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

GUSTAVO LUIZ ANDRADE CORRÊA

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

PROPOSTA DE TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2019

GUSTAVO LUIZ ANDRADE CORRÊA

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

Proposta de 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
2019

RESUMO

CORRÊA, Gustavo L. Andrade. Sistema *open-source* para a atualização de *firmware Over-The-Air* para dispositivos de IoT baseado nas bibliotecas LwIP, Mbed TLS e FatFs. 2019. 29 f. Proposta de Trabalho de Conclusão de Curso – DAINF — Departamento Acadêmico de Informática , Universidade Tecnológica Federal do Paraná. Pato Branco, 2019.

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 existem uma grande variedade de implementações dessa funcionalidade, mas a falta de um padrão dificulta a sua ampla utilização. Assim, esse trabalho propõe uma solução *open-source* 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 e um *bootloader* que fazem todo o processo de obtenção e atualização de *firmware*.
Palavras-chave: Atualização. Firmware. Over-The-Air. Portável. IoT.

LISTA DE FIGURAS

LISTA DE TABELAS

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 cada vez 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. Em algumas dessas funções era fundamental a presença de agentes humanos para serem realizadas, ou não existiam, pois, uma pessoa não a exerceria em tempo hábil.

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 vemos em *desktops* e *notebooks* atuais.

Com a necessidade cada vez maior da implementação desses sistemas no nosso dia a dia, 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 instalação final desse projeto 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 gerarmos um maior valor e confiabilidade ao sistema. A possibilidade de serem feitas manutenções futuras no *software*, que no contexto de sistemas embarcados é chamado de *firmware*, 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.

Esse trabalho de conclusão de curso propõe um método de manutenção desses *firmwares* de forma remota, que possa ser o mais portável possível. 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 *software* 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.

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 entrar em ação. Como é o primeiro programa a ser inicializado 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).

IoT (*Internet of things*), 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. 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.

1.1 OBJETIVO GERAL

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

1.2 OBJETIVOS ESPECÍFICOS

- Desenvolver o *bootloader* que utiliza o sistema de arquivo FAT.
- 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. 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

Nesse capítulo será feita uma revisão da literatura necessária para o entendimento desse 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 biblioteca Mbed TLS, alguns protocolos e algoritmos de segurança implementados por ela. Será mostrado o que é um sistema de arquivo, 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 código de um endereço pré-definido e gravado permanentemente na memória. O código contido nessa localização da memória é chamado de *reset vector*. O *reset vector* é usualmente uma instrução de salto para outro espaço da memória em que o real código de inicialização se encontra. A razão desse salto para outra localidade da memória é para manter o *reset vector* pequeno. O *reset vector* pertence a uma pequena área da memória reservada pelo sistema por motivos especiais. O *reset vector*, assim como o código de inicialização do sistema, precisa estar armazenado permanentemente. Por causa desse problema, o código de inicialização, chamado de código *bootstrap*, reside na memória somente de leitura (ROM), na flash ou em outra memória não volátil. O termo *loader* se refere ao código que é responsável por executar o *bootstrap*, fazer o possível *download* de uma imagem de outro local e inicialização da aplicação final.

O conceito é melhor explicado através de um exemplo. Nesse exemplo, iremos assumir 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 *reset vector* 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) usável.

No exemplo ilustrado na Figura 1, 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 nesse 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.

A primeira parte do processo de *bootstrap* do sistema é colocar o sistema em um estado conhecido. São colocados 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. Nesse 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.

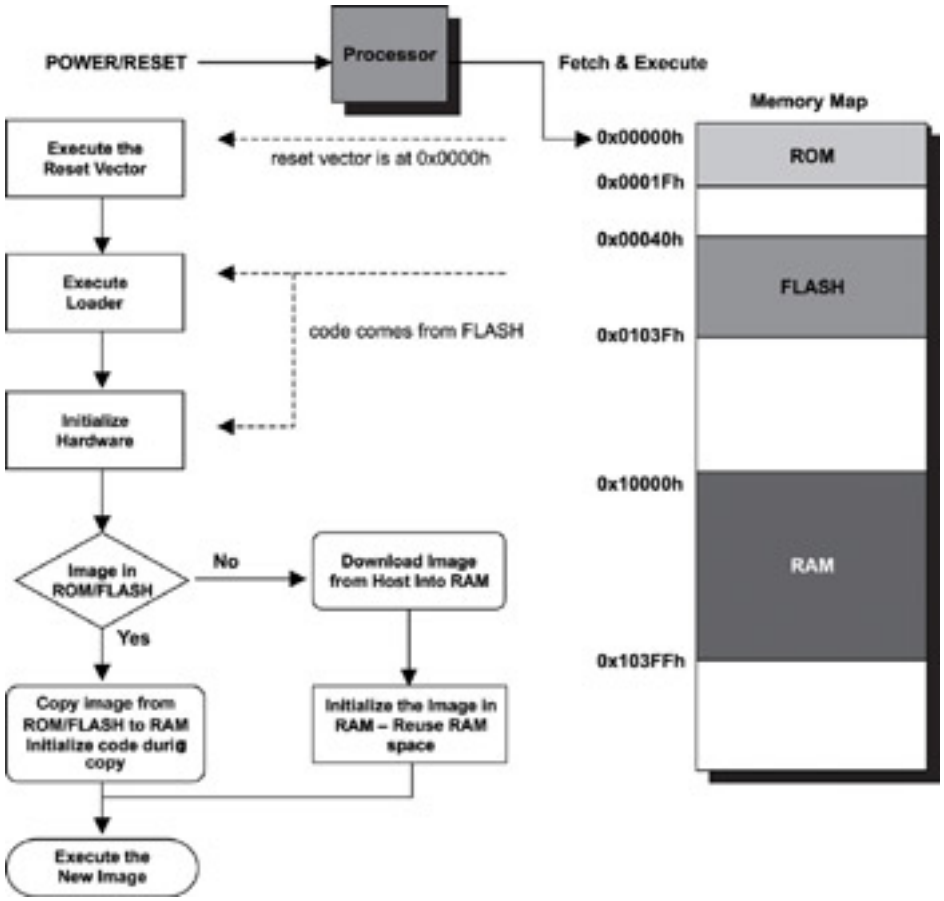

Figura 1 – Processo de inicialização de um sistema embarcado.
Fonte:(QING, 2003)

