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 BASEADO NAS BIBLIOTECAS LWIP, MBEDTSL 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 BASEADO NAS BIBLIOTECAS LWIP, MBEDTSL 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

LISTA DE FIGURAS

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 mais pequenos 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 atendem 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, e 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 como um 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*, 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 necessária 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.

Este trabalho de conclusão de curso propõem 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 LwIP (DUNKELS, 2002), Mbedtls (DEVINE, 2006) e 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 neste 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).

1.1 OBJETIVO GERAL

Este trabalho de conclusão de curso tem como objetivo geral o desenvolvimento de um sistema *open-source* para a atualização de *firmwares OverThe Air* de sistemas embarcados baseado nas bibliotecas FatFs, LwIP e mbedTLS.

1.2 OBJETIVOS ESPECÍFICOS

- Desenvolver o *bootloader* que identifica versões e atualiza o *firmware*.
- Criar um servidor HTTP para a comunicação cliente-servidor.
- Implementar a tarefa que fará a comunicação segura entre o servidor HTTP e a plataforma embarcada, verificará de disponibilidade de atualização e fará o *download* da nova versão, se existente, será baseando nas bibliotecas LwIP e mbedTLS.
- Produzir a tarefa de leitura de um cartão micro SD que guarda uma nova versão do *firmware* e uma cópia do anterior, utilizando a biblioteca FatFs.
- Comprovar o funcionamento da técnica de atualização remota de *firmware*, utilizando a plataforma embarcada STM32F7.

2 REVISÃO DA LITERATURA

Neste capítulo será feita uma revisão da literatura necessaria para o entendimento deste trabalho de conclusão de curso, abordando os temas como, o processo de inicialisação de um sistema embarcado para ser compreendido como o sistema inicia e chama um bootloader e o papel do linker neste processo. Também será discutido sobre a comunicação do tipo cliente servidos que será utilizada neste trabalho, assim como uma breve resisão sobre pilhas TCP/IP e então a biblioteca LwIP, utilizada neste trabalho que ficará responsavel pela implementação desta pilha, assim como a biblioteca mbedTSL que ficara com o cargo de tornar essa comunicação segura.

Será exposto sobre o Cartão Secure digital, que ficará responsavel pelo armazenamento das versões de software que serão substituidas, assim como o Sistema de arquivos FAT que será utilizado neste trabalho e a biblioteca FatFs, que implementará o sistema de arquivos necessário para que sejam armazenadas as diferentes versões do firmware que substituirá o atual. Com o conhecimento passado sobre todos esses temas o leitor será capaz de compreender o método de atualização de firmware.

2.1 Processo de Inicialização do Sistema

Conhecer o processo de inicialização dos sistemas embarcados, também chamado de boot, é muito importante para nos dar noção sobre quando e como será entrará em execução tanto a função main, quanto um bootloader. Dependendo do fabricante do chip o boot pode ser diferente, porem em geral ele segue a mesma linha de execução, podendo ou não ter interfaces fisicas para mudar a origem do boot, como no caso de algumas placas da ST, que contem um mecanismo chamado de *physic remap*, para que o boot ocorra a partir de outras memórias e não da FLASH, que é a padrão (NOVIELLO, 2018).

Como dito por Beningo (2015), o fluxo padrão de boot de um sistema consiste em após a tensão de referência de um MCU se estabilizar, o processador procura pelo vetor de reset, para a obter a localização na FLASH da instrução de iniciação. O vetor de reset esta localizado em um endereço especial da memoria FLASH, geralmente no inicio. Como pode ser visto na imagem 1 que contém o mapa de memoria do MCU PIC24F16KLXXX, o vetor de reset se encontra no endereço 0x0002.

