que tenho certeza que serão amizades que irei levar para fora da UTFPR. Além de outras amizades que fiz e que certamente lembrarei com muito carinho.

Technology is just nature we taught to do cool tricks. (EXURB1A, 2019)

RESUMO

CORRÊA, Gustavo L. Andrade. Sistema de atualização de *firmware Over-The-Air* para dispositivos de IoT baseado nas bibliotecas LwIP, Mbed TLS e FatFs. 2021. 62 f. Trabalho de Conclusão de Curso – DAINF — Departamento Acadêmico de Informática , Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

Com a ampla utilização de internet das coisas, conceito em que dispositivos embarcados estão conectados a internet, surge a necessidade da atualização automática do *firmware* desses dispositivos para correções ou aperfeiçoamentos. Atualmente existe uma grande variedade de implementações dessa funcionalidade, mas a falta de um padrão dificulta a sua ampla utilização. Assim, esse trabalho propõe uma solução de atualização de *firmware Over-The-Air* para dispositivos IoT, utilizando as bibliotecas amplamente difundidas LwIP, FatFs e Mbed TLS. O sistema proposto pretende disponibilizar uma API, que pode ser integrada a qualquer plataforma embarcada, que irá obter o novo *firmware* de um servidor, e um *bootloader* que faz todo o processo de troca do *firmware*.
Palavras-chave: Atualização. Firmware. Over-The-Air. Portável. IoT.

ABSTRACT

CORRÊA, Gustavo L. Andrade. Over-The-Air firmware update system for IoT devices based on LwIP, Mbed TLS and FatFs libraries. 2021. 62 f. Trabalho de Conclusão de Curso – Curso de Graduação em Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

With the widespread use of the internet of things, the concept in which embedded devices are connected to the internet, there is a need to automatically update the firmware of these devices for corrections or enhancements. There are currently a wide variety of implementations of this functionality, but the lack of a standard makes it difficult to use widely. Thus, this work offers an Over-The-Air firmware update solution for IoT devices, using as widespread libraries LwIP, FatFs and Mbed TLS. The proposed system intends to provide an API, which can be integrated into any embedded platform, which will obtain the new firmware from a server, and a bootloader that performs the entire firmware change process.

Keywords: Update. Firmware. Over-The-Air. Portable. IoT.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CAN	Controller Area Network
FAT	File Allocation Table
HAL	Hardware Abstraction Layer
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I2C	Inter-Integrated Circuit
IoT	Internet of things
IP	Internet Protocol
OTA	Over-The-Air
RAM	Random Access Memory
ROM	Read Only Memory
RTOS	Real-Time Operating System
SD	Secure Digital
SHA	Secure Hash Algorithm
SPI	Serial Peripheral Interface
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UART	Universal asynchronous Receiver-Transmitter
USB	Universal Serial Bus

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Com a evolução da microeletrônica e, por consequência, a redução de custo de periféricos e o crescimento do poder computacional de processadores, os sistemas computacionais se tornaram menores e baratos. Devido a isso, processadores e microcontroladores passaram a ser instalados em produtos, o que deu origem ao conceito de sistema embarcado, que são sistemas de processamento de informação embutidos em produtos (MARWEDEL, 2006). A utilização desses sistemas foi disseminada em várias áreas como, a automobilística, aeronáutica, ferroviária, industrial, médica, entre outras, automatizando as mais diversas funções.

Os sistemas computacionais embarcados são compostos pelos mesmos componentes utilizados para a constituição de computadores pessoais, porém com tamanhos, capacidades e custos reduzidos. Tais dispositivos operam de forma independente e geralmente são projetados para realizar tarefas específicas e repetitivas. Sistemas embarcados estão presentes no dia a dia da maioria das pessoas, em micro-ondas, geladeiras, TVs, aparelhos de som, video games e outros produtos eletrônicos (MARWEDEL, 2006), logo, esses dispositivos se distanciam dos computadores de propósito geral, como é observado em *desktops* e *notebooks* atuais.

Com a necessidade cada vez maior da implementação desses sistemas no cotidiano, é imprescindível se obter *hardwares* e *softwares*, cada vez mais robustos e que atendam todas as necessidades dos seus usuários. Assim, o projeto desses produtos devem ser muito bem planejado, e executado de forma a serem entregues produtos de qualidade, à prova de falhas e que possam reagir a erros, de forma a não causar danos a seus utilizadores.

Durante a fase de projeto de um sistema embarcado, deve-se avaliar diversos âmbitos, como desempenho, confiabilidade, consumo de energia, manufaturabilidade, etc. É também necessário validar essas avaliações, com o intuito de verificar se atenderão os requisitos de projeto. Pela necessidade desses produtos serem eficientes, é indispensável que esses sistemas passem por uma fase de otimização, em que mudanças no projeto podem melhorar a eficiência energética do produto ou até mesmo gerar novas funcionalidades a esses equipamentos. Portanto, o projeto todo precisa ser testado para evitar que erros e *bugs* possam vir a permanecer no produto final (MARWEDEL, 2006), criando um ciclo de desenvolvimento que deve ser repetido até se obter um produto eficiente, de qualidade e completo.

Após a venda e instalação de um sistema embarcado para seu cliente, eventualmente pode ser necessária uma nova funcionalidade, uma otimização ou então, podem ser exigidos testes nesse sistema. Logo, é preciso que haja uma forma de se alterar esse produto mesmo após seu lançamento, para assim dar um maior valor ao produto e confiabilidade ao sistema. A possibilidade de serem feitas manutenções futuras no *software*, que no contexto de sistemas embarcados é chamado de *firmware* (*firmware* é uma classe específica de *software* de computador que fornece controle de baixo nível para o *hardware* específico do dispositivo (NEAGU, 2021)), é conhecida como atualização OTA (*Over-The-Air*). Esse recurso não é obrigatório no projeto

de um sistema embarcado, mas é muitas vezes necessário, podendo ser uma funcionalidade muito útil dependendo da aplicação do sistema em concepção. A decisão de utilizar ou não a atualização OTA pode influenciar na escolha do *hardware* utilizado no projeto (BALL, 2002), podendo aumentar o custo do produto final. Uma das principais soluções adotadas para a manutenção desses programas é criar métodos de atualização em que, é necessário a presença de um agente humano fisicamente próximo do sistema para fazer a manutenção do *software*, o que acaba aumentando o custo de manutenção do produto e o tornando menos atrativo para os seus compradores.

Os *bootloaders* estão atualmente presentes em todos os computadores pessoais e em alguns sistemas embarcados. Esse *software* prepara a maioria dos *hardwares* presentes na máquina para um sistema operacional ou outro programa ser executado. Como é o primeiro programa a ser executado após um sistema ser iniciado ou após um *reset*, ele pode ter várias funções, como, realizar checagem de periféricos, verificar se o *firmware* presente na memória não está corrompido, além de poder fazer a troca do *software* presente na memória (DAVIS; DURLIN, 2013), que será sua principal utilização nesse trabalho.

Um dos seus principais usos é em *smartphones*, em que são utilizados para a atualização de sistemas operacionais como *Android* e iOS, e como garantia de restauração em caso de erros irreversíveis no sistema operacional. É desenvolvido pelo próprio fabricante do dispositivo, e por padrão é bloqueado para os usuários, evitando a substituição do *software* original do aparelho por uma versão customizada, mas ainda assim existem opções de desbloqueio do *bootloader*, dependendo do modelo do aparelho e do fabricante (SALUTES, 2018).

Internet of things ou IoT é o conceito que se refere à interconexão digital de objetos cotidianos com a internet. A internet das coisas em outras palavras pode ser descrita como uma rede de dispositivos embarcados, como sensores, câmeras, carros e demais objetos do cotidiano, capazes de obter e transmitir dados pela internet. Em 2019, a empresa de consultoria Gartner (GARTNER, INC., 2019) diz que, até 2020, são esperados mais de 20 bilhões de "coisas" conectadas a internet. Essas "coisas" não são dispositivos de uso geral como *smartphones* e PCs, mas objetos de função única.

Nesse cenário de crescimento constante de dispositivos embarcados conectados a internet, o presente trabalho de conclusão de curso propõe um método de manutenção desses *firmwares* de forma remota, que possa ser o mais portável possível para a família de microcontroladores STM32. Na solução proposta, o dispositivo embarcado poderá verificar periodicamente um servidor a procura de uma nova versão do seu *firmware*. Quando encontrado, será realizado o *download* do novo *firmware* para a memória interna do dispositivo, para posterior atualização do equipamento. O diferencial da abordagem proposta é basear a solução em bibliotecas amplamente difundidas em sistemas embarcados, como a LwIP (DUNKELS, 2002), Mbed TLS (DEVINE, 2006) e a FatFS(CHAN, 2016). Dessa forma, o código do sistema de atualização é totalmente portável, desde que a plataforma escolhida tenha suporte a tais bibliotecas. A única peça de *software* que não será totalmente portável será o *bootloader* que

substituirá o *firmware* antigo pelo novo na memória flash do dispositivo, por ser dependente do *hardware* utilizado.

1.1 OBJETIVO GERAL

Esse trabalho tem como objetivo geral o desenvolvimento de um sistema de atualização de *firmwares Over The Air* para dispositivos IoT baseado nas bibliotecas FatFs, LwIP e Mbed TLS.

1.2 OBJETIVOS ESPECÍFICOS

- Desenvolver o *bootloader* para microcontroladores STM32 utilizando a biblioteca FatFS.
- Implementar uma API que fará a comunicação segura entre o servidor e a plataforma embarcada, verificará a disponibilidade de atualização e fará o *download* da nova versão, se existente. Ainda, armazenará o *firmware* recebido em um cartão SD (Secure Digital).
- Comprovar o funcionamento da técnica de atualização remota de *firmware*, utilizando a plataforma embarcada STM32F746G-DISCOVERY.

2 REVISÃO DA LITERATURA

Neste capítulo será feita uma revisão da literatura necessária para o entendimento deste trabalho de conclusão de curso. Serão abordados os temas como: o processo de inicialização de um sistema embarcado, o que é um *bootloader* e o papel do *linker* na criação de um arquivo executável e no mapeamento da memória da aplicação. Também será discutido sobre a comunicação cliente-servidor, a pilha TCP/IP, a biblioteca LwIP e alguns protocolos implementados por ela, assim como a Mbed TLS, alguns protocolos e algoritmos de segurança disponíveis pela biblioteca. Será mostrado o que é um sistema de arquivos, o sistema de arquivos FAT e a biblioteca FatFs. Com o conhecimento obtido sobre todos esses temas o leitor será capaz de compreender como será desenvolvido o método de atualização de *firmware* OTA.

2.1 PROCESSO DE INICIALIZAÇÃO DO SISTEMA

Segundo Qing (2003), um processador embarcado, após ser ligado, busca e executa o programa a partir de um endereço pré-definido e gravado permanentemente na memória. A instrução contida nessa localização da memória é geralmente chamada de vetor de *reset*. O vetor de *reset* contém diversos ponteiros que apontam para diferentes regiões de memória e dependendo da posição desses ponteiros ele pode apontar para rotinas de inicialização, tratamento de interrupção entre outras rotinas. O ponteiro contido na região de inicialização do vetor de *reset* geralmente aponta para uma instrução de salto para outro espaço da memória em que a real rotina de inicialização se encontra. A razão deste salto para outra localidade da memória é para manter o vetor de *reset* pequeno. O vetor de *reset* pertence a uma pequena área da memória reservada pelo sistema por motivos especiais. O vetor de *reset*, assim como o código de inicialização do sistema, precisa estar armazenado permanentemente. Por causa deste inconveniente, a rotina de inicialização, chamada de programa *bootstrap*, reside na memória somente de leitura (ROM), na *flash* ou em outra memória não volátil. O termo *loader* se refere ao subprograma que é responsável por executar o *bootstrap*, fazer o possível *download* de um binário do *firmware* executável, também conhecido como imagem, de outro local e inicialização da aplicação final.

O conceito é melhor explicado por meio do exemplo criado por Qing (2003). Neste exemplo, é assumido que o *loader* foi desenvolvido e programado na memória *flash*. Além disso, será assumido que a imagem alvo contém várias seções de programa. Cada seção tem um lugar designado no mapa de memória. O vetor de *reset* está contido em uma pequena ROM, que está mapeada na localização 0x0h do espaço de endereços. A ROM contém alguns valores iniciais essenciais requeridos pelo processador quando o sistema é reinicializado (*reset*). Esses valores são o *reset vector*, o *stack pointer* (ponteiro de pilha) inicial e o endereço da memória de acesso randômico (RAM).

No exemplo ilustrado na Figura 1, mostra o processo de inicialização de um sistema linux, mas pode ser utilizado para um sistema embarcado, em que não necessariamente se copiaria a imagem da *flash* para a memória RAM. Como ilustrado o *reset vector* é uma instrução de salto para o endereço de memória 0x00040h; o *reset vector* transfere o controle do programa para a instrução neste endereço. O código de inicialização do sistema contém, entre outras coisas o programa *loader* da imagem destino e o vetor de exceção padrão do sistema (*exception vector*). O vetor de exceção do sistema aponta para uma instrução que reside na memória *flash*.

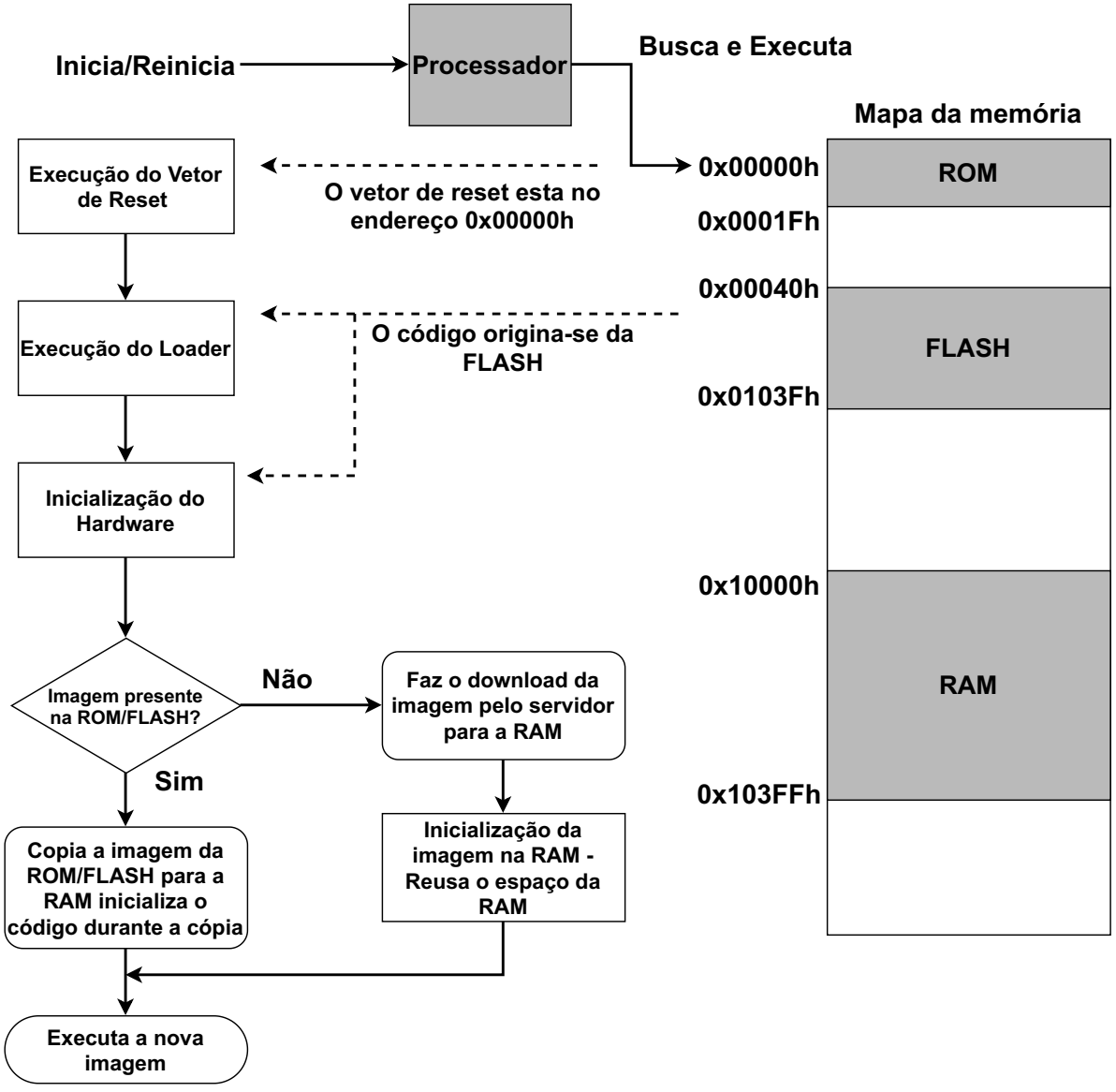

Figura 1 – Processo de inicialização de um sistema embarcado.
Fonte: adaptado de Qing (2003)

A primeira parte do processo de *bootstrap* do sistema é colocar o sistema em um estado conhecido. São posicionados valores padrões apropriados nos registradores do processador. São colocados no *stack pointer* os valores encontrados na ROM. O *loader* desabilita as

interrupções do sistema, pois o sistema ainda não está preparado para lidar com interrupções. O *loader* também inicializa a memória RAM e possivelmente a *cache* (memória transitória) do processador. Neste ponto, o *loader* executa um diagnóstico de *hardware* limitado nos dispositivos necessários para essas operações.

A execução do programa é mais rápida na RAM quando comparada ao mesmo código executado diretamente na memória *flash*. O *loader* pode opcionalmente copiar o código da memória *flash* para a RAM. Por causa dessa capacidade, uma seção de programa pode tanto ter um endereço de carregamento, quanto um endereço de execução. O endereço de carregamento é onde a seção do programa reside, enquanto o endereço de execução é o endereço em que o *loader* copia a seção do programa e a prepara para a execução (QING, 2003).

Uma imagem executável possui seções de dados inicializados e não inicializados. Essas seções são ambas legíveis e graváveis. Essas seções precisam residir na RAM, assim sendo são copiadas da memória *flash* para a RAM como parte do sistema de inicialização. A seção de dados inicializados (chamadas pelo *linker* de .data e .sdata) contém os valores iniciais para as variáveis globais e estáticas. O conteúdo dessa seção, portanto, faz parte da imagem executável final e é transferido completamente pelo *loader*. Por outro lado, o conteúdo da seção de dados não inicializado (chamado pelo *linker* de .bss e .sbss) é vazio. O *linker* reserva espaço para essa seção no mapa de memória. As informações de alocação dessas seções, como o tamanho da seção e o endereço de execução da seção, são parte do cabeçalho da seção. É trabalho do *loader* obter essas informações dos cabeçalhos de seção e alocar a mesma quantidade de memória na RAM durante o processo de carregamento. O *loader* coloca essas seções na RAM de acordo com o endereço de execução das seções.