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 contéudo dessa seção, portanto, faz parte da imagem executável final e é transferido completamente pelo *loader*. Por outro lado, o contéudo 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 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 2 mostra como geralmente fica alocado um *bootloader* e o *firmware* na memória.

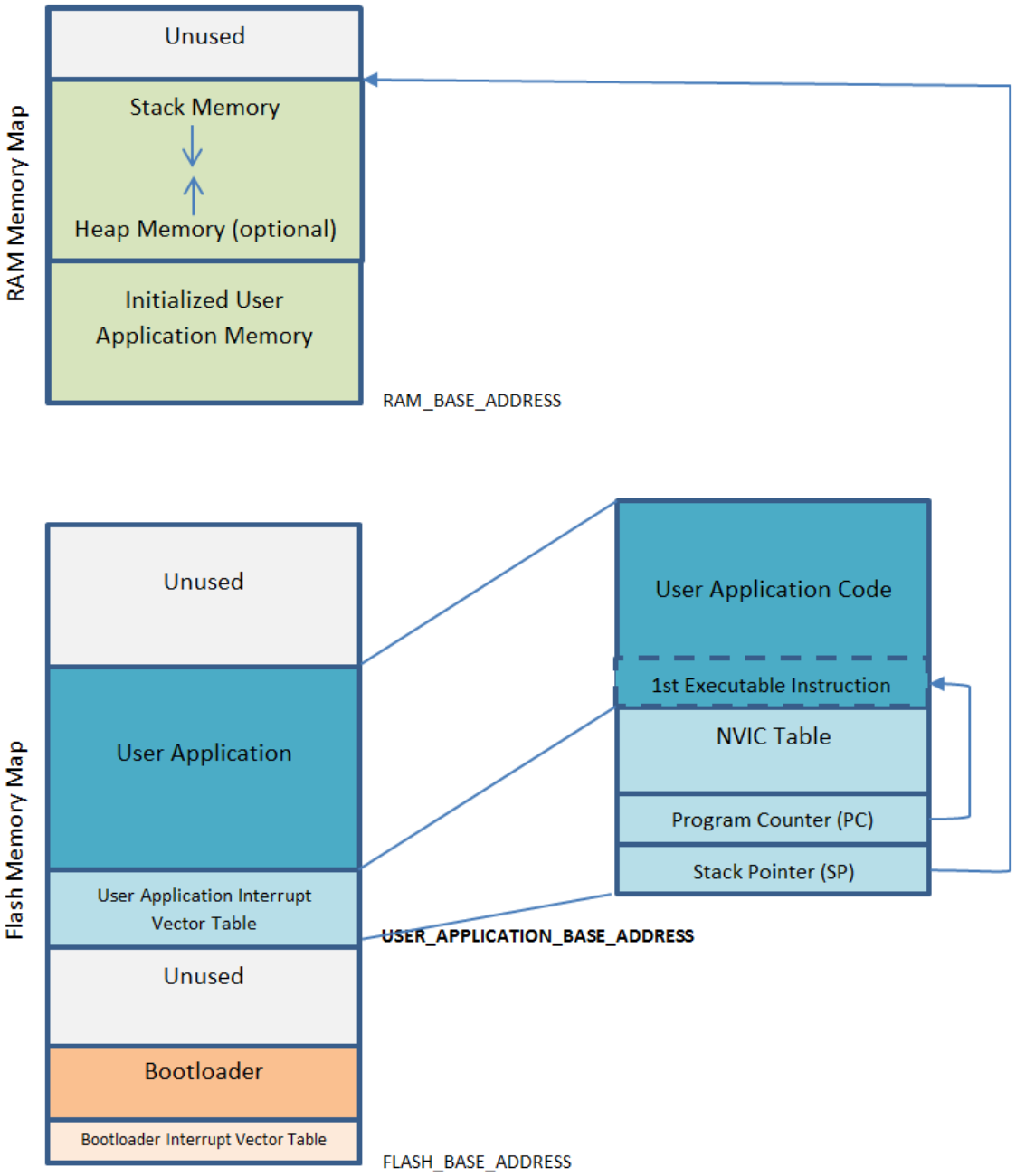

Figura 2 – Alocação do bootloader e firmware nas memórias flash e ram.
Fonte:(DAVIS; DURLIN, 2013)

Sua função se resume geralmente a: comunicar-se com outro servidor, ler os arquivos enviados pelo *host*, atualizar o *firmware* de seu microcontrolador, e iniciar esse novo *software*. Pode conter instruções e comandos definidos pelo projetista para somente o circuito integrado em uso, impossibilitando a utilização do mesmo código em outras placas. Portanto, é uma peça de *software* que não é portável para várias plataformas. A Figura 3 mostra o funcionamento de um bootloader padrão.