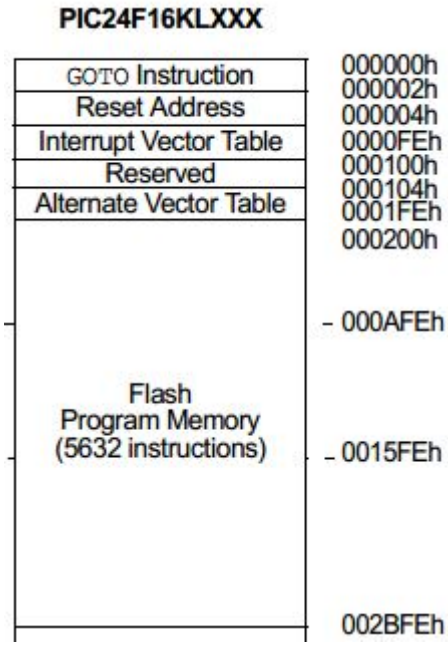

Figura 1 – Mapa de memória do MCU PIC24F16KLXXX.
Fonte: (BENINGO, 2015)

Então o endereço contido no vetor de reset é carregado pelo microcontrolador, e a instrução contida neste endereço é carregada e executada pelo processador. Essa instrução ainda não é a main que foi desenvolvida pelo projetista, ela é somente uma instrução de como o MCU irá iniciar. Esta instrução geralmente tem como função copiar o *Vector Table* que esta na memória FLASH para a RAM, copiando e escrevendo em um endereço determinado pelo arquivo de linker.

A variavéis que são inicializadas durante o tempo de compilação que estão armazenadas na seção .data do arquivo de linker são então gravadas na memória RAM, e as variavéis não inicializadas explicitamente ou iniciadas com zero que estão na seção .bss do arquivo de linker, são armazenadas na memória RAM. O MCU então copia funções que estão na FLASH para a RAM, estas etapa é opcional, o projetista deve determinas se alguma função é necessaria ser executada a partir da memória RAM.

Todo esse processo é chamado de *"C Copy Down"*, que faz a preparação do ambiente para iniciar uma aplicação, após essa etapa ser concluída o MCU pode então chamar a função main, que é a firmware principal do sistema desenvolvido pelo projetista. Toda a sequência padrão de boot pode ser observada na imagem 2.

Figura 2 – Sequencia de Boot.

É possivel alterar a função da tarefa contida no vetor de reset no arquivo de linker, fazendo com que o MCU passe pelo processo de *C Copy Down*, preparando o sistema para outra função diferente da aplicação principal do sistema embarcado, como é utilizado na criação de bootloaders para a atualização de firmware.

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 memória 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 3 mostra como geralmente fica alocado um bootloader e o firmware nas memorias.