Uma imagem executável provavelmente possui constantes. Os dados das contantes são parte da seção chamada pelo *linker* de .const, que é somente leitura. Sendo assim, é possível manter a seção .const na memória somente de leitura durante a execução do programa. Constantes de acesso frequente, como tabelas de *lookup*, necessitam ser transferidas para a RAM para melhorar o desempenho do sistema.

O próximo passo no processo de inicialização do sistema é o *loader* inicializar os dispositivos do sistema. Apenas os dispositivos necessários são inicializados nessa etapa. Em outras palavras, um dispositivo é inicializado na medida em que um subconjunto necessário dos recursos e recursos do dispositivo estejam ativados e operacionais. Geralmente, os dispositivos são parte da interface de entrada e saída do sistema, portanto, esses dispositivos são completamente iniciados quando existe a necessidade de se fazer *download* de uma imagem de outro local.

Agora o *loader* está pronto para transferir a imagem da aplicação para o sistema alvo. A imagem da aplicação pode conter um RTOS, um *kernel*, e os demais códigos das aplicações que o desenvolvedor necessita.

2.1.1 LINKER

Segundo Qing (2003), os arquivos de uma aplicação são processados pelo compilador e *assembler*. Criando assim os arquivos objetos, que contém os códigos de máquina binários (*machine binary code*) e dados de programa (*program data*). O utilitário de arquivo é um programa que concatena uma coleção de arquivos objetos para formar uma biblioteca. Então o *linker* obtém esses arquivos objetos como entrada e produz ou um arquivo executável, ou um arquivo objeto que pode ser utilizado em outro *linker* com outros arquivos objetos. O arquivo de comandos de *linker* (*linker command file*) orienta o *linker* em como combinar esses diferentes arquivos objetos e em que local da memória colocar o código binário e os dados no sistema embarcado alvo. Em conclusão, a função principal de um *linker* é combinar múltiplos arquivos objetos em um arquivo objeto maior, um arquivo objeto compartilhado ou uma imagem executável final. Esse processo pode ser observado na Figura 2.

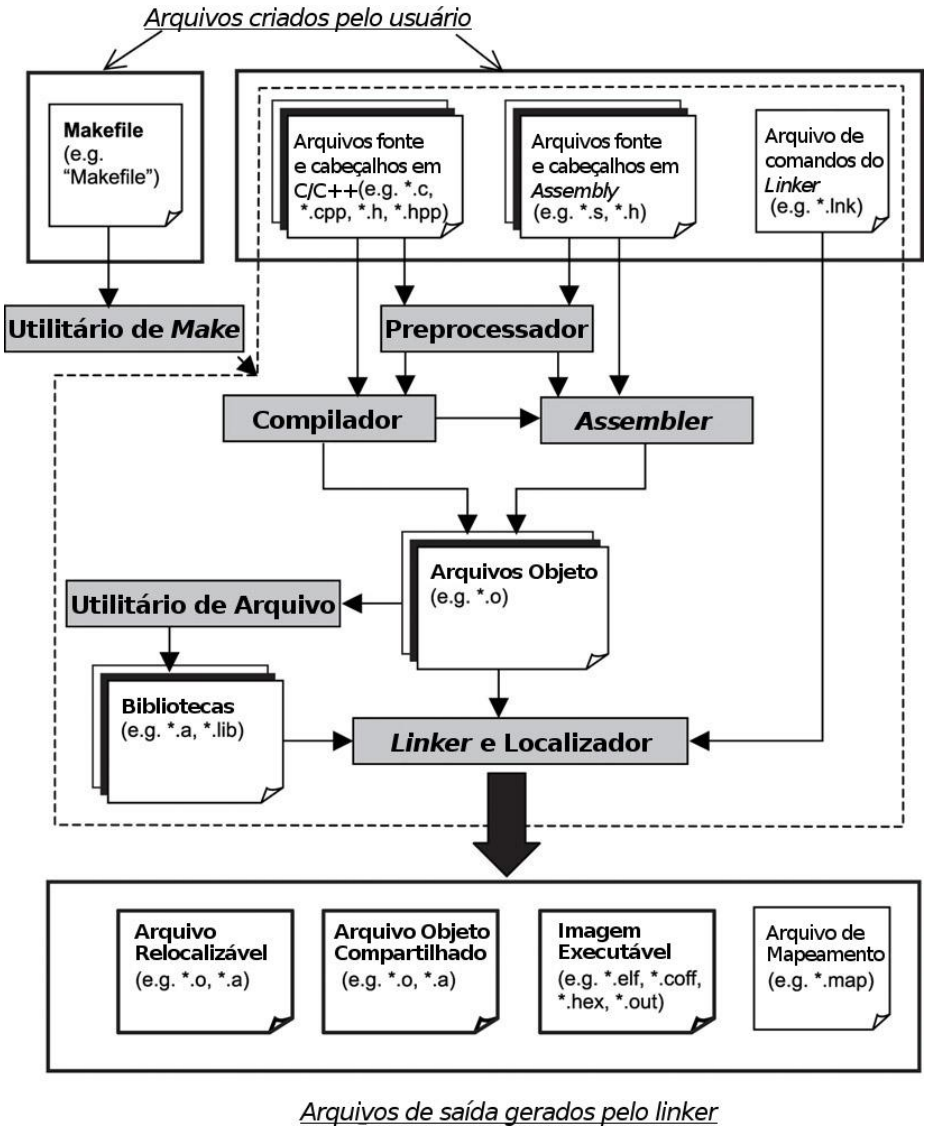

Figura 2 – Criando um arquivo de imagem para um sistema.
Fonte: adaptado de Qing (2003)

O *linker* precisa combinar esses arquivos objetos e fundir as seções de diferentes arquivos em um segmento de programa. Esse processo cria uma única imagem executável para o sistema embarcado alvo. O desenvolvedor utiliza comandos de *linker* (chamados de *linker directives*) para controlar como o *linker* combina essas seções e aloca seus segmentos no sistema alvo. As diretivas de *linker* ficam contidas no arquivo de comando de *linker*. O objetivo de criar esse arquivo de comando de *linker* é para que o desenvolvedor de sistemas embarcados possa mapear a imagem executável para o *hardware* alvo de forma precisa e eficiente.

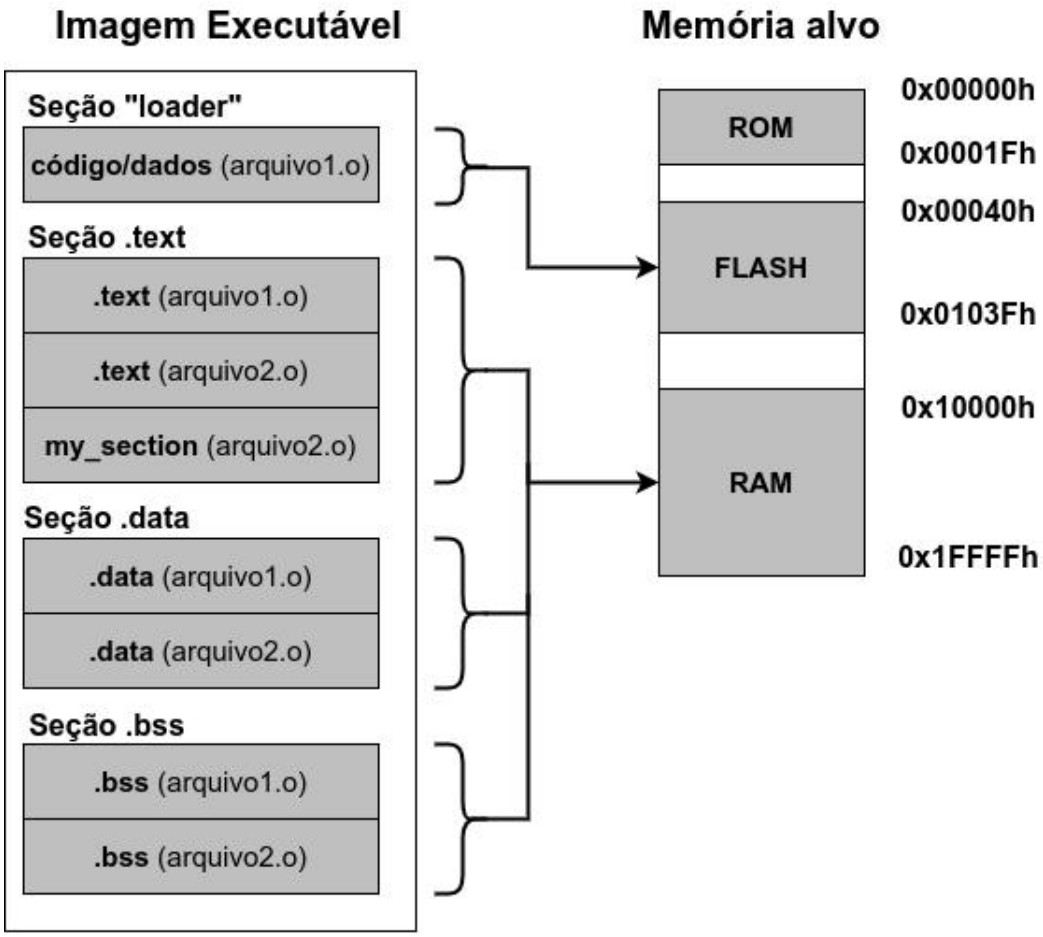

Figura 3 – Mapeando uma imagem executável em um sistema alvo.
Fonte: adaptado de Qing (2003)

2.1.2 BOOTLOADER

O *bootloader* é um *software* que tem como responsabilidade a atualização do *firmware* do sistema, operação também conhecida como *in-aplicattion programing* (IAP). Reside em uma área protegida da memória, geralmente colocado no início da *flash* ou na ROM, e é o primeiro *software* a ser executado após o *reset* ou iniciação do sistema. É desenvolvido para receber comandos via periféricos de comunicação como: UART, I2C, SPI, CAN e Ethernet, e entender o mapa de memória do microcontrolador (DAVIS; DURLIN, 2013). A Figura 4 mostra como geralmente fica alocado um *bootloader* e o *firmware* na memória.

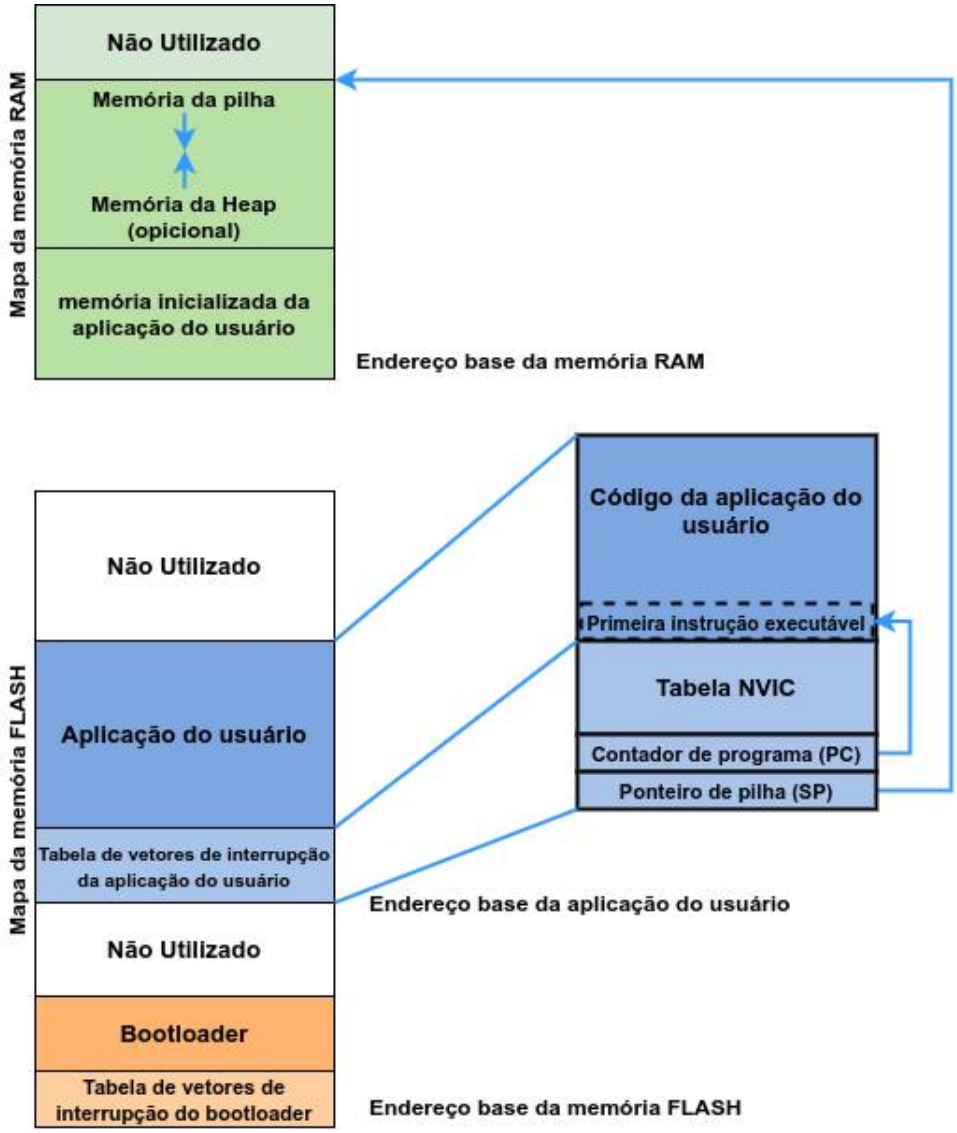

Figura 4 – Alocação do *bootloader* e *firmware* nas memórias *flash* e RAM.
Fonte: Adaptado de Davis e Durlin (2013)

A função do *bootloader* se resume geralmente a: comunicar-se com outro servidor, ou por comunicação serial, USB, cartão SD, entre outros meios, e ler os arquivos enviados pelo *host*, atualizar o *firmware* de seu microcontrolador, e iniciar esse novo programa. Pode conter instruções e comandos definidos pelo projetista para somente o circuito integrado em uso, impossibilitando a utilização do mesmo código em outras plataformas. Portanto, é um programa que não é portável para vários modelos de sistemas embarcados. A Figura 5 mostra o funcionamento de um *bootloader* padrão.

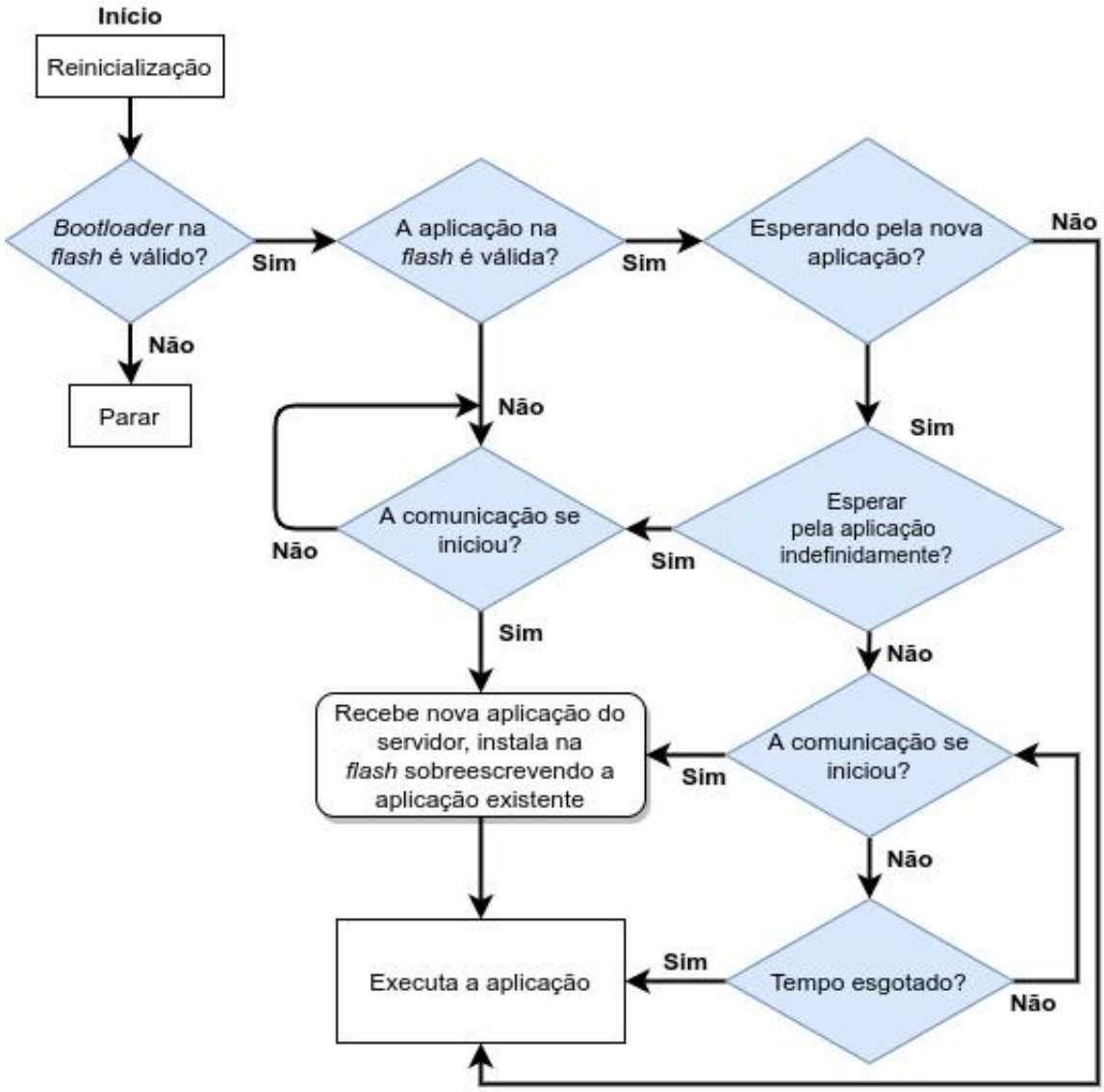

Figura 5 – Fluxograma de operações de um *bootloader*.
Fonte: Adaptado de Davis e Durlin (2013)

Como é possível observar na Figura 5, após a inicialização ou reinicialização de um sistema, o microcontrolador verifica se o *bootloader* presente na memória *flash* é válido, caso ele seja, é executado e o *bootloader* observa se a aplicação é válida, caso ela não seja válida, o *bootloader* fica aguardando em um *loop* que verifica se já houve uma comunicação com o servidor. Caso a aplicação seja válida, o *bootloader* deve decidir se deve esperar por uma nova aplicação e se essa espera deve ser por um tempo indefinido. Se for necessário que se espere indefinidamente pela aplicação, o *bootloader* fica aguardando no *loop* citado anteriormente, e caso não seja necessário se esperar indefinidamente, o *bootloader* fica verificando se a comunicação se iniciou, e se o tempo de espera estipulado foi esgotado. Se o tempo for esgotado ele inicializa a aplicação não atualizada. Caso essa comunicação se inicie, o *bootloader* recebe esse nova aplicação do servidor e a instala na *flash* sobrescrevendo a aplicação anterior e após isso executa essa nova aplicação.

