CENTRO UNIVERSITÁRIO DO PARÁ - CESUPA
ESCOLA DE NEGÓCIOS, TECNOLOGIA E INOVAÇÃO - ARGO
CURSO DE ENGENHARIA DA COMPUTAÇÃO

FELIPE GARCIA DOS SANTOS

ABORDAGEM TEÓRICA E PRÁTICA DA APLICAÇÃO DE LINUX EM SISTEMAS EMBARCADOS: UM TUTORIAL PARA A CONFIGURAÇÃO DA BEAGLEBONE BLACK

BELÉM
2021

FELIPE GARCIA DOS SANTOS

ABORDAGEM TEÓRICA E PRÁTICA DA APLICAÇÃO DE LINUX EM SISTEMAS EMBARCADOS: UM TUTORIAL PARA A CONFIGURAÇÃO DA BEAGLEBONE BLACK

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Engenharia da Computação na modalidade MONOGRAFIA.

Orientadora: Msc. Suzane Alfaia Dias
Coorientador: Eng. Dr. José Adolfo da Silva Sena

BELÉM
2021

Dados Internacionais de Catalogação-na-publicação (CIP)
Biblioteca do CESUPA, Belém – PA

Santos, Felipe Garcia dos

Abordagem teórica e prática da aplicação de Linux em sistemas embacados: um tutorial para a configuração da Beaglebone Black / Felipe Garcia dos Santos; Suzane Alfaia Dias, orient., José Adolfo da Silva Sena (coorient.) – 2021.

Trabalho de Conclusão de Curso (Graduação) – Centro Universitário do Estado do Pará, Engenharia de Computação, Belém, 2021.

1. Sistemas embarcados (Computadores). 2. Linux (Sistema operacional de computadores. I. Dias, Suzane Alfaia (orient.) II. Sena, José Adolfo da Silva (coorient). III. Título.

CDD 23º ed. 621.3815

FELIPE GARCIA DOS SANTOS

ABORDAGEM TEÓRICA E PRÁTICA DA APLICAÇÃO DE LINUX EM SISTEMAS EMBARCADOS: UM TUTORIAL PARA A CONFIGURAÇÃO DA BEAGLEBONE BLACK

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Engenharia da Computação na modalidade MONOGRAFIA.

Data da aprovação: 23/ 06 /21
Nota final: 10,0

Banca examinadora

Prof. Msc. Suzane Alfaia Dias
Orientadora e Presidente da banca

Eng. Dr. José Adolfo da Silva Sena
Coorientador

Prof. Msc. Moshe Dayan Souza Ribeiro
Examinador

Dedico este trabalho
a Deus, pela graça e dom da vida,
a minha familia, pelo apoio e sacrifício incondicionais,
aos meu amigos e conhecidos que me auxiliaram de alguma forma,
e a todos que acreditam na recuperação da indústria e da ciência nacional brasileira.

AGRADECIMENTOS

Este trabalho é fruto de um grande esforço intelectual; uma medida para reforçar a literatura técnica nacional com conteúdo útil, afim de auxiliar a formação de engenheiros capacitados para trabalhar na indústria (e aonde mais os sistemas embarcados forem requeridos). Isso não seria possível se não houvessem muitos me apoiando e me dando suporte, de todos os lados. Portanto, existem muitas pessoas e entidades as quais eu devo profundos agradecimentos.

Em primeiro lugar, agradeço a Eletronorte pela oportunidade de estágio dada em 2019, que me permitiu iniciar uma nova fase da minha vida, além de obter novos e grandiosos conhecimentos (não somente técnicos). Grande parte disso está representado na pessoa do meu supervisor, Eng, Adolfo Sena, responsável pelo projeto SIMMEDAQ e primeiro idealizador deste trabalho, o qual eu também absorvi para mim como uma meta pessoal. Também cabem agradecimentos a toda equipe do DEAM e aos outros funcionários, que tão bem me acolheram.

Em segundo lugar, agradeço a minha família, pelo apoio e suporte incondicionais. Todos, dentro das suas capacidades, contribuíram sobremaneira, não só para a minha formação acadêmica, mas também para a moldagem do meu caráter atualmente. Eu não posso deixar de destacar, nesse meio, a minha mãe, que sempre fez todo tipo de sacrifício por mim para que eu pudesse estar aqui, hoje. Este trabalho só foi possível graças a esse amor dos meus.

Em terceiro lugar, também é válido agradecer a minha orientadora (por parte da instituição), Suzane Dias. Embora eu já tivesse um plano de trabalho e uma estrutura praticamente completa de como eu planejava escrevê-lo, a professora Suzane me concedeu total apoio, vindo da sua experiência em trabalhos acadêmicos no mestrado, indicando melhorias para deixar este trabalho ainda mais completo. Definitivamente, orientar um trabalho desse porte não é tarefa fácil, portanto, a agradeço por ter aceitado essa árdua missão.

Em quarto lugar, este trabalho também representa o desfecho da minha formação de quatro anos e meio como Engenheiro da Computação. Portanto, o mesmo contém muitas experiências adquiridas nesse tempo. Não posso deixar de agradecer, portanto, ao suporte dado pela instituição de ensino, nas pessoas da coordenação, corpo docente e até os funcionários. Cada um, da sua maneira e na sua medida, contribuíram para esse resultado: desde uma dúvida tirada, um suporte dado, uma conversa produtiva, até o auxílio na resolução de problemas – agradeço muito a quem me ajudou nesses anos.

Em quinto lugar, gostaria de fazer uma menção especial a todos os meus amigos, especialmente os que eu fiz durante esse tempo de curso, já que estiveram mais presentes no meu dia a dia, me ajudando e me apoiando (e me suportando também). Sinto que, se eu começar a citar nomes, com certeza esquecerei de alguém. Apesar disso, agradeço especialmente ao Arthur Ichi, ao Rafael Veiga, ao Gabriel Melém e ao Renzo Andrade. Também agradeço aos grupos dos

quais participei, que me concederam ótimas experiências e momentos, em especial o grupo dos nossos projetos integrados e ao grupo de automação e robótica.

Apesar do que foi dito no parágrafo anterior, é necessário explicitar aqui o nome do meu amigo Heydrigh Ribeiro, nosso mestre em desenvolvimento web da turma. Agradeço imensamente pelo suporte dado no capítulo 15 – Interface web, visto que essa era uma área na qual eu ainda não domino (no momento em que esse texto foi escrito), além de ter sido o capitulo mais difícil de achar material (para sistemas embarcados). Seus conhecimentos e experiências foram cruciais para que eu pudesse organizar o capítulo e desenvolvê-lo. Também agradeço ao professor Moshe que, como integrante da banca, apontou um erro em uma imagem, neste mesmo capítulo.

Em último lugar e – quebrando o padrão – mais importante de todos, agradeço a Deus. Para quem conhece a minha história, sabe das coisas que eu tive que passar para chegar até aqui. Apesar disso, obtive resultados que eu jamais sonharia. Acredito que a Sua soberania sobre as circunstâncias, aliada ao seu amor incondicional, me fizeram chagar onde cheguei, de tal modo que, das coisas, planos e projetos que planejei, nenhum saiu com eu esperava – todos superaram as minhas expectativas, e esse trabalho é um deles. Definitivamente, devo dar todos os créditos a Ele.

Este trabalho contém muito de mim – conhecimentos, experiências, esforço e esperanças. Foi desenvolvido para que qualquer pessoa da área de TI que queira ingressar na área de Linux embarcado não se sinta tão deslocado (ou sem embasamento) como eu me senti. Portanto, eu também tenho que agradecer a você, leitor, que dedicou um tempo a ler esse documento. Espero que encontre conhecimento úteis e interessantes aqui, e que de algum modo eu lhe auxilie. Se você, assim como eu, também acredita no futuro dessa nação por meio da ciência e da tecnologia, então eu lhe devo o dobro de agradecimentos. Desejo uma boa leitura.

“A computer is like air conditioning – it becomes useless when you open Windows.”

Linus Torvalds

RESUMO

Com o advento da tecnologia, visível e ubíqua, veio, também, a necessidade de produção de mais dispositivos IoT, que nada mais são que sistemas embarcados, muitos deles, microprocessados. Juntamente com isso, nasce a oportunidade de mais vagas para profissionais de engenharia da computação. Porém, é notável que muitos acadêmicos da área desconhecem o passo a passo de construção de um sistema embarcado microprocessado, assim como o conceito de Linux embarcado. Este trabalho tem como objetivo desenvolver uma documentação que contenha o desenvolvimento de um sistema embarcado microprocessado em Linux, usando como base a placa de desenvolvimento BeagleBone Black. O processo será descrito na forma de relatório, utilizando pesquisa bibliográfica e prática com o ambiente de desenvolvimento Texas Instruments.

Palavras-chave: Linux. Sistemas Embarcados. Microprocessadores. BeagleBone Black. IoT.

ABSTRACT

With the advent of technology, visible and ubiquitous, there was also the need to produce more IoT devices, which are nothing but embedded systems, many of them, microprocessed. Along with this, the opportunity arises for more vacancies for computer engineering professionals. However, it is notable that many academics in the field are unaware of the step-by-step construction of a microprocessed embedded system, as well as the concept of embedded Linux. This work aims to develop a documentation that contains the development of an embedded microprocessed system in Linux, using as basis the BeagleBone Black development board. The process will be described in the form of a report, using bibliographic and practical research with the Texas Instruments development environment.

Keywords: Linux. Embedded Systems. Microprocessors. BeagleBone Black. IoT.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Hodiernamente, a tecnologia está presente em todos os ramos da sociedade; onipresente na vida cotidiana (SOUZA, 2018; GARCIA, 2018). Presenciou-se uma rápida evolução, nas últimas décadas, das áreas da eletrônica, dos computadores pessoais, equipamentos eletrônicos em geral, e assim por diante (GARCIA, 2018). Dessarte, os sistemas embarcados, expoentes nessa evolução tecnológica, têm ganhado bastante notoriedade, sendo cada vez mais presentes no dia a dia, aborda Garcia (2018).

Conforme cita Souza (2018), os sistemas embarcados revolucionam o mundo continuamente melhorando a vida das pessoas, impulsionando o desenvolvimento tecnológico de todas as áreas de conhecimento humano. Segundo Stallings (2010), atualmente, bilhões de sistemas computacionais são produzidos anualmente e aplicados em produtos dos mais variados segmentos. Esses sistemas, na sua maioria embarcados, são considerados como pervasivos ou ubíquos, devido ao fato de passarem despercebidos no dia a dia (PATTERSON; HENNESSY, 2012; SOUZA, 2018), impulsionando uma caminhada a era da computação ubíqua (SOUZA, 2018).

Um sistema embarcado é definido como um sistema computacional - isto é, composto de hardware e software (STALLINGS, 2010; REIS, 2015) - que possui uma função dedicada, ou seja, é construído para uma aplicação específica (STALLINGS, 2010; REIS, 2015; EMBARCADOS, 2013), geralmente operando dentro de um outro sistema, elétrico, mecânico, hidráulico, etc. (STALLINGS, 2010; REIS, 2015) e que é desenvolvido para não ser programado ou alterado após a sua fabricação (HEATH, 2003; STALLINGS, 2010; REIS, 2015). Esses objetos, de acordo com Souza (2018) e Reis (2015), geralmente contam com uma quantidade reduzida de recursos, como memória e poder de processamento, além de requisitos tais como: tamanho reduzido, baixo consumo energético, tamanho reduzido, operação em tempo real, entre outros.

As aplicações dos sistemas embarcados são múltiplas, abrangendo praticamente tudo o que é programável, conforme Reis. Os sistemas embarcados são, geralmente, classificados em quatro áreas (REIS, 2015) - computação geral, sistemas de controle, processamento de sinais e comunicação, e abrangem diversos aparelhos, tais como: smartwatches, CLPs, eletrodomésticos, controles de bordo automotivos, automação residencial, sistemas de monitoramento médico e muitos outros (REIS, 2015; SOUZA, 2018; EMBARCADOS, 2013).

Em relação a arquitetura base, os sistemas embarcados são separados, basicamente, em dois tipos: os microcontrolados e os microprocessados, cada um tendo o seu cerne em um microcontrolador e um microprocessador, respectivamente (VAHID; GIVARGIS, 1999). No que diz respeito à engenharia, os microcontroladores diferem dos microprocessadores na medida em que aqueles possuem periféricos como RAM, ROM e Clock em um único chip, enquanto

que esses possuem apenas o processador em si; fato esse que se justifica pelo fato de que os últimos em questão são mais potentes em processamento (COMPONENTS101, 2019) No que diz respeito à aplicabilidade, os microcontroladores são usados em sistemas de tarefas únicas, de baixa ou média complexidade; já os microprocessados, por sua vez, geralmente são usados em situações multi-tarefas, onde pode-se tirar vantagem do uso de um sistema operacional, que gerencia o acesso de programas ao hardware (COMPONENTS101, 2019).

O desenvolvimento de produtos tecnológicos, atualmente, têm se focado na criação de sistemas embarcados, tanto microcontrolados, quanto microprocessados, visando, principalmente, a aplicação desses em projetos de IoT, seja ela residencial, industrial, agrícola, entre muitas outras (SOUZA, 2018). Esse crescimento na produção de tecnologia embarcada têm proporcionado um aumento na demanda por profissionais capacitados na área, sobretudo engenheiros de computação especializados em sistemas operacionais para embarcados.

Apesar do que foi explicitado, dentre os profissionais da engenharia da computação, especialmente entre aqueles que estão na graduação, atualmente nota-se uma deficiência na formação acerca do desenvolvimento de sistemas embarcados microprocessados. Embora esse fenômeno não se observe na área dos microcontroladores, principalmente pelo advento do Arduino, é notável que a maioria desconhece o processo de configuração e/ou desenvolvimento de um sistema embarcado microprocessado. Grande parte disso deve-se, principalmente, ao fato de que há uma carência de materiais sobre o assunto na literatura, conforme citam Prado (2011), Yaghmour et al. (2008), assim como a própria natureza do conteúdo, por vezes difícil, afaste os interessados.

Uma atitude para a situação exposta é necessária, sobretudo face a necessidade da indústria, agora e no futuro, de mão de obra especializada para lidar com as requisições da sociedade moderna. Para isso, é necessário uma pesquisa que busque organizar os muitos conceitos que envolvem a área - por vezes soltos, da mesma maneira que demonstre o processo de configuração de uma placa de desenvolvimento microprocessada de forma sistematizada. Tal documento é muito escasso, e pesquisas sobre o assunto o são ainda mais, inclusive nas principais bases de pesquisa, o que ilustra a inovação de tal proposta.

O presente trabalho visa produzir um tutorial acerca do processo de configuração de uma placa de desenvolvimento (sistema embarcado), explicando conceitos do sistema operacional Linux onde for possível. A pesquisa será feita utilizando a placa de desenvolvimento *open hardware* BeagleBone Black, cujo processador é da arquitetura ARM e o SDK é cedido pela Texas Instruments. Serão explanados os conceitos básicos acerca da área, um relatório de configuração da placa seguindo a documentação oficial da Texas Instruments (assim como outros materiais) e a demonstração de algumas aplicações práticas.

1.1 SITUAÇÃO PROBLEMA

No meio acadêmico da engenharia de computação, em especial na graduação, o conhecimento acerca do desenvolvimento de sistemas embarcados microcontrolados é bem conhecido (isto é, as suas etapas, desde o início - protótipo em placa de desenvolvimento - até o final do processo - placa standalone). Grande parte disso deve-se à popularidade da placa de desenvolvimento Arduino, a qual possui bons e acessíveis modelos de hardware e software, com ampla documentação (MADEIRA, 2018). No entanto, quando se trata do desenvolvimento de sistemas embarcados microprocessados, percebe-se uma deficiência desse conteúdo na formação acadêmica, desconhecendo, por exemplo, quais são as etapas correlatas no desenvolvimento (considerando as etapas mostradas acima).

Além disso, os materiais acerca de sistemas embarcados microprocessados são rarefeitos e escassos (PRADO, 2011; YAGHMOUR et al., 2008). Falta literaturas de referência e, quando existem, geralmente não são acessíveis a iniciantes, levando um indivíduo que queria ingressar/se aprofundar na área a sentir bastante dificuldade para organizar o conhecimento.

Em relação ao sistemas operacionais, é conhecido, na área da computação, que o Linux possui um gigantesco potencial de adaptação em diversos tipos de dispositivos e arquiteturas, desde sistemas muito reduzidos até a mainframes (MACIEL, 2014). Porém, entre os alunos de graduação, o conhecimento acerca de Linux geralmente gira em torno apenas de distribuições e comandos do *shell*, sendo o conhecimento do kernel e coisas mais profundas quase desconhecido. Uma grande possibilidade de aplicação do conhecimento de kernel, assim como uma boa oportunidade de aplicação prática é a adaptação do kernel Linux para sistemas embarcados microprocessados (MACIEL, 2014), assunto esse também raro entre os acadêmicos, assim como também é deficiente de material.

Existem algumas situações em que há um estudo envolvendo Linux e microprocessadores na graduação, geralmente envolvendo o computador de placa única Raspberry Pi. Porém, esse estudo é bastante superficial, limitado a códigos de *scripts*, sem envolver alterações em níveis mais profundos de hardware. Ademais, a placa Raspberry Pi não é aberta, ou seja, não é possível montar uma placa própria baseada nela, assim como é possível em outras, como Arduino.

À luz do que foi dito, é necessário produzir uma documentação acerca do processo de configuração de um sistema embarcado tomando como base uma placa de hardware aberto. Logo, esse trabalho busca responder a seguinte pergunta problema: Como configurar um sistema embarcado microprocessado utilizando Linux como núcleo?

1.2 OBJETIVOS DO ESTUDO

1.2.1 Objetivo Geral

Desenvolver um documento contendo o passo a passo do processo de configuração de uma placa de desenvolvimento microprocessada, conciliando teoria (conceitos) e prática (demonstrações).

1.2.2 Objetivos Específicos

- Introduzir a área dos sistemas embarcados microprocessados baseados em Linux;
- Preparar o ambiente de desenvolvimento embarcado, em software e hardware;
- Configurar os softwares fundamentais ao funcionamento de um dispositivo baseado em Linux;
- Demonstrar a criação de um sistema de arquivos embarcado;
- Instalar utilitários responsáveis pelas interfaces do sistema embarcado;
- Desenvolver uma aplicação embarcada como exemplo;
- Demonstrar os conceitos e demonstrações abordados com um exemplo de sistema embarcado real.

1.3 JUSTIFICATIVA

É mandatório que haja uma literatura sobre o tema explanado, de modo que permita a evolução do conhecimento na área, servindo como referência bibliográfica para futuros trabalhos e base para desenvolvimento de produtos reais.

Outrossim, as principais bases de pesquisa (IEEE, Scielo, Google Acadêmico) não apresentam nenhum trabalho que possua a mesma proposta deste (embora haja alguns correlatos), o que demonstra a sua originalidade.

1.4 METODOLOGIA DA PESQUISA

O intuito da pesquisa é demonstrar o passo a passo do processo de configuração da placa BeagleBone Black conforme a documentação oficial e o SDK da Texas Instruments, também adicionando personalizações quando necessário. Todo o processo será documentado na forma de relatório. Além disso, também se demonstrarão conceitos da área ao longo do texto, conforme a necessidade.

A pesquisa é de natureza exploratória, pois pretende promover a propagação do conhecimento da área dos sistemas embarcados microprocessados. Quanto ao delineamento, é majoritariamente bibliográfica, mas também aplicando a prática via experimentação na placa BeagleBone Black.

A pesquisa prática se sucederá em um ambiente de desenvolvimento composto de: Placa de desenvolvimento (*target*), um computador para controle e testes (*host*) assim como outros componentes necessários, como: conversor FTDI para USB, cabo de rede, entre outros. Além disso, far-se-á uma extensa pesquisa bibliográfica com o intuito de embasar os conceitos que forem surgindo.

Os dados da pesquisa serão coletados via testes e prática no ambiente de desenvolvimento. A documentação do processo será registrada via capturas de tela, fotos e outros meios quando forem necessários.

1.5 ESTRUTURA DO TRABALHO

Este trabalho está dividido em: Introdução (capítulo 1), Referencial Teórico (capítulo 2) e Desenvolvimento da Proposta (capítulo 3) - que são os capítulos introdutórios; Partes I a VII - referentes a documentação (ou desenvolvimento) em si; Considerações finais (capítulo 19) e, por fim, um apêndice A.

2 REFERENCIAL TEÓRICO

Esse capítulo abordará os conceitos das áreas bases para a elaboração desse trabalho – tópicos 2.1 a 2.3, fará uma revisão do estado da arte – tópico 2.4 e analisará os trabalhos relacionados – tópico 2.5. Serão tratados, como embasamento teórico, os temas de sistemas embarcados, sistemas operacionais, Linux embarcado, placas de desenvolvimento e a BeagleBone Black.

2.1 SISTEMAS EMBARCADOS

Os sistemas embarcados são dispositivos que combinam hardware e software para realizar uma tarefa específica (ou dedicada). Baseados, geralmente, em microprocessadores ou microcontroladores, esses sistemas são construídos para controlar uma (ou várias) função(ões) (STALLINGS, 2010; VAHID; GIVARGIS, 1999), sendo comumente aplicados em conjunto a outros sistemas – mecânicos, por exemplo (WOLF, 2012). Além disso, tais objetos não são desenvolvidos para permitir a alteração do sistema pós-construção, ou seja, o usuário do sistema não pode modificar seu software ou mesmo o seu hardware (HEATH, 2003).

Os dispositivos embarcados existem em quantidade muito maior que os computadores de uso geral, pelo fato de cobrirem uma gama muito grande de aplicações (STALLINGS, 2010). Bilhões de sistemas embarcados são produzidos ano após ano, em escala crescente. É possível dizer, ainda, que praticamente todos os dispositivos que funcionam à energia elétrica têm, ou terão, um dispositivo embarcado embutido (VAHID; GIVARGIS, 1999).

De um modo geral, os sistemas embarcados possuem as seguintes características (VAHID; GIVARGIS, 1999; STALLINGS, 2010; WOLF, 2012): função única e específica; restrições de tamanho, custo e consumo energético; reação em tempo real; aplicação em condições ambientais mistas; flexilidade e segurança. Vale destacar que essas características costumam variar dependendo dos requisitos do projeto, já que para cada aplicação há uma necessidade específica (STALLINGS, 2010).

Diversas das situações envolvendo sistemas embarcados não envolvem problemas da engenharia de computação propriamente. Ao invés, as áreas que mais demandam dispositivos desse tipo são as de aplicações em engenharia, sobretudo as que envolvem mecânica ou termodinâmica, seja em medição de variáveis ambiente ou controle (WOLF, 2012). Por conta disso, esses sistemas são geralmente inseridos junto a outros sistemas maiores; por isso o nome embarcado (STALLINGS, 2010).

O mercado de sistemas embarcados é significativo, é está crescendo ano a ano. Tais equipamentos são encontrados em uma grande variedade de eletrônicos comuns no dia a dia (VAHID; GIVARGIS, 1999), tais como: celulares, câmeras, assistentes virtuais, equipamentos

de segurança, automação residencial, entre muitos outros. Além disso, são também encontrados em automóveis (sistemas de ignição e controle de motor, por exemplo), aparelhos médicos (monitores cardíacos e bombas de infusão) e controles industriais (robótica e sistemas de controle) (STALLINGS, 2010; WOLF, 2012).

Com o avanço da tecnologia, percebeu-se que era extremamente vantajoso utilizar sistemas embarcados para controlar o mundo real (HEATH, 2003). Dentre essas vantagens, destacam-se: A substituição dos circuitos baseados em portas lógicas discretas; a evolução das funcionalidades de sistemas; manutenção facilitada; proteção de propriedade intelectual, entre outras.