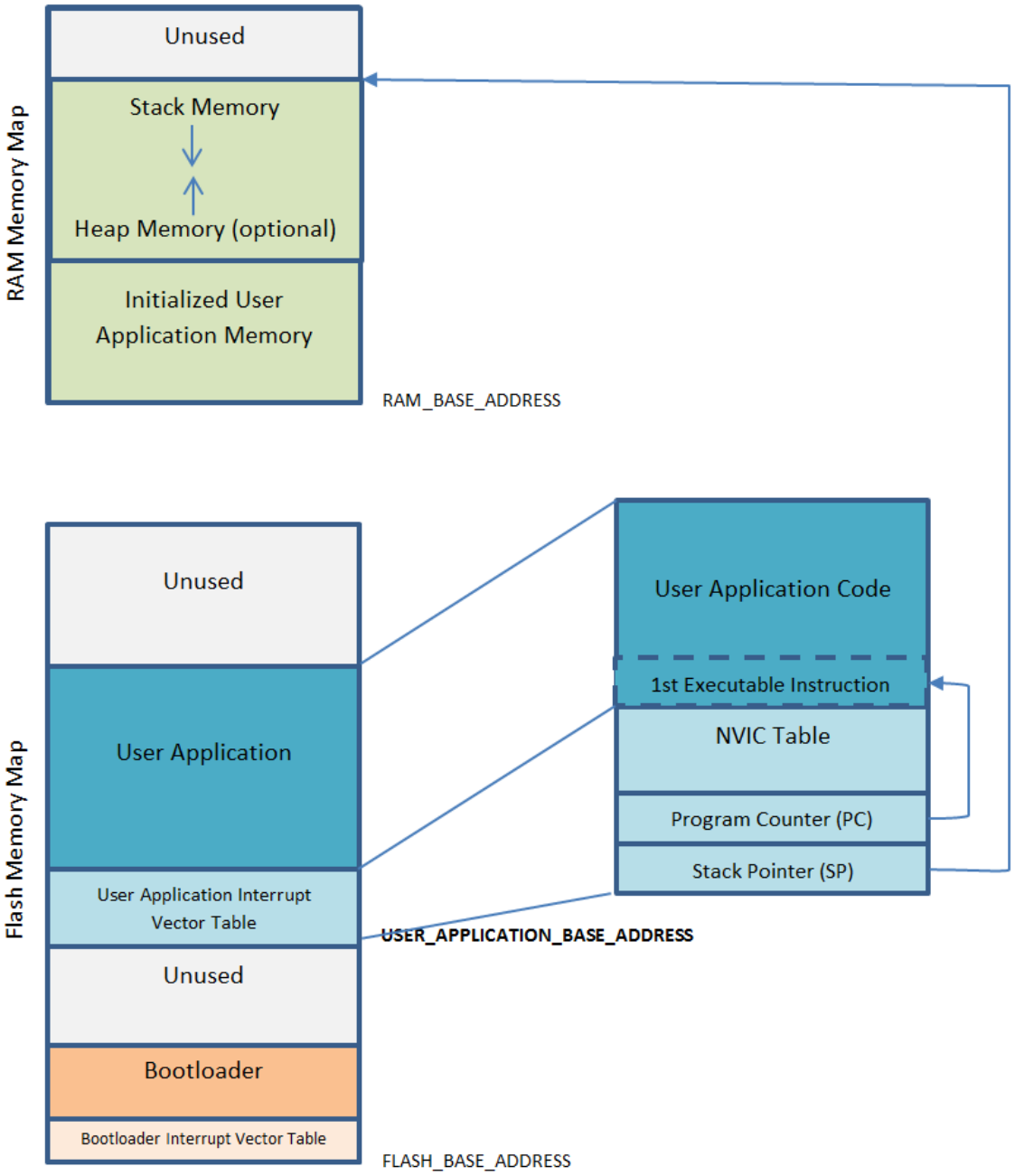

Figura 3 – Alocação do bootloader e firmware nas memórias flash e ram.

Sua função se resume a: comunicar-se com outro servidor, ler os arquivos enviados pelo host, atualizar o firmware de seu microcontrolador, e iniciar este novo firmware. Pode conter instruções e comandos definidos pelo projetista para somente o circuito integrado em uso, impossibilitando o uso do mesmo código em outras placas. Portanto, é uma peça de *software* que não é portável para várias plataformas. A imagem 4 mostra o funciomento de um bootloader padrão.

Figura 4 – Fluxograma de operações de um bootloader.

Os sistemas microcontrolados STM32 vem de fábrica com um bootloader pré programado na memória ROM, este bootloader pode ser utilizado por meio de diversos periféricos de comunicação, e para cada periférico diferente a ST padronizou diferentes protocolos que permitem varias operações, como: obter o ID do chip, escrever e ler bytes na memória RAM e FLASH, apagar setores das memórias, ativar áreas de proteção na memória e pular para o código principal do sistema (NOVIELLO, 2018).

É possivel ser criado um bootloader customizado com o diversas funções adicionais, uma frequentemente usada é o uso do bootloader para descriptografar firmwares que podem chegar de via internet, para se garantir a segurança e origem de um firmware, assim após esse processo o sistema pode substituir o software anterior pelo recebido.

2.1.2 Linker

Segundo Qing (2003), os arquivos fonte são processados pelo compilador e assembler, criando assim os arquivos objetos, que contem 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. O linker obtem 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 multiplos arquivos objetos em um maior arquivo objeto reloacável, um arquivo objeto compartilhado ou uma imagem executavel final. Esse processo pode ser observado na imagem 6.