Os microcontroladores da família STM32 desenvolvidos pela empresa ST STMicroelectronics (2019) possuem um *bootloader* pré-programado na ROM desde sua fabricação. Esse *bootloader* pode utilizar diversos periféricos de comunicação, e para cada periférico diferente a ST padronizou diferentes protocolos que permitem várias operações, como: obter o ID do chip, escrever e ler *bytes* na RAM e memória *flash*, apagar setores das memórias, ativar áreas de proteção na memória e pular para o código principal do sistema (NOVIELLO, 2018). Como esse *bootloader* é muito dependente do microcontrolador, ele não foi utilizado neste trabalho.

Um *bootloader* customizado pode conter diversas funções adicionais, uma função frequentemente usada é o uso do *bootloader* para descriptografar *firmwares* que podem chegar via internet, para se garantir a segurança e origem do *firmware*.

2.2 COMUNICAÇÃO CLIENTE-SERVIDOR

Um programa *cliente* é um programa que funciona em um sistema computacional, que solicita e recebe um serviço de um programa servidor, que funciona em outro sistema final. Uma vez que o programa cliente é executado em um computador e o programa servidor, é executado em outro, aplicações cliente-servidor são, por definição, aplicações distribuídas. Os programas cliente e o servidor interagem enviando mensagens um para o outro, pela internet ou qualquer outra rede local ou remota. Neste nível de abstração, os roteadores, enlaces e outros componentes da internet funcionam como uma caixa-preta que transferem mensagens entre os componentes distribuídos, comunicantes, de uma aplicação (KUROSE; ROSS, 2010).

A comunicação cliente-servidor na internet é feita por meio de diversos protocolos de rede, cujo conjunto de protocolos é conhecido como pilha TCP/IP (*Transmission Control Protocol/Internet Protocol*). Essa pilha é dividida em quatro camadas, em que cada camada é encarregada de realizar uma série de funções, concedendo um grupo de serviços bem definidos para o protocolo da camada superior. A Figura 6 ilustra a pilha TPC/IP e seus protocolos.

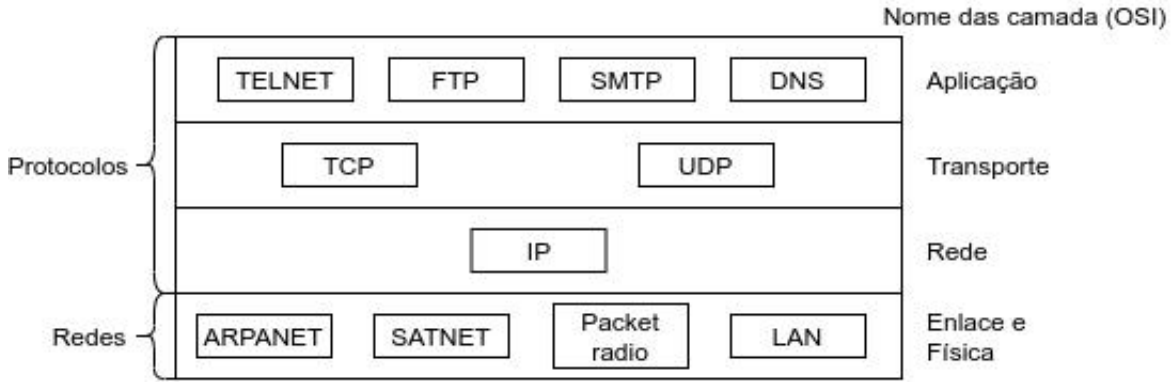

Figura 6 – Pilha TCP/IP e seus protocolos.
Fonte: adaptado de Tanenbaum (2003)

Segundo Kurose e Ross (2010), essas camadas são denominadas:

- Camada de aplicação: em que residem os protocolos de nível mais alto. Um protocolo da camada de aplicação é distribuído por diversos sistemas finais, sendo que a aplicação em um sistema usa o protocolo para trocar pacotes de informação com a aplicação em outro sistema. Exemplos de protocolos:
  - HTTP (*Hypertext transfer protocol*).
  - SMTP (*Simple Mail Transfer Protocol*).
  - FTP (*File Transfer protocol*).
  - DNS (*Domain Name System*).
- Camada de transporte: em que reside os protocolos que fazem o transporte de dados da camada de aplicação, transportando mensagens entre os lados do cliente e do servidor de uma aplicação. Exemplos de protocolos:
  - TCP (*Transmission Control Protocol*).
  - UDP (*User Datagram Protocol*).
- Camada de rede: camada que contém o protocolo responsável pela movimentação, de uma máquina para outra, de pacotes de camada de rede conhecidos como datagramas. Um protocolo da camada de transporte passa um segmento TCP ou UDP e um endereço de destino à camada de rede. A camada de rede prove o serviço de entrega do segmento à camada de transporte da máquina destinatária. O protocolo da camada de rede é chamado de IP (*Internet Protocol*).
- Camada de enlace: é a camada que contém os protocolos que ficam responsáveis por enviar datagramas de um nó a outro, ou seja, faz o transporte do datagrama entre elementos da rede, como de uma máquina ao roteador, ou de roteador para roteador. Exemplos de protocolos:
  - Arpanet.
  - LAN (*local area network*).
  - WLAN (*wireless local area network*).

Com a popularização da internet essa comunicação se tornou cada vez mais comum, atingindo bilhões de usuários no mundo todo. Assim a comunicação cliente-servidor precisa ser segura. A segurança de redes se preocupa em garantir que pessoas mal-intencionadas não leiam ou modifiquem secretamente mensagens enviadas a outro destinatário. Ela também é responsavel por identificar se uma mensagem recebida é verdadeira e tem uma origem confiável (TANENBAUM, 2003).

2.2.1 LWIP

A Biblioteca LwIP é uma implementação da pilha TCP/IP, focada em ser pequena e portável, reduzindo a utilização de recursos como memória RAM e ainda tendo um TCP completo, se tornando adequada para sistemas embarcados. Foi originalmente desenvolvida por Adam Dunkels nos laboratórios da *Computer and Networks Architectures* (CNA), no Instituto

Sueco de Ciência da Computação (SICS) e agora é desenvolvida e mantida por uma rede mundial de desenvolvedores (DUNKELS, 2002). Possui três *Aplication Programing Interfaces* (APIs):

- RAW API (API básica): É a API nativa do LwIP, possui melhor desempenho e o menor tamanho de código, porém torna o desenvolvimento de aplicações mais complexo.
- Netconn API: É uma API sequencial de alto nível que requer um sistema operacional de tempo real (RTOS). Habilita operações com múltiplas *threads*.
- BSD *sockets* API: API de *sockets* de Berkeley, desenvolvida em cima da API Netconn. Essas API's implementam diversos protocolos de rede, incluindo:
  - HTTP permite a obtenção de recursos, tais como documentos HTML, imagens, scripts e outros tipos de arquivos.
  - FTP para o envio e recebimento de arquivos.
  - SMTP para o envio de mensagens de correio eletrônico através da internet.
  - ICMP (*Internet Control Message Protocol*) para manutenção e *debugging* da rede.
  - TCP com controle de congestionamento, estimativa de latência, recuperação e retransmissão rápida.
  - IP incluindo o envio de pacotes para múltiplas interfaces de rede.

Segundo Kurose e Ross (2010), o *Hypertext Transfer Protocol* (HTTP) define a estrutura dessas mensagens assim como o modo como o cliente e o servidor as trocam. O HTTP define como clientes requisitam documentos aos servidores e como eles os transferem ao cliente. Ele utiliza o TCP como seu protocolo de transporte subjacente. O cliente HTTP primeiramente inicia uma conexão TCP com o servidor. Após essa conexão ser estabelecida, os processos da aplicação e do servidor acessam o TCP por meio de sua interface de *sockets*. No lado do cliente a interface de socket é uma porta entre o processo cliente e a conexão TCP. No lado do servidor, ela é uma porta entre o processo servidor e a conexão TCP.

O cliente envia mensagens de requisição HTTP para sua interface de socket e recebe uma mensagem de resposta HTTP de sua interface de socket. De uma maneira parecida acontece do lado do servidor, onde ele recebe mensagens de requisição HTTP de sua interface de socket e envia mensagens respostas a sua interface. Assim a mensagem sai da camada de aplicação e passa para a camada de transporte.

Segundo Mozilla Developer Network (2021), alguns exemplos de requisições são:

- GET: requisição que solicita a apresentação de um recurso especifico, e deve sempre retornar dados.
- POST: requisição utilizada para submeter uma entidade a um recurso especifico, modificando assim o recurso.
- PUT: requisição que substitui todas as atuais representações do recurso de destino pelos dados da requisição.
- DELETE: requisição que remove recurso especifico.

Segundo Tanenbaum (2003), o *Transmission Control Protocol* (TCP) foi projetado especificamente para oferecer um fluxo de *bytes* fim-a-fim confiável em uma inter-rede não-confiável. Uma inter-rede é diferente de uma única rede porque suas diversas partes podem ter topologias, larguras de banda, retardos, tamanhos de pacotes e outros parâmetros totalmente diferentes. O TCP foi projetado para se adaptar dinamicamente às propriedades da inter-rede e ser robusto diante de muitas categorias de falhas que podem ocorrer.

Cada sistema computacional compatível com TCP tem uma entidade de transporte TCP, que pode ser um procedimento de biblioteca, um processo do usuário ou parte do núcleo. Em todos os casos, ele gerencia fluxos e interfaces TCP para a camada IP. Uma entidade TCP aceita fluxos de dados de usuários provenientes de processos locais, divide-os em partes de no máximo 64 kB e envia cada parte em um datagrama IP distinto. Quando os datagramas IP que contém dados TCP chegam a um sistema computacional, eles são enviados à entidade TCP, que restaura o fluxo de *bytes* originais.

A camada IP não oferece garantia que os datagramas serão entregues de forma apropriada, portanto, cabe ao TCP administrar os *timers* e retransmiti-los sempre que necessário. Os datagramas também podem chegar fora de ordem, o TCP também terá que os reorganizar em mensagens na sequência correta.

2.2.2 MBED TLS

A Mbed TLS provê a implementação da camada de segurança para a comunicação com o servidor, que pode evitar que uma versão maliciosa do *software* seja recebida de uma fonte não confiável. Também fornece peças de *software* contendo algoritmos criptográficos, que podem ser facilmente acoplados a qualquer aplicação. Como é o caso do algoritmo SHA-256 que é responsável por fazer a verificação da integridade dos arquivos baixados.

A biblioteca Mbed TLS foi desenvolvida para se integrar facilmente a aplicações embarcadas existentes, e fornecer os blocos de construção para uma comunicação segura, criptografia e gerenciamento de chaves. Como o seu intuito é ser o mais flexível possível, permite que sejam integrados ao sistema somente as funcionalidades necessárias, diminuindo assim o tamanho total que a biblioteca ocuparia no sistema (DEVINE, 2006).

A Figura 7 ilustra como a biblioteca cria uma camada intermediária entre a aplicação final e a camada TCP/IP, chamada de TLS (*Transport Layer Security*). A Mbed TLS pode ser usada para criar um servidor e cliente SSL (*Secure Sockets Layer*)/TLS, fornecendo uma estrutura para a configurar e se comunicar por meio de um canal de comunicação SSL/TLS. A camada TLS criada depende diretamente dos módulos de análise de certificado, criptografia simétrica ou assimétrica e *hash* da biblioteca utilizada.

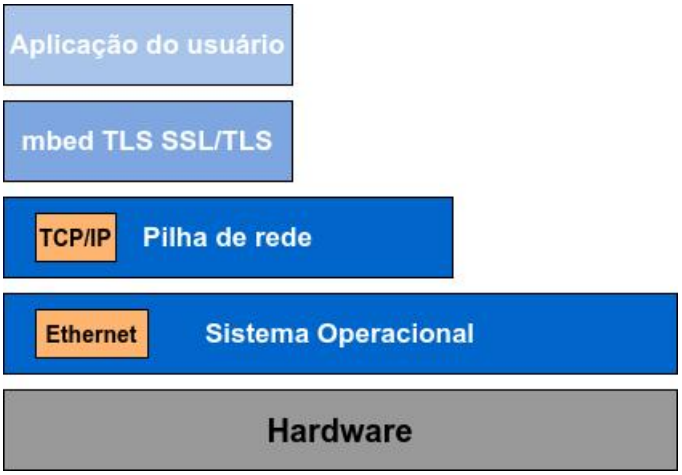

Figura 7 – Pilha de comunicação.
Fonte: adaptado de Devine (2006)

2.2.2.1 Transport Layer Security (TLS)

A *Transport Layer Security* assim como seu precursor *Secure Sockets Layer*, é um protocolo de segurança projetado para fornecer segurança na comunicação sobre uma rede de computadores. Segundo Dierks e Rescorla (2008), o protocolo TLS visa fornecer privacidade e integridade de dados entre aplicações que se comunicam. Quando uma rede está protegida por TLS, conexões entre um cliente e um servidor devem ter uma ou mais das seguintes propriedades:

- A conexão é privada, pois, utilizada criptografia simétrica para criptografar os dados transmitidos. As chaves para essa criptografia são geradas exclusivamente para cada conexão e são baseadas em um segredo compartilhado que foi negociado no início da sessão (conhecido como *Handshake Protocol*). No protocolo de *Handshake*, o servidor e o cliente negociam qual algoritmo de criptografia e chaves criptográficas usar antes que o primeiro dado seja transmitido. Como a negociação ocorre somente no início da transmissão, qualquer invasor que intercepte a transmissão não poderá decifrar as mensagens, enviar dados e alterar os termos dessa negociação. Então a negociação de um segredo compartilhado é segura e confiável.
- A conexão é confiável, pois cada mensagem transmitida inclui uma verificação de integridade de mensagem, utilizando um código de autenticação de mensagem, como uma *hash* criptográfica, para evitar perda não detectada ou alteração dos dados durante a transmissão.

Uma vantagem do TLS é que ele é independente do protocolo da aplicação. No entanto, ele não especifica como os protocolos adicionam segurança ao TLS, as decisões sobre como iniciar o *handshaking* e como interpretar os certificados de autenticação trocados são deixadas ao critério dos projetistas e desenvolvedores de protocolos executados sobre o TLS.

2.2.2.2 Função Hash

Segundo Kurose e Ross (2010), uma função *hash* é um algoritmo que recebe uma entrada, m, e computa uma cadeia de bits de tamanho fixo $H(m)$ conhecida como *hash*. Uma função de *hash* criptográfica deve apresentar a seguinte propriedade: no processamento, é impraticável encontrar duas mensagens diferentes x e y em que $H(x) = H(y)$.

A SHA-1 (Secure *Hash* Algorithm) é um conjunto de funções *hash* criptográficas projetadas pela NSA (NSA, 1952). Esse algoritmo, de forma resumida, se baseia em processar um resumo de mensagem de 160 bits por meio de um processo de quatro etapas, formado por uma etapa de enchimento (Adicionando 'uns' seguidos de 'zeros' suficientes, de maneira em que o comprimento da mensagem satisfaça determinados critérios), uma etapa de anexação (anexação de uma representação de alguns bits do comprimento da mensagem antes do enchimento), uma etapa de inicialização de um acumulador e uma etapa final iterativa, em que os blocos de palavras da mensagem são processados (misturados) em quatro rodadas de processamento.

Comparando o *hash* computado (a saída de execução do algoritmo) a um valor de *hash* conhecido e esperado, pode-se determinar a integridade dos dados. Por exemplo, calcular o *hash* de um arquivo baixado e comparar o resultado com um *hash* conhecido, pode comprovar que o arquivo foi modificado ou adulterado.

SHA-2 é um conjunto de funções *hash* criptográficas que contém mudanças significativas de seu antecessor. É composta por seis funções *hash* com valores de *hash* que são de 224, 256, 384 ou 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

2.3 SISTEMAS DE ARQUIVO

Segundo Tanenbaum (2007), para resolver diversos problemas como o armazenamento de grandes quantidades de dados, a perda de dados após o fim da execução do processo que os criou e a necessidade de tornar esses dados independentes de quaisquer processos. Foi necessário criar uma estrutura de armazenamento de informações a longo prazo. Os três requisitos fundamentais para essa estrutura são:

- Deve permitir armazenar um volume grande de informações;
- Os dados devem sobreviver ao término do processo que os estão utilizando;
- Vários processos devem ser capazes de acessar os dados concomitantemente.

Essa estrutura é chamada de arquivo, e é vastamente utilizada por diversos sistemas. Os arquivos são utilizados para armazenar dados em discos e outras mídias externas. Então os processos podem ler e escrever novos dados quando necessário. As informações armazenadas em arquivos devem ser persistentes, logo, não devem ser afetadas pela criação e pelo término do processo. Um arquivo só deve desaparecer quando o seu criador o apagar (TANENBAUM, 2007).

O modo como os arquivos são estruturados, nomeados, acessados, usados, protegidos e implementados são definidos geralmente pelo sistema operacional. A parte do sistema operacional responsável por esse gerenciamento é chamada de sistema de arquivos. Os arquivos podem, no caso de sistemas embarcados, ser gerenciados por API's que criam uma camada independente do sistema operacional e gerenciam os arquivos, como é o caso da biblioteca FatFs.