Existem diversos tipos de sistemas embarcados. Baseados em um modelo computacional (isto é, com uma arquitetura definida), são dois os padrões: microcontrolados e microprocessados (VAHID; GIVARGIS, 1999). Existem também outros, como os FPGAs (Matriz de Portas Programáveis), os DSPs (Processadores de Sinais Digitais) e os baseados em circuitos integrados e portas lógicas (ASICs e outros) (WOLF, 2012). Porém, em virtude desses fugirem do escopo desse trabalho, não serão aprofundados.

Os sistemas embarcados baseados em processadores são dois: microcontrolados e microprocessados, que se caracterizam por terem como o seu controlador principal um microcontrolador e um microprocessador, respectivamente (COMPONENTS101, 2019). A principal diferença entre os dois é que o microcontrolador engloba todos os periféricos (RAM, ROM, clock, etc.) em um único CI, enquanto que o microprocessador conta apenas com a CPU, necessitando de conexões externas com os periféricos para funcionar. Além disso, os microcontroladores são de tarefas únicas e mais simples, enquanto que os microprocessadores são em geral multitarefa e capazes de executar ações mais complexas (COMPONENTS101, 2019).

2.2 SISTEMAS OPERACIONAIS E LINUX EMBARCADO

Um sistema operacional é um software que controla o acesso de programas do usuário ao hardware, fornecendo camadas de abstração para facilitar a modularização do desenvolvimento de programas (TANENBAUM; BOS, 2016). Suas funções principais são duas, conforme Tanenbaum e Bos (2016): fornecer a programadores (e programas) uma camada mais fácil de ser compreendida e gerenciar recursos de hardware.

Existem diversos tipos de sistemas operacionais, cada um para uma aplicação específica. Dentre esses, destacam-se (TANENBAUM; BOS, 2016): os de computadores de grande porte, os de servidores, desktops, mobile, de tempo real e embarcados. Esses últimos são utilizados em sistemas embarcados microprocessados, nos quais se caracterizam, principalmente, por não serem configuráveis pelo usuário ou terceiros, implicando em simplificações no design (TANENBAUM; BOS, 2016).

Existem, também, diversas marcas de software de sistemas operacionais. No meio dos

computadores pessoais, destacam-se o Windows, Linux e MacOS; enquanto que no meio mobile destacam-se o Android e iOS. Em quesito de portabilidade, os sistemas baseados em Linux se destacam, principalmente devido a sua performance (TANENBAUM; BOS, 2016).

Seguindo nessa linha, o Linux, que é um núcleo de sistema operacional, é um software de código aberto criado por Linus Torvards, tendo sua primeira versão lançada em 1991. O desenvolvimento do sistema operacional Linux contou com a colaboração de milhares de desenvolvedores desde então, e novos aplicativos foram adicionados que permitiram o uso do sistema nos mais diversos dispositivos e fins (TANENBAUM; BOS, 2016). Como afirma Maciel (2014), "O mesmo Linux que roda em um supercomputador pode rodar também em uma simples placa eletrônica".

Tendo em vista esse potencial do kernel Linux, logo ele também foi adaptado para diversas arquiteturas. Sendo assim, Linux Embarcado é a aplicação e uso do kernel Linux em uma placa eletrônica cujo principal elemento é o *SoC* (*System-on-a-chip*) (MACIEL, 2014). Linux suporta uma grande variedade de arquiteturas e processadores, e as mais comumente usadas em sistemas embarcados são ARM, PowerPC e MIPS (MACIEL, 2014).

2.3 PLACAS DE DESENVOLVIMENTO E A BEAGLEBONE BLACK

Denomina-se placa de desenvolvimento a placa eletrônica que serve para testes de laboratório, tanto de software quanto de hardware, usada, geralmente, para testes de produto. Geralmente essas placas contém elementos de fácil montagem e desmontagem (MADEIRA, 2018; SAMPAIO, 2014; FILIPEFLOP, 2020).

Existem diversos tipos de placas de desenvolvimento, para todos os tipos de hardware citados anteriormente. Porém, limitando o escopo aos baseados em arquitetura computacional, destacam-se as placas de desenvolvimento microcontroladas e microprocessadas. Dentre esses tipos, destacam-se os modelos, já bem conhecidos, Arduino e ESP8266, para as microcontroladas, e o Raspberry Pi e a BeagleBone Black, que são microprocessadas.

Em se tratando da BeagleBone Black – placa alvo deste trabalho – é um pequeno computador com todos os recursos de um computador pessoal, o que significa que ele também precisa de um sistema operacional (SO). Como BeagleBone é um projeto de hardware aberto, ele é favorável a utilização de Linux – que possui código aberto. O alinhamento desses requisitos tornou a placa perfeita para o aprendizado do desenvolvimento de sistemas embarcados, isto é, hardware e software (SANTOS; PERESTRELO, 2015).

A BeagleBone Black Rev.C é um kit de desenvolvimento baseado no processador AM3358 que integra um ARM Cortex™-A8 core, rodando a 1GHz e disponibilidade de vários periféricos. Além disso a placa possui várias interfaces como Ethernet, USB, OTG, cartão TF, serial, JTAG (sem conector), HDMI tipo D< Emmc, ADC, I2C, SPI, PWM e LCD (COLEY, 2014).

A maioria das placas, sejam microcontroladas ou microprocessadas, são open hardware, ou seja, o projeto esquemático da placa (e do microcontrolador/processador) está disponível ao público, assim como os componentes da mesma estão à venda. Porém, no caso das microprocessadas, existe um exemplo de placa muito conhecida, a Raspberry Pi, que não é aberta, não permitindo, portanto, o desenvolvimento de produtos a partir dela. Por outro lado, a BBB possui não somente o hardware aberto, mas também possui um SDK (*Software Development Kit*) gratuito, com documentação disponível, além de outros treinamentos online, também gratuitos.

2.4 REVISÃO DA LITERATURA

Foi feita uma pesquisa na literatura impressa e em materiais digitais, na internet, sobre o tema proposto neste trabalho. O foco principal da busca foi: livros de referência (de vários assuntos), materiais de sites especializados em sistemas embarcados, materiais audiovisuais (no YouTube) e trabalhos acadêmicos. Dentre toda a pesquisa realizada, poucos trabalhos relacionados foram encontrados.

Em relação aos livros, foram selecionadas literaturas de referência de diversas áreas, tais como: Sistemas Operacionais (Linux), Sistemas Embarcados (geral e microprocessado), Linux embarcado, ARM e BeagleBone Black. Desses materiais, apenas os da área de Linux embarcado encontraram afinidade com o tema, sendo o livro *Mastering Embedded Linux Programming* (SIMMONDS, 2017) o mais próximo, devido ao fato de usar como base a placa BeagleBone Black; seguido pelos livros *Building Embedded Linux Systems* (YAGHMOUR et al., 2008) e *Embedded Linux Primer: A Practical Real-World Approach* (HALLINAN, 2006), mais gerais, que abordam a configuração de sistemas embarcados microprocessados baseados em Linux. No entanto, o primeiro não utilizou o SDK da Texas Instruments, diferente da proposta deste trabalho.

Em relação aos sites especializados, encontraram-se dois sites relevantes na área de Linux embarcado. O primeiro é o blog do brasileiro Sérgio Prado (PRADO, 2020), que possui diversos artigos online sobre o desenvolvimento em Linux embarcado, palestras e outros materiais. O outro é um blog especializado chamado *Embedded Related* (EMBEDDEDRELATED, 2020), que possui contribuições de diversos autores e profissionais em diversas áreas do desenvolvimento embarcado, em especial o microprocessado.

Nos materiais audiovisuais, um canal de um brasileiro, chamado *Embedded Linux* (MUñOZ, 2019) se destacou devido à qualidade dos vídeos e aprofundamento do conhecimento embarcado. Fora esse, alguns outros vídeos dispersos se destacaram, mas não foram considerados úteis para o trabalho.

Em se tratando dos materiais acadêmicos, foi feita uma pesquisa nas principais bases de dados científicas (IEEE, Scielo, Google Acadêmico) pelo termo da família do processador em questão: AM335X. A única base que retornou algo relevante foi o IEEE, nos quais os trabalhos

relacionados serão expostos a seguir. Fora isso, não fora encontrado nenhum trabalho com tema igual ou semelhante a esse.

Em suma, foi feita uma pesquisa em diversas fontes de dados diferentes, porém apenas um material, que é um livro, foi encontrado como sendo de temática bastante parecida a esse trabalho. Apesar de terem sido encontradas literaturas de temáticas semelhantes, não foi encontrado nada igual.

2.5 TRABALHOS RELACIONADOS

Após uma pesquisa sobre o tema em bases de trabalhos acadêmicos, foram encontrados cinco trabalhos correlatos. Dentre esses, dois são de aplicações práticas utilizando o processador em questão, enquanto que três são de formato de tutorial/descrição.

Os dois trabalhos de aplicação são: *Integrating The Accelerometer Of The Am335x Sitara Starter Kit In A Qt Application* (MOSNEANG; MISCHIE; PAZSITKA, 2014) e *Design of an embedded multi-antenna satellite data acquisition system based on ARM-Linux* (LU; WANG; ZHU, 2018). O primeiro é uma demonstração de uma aplicação de um kit de desenvolvimento do processador am335x, enquanto que o segundo diz respeito a um sistema embarcado de aquisição de sinais de satélite. Embora os dois projetos sejam muito diferentes em complexidade, sendo o segundo o mais elaborado, ambos são relacionados a esse trabalho, já que demonstram aplicações embarcadas, conteúdo que esse trabalho visa mostrar a teoria.

Os outros três trabalhos têm um formato de tutorial ou relatório de experiência. O artigo *On Using Sitara AM335x Starter Kit to achieve basic applications based on Linux operating system* (MISCHIE; PAZSITKA et al., 2013) e o trabalho de conclusão de curso Customização Da Plataforma Android (FILHO, 2013) demonstram sequências de passos na configuração de objetos ou tecnologias, sendo que um deles é sobre o processador abordado, que é o objetivo desse trabalho – fornecer uma fonte de passos da configuração de uma tecnologia, no caso uma placa de desenvolvimento. Por último, o artigo *Experience of Teaching Embedded Systems Design with BeagleBone Black Board* (HE; QIAN; HUANG, 2016) descreve a experiência (dos autores) em utilizar a BeagleBone Black no ensino de design de sistemas embarcados para alunos de TI.

3 DESENVOLVIMENTO DA PROPOSTA / METODOLOGIA

O desenvolvimento da pesquisa e, consequentemente, do trabalho, se sucederá principalmente a partir de testes na placa de desenvolvimento BeagleBone Black, além de desenvolvimento teórico. Por isso, é necessário explicar o processo de montagem do ambiente de trabalho e disposição dos materiais, além de esclarecer de que modo a elaboração desse trabalho ocorrerá.

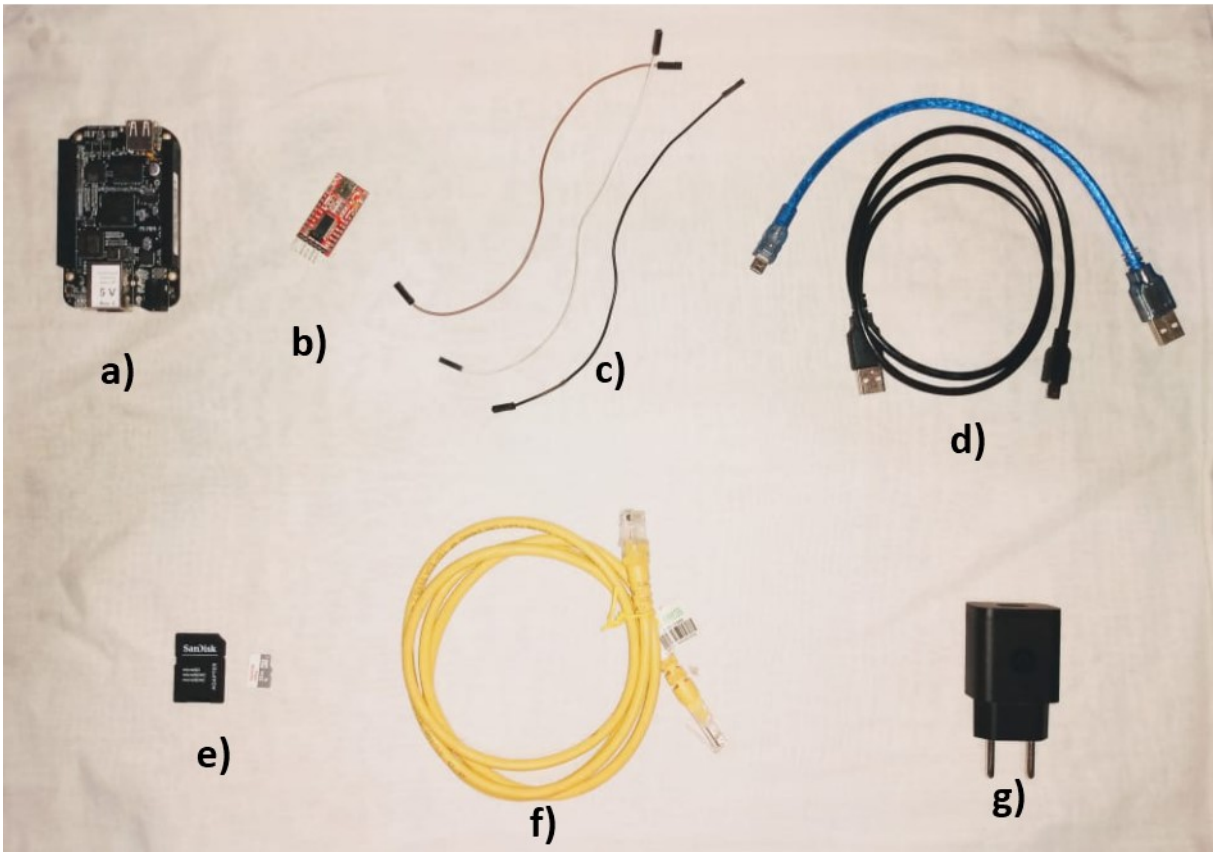
Esse capítulo listará os materiais a serem utilizados, a estrutura do ambiente de desenvolvimento, o modo como o desenvolvimento prático transcorrerá – isto é, qual a função de cada elemento, como será a troca de dados entre *host* e *target*, como será feito o registro do desenvolvimento da pesquisa – além do modo como a documentação será redigida.

3.1 MATERIAIS UTILIZADOS

Os materiais a serem utilizados no desenvolvimento prático desse trabalho serão:

a) BeagleBone Black;
b) Placa FTDI FT232RL – Conversor Serial/USB;
c) Jumpers fêmea-fêmea (3x);
d) Cabos de mini USB tipo B (2x);
e) Cartão micro SD de 2 GB (e adaptador);
f) Cabo par trançado RJ45;
g) Carregador USB turbo Motorola de 5 V;
h) Notebook (não incluso na imagem).

Figura 1 – Foto dos materiais utilizados



Fonte: AUTOR, 2020

Abaixo, lista-se uma breve explicação de cada item (maiores informações serão cedidas durante o desenvolvimento):

a) BeagleBone Black: A placa *target*, principal objeto deste trabalho. É nela onde o desenvolvimento embarcado ocorrerá;

b) Placa FTDI FT232RL – Conversor Serial/USB: Escolheu-se essa placa para a conversão serial pois era a que se tinha disponível. Em teoria, qualquer conversor USB para serial TTL funcionará, desde que esteja na voltagem correta da BeagleBone Black, que é de 5 V;

c) Jumpers fêmea-fêmea: Necessários para conectar o conversor serial/USB à BeagleBone Black;

d) Cabos de mini USB tipo B: Um dos cabos é para conectar o conversor serial/USB ao notebook, enquanto que o outro serve para energização da placa BeagleBone Black;

e) Cartão micro SD de 2 GB (e adaptador): Memória principal do *target*, será responsável por permitir o porte do sistema operacional embarcado, assim como a alteração fácil, via adaptador;

f) Cabo par trançado RJ45: Garante a conexão de rede entre *host* e *target*;

g) Carregador USB turbo Motorola de 5 V: Fonte de alimentação da placa. A BeagleBone Black tem duas possibilidades de alimentação: via USB e entrada P4, ambas 5 V. Na falta

de uma fonte dedicada, optou-se por um carregador de celular de qualidade que fornecesse o mais próximo possível de 5 V. Qualquer outro carregador USB serve, desde que respeite a corrente mínima de funcionamento da placa (500mA);

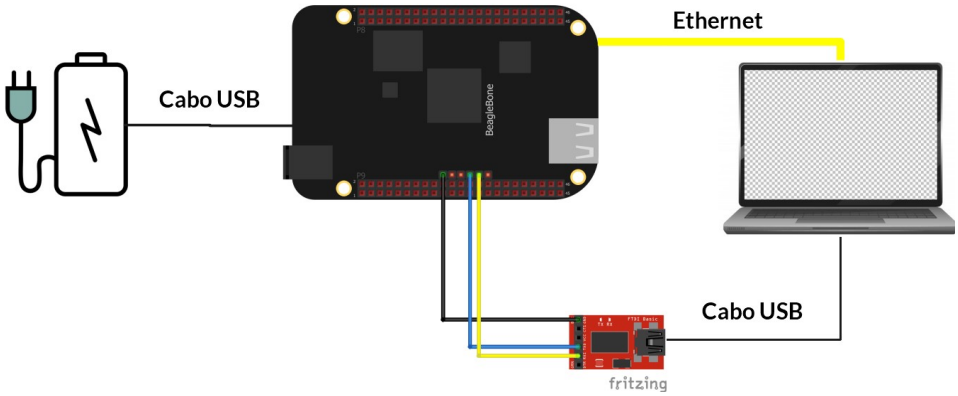
h) Notebook: Essencial ao desenvolvimento, o notebook fará o papel de *host*, que será o ambiente virtual onde a construção do software do *target* ocorrerá.

3.2 DIAGRAMA DO AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento deste trabalho será uma junção de teoria e prática, isto é, experimentação prática juntamente da pesquisa bibliográfica. Em relação à parte prática, será necessário a organização de um ambiente de desenvolvimento, em hardware e software, para a realização do experimento.

O esquema do ambiente de desenvolvimento é bem simples, formado por uma parte física (hardware) e virtual (software). No tocante ao hardware, é necessário conectar a placa ao computador via conversor serial/USB, além de energizá-la utilizando a fonte de 5 V (carregador, nesse caso). No requisito de software, porém, é necessário instalar o sistema operacional Ubuntu (ou variante) 16.04 na máquina *host*.

Figura 2 – Diagrama do ambiente de desenvolvimento



Fonte: AUTOR, 2020

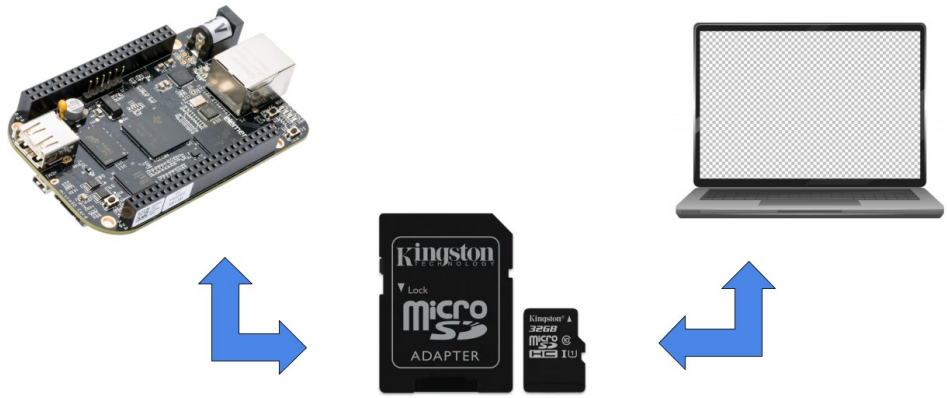
O processo de experimentação prática acontecerá da seguinte forma: O *host*, rodando o sistema operacional Ubuntu, se conectará à placa BeagleBone Black, *target*, via o conversor serial para USB. A placa, após ser energizada, iniciará o boot do sistema, cujas mensagens de console aparecerão no terminal do *host*, via conversor. Além disso, o cabo par trançado será utilizado para utilizar interfaces e protocolos de rede entre *host* e *target*, como o SSH.

3.3 MÉTODO DE TROCA DE ARQUIVOS ENTRE *HOST* E *TARGET*

Durante o desenvolvimento, será necessário fazer trocas de arquivos e até de sistemas de arquivos inteiros do *host* para o *target* (e vice-versa). Para isso, far-se-á uso do cartão de

memória da placa para a transferência.

Figura 3 – Imagem ilustrativa do método de troca de arquivos



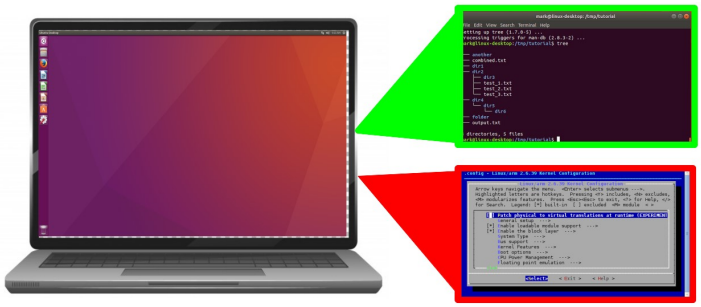
Fonte: AUTOR, 2020

Utilizando-se um adaptador microSD, os arquivos serão passados para o cartão, que em seguida será inserido na placa desligada. O inverso segue o mesmo princípio: desliga-se a placa, retira-se o cartão, conecta-se ao *host* e efetuam-se as operações.

3.4 REGISTROS VIA CAPTURAS DE TELA

Como o esquema do ambiente de desenvolvimento praticamente não mudará (é só um), não é necessário tirar além de uma foto dele. Porém, o desenvolvimento que ocorrerá virtualmente (em software), o qual corresponde a maior parte da pesquisa, deve ser registrado. O processo de passo a passo (evolução) da pesquisa será feito via capturas de tela.

Figura 4 – Ilustração do registro do desenvolvimento via capturas de tela



Fonte: AUTOR, 2020

As capturas de tela serão feitas principalmente dos(as):

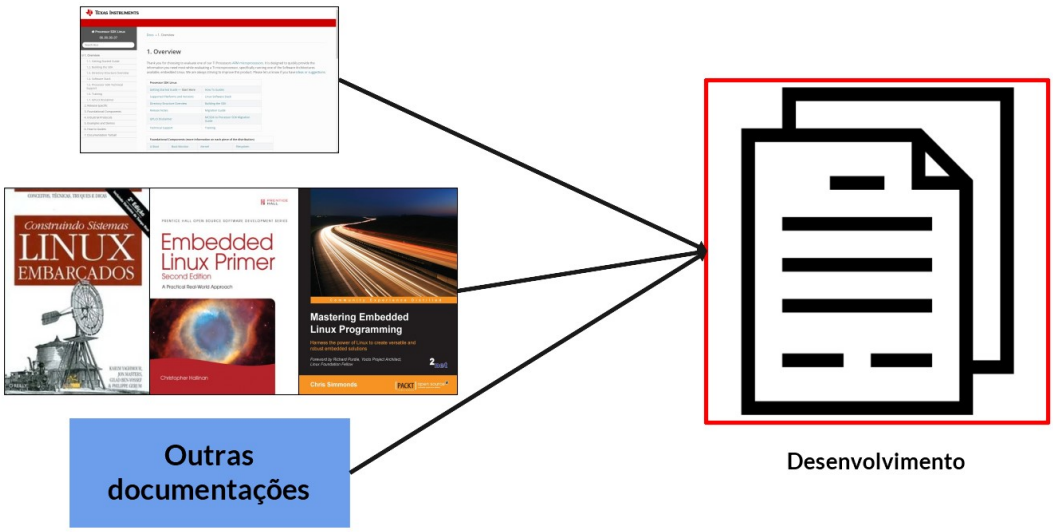
- Terminais;
- Telas de programas (em interface gráfica);
- Sistemas de arquivos;
- E o que mais for necessário registrar.