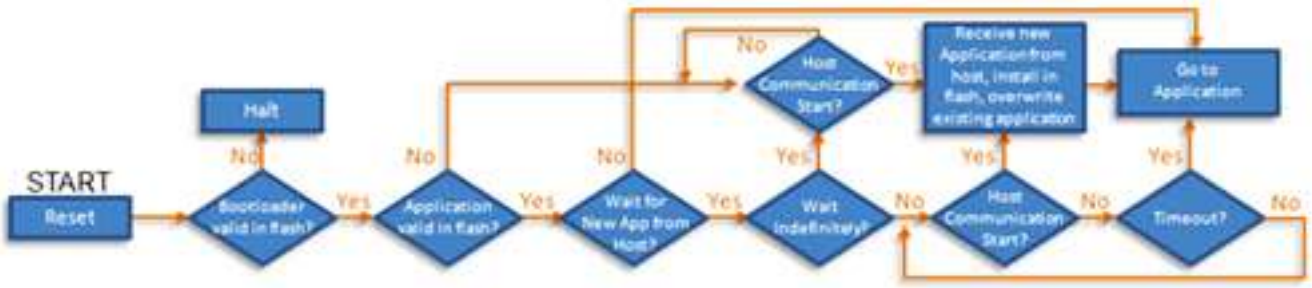

Figura 3 – Fluxograma de operações de um bootloader.

Os sistemas microcontrolados STM32 vem de fábrica com um *bootloader* pré programado na ROM. 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).

É possível ser criado um *bootloader* customizado com 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*, assim após esse processo o sistema pode substituir o *software* anterior pelo recebido.

2.1.2 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 *archive utility* 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 aonde colocar o código binário e os dados no sistema embarcado alvo. Assim podemos concluir que, a função principal de um *linker* é combinar múltiplos arquivos objetos em um maior arquivo objeto relocável, um arquivo objeto compartilhado ou uma imagem executável final. Esse processo pode ser observado na Figura 5.

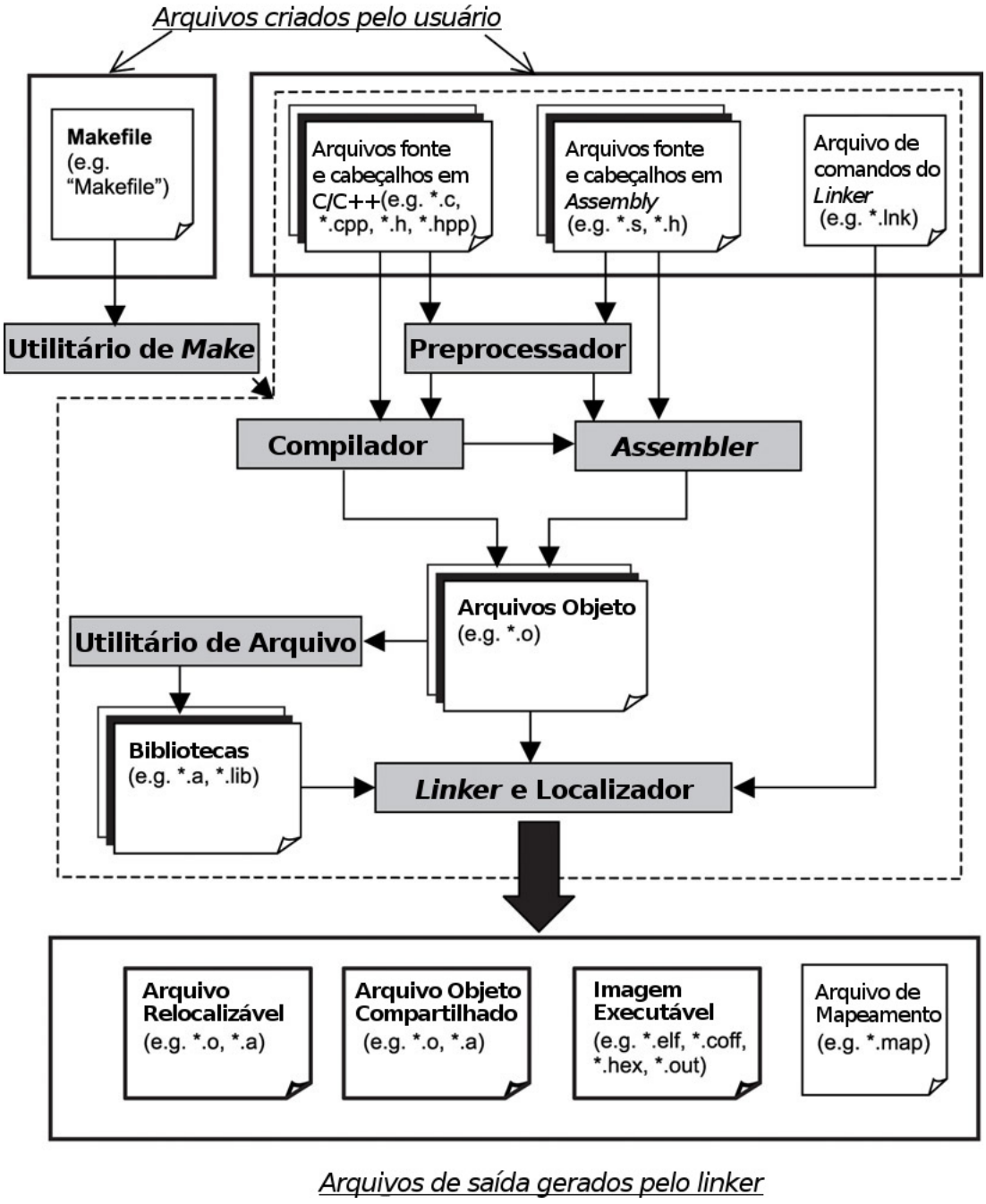

Figura 4 – Criando um arquivo de imagem para um sistema alvo.
Fonte:(QING, 2003)

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.