Existem diversas formas de implementar um sistema de arquivo. Nessa implementação é necessário saber como os arquivos e seus diretórios são armazenados, como o espaço em disco é gerenciado e em como fazer tudo funcionar de modo eficiente e confiável. Um dos sistemas de arquivos padrões para o uso em memórias que são divididas em bloco é o FAT32. O sistema FAT é aplicado em dispositivos de armazenamento como cartões SD. Um cartão SD é um dispositivo de memória não volátil criado pela SD Card Association (SD CARD ASSOCIATION, 2016), que possui sua memória dividida em blocos e que por padrão faz uso do sistema de arquivo FAT.

2.3.1 SISTEMA DE ARQUIVO FAT

Segundo Tanenbaum (2007), o sistema de arquivo FAT (*File Allocation Table*) é implementado por meio de uma alocação de memória encadeada usando uma tabela na memória. Nessa organização, o bloco de memória inteiro está disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Mesmo que o encadeamento ainda tenha que ser seguido para encontrar determinado deslocamento dentro do arquivo, ele está inteiramente na memória, de modo que, pode ser seguido sem necessidade nenhuma de referenciar o disco.

A Figura 8 ilustra como é a tabela, mostrando que o arquivo A inicia-se no bloco 4 e segue o encadeamento até o seu fim, assim como o arquivo B que se inicia no bloco 6. Ambos terminam com um marcador especial que no caso é o número -1.

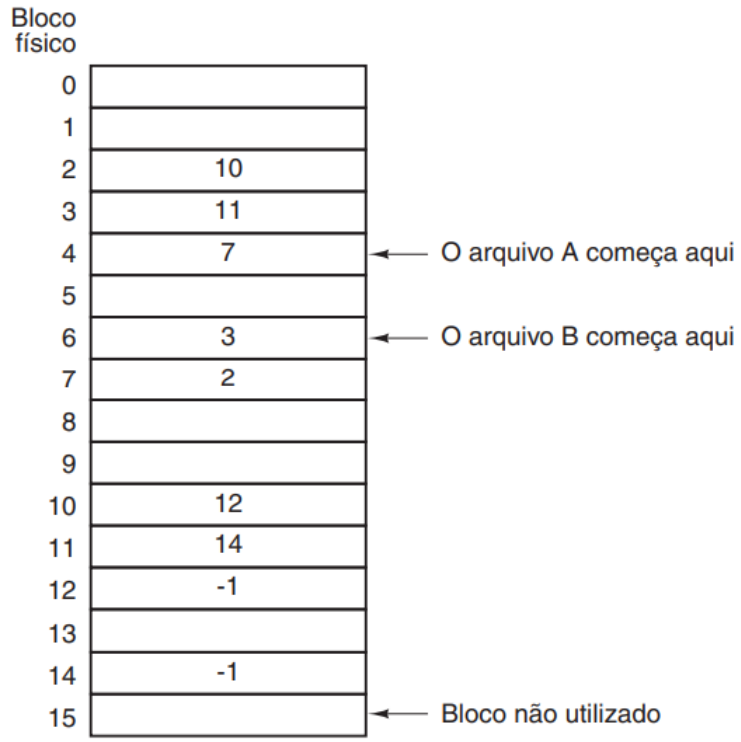

Figura 8 – Alocação encadeada usando tabela de alocação de arquivo.
Fonte:Tanenbaum (2007)

A principal desvantagem deste método é que a tabela inteira precisa estar na memória o tempo todo. Com um disco de 20 GB e um tamanho de bloco de 1 KB, a tabela precisa de 20 milhões de entradas, uma para cada um dos 20 milhões de blocos do disco. Cada entrada deve ter no mínimo 3 *bytes* para manter o endereço dos blocos. Para facilitar sua pesquisa, as entradas acabam ocupando 4 *bytes*. Assim, a tabela ocupará 60 MB ou 80 MB de memória principal o tempo todo.

Existe também o sistema de arquivo exFAT que é utilizado para mídias com capacidade de armazenamento maiores que 4 GB, que foi adotada como sistema de arquivo padrão para cartões de memória (SD card) maiores de 4 GB, pela SD Card Association (SD CARD ASSOCIATION, 2016).

2.3.2 FATFS

FatFs é um módulo genérico de um sistema de arquivo FAT/exFAT, para pequenos sistemas embarcados com recursos computacionais reduzidos. É escrito em conformidade com a ANSI C (C89) e é completamente separado da camada de entrada e saída do sistema, portanto, é independente da plataforma utilizada. É um *software* livre de código aberto e proporciona a leitura, escrita, criação e remoção de arquivos, além do gerenciamento e navegação de diretórios. A Figura 9 ilustra como o módulo é independente da aplicação e da plataforma utilizada.

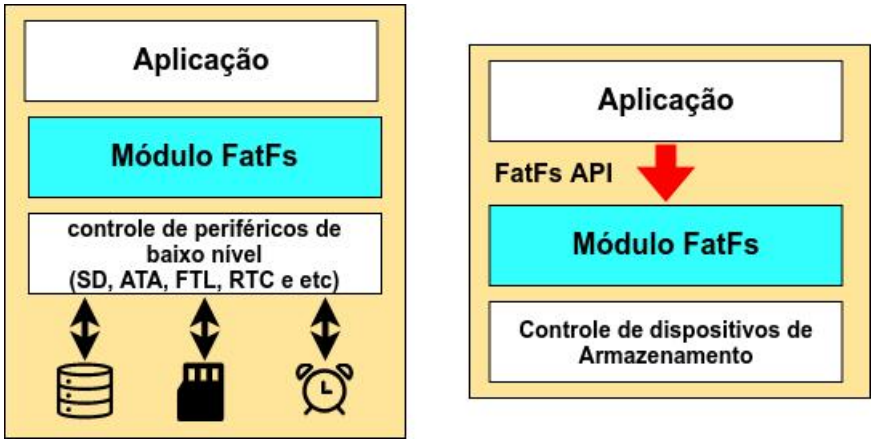

Figura 9 – Posição da biblioteca FatFs na aplicação.
Fonte: Adaptado de Chan (2016)

A FatFs fornece várias funções do sistema de arquivos para a aplicação. Assim pela aplicação é possível gerenciar os arquivos e diretórios, como é ilustrado da Figura 9. As principais funções fornecidas pela FatFs para a aplicação são (CHAN, 2016):

- Acesso a arquivos.
  - f_open - Abre/Cria um arquivo.
  - f_close - Fecha um arquivo aberto.
  - f_read - Lê os dados de um arquivo.
  - f_write - Escreve dados em um arquivo.
- Acesso a diretórios.
  - f_opendir - Abre um diretório.
  - f_closedir - Fecha um diretório aberto.
- Gerenciamento de arquivos e diretórios.
  - f_stat - Verifica a existência de um arquivo ou diretório.
  - f_unlink - Remove um arquivo ou diretório.
  - f_rename - Renomeia ou move um arquivo, ou diretório.
  - f_mkdir - Cria um diretório.
  - f_chdir - Muda o diretório atual.
- Gerenciamento de volume e configurações do sistema.
  - f_mount - Registra ou remove registro da área de trabalho da partição.
  - f_mkfs - Cria uma partição FAT na unidade lógica.
  - f_fdisk - Cria uma partição na unidade física.
  - f_getfree - Obtém o espaço livre da partição.

2.4 HARDWARE ABSTRACTION LAYER (HAL)

A camada de abstração de *hardware* ou HAL (*Hardware Abstraction Layer*) é uma divisão lógica de código que serve como camada de abstração entre o *hardware* e o *software*

de um sistema computacional. Ela prove uma interface de *drivers* dos periféricos do microcontrolador com o programa, permitindo assim uma comunicação mais fácil entre *hardware* e *software*.

O principal objetivo dessa camada é permitir que diferentes arquiteturas de *hardwares* funcionem com o mesmo *software*, para isso fazem uso de uma interface uniforme dos periféricos do sistema, mantendo sempre as mesmas chamadas de funções e de periférico desde a sua inicialização até sua utilização.

O HAL está incluído em diversos sistemas operacionais para evitar que se modifiquem o *Kernel* para que ele possa ser executado em diferentes arquiteturas com diferentes conjuntos de *hardwares*. Um computador pode possuir uma camada de abstração de *hardware* dentro do *kernel* de seu sistema operacional, ou em forma de *drivers* que garantem uma interface consistente para a aplicação interagir com os periféricos fornecidos pelo *hardware* disponível.

Segundo Techopedia (2021), as vantagens que a camada de abstração de *hardware* fornece são:

- Permite que aplicações extraiam o máximo de desempenho possível dos *hardwares*.
- Possibilitar que os sistemas operacionais e diversos *software* funcionem em diferentes arquiteturas de *hardware*.
- Permite que *driver* de dispositivos forneçam acesso direto a periféricos do *hardware*, o que permite que programas sejam independentes dos dispositivos.
- Facilita assim a portabilidade de um *software*.

2.4.1 STM32CUBE HAL

Segundo a STMicroelectronics (2021), o *software* STM32Cube é uma iniciativa da STMicroelectronics para melhorar significativamente a produtividade do desenvolvedor, reduzindo o esforço, o tempo e o custo de desenvolvimento, e ele cobre toda a linha de produtos da STM32.

É um recurso de desenvolvimento que é composto por um *software* que permite a geração de códigos em C iniciais de um novo projeto. Nesta ferramenta é possível já fazer as configurações iniciais de periféricos, bibliotecas, configurações de *clock* entre outras definições necessárias. Todos esses ajustes são feitos a partir de uma interface gráfica facilitando o entendimento e tornando ainda mais rápido o desenvolvimento. A interface do STM32Cube pode ser vista na Figura 10.

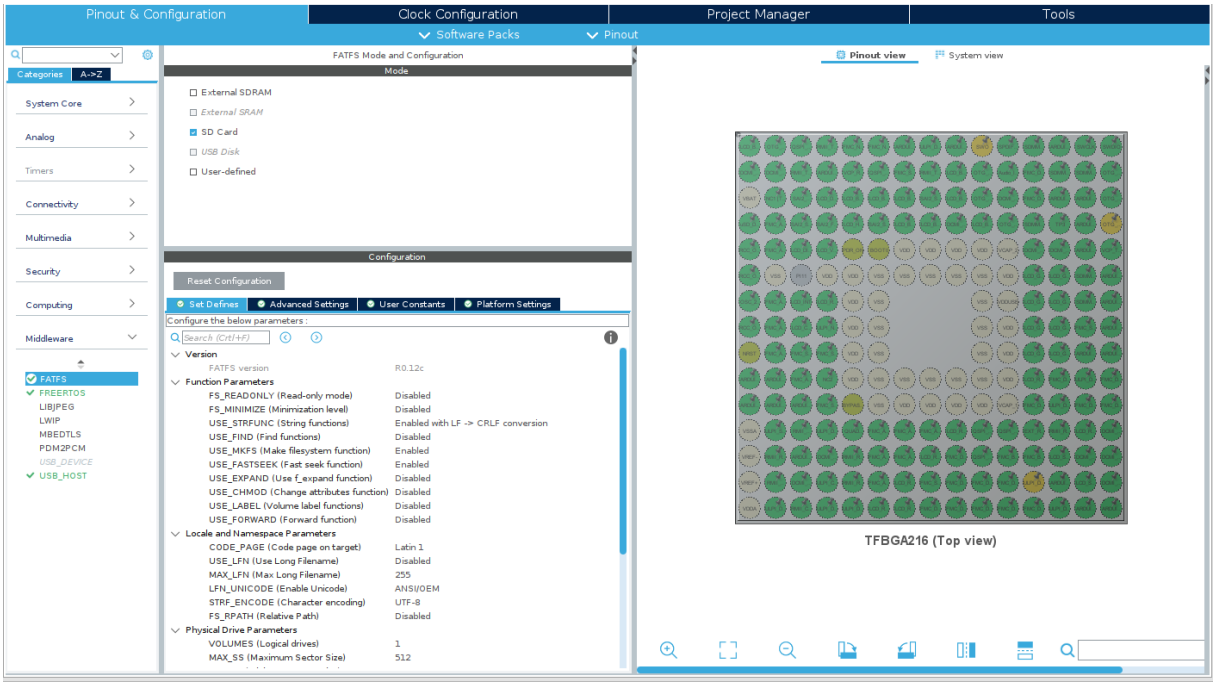

Figura 10 – Interface do stm32cube.
Fonte: autoria própria.

Segundo a STMicroelectronics (2021), as APIs do driver HAL são divididas em duas categorias: APIs genéricas, que fornecem funções comuns e genéricas para todas as séries STM32 e APIs estendidas, que incluem funções específicas e personalizadas para uma determinada família ou microcontrolador especifico. Os *drivers* HAL incluem um conjunto completo de APIs prontas para uso que simplificam a implementação do aplicativo do usuário. Por exemplo, os periféricos de comunicação contêm APIs para inicialização e configuração, gerenciar transferências de dados, lidar com interrupções ou acesso direto a memória e gerenciar erros de comunicação.

2.5 TRABALHOS CORRELATOS

Durante a elaboração desse trabalho foram identificados alguns trabalhos que produziam alguns métodos de atualização OTA para diversas aplicações, que serão explanados a seguir:

- **Firmware over the air for automotive, Fotamotive**: Esse trabalho introduz uma solução para atualização de veículos com a ajuda de fabricantes de equipamentos automotivos para reduzir custos com *recalls* e assim aumentar a qualidade de seus produtos, facilitar as atualizações e controlar melhor a frota de veículos no mercado. Essa solução consiste em atualizar a unidade de controle do motor por meio de métodos OTA, sem a necessidade de uma conexão física com o veículo (ODAT; Ganesan, 2014).

- **Firmware over the air for home cybersecurity in the Internet of Things**: Esse trabalho descreve a utilização de um método de atualização de *firmware* para roteadores caseiros, utilizando sistemas de gerenciamento de rede e de suporte de operações de fornecedores de acesso à *internet* (TENG et al., 2017).
- **Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development**: Esse trabalho introduz um novo sistema de atualização de *firmware Over-The-Air* (OTA) baseado no protocolo de rede *Lightweight mesh*, que provê descoberta de rotas, estabelecimento de comunicação e um protocolo de malha de baixo consumo de energia (CHANDRA et al., 2016).

Como observado, todos os trabalhos correlatos tem um objetivo único e diferente para a aplicação de sua atualização OTA, enquanto esse trabalho tem como meta produzir um sistema que pode abranger diferentes aplicações.

3 SISTEMA DE ATUALIZAÇÃO DE FIRMWARE OVER-THE-AIR

Nesse capítulo é retratado como é o funcionamento do sistema que será desenvolvido, mostrando inicialmente uma visão geral do projeto, como o conjunto de *firmwares* (OTA e *bootloader*) funcionam em conjunto para executar a atualização da aplicação. São listadas as funcionalidades de cada uma das partes do *software*, explicando suas atividades e a forma com que foram implementadas, como cada biblioteca foi utilizada nestas aplicações. Aqui também são apresentados quais os materiais utilizados, e quais os *firmwares* criados para a prova de que o sistema funciona, e uma explicação de como utilizar o sistema desenvolvido neste trabalho em outras plataformas embarcadas da família de microcontroladores STM32.

3.1 VISÃO GERAL

O *software* que foi desenvolvido nesse trabalho é dividido em duas partes, um *bootloader*, e uma API para implementar um cliente OTA. O *bootloader*, com o auxílio da biblioteca FatFs, realizará a comunicação com o cartão SD, que conterá a última versão do *firmware* previamente recebido. Assim poderá substituir o *software* anterior da aplicação por um novo. Essa parte do *software* ficará armazenada em uma região da memória que dificilmente será reescrita, podendo ser reescrita somente com o auxílio de ferramentas de desenvolvimento como um JTAG e/ou *debugger*, então é uma peça do programa que improvavelmente será substituída. Será uma parte que é portável para todos os microcontroladores da família STM32, pois faz uso direto de funções de escrita na memória *flash* obtidas por meio do STM32Cube HAL, ficando a cargo do projetista fazer uma pequena adaptação, que posteriormente será explicado neste trabalho.

A API para o cliente OTA contem as funções necessárias para se comunicar com um servidor e fazer a busca pela versão mais atual do *firmware*, bem como para fazer o *download* dessa nova versão do *firmware* para um cartão SD. Para tanto, são utilizadas as bibliotecas LwIP que implementa o protocolo TCP/IP, a biblioteca MbedTLS que implementa o protocolo de criptografia e autenticação TLS e, a biblioteca FATFs para a leitura e escrita do cartão SD.

Após ser confirmada a existência de uma nova versão de *firmware*, o cliente OTA irá se comunicar novamente com o servidor com o intuito de fazer o *download* dessa nova versão, e a armazena no cartão SD do sistema. Com o *download* concluído com sucesso, a API é utilizada novamente para obter do servidor o arquivo contendo o *hash* deste *firmware*. Após, o cliente OTA executa uma função que cria o *hash* do *firmware* baixado anteriormente e o compara com o arquivo *hash* obtido do servidor. Para assim garantir que o *firmware* está integralmente e não corrompido no cartão SD. Finalmente, ao confirmar um novo *firmware* válido, é gerada uma reinicialização do sistema. Na inicialização, o *bootloader* é executado para a troca de *firmware*. O funcionamento mais detalhado do *bootloader* e da API de atualização OTA são explanadas

ainda neste trabalho, mas de forma geral a sequência de operação é descrita no fluxograma apresentado na Figura 11.