3.5 PROCEDIMENTO DE ELABORAÇÃO DA DOCUMENTAÇÃO

O processo de documentação deste trabalho será constituído da soma da descrição do desenvolvimento prático, na placa BeagleBone Black (passos, comentários, prints, etc.) com o detalhamento, teórico, dos conceitos envolvidos (tais como: Kernel, conceitos de Linux – *shell* e sistema de arquivos, comunicação serial, arquitetura de computadores – ARM, inicialização, e muitos outros).

A busca por material teórico transcorrerá a partir de uma ordem de prioridade de fontes, dando mais valor para as literaturas com maior peso acadêmico (ou dados mais oficiais possíveis). Assim sendo, a documentação do SDK da Texas Instruments será a principal fonte de informações para o desenvolvimento, seguida pelos livros de referência na área de Linux embarcado. Dependendo da necessidade, outras fontes também podem ser usadas.

Figura 5 – Imagem ilustrativa do modo da elaboração da documentação



Fonte: AUTOR, 2020

Dessa forma, as fontes de pesquisa bibliográfica para a elaboração da documentação seguirão a seguinte prioridade:

- Fonte 1 – Documentação do SDK Texas Instruments
- Fonte 2 – Livros de Linux embarcado: Building Embedded Linux Systems, Embedded Linux Primer: A Practical Real-World Approach e Mastering Embedded Linux Programming.
- Fonte 3 – Documentações oficiais de Linux e ARM: Tentar-se-á o máximo possível pelas documentações oficiais; quando não for possível, passar-se-á para os livros especializados, depois para as páginas web especializadas, depois para os fóruns, sempre dando-se prioridade às informações mais oficiais possíveis.

Parte I

Introdução ao Linux embarcado

4 CONTEXTUALIZAÇÃO

O Linux, desde a sua criação, ocupou um papel muito importante na tecnologia da informação atual, adentrando até mesmo no mundo dos sistemas embarcados. O objetivo deste capítulo é contextualizar a área de Linux embarcado, expondo conceitos gerais, classificações e o seu impacto no mundo.

Sendo assim, esse capítulo está separado em: contextualização geral do desenvolvimento embarcado – tópico 4.1 – classificação dos tipos de dispositivos Linux embarcados – tópico 4.2 – descrição das principais entidades envolvidas na área – tópico 4.3 – principais formas de desenvolvimento – tópico 4.4 – e as vantagens no uso do Linux na construção de um sistema embarcado – tópico 4.5.

4.1 OS SISTEMAS EMBARCADOS BASEADOS EM LINUX

No contexto do desenvolvimento de Linux embarcado, geralmente são encontradas duas formas de uso. São elas: os sistemas Linux embarcados – dispositivos (em geral, já prontos) que usam o kernel Linux e vários outros softwares (como a placa *Raspberry Pi*) – e distribuições Linux embarcadas – pacotes de softwares e códigos-fonte feitos para o desenvolvimento de um *target* específico e completo, como a distribuição Yocto (YAGHMOUR et al., 2008). Essas distribuições podem ser gratuitas ou pagas, conforme comenta Yaghmour et al. (2008).

As distribuições Linux trazem, além do que foi citado, ferramentas de desenvolvimento, como compiladores cruzados, depuradores, geradores de imagens de *boot*, entre outros. A opção por usar uma distribuição (indiretamente, pelo uso do SDK) foi escolhida para o desenvolvimento deste trabalho.

Em se tratando do uso de distribuições Linux, apesar do esforço envolvido na adaptação de um sistema para um hardware específico, o código fonte do kernel Linux não precisa ser modificado para encaixar na maioria das arquiteturas de processadores (dentre elas o ARM). Ao invés disso, ele fornece dezenas de opções para configuração de funcionalidades; por exemplo, o suporte a multiprocessadores, funcionalidade geralmente presente em servidores, mas rara em sistemas embarcados. Esses fatores favorecem o uso de uma distribuição (ou um pacote de software) para o desenvolvimento de um sistema embarcado microprocessado (YAGHMOUR et al., 2008).

4.2 TIPOS DE SISTEMAS LINUX EMBARCADO

Existem muitos exemplos de sistemas embarcados que utilizam Linux atualmente, e diversas formas de organizá-los. Apesar disso, é possível enquadrá-los de acordo com a estrutura

do sistema, isto é, em relação ao tamanho, as restrições de tempo, a capacidade de operar em rede e as interfaces de usuário, por exemplo (YAGHMOUR et al., 2008). Nesse sentido, os sistemas embarcados baseados em Linux podem ser classificados por:

- **Tamanho:**

  O tamanho do sistema é influenciado por diversos fatores, dentre eles o próprio tamanho físico da placa – e o número de periféricos e chips que essa possui (YAGHMOUR et al., 2008). Nesse sentido, os sistemas Linux embarcados podem ser classificados – de acordo com Yaghmour et al. (2008) – em relação ao tamanho, em: pequenos, médios ou grandes.

  Em síntese, conforme Yaghmour et al. (2008), os sistemas pequenos são aqueles que possuem uma CPU de baixo consumo e entre 8 e 16 MB de RAM. Já os médios são aqueles que possuem uma CPU de capacidade média e entre 64 e 28 MB de RAM, além de uma forma de armazenamento secundária baseada em NAND, geralmente um cartão de memória. Por outro lado, os sistemas grandes são aqueles que possuem uma CPU poderosa para aplicações embarcadas e muita memória RAM para ser usada (a partir de 512 MB).

  Em relação a placa de desenvolvimento deste trabalho, a BeagleBone Black – e, consequentemente, um sistema baseado nela – poderia ser classificado como de grande capacidade, pois possui um processador ARM de até 1GHz, 512 MB de RAM, além de uma série de outras funcionalidades aplicadas no *SoC am335x* (COLEY, 2014).

- **Restrição de tempo:**

  Existem dois tipos de restrições de tempo para sistemas embarcados, conforme Yaghmour et al. (2008): rigorosas – também chamadas de *real-time* (tempo real) – e leves. As restrições de tempo real, dividem-se ainda em: *hard* e *soft*.

  Restrições de tempo rigorosas requerem do sistema uma resposta em um determinado intervalo de tempo, ou algum evento danoso pode ocorrer. Por outro lado, restrições leves, apesar de variarem bastante em requisitos, geralmente aplicam-se a sistemas nos quais o tempo de resposta não necessariamente é crítico; que não afete o resultado. Um exemplo disso pode ser um sistema embarcado aplicado à automação residencial, que demore alguns segundos para iniciar uma aplicação.

  O *SoC am335x* e, por consequência, a placa BeagleBone Black, não necessariamente se restringe a um ou outro cenário. A Texas Instruments fornece SDKs tanto para uma situação como para outra, ambas baseadas em Linux – kernel normal para a primeira e Linux RT para a segunda (TI, 2019). Porém, o desenvolvimento mais favorecido para a placa, assim como o modelo que foi escolhido para esse trabalho, é a restrição de tempo leve, suportada pelo kernel Linux normal.

- **Operações em rede:**

Nos dias atuais, é uma tendência que cada dispositivo seja conectado a rede, seja por via cabeada ou sem fio – fenômeno conhecido como Internet das Coisas (BUTUN, 2020; YAGHMOUR et al., 2008). Isso, por consequência, trás requisitos específicos aos dispositivos que hão de ser desenvolvidos.

Um dos fatores que mais impulsiona fabricantes a utilizarem Linux em seus produtos embarcados, de acordo com Yaghmour et al. (2008), é o fato do kernel possuir suporte a diversos protocolos de comunicação em rede, sejam esses populares – como o Ethernet – ou não – como o CAN, industrial.

Um dispositivo baseado na BeagleBone Black pode possuir não apenas acesso à rede, como também suporte a diversos protocolos, muitos deles industriais, como o PRU-ICSS (COLEY, 2014). Com a vantagem de diversos *drivers* de dispositivo já prontos no SDK, esse produto estaria preparado aos requisitos do mercado atual.

- **Interface com o usuário:**

  O grau de interação com o usuário pode variar bastante, dependendo dos requisitos do sistema, principalmente em relação ao consumo de energia (YAGHMOUR et al., 2008; PRAKASH; SHIN, 2013). Pode ser que haja um *display*, pode ser que hajam botões e LEDs, pode ser que haja uma interface web, uma interface SSH, ou até mesmo nenhuma interface (YAGHMOUR et al., 2008; TI, 2019).

  O *SoC am335x* possui suporte ao uso até mesmo de uma interface HDMI, mas também possui interface para um *display* próprio (COLEY, 2014). Além disso, as muitas pinagens GPIO disponíveis, aliadas ao suporte do kernel Linux, podem oferecer diversas outras possibilidades.

4.3 ENTIDADES NO CENÁRIO DE LINUX EMBARCADO

Diferentemente de sistemas operacionais proprietários, o Linux não é controlado por uma só autoridade central. Pelo contrário, diversas entidades com diferentes vocações determinam o futuro e a filosofia do software, assim como de que modo tecnologias serão adotadas (YAGHMOUR et al., 2008; SIMMONDS, 2017). As mais importantes são destacadas: a comunidade de desenvolvimento aberto e as indústrias de semicondutores.

A comunidade de desenvolvimento de código aberto é a base de todo o desenvolvimento de Linux – seja no ambiente desktop, no embarcado, como em todos os outros – e a entidade mais importante na área de Linux embarcado (YAGHMOUR et al., 2008; SIMMONDS, 2017). É constituída de desenvolvedores que mantém, expandem e suportam os diversos componentes que constituem um sistema Linux (YAGHMOUR et al., 2008), e muitos desses desenvolvedores são representados por instituições sem fim lucrativos, tais como universidades e algumas instituições comerciais (SIMMONDS, 2017). O trabalho de desenvolvimento envolve tanto o projeto principal quanto projetos satélites, tais como o U-Boot, Busybox e outros sob a responsabilidade

do projeto GNU (YAGHMOUR et al., 2008; SIMMONDS, 2017), exemplos esses que serão intensivamente estudados neste trabalho.

As indústrias de semicondutores também representam papel ímpar no mundo Linux, especialmente no embarcado. Tendo reconhecido o potencial do Linux no mundo embarcado, diversas empresas passaram a adotar e a promover o uso de Linux nos seus produtos. Esses, embarcados com Linux, chegam ao consumidor final por meio delas, e elas recebem o retorno dos mesmos, servindo de ponte entre a teoria – os desenvolvedores – e a prática – os consumidores (YAGHMOUR et al., 2008).

Além disso, essas empresas possuem uma função muito importante no desenvolvimento do software Linux, pois elas adaptam o kernel e a *toolchain* para os seus processadores, acarretando a adição de mais suporte ao kernel original; como a Texas Instruments, por exemplo ((YAGHMOUR et al., 2008; SIMMONDS, 2017; TI, 2019). Por fim, essas indústrias criam placas de desenvolvimento, que são usadas para a produção de dispositivos reais por outras empresas ou pessoas, como a BeagleBone Black, por exemplo.

4.4 DISTRIBUIÇÕES LINUX EMBARCADAS E O USO DE UM SDK

Para desenvolver um sistema embarcado baseado em Linux, existem, basicamente, dois caminhos a se tomar: escolher uma distribuição pronta, seja paga ou gratuita, ou escolher montar o sistema operacional, isto é, contendo toda a estrutura de kernel, sistema de arquivos etc. do zero, a partir dos pacotes de software.

Em relação as distribuições, que são sistemas operacionais completos prontos para a instalação (HALLINAN, 2006), existem dois tipos: as pagas, que são as comerciais; e as gratuitas (YAGHMOUR et al., 2008). As distribuições pagas são aquelas cedidas por alguma empresa de software, e podem ter o código total ou parcialmente fechado. Já as gratuitas geralmente de código aberto são aquelas construídas pela comunidade de software, sem uma autoridade central (YAGHMOUR et al., 2008). O uso de distribuições pode ser vantajoso quando o *time-to-market* for muito pequeno, ou seja, quando o produto precisa ser lançado logo e não há necessidade de personalizar todo o sistema embarcado (YAGHMOUR et al., 2008).

Por outro lado, existe a possibilidade de o desenvolvedor escolher otimizar ao máximo o seu sistema para a aplicação desejada; de querer manter os próprios pacotes ou códigos-fonte desenvolvidos para uma possível patente. Nesse caso, é mais vantajoso montar o sistema usando os pacotes (*bootloader*, bibliotecas, kernel, entre outros) separadamente; é o método conhecido como *do it yourself* (faça você mesmo) (YAGHMOUR et al., 2008). Para tanto, duas alternativas são possíveis ao desenvolvedor de sistemas embarcados: baixar, um a um, os pacotes da internet, separadamente; ou utilizar um pacote de desenvolvimento contendo todas as ferramentas, códigos e tudo o que for necessário para o desenvolvimento, de forma otimizada – um SDK (Software Development Kit – Kit de desenvolvimento de software).

4.5 VANTAGENS DO LINUX EMBARCADO

Devido aos inúmeros benefícios econômicos e técnicos, vê-se um forte crescimento na adoção do Linux para dispositivos embarcados (HALLINAN, 2006). Essa tendência cruzou praticamente todos os mercados e tecnologias. O Linux tem obtido sucesso em aparelhos automotivos, aparelhos *mobile*, impressoras, *switches* e roteadores corporativos e muitos outros produtos. Essa taxa de adoção do Linux embarcado continua crescendo, sem fim à vista (HALLINAN, 2006).

Existem muitos motivos para a escolha de Linux em detrimento de outro sistema operacional para sistemas embarcados. Muitos desses motivos são compartilhados pelo mundo desktop e de servidores, enquanto outros são mais exclusivos para o uso do Linux em dispositivos embarcados (YAGHMOUR et al., 2008). Tais vantagens são: qualidade e confiabilidade do código, suporte de hardware, protocolos de comunicação e padrões de software, além do custo.

Em primeiro lugar, qualidade e confiabilidade são medidas subjetivas do nível de confiança no código que compreende softwares como o kernel e os aplicativos fornecidos por distribuições (YAGHMOUR et al., 2008). Algumas características de um código de qualidade e confiável seguem abaixo, segundo Yaghmour et al. (2008).

Compreende-se um código de qualidade aquele que possui: Modularidade e estrutura – isto é, cada funcionalidade separada deve ser encontrada em um módulo separado, e o layout do arquivo do projeto deve refletir isso; legibilidade – o código deve ser legível e (mais ou menos) fácil de consertar para quem entende seu interior; extensibilidade – a adição de recursos ao código deve ser simples e configurabilidade – a possibilidade de selecionar quais recursos do código devem fazer parte do aplicativo final.

Quanto a confiabilidade, o código deve possuir: Previsibilidade – isto é, o comportamento do programa deve estar dentro de uma estrutura definida; capacidade de recuperação de erros e longevidade – o programa funcionará sem intervenção por longos períodos de tempo e preservará sua integridade, mesmo se algum recurso faltar.

Em segundo lugar, o Linux foi portado para uma ampla gama de arquiteturas de processador, incluindo algumas que são comumente encontradas em designs de *SoCs* - ARM, MIPS, x86 e PowerPC (YAGHMOUR et al., 2008; HALLINAN, 2006; SIMMONDS, 2017). Embora vários fornecedores ainda não forneçam *drivers* Linux, um progresso considerável foi feito e mais é esperado. Como um grande número de *drivers* é mantido pela própria comunidade Linux, o projetista pode usar componentes de hardware com confiança, sem medo de que o fornecedor possa um dia interromper o suporte a *drivers* para essa linha de produtos (YAGHMOUR et al., 2008). Dada uma CPU e uma plataforma de hardware baseada ou construída nela, o projetista pode razoavelmente esperar que o Linux rode nela ou que outra pessoa tenha passado por um processo de portabilidade semelhante e possa ajudá-lo em seus esforços (YAGHMOUR et al., 2008).

Em terceiro lugar, o kernel Linux provê amplo suporte a protocolos de comunicação e

padrões de software (YAGHMOUR et al., 2008; HALLINAN, 2006; SIMMONDS, 2017). Ele tem um bom agendador, uma boa pilha de rede, suporte para USB, Wi-Fi, Bluetooth, muitos tipos de mídia de armazenamento, bom suporte para dispositivos multimídia e assim por diante (SIMMONDS, 2017). Isso torna mais fácil integrar o Linux às estruturas existentes e portar software legado para o Linux. Como tal, pode-se integrar facilmente um sistema Linux dentro de uma rede Windows existente e esperar que ele atenda clientes por meio do Samba (usando o *Active Directory*), por exemplo (YAGHMOUR et al., 2008).

Por último, O custo do Linux é resultado do licenciamento de código aberto e é diferente do que pode ser encontrado com sistemas operacionais integrados tradicionais, ou seja, é gratuito (YAGHMOUR et al., 2008; SIMMONDS, 2017).

Existem três componentes de custo de software na construção de um sistema embarcado tradicional (não Linux) segundo Yaghmour et al. (2008): configuração inicial de desenvolvimento, ferramentas adicionais e royalties de tempo de execução. Com o Linux, esse modelo de custo está invertido. A maioria das ferramentas de desenvolvimento e componentes do sistema operacional estão disponíveis gratuitamente, e as licenças sob as quais eles são normalmente distribuídos evitam a cobrança de quaisquer royalties sobre esses componentes principais (YAGHMOUR et al., 2008).

5 PRINCIPAIS CONCEITOS

A área de Linux embarcado possui muitas peculiaridades. Embora usem como núcleo do seu firmware o kernel Linux, sistemas embarcados baseados em Linux possuem muitas diferenças na sua construção comparados a sistemas convencionais, como os desktops, por exemplo. O objetivo desse capítulo é introduzir os principais conceitos do desenvolvimento de sistemas embarcados Linux, vitais para qualquer projetista.

Assim sendo, este capítulo está dividido em: detalhamento das fases de um projeto de sistema embarcado Linux – tópico 5.1 – tipos de *hosts* – tópico 5.2 – modelo de configuração vinculada – tópico 5.3 – e a arquitetura de um sistema Linux embarcado, em software e hardware – tópico 5.4.

5.1 CICLO DE VIDA DE UM PROJETO

Um projeto de desenvolvimento de um sistema embarcado baseado em Linux usualmente possui algumas etapas determinadas, apesar da singularidade de cada dispositivo. Tais etapas são, conforme Simmonds (2017): Configuração do ambiente de desenvolvimento, escolha da arquitetura e design, desenvolvimento de aplicações embarcadas e depuração e otimização de desempenho. Vale ressaltar que essas etapas não são necessariamente sequenciais, e o desenvolvedor precisa frequentemente voltar alguns passos para revisar partes do projeto.

A etapa de configuração do ambiente de desenvolvimento é aquela em que o projetista preparará o seu ambiente, tanto físico quanto virtual, para o início do projeto (SIMMONDS, 2017). Envolve instalação de ambientes de desenvolvimento integrados e ferramentas de projeto.

A escolha da arquitetura e design do projeto diz respeito as decisões que o desenvolvedor terá que tomar em relação ao hardware e software do dispositivo, por exemplo, ao armazenamento de programas e dados (hardware), como dividir o trabalho entre os *drivers* de dispositivo do kernel e aplicativos e como inicializar o sistema (software) (SIMMONDS, 2017).

A fase de desenvolvimento de aplicações embarcadas corresponde à programação eficiente, isto é, como fazer uso eficaz do modelo de processos e *threads* do Linux e como gerenciar a memória em um dispositivo com recursos limitados (SIMMONDS, 2017).

Por fim, a etapa de depuração e otimização de desempenho é relativa ao rastreio e depuração do código nos aplicativos e no kernel (SIMMONDS, 2017). Serve para identificar erros ou imprevistos e pode ser executada em laboratório ou em campo (YAGHMOUR et al., 2008).

5.2 TIPOS DE *HOSTS*

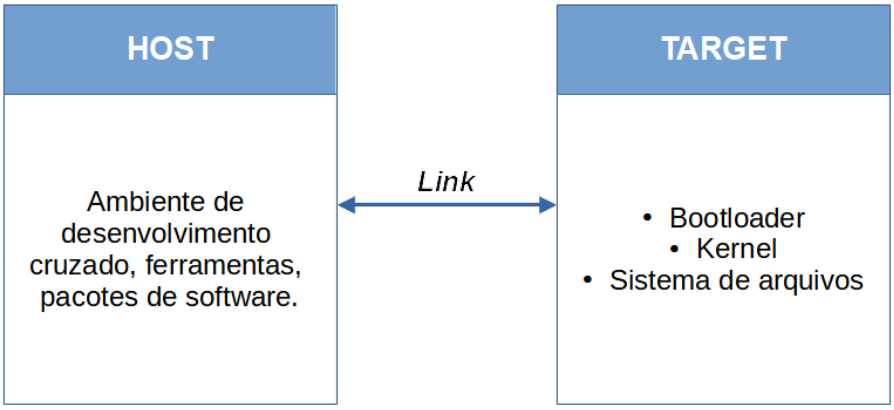
Diferentes tipos de projetos podem demandar diferentes tipos de estações de trabalho, também chamadas de *hosts* no mundo Linux embarcado (YAGHMOUR et al., 2008). Existem diversos tipos de *hosts* e, dentre eles, destacam-se, conforme Yaghmour et al. (2008) os *hosts* baseados em Linux, em Unix e Windows. Pelo fato de, neste trabalho, ser usado um *host* Linux, os *hosts* baseados em Unix e Windows não serão explorados.

A estação de trabalho baseada em Linux é, segundo Yaghmour et al. (2008), o tipo mais comum de desenvolvimento encontrado. A razão para isso é que o uso de um sistema Linux para a programação e configuração de um outro sistema acaba contribuindo para o ganho de experiência no uso do próprio Linux, o que favorece a resolução de certas situações e acelera o desenvolvimento. Um computador pessoal comum pode ser o hospedeiro para um *host* (como é o caso desse trabalho) e, dependendo das restrições de software, várias distribuições Linux podem ser usadas (YAGHMOUR et al., 2008; TI, 2019).

5.3 MODELO DE CONFIGURAÇÃO VINCULADA *HOST/TARGET*

Existem diferentes tipos de modos de ligação entre *host* e *target* em um projeto. De acordo com Yaghmour et al. (2008), são três: a configuração vinculada, a configuração de armazenamento removível e a configuração *standalone*. Pelo fato de, neste trabalho, ser utilizada a configuração vinculada, as duas últimas não serão aprofundadas. Abaixo, na figura 6, é mostrado um diagrama dessa configuração, com a explicação em seguida.

Figura 6 – Modelo de configuração vinculada *host/target*



Adaptado de (HALLINAN, 2006)

Na configuração vinculada (também chamada de *linked setup*) o *target* e o *host* são conectados via um cabo físico, que pode ser uma interface serial, Ethernet, ou ambos (YAGHMOUR et al., 2008). O objetivo desse modelo é garantir que toda a transferência de dados e arquivos seja feita via link, e não via armazenamento removível (YAGHMOUR et al., 2008).

Essa configuração é a mais comum, e também serve para fins de depuração. Geralmente é usada a interface serial para depurar, e a interface Ethernet para transferência de arquivos (YAGHMOUR et al., 2008).