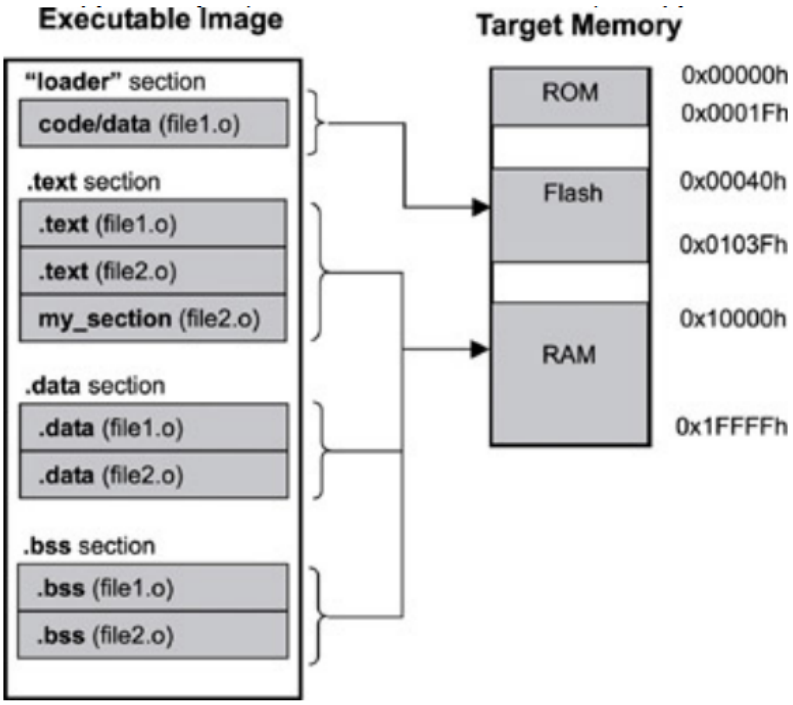

Figura 5 – Mapeando uma imagem executável em um sistema alvo.
Fonte:(QING, 2003)

2.2 COMUNICAÇÃO CLIENTE-SERVIDOR

Um programa cliente é um programa que funciona em um sistema final, 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. O programa cliente e o servidor interagem enviando mensagens um para o outro pela internet. Nesse 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, esse conjunto de protocolos é conhecido como pilha TCP/IP (*Transmission Control Protocol/Internet Protocol*). Essa pilha é dividida em quatro camadas, onde 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.

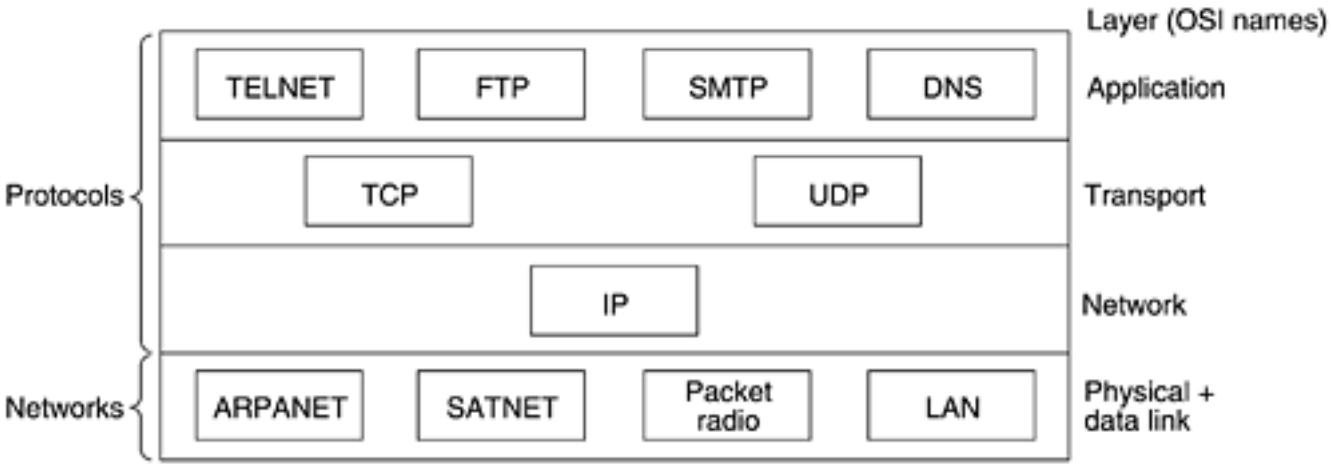

Figura 6 – Pilha TCP/IP e seus protocolos.
Fonte:(TANENBAUM, 2003)

Essas camadas são denominadas:

- Camada de aplicação: onde 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 transporte: onde 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 da mesma e um endereço de destino a camada de rede. A camada de rede prove o serviço de entrega do segmento a 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 lida com meio de 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 desenvolvido e mantido por uma rede mundial de desenvolvedores (DUNKELS, 2002).

Possui três *Aplication Programing interfaces* (APIs):

- RAW API (API Crua): É a API nativa do LwIP, possui a 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, entre outros, os seguintes protocolos:

- 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.

A seguir serão explanados com mais profundidade alguns dos protocolos implementados pela LwIP.

2.2.1.1 HYPERTEXT TRANSFER PROTOCOL (HTTP)

Segundo Kurose e Ross (2010), o protocolo da camada de aplicação HTTP é implementado em dois programas, um programa cliente e outro servidor. Os dois são executados em sistemas finais diferentes, se comunicam entre eles por meio de uma troca de mensagens HTTP. O 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.

2.2.1.2 TRANSMISSION CONTROL PROTOCOL (TCP)

Segundo Tanenbaum (2003), O 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 muitos categorias de falhas que podem ocorrer.