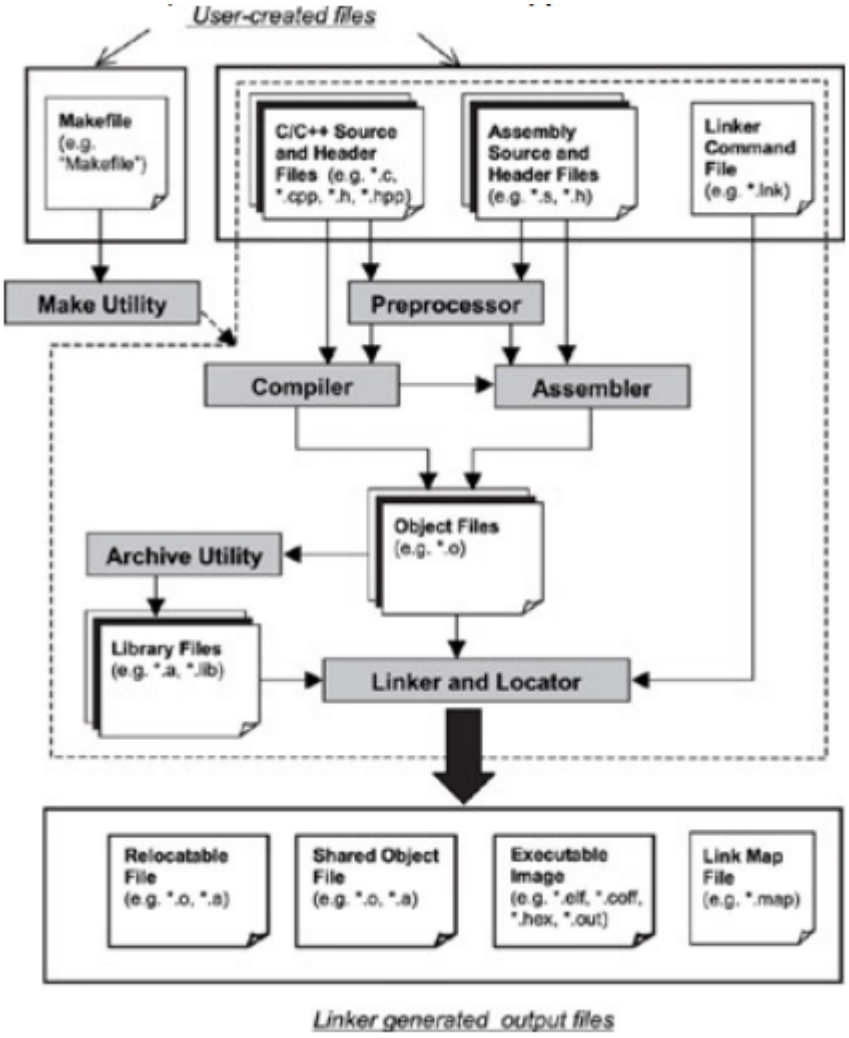

Figura 5 – Criando um arquivo de imagem para um sistema operacional alvo.
Fonte:(QING, 2003)

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.