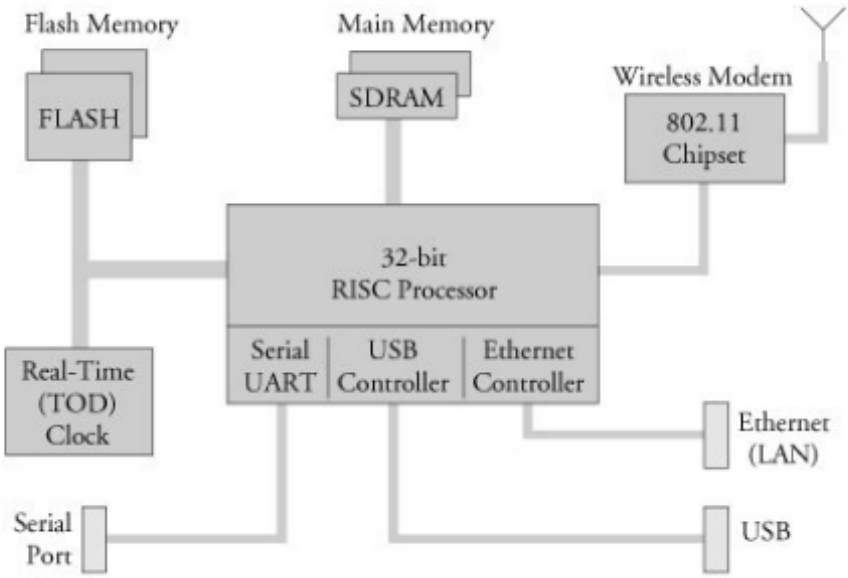
Este trabalho segue o modelo de configuração vinculada (representado na seção 3.2), apesar de usar o cartão de memória para a transferência de arquivos por um bom período do desenvolvimento. O trabalho fora enquadrado nesse modelo pois ambas as interfaces mencionadas serão utilizadas, mas, enquanto o SSH (que usa Ethernet) não estiver configurado, apenas a interface serial poderá ser usada, juntamente do cartão micro SD.

5.4 ARQUITETURA GENÉRICA DE UM SISTEMA LINUX EMBARCADO

Os sistemas embarcados Linux possuem muitos componentes, tanto físicos (hardware) como virtuais (software). Apesar da especificidade de cada projeto, pode-se generalizar, em um modelo de diagrama de blocos, uma arquitetura genérica dos sistemas embarcados Linux. Tais modelos serão descritos abaixo.

Em relação a arquitetura de hardware, o sistema embarcado obedece a estrutura da figura 7. É um exemplo simples, cedido por Hallinan (2006), de um *Acess Point* (sem fio), mas o modelo pode ser levado em consideração para a maioria dos sistemas embarcados atuais, pois praticamente todos possuem algum tipo de conexão sem fio.

Figura 7 – Arquitetura de hardware de um sistema embarcado Linux



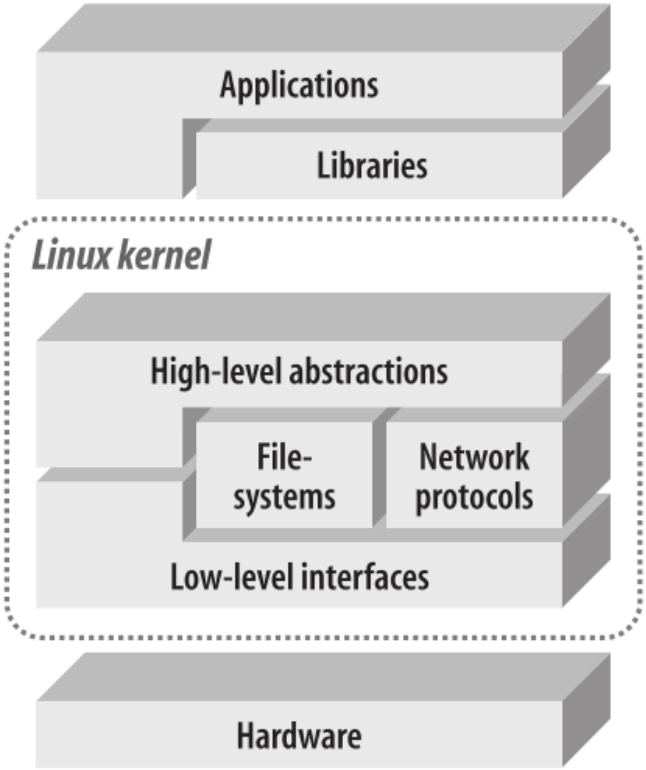
Fonte: (HALLINAN, 2006)

Conforme Yaghmour et al. (2008), Simmonds (2017), os requisitos de hardware para um *target* Linux são: um processador de (pelo menos) 32 bits, uma quantidade suficiente de memória RAM (que depende da aplicação), capacidades mínimas de entrada e saída, além de uma mídia

de armazenamento permanente. Nota-se que a arquitetura mostrada na figura 2 contempla todos esses requisitos. A placa de desenvolvimento BeagleBone Black também os suporta.

Quanto a arquitetura de software, o kernel Linux possui diversos componentes. A figura 3 apresenta a arquitetura de um sistema Linux genérico, cedida por Yaghmour et al. (2008). Observa-se que há pouca diferença na descrição a seguir entre um sistema embarcado e uma estação de trabalho ou sistema de servidor, uma vez que os sistemas Linux são todos estruturados da mesma forma neste nível de abstração (YAGHMOUR et al., 2008).

Figura 8 – Arquitetura de software de um sistema embarcado Linux



Fonte: (YAGHMOUR et al., 2008)

Imediatamente acima do hardware está o kernel, o componente central do sistema operacional. Seu objetivo é gerenciar o hardware de uma maneira coerente enquanto fornece abstrações familiares de alto nível para software de nível de usuário (YAGHMOUR et al., 2008).

Dentro do kernel, duas categorias amplas de serviços em camadas fornecem a funcionalidade exigida pelos aplicativos. As interfaces de baixo nível são específicas para a configuração do hardware. Acima dos serviços de baixo nível fornecidos pelo kernel, os componentes de alto nível fornecem as abstrações comuns a todos os sistemas (baseados em) Unix, incluindo processos, arquivos, soquetes e sinais (YAGHMOUR et al., 2008).

Entre esses dois níveis de abstração, o kernel às vezes precisa do que poderiam ser chamados de componentes de interpretação para entender e interagir com dados estruturados

vindos de ou indo para determinados dispositivos. Os tipos de sistema de arquivos e protocolos de rede são exemplos importantes de fontes de dados estruturados que o kernel precisa entender e interagir para fornecer acesso aos dados que vão e vêm dessas fontes (YAGHMOUR et al., 2008).

Por fim, as aplicações não podem acessar o kernel diretamente. Ao invés disso, precisam de bibliotecas e *daemons* de sistema especiais para fornecer APIs familiares e serviços abstratos que interagem com o kernel em nome do aplicativo para obter a funcionalidade desejada (YAGHMOUR et al., 2008). A principal biblioteca usada pela maioria dos aplicativos Linux é a biblioteca GNU C, *glibc*.

6 DESENVOLVIMENTO DESTE TRABALHO

O processo de construção de um sistema embarcado baseado em Linux é extenso e complexo; envole muitas etapas. Além disso, é necessário escolher um kit de desenvolvimento e uma placa de desenvolvimento aliados ao objetivo do projeto em questão. O objetivo desse capítulo é descrever o desenvolvimento deste trabalho, isto é, o modo pelo qual a configuração será feita.

Dessarte, esse capítulo está dividido em: detalhes do hardware utilizado – tópico 6.1 – detalhes do software utilizado – tópico 6.2 – descrição dos capítulos – tópico 6.3 e convenções ao longo do texto – tópico 6.4.

6.1 HARDWARE USADO – A BEAGLEBONE BLACK

A placa de desenvolvimento – *target* – a ser usada nesse trabalho será a popular e *open hardware* BeagleBone Black. As justificativas para isso são várias: extenso uso em projetos reais, várias funcionalidades e recursos, além de vasta documentação e suporte.

Primeiro, esta é uma placa de desenvolvimento de baixo custo e amplamente disponível que pode ser usada em projetos embarcados sérios (SIMMONDS, 2017; COLEY, 2014). Além disso, a sua popularidade lhe confere um certo grau de longevidade e significa que continuará a estar disponível durante alguns anos – mesmo já sendo um pouco antiga (desde 2013) (SIMMONDS, 2017).

Segundo, essa placa possui suporte a diversos protocolos e interfaces disponíveis no *SoC am335x*, além de ser um hardware poderoso e bem construído (COLEY, 2014). Como exemplo, pode-se citar o processador de 1GHz, a memória RAM de 512 MB, a interface serial, a interface Ethernet 10/100, RJ45, interface para duas MMC (na placa, uma é a eMMC padrão e a outra é o cartão SD), 69 portas GPIO, entre muitas outras. Embora muitos dos recursos e interfaces fornecidos pelo processador não são acessíveis a partir do BeagleBone Black – segundo Coley (2014), ainda assim, a placa constitui uma excelente forma de desenvolvimento.

Terceiro, como usa um *SoC* da Texas Instruments, essa placa é coberta pelo SDK desse *SoC*, além de possuir uma série de treinamentos. Existe a documentação oficial do SDK, além de vídeos e tutoriais mais específicos no portal de treinamento da Texas Instruments (TI, 2019). Existe, ainda, um fórum onde questões e problemas mais específicos ainda podem ser comentados pela comunidade (ou pelos próprios desenvolvedores da Texas Instruments).

6.2 SOFTWARE USADO – O SDK TEXAS INSTRUMENTS

O kit de desenvolvimento a ser utilizado no desenvolvimento deste trabalho será o SDK (*Software Development Kit*) do *SoC am335x*, fornecido pela Texas Instruments. Esse *SoC* e, com isso, esse SDK, são usados pela BeagleBone Black.

Um dos grandes desafios para iniciar o desenvolvimento em uma nova plataforma (ou um novo sistema operacional, para muitos), é obter um ambiente configurado onde o desenvolvedor possa construir e depurar código no hardware. O SDK ataca esse problema fornecendo tudo o que o desenvolvedor precisa para fazer o desenvolvimento. Ademais, é validado em plataformas de hardware padrão da Texas Instruments (conhecidas como EVMs) (TI, 2019).

O objetivo do SDK é envolver tudo isso em um instalador simples que ajuda a colocar tudo o que o projetista precisa no lugar certo para fazer o desenvolvimento (TI, 2019). Conforme a TI (2019), alguns dos recursos do SDK que permitem a criação de um sistema embarcado "do zero" são:

- Códigos-fonte de U-Boot e arquivos de configuração;
- Códigos-fonte de kernel e arquivos de configuração;
- Um conjunto de ferramentas de compilação cruzada ARM, bem como outros binários e componentes de host;
- Um sistema de arquivos compatível com Yocto e códigos-fonte de algumas aplicações, como exemplos;
- Uma variedade de scripts e Makefiles para automatizar certas tarefas;
- Outros componentes necessários para construir um sistema Linux embarcado que não se encaixam perfeitamente em um dos pontos acima.

Apesar de haver a possibilidade de se obter cada pacote de software necessário para o sistema separadamente (como BusyBox, U-Boot, o kernel Linux, etc.) conforme foi citado anteriormente, o uso de um kit de desenvolvimento é muito mais vantajoso nesse caso, pois, além do suporte da empresa, já foi testado na placa alvo e possui diversos pacotes pré-compilados, o que evita que o projetista perca tempo em retrabalho, favorecendo, assim o *time-to-market*.

6.3 DESCRIÇÃO DOS CAPÍTULOS

A divisão e a ordem dos capítulos foi tomada com base das referências bibliográficas deste trabalho (tópico 3.5), no ciclo de vida de um projeto descrito no tópico 5.1, além de experiências de desenvolvimento do autor.

Procurou-se separá-los em uma ordem lógica, agrupados, aproximadamente, por fases, que geralmente estão associadas ao desenvolvimento linear de um projeto real. Para isso, essas fases foram divididas em partes (hierarquia acima dos capítulos), para melhor organização do documento.

A divisão deste trabalho em parte é, respectivamente: Introdução ao Linux embarcado, Preparação do ambiente, Configuração dos softwares essenciais, Construção do sistema de arquivos, Configuração das interfaces de utilização, Programação em Linux embarcado utilizando C++ e Aplicação real. Vale ressaltar que a primeira e a última parte não estão associadas as fases do desenvolvimento; foram dispostas desse modo apenas para melhorar a organização do trabalho.

A Parte 2 – Preparação do ambiente (capítulos 7 e 8) – corresponde a configuração, tanto do *host* quanto do *target*, para o desenvolvimento. Em uma situação real, serve para familiarizar o projetista ao hardware a ser utilizado e às ferramentas de trabalho. O capítulo 7, Organização do ambiente de desenvolvimento, mostra a instalação do sistema operacional e do SDK no *host*. O capítulo 8, Preparação da placa para o primeiro *boot*, apresenta a instalação de um sistema de arquivos pronto na placa para demonstração.

A Parte 3 – Configuração dos softwares essenciais (capítulos 9, 10 e 11) – aborda a compilação e instalação dos programas que dão vida à placa, que são o *bootloader*, o kernel Linux e Busybox. O capítulo 9, Configuração do inicializador: U-boot, trata sobre o inicializador escolhido: *Das U-Boot*. O capítulo 10, Escolha dos comandos básicos: BusyBox, aborda a configuração de um recurso que gera comandos para o sistema. Por fim, o capítulo 11, Configuração do Kernel Linux, trata da configuração e especificidades do kernel Linux.

A Parte 4 – Construção do sistema de arquivos (capítulos 12 e 13) – disserta sobre o sistema de arquivos Linux para sistemas embarcados, teórica e praticamente. O capítulo 12, Conceitos importantes, descreve os principais conceitos acerca dos sistemas de arquivos aplicados a sistemas embarcados. O capítulo 13, Construção do sistema de arquivos do zero, é a descrição do desenvolvimento de um sistema de arquivos mínimo e genérico para o *target*.

A Parte 5 – Configuração das interfaces de utilização (capítulos 14 e 15) – aborda a compilação e utilização de dois softwares – *OpenSSH* e o *lighttpd* – para garantir duas funcionalidades muito presentes em sistemas embarcados atuais – acesso remoto e interface web. O capítulo 14, Habilitação do SSH como *shell* remoto, descreve a compilação e utilização do cliente *OpenSSH*. O capítulo 15, Construção de uma interface web utilizando *lighttpd*, aborda os mesmos passos, mas em relação a esse servidor web visando a construção de uma página web de configuração.

A Parte 6 – Programação em Linux embarcado utilizando C++ (capítulos 16 e 17) – tratam da questão do desenvolvimento de software embarcado aplicado à BeagleBone Black, isto é, sua produção e depuração. O capítulo 16, Aplicação: Software de aquisição de dados para um servidor MySQL, aborda o desenvolvimento de uma aplicação de aquisição de dados em C++. O capítulo 17, Depuração: Uso do GDB Server para análise remota de software, exemplifica o uso do programa de depuração do GNU aplicado à sistemas embarcados.

Por último, a Parte 7 – Aplicação real (capítulos 18) - trata da aplicação do conteúdo

do trabalho em um projeto real, a fim de demonstrar a importância desse conhecimento e exemplificar conceitos apresentados ao longo do trabalho. O capítulo 18, Cap. 18 - Plataforma de aquisição de dados para o sistema de monitoramento de máquinas da Eletronorte: SIMMEDAQ, descreve o objetivo e o ciclo de vida de um sistema embarcado baseado em Linux, voltado à usinas hidroelétricas.

6.4 CONVENÇÕES

Durante o desenvolvimento, uma estrutura será usada com frequência. São os comandos do *shell* Linux, que serão executados no terminal, tanto do *host* quanto do *target*. Para separá-los do restante do texto, foi adotada uma estrutura de caixa de texto retangular. O texto nessa caixa será formatado no estilo "itálico", e cada início (entrada) de *shell*, terá um "$".

Para distinguir um terminal do *host* para o *target*, as caixas de texto terão seus contornos diferenciados. O terminal do *host* terá o seu contorno contínuo e o fundo branco, enquanto que o *target* terá linhas em cima e embaixo, além do fundo cinza. Um exemplo de ambos segue abaixo, respectivamente.

```
1 $ uname -a
2 Linux SDKdesenvolvimento 4.15.0-132-generic #136-Ubuntu SMP Tue Jan
    ↪ 12 14:58:42 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

```
1 $ uname -a
2 Linux am335x-evm 4.19.38-g4dae378bbe #1 PREEMPT Sun Jul 7 04:39:33
    ↪ UTC 2019 armv7l GNU/Linux
```

Parte II

Preparação do ambiente

7 ORGANIZAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

A produção de sistemas embarcados sempre requer a instalação de ferramentas. Porém, para Linux embarcado, é necessário que haja um ambiente ainda mais robusto para suportar o desenvolvimento, visto que o processo é complexo. O objetivo deste capítulo é demonstrar a organização do ambiente de desenvolvimento virtual, isto é, no *host*.

Assim, esse capítulo abordará: a instalação do sistema operacional no *host* – tópico 7.1, a instalação do SDK – tópico 7.2, a configuração do SDK – tópico 7.3, comandos adicionais na configuração – tópico 7.4 e a instalação do PuTTY – tópico 7.5.

7.1 INSTALAÇÃO DO SISTEMA OPERACIONAL - UBUNTU 18.04 LTS

Para começar o desenvolvimento do sistema embarcado Linux com o SDK, é necessário um computador com alguma distribuição Linux instalada. Existem diversas opções, porém, a Texas Instruments validou o seu kit apenas na distribuição Ubuntu, nas versões de suporte de longo prazo (LTS – *Long Term Support*) disponíveis na época do lançamento do SDK (14.04, 16.04 ou 18.04), de 64 bits (TI, 2019). Existe a possibilidade de usar outras distros, porém, neste trabalho optou-se por utilizar o sistema operacional padrão mais atualizado – Ubuntu 18.04 LTS.

Devido ao fato do processo de instalação de uma distribuição Linux ser, de certo modo, trivial e bastante repetitivo, ele não será detalhado aqui. Apenas alguns detalhes ou considerações na instalação serão comentados.

Para começar, foi baixado do site oficial do Ubuntu a ISO com a versão 18.04 LTS do sistema operacional (*ubuntu-18.04.1-desktop-amd64.iso*). Embora não seja a mais atual no momento da escrita deste trabalho, é a ISO na qual o SDK fora testado anteriormente.

Em relação aos requisitos mínimos, a documentação do SDK não especifica nenhum para o *host*. Porém, experiências no uso de diferentes máquinas indicam que uma pouco potente atrasa o desenvolvimento, principalmente nas partes de compilação intensiva, como na compilação do kernel, no capítulo 11.

Além disso, o site oficial do Ubuntu possui informações sobre os requisitos mínimos que o sistema precisa para rodar. Considerando que o *host* não será uma máquina de uso geral e sim de trabalho – na maioria das vezes – os requisitos mínimos do Ubuntu 18.04 podem ser levados em consideração como sendo os requisitos mínimos do SDK. Nesse sentido, os requisitos são (em relação a processador e memória RAM): Processador *dual core* 2 GHz (ou melhor) e 2 GB de RAM.

Durante a instalação, um detalhe a ser levado em consideração é que não é necessário instalar a versão completa do sistema operacional. O motivo disso é que a máquina *host*, em um

ambiente de desenvolvimento, geralmente serve apenas para o trabalho com o sistema embarcado, além de algumas pesquisas e referências pontuais na internet; não serve para uso pessoal. Por isso, a instalação mínima foi marcada, conforme a figura 9.

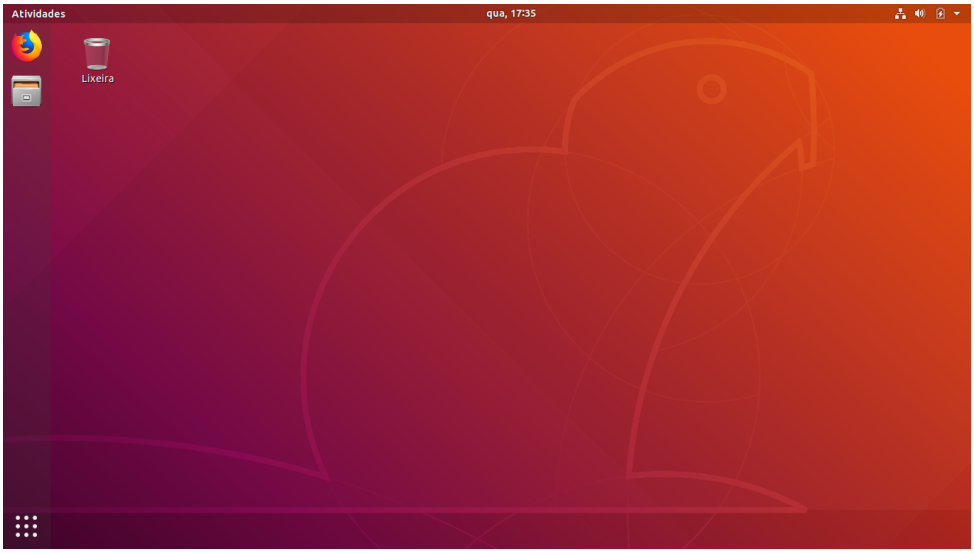
Figura 9 – Instalação mínima do Ubuntu e a instalação dos *drivers* de terceiros

Fonte: AUTOR, 2021

Além disso, na mesma figura 9 também se vê que foi marcada a opção de instalação de *drivers* de terceiros. Conforme alguns testes realizados, não marcar essa opção causa instabilidade no sistema. Essa função serve principalmente para garantir que o sistema operacional se adaptará ao hardware, portanto, deve ser marcada.

Após isso, instala-se o sistema normalmente. A área de trabalho do Ubuntu 18.04 instalado é a que está na figura 10. Vale ressaltar que, embora outras distribuições Linux diferentes do Ubuntu não foram validadas pela TI, outras baseadas nele (como Lubuntu, Xubuntu, Ubuntu MATE, etc.) são lícitas, pois, embora possuam uma interface gráfica diferente do Ubuntu (que é GNOME), o núcleo ainda é o mesmo. Essa alternativa é recomendável para computadores fracos.

Figura 10 – Área de trabalho do Ubuntu 18.04



Fonte: AUTOR, 2021

7.2 INSTALAÇÃO DO SDK TEXAS INSTRUMENTS

Feita a instalação do sistema operacional do *host*, o próximo passo é instalar o SDK da Texas Instruments. O instalador do SDK é um arquivo *.bin*, no formato *ti-processor-sdk-linux-[NomeDaPlataforma]-evm-xx.xx.xx.xx-Linux-x86-Install.bin*, nos quais NomeDaPlataforma significa o nome da família de processadores usados – *am335x*, nesse caso – e a sequência de "x" indica a versão (neste trabalho será usada a versão *06_00_00_07*).

Esse objeto instalará os componentes necessários para iniciar o desenvolvimento no microprocessador. O SDK contém um sistema de arquivos de desenvolvimento, sistemas de arquivos para o *target*, aplicativos de exemplo, conjunto de ferramentas e pacote de suporte de placa, *scripts* de fácil uso, documentação e os códigos-fonte de diversos aplicativos de exemplo. O SDK inclui também o conjunto de ferramentas para compilação cruzada ARM GCC da Linaro (TI, 2019).

Existem duas formas de se obter o SDK. Uma é pelo cartão SD que acompanha as EVMs (*Evaluation module* – Módulo de avaliação), outra é o download do instalador pelo site. Como a BeagleBone Black não é uma EVM, o SDK não vem no seu cartão SD, o que requer o download na página oficial.

Antes de iniciar a instalação, é necessário confirmar se o executável possui permissão de execução. Existem dois modos de fazer isso: um é pelo terminal e o outro, pela interface gráfica do explorador de arquivos. Pelo *shell*, o resultado se obtém via o comando *chmod*:

```
$ cd Downloads
$ chmod +x ti-processor-sdk-linux-am335x-evm-06.00.00.07-Linux-x86-
    ↪ Install.bin
```

Além desse passo, é necessário confirmar se o sistema operacional instalado é de 64 bits. Foi baixada uma versão correspondente para o desenvolvimento, mas, para confirmar, insere-se o comando:

```
$ uname -m
x86_64
```

Obedecendo-se esses pré-requisitos, pode-se dar início a instalação.

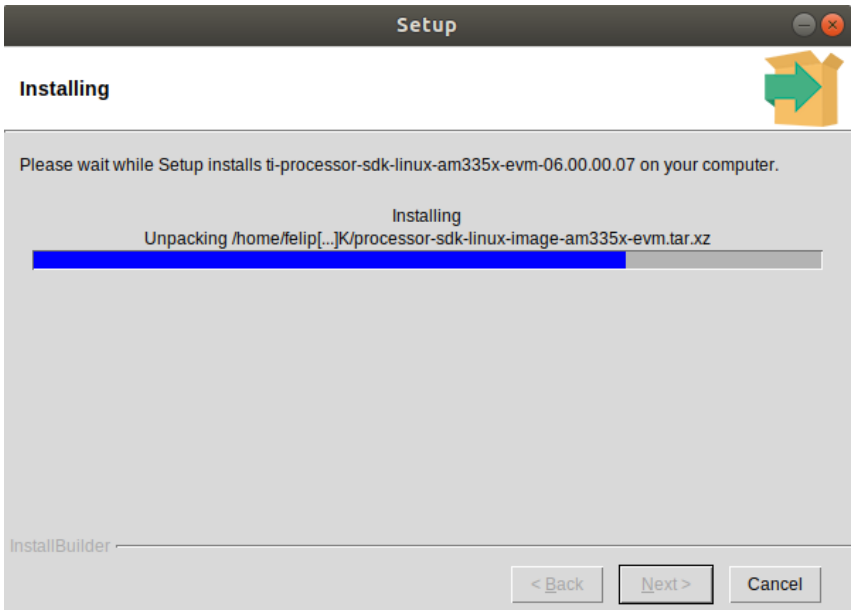
```
$ ./ti-processor-sdk-linux-am335x-evm-06.00.00.07-Linux-x86-Install.
    ↪ bin
```

Conforme a documentação do SDK (TI, 2019), os passos executados pelo instalador são:

- **Confirmação:** Verifica se o usuário está tentando reinstalação;
- **Local de instalação do diretório:** O usuário será solicitado a fornecer um local onde colocar o SDK;
- **Instalação:** O software será descompactado e instalado.

Seguindo com a instalação, chega-se a etapa de escolha do local de instalação do SDK. A alternativa automática é a pasta de nome *ti-processor-sdk-linux-am335x-evm-06.00.00.07*, na pasta */home/felipe*. Porém, resolveu-se – por experiências anteriores – alterar o nome da pasta para *texasSDK*. O motivo disso é que esse nome mais curto facilita encontrar o diretório mais facilmente, além do fato de não causar nenhum prejuízo futuro. Após isso, inicia-se o processo de instalação, que é basicamente a descompactação do conteúdo do SDK na pasta escolhida, conforme a figura 11.

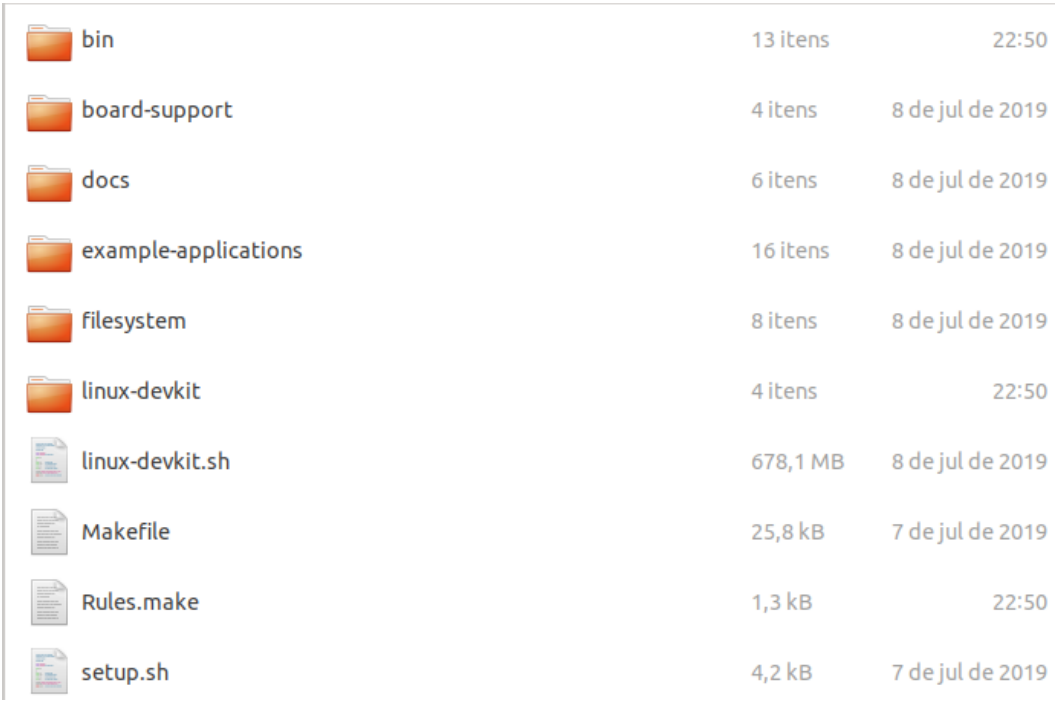
Figura 11 – Instalação do SDK



Fonte: AUTOR, 2021

E, com isso, tem-se o diretório do SDK instalado na máquina *host*, conforme a figura 12, que representa a estrutura do SDK. Esses diretórios contêm o código e as ferramentas usadas para desenvolver dispositivos (TI, 2019). Vale ressaltar que nem todos esses componentes serão abordados neste trabalho.

Figura 12 – Estrutura do diretório do SDK



Fonte: AUTOR, 2021

De acordo com a documentação oficial (TI, 2019), a utilidade básica de cada diretório (ou arquivo) é:

- ***bin*** – Contém os *scripts* auxiliares para configurar o sistema *host* e o *target*. A maioria desses *scripts* é usada pelo *script setup.sh*;
- ***board-support*** – Contém os códigos-fonte que precisam ser modificados ao portar para uma plataforma personalizada. Isso inclui o kernel e os bootloaders, bem como qualquer *driver* fora da árvore;
- ***docs*** – Contém várias documentações, como o manifesto do software e guia do usuário adicional;
- ***example-applications*** – Contém os códigos-fonte dos aplicativos de exemplo fornecidos pela TI, vistos durante a demonstração pronta para uso;
- ***filesystem*** – Contém os sistemas de arquivos de referência. Isso inclui o sistema de arquivos básico (menor), bem como o sistema de arquivos com todos os recursos;
- ***linux-devkit*** – Contém o conjunto de ferramentas de compilação cruzada e bibliotecas para acelerar o desenvolvimento para o dispositivo de destino;
- ***Makefile*** – Serve para facilitar a compilação de alguns recursos do SDK;
- ***Rules.make*** – Define os valores padrão usados pelo *Makefile* de nível superior, bem como pelos *Makefiles* de subcomponentes;
- ***setup.sh*** – Configura o sistema de *host* do usuário, bem como o *target* para desenvolvimento.

7.3 CONFIGURAÇÃO DO SDK - *SCRIPT SETUP.SH*

Após a instalação do SDK, é necessário a execução do *script* de configuração para preparar o *host* para o desenvolvimento; funciona como uma segunda parte da instalação. O *script* de configuração faz o seguinte procedimento (TI, 2019):

- Verifica se o host Linux é a versão Ubuntu LTS recomendada;
- Instalação de pacotes de host necessários;
- Instala de sistema de arquivos do target;
- Configuração do NFS;
- Configuração do TFTP;
- Configuração do minicom;
- Configuração do U-Boot;
- Carrega o script U-Boot.

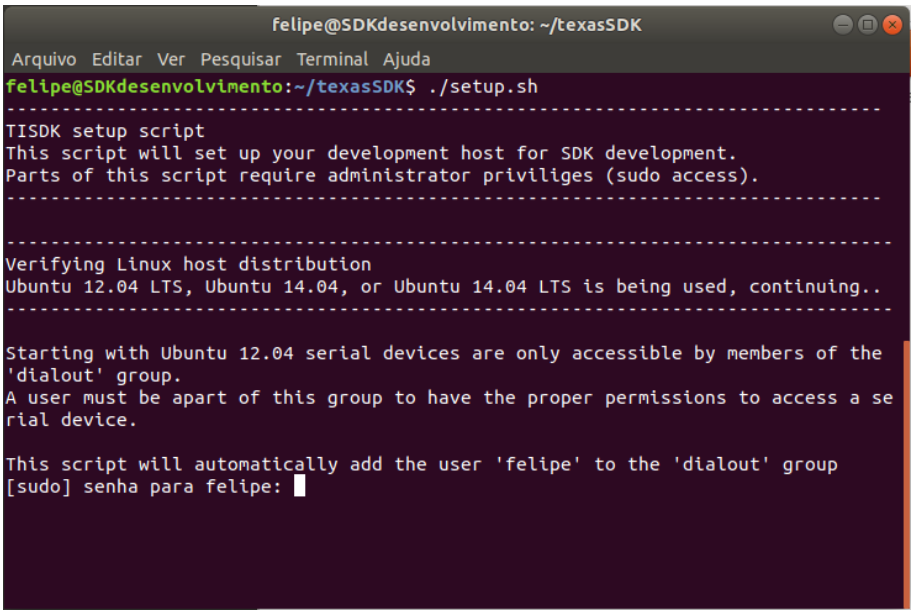
No entanto, apenas os dois primeiros passos descritos acima serão realmente importantes, pois os outros tratam apenas de configurações e recursos que não serão utilizados, como os protocolos NFS e TFTP. Portanto, após a instalação dos pacotes necessários o *script* foi interrompido com o comando *Ctrl+C*.

O *script* está na raiz do SDK, e para executá-lo basta abrir o diretório no terminal e digitar:

```
$ cd ~/texasSDK
$ ./setup.sh
```

A primeira atitude a ser tomada pelo *script* é verificar qual distribuição Linux o usuário está executando (conforme a figura 13). Se não for nenhuma das suportadas, exatamente nas versões dadas, eles interromperá o processo. Embora não haja problemas futuros em usar distros diferentes, o *script* impede o prosseguimento.

Figura 13 – Início da execução do *setup.sh*



Fonte: AUTOR, 2021

É possível instalar esse SDK em outra distro, porém, todos os passos do *script* terão de ser feitos manualmente, ou então pode-se alterar o *script* para ignorar essa parte. Como serão utilizadas apenas as duas primeiras configurações do *script*, seria simples fazer essa etapa manualmente em outra distribuição Linux (Mint, por exemplo), porém, neste trabalho resolveu-se seguir a instalação padrão.

Após isso, o primeiro passo da configuração do SDK é a adição do usuário corrente ao grupo de discagem (*Add to Dialout Group*). Por padrão, o usuário não tem as permissões adequadas para acessar um dispositivo serial (ex ttyS0, ttyUSB0, etc.). Um usuário deve fazer

parte de um grupo *dialout* (discagem) para acessar esses dispositivos seriais (neste caso, a BeagleBone Black) sem privilégios de *root* (TI, 2019).

Durante esta etapa, o *script* verificará se o usuário Linux atual faz parte do grupo *dialout*. Caso contrário, o usuário Linux atual será automaticamente adicionado ao grupo de discagem, conforme se vê na console abaixo.

```
1 ----------------------------------------------------------
2
3 Starting with Ubuntu 12.04 serial devices are only accessible by
    ↪ members of the 'dialout' group.
4 A user must be apart of this group to have the proper permissions to
    ↪ access a serial device.
5
6 This script will automatically add the user 'felipe' to the 'dialout'
    ↪  group
```

A próxima configuração, e última antes do *script* ser interrompido, é instalar pacotes obrigatórios para o *host* (*Installation of Required Host Packages*). Esta etapa verificará se o *host* possui os pacotes de suporte adequados para permitir a execução das seguintes tarefas (TI, 2019):

- Executar o telnet;
- Abrir o menuconfig, a ferramenta de configuração do kernel (e outras bilbiotecas);
- Montagem do sistema de arquivos via NFS;
- Executar o protocolo TFTP;
- Executar o minicom;
- Recompilar o u-boot.

```
1 ----------------------------------------------------------
2 setup package script
3 This script will make sure you have the proper host support packages
    ↪ installed
4 This script requires administrator priviliges (sudo access) if
    ↪ packages are to be installed.
5 ----------------------------------------------------------
```

Se o *host* não tiver algum dos pacotes necessários, eles serão instalados automaticamente. Embora nem todos esses recursos listados sejam usados, é importante manter a máquina preparada e atualizada. Essa etapa pode ser feita manualmente a partir do comando abaixo, conforme a documentação do SDK (TI, 2019).

```
$ sudo apt-get install xinetd tftpd nfs-kernel-server minicom build-
    ↪ essential libncurses5-dev autoconf automake dos2unix screen
    ↪ lrzsz lzop u-boot-tools
```

7.4 OUTROS COMANDOS

Existem outros comandos – além do *script* de configuração – que precisam ser executados no *host* para configurar o ambiente. Esses comandos estão associados, comumente, à instalação de pacotes para compilação ou pacotes de bibliotecas compartilhadas para alguns programas utilitários.

Primeiramente, é necessário instalar alguns pacotes para a configuração e compilação de pacotes de software, que serão utilizados nas etapas de compilação nos próximos capítulos (U-Boot, BusyBox, kernel, etc.). O principal deles é o bison, que é, conforme GNU (2014), um gerador de analisador de propósito geral que converte uma descrição gramatical para um programa C. Para instalá-lo, executam-se os seguintes comandos:

```
$ sudo apt-get update
$ sudo apt-get install linux-headers-`uname -r` bison build-essential
    ↪ dkms flex gcc cpp g++ libncurses5-dev make
```

Além disso, para compilar o kernel Linux, é necessário instalar uma dependência:

```
$ sudo apt install libssl-dev
```

Secundariamente, é necessário exportar o caminho do compilador cruzado para o PATH do sistema. Devido ao fato de existirem diversos pacotes de software e programas que hão de serem compilados neste trabalho, é necessário que o compilador cruzado (ARM – Linaro) esteja disponível para acesso rápido no console tal como o compilador convencional, GCC. Para isso, é necessário modificar um arquivo do sistema para que a adição seja permanente.

Basicamente, é preciso adicionar o comando de *export* da pasta dos compiladores ao PATH em um arquivo de configuração, *.profile*, na pasta *home*. Após isso, basta reiniciar a sessão e os compiladores cruzados já estarão disponíveis, conforme apresentado no console abaixo. A vantagem desse método é que a alteração é permanente, ou seja, não se vai quando o sistema é reiniciado ou quando o console atual é fechado.

```
# Abrindo o arquivo de configuração
$ gedit ~/.profile

# Deve-se adicionar a seguinte linha no final do arquivo e salvar
export PATH=$PATH:/home/felipe/texasSDK/linux-devkit/sysroots/x86_64-
    ↪ arago-linux/usr/bin

# Após reiniciar a sessão, os compiladores já estão disponíveis
$ arm-linux-gnueabihf-gcc --version
arm-linux-gnueabihf-gcc (GNU Toolchain for the A-profile Architecture
    ↪  8.3-2019.03 (arm-rel-8.36)) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
Este é um software livre; veja as fontes para as condições de cópia.
    ↪ NÃO HÁ garantias; nem mesmo de COMERCIALIZAÇÃO ou ADEQUAÇÃO A
    ↪ UMA FINALIDADE ESPECÍFICA.
```

7.5 INSTALAÇÃO DO EMULADOR DE TERMINAL PUTTY

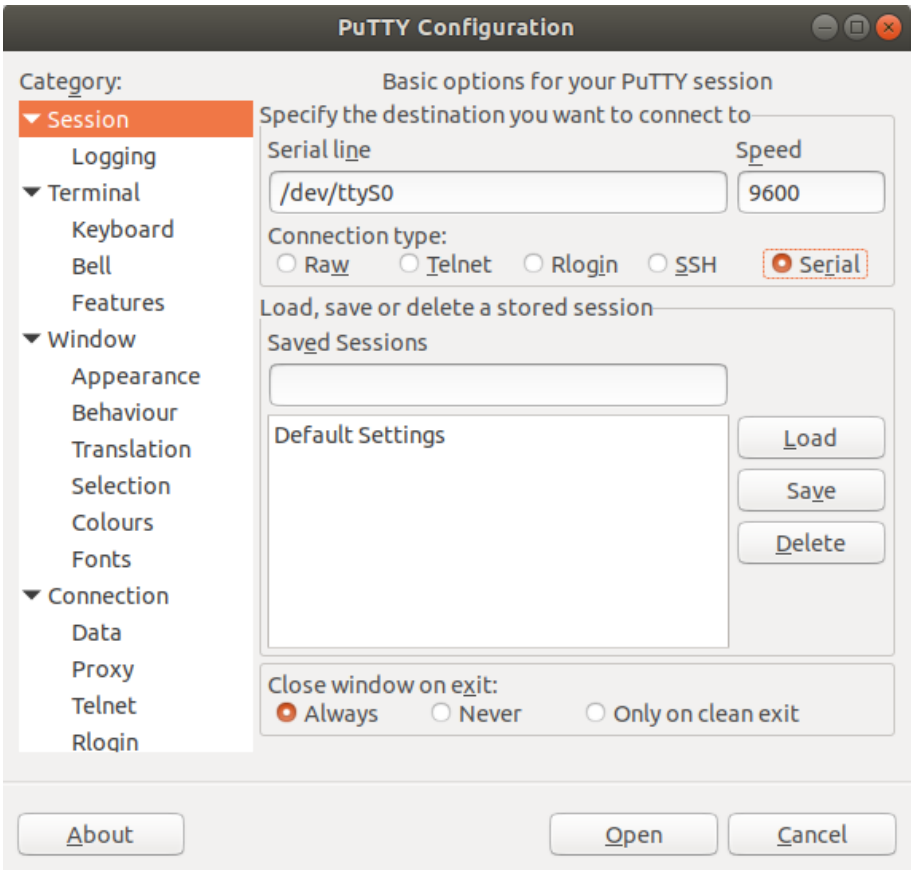
O PuTTY é um software de emulação de terminal grátis e de código livre. É o cliente SSH gratuito mais popular do mundo. Suporta SSH, *telnet*, serial e outras conexões com boa emulação de terminal. Também inclui implementações de SFTP e SCP na linha de comando (SSH, 2021c).

Embora haja o software padrão do Linux para acesso à serial – *minicom* – padrão inclusive no SDK, neste trabalho optou-se por um programa mais robusto para essa função – o PuTTY. Essa decisão foi tomada principalmente por experiências anteriores, já que o PuTTY geralmente mostrava melhor performance para exibir a interface serial, além de diversos outros recursos (como salvar o *log* em um arquivo), além de ser baseado em GUI.

A instalação desse software no *host* pode ser feita pelos comandos abaixo. Na figura 14 é mostrada a interface básica do programa, porém, a demonstração só será feita no capítulo 8, quando a serial do target for utilizada, na primeira inicialização.

```
$ sudo apt update
$ sudo apt install -y putty
```

Figura 14 – Interface do emulador de terminal PuTTY



Fonte: AUTOR, 2021

8 PREPARAÇÃO DA PLACA PARA O PRIMEIRO *BOOT*

Por ser microprocessada, a placa de desenvolvimento precisa ser impulsionada por um sistema operacional, por menor que seja, para que o desenvolvimento possa iniciar. Esse etapa também faz parte da organização do ambiente de trabalho; nesse caso, em relação ao *target*. O objetivo desse capítulo é demonstrar a preparação da BeagleBone Black para o desenvolvimento embarcado.

Sendo assim, este capítulo está dividido da seguinte forma: formatação e particionamento do cartão micro SD – tópico 8.1, organização da bancada – tópico 8.2 e demonstração da primeira inicialização – tópico 8.3.

8.1 PREPARAÇÃO DO CARTÃO MICRO SD

Para utilizar o *target*, é necessário, antes, preparar o seu dispositivo de memória, neste caso, o cartão micro SD. O SDK contém um *script* no diretório */texasSDK/bin* chamado *create-sdcard.sh*. O que esse *script* faz é, basicamente, particionar o SD em *boot* e *rootfs*, além de populá-lo com arquivos essenciais, que são o *bootloader* e o sistema de arquivos, um em cada partição, respectivamente (TI, 2019).

Essa etapa é necessária pois o cartão micro SD precisa ser particionado de uma forma específica, e fazer isso manualmente seria muito difícil, por isso a TI fornece um programa para facilitar a preparação de cartões para as suas placas.

Primeiramente, é necessário conectar o cartão micro SD no *host*. A documentação do SDK informa que o mínimo de armazenamento são 8 GB, porém, testes realizados indicam que 4 GB é o suficiente – mesmo para a maior das imagens de sistemas de arquivos disponíveis, por isso, um cartão de 4 GB será utilizado.

Antes de executar o *script* é recomendável formatar o cartão micro SD. Uma boa (e completa) ferramenta para se fazer isso é o *gparted*. O procedimento não será demonstrado aqui, mas o cartão pode ser formatado em FAT32 normalmente, pois depois será reparticionado no formato correto pelo *script*.

Tendo feito isso, o *script create-sdcard.sh* já pode ser executado. É necessário acessar a pasta em que ele está e executá-lo, pelo terminal, com permissão de *root*. O processo é demonstrado no console abaixo.

```
1 $ cd home/felipe/texasSDK/bin
2 $ sudo ./create-sdcard.sh
```

A primeira etapa do *script* é selecionar o cartão micro SD. Geralmente, haverá apenas um conectado à máquina no momento, então é esperado que apareça apenas um dispositivo para

a seleção. Além disso, o sistema de arquivos do *host* (geralmente o *sda*) não é mostrado para evitar danos permanentes à máquina por formatação acidental (SDKTEXAS).

```
Available Drives to write images to:

#  major    minor     size    name
1:   8        16     3929088 sdb

Enter Device Number or n to exit: 1

sdb was selected
```

Em seguida, o *script* perguntará ao usuário se deseja particionar o SD com 2 ou 3 partições. Na maioria das vezes (e neste caso também) serão selecionadas 2 partições. No caso de 3, deve ser usado apenas por fabricantes de placas que fazem cartões micro SD para ir na caixa com as EVMs (TI, 2019). A primeira partição receberá o inicializador do sistema, U-Boot, que será abordado com mais detalhes no capítulo 10; a segunda partição será receptáculo do sistema de arquivos do sistema, nesse caso, um dos sistemas de arquivos do SDK.

```
###############################################################

  Select 2 partitions if only need boot and rootfs (most users).
  Select 3 partitions if need SDK & other content on SD card.  This
   ↪ is
        usually used by device manufacturers with access to
   ↪ partition tarballs.

  ****WARNING**** continuing will erase all data on sdb

###############################################################

Number of partitions needed [2/3] : 2


Now partitioning sdb with 2 partitions...
```

Depois que o cartão micro SD for particionado, o usuário será questionado se deseja continuar instalando o sistema de arquivos ou sair com segurança do *script* para continuar mais tarde. Foi selecionado “y” (sim). A partir desse ponto, o *script* iniciará a etapa de instalação do conteúdo no cartão SD.

O *script* chegará a etapa mostrada no console abaixo. O usuário será questionado se deseja instalar imagens já compiladas (prontas para transferência) ou escolher outras. Nesse caso,

será escolhida a opção 1, para que o *script* selecione o *bootloader* automaticamente.

```
##################################################################

  Choose file path to install from

  1 ) Install pre-built images from SDK
  2 ) Enter in custom boot and rootfs file paths

##################################################################

Choose now [1/2] : 1
```

Além disso, é necessário escolher, também, qual sistema de arquivos será copiado para o SD. O *script* fornece duas opções nessa etapa: a completa (*rootfs*) e a *docker*. Existem outras duas que não são apresentadas nessa opção, que são a *base* e a *tiny*, porém, podem ser escolhidas caso na pergunta anterior seja escolhida a 2ª opção. No caso dessa primeira inicialização, escolheu-se a versão mais completa do sistema de arquivos. Detalhes acerca dessas imagens prontas serão dados no capítulo 13, na construção do sistema de arquivos.