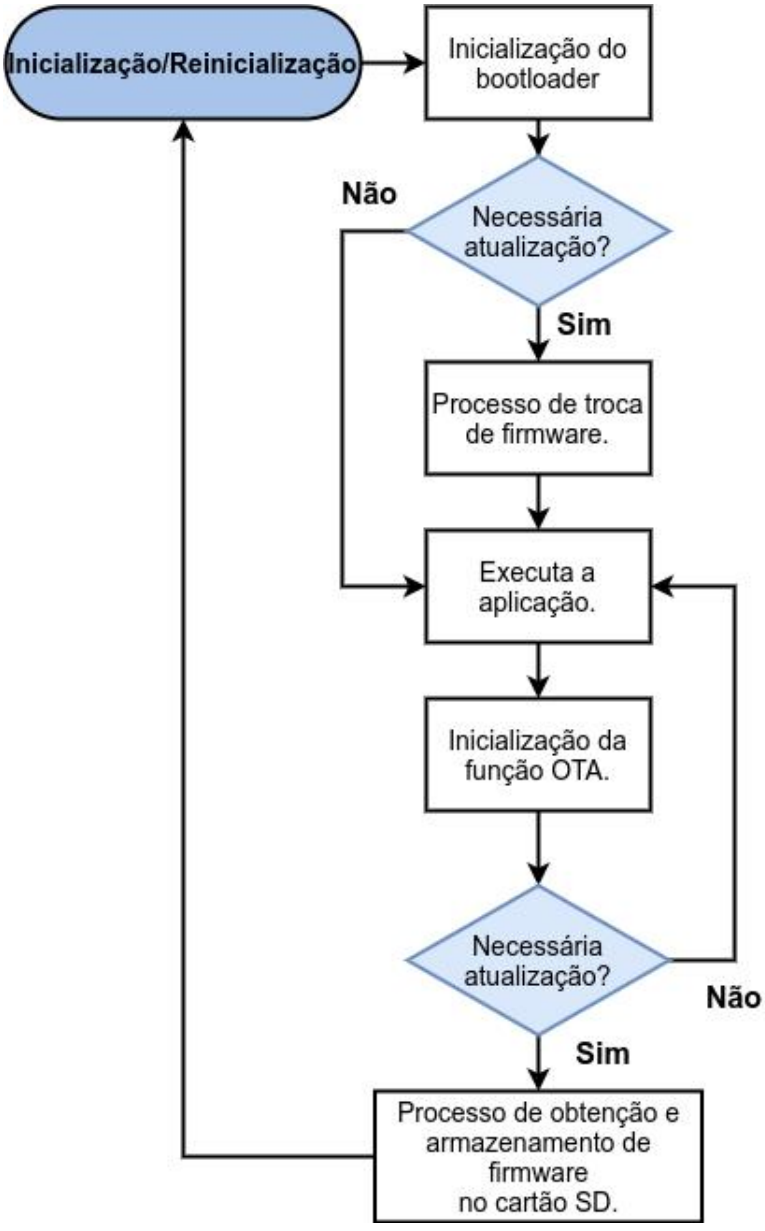

Figura 11 – Visão geral do funcionamento do sistema de atualização.
Fonte: autoria própria.

A API será uma peça de *software* que poderá ser substituída e atualizada em conjunto com as demais aplicações do sistema, como as bibliotecas LwIP, MbedTLS, o sistema operacional, entre outras peças de *software* utilizadas pela aplicação. O sistema de atualização OTA utiliza-se de bibliotecas já conhecidas e vastamente utilizadas por desenvolvedores de sistemas embarcados. Assim, projetos que necessitem fazer comunicação segura via rede, leitura e escrita de cartões SD, possam utilizar esse sistema de modo a poupar espaço na memória, visando a reutilização dessas bibliotecas. Portanto, o sistema pode ser amplamente utilizado por sistema de IoT.

Em caso de uma falha durante o processo de atualização OTA, o sistema tem a habilidade de se recuperar de forma autônoma. Como não haverá sobrescrita na área em que o *firmware* está posicionado no cartão SD, uma simples reinicialização do sistema pode fazer com que o *bootloader* seja ativado novamente e refaça o processo de cópia da memória.

É importante lembrar que o armazenamento do *firmware* e demais arquivos com informações para o processo de atualização poderia ser feito em qualquer mídia de armazenamento presente no sistema embarcado que se deseja utilizar. Seria possível, por exemplo, utilizar a o espaço restante da memória *flash* para armazenar esses dados, uma memória EEPROM, ou mesmo os 128 Mbits de memória Quad-SPI *flash* presentes no kit de desenvolvimento utilizado para teste neste projeto. Por definição de projeto desse trabalho foi selecionado o cartão SD para esse armazenamento.

3.2 *BOOTLOADER*

A partir de um arquivo de *linker*, a memória da plataforma será customizada com o intuito de abrigar os arquivos necessários para o *bootloader* e protegê-lo de eventuais sobrescritas que podem vir a ocorrer. Esse arquivo de *linker*, assim como o próprio *bootloader*, será escrito somente para os microcontroladores da família STM32, visto que cada plataforma tem suas próprias características como, tamanho de memória e endereços diferentes para cada fabricante e/ou arquitetura.

No arquivo de *linker* será especificada uma área especial na memória *flash* do sistema em que será abrigado o *bootloader*. Também será responsável por fazer com que o *bootloader* seja chamado após a inicialização do sistema. Assim será garantido que o *bootloader* sempre seja executado após a reinicialização do sistema embarcado.

O *bootloader* será responsável em fazer a troca de cada versão de *firmware* instalado no sistema embarcado. Sempre que o sistema for iniciado, o *bootloader* será inicializado e fará a procura de arquivos. Essa busca será possível pelo fato da biblioteca FatFs, que está implementada junto ao *bootloader*, criar um sistema de arquivos no cartão SD do sistema alvo, assim o *bootloader* pode acessar a memória do cartão sem a necessidade da aplicação final ser inicializada.

Se após o *reset* a procura do arquivo contendo a versão do *firmware* novo retornar com um resultado positivo, ele converte o valor contido neste arquivo de uma string para um tipo inteiro e o compara com os quatro últimos bytes da memória *flash*, posição onde se encontra a versão atual do *firmware*, caso a versão do novo *firmware* seja maior que a do *firmware* atual o *bootloader* inicia o processo de atualização.

Nesse processo o *bootloader* substitui completamente o *firmware* que contém as demais bibliotecas e API's em áreas não protegidas na memória, pelo binário do *firmware* presente no cartão SD. Esse processo é implementado com o uso das funções de escrita na memória *flash* do HAL fornecido pela STM32, onde inicialmente é feito o processo de

apagamento massivo dos setores da aplicação da memória *flash*, dessa forma esses setores são preenchidos com o valor 0xFF em cada byte.

Após o processo de apagamento é iniciado o processo de escrita, em que o arquivo contendo o *firmware* é lido a cada 512 bytes para um buffer, e a partir deste buffer são escritos 4 bytes por vez na memória *flash*, assim esse processo se repete até que o arquivo contendo o novo firmware seja completamente escrito na memória. Finalizada a escrita do novo *firmware*, acontece a escrita do número de versão nos quatro últimos bytes da memória, somente após essa escrita o processo de atualização pode ser considerado concluído com sucesso, e então ele pode apagar do cartão SD os arquivos do *firmware* e versão.

Caso o *bootloader* seja iniciado e em algum momento detectar que o valor dos quatro últimos bytes são iguais a 0xFFFFFFFF, ele identifica que a ultima operação de atualização não foi concluída com exito. Neste estado o *bootloader* fica sempre tentando encontrar uma nova versão do *firmware* no cartão SD e caso não encontre ele fica reiniciando o microcontrolador, e caso ele encontre, o *bootloader* tenta atualizar novamente com o *firmware* encontrado, independente da versão dele, garantindo assim que o sistema seja resistente a eventuais erros e eventos não previstos, como quedas de energia durante o processo. O funcionamento do *bootloader* pode ser observado na Figura 12.

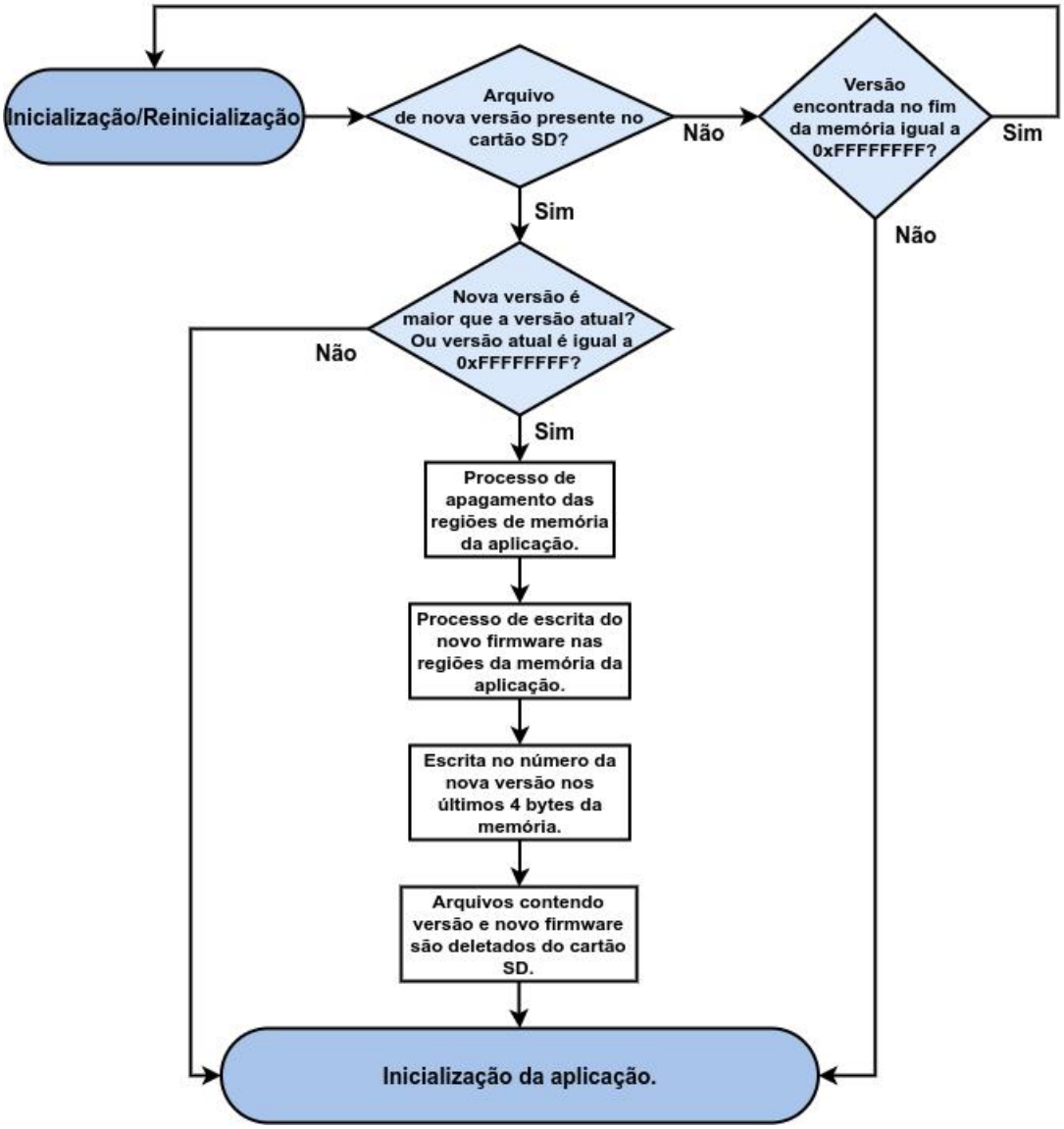

Figura 12 – Diagrama de funcionamento do *bootloader*.
Fonte: autoria própria.

3.3 API DE ATUALIZAÇÃO OTA

A API de atualização OTA que foi desenvolvida nesse trabalho tem o propósito de ser o mais portável possível, para assim, ser reutilizada por diversos projetos que necessitem da troca de seu *software* e com isso pode ser chamada quando o desenvolvedor necessitar, como após uma interrupção externa ou comando do servidor. Com esse objetivo, foram utilizadas as bibliotecas já bem difundidas, a LwIP para a criação da pilha TCP/IP, a Mbed TLS para criar uma camada se segurança nessa pilha, e a FATFS para a criação de um sistema de arquivos FAT. Assim desenvolvedores podem se aproveitar do fato de que essas bibliotecas já estão em seus sistemas como padrão para utilizá-las em suas próprias funcionalidades. A seguir será retratado como são cada uma das funcionalidades necessárias na API.

3.3.1 COMUNICAÇÃO COM O SERVIDOR

Com o uso da biblioteca LwIP e Mbed TLS foi criada uma pilha de comunicação no sistema alvo, que é responsável pela conexão segura com o servidor que fornecerá o novo *firmware*. Na implementação da pilha de comunicação, foi utilizada a API BSD Sockets, pois, o intuito é deixar o sistema de atualização portável, e essa API fornece suporte a sistemas operacionais de tempo real entre outra vantagens.

Como a biblioteca Mbed TLS já foi desenvolvida para ser integrada facilmente a várias aplicações embarcadas, ela foi utilizada para criar protocolos de segurança nessa comunicação com o servidor. Foram utilizados padrões SSL/TLS para ser criado um canal criptografado entre o servidor e o sistema alvo, para garantir que todos os dados transmitidos sejam sigilosos e seguros.

A comunicação com o servidor será feita por meio de um servidor HTTPS que utilizará o protocolo TCP para garantir que todos os dados obtidos pelo servidor sejam íntegros, evitando que o novo *firmware* e demais dados obtidos sejam corrompidos.

3.3.1.1 SERVIDOR HTTPS

Com o uso do servidor web Apache (THE APACHE SOFTWARE FOUNDATION, 1999), foi criado um servidor HTTP, em um computador que tem o sistema operacional Linux, configurando o servidor para utilizar-se de criptografia para haver uma comunicação segura entre o servidor e o cliente, plataforma embarcada, e assim ser possível o envio e recebimento de informações de forma segura.

Para que o servidor apache criado tenha uma camada de segurança adicional, foi necessário configurar ele de forma a aceitar requisições na porta 443, porta característica de um servidor HTTPS (*Hypertext Transfer Protocol Secure*). Foi configurado também um certificado *Secure Sockets Layer* (SSL) para a autenticação da comunicação entre o servidor e seus possíveis clientes.

3.3.2 DOWNLOAD E ARMAZENAMENTO DO FIRMWARE

A partir da utilização da biblioteca FatFs, é criado um sistema de arquivo FAT, que gerenciará a memória presente no cartão SD dentro da aplicação, e ele pode ser utilizado tanto pela aplicação final do sistema embarcado, quanto pela API. Esse sistema de arquivo será utilizado para que se possa identificar a posição na memória em que o *firmware* novo será colocado após o seu *download*, evitando que outros arquivos, pertencentes a aplicação final, sejam colocados com o mesmo nome, e fazendo com que o *bootloader* interprete de forma errada os arquivos gerando erros.

Quando o desenvolvedor iniciar a função OTA, a API iniciará novamente a comunicação com o servidor que contém a última versão do *firmware*, dessa vez com o propósito de fazer *download* do arquivo contendo o número da nova versão do *software*. Após o *download* a API

ainda irá verificar se o valor obtido por meio deste arquivo é maior que o presente nos últimos 4 bytes da memória, para assim verificar se existe a necessidade de se continuar com o processo de atualização.

Caso o valor da nova versão seja maior que a versão atual do firmware a API inicia o *download* do novo firmware, após esse processo a ferramenta inicia a comunicação novamente para obter agora o arquivo contendo o *hash* do *firmware* posteriormente baixado, e com isso é feito um teste de integridade neste arquivo onde inicialmente se gera um *hash* a partir do algoritmo SHA-256 fornecido pela biblioteca MBED TLS, e esse *hash* é comparado com aquele baixado do servidor. Se esse teste retornar com sucesso o processo de atualização é continuado, o arquivo de *hash* baixado do servidor é apagado do cartão SD e é iniciado o processo de reinicialização do microcontrolador com o intuito de se iniciar o bootloader para completar o processo de atualização. A Figura 13 ilustra o funcionamento da API.

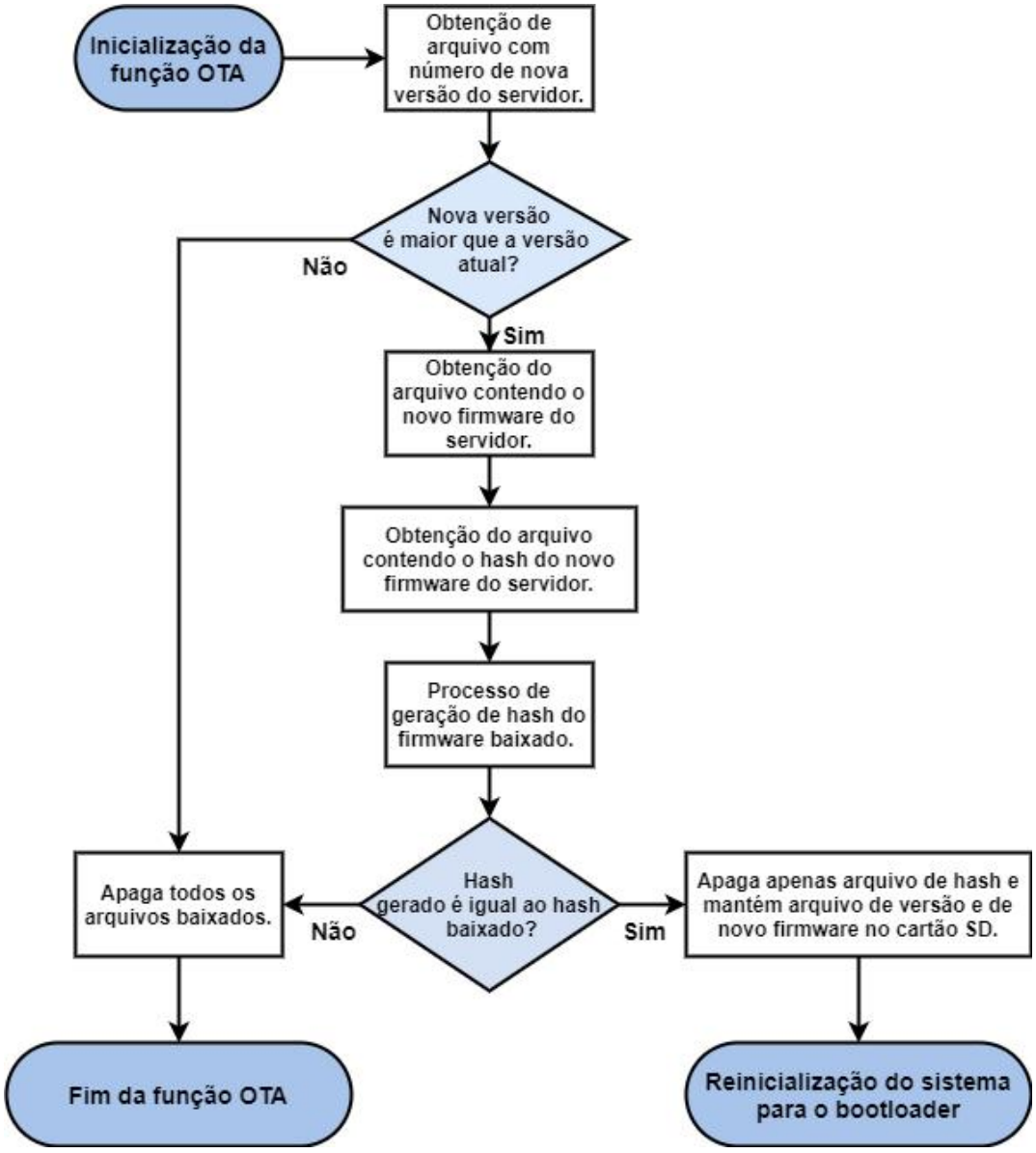

Figura 13 – Diagrama de funcionamento da API.
Fonte: autoria própria.