Cada máquina 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 contem dados TCP chegam a uma máquina, 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 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.

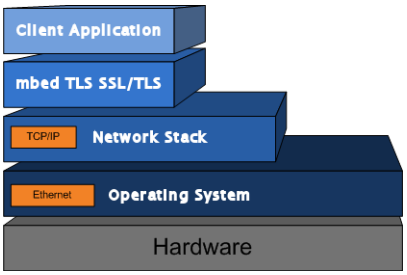

Figura 7 – Pilha de comunicação.
Fonte:(DEVINE, 2006)

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 nos fornece peças de software contendo algoritmos criptográficos, que podem ser facilmente acoplados a qualquer aplicação. Como é o caso do algoritmo SHA-2 que ficará responsável por fazer a verificação da integridade dos arquivos baixados.

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 processamento e execução do

processo que os criou e tornar as informações independentes de quaisquer processos. Existe a necessidade da criação de uma estrutura de armazenamento de informações a longo prazo. Os três requisitos fundamentais para essa estrutura são:

- Deve ser possivel 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.

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. 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.

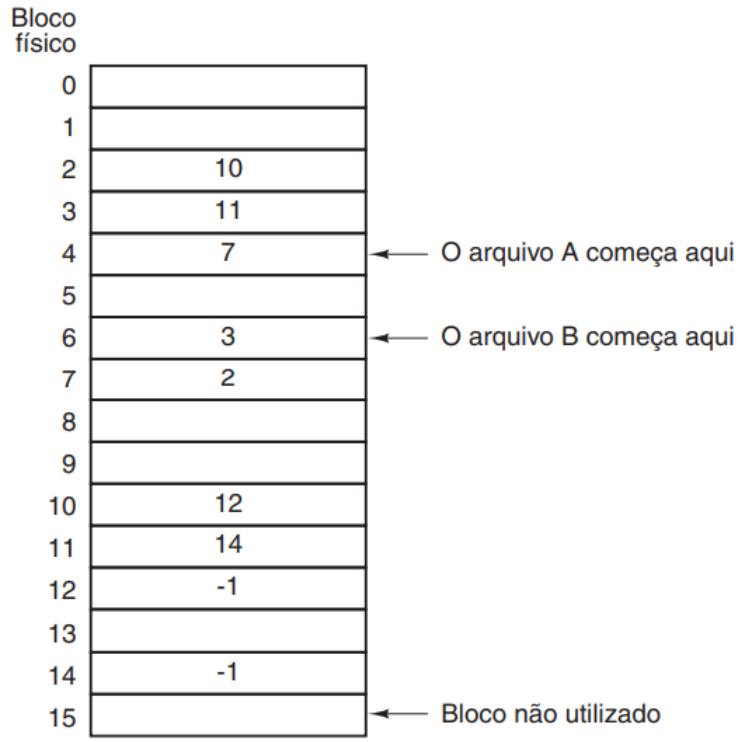

Figura 8 – Alocação encadeada usando tabela de alocação de arquivo.
Fonte:(TANENBAUM, 2007)

A principal desvantagem desse 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 tem de 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, dependendo do sistema para ser otimizado para espaço ou para tempo.

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.

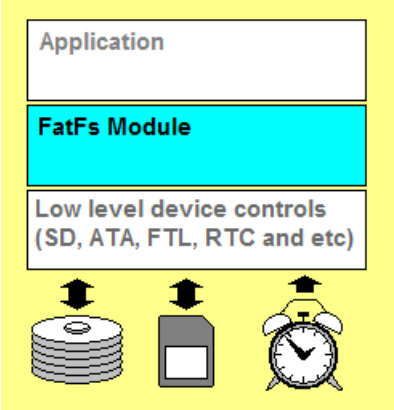

Figura 9 – Posição da biblioteca FatFs na aplicação.
Fonte:(CHAN, 2016)

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 10. Umas das principais funções fornecidas pela FatFs para a aplicação são:

- 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.

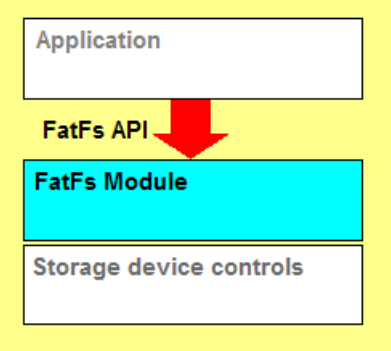

Figura 10 – Manipulação da memória por meio da API da FatFs.
Fonte:(CHAN, 2016)

2.4 TRABALHOS CORRELATOS

Foram identificados três trabalhos correlatos.

- **Firmware over the air for automotive, Fotamotive**: Esse trabalho introduz uma solução para atualização de veículos com a ajuda de fabricantes de equipamentos originais para reduzir custos, *recalls* aumentar a qualidade, 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 prove descoberta de rotas, estabelecimento e um protocolo de malha de baixa potência (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 capitulo é retratado como será o funcionamento do sistema que será desenvolvido, mostrada uma visão geral do projeto, listada as funcionalidades de cada uma das partes do *software*, explicando sua atividade, sua implementação, e as ferramentas utilizadas para o seu teste e os materiais utilizados.

3.1 VISÃO GERAL

O *software* que será desenvolvido nesse trabalho é dividido em duas partes, uma contendo o *bootloader*, com o auxílio da biblioteca FatFs, que é um módulo genérico do sistema de arquivo FAT, realizará a comunicação com o cartão SD, que conterá o novo *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 dificilmente será substituída. Será uma parte que não é portável para todas as plataformas, ficando a cargo do projetista fazer o porte para outras placas.