```
##################################################################

   Multiple rootfs Tarballs found

##################################################################

   1:tisdk-rootfs-image-am335x-evm.tar.xz
   2:tisdk-docker-rootfs-image-am335x-evm.tar.xz

Enter Number of rootfs Tarball: 1
```

Terminada essa operação de transferência, o *script* finaliza a sua execução e o cartão micro SD está pronto para ser inserido na BeagleBone Black. Nota-se que, após o término, ambas as partições construídas (*boot* e *rootfs*) são remontadas no explorador de arquivos.

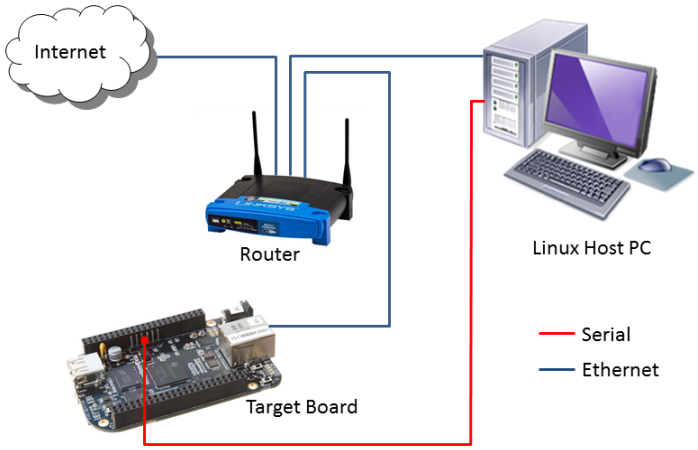
```
Copying boot partition
MLO copied
u-boot.img copied
Copying rootfs System partition
Syncing...
Un-mount the partitions
Remove created temp directories
Operation Finished
```

8.2 ORGANIZAÇÃO DA BANCADA

Uma vez que o cartão micro SD esteja preparado, pode-se passar para a etapa de arrumação dos componentes físicos na bancada, isto é, a BeagleBone Black e os seus subitens. A bancada de desenvolvimento, que é a parte física da organização do ambiente, serve para garantir ao desenvolvedor clareza e organização na realização do seu trabalho, além de permitir a conexão entre *host* e *target*, conforme fora comentado no capítulo 3 – desenvolvimento da proposta.

Abaixo, na figura 15, observa-se um modelo do ambiente físico fornecido pela Texas Instruments. Nota-se que o ambiente determinado para esse trabalho no desenvolvimento da proposta – figura 2 – foi diretamente baseado nele: O computador – *host* – (configurado no capítulo anterior) conecta-se à placa – *target* – pelo cabo serial e um roteador garante a conexão cabeada entre o *host*, o *target* e a internet. No caso deste trabalho, não será usado um roteador pois não há a necessidade de conectar a placa à internet; a conexão entre os dois é direta.

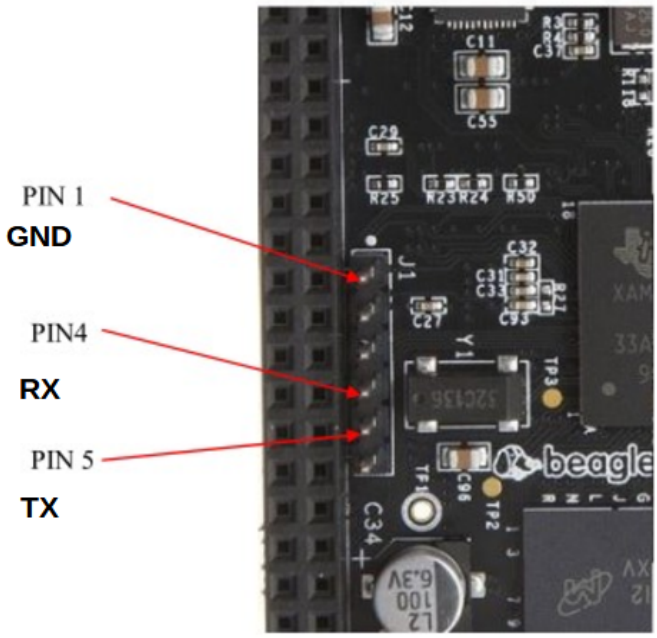
Figura 15 – Arquitetura do ambiente físico de desenvolvimento



Fonte: TI (2019)

Nesse momento, pode-se começar a montar os componentes na bancada (ou mesa de trabalho, pois o termo é apenas de referência). A ordem recomendada é a seguinte: primeiro, insere-se o cartão SD na placa. Em seguida, conecta-se o conversor FTDI para serial na BeagleBone Black, na sua interface serial com os *jumpers* fêmea-fêmea. Apenas 3 dos pinos do conversor precisam ser conectados, que são o GND, o RX e o TX; os pinos RX e TX do conversor devem ser conectados ao TX e RX da placa, respectivamente. A pinagem da BeagleBone Black é mostrada abaixo, na figura 16.

Figura 16 – Pinagem serial na BeagleBone Black



Adaptado de Coley (2014)

Em seguida, conecta-se o conversor FTDI para serial no *host* utilizando um dos cabos mini USB. Não se deve ligar a placa ainda, pois isso a iniciaria automaticamente. Percebe-se que o dispositivo serial já é reconhecido pelo *host*, pois o comando *ls /dev/tty**, que retorna o nome de cada dispositivo *tty* conectado, mostra o nome da interface, que é */dev/ttyUSB0*.

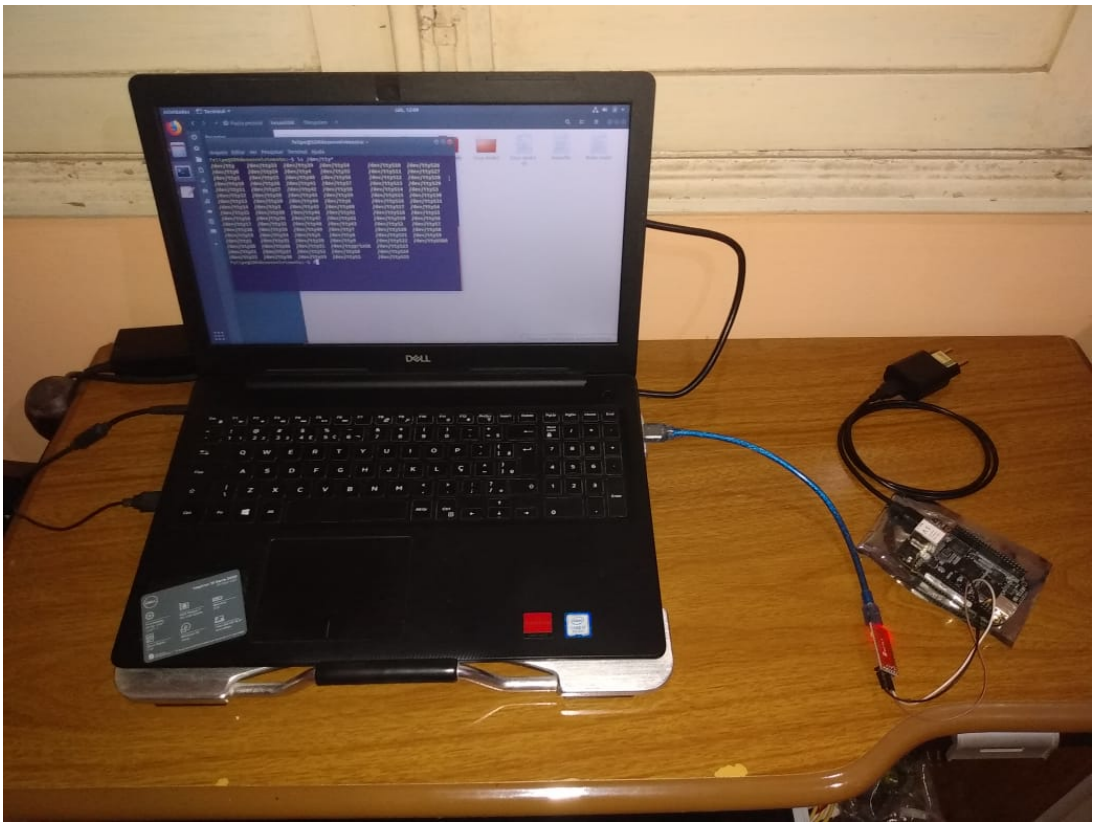
Até esse ponto, quase todos os materiais apresentados no capítulo de desenvolvimento da proposta (3.1 – Materiais utilizados) já foram apresentados, isto é, qual a sua função no desenvolvimento. O único ainda não mostrado é o cabo par trançado, que será utilizado no capítulo 14 - Habilitação do SSH como *shell* remoto.

Um comentário é válido acerca da fonte de alimentação da placa. Existem duas formas de ligá-la, conforme Coley (2014): pela por uma fonte de alimentação (entrada P4) ou pela entrada mini USB. A primeira exige uma fonte com interface compatível, e pode fornecer até 1A. Já a segunda é limitada a apenas 500 mA. Como neste desenvolvimento não havia uma fonte de alimentação disponível, optou-se por alimentar a placa pela porta mini USB, pois, mesmo com a menor corrente, isso é o suficiente para situações de desenvolvimento, nas quais a placa não está sob processamento muito intensivo. Além disso, o carregador original Turbo da Motorola fornece uma tensão confiável de 5V, o que contribui para o seu uso.

Dessa forma, a bancada de trabalho está pronta. A organização dela é apresentada na figura 17. Percebe-se a BeagleBone Black conectada ao notebook pelo cabo serial, já reconhecida pelo mesmo, conforme apresentado no *shell*. Além disso, o cabo de energização (mini USB) já está conectado à placa, juntamente com o carregador de 5V, porém, não está ligado ainda. A

partir de agora, o ambiente está pronto para o desenvolvimento, tanto na parte física – bancada – quanto virtual – sistema operacional e SDK.

Figura 17 – Bancada de desenvolvimento embarcado Linux



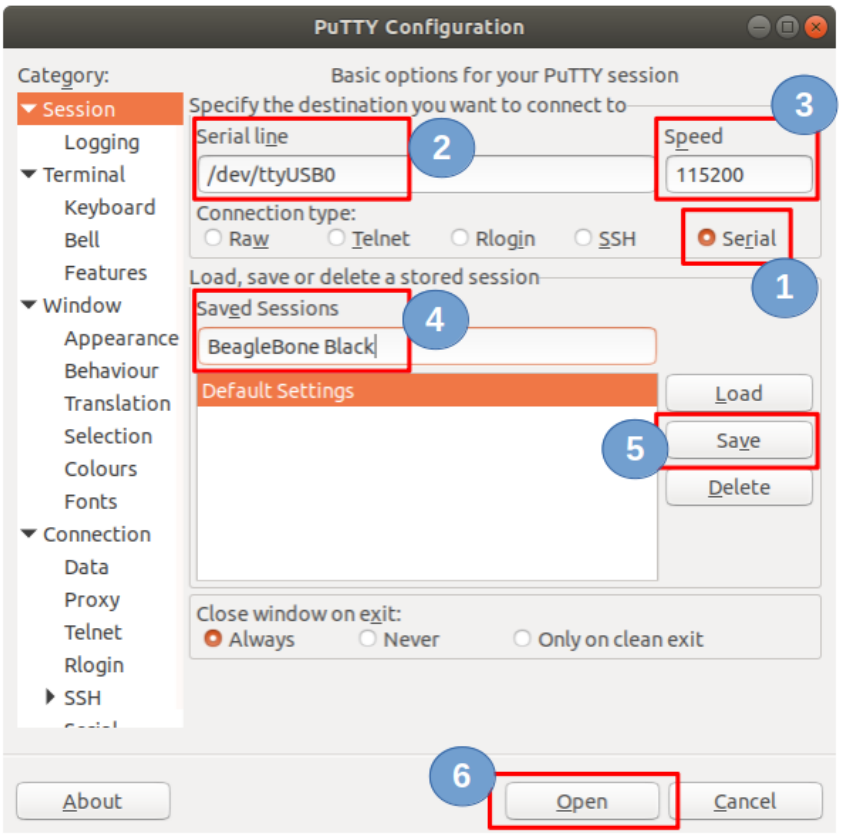
Fonte: AUTOR (2021)

8.3 DEMONSTRAÇÃO DA INICIALIZAÇÃO

Tendo o ambiente pronto, pode-se inicializar a placa pela primeira vez. Nesse momento, inicia-se o emulador de terminal PuTTY, instalado no capítulo anterior, e abre-se uma conexão serial com a BeagleBone Black.

As especificação são conforme as demonstradas na figura 18: Escolhe-se a opção *Serial* no programa e seta-se os valores de *Serial line* como */dev/ttyUSB0* e *Speed* como 115200. Além disso, pode-se também salvar essa configuração para uso posterior, preenchendo o campo *Saved Sessions* com “BeagleBone Black” e clicando em *Save*. Feito isso, a interface pode ser aberta clicando em *Open*. Uma tela vazia aparecerá esperando por informações vindas da placa.

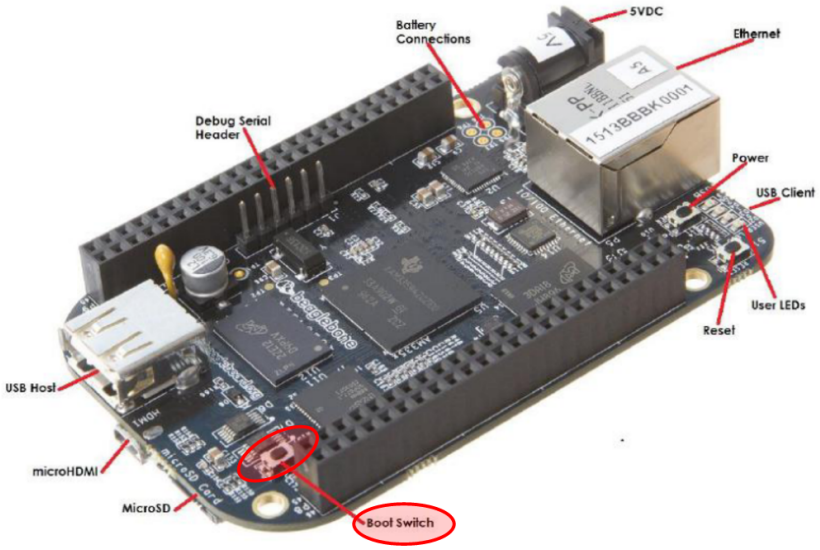
Figura 18 – Utilização do PuTTY para o acesso à serial



Fonte: AUTOR (2021)

No momento de ligar a placa, é necessário manter pressionado o botão de BOOT, destacado na figura 19. Segundo Coley (2014), isso é necessário pois o processador possui duas interfaces de memória, que são chamadas de MMC. Uma delas, a MMC1, está ocupada com a eMMC que já vem soldada na placa e a outra, MMC0, corresponde a entrada do cartão SD. Para iniciar o sistema utilizando o cartão de memória (e não a eMMC), é necessário manter pressionado o botão de BOOT para que o *SoC* faça a seleção correta. Essa configuração pode ser alterada em uma situação de produção de uma placa real, via multiplexação do *SoC am335x* com a ferramenta *PinMux*, da Texas Instruments (COLEY, 2014; TI, 2019).

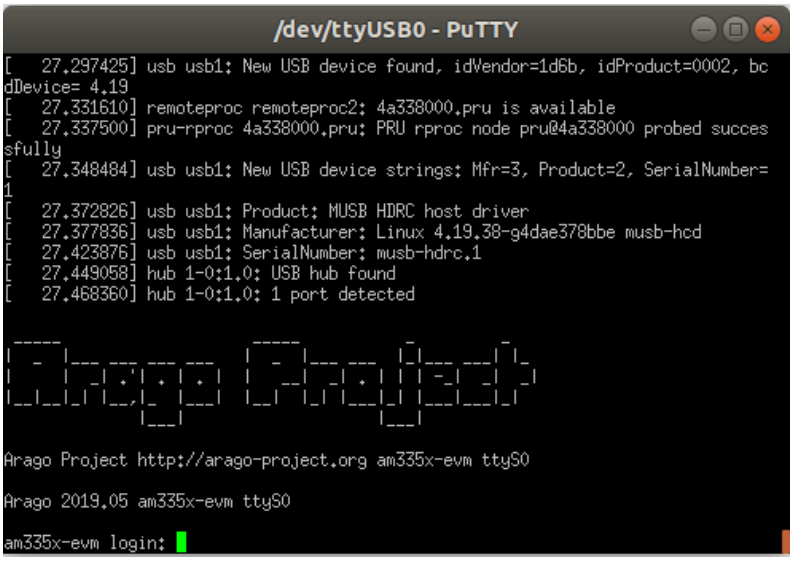
Figura 19 – Botão de Boot da BeagleBone Black



Adaptado de Coley (2014)

Finalmente, a placa pode ser energizada conforme as instruções dadas acima. Se tudo foi feito corretamente, diversas informações de inicialização deverão aparecer na tela do terminal PuTTY. Essas informações, bem como questões envolvendo o *bootloader*, o kernel Linux, o sistema de arquivos e outras coisas serão analisadas com mais detalhes nos próximos capítulos. Por fim, a tela de login aparece – figura 20 – e a placa está pronta para ser utilizada.

Figura 20 – Tela de login BeagleBone Black



Fonte: AUTOR (2021)

Parte III

Configuração dos softwares essenciais

9 CONFIGURAÇÃO DO INICIALIZADOR: U-BOOT

Os sistemas computacionais precisam ser inicializados, e isso varia sobremaneira, dependendo da aplicação. Nos sistemas embarcados baseados em Linux – aplicação robusta – se faz muito uso do U-Boot, um *bootloader* completo, porém leve. O objetivo desse capítulo é introduzir o U-Boot, explicar sua funcionalidade, sua compilação e uso.

Desse modo, este capítulo está distribuído em: introdução aos inicializadores – tópico 9.1, introdução ao U-Boot – tópico 9.2, funcionamento do U-Boot – tópico 9.3, compilação – tópico 9.4, instalação – tópico 9.5, utilização – tópico 9.6 e personalização – tópico 9.7.

9.1 INTRODUÇÃO AOS INICIALIZADORES

Quando a alimentação é aplicada em uma placa, muitos elementos de hardware devem ser inicializados antes mesmo que o programa mais simples possa ser executado. Cada arquitetura e processador tem um conjunto de ações e configurações predefinidas, que incluem buscar algum código de inicialização em um dispositivo de armazenamento (geralmente memória Flash) (HALLINAN, 2006). Esse programa de inicialização, ou *bootloader*, é responsável por dar vida ao processador e aos componentes de hardware relacionados.

Embora um *bootloader* seja executado por um período muito curto durante a inicialização do sistema, ele é, no entanto, um componente de sistema extremamente importante. Quase qualquer sistema executando um kernel Linux precisa de um *bootloader*, mas os sistemas embarcados geralmente têm outras restrições que tornam o processo um pouco diferente daquele usado por um sistema de desenvolvimento Linux para desktop típico (YAGHMOUR et al., 2008).

A maioria dos sistemas de desktop e servidores possuem um firmware – BIOS e UEFI, por exemplo – que fornece informações como a configuração de dispositivos de hardware, detalhes de roteamento de interrupções e outras informações que o Linux precisará usar posteriormente. Os sistemas Linux embarcados, no entanto, geralmente não têm esse privilégio. Em vez disso, eles executam essas tarefas por meio do inicializador, que contém a funcionalidade dos firmwares usados em sistemas maiores (YAGHMOUR et al., 2008).

Segundo Yaghmour et al. (2008), há uma grande quantidade de *bootloaders* disponíveis para Linux, milhares de placas embarcadas e muitas configurações de inicialização possíveis para uma única placa. Além disso, o número e a qualidade desses variam muito entre as arquiteturas. Algumas arquiteturas possuem inicializadores bem conhecidos e estabelecidos que fornecem suporte para uma variedade de hardware. Outros têm poucos ou nenhum inicializador padrão e usam principalmente outros fornecidos pelo fabricante do hardware.

Em um sistema Linux embarcado, o inicializador tem duas tarefas principais, de acordo

com Simmonds (2017), Yaghmour et al. (2008): inicializar o sistema em um nível básico e carregar o kernel do armazenamento. Na realidade, o primeiro trabalho é um tanto subsidiário do segundo, pois é necessário apenas preparar o sistema (hardware) para alcançar a etapa de inicialização do kernel.

Em relação a inicialização básica da placa, o programa precisará inicializar as temporizações da RAM nos circuitos do controlador de memória (MMU), liberar os caches do processador, habilitar vários dispositivos de hardware, implementar diretamente o suporte da infraestrutura de rede, programar os registros da CPU com os valores padrão e realizar uma série de outras tarefas (YAGHMOUR et al., 2008; HALLINAN, 2006). Ele também precisará determinar precisamente qual hardware está instalado no sistema e fornecer isso ao kernel do Linux na forma de tabelas de software dependentes da arquitetura (HALLINAN, 2006). Esse código de inicialização inicial quase sempre é escrito na linguagem *Assembly* nativa do processador (HALLINAN, 2006).

No que diz respeito ao carregamento e inicialização do kernel, o *bootloader* é responsável por localizar, carregar e passar a execução para o sistema operacional principal. De forma mais específica, a etapa de inicialização do sistema por *bootloaders* envolve a seleção do kernel (caso haja mais de um) e o carregamento um sistema de arquivos inicial baseado em RAM (seja *initrd*, *initramfs* ou algo diferente) (YAGHMOUR et al., 2008). Ao contrário do modelo PC-BIOS tradicional, quando o SO assume o controle, o inicializador é sobrescrito e deixa de existir (HALLINAN, 2006).

Uma função secundária do *bootloader* é fornecer um modo de manutenção para atualizar as configurações de inicialização, carregar novas imagens de inicialização na memória e, talvez, executar diagnósticos. Isso geralmente é controlado por uma interface de usuário de linha de comando simples – serial, na maioria dos casos (SIMMONDS, 2017).

9.2 INTRODUÇÃO AO U-BOOT

Apesar de existirem muitos inicializadores, no mundo embarcado Linux, existe um que se destaca. O *Das U-Boot* – nome completo do software – é, de longe, o mais rico, mais flexível, e o *bootloader* embarcado de código aberto em desenvolvimento mais ativo disponível atualmente (YAG). Wolfgang Denk, da *DENX Software Engineering*, escreveu e atualmente mantém o U-Boot, e uma ampla gama de desenvolvedores contribui para ele (YAGHMOUR et al., 2008; SIMMONDS, 2017; DENK; ZUNDEL, 2011).

O U-Boot foi desenvolvido, à priori, para processadores embarcados baseados em PowerPC. Depois, ele foi portado para outras arquiteturas, como ARM (foco deste trabalho), MIPS e SH (SIMMONDS, 2017). Atualmente, ele suporta diversos processadores e diversas placas de desenvolvimento (YAGHMOUR et al., 2008; DENK; ZUNDEL, 2011). Se o desenvolvedor escolher uma placa de desenvolvimento cujo suporte ao Linux está garantido, é muito provável que essa placa use o U-Boot (YAGHMOUR et al., 2008). Com a placa deste trabalho, a

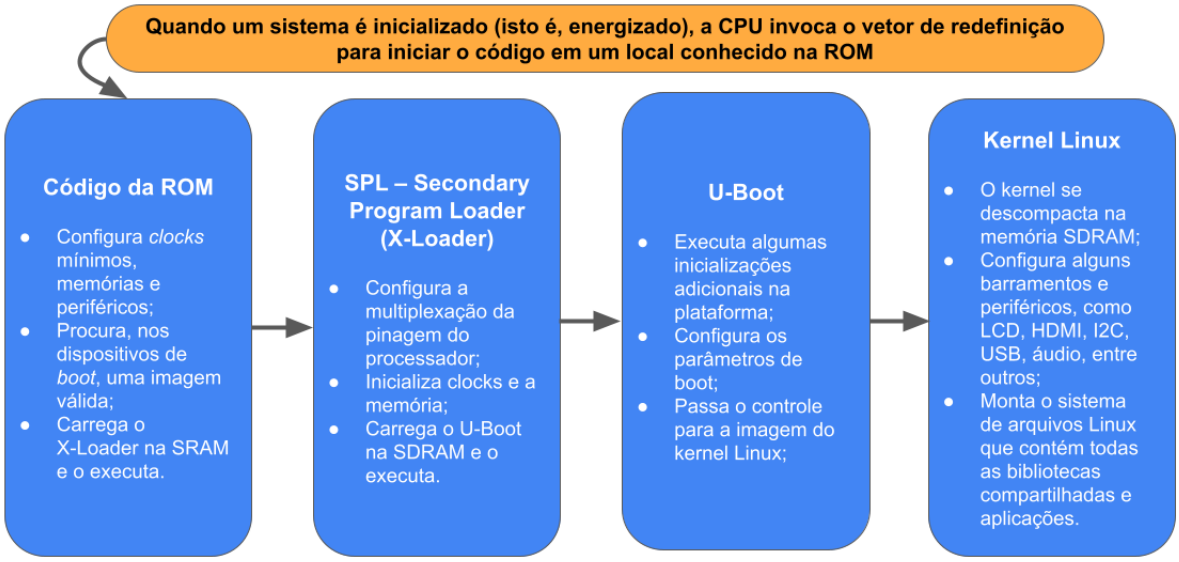
situação não é diferente.

O U-Boot possui diversas funcionalidades, que contribuem para a sua fama. A principal delas é o suporte a interface de rede (Ethernet), com a capacidade de iniciar o kernel via rede pelo protocolo TFTP, além de permitir isso pela entrada USB ou pela memória FLASH (método padrão deste trabalho) (YAGHMOUR et al., 2008). O U-Boot também possui suporte a diversos sistemas de arquivos, possibilidade de configuração de diversas funcionalidades e comandos, além de uma extensa e detalhada documentação (YAGHMOUR et al., 2008).

9.3 FUNCIONAMENTO

O objetivo básico do U-Boot é inicializar o processador e os seus barramentos, preparando, assim, o dispositivo para o carregamento do sistema operacional. Essa função, porém, possui algumas etapas pré-definidas, visto que o processador é consideravelmente complexo, e nem todos as suas unidades são iniciadas ao mesmo tempo, o que justifica essa necessidade de divisão. O funcionamento do U-Boot, isto é, as suas operações enquanto ativo na memória, são apresentadas na forma de diagrama de blocos, na figura 21. O detalhamento dessas fases é apresentado logo em seguida.

Figura 21 – Sequência de inicialização do U-Boot



Adaptado de TI (2019)

Primeiramente, em uma situação de *reset* ou energização da placa, o primeiro código a ser executado reside no próprio processador, e é conhecido como código da ROM. Esse código vem de fábrica e não pode ser alterado. Esse código possui apenas a memória SRAM, interna ao *SoC*, para uso, já que a DRAM, externa, ainda precisa ser inicializada. A função básica desse pequeno programa seria chamar o U-Boot da memória Flash, porém, a SRAM é muito pequena

para ele (é da ordem de KBs em *SoCs* embarcados). Nesse sentido, cabe ao código de RAM chamar um inicializador intermediário, o MLO (SIMMONDS, 2017).

O inicializador de primeiro estágio, também chamado de SPL – *Secondary Program Loader* – configura a memória RAM (DRAM) para o carregamento do U-Boot. Devido ao fato de o controlador da DRAM ser muito dependente do dispositivo em que ele está para ser inicializado, é necessário um programa configurável para tal. O MLO, também chamado de *X-loader* no caso de *SoCs* da Texas Instruments, cumpre esse requisito, pelo fato de ser compilado junto do U-Boot, que é aberto. Uma vez que a memória RAM está utilizável, o U-Boot pode, finalmente, ser carregado da memória flash com o nome *u-boot.img* (SIMMONDS, 2017).

Por fim, o inicializador de segundo estágio – o U-Boot propriamente dito – é carregado para a memória RAM. É nessa etapa que a interface via linha de comando aparece, assim como a possibilidade de usar algumas funções do U-Boot. Esse programa termina a inicialização de outras interfaces e periféricos que possam ter faltado na etapa anterior e, principalmente, carrega o kernel. Ao final dessa terceira fase, o *bootloader* é sobrescrito pelo kernel na memória RAM (SIMMONDS, 2017).

9.4 COMPILAÇÃO

Deste ponto em diante, será descrita a configuração do U-Boot. Esse padrão se repetirá em vários outros capítulos deste trabalho, nos quais serão abordados outros pacotes de softwares, como o BusyBox e o kernel Linux.

Para desenvolver com o U-Boot, é necessário ter o seu código-fonte. O SDK da Texas Instruments já os traz, na versão 2019.01. Caso não se estivesse usando o SDK, ele precisaria ser baixado da página oficial, o que é recomendado por Simmonds (2017), Yaghmour et al. (2008), Denk e Zundel (2011) para ser feito pelo comando abaixo, para garantir sempre a versão mais atual.