3.4 CRIAÇÃO DE FIRMWARE E ARQUIVOS PARA O SISTEMA DE ATUALIZAÇÃO

Para a utilização do sistema de atualização OTA desenvolvido nesse trabalho é necessário que o *firmware* e arquivos que serão disponibilizado no servidor mantenham um padrão determinado, assim esta sessão explicará como dever ser feito o processo de criação destes arquivos e a criação do *firmware*.

3.4.1 CRIAÇÃO DE ARQUIVOS AUXILIARES

Os arquivos auxiliares para a atualização, arquivos esses, que determina a nova versão do *firmware* presente no servidor e o arquivo contendo a *hash* deste *firmware*, devem ser escritos

em modo texto, o arquivo de versão deve conter somente o número da versão e nenhum outro dado, ficando um arquivo de texto como o apresentado na Figura 14.

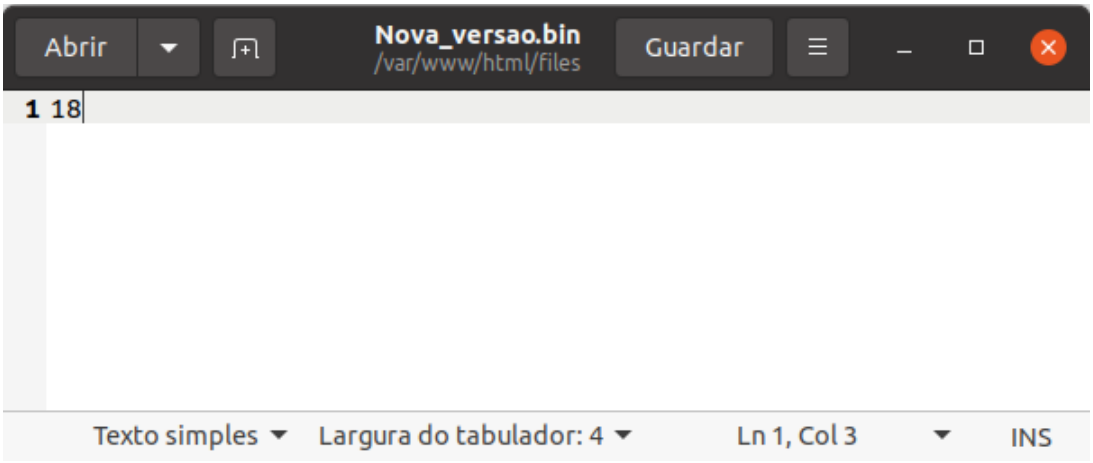

Figura 14 – Arquivo de versão que somente contém o número da versão.
Fonte: autoria própria.

O arquivo contendo a *hash* do *firmware* deve ser obtido a partir da função de *hash* criptográfico SHA-256. Em sistemas linux essa função pode ser obtida pelo comando "sha256sum"como demonstrado na Figura 15. Este dado deve ser salvo em modo texto também, como também é possível ser observado na Figura 15.

Figura 15 – Criação de um novo arquivo contendo o *hash* do *firmware*.
Fonte: autoria própria.

3.4.2 CRIAÇÃO DE FIRMWARE PARA ATUALIZAÇÃO

O arquivo contendo o *firmware* deve ser um binário (.bin) gerado pelo compilador. Geralmente os compiladores geram arquivos .elf que possuem símbolos de depuração que sempre devem ser gerados quando o intuito é depurar o programa que se esta desenvolvendo. São gerados também arquivos .hex que consistem de um arquivo em *ascii*, contendo as instruções em hexadecimal com os endereços de memória de cada instrução. Os arquivos .bin são arquivos binários contendo as instruções sem nenhuma padronização, contendo o *firmware* de forma integra e contínua.

Para gerar um arquivo .bin a partir da compilação do *firmware* é necessário a modificação de algumas configurações da IDE que é utilizada, neste caso o Eclipse (ECLIPSE FOUNDATION, 2001), onde é necessário modificar as configurações do projeto adicionando o comando apresentado no Algoritmo 1 da forma ilustrada na Figura 16.

Algoritmo 1: Comando necessário para a compilação do projeto e criação do arquivo .bin.
Fonte: Autoria própria.

```
arm-none-eabi-objcopy -O binary
"${BuildArtifactFileBaseName}.elf"
"${BuildArtifactFileBaseName}.bin"
&& arm-none-eabi-size "${BuildArtifactFileName}"
```

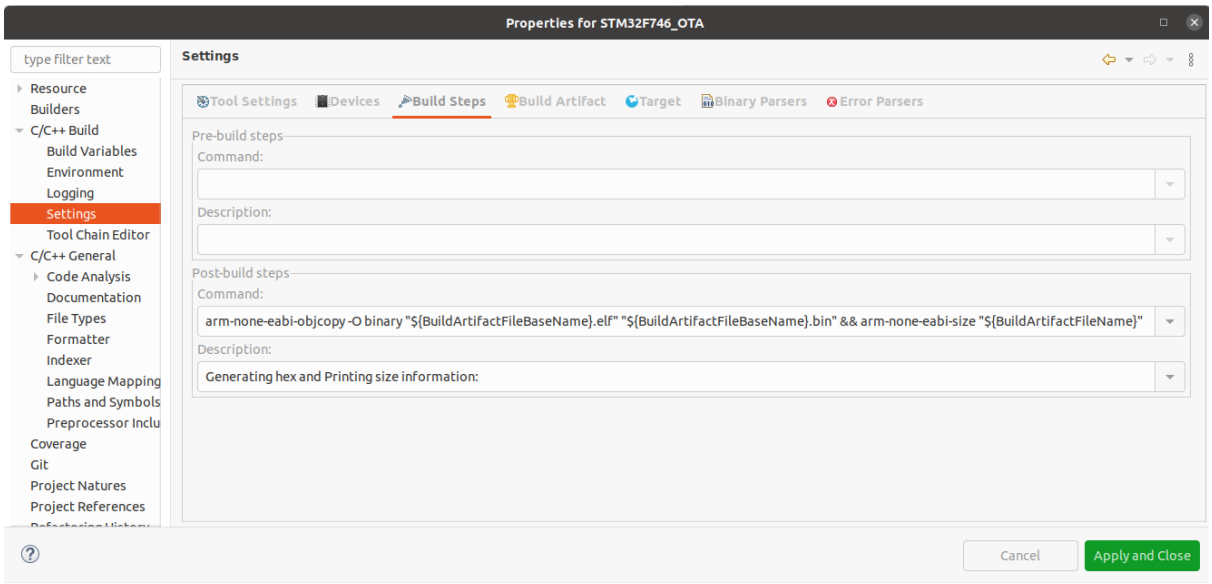

Figura 16 – Mudança no projeto para a criação de arquivos .bin.
Fonte: autoria própria.

3.4.3 FIRMWARES DE TESTE PROPOSTOS

Para os testes apresentados neste trabalho foram desenvolvidos dois *firmwares* muito parecidos, ambos possuem todas as bibliotecas necessárias para que o sistema de atualização OTA proposto funcione corretamente, além de possuírem o sistema operacional de tempo real FreeRTOS. Esses *firmwares* foram desenvolvidos para somente darem suporte ao sistema e não exercerem nenhuma outra função, com isso é obtida uma estimativa do tamanho que somente o sistema de atualização, suas bibliotecas e o FreeRTOS ocupam em memória.

Esses *firmwares* contém, além de inicializações necessárias de drivers, bibliotecas e do sistema operacional, uma pequena função que apresenta uma mensagem de apresentação mostrando a versão do *firmware* atual e uma mensagem de teste somente para fazer uma

diferenciação entre as versões, as frases são *"Firmware 1 - Technology is just nature we taught to do cool trick"* e *"Firmware 2 - The universe is a complexity machine"*. A mensagem de boas vindas do *firmware* 1 pode ser observado na Figura 17 e a do *firmware* 2 na Figura 18.

Figura 17 – Mensagem de boas vindas do *firmware* 1.
Fonte: Autoria própria.

Figura 18 – Mensagem de boas vindas do *firmware* 2.
Fonte: Autoria própria.

A função padrão desses *firmwares* é um laço infinito que a cada um segundo inicializa a função OTA que realiza o processo de atualização, assim é garantido que o *firmware* irá sempre executar a função OTA.

3.5 MATERIAIS UTILIZADOS

A API e o *bootloader* foram escritos na linguagem C, enquanto o *linker* será escrito em comandos de *linker*. A escrita desses códigos será feita com o uso do ambiente de desenvolvimento integrado (IDE) Eclipse (ECLIPSE FOUNDATION, 2001). O sistema de atualização OTA desenvolvido nesse trabalho foi desenvolvido para a plataforma STM32F746G-Discovery.

3.5.1 PLATAFORMA STM32F746G-DISCOVERY

O STM32F7 Discovery é um kit de desenvolvimento que permite ao usuário desenvolver e compartilhar aplicações com toda a série de microcontroladores STM32F7 baseados no processador ARM® Cortex®-M7 core. O kit discovery permite uma ampla diversidade de aplicações que podem se beneficiar de suporte a múltiplos sensores, áudio, tela gráfica, segurança, vídeos e conexões de alta velocidade (STMICROELECTRONICS, 2019). A Figura 19 ilustra o kit STM32F746NGH6-Discovery.

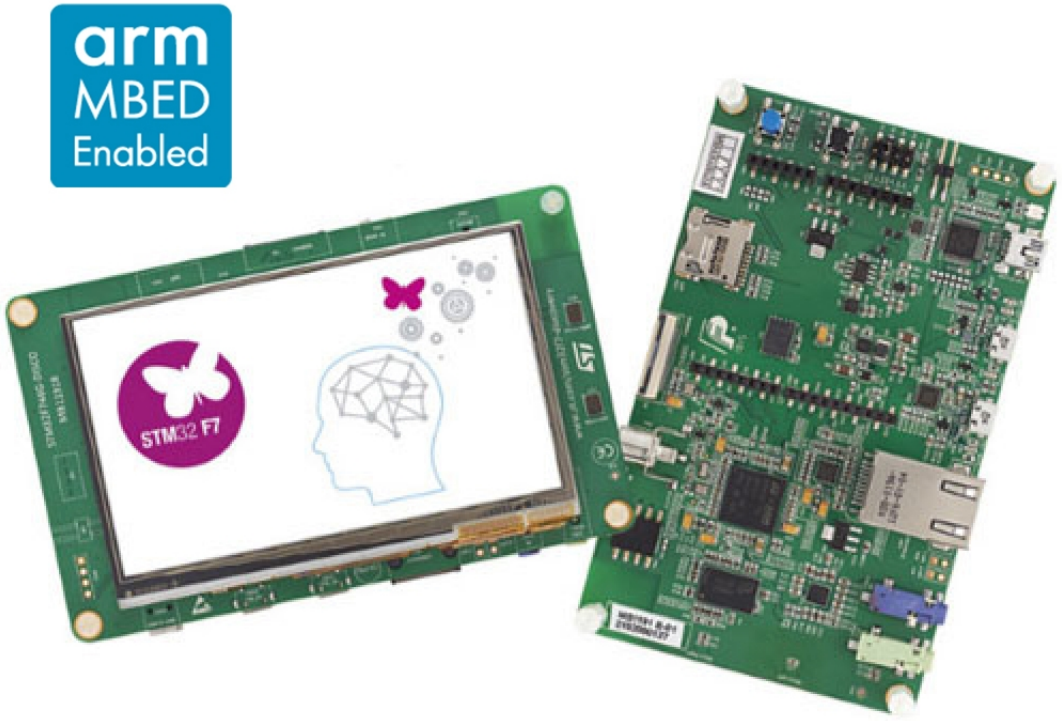

Figura 19 – Kit de desenvolvimento STM32F746G-Discovery.
Fonte: STMicroelectronics (2019).

Algumas de suas principais características são (STMICROELECTRONICS, 2019):

- Microcontrolador STM32F746NGH6 com 1 Mbytes de memória *flash* e 340 Kbytes de RAM, em um pacote BGA216.
- 128-Mbit de memória Quad-SPI *flash*.
- 128-Mbit SDRAM (Com 64 Mbits acessível).
- Conector para cartão microSD.
- Conector Ethernet em conformidade com a IEEE-802.3-2002
- Tela LCD de 4,3 polegadas, com resolução de 480x272 com *touch-screen* capacitivo.

3.6 UTILIZANDO O SISTEMA

Com o intuito de fazer a ferramenta criada neste projeto ser facilmente utilizada por diversos desenvolvedores essa sessão foi criada para facilitar o entendimento de como utilizar o sistema de atualização OTA proposto. Destacando como portar e utilizar o *bootloader* e toda a API de atualização OTA. Para a utilização de ambos os programas desenvolvidos neste trabalho é sempre necessário que as bibliotecas que são utilizadas por eles já estejam incluídas no projeto. Para o *bootloader*, é necessário somente a biblioteca FATFS, enquanto para o OTA são necessárias as bibliotecas FATFS, LWIP e MBED TLS.

3.6.1 PORTANDO O BOOTLOADER

Para se utilizar o *bootloader* proposto em outros sistemas embarcados da família STM32 deve-se primeiramente observar a organização da memória *flash* do sistema microcontrolado que se deseja portar o *bootloader*. Como é observado na Tabela 1 a memória *flash* do microcontrolador STM32F746NGH6 possui oito setores de memória com tamanhos distintos, com isso foi selecionado o primeiro setor de memória para abrigar o *bootloader*.

Nome	Endereço do bloco	Tamanho do setor
Setor 0	0x0800 0000 - 0x0800 7FFF	32 Kbytes
Setor 1	0x0800 8000 - 0x0800 FFFF	32 Kbytes
Setor 2	0x0801 0000 - 0x0801 7FFF	32 Kbytes
Setor 3	0x0801 8000 - 0x0801 FFFF	32 Kbytes
Setor 4	0x0802 0000 - 0x0803 FFFF	128 Kbytes
Setor 5	0x0804 0000 - 0x0807 FFFF	256 Kbytes
Setor 6	0x0808 0000 - 0x080B FFFF	256 Kbytes
Setor 7	0x080C 0000 - 0x080F FFFF	256 Kbytes

Tabela 1 – Organização do bloco de memória *flash* do microcontrolador STM32F746NGH6. Adaptado de:(STMICROELECTRONICS, 2019).

Para fazer o porte foi necessário modificar no arquivo de *linker* o tamanho máximo da memória utilizável, para evitar que os dados do *bootloader* ultrapassem o tamanho que é determinado para ele. Assim, no arquivo de *linker* foi modificado de forma que o tamanho máximo da memória seja de 32 Kbytes e iniciada no setor 0. Com isso a configuração de

memória do *bootloader* para o microcontrolador utilizado neste trabalho ficou da seguinte forma:

Algoritmo 2: Trecho do arquivo de comandos de *linker* que é necessário alterar para o porte do *bootloader*.
Fonte: Autoria própria.

```
/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
FLASH (rx)     : ORIGIN = 0x8000000, LENGTH = 32K
}
```

Como cada microcontrolador tem sua configuração de memória, o desenvolvedor deve observar a configuração do sistema alvo e colocar sempre o *bootloader* no setor 0 de seu microcontrolador.

Feita a modificação no arquivo de linker, o desenvolvedor iniciará a alteração do arquivo **bootloader.h** que contém algumas definições dependentes do seu microcontrolador. Neste arquivo as definições que precisam ser alteradas são:

- FIRMWARE_VERSION_ADDRESS: Esta definição mostra ao *bootloader* a posição na memória *flash* em que se deve armazenar a versão atual da aplicação presente na placa, deve ser sempre os quatro últimos bytes da memória, e necessita sempre ser a mesma posição configurada na API de atualização OTA. No caso do microcontrolador deste trabalho foi utilizada a posição: 0x080FFFFC.
- FIRMWARE_PATH: Esta definição mostra qual o caminho do arquivo em que se encontra o novo *firmware* que foi baixado pela API OTA, deve ser sempre igual ao que será definido na API.
- FIRMWARE_NEW_VERSION_PATH: Esta definição mostra qual o caminho do arquivo em que se encontra a versão do novo *firmware* que foi baixado pela API OTA, deve ser sempre igual ao que será definido na API.
- APP_START_ADDRESS: Esta definição mostra ao *bootloader* a posição na memória *flash* em que se deve armazenar o inicio da aplicação que será trocada sendo sempre o inicio do setor 1. No caso do microcontrolador deste trabalho foi utilizada a posição: 0x08008000.

Com essas alterações já é possível utilizar o *bootloader* em sistemas embarcados que utilizam microcontroladores da família STM32, restando agora a configuração da API de atualização OTA que será utilizada em conjunto com o *bootloader*.

3.6.2 CONFIGURAÇÃO DA API OTA

Assim como no *bootloader*, a aplicação também precisa ter seu arquivo de *linker* modificado para evitar sobrescritas no espaço reservado para o *bootloader* e reservar o espaço necessário para a variável de versão. Com isso o desenvolvedor deve novamente observar a Tabela 1 e verificar qual posição de memória se inicia sua aplicação e o tamanho total dela deve ser obtido com a seguinte fórmula:

$$T_f = FF - (F_{S1} + 4) \tag{1}$$

Em que T_f representa o tamanho final do *firmware*, FF representa o final da memória *flash* e F_{s1} representa o inicio do setor 1 da memória *flash*.

Utilizando a fórmula para o microcontrolador STM32F746NGH6 é obtido que o inicio do setor 1 de memória é 0x08008000 e o fim do ultimo setor de memória é 0x080FFFFF, assim é possivel obter os valores de inicio e tamanho da aplicação e assim configurar o arquivo de linker da aplicação da seguinte forma:

Algoritmo 3: Trecho do arquivo de comandos de *linker* que é necessário alterar para o porte da aplicação.
Fonte: Autoria própria.

```
/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
FLASH (rx)     : ORIGIN = 0x08008000, LENGTH = 0xF7FFB
}
```

Concluídas as alterações necessárias no arquivo de *linker*, o desenvolvedor tem que alterar algumas definições no arquivo **ota_server.h** com o intuito de adequar a API de atualização OTA para a sua aplicação. Neste arquivo as definições que precisam ser alteradas são:

- FIRMWARE_VERSION_ADDRESS: Esta definição mostra à aplicação a posição na memória *flash* em que se deve armazenar a versão atual da aplicação presente na placa e deve ser sempre os quatro últimos bytes da memória, e deve sempre ser o mesmo configurado no *bootloader*. No caso do microcontrolador deste trabalho foi utilizada a posição: 0x080FFFFC.
- FIRMWARE_PATH: Esta definição mostra qual o caminho do arquivo em que se deve armazenar o novo *firmware* que deverá ser baixado pela API OTA, deve ser sempre igual ao que será definido no *bootloader*.
- FIRMWARE_NEW_VERSION_PATH: Esta definição mostra qual o caminho do arquivo em que se deve armazenar a versão do novo *firmware* que será baixado pela API OTA, deve ser sempre igual ao que será definido no *bootloader*.