A outra parte desse trabalho será uma API, contendo as demais funções necessárias para a comunicação com o servidor, que enviará para o sistema o novo *firmware*, por meio do uso da biblioteca LwIP, garantirá a segurança dessa conexão com a utilização de uma camada extra de proteção, com biblioteca Mbed TLS. Essa API irá conectar-se a um servidor em um intervalo de tempo determinado pelo projetista, em que verificará a disponibilidade de uma nova versão de *software*.

Após ser confirmada a existência de uma nova versão de *firmware*, em um tempo determinado pelo projetista a API irá se comunicar novamente com o servidor com o intuito de fazer o *download* dessa nova versão, e a armazenar em um endereço de memória predeterminado no cartão SD do sistema, para que então o *bootloader* entre em ação após uma reinicialização do sistema. 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, Mbed TLS, 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. Para que 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 e o sistema não responda após a atualização, o sistema tem a habilidade de se recuperar. Como não haverá sobreescrita

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.

O sistema de atualização irá funcionar da seguinte forma, após a inicialização do sistema o *bootloader* entra em ação e verifica a existência de um novo *firmware*, no caso negativo ele inicia normalmente o *software* principal da aplicação. No caso positivo, o *bootloader* inicia o processo de substituição de *firmware*, após a troca, o novo *software* é iniciado. Durante a execução da aplicação principal do sistema e em um tempo determinado pelo projetista a API entra em contato com o servidor para verificar a disponibilidade de um *software* novo, em caso positivo ela agenda um período para a atualização do *firmware*, no caso negativo ele continua a execução da aplicação.

Caso haja uma atualização quando o horário do agendamento chegar, a API irá conectar-se ao servidor e iniciar o processo de *download* do *firmware* novo e armazena-lo em uma área já predefinida do cartão SD para que o *bootloader* possa o encontrar. Após o *download* o sistema irá ser reiniciado e o *bootloader* entrará em ação novamente.

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 a plataforma STM32F7, 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* e a biblioteca FatFs que fará parte do *bootloader*. Também será responsável por fazer com que o *bootloader* seja chamado no lugar da função *main*. Assim podemos garantir que o *bootloader* sempre entre em ação 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 um novo *firmware* em um endereço de memória já predeterminado. 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 a aplicação final ser inicializada.

Se a procura retornar com um resultado positivo para a existência de uma nova versão de *software*, o *bootloader* inicia o processo de atualização. Nesse processo o *bootloader* substitui completamente o *software* e demais bibliotecas e API's em áreas não protegidas na memória, pelo binário do *firmware* presente no cartão SD. Esse processo será implementado com o uso da função memcpy, já conhecida e utilizada por diversos desenvolvedores, com o

intuito de se deixar o código do *bootloader* mais reutilizável. O funcionamento resumido do *bootloader* pode ser observado na Figura 11.

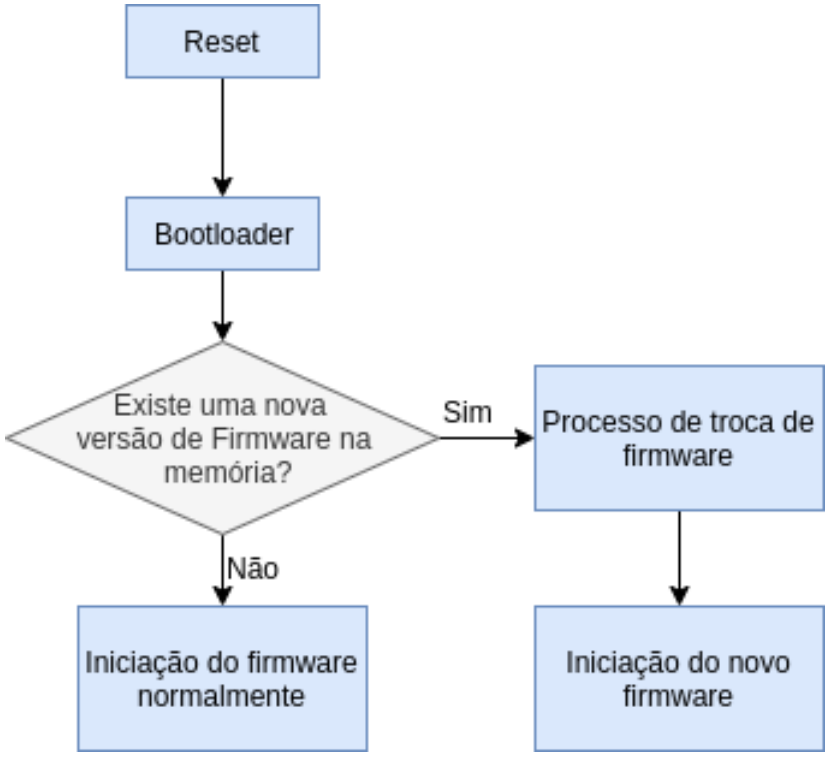

Figura 11 – Diagrama de funcionamento do *bootloader*.
Fonte: autoria própria.

3.3 API DE ATUALIZAÇÃO OTA

A API de atualização OTA que será 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*. Com esse objetivo, serão 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 utiliza-las em suas próprias funcionalidades. A Figura 12 ilustra de forma resumida o funcionamento da API.