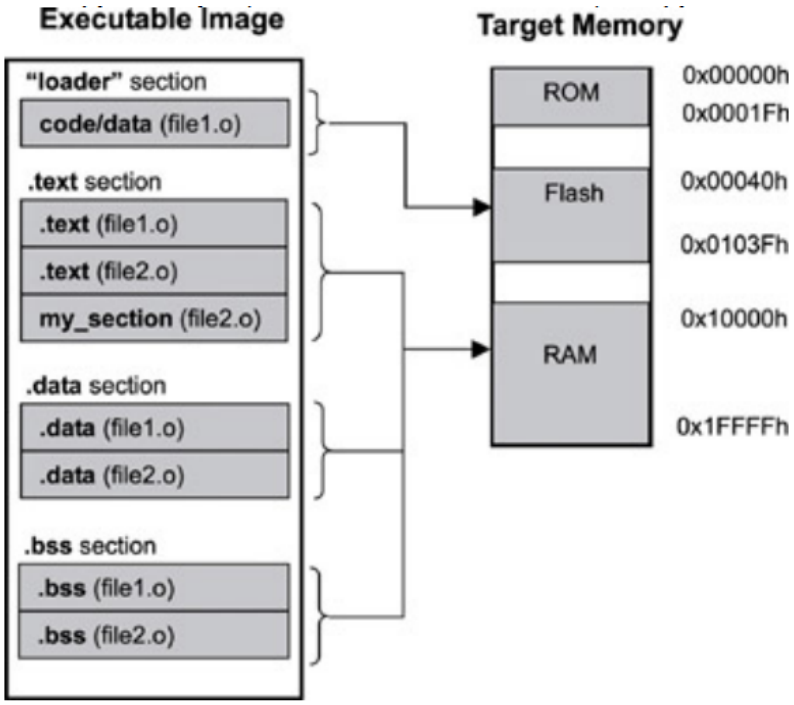

Figura 6 – Mapeando uma imagem executável em um sistema alvo.
Fonte:(QING, 2003)

2.2 TCP

Segundo Tanenbaum (2003), O TCP (*Transmission Control Protocol*) 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 paramêtros 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 maquina 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 maquina, 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. Neste trabalho será utilizada a biblioteca LwIP para o implementação da entidade de transporte TCP.

2.2.1 Lwip

A Biblioteca LwIP é uma implementação do protocolo 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 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 sequêncial de alto nivel que requer um sistema operacional de tempo real (RTOS). Habilita operações com multiplas threads.
- BSD Sockets API: API de sockets de Berkeley, desenvolvida em cima da API Netconn.

2.2.2 Mbed TLS

A biblioteca mbed TLS foi desenvolvida para se integrar facilmente a aplicações embarcados existentes, e fornecer os blocos de contrução para uma comunicação segura, criptografia e gerenciamento de chaves. Como o seu intuito é ser o mais flexivel possivel, 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 faz a comunicação intermediaria entre a aplicação final e a camada TCP/IP.

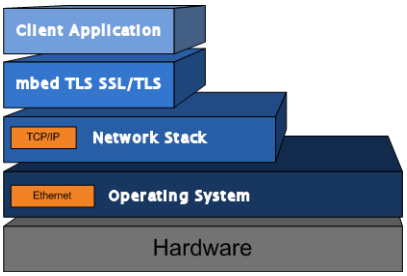

Figura 7 – Pilha de comunicação.
Fonte:(DEVINE, 2006)

2.3 **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. Nesta organização, o bloco de memória inteiro esta disponivel para dados. Além disso, o acesso aleatório é muito mais fácil. Mesmo que o encadeamento ainda tenha que ser seguido para

encontrar determino deslocamento dentro do arquivo, ele esta inteiramente na memória. De modo que pode ser seguido sem necessidade nenhhuma referencia de 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. Ambos terminam com um marcador especial que no caso é o número -1.

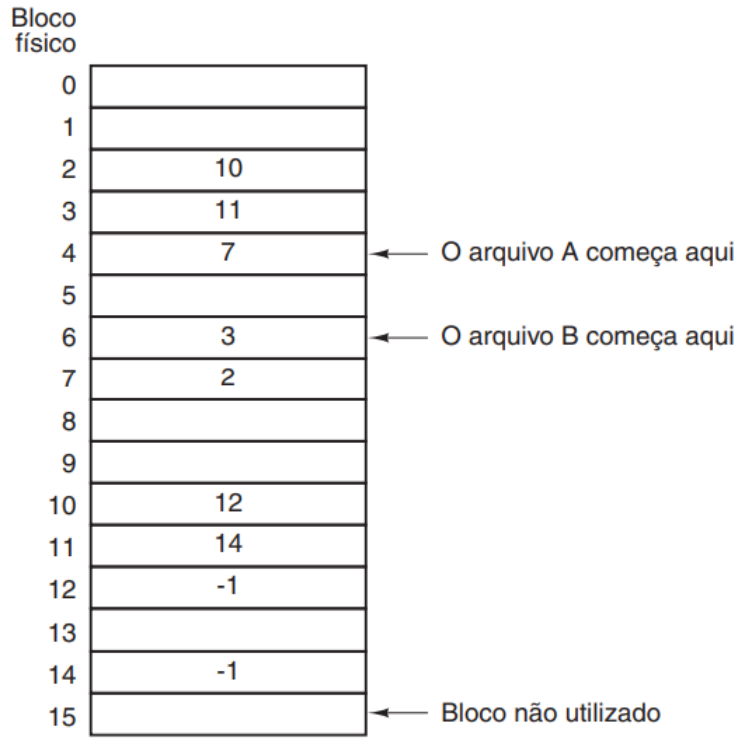

Figura 8 – Alocação encadeada usando tabela de alocação de arquivo.
Fonte:(TANENBAUM, 2007)