- BOOTLOADER_START_ADDRESS: Esta definição mostra à aplicação a posição na memória *flash* em que está armazenado o inicio do *bootloader* sendo sempre o inicio do setor 0. No caso do microcontrolador deste trabalho foi utilizada a posição: 0x08000000.
- FIRMWARE_NEW_VERSION_HASH_PATH: Esta definição mostra qual o caminho do arquivo em que de deve armazenar o *hash* do novo *firmware*.
- AUTH_SERVER: Esta definição mostra qual deve ser o endereço do servidor em que se deve obter os arquivos necessários para a atualização de *firmware*, então deve ser personalizada para cada aplicação.
- AUTH_PORT: Esta definição mostra qual deve ser a porta que se deve conectar no servidor, como é utilizado uma camada de TLS esta porta deve ser a 443.
- AUTH_REQUEST_VERSION: Esta definição mostra qual a requisição HTTP do arquivo contendo o número da nova versão que deverá ser baixada pela API.
- AUTH_REQUEST_FIRMWARE: Esta definição mostra qual a requisição HTTP do firmware da nova versão que deverá ser baixada pela API.
- AUTH_REQUEST_HASH: Esta definição mostra qual a requisição HTTP do arquivo contendo o *hash* da nova versão que deverá ser baixada pela API.
- SSL_CA_PEM: Esta definição mostra o certificado SSL utilizado na comunicação segura com o servidor.

Com essas definições corretamente configuradas pode-se utilizar facilmente a API de atualização OTA proposta neste trabalho, ficando a cargo do desenvolvedor somente programar em sua aplicação o algoritmo que determina quando a função que inicia a atualização é chamada, para que o processo de atualização OTA proposto neste trabalho se encarregue de fazer todo o processo de forma autônoma.

4 RESULTADOS

Este capítulo apresenta o que foi obtido como resultado deste trabalho, e as experiências do autor durante o desenvolvimento do sistema de atualização de *firmware* over-the-air. Primeiro serão apresentados os resultados mostrando como é um processo de atualização realizado com sucesso e em seguida discutido o uso, vantagens e desvantagens do sistema.

4.1 PROCESSO DE ATUALIZAÇÃO DE FIRMWARE

Aqui é apresentado todo o processo de atualização feito para comprovar o funcionamento do sistema desenvolvido neste trabalho, com isso é abordado desde a inicialização do *firmware* 1 com sua mensagem de apresentação, a disponibilização do *firmware* 2 no servidor, o processo de atualização, a inicialização do *firmware* 2 na plataforma embarcada STM32F746G e sua mensagem de apresentação.

4.1.1 ESTADO INICIAL DO MICROCONTROLADOR

Inicialmente é gravado no microcontrolador o *bootloader* e depois o *firmware* 1, com isso a memória *flash* é preenchida separadamente, visto que foram definidas áreas diferentes para aplicação e *bootloader* no arquivo de linker, assim está o estado inicial da memória do microcontrolador no inicio do teste.

Com o *bootloader* e *firmware* 1 gravados no *hardware*, já é possível fazer a primeira inicialização do sistema, com a reinicialização do *hardware* o sistema inicializa o *bootloader* para poder dar sequência de atualização ou salto para a aplicação. A Figura 20 mostra a configuração de memória *flash* do microcontrolador.

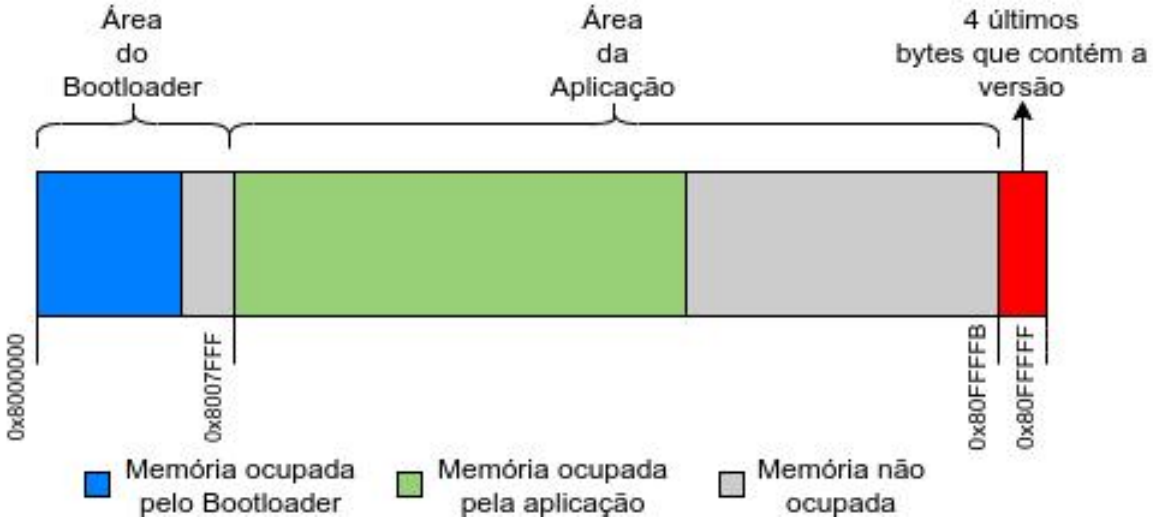

Figura 20 – Imagem ilustrativa da configuração da memória *flash* .
Fonte: Autoria própria.

4.1.2 INICIALIZAÇÃO DO BOOTLOADER

Após todos os processos de inicialização o *bootloader* é executado. Como o intuito é que o *bootloader* não ocupe muito espaço da memória *flash* do microcontrolador, e de forma que se adapte para todos os tipos de microcontroladores da família STM32, não foi previsto nenhum tipo de sinalização durante sua operação. Dessa forma, a substituição do *firmware* e o salto para a aplicação são executadas sem nenhuma informação visual para o usuário.

Durante sua execução o *bootloader* verifica a versão do *firmware* atual e o compara com a versão que está no cartão SD, caso a versão atual for menor ou não seja encontrada a nova versão do *firmware* o *bootloader* inicializa o *firmware* que está na memória *flash* . Como neste caso de teste o cartão SD se encontra vazio, o *bootloader* assume que não há nenhuma atualização a ser efetuada, assim ele inicializa a sequência de salto para a aplicação.

4.1.3 INICIALIZAÇÃO DO FIRMWARE 1

Com a inicialização da aplicação efetuada pelo *bootloader*, são executadas todas as inicializações de *drivers* , das bibliotecas utilizadas pelo *firmware* 1, e assim inicializado o sistema operacional FreeRTOS, e exibida via UART a mensagem de confirmação de montagem do *driver* do cartão SD e a mensagem de apresentação contendo a versão do *software* atual e o sua frase de identificação. A partir da impressão dessas frases o sistema operacional iniciará a tarefa OTA.

4.1.4 OTA

Ao iniciar a função OTA será buscado no servidor o arquivo contendo a versão do *firmware* 2, com esse arquivo baixado ele fará a comparação com a versão atual do *firmware* e caso necessário irá fazer o *download* do novo *firmware*, e do arquivo contendo o *hash* deste *firmware*. Tendo esses arquivos salvos no cartão SD ele irá fazer o *hash* do *firmware* baixado e verificar assim a integridade do arquivo baixado. Caso todos os processos ocorram com sucesso ele irá fazer com que esses arquivos permaneçam no cartão SD e iniciará o processo de salto para o *bootloader*, que encerra todos os *driver* e o sistema operacional e pula para a região do *bootloader* para o executar. Todo esse processo pode ser visto na Figura 21.

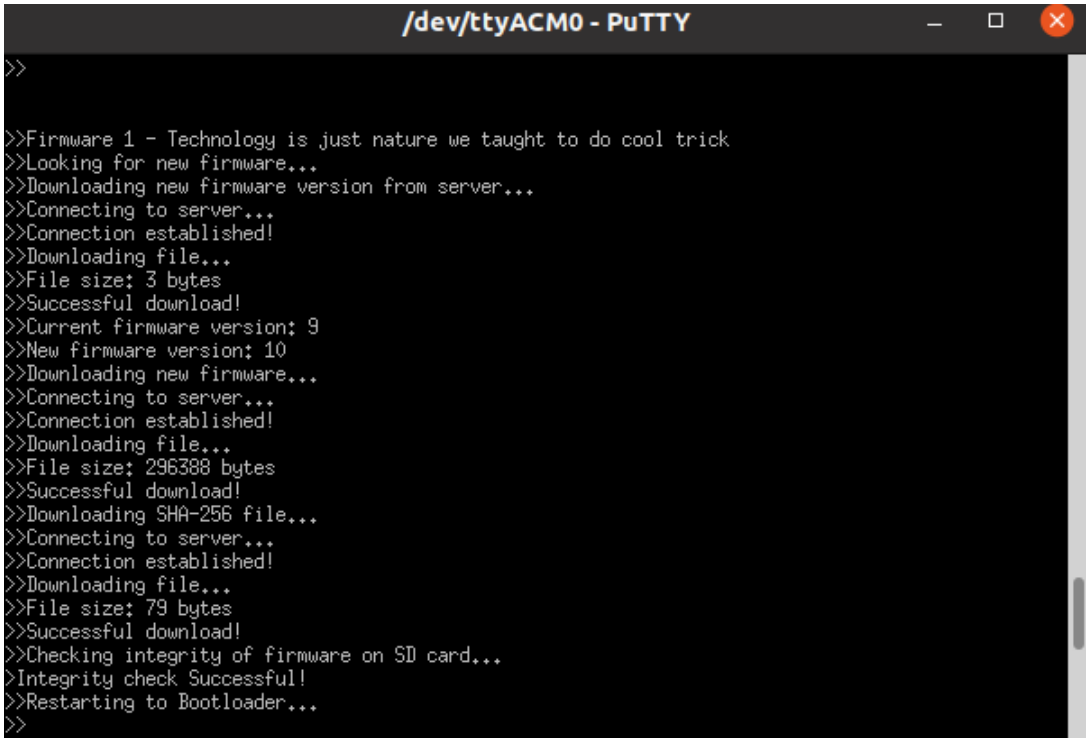

Figura 21 – Processo de atualização realizado pela função OTA no *firmware* 1
Fonte: Autoria própria.

4.1.5 REINICIALIZAÇÃO DO SISTEMA PARA O BOOTLOADER

O *bootloader* é inicializado novamente, mas agora ele consegue achar um arquivo de versão escrito no cartão SD, com isso fazer a comparação com a versão de *firmware* e como a versão encontrada no cartão SD é maior que a escrita na *flash* , ele iniciará o processo de troca de *firmware*. Após a troca de *firmware* ele escreverá nas 4 últimas posição da memória a versão do novo *firmware*, fazendo com que atualização seja concluída com sucesso, assim podendo novamente fazer o processo de pulo para a aplicação.

4.1.6 INICIALIZAÇÃO DO FIRMWARE 2

Após o pulo para aplicação efetuada pelo *bootloader* após a atualização são executadas todas as inicializações de *drivers* , das bibliotecas utilizadas pelo *firmware* 2, e assim inicializado o sistema operacional, exibida a mensagem de confirmação de montagem do *driver* do cartão SD e a mensagem de apresentação contendo a versão do *software* atual e a sua nova frase de identificação como pode ser observado na Figura 22. Assim concluindo com sucesso um processo de atualização de *firmware*.

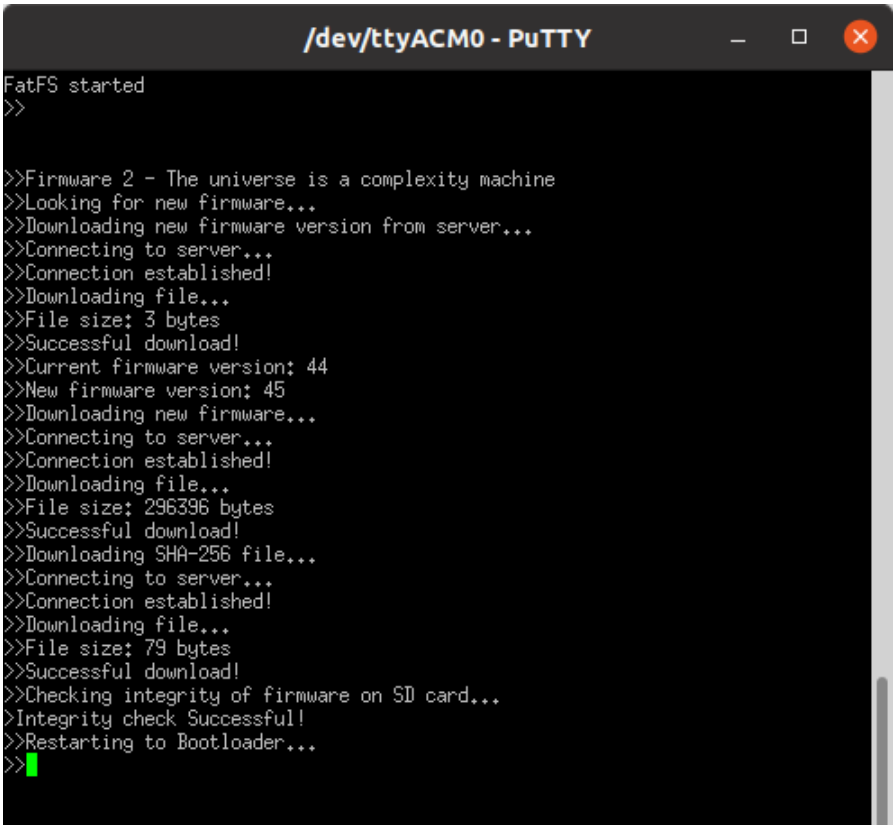

Figura 22 – *Firmware* 2 iniciado com processo de atualização.
Fonte: Autoria própria.

4.2 TRATAMENTOS DE ERROS IMPLEMENTADOS

Durante o processo de atualização e obtenção de arquivos diversos erros podem ocorrer, desde quedas de energia, a falhas de conexão com o servidor. Para evitar que esses erros deixem a plataforma embarcada em um estado não conhecido foram implementados diversos tratamentos de erros que serão explanados nessa sessão.

4.2.1 FALHA AO ENCONTRAR ARQUIVOS NO SERVIDOR

Caso o cliente OTA não conseguir encontrar no servidor um dos arquivos que são necessários para a atualização, como o arquivo contendo o número versão, ou o arquivo contendo a nova versão do *firmware*, ou mesmo o arquivo com o *hash* desse *firmware*. O cliente OTA tem a capacidade de detectar esse tipo de erro e apresentar ao usuário este erro, tornando fácil a identificação dessa falha. A Figura 23 apresenta como o cliente OTA informa o usuário do erro que ocorreu.

Figura 23 – Erro ao não encontrar arquivo no servidor.
Fonte: Autoria própria.

4.2.2 FALHA AO COMPARAR HASH

Após o *download* da última versão do *firmware*, o cliente OTA gera um *hash* utilizando a função de *hash* sha-256 que a biblioteca MbedTLS fornece, esse *hash* é comparado com o que foi obtido do servidor para garantir que o *firmware* obtido está integro e não corrompido. Caso essa comparação falhe, uma mensagem é mostrada para o usuário, essa mensagem pode ser observada na Figura 24.

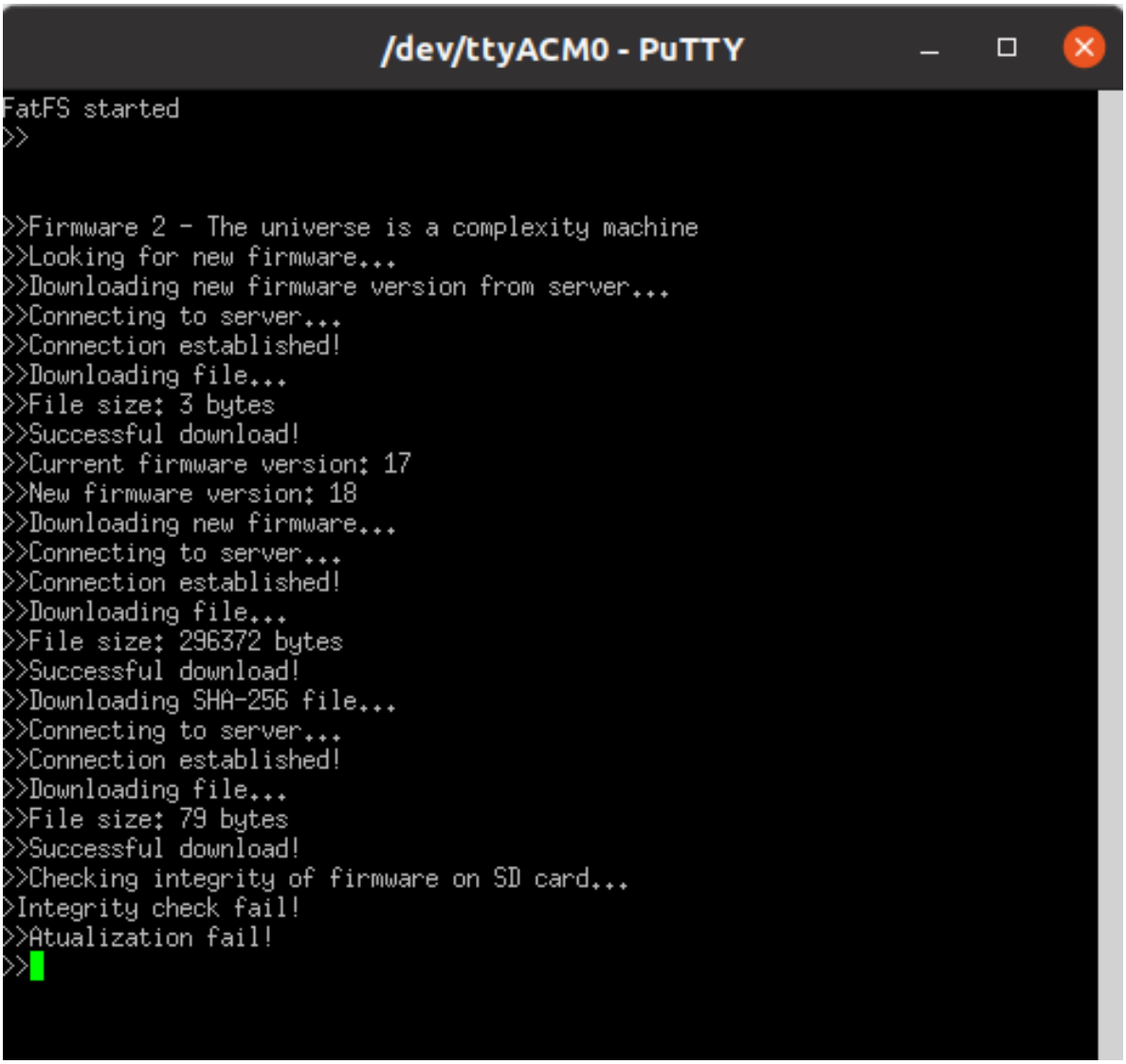

Figura 24 – Erro ao comparar *hash* do servidor com a *hash* gerada a partir do *firmware* obtido do servidor.
Fonte: Autoria própria.