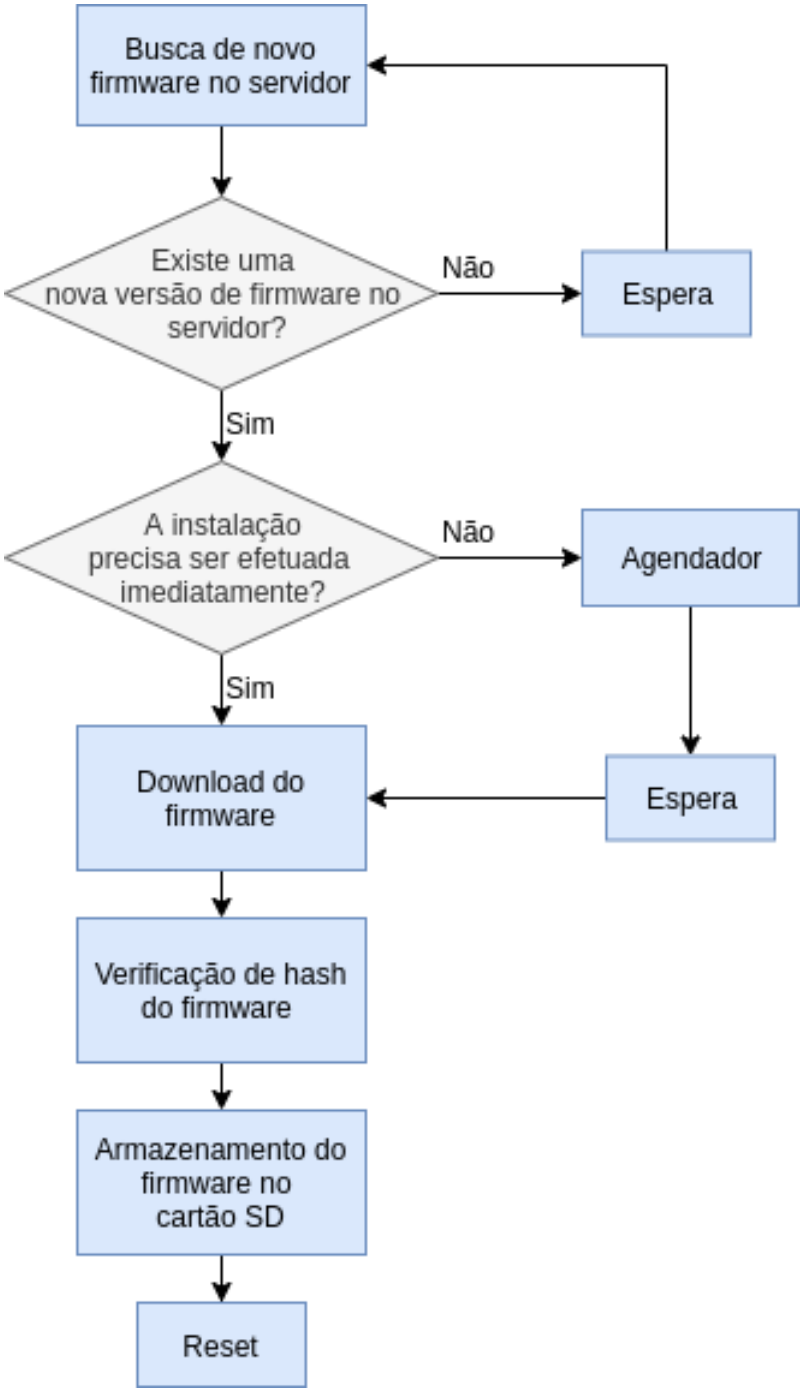

Figura 12 – Diagrama de funcionamento da API.
Fonte: autoria própria.

A seguir será retratado como serão cada uma das funcionalidades necessárias na API.

3.3.1 COMUNICAÇÃO COM O SERVIDOR

Com o uso da biblioteca LwIP e Mbed TLS será criada uma pilha de comunicação no sistema alvo, que ficará responsável pela conexão segura com o servidor que fornecerá o novo firmware. Na implementação da pilha de comunicação, será 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.

Como a biblioteca Mbed TLS já foi desenvolvida para ser integrada facilmente a várias aplicações embarcadas, ela será utilizada para criar protocolos de segurança nessa comunicação com o servidor. Serão 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, e também algoritmos de criptografia para o *firmware* que será enviado por meio dessa conexão.

3.3.2 VERIFICAÇÃO E AGENDAMENTO DE ATUALIZAÇÃO

O desenvolvedor será responsável em decidir a frequência e/ou horário em que a API irá consultar o servidor para a verificação de uma nova versão de *firmware*. A API evitará que em aplicações onde existam atividades críticas, elas sejam interrompidas para que a API possa fazer essa constatação, sabendo que a prioridade dessa ação deve ser a menor possível na aplicação.

Após a consulta, caso exista uma nova atualização disponível o desenvolvedor pode optar em a executar imediatamente, ou agendar para ser feita em outro horário, em que a aplicação esteja menos ocupada. Essa escolha pode ser programada desde o início do desenvolvimento do *firmware*, ou a partir de um sinal durante a consulta por uma nova atualização. Assim caso haja a necessidade da troca de *software* imediatamente, esse sinal pode avisar o agendador e já executar todo o processo de substituição de *firmware*.

3.3.3 DOWNLOAD E ARMAZENAMENTO DO FIRMWARE

Quando chegar no tempo esperado pelo agendador, a API iniciará novamente a comunicação com o servidor que contém o *firmware* novo, dessa vez com o propósito de fazer *download* da nova versão do *software*. Com o propósito de aumentar a segurança da API o *firmware* que se apresenta após a transferência ainda está criptografado, assim ele deve passar por uma descriptografia para então ser armazenado no cartão SD. Após essa descriptografia a API ainda irá verificar se o *firmware* recebido pelo servidor está correto a partir da verificação de *hash* criptográfica.

A partir da utilização da biblioteca FatFs, será 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 nessa localização, e fazendo com que o *bootloader* interprete de forma errada os arquivos gerando erros.