```
1 $ git clone git://git.denx.de/u-boot.git
```

Uma vez com o código-fonte do U-Boot em mãos, parte-se para a configuração. O código-fonte do U-Boot está na pasta *board-support* do SDK. Antes de proceder com alguma operação, aplica-se a boa prática de clonagem do diretório, para servir de backup, conforme explicitado no Apêndice A – Compilação Cruzada. Nomeou-se essa pasta para *u-boot-2019.01-RASCUNHO*, conforme demostrado na sequência de comandos abaixo. Após tudo isso, acessa-se o diretório para iniciar a configuração.

```
1 $ cd ~/texasSDK/board-support
2 $ cp -r u-boot-2019.01+gitAUTOINC+8b90adfb16-g8b90adfb16/ u-boot
    ↪ -2019.01-RASCUNHO
3 $ cd /home/felipe/texasSDK/board-support/u-boot-2019.01-RASCUNHO
```

O primeiro passo na compilação cruzada de um software é declarar a arquitetura pra a qual se quer produzir o executável. O U-Boot, assim com a maioria dos programas expostos neste trabalho, não precisam de mais nada além da exportação das variáveis de ambiente ARCH e CROSS_COMPILE, portanto, declaram-se as variáveis no início do *script*.

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

No procedimento de compilação, é comum que haja a possibilidade de escolher outra pasta de saída para o produto de software (seja um executável, sejam bibliotecas compartilhadas, etc.), já que o padrão é que a saída seja dentro do próprio diretório do código-fonte. O manual do U-Boot (DENK; ZUNDEL, 2011) especifica que pode ser adicionado um parâmetro ao *make* para apontar para uma pasta de saída. O formato é *O=/pasta/de/saída*. Levando em consideração, será declarado o caminho de saída da compilação, conforme a boa prática exposta no apêndice de compilação cruzada.

```
$ export RAIZ="/home/felipe/texasSDK/board-support/u-boot_build/"
```

Com todos esses passos feitos, é o momento de iniciar a configuração. À priori, o diretório do U-Boot está limpo, isto é, não foi feita nenhuma compilação nele ainda, então não é necessário limpá-lo (TI, 2019; DENK; ZUNDEL, 2011). Por conta disso também, o U-Boot não tem nenhuma configuração pré-definida, ou seja, não está preparado para ser compilado para nenhum tipo de *SoC* ou placa. Portanto, é necessário selecionar qual modelo de placa se está utilizando.

O U-Boot possui suporte a diversas plataformas de desenvolvimento, placas e *SoCs* diferentes. Os arquivos de configuração das placas suportadas se encontram na pasta *configs*. Para selecionar a configuração padrão cedida pelo SDK, usa-se o comando abaixo (TI, 2019; DENK; ZUNDEL, 2011).

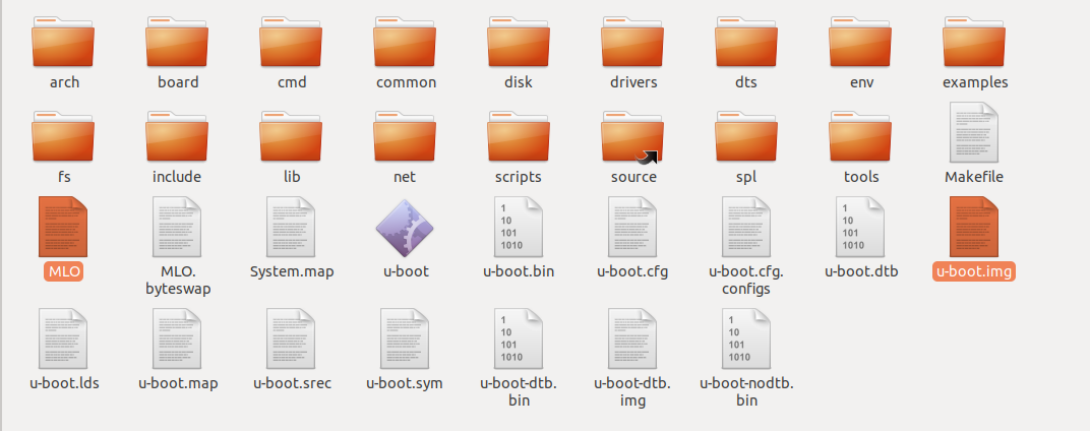
```
$ make O=$RAIZ am335x_evm_defconfig
```

Nota-se que a variável RAIZ está sendo usada com um "$", o que indica um chamamento do conteúdo dela. Além disso, o *O=/pasta/de/saída* está sendo utilizado aqui, embora não seja a compilação ainda, visto que, conforme o manual oficial (DENK; ZUNDEL, 2011), caso se queira utilizar outra pasta para a saída, deve-se utilizar esse parâmetro em todos os comandos de *make*.

Assim, tudo está pronto para a compilação, executada com o *make* descrito no console abaixo. Na pasta de saída são produzidos diversos arquivos. Esses arquivos costumam ter utilidade em outras plataformas e situações (DENK; ZUNDEL, 2011), porém, aqui apenas dois deles serão utilizados. Tais arquivos estão destacados na figura 22.

```
$ make O=$RAIZ -j9
```

Figura 22 – Resultado da compilação do U-Boot

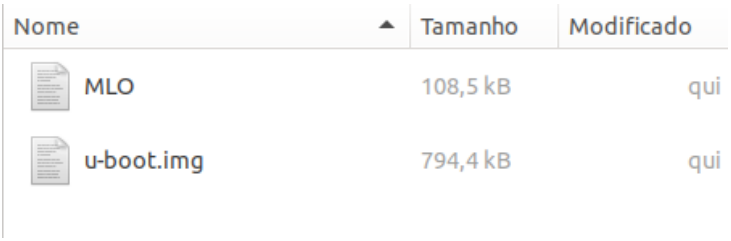


Fonte: AUTOR (2021)

9.5 INSTALAÇÃO

Para instalar o U-Boot compilado, insere-se o cartão micro SD no *host* e faz-se a transferência para a partição *boot* particionada anteriormente. Percebe-se que já existe o U-Boot e o MLO nessa partição (que serão sobrescritos), pois fora transferido um sistema pronto e funcional para o cartão SD para fins de testes. Percebe-se, também, que o tamanho dos arquivos, compilados e que já estão no SD, são os mesmos, indicando que a compilação foi feita usando a configuração padrão em ambos os casos. A partição *boot* do cartão de memoria é mostrada na figura 23.

Figura 23 – Partição *boot* do cartão micro SD



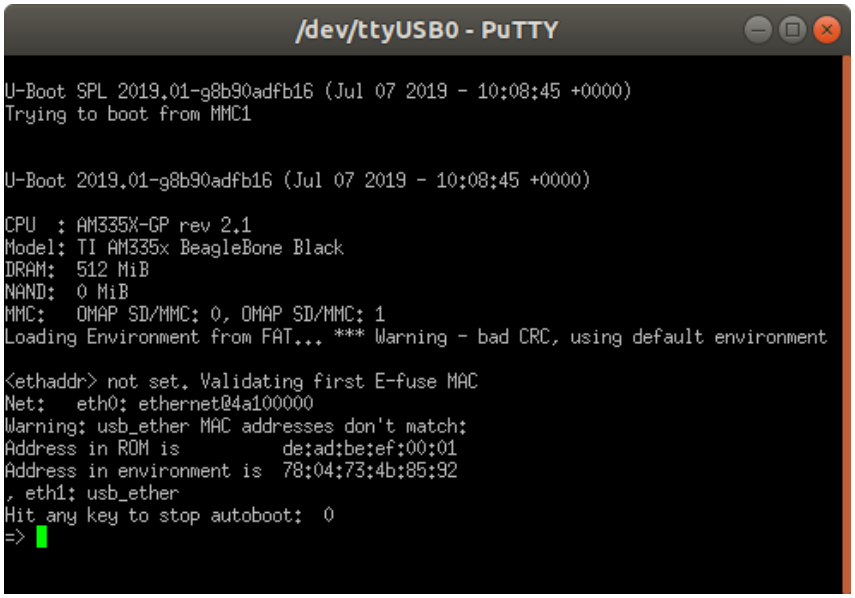
Fonte: AUTOR (2021)

Existem diversos outros métodos para passar o programa referido para a placa de desenvolvimento. Dependendo da situação (e da placa), pode haver a necessidade de transferir o U-Boot utilizando a UART; ou a interface JTAG (para depuração); ou ainda, o *boot* pode ser feito via rede, pelo protocolo TFTP (YAGHMOUR et al., 2008; SIMMONDS, 2017). Inclusive, existe a possibilidade de atualizar o firmware da placa – no caso, o inicializador – via rede utilizando esse último protocolo (TI, 2019; YAGHMOUR et al., 2008).

9.6 UTILIZAÇÃO

Após a transferência, testa-se o *bootloader* compilado, e vê-se que o resultado é como no primeiro teste realizado, no qual o U-Boot apresenta informações iniciais no console, conta até 3 (tempo para o usuário intervir) e, caso não haja entrada pelo usuário, inicia o kernel Linux.

Figura 24 – Interface do U-Boot



Fonte: AUTOR (2021)

Se o usuário escolher intervir durante a contagem, uma interface baseada em comandos aparecerá, similar à do próprio Linux – vide figura 24. O U-Boot, além de inicializador, serve também para verificar o estado do hardware e fazer testes como uma ferramenta, fator verificado pelas primeiras linhas do console, no qual o programa imprime informações sobre o hardware em que ele está rodando (YAGHMOUR et al., 2008).

Durante o processo de desenvolvimento de um sistema embarcado, é comum que determinados componentes apresentem falhas (por soldagem, por exemplo). Portanto, conseguir analisar o seu estado, em nível baixo, é extremamente útil para um desenvolvedor de hardware. O U-Boot fornece diversos comandos de baixo nível que permitem analisar o estado de um componente, escrevendo e lendo bytes diretamente ou obtendo o estado dele, quando possível (YAGHMOUR et al., 2008; DENK; ZUNDEL, 2011).

Tendo isso em vista, nesta seção serão mostradas algumas capacidades básicas do U-Boot, baseadas no manual oficial (DENK; ZUNDEL, 2011) e na documentação do SDK (TI, 2019), visando demonstrar a utilização do mesmo conforme se faz em situações reais, da melhor forma possível. Será dado foco aos comandos do U-Boot e as suas variáveis de ambiente.

O primeiro comando – e mais importante – do terminal do U-Boot é o *help*, que imprime todos os comandos compilados com a versão usada do software, além de fornecer ajuda específica

sobre comandos (YAGHMOUR et al., 2008; DENK; ZUNDEL, 2011).

```
=> help
?       - alias for 'help'
askenv  - get environment variables from stdin
base    - print or set address offset
bdinfo  - print Board Info structure
blkcache- block cache diagnostics and control
boot    - boot default, i.e., run 'bootcmd'
(...)
```

```
=> help cp
cp - memory copy

Usage:
Cp [.b, .w, .l] source target count
```

O U-Boot possui diversos comandos para análise que retornam informações sobre componentes, como versão e estrutura. Alguns dos mais importantes são listados abaixo, e demonstrados via console logo em seguida, respectivamente.

- ***dm tree:*** Esse comando imprime a lista de *drivers* de baixo nível do *SoC*;
- ***bdinfo:*** Informações da placa na qual está sendo executado o U-Boot (*Board info*);
- ***coninfo:*** Dispositivos de console e informações (*Console info*);
- ***gpio status:*** Consulta e controle do GPIO, nesse caso, imprime as informações em baixo nível, para desenvolvedores;
- ***i2c bus:*** Idêntico ao anterior, só que aplicado à interface I2C;
- ***mmc (info, part e list):*** Consulta e controle da MMC, especialmente a que o U-Boot fora carregado (o cartão SD, no caso). Permite mostrar informações sobre a MMC requerida, mostrar partições e quantas estão conectadas ao *SoC*, respectivamente.

```
=> dm tree
 Class       index  Probed    Driver                Name
-----------------------------------------------------------------
 root            0    [ + ]     root_driver           root_driver
 rsa_mod_ex      0    [   ]     mod_exp_sw            |-- mod_exp_sw
 simple_bus      0    [ + ]     generic_simple_bus    `-- ocp
 simple_bus      1    [   ]     generic_simple_bus        |-- l4_wkup@44c00000
 simple_bus      2    [   ]     generic_simple_bus        |   |-- prcm@200000
 simple_bus      3    [   ]     generic_simple_bus        |   `-- scm@210000
 simple_bus      4    [   ]     generic_simple_bus        |       `-- scm_conf@0
 gpio            0    [ + ]     gpio_omap                 |-- gpio@44e07000
 (...)
```

```
=> bdinfo
arch_number = 0x00000e05
boot_params = 0x80000100
DRAM bank   = 0x00000000
-> start    = 0x80000000
-> size     = 0x20000000
baudrate    = 115200 bps
(...)
```

```
=> coninfo
List of available devices:
serial@44e09000 00000007 IO stdin stdout stderr
serial   00000003 IO
```

```
=> gpio status
Bank gpio@44e07000_:
gpio@44e07000_6: input: 0 [x] mmc@48060000.cd-gpios
```

```
=> i2c bus
Bus 0:  i2c@44e0b000
Bus 2:  i2c@4819c000
```

```
=> mmc info
Device: OMAP SD/MMC
Manufacturer ID: 3
OEM: 5344
Name: SU04G
Bus Speed: 48000000
Mode : SD High Speed (50MHz)
(...)

=> mmc part

Partition Map for MMC device 0  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
  1     2048            143360          33f0d4fd-01     0c Boot
  2     145408          7680000         33f0d4fd-02     83

=> mmc list
OMAP SD/MMC: 0 (SD)
OMAP SD/MMC: 1
```

As informações retornadas por esses comandos são usadas por projetistas para diagnóstico do dispositivo – isto é, saúde dos chips (RAM, MMC, controladores) e se a resposta dos mesmos está de acordo com o planejado (ou seja, se o projeto não precisa de alteração) (SIMMONDS, 2017; YAGHMOUR et al., 2008; OSBORN, 2018).

O ambiente U-Boot é um bloco de memória que é salvo na memória permanente e copiado para a memória RAM durante a inicialização. Ele é usado para guardar variáveis de ambiente que podem ser usadas para configurar o sistema (DENK; ZUNDEL, 2011). O uso dessas variáveis é bem similar às dos *shells* Unix como o *bash* (YAGHMOUR et al., 2008), portanto, servem para ajustar o comportamento do inicializador (DENK; ZUNDEL, 2011). Para

visualizá-las, usa-se o comando *printenv*. Algumas das principais foram selecionadas para ser em mostradas no console abaixo, de forma abreviada.

```
=> printenv
arch=arm
baudrate=115200
board=am335x
board_name=A335BNLT
board_rev=000C
boot_targets=mmc0 legacy_mmc0 mmc1 legacy_mmc1 nand0 pxe dhcp
bootdir=/boot
bootenvfile=uEnv.txt
bootfile=zImage
cpu=armv7
ethaddr=78:04:73:4b:85:90
soc=am33xx
stderr=serial@44e09000
stdin=serial@44e09000
stdout=serial@44e09000
vendor=ti
ver=U-Boot 2019.01-g8b90adfb16 (Jul 07 2019 - 10:08:45 +0000)
```

As variáveis de ambiente são montadas em tempo de compilação e podem ser alteradas definitivamente com o comando *saveenv*. Algumas dessas variáveis são passadas ao kernel Linux no momento da inicialização por meio de uma estrutura chamada *device tree* (DENK; ZUNDEL, 2011).

Por fim, o kernel pode ser iniciado com o comando *boot*:

```
=> boot
switch to partitions #0, OK
mmc0 is current device
SD/MMC found on device 0
** Unable to read file boot.scr **
** Unable to read file uEnv.txt **
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
switch to partitions #0, OK
mmc0 is current device
SD/MMC found on device 0
4080128 bytes read in 265 ms (14.7 MiB/s)
36717 bytes read in 4 ms (8.8 MiB/s)
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Device Tree to 8fff4000, end 8fffff6c ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.19.38-g4dae378bbe (oe-user@oe-host) (gcc version 8.3.0 (GNU
    ↪ Toolchain for the A-profile Architecture 8.3-2019.03 (arm-rel-8.36))) #1 PREEMPT Sun
    ↪ Jul 7 04:39:33 UTC 2019
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] OF: fdt: Machine model: TI AM335x BeagleBone Black
(...)
```

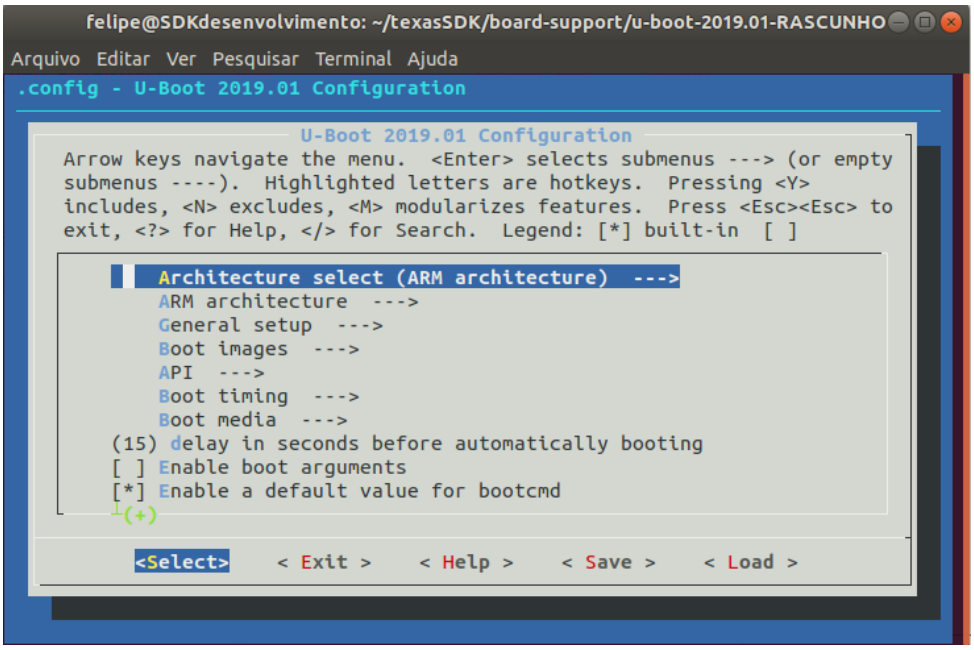
9.7 PERSONALIZAÇÃO

Embora o U-Boot já seja um software muito eficiente e otimizado para as suas funções, é normal que se queira performá-lo ainda mais, tornando a sua configuração específica para o sistema embarcado em desenvolvimento, aumentando, assim, a sua performance e segurança para o produto final. Desse modo, é necessário personalizá-lo, conteúdo esse que será abordado nesse tópico.

O U-Boot, assim como vários outros pacotes de software presentes neste trabalho, permitem configuração, isto é, adição ou remoção de recursos. Essa alteração é feita, em geral, utilizando o comando *menuconfig*. Antes de iniciá-lo, é recomendável limpar as informações das últimas compilações realizadas utilizando o comando *distclean*. Após isso, seleciona-se novamente o modelo de configuração do *SoC am335x* e, finalmente, inicia-se o *menuconfig*. Os respectivos comandos estão listados no console abaixo, e a interface é mostrada na figura 25.