A principal desvantagem deste método é que a tabela inteira precisa estar na memória o tempo todo. Com um disco de 20GB e um tamanho de bloco de 1KB, a tabela precisa de 20 milhões de entradas, uma para cada um do 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 80MB de memória principal o tempo todo, dependendo do sistema para ser otimizado para espaço ou para tempo.

2.3.1 FatFs

FatFs é um modulo generico de um sistema de arquivo FAT/exFAT, para pequenos sistemas embarcados. É escrito em conformidade com a ANSI C (C89) e é completamente separado da camada de entrada e saida do sistema, portanto é independente da plataforma utilizada.

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

Neste capitulo é retratado como será o funcionamento do sistema que será desenvolvido, mostrada uma visão geral do software, listada as funcionalidades de cada uma das partes do software, explicada sua atividade e sua implementação, os problemas enfrentados durante o processo de desenvolvimento e as ferramentas utilizadas para o seu teste, Possiveis trabalhos futuros?.

3.1 VISÃO GERAL

O software que será desenvolvido neste trabalho é dividido em duas partes, uma contendo o bootloader, que com o auxilio da biblioteca FatFs, que é um modulo generico do sistema de arquivo FAT, realizará a comunicação com o cartão SD, que conterá o novo firmware previamente recebido, e assim poderá substituir o software anterior da aplicação por um novo. Essa parte do software ficará armazenada em uma região da memoria que não poderá ser reescrita, então é uma peça do programa que não poderá ser substitúida. Será uma parte que não é portavel para todas as plataformas, ficando a cargo do projetista fazer o port para outras placas.

A outra parte deste 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 comunicação com a utilização de uma camada extra de segurança com biblioteca Mbed TLS. Essa API irá conectar-se a um servidor em um intervalo de tempo determinado pelo projetista, para consulta irá verificar a disponibilidade de uma nova versão de software.

Após a confirmação de um novo firmware ser confirmada, em um tempo determinado pelo projetista a api irá se comunicar novamente com o servido com o intuito de fazer o download desta 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. A API será uma peça de software que poderá ser substituida e atualizada em conjunto com as demais aplicações do sistema, como, as bibliotecas LwIP, Mbed TSL, o sistema operacional, entre outras peças de software utilizadas pelo sistema.

Por conter biliotecas 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. Será utilizada o *kit* de desenvolvimento STM32F7 *Discovery*, onde será inicialmente desenvolvida a API e suas tarefas e o *bootloader* que serão os principais componentes desse sistema.

3.2 O *BOOTLOADER*

O *bootloader* será responsável em fazer a validação e troca de cada versão de *firmware* instalado no sistema embarcado. Sempre que o sistema for iniciado, o *bootloader* fará a procura de um novo *firmware* na memória interna do sistema. Esse processo, irá verificar o *hash* da nova versão, verificando a integridade e origem do *software*, para então poder ocorrer a atualização. Caso haja alguma falha durante esse processo, o *bootloader* terá a habilidade de verificar esse erro e corrigi-lo, revertendo a atualização, e instalando o *firmware* anterior. O funcionamento do *bootloader* pode ser observado na figura 9.