3.4 MATERIAIS UTILIZADOS

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

3.4.1 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 13 ilustra o kit STM32F746NGH6-Discovery.

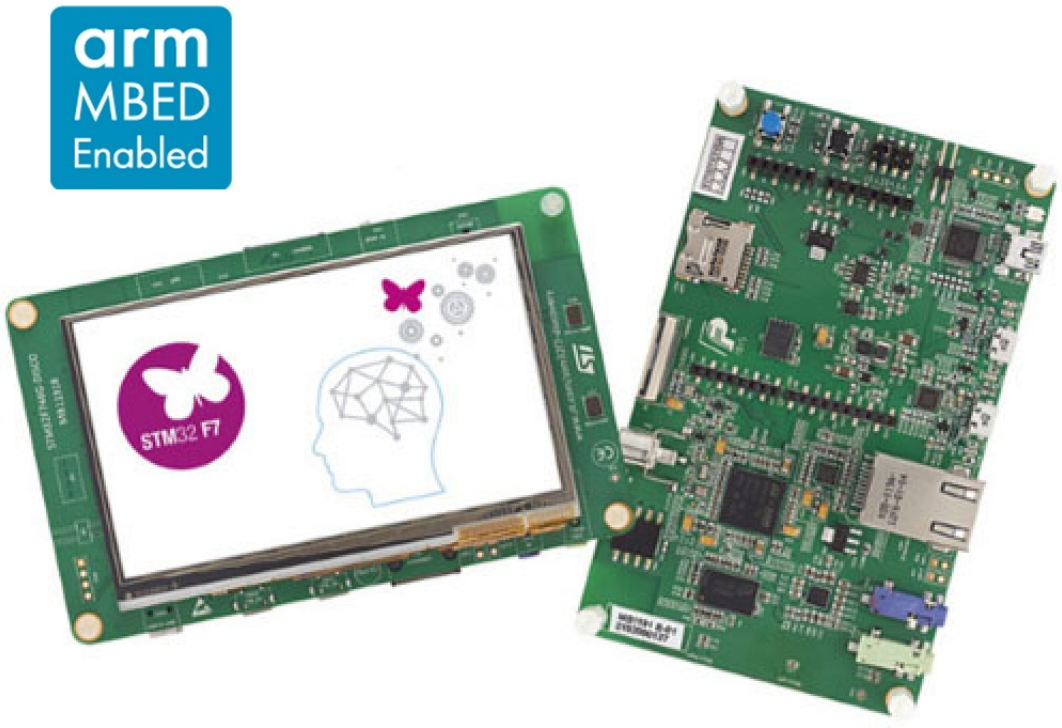

Figura 13 – Kit de desenvolvimento STM32F746G-Discovery.
Fonte:(STMICROELECTRONICS, 2019).

Algumas de suas principais características são:

- 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.

4 CRONOGRAMA

Para o desenvolvimento desta proposta foi definido o cronograma da Tabela 1

Tabela 1 – Cronograma para o desenvolvimento do projeto

Atividades	Ago	Set	Out	Nov	Dez	Jan	Fev	Mar	Abr	Mai	Jun
Estudo bibliográfico	X	X	X	X	X		X	X	X	X	X
Desenvolvimento da proposta	X										
Familiarização com plataforma	X	X	X	X							
Projeto do *bootloader*e API			X	X	X						
Entrega do TCC1					X						
Desenvolvimento do *bootloader*					X		X	X			
Implementação da API							X	X	X	X	
Testes de integração								X	X	X	X
Escrita da monografia e artigo científico	X	X	X	X	X		X	X	X	X	X

Referências

BALL, S. **Embedded Microprocessor Systems: Real World Design**. 3rd. ed. Newton, MA, USA: Butterworth-Heinemann, 2002. ISBN 0750675349. Citado na página 2.

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 4 vezes nas páginas , 2, 18 e 19.

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 19.

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 4 vezes nas páginas , 2, 6 e 7.

DEVINE, C. **SSL Library mbed TLS / PolarSSL**. 2006. Disponível em: <(https://tls.mbed.org)>. Acesso em: 06 de setembro de 2019. Citado 3 vezes nas páginas , 2 e 14.

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 14.

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 2 e 12.

ECLIPSE FOUNDATION. **Eclipse Foundation, Inc.** 2001. Disponível em: <(https://www.eclipse.org/)>. Acesso em: 28 de novembro de 2019. Citado na página 25.

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 2.

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 10, 13 e 15.

MARWEDEL, P. **Embedded System Design**. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 1402076908. Citado na página 1.

NOVIELLO, C. **Mastering STM32**. 2018. Disponível em: <(https://leanpub.com/mastering-stm32)>. Acesso em: 06 de setembro de 2019. Citado na página 8.

NSA. **National Security Agency**. 1952. Disponível em: <(https://www.nsa.gov/)>. Acesso em: 28 de novembro de 2019. Citado na página 15.

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 19.

QING, Y. C. L. **Real-time concepts for embedded systems**. [S.l.]: CRC Press, 2003. Citado 6 vezes nas páginas , 4, 5, 8, 9 e 10.

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 2.

SD CARD ASSOCIATION. **SD Card**. 2016. Disponível em: <(https://www.sdcard.org/)>. Acesso em: 28 de novembro de 2019. Citado na página 16.

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 2 vezes nas páginas e 25.

TANENBAUM, A. S. **Computer Networks**. [S.l.]: Pearson Education, inc, 2003. Citado 4 vezes nas páginas , 11, 12 e 13.

TANENBAUM, A. S. **OPERATING SYSTEMS DESIGN AND IMPLEMENTATION**. [S.l.]: Pearson Education, inc, 2007. Citado 4 vezes nas páginas , 15, 16 e 17.

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 19.