4.2.3 QUEDA DE ENERGIA DURANTE PROCESSO DE ATUALIZAÇÃO

Caso exista uma queda de energia, ou algo que faça com que o microcontrolador seja reiniciado durante o processo de escrita do *firmware* na *flash* , que ocorre no *bootloader*, o sistema tem a capacidade de identificar este erro a partir do número de versão que está presente no fim da memória. No evento desse valor ser igual a 0xFFFFFFFF, significa que a última operação de escrita da *flash* não foi concluída com sucesso, assim é necessário que ela seja refeita. Assim o *bootloader* executa uma rotina de reinicialização do sistema caso não encontre o *firmware* para realizar a atualização no cartão SD, e caso encontre ele executa a atualização e salta para a nova aplicação.

4.3 DISCUSSÃO

Os resultado obtidos mostraram que o sistema de atualização de *firmware Over-The-Air* proposto neste trabalho funciona, e que todo o processo desde a verificação de uma aplicação válida feita pelo *bootloader*, seu processo de atualização e as obtenções e verificações de arquivos da API de atualização OTA ocorrem com sucesso.

Como o *bootloader* ocupa 22 kbytes de espaço na memória *flash* , pode ser utilizado por diversas placas da família de microcontroladores STM32 que possuam um setor da memória maior que esse espaço. No entanto, o *bootloader* proposto tem como único objetivo, substituir o *firmware* na memória *flash* do microcontrolador por um versão atualizada, desde que esse novo *firmware* esteja no cartão SD. Já o código que realiza o procedimento de verificação e *download* desse novo *firmware* a partir de um servidor na internet para o cartão SD é conhecido por atualização *Over-The-Air*.

A atualização OTA é executada por uma tarefa instalada no sistema e utiliza várias bibliotecas para sua operação. O conjunto de aplicação e bibliotecas da atualização OTA proposta ocupou 294kB de memória de programa em um microcontrolador ARM Cortex-M7, utilizando o compilador gcc. É importante destacar que as bibliotecas que adicionam suporte a concorrência, sistema de arquivos, comunicação TCP/IP e segurança por TLS podem e devem ser utilizadas pelo código de aplicação, sendo consideradas bibliotecas compartilhadas. Considerando que o microcontrolador utilizado possui 1024kB de memória *flash* , a aplicação do usuário ainda teria 698 kbytes restante na memória de programa, se já for subtraído os 32 kbytes reservados para o *bootloader*.

As bibliotecas utilizadas ocupam a maior parte destes 294kB, como pode ser verificado no arquivo de mapeamento de memória gerado na compilação do projeto. Nesse arquivo pode-se identificar a quantidade de memória de dados e de programa utilizada por cada uma das bibliotecas incluídas na projeto. Considerando a memória de programa, a biblioteca FatFS ocupa cerca de 8 kbytes de memória, a LwIP em torno de 48 kbytes de memória *flash* e por fim a MbedTLS que ocupa aproximadamente 130 kbytes de memória, sendo essa a biblioteca que mais ocupa espaço na memória *flash* . É obtido então que aproximadamente 186 kbytes do *firmware* é ocupados pelas bibliotecas, assim dois terços do *firmware* desenvolvido é ocupado por funções e outras definições das bibliotecas. A API de atualização desenvolvida ocupa por volta de 2 kbytes, ficando o restante do *firmware* ocupado pelo sistema operacional, outras funções do HAL e demais códigos de inicialização.

A ocupação de memória de programa poderia ser reduzida drasticamente caso não fosse utilizada a criptografia e autenticação provida pela biblioteca MbedTLS, devido a sua ocupação de 135 kB de memória *flash* . No entanto, a segurança na atualização de um *firmware* é extremamente importante para se garantir a origem e integridade do novo *firmware*. Outra possibilidade que não foi abordada neste trabalho é a otimização da compilação com o objetivo de se diminuir a ocupação da memória *flash* , que poderia diminuir o tamanho total do código de 294 kbytes para cerca de 260 kbytes. Outra otimização que pode ser feita é a otimização com

o intuito de se aumentar o desempenho, fazendo com que o código ocupe aproximadamente de 350 kbytes.

Outra análise importante de ser feita é o uso de memória RAM dessas aplicações, em que é possível observar que essas bibliotecas acabam por tomar muita memória RAM do sistema, assim como acontece na memória *flash* . A quantidade de memória RAM gasta pela FatFS fica em torno de 20 bytes tornando ela a biblioteca que menos ocupa espaço em ambas as memórias. É seguida pela MbedTLS que consome cerca de 9 kbytes, e então pela LwIP que por ter que criar uma pilha de protocolos chega a ocupar aproximadamente 40 kbytes de memória RAM. Além disso, existe o *heap* do FreeRTOS que é utilizado para pilhas de tarefas, alocação dinâmica de memória, semáforos, filas, etc. Durante o processo de atualização, dos 186 kbytes disponíveis na *heap*, 65 kbytes são alocados dinamicamente para as funções das bibliotecas.

Dessa maneira é observado que o sistema fica ainda menos portável para *hardwares* menos robustos, visto que o consumo de memória RAM das bibliotecas é muito alto, onde somente observando as bibliotecas chega a 50 kbytes. Outra observação importante é que nos *firmwares* desenvolvidos foi utilizado o sistema operacional FreeRTOS que ocupa 4 kbytes de memória RAM, e tem 186 kB de *heap*. Porém esse número pode ser reduzido se for diminuído o *heap* do sistema operacional. Enquanto outras funções e inicializações ocupam mais 29 kbytes da memória RAM. Com isso é obtido que dos 340 kbytes de memória RAM global disponíveis no microcontrolador STM32F746G, é ocupado 65 Kbytes do *heap*, mais 50 kbytes das bibliotecas e 29 kbytes de outras funções. Assim é notado que a ocupação de memória RAM chega a 144 kbytes, totalizando 45% da memória RAM disponível no microcontrolador.

Com essa informação sobre o tamanho dos arquivos, é possível notar que este sistema de atualização não é tão portável quanto se esperava no inicio deste trabalho, visto que só para ter um sistema de atualização funcional é necessário que o *hardware* em que se deseja ter esse sistema tenha ao menos 512 kbytes de memória *flash* , e no mínimo 115 kbytes de memória RAM, assim diminuindo muito a quantidade de microcontroladores que poderiam utilizar este sistema. Esse sistema acaba sendo muito proveitoso para aplicações que já utilizam essas bibliotecas, pois assim não precisariam incluir novamente as bibliotecas FATFS, MbedTLS, LWIP, como é o caso de dispositivos de IoT. De forma geral é possível observar os gastos de memórias com as bibliotecas na Tabela 2.

Biblioteca	Memória RAM	Memória flash
FatFs	20 bytes	8 kbytes
LwIP	40 kbytes	48 kbytes
MbedTLS	9 Kbytes	130 kbytes
FreeRTOS	4 Kbytes	11 kbytes

Tabela 2 – Ocupação das memórias do microcontrolador STM32F746NGH6 pelas bibliotecas. Fonte: Autoria própria.

Ainda assim é possível afirmar que o sistema cumpre com o objetivo geral e específicos propostos, visto que o sistema é capaz de efetuar a atualização da plataforma embarcada STM32F746NGH6 da forma proposta e ainda é capaz de identificar quando houve falhas em seu sistema de atualização e se recuperar de forma autônoma dessas falhas.

Um vídeo demonstrando todo processo de atualização pode ser encontrado no seguinte link do YouTube:

https://www.youtube.com/watch?v=_nfL7P5aDas

Os códigos do *bootloader* e do sistema de atualização OTA desenvolvidos nesse trabalho podem ser encontrados no seguinte link do GitHub:

https://github.com/GustavoCorrea-GC/OTA

5 CONCLUSÃO

O trabalho conseguiu alcançar o objetivo geral de criar um sistema de atualização de *firmware Over-The-Air* utilizando as bibliotecas FatFs, MBED TLS e LWIP. Todos os objetivos específicos também foram atingidos visto que foi desenvolvido um *bootloader* que utiliza o sistema de aquivo FAT, foi implementada uma API que faz a comunicação com o servidor, armazena arquivos no cartão SD e também foi comprovado o funcionamento do sistema na plataforma embarcada STM32F746NGH6-DISCOVERY.

Durante o desenvolvimento do trabalho foi possível observar que o *bootloader* desenvolvido é muito mais portável do que inicialmente se planejava. Como o processo de troca de *firmware* foi desenvolvido utilizando o HAL da STM, foi possível fazer com que o *bootloader* funcione para toda a linha de microcontroladores da família STM32 que possuam a biblioteca FatFS e acesso a um cartão SD. Além de ser um *bootloader* robusto que consegue se recuperar em caso de falhas como quedas de energia durante o processo de troca de *firmware*.

A API que faz a busca dos arquivos no servidor HTTPS e os armazena no cartão SD se mostrou perfeitamente funcional, conseguindo fazer a comparação de versões e a verificação de integridade do *firmware* baixado a partir de um *hash* obtido do servidor, deixando o sistema muito mais seguro contra falhas, além de ter a habilidade de encerrar todos os processos da aplicação para que o *bootloader* seja executado novamente.

O sistema de atualização implementado na plataforma embarcada STM32F746NGH6 se mostrou funcional, conseguindo demonstrar como diferentes *firmwares* puderam ser trocados de forma rápida, dando ao usuário sinalizações que mostravam cada etapa do processo obtenção do novo *firmware*, que é utilizado na atualização do sistema.

Um dos inconvenientes deste trabalho foi o fato que o sistema de atualização ocupar um espaço muito grande na memória *flash* e RAM do microcontrolador, fazendo com que o sistema não pudesse ser utilizado em *hardwares* com pouca memória, assim diminuindo a portabilidade do sistema. Entretanto ainda é um ferramenta muito útil e poderosa para sistemas mais robustos que necessitem de comunicação segura com um servidor e necessite de um sistema de arquivos em um cartão SD, ou seja sistemas de IoT. Com esses inconvenientes é possível que trabalhos futuros sejam feitos com o intuito de diminuir ou extinguir esses contratempo, além de ser possível modificar a forma de obtenção de novos *firmwares*.

Com o intuito de se tornar ainda mais robusto e ocupar uma quantidade de memória menor, esta seção aborda algumas propostas de melhoria do sistema que podem ser feitas em trabalhos futuros. Essa proposta faz com que o sistema seja mais atrativo para ser utilizado em sistemas com uma quantidade menor de memória.

Uma proposta que consiste em utilizar todas as bibliotecas já utilizadas no *bootloader* no próprio *firmware* da aplicação em que será atualizado. O *firmware* pode, a partir de ponteiros de função, utilizar as funções presentes na biblioteca do FatFS e do HAL que já

estão implementadas no *bootloader* em sua aplicação, não sendo necessário reimplementa-las no *firmware* da aplicação, tornando o tamanho final que a aplicação ocupa na memória de programa seja menor, assim fazendo com que placas com menor memória possam utilizar o sistema.

Pode-se utilizar outros meios de comunicação para a obtenção do novo *firmware*, fazendo com que não exista a necessidade de se utilizar toda a comunicação com o servidor e assim diminuindo a quantidade de memória RAM ocupada pelo sistema. Uma implementação possível é o uso de UART para a obtenção do *firmware*, onde o arquivo é dividido em diversos pacotes e esses são enviados aos poucos para o *hardware* e podem ser conferidos os CRCs no final de cada pacote para se obter uma prova de que o *firmware* obtido é integro.

Referências

BALL, S. **Embedded Microprocessor Systems: Real World Design**. 3rd. ed. Newton, MA, USA: Butterworth-Heinemann, 2002. ISBN 0750675349. Citado na página 13.

CHAN. **FatFs - Generic FAT File System Module**. 2016. Disponível em: <(http://elm-chan.org/fsw/ff/00index_e.html)>. Acesso em: 06 de setembro de 2019. Citado 3 vezes nas páginas , 13 e 30.

CHANDRA, H. et al. Internet of things: Over-the-air (ota) firmware update in lightweight mesh network protocol for smart urban development. In: **2016 22nd Asia-Pacific Conference on Communications (APCC)**. [S.l.: s.n.], 2016. p. 115–118. Citado na página 33.

DAVIS, T.; DURLIN, D. **Bootloaders 101: making your embedded design future proof**. 2013. Disponível em: <(https://www.embedded.com/design/prototyping-and-development/4410233/Bootloaders-101--making-your-embedded-design-future-proof)>. Acesso em: 06 de setembro de 2019. Citado 5 vezes nas páginas , 13, 19, 20 e 21.

DEVINE, C. **SSL Library mbed TLS / PolarSSL**. 2006. Disponível em: <(https://tls.mbed.org)>. Acesso em: 06 de setembro de 2019. Citado 4 vezes nas páginas , 13, 25 e 26.

DIERKS, T.; RESCORLA, E. **The Transport Layer Security (TLS) Protocol, Version 1.2**. 2008. Disponível em: <(https://tools.ietf.org/html/rfc5246)>. Acesso em: 28 de novembro de 2019. Citado na página 26.

DUNKELS, A. **lwIP - A Lightweight TCP/IP stack**. 2002. Disponível em: <(https://savannah.nongnu.org/projects/lwip/)>. Acesso em: 06 de setembro de 2019. Citado 2 vezes nas páginas 13 e 24.

ECLIPSE FOUNDATION. **Eclipse Foundation, Inc.** 2001. Disponível em: <(https://www.eclipse.org/)>. Acesso em: 28 de novembro de 2019. Citado 2 vezes nas páginas 43 e 44.

GARTNER, INC. **Leading the IoT, Gartner Insights on How to Lead in a Connected World**. 2019. Disponível em: <(https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf)>. Acesso em: 28 de novembro de 2019. Citado na página 13.

KUROSE, J. F.; ROSS, K. W. **REDES DE COMPUTADORES E A INTERNET - Uma Abordagem Top-Down**. [S.l.]: Pearson Education, inc, 2010. Citado 3 vezes nas páginas 22, 24 e 27.

MARWEDEL, P. **Embedded System Design**. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 1402076908. Citado na página 12.

MOZILLA DEVELOPER NETWORK. **Métodos de requisição HTTP**. 2021. Disponível em: <(https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods)>. Acesso em: 2 de julho de 2021. Citado na página 24.

NEAGU, C. **What is firmware? What does firmware do?** 2021. Disponível em: <(https://www.digitalcitizen.life/simple-questions-what-firmware-what-does-it-do/)>. Acesso em: 16 de agosto de 2021. Citado na página 12.

NOVIELLO, C. **Mastering STM32**. 2018. Disponível em: <(https://leanpub.com/mastering-stm32)>. Acesso em: 06 de setembro de 2019. Citado na página 22.

NSA. **National Security Agency**. 1952. Disponível em: <(https://www.nsa.gov/)>. Acesso em: 28 de novembro de 2019. Citado na página 27.

ODAT, H. A.; Ganesan, S. Firmware over the air for automotive, fotamotive. In: **IEEE International Conference on Electro/Information Technology**. [S.l.: s.n.], 2014. p. 130–139. Citado na página 32.

QING, Y. C. L. **Real-time concepts for embedded systems**. [S.l.]: CRC Press, 2003. Citado 6 vezes nas páginas , 15, 16, 17, 18 e 19.

SALUTES, B. **Bootloader: o que é e para que serve?** 2018. Disponível em: <(https://www.androidpit.com.br/bootloader-o-que-e-para-que-serve)>. Acesso em: 06 de setembro de 2019. Citado na página 13.

SD CARD ASSOCIATION. **SD Card**. 2016. Disponível em: <(https://www.sdcard.org/)>. Acesso em: 28 de novembro de 2019. Citado 2 vezes nas páginas 28 e 29.

STMICROELECTRONICS. **Discovery kit with STM32F746NG MCU**. 2019. Disponível em: <(https://www.st.com/en/evaluation-tools/32f746gdiscovery.html)>. Acesso em: 15 de novembro de 2019. Citado 4 vezes nas páginas , 22, 45 e 46.

STMICROELECTRONICS. **Description of STM32F7 HAL and low-layer drivers**. 2021. Disponível em: <(https://www.st.com/resource/en/user_manual/dm00189702-description-of-stm32f7-hal-and-lowlayer-drivers-stmicroelectronics.pdf)>. Acesso em: 15 de junho de 2021. Citado 2 vezes nas páginas 31 e 32.

TANENBAUM, A. S. **Computer Networks**. [S.l.]: Pearson Education, inc, 2003. Citado 4 vezes nas páginas , 22, 23 e 25.

TANENBAUM, A. S. **OPERATING SYSTEMS DESIGN AND IMPLEMENTATION**. [S.l.]: Pearson Education, inc, 2007. Citado 4 vezes nas páginas , 27, 28 e 29.

TECHOPEDIA. **Hardware Abstraction Layer (HAL)**. 2021. Disponível em: <(https://www.techopedia.com/definition/4288/hardware-abstraction-layer-hal)>. Acesso em: 16 de agosto de 2021. Citado na página 31.

TENG, C. et al. Firmware over the air for home cybersecurity in the internet of things. In: **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.: s.n.], 2017. p. 123–128. Citado na página 33.

THE APACHE SOFTWARE FOUNDATION. **The Apache Software Foundation**. 1999. Disponível em: <(https://www.apache.org/)>. Acesso em: 11 de julho de 2021. Citado na página 39.