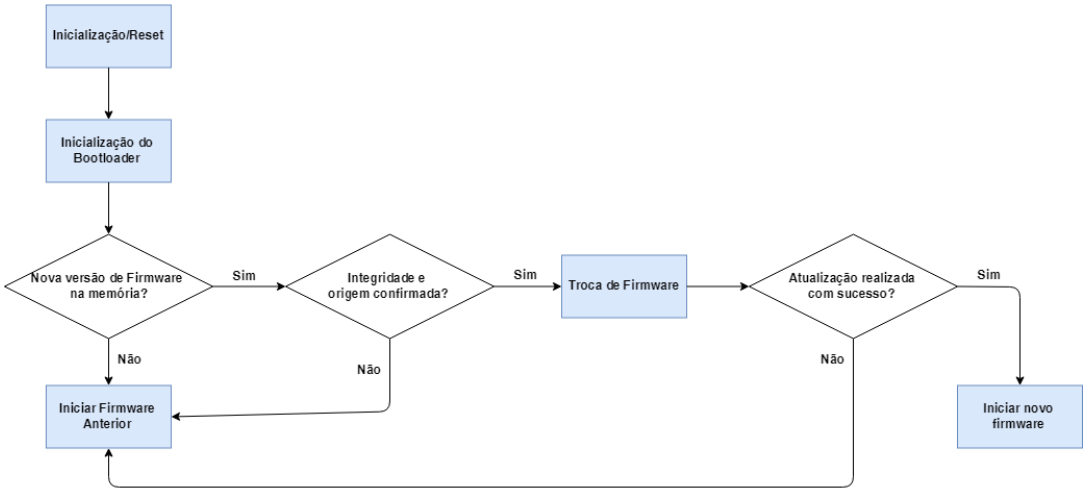

Figura 9 – Diagrama de funcionamento do *bootloader*. Fonte: autoria própria.

3.3 SERVIDOR HTTP

A partir de um computador conectado à mesma rede que o sistema embarcado, haverá um servidor HTTP que ficará responsável por esperar requisições do dispositivo, para consultar a disponibilidade de uma versão atualizada do *software*, e após a confirmação dessa nova versão, esse servidor irá enviar o *firmware* para a plataforma embarcada.

3.4 TAREFAS DO SISTEMA

3.4.1 COMUNICAÇÃO

Com o uso da biblioteca LwIP e MbedTLS essa tarefa estará responsável por criar uma comunicação segura entre o *hardware* e o servidor HTTP, a partir dessa comunicação será feito a verificação do sinal de disponibilidade de novo *software* e *download* do mesmo quando o sistema estiver ocioso.

3.4.2 ARMAZENAMENTO

A partir da utilização da biblioteca FatFs, essa tarefa fará a leitura e escrita sobre um cartão SD instalado no *hardware*. Quando houver a transferência de uma nova versão, tanto o programa anterior quanto o novo, serão colocadas nesta memória, para futuras instalações que serão realizadas pelo *bootloader*. A escolha da mídia de armazenamento de versões do *software* estará a cargo do projetista, sendo escolhido para esse trabalho um cartão SD.

4 CRONOGRAMA

1. Estudo bibliográfico;
2. Desenvolvimento da proposta;
3. Familiarização com plataforma;
4. Projeto de *bootloader* e tarefas;
5. Entrega do TCC1;
6. Desenvolvimento do *bootloader*;
7. Criação do servidor HTTP;
8. Implementação das tarefas;
9. Testes de integração;
10. Escrita da monografia e artigo científico.

Atividade	**08/19**	**09/19**	**10/19**	**11/19**	**12/19**	**01/20**	**02/20**	**03/20**	**04/20**	**05/20**
1	X	X	X	X	X	X	X	X	X	X
2	X									
3	X	X	X	X						
4		X	X	X	X					
5					X					
6					X	X	X			
7						X				
8						X	X	X	X	
9								X	X	X
10	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.

BENINGO, J. **Embedded Basics – Understanding the Microcontroller Boot Process**. 2015. Disponível em: <(https://www.beningo.com/understanding-the-microcontroller-boot-process/)>. Acesso em: 25 de outubro de 2019. Citado 3 vezes nas páginas , 4 e 5.

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 na página 2.

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 2 vezes nas páginas 2 e 6.

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 11.

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 11.

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 2 vezes nas páginas 4 e 8.

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

TANENBAUM, A. S. **Computer Networks**. [S.l.]: Pearson Education, inc, 2003. Citado na página 10.

TANENBAUM, A. S. **OPERATING SYSTEMS DESIGN AND IMPLEMENTATION**. [S.l.]: Pearson Education, inc, 2007. Citado 3 vezes nas páginas , 11 e 12.