```
1 $ make O=$RAIZ distclean
2 $ make O=$RAIZ am335x_evm_defconfig
3 $ make O=$RAIZ menuconfig
```

Figura 25 – Tela de configuração (*menuconfig*) do U-Boot



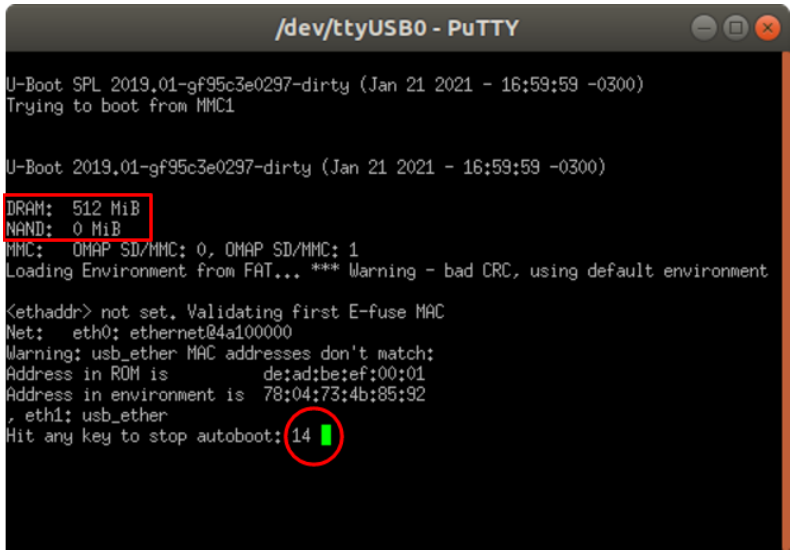
Fonte: AUTOR (2021)

A interface do *menuconfig* apresenta as configurações disponíveis baseadas na configuração padrão selecionada, *am335x_evm_defconfig*. O motivo de escolher uma configuração padrão é que ela já está plenamente ajustada à placa/*SoC*, poupando, assim, tempo do desenvolvedor nessa etapa. Quando se altera alguma configuração, geralmente é com o objetivo de otimização para o dispositivo, visando menor tempo de *boot* em *smartphones*, por exemplo (TI, 2012).

O U-Boot permite, em geral, configurar diversos comandos do seu *shell*, além de suporte a interfaces, como SPI e Ethernet. Permite seleção de arquiteturas (pois suporta várias); configurações de *boot*; configurações gerais, como frequência de alguns subcomponentes na inicialização; dentre muitas outras. Percebe-se, pela figura 25, que muitas delas já estão ajustadas devido a configuração padrão do *SoC am335x*.

Está fora do escopo deste trabalho demonstrar todas as configurações possíveis, porém, pode-se demonstrar o efeito de algumas alterações. Uma delas é ajustar, ou até mesmo retirar, a contagem para o *boot* do kernel, mostrado anteriormente. Conforme a figura 25, esse valor foi ajustado para 15, porém, é possível retirá-lo também, o que é útil para ambientes de produção, pois acelera a inicialização (TI, 2012). Além disso, também removeu-se a função de imprimir as informações da placa durante o *boot*. Após a recompilação, o resultado fica conforme a figura 26, na qual percebe-se as mudanças conforme as marcações em vermelho.

Figura 26 – Interface do U-Boot reconfigurada



Fonte: AUTOR (2021)

Dependendo do projeto, talvez seja interessante configurar algumas opções nessa interface *menuconfig* para fins de otimização. Para isso, a Texas Instruments fornece um treinamento específico (TI, 2012). Apesar disso, pode-se usar essa interface para configurar outras coisas, como a função de atualização e uma tela de carregamento (caso o dispositivo possua um *display*), chamada tela de abertura (*Splash Screen*) (TI, 2019).

O *script* completo da configuração realizada nesse capítulo é mostrado abaixo, na figura 27.

Figura 27 – *script* de compilação e personalização do U-Boot

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/u-boot_build/"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/u-boot-2019.01-RASCUNHO"

## INÍCIO ##

# abrir a pasta do código-fonte u-boot
cd $SOURCE

# limpar compilações anteriores
make O=$RAIZ distclean

# selecionar a configuração padrão do SoC da BeagleBone Black
make O=$RAIZ am335x_evm_defconfig

# personalizar
make O=$RAIZ menuconfig

# compilar
make O=$RAIZ -j9
```

Fonte: AUTOR (2021)

10 ESCOLHA DOS COMANDOS BÁSICOS: BUSYBOX

Os comandos de sistema são essenciais em sistemas Linux; sem eles, não é possível utilizar o dispositivo. É preciso configurá-los como parte do processo de desenvolvimento de um sistema Linux embarcado. Assim, o objetivo deste capítulo é conceituar e demonstrar o uso do canivete suíço do Linux embarcado: Busybox.

Por conseguinte, este capítulo está classificado da seguinte forma: conceito do Busybox – tópico 10.1, sua importância – tópico 10.2, download do utilitário – tópico 10.3, personalização e compilação – tópico 10.4, instalação no *target* – tópico 10.5 e demonstração de uso – tópico 10.6.

10.1 CONCEITO

Busybox é um software que combina diversos comandos comuns do Linux em um único executável (ANDERSEN; LANDLEY, 2012). Ele provê suporte para a maioria dos comandos do pacote GNU: manipuladores de arquivos, comandos gerais e configuradores de rede (HALLINAN, 2006; ANDERSEN; LANDLEY, 2012). O Busybox fornece um ambiente completo de comandos para qualquer pequeno (e grande) sistema embarcado baseado em Linux (ANDERSEN; LANDLEY, 2012).

BusyBox foi escrito do zero para executar as funções essenciais dos utilitários essenciais do Linux. Os desenvolvedores aproveitaram a regra 80:20 da programação: os 80% mais úteis de um programa são implementados em 20% do código (SIMMONDS, 2017). Esses comandos costumam ser reduzidos se comparados com as suas contrapartes completas, porém, são o suficiente para a maioria das aplicações embarcadas (SIMMONDS, 2017; ANDERSEN; LANDLEY, 2012).

O entusiasmo da comunidade em torno do Busybox reside na funcionalidade que ele fornece, embora seja um aplicativo muito pequeno (1 MB se compilado estaticamente com *glibc*) (YAGHMOUR et al., 2008). Por exemplo, o Busybox inclui cliente e servidor DHCP, gerenciadores de pacotes (*dpkg* e *rpm*), editor de texto (*vi*) e até um servidor web completo, que satisfaz as demandas típicas de sistemas embarcados Linux.

Além desses fatores, o BusyBox é uma coleção de miniaplicativos que exportam a sua função principal no formato *[miniaplicativo]_main.c* – por exemplo, o comando *cat* reduzido se exporta como *cat_main*. Desse modo, o *main* do Busybox em si despacha a chamada para o miniaplicativo correto, baseado nos argumentos na linha de comando (SIMMONDS, 2017).

Em síntese, uma instalação típica do BusyBox consiste em um único programa com um link simbólico para cada miniaplicativo, mas que se comporta exatamente como se fosse uma

coleção de programas individuais (SIMMONDS, 2017).

10.2 IMPORTÂNCIA DO BUSYBOX

O Busybox ganhou bastante destaque na comunidade de Linux embarcado, pois foi desenvolvido levando em consideração otimização por tamanho e recursos limitados do ambiente (ANDERSEN; LANDLEY, 2012; YAGHMOUR et al., 2008; SIMMONDS, 2017). Tendo isso em vista, algumas vantagens são destacadas.

Primeiramente, os comandos presentes nesse software são implementações mais simples das suas versões de desktop; às vezes apenas com algumas funcionalidades. Na prática, porém, essas funcionalidades são mais que o suficiente para a maioria dos sistemas embarcados (HALLINAN, 2006; YAGHMOUR et al., 2008).

Secundariamente, o Busybox é extremamente modular, ou seja, o desenvolvedor pode incluir ou excluir facilmente comandos ou funcionalidades durante a fase de compilação (ANDERSEN; LANDLEY, 2012; HALLINAN, 2006). Isso facilita consideravelmente a personalização de sistemas embarcados, bastando adicionar um sistema de arquivos diminuto e um kernel Linux para executá-lo (ANDERSEN; LANDLEY, 2012), tarefa essa que será explorada no capítulo 13 – Construção do sistema de arquivos do zero.

Terciariamente, o uso dessa ferramenta salva uma grande quantidade de espaço de armazenamento – dezenas de MB – além de salvar tempo, do desenvolvedor, em ter que configurar o sistema, já que o uso do Busybox torna necessária apenas a compilação de um único executável, em vez de vários para cada comando (YAGHMOUR et al., 2008).

Por fim, o Busybox está disponível para a grande maioria das arquiteturas de processadores embarcados, inclusive a ARM, alvo deste trabalho (YAGHMOUR et al., 2008). Ele permite compilação tanto estática quanto dinâmica, utilizando a *glibc*.

10.3 OBTENÇÃO DO CÓDIGO-FONTE

Diferentemente do U-Boot e do kernel, o código-fonte do Busybox não está presente no SDK da Texas Instruments, embora todos os seus sistemas de arquivos prontos o utilizem. Por isso, é necessário baixá-lo da página oficial antes de iniciar o desenvolvimento com ele.

O procedimento é bastante simples. Primeiro, acessa-se a página oficial do Busybox que contém, em destaque, a versão mais atual (estável) disponível para download (neste trabalho, a versão usada foi a 1.32.1). Após isso, move-se – como boa prática – o código-fonte (já extraído) para a pasta de códigos-fontes do SDK, a fim de manter uma boa organização dos softwares usados no desenvolvimento. Em seguida, clona-se a pasta a fim de manter uma cópia. A ordem desse procedimento em console é mostrada abaixo como referência.

```
# Download do código-fonte
$ cd ~/Downloads/
$ wget https://busybox.net/downloads/busybox-1.32.1.tar.bz2

# Extração
$ tar -xvjf  busybox-1.32.1.tar.bz2

# Movendo o código-fonte para a pasta board-support
$ mv busybox-1.32.1 ~/texasSDK/board-support

# Clonando a pasta
$ cd ~/texasSDK/board-support
$ cp -r busybox-1.32.1 busybox-1.32.1-RASCUNHO
```

10.4 PERSONALIZAÇÃO E COMPILAÇÃO

Uma vez que o código-fonte está pronto, pode-se iniciar o procedimento de compilação cruzada. Como anteriormente, abre-se um terminal e exportam-se as variáveis de ambiente que definem a arquitetura:

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Desse mesmo modo se exporta o caminho de saída e a pasta dos fontes. Uma pasta foi criada no mesmo diretório, chamada *busybox_build* e será o local onde o produto da compilação ficará.

```
$ export RAIZ="/home/felipe/texasSDK/board-support/busybox_build"
$ export SOURCE="/home/felipe/texasSDK/board-support/busybox-1.32.1-
    ↪ RASCUNHO"
```

Tendo-se exportado as varáveis necessárias, abre-se a pasta do código fonte pelo terminal para iniciar a configuração do Busybox pelo comando *make*:

```
$ cd $SOURCE
```

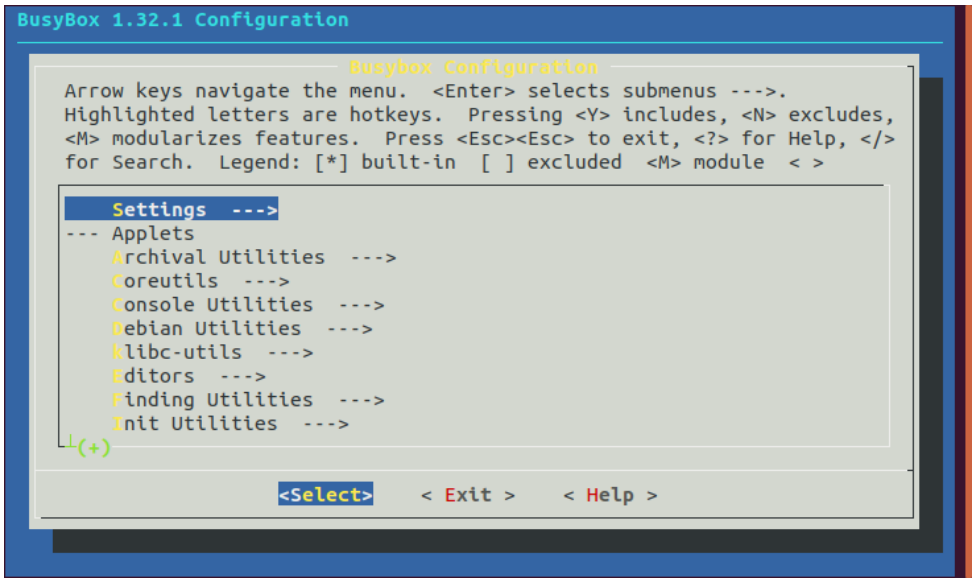
Uma vez estando no diretório fonte, o primeiro passo, conforme a documentação oficial, é selecionar a configuração padrão do Busybox, que é um arquivo *.config* com a maioria das funcionalidades e comandos ativados, inclusive os opcionais (ANDERSEN; LANDLEY, 2012). É uma configuração genérica visando a escolha das funcionalidades posteriormente, por parte do projetista.

```
$ make defconfig
```

Dessa forma, o Busybox está pronto para ser configurado. O comando abre a interface *menuconfig* para que o usuário selecione aquilo que lhe convém que fique na compilação final. A tela de configuração é apresentada na figura 28.

```
$ make menuconfig
```

Figura 28 – Tela de configuração (*menuconfig*) do Busybox



Fonte: AUTOR (2021)

A interface de personalização do Busybox permite que não só sejam selecionados os comandos que estarão no executável final, como também permite a configuração de outras funções, como suporte a arquivos, suporte a arquiteturas e outras opções de compilação, todas disponíveis na seção *Settings*, primeira opção da interface mostrada acima, na figura 28.

Por fim, pode-se salvar as alterações (se foi feita alguma) na opção *Save Configuration to an Alternate File*. No caso deste trabalho, resolveu-se não modificar nenhuma das opções padrão para a compilação. Essa pode ser feita, enfim, com o comando *make*.

```
$ make -j9
```

10.5 INSTALAÇÃO

Após o término da compilação, o produto dessa pode ser instalado na pasta definida para tal: *busybox_build*. A documentação do Busybox define a instalação a partir do comando mostrado abaixo (ANDERSEN; LANDLEY, 2012).

```
$ make CONFIG_PREFIX=$RAIZ install
```

A estrutura do diretório de saída é conforme o console abaixo, que apresenta uma distribuição simplificada da compilação (ou seja, foram excluídos a maior parte dos comandos).

A pasta contém 398 arquivos, portanto, apenas os mais importantes estão representados, além de outros para exemplo. O comando *tree* facilita a visualização.

```
$ tree
.
|---- bin
|    |---- busybox
|    |---- cat -> busybox
|    |---- cpio -> busybox
|    |---- sh -> busybox
|---- linuxrc -> bin/busybox
|---- sbin
|    |---- init -> ../bin/busybox
|    |---- insmod -> ../bin/busybox
|    |---- ip -> ../bin/busybox
|    |---- ipaddr -> ../bin/busybox
+---- usr
     |---- bin
     |    |---- time -> ../../bin/busybox
     |    |---- timeout -> ../../bin/busybox
     |    |---- top -> ../../bin/busybox
     +---- sbin
          |---- addgroup -> ../../bin/busybox
          |---- telnetd -> ../../bin/busybox
          |---- tftpd -> ../../bin/busybox
          (...)
```

Basicamente, o Busybox comporta dentro de si (isto é, o seu executável) versões simplificadas de diversos comandos encontrados em ambientes Linux convencionais. Cada um desses comandos pode ser acessado como um parâmetro do executável original (como *busybox cp* equivale ao *cp* padrão). Porém, para facilitar o acesso, são feitos links simbólicos (durante a instalação) contendo o nome de cada comando adaptado, todos eles apontando para o executável final, o que equivale a digitar o comando como parâmetro.

A grande vantagem no uso do Busybox para sistemas embarcados consiste no fato de que o executável principal contém todos os comandos, e os links simbólicos não representam peso para o sistema de arquivos, ou seja, toda essa hierarquia de executáveis pesa apenas 1 MB, que é o peso do executável Busybox na configuração completa (isto é, com todos os comandos ativos).

Nesse momento, vale destacar que tanto faz instalar no cartão micro SD diretamente (isto é, no sistema de arquivos do *target*) ou em uma pasta no *host* e depois transferir, pois a estrutura do Busybox depende apenas dos seus links simbólicos, ou seja, contanto que a hierarquia dos

diretórios obedeça a fornecida, os links não quebrarão.

O executável do Busybox (e os seus links) são feitos para a instalação em um sistema de arquivos limpo, feito do zero, em geral. Tal ação será demonstrada no capítulo 13. Nesse capítulo, será feita a instalação no sistema de arquivos pronto que já foi passado para o *target*.

Nesse momento, seria feita a instalação de todo o conteúdo da compilação (executável e links simbólicos), porém, apenas o Busybox em si será passado. Isso ocorre pois, como o sistema de arquivos já está pronto, mudanças na sua estrutura causam conflitos. Apenas essa situação é que precisa ser levada em consideração; o resultado final será como se toda a instalação tivesse sido passada, portanto, assim se considerará.

O comando apresentado abaixo demonstra a instalação completa no cartão micro SD. Percebe-se que foi adicionado o sudo ao comando, pois a região do diretório */media* pertence ao *root*, logo, precisa de privilégios de *root*.

```
1 $ sudo make CONFIG_PREFIX=/media/rootfs install
```

O *script* completo da configuração realizada nesse capítulo é mostrado abaixo, na figura 29.

Figura 29 – *script* de compilação e personalização do Busybox

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/busybox_build"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/busybox-1.32.1-RASCUNHO"

## INÍCIO ##

# abrir a pasta do código-fonte do Busybox
cd $SOURCE

# limpar compilações anteriores (se houverem)
make distclean

# selecionar a configuração padrão/genérica do Busybox
make defconfig

# personalizar
make menuconfig

# compilar
make -j9

# instalar
make CONFIG_PREFIX=$RAIZ install
```

Fonte: AUTOR (2021)

10.6 DEMONSTRAÇÃO DE USO NA PLACA

Uma vez instalado, o Busybox possui a interface – informações quando executado diretamente – mostrada no comando abaixo. Esse foi simplificado para mostrar apenas o essencial. O programa retorna informações de definição, uso e comandos disponíveis (na atual versão).

```
$ busybox
BusyBox v1.32.1 (2021-01-29 22:10:24 -03) multi-call binary.
BusyBox is copyrighted by many authors between 1998-2015.
Licensed under GPLv2. See source distribution for detailed
copyright notices.

Usage: busybox [function [arguments]...]
   or: busybox --list[-full]
   or: busybox --show SCRIPT
   or: busybox --install [-s] [DIR]
   or: function [arguments]...

        BusyBox is a multi-call binary that combines many common Unix
        utilities into a single executable.  Most people will create
    ↪ a
        link to busybox for each function they wish to use and
    ↪ BusyBox
        will act like whatever it was invoked as.

Currently defined functions:
        [, [[, acpid, add-shell, addgroup, adduser, adjtimex, arch,
    ↪ arp,
        (...)
```

Conforme demonstrado anteriormente, o Busybox fornece acesso a diversos comandos (simplificados) diretamente, via links simbólicos. Como exemplo, vê-se a saída de um comando famoso, *cp* (cópia).

```
$ cp --help
BusyBox v1.32.1 (2021-01-29 22:10:24 -03) multi-call binary.

Usage: cp [OPTIONS] SOURCE... DEST

Copy SOURCE(s) to DEST

        -a      Same as -dpR
        -R,-r   Recurse
```

```
(...)
```

Para efeito de comparação, mostra-se abaixo, também, a sua versão completa, no *host*. Observa-se que, no caso do *target*, apenas o estritamente essencial é mostrado (e feito, pelo comando), demonstrando, assim, a otimização da ferramenta para sistemas embarcados.

```
$ cp --help
Uso: cp [OPÇÃO]... [-T] ORIGEM DESTINO
 ou: cp [OPÇÃO]... ORIGEM... DIRETÓRIO
 ou: cp [OPÇÃO]... -t DIRETÓRIO ORIGEM...
Copia ORIGEM para DESTINO, ou múltiplas ORIGENs para DIRETÓRIO.

Argumentos obrigatórios para opções longas também o são para opções
    ↪ curtas.
  -a, --archive                o mesmo que -dR --preserve=all
      --attributes-only        não copia os dados do arquivo, só seus
    ↪  atributos
(...)
```

Os links simbólicos são o que tornam possíveis a utilização do utilitário como ferramenta multiuso. Cada um deles funciona como um parâmetro ao programa principal. Por exemplo, o comando *busybox ifconfig* tem o mesmo efeito de *ifconfig*.

O Busybox também traz consigo outros dois programas muito importantes para um sistema Linux: o *shell* (*sh*) e o *init*. Esses são responsáveis por iniciar o console (entrada de dados) e iniciar os outros programas e serviços durante a inicialização do sistema, respectivamente. Essas funcionalidades que o Busybox apresenta serão abordadas com mais detalhes no capítulo 13.

11 CONFIGURAÇÃO DO KERNEL LINUX

O núcleo é a parte mais importante de um sistema operacional. O Linux, como um kernel, determina a qualidade de um dispositivo, visto que pode ser configurado de inúmeras formas, para várias aplicações. Sendo assim, este capítulo objetiva contextualizar o kernel Linux e a sua configuração, visando sistemas embarcados.

Dessa forma, este capítulo está organizado em: conceito do kernel Linux – tópico 11.1, funcionalidades – tópico 11.2, preparação para a compilação – tópico 11.3, personalização do kernel – tópico 11.4, compilação – tópico 11.5 e, por fim, instalação e algumas demonstrações - – tópico 11.6.

11.1 CONCEITO

O kernel Linux é um executável, baseado em UNIX, responsável por gerenciar recursos de hardware e garantir a correta utilização do mesmo por parte de todos os processos do sistema (TANENBAUM; BOS, 2016). Foi criado em 1991 por Linus Torvalds e, atualmente, corresponde a uma grande parte dos sistemas computacionais globais, com vários desenvolvedores ao redor do mundo (TANENBAUM; BOS, 2016).

Em um sistema Linux genérico, há três elementos fundamentais ao funcionamento, de acordo com Simmonds (2017): o sistema de arquivos, o interpretador de comandos e o núcleo. Esse último – o kernel – é a parte mais importante do sistema, pois é responsável por manter o sistema em todos os aspectos. Apesar disso, ele ainda precisa de bibliotecas e aplicações para prover recursos para os usuários finais – ou para cumprir sua função como sistema embarcado (BOOTLIN, 2021). Por esse motivo, o kernel não existe sozinho no sistema, mas vem acompanhado dos elementos citados.

Pelo fato de ser a camada entre o hardware e os programas, o modo como ele é construído afeta todos os aspectos do sistema final. Em sistemas embarcados, geralmente ele é configurado para se encaixar exatamente no hardware final, conforme os requisitos de cada projeto (SIMMONDS, 2017; YAGHMOUR et al., 2008).

O Linux foi criado como um *hobby* em 1991, por Linus Torvalds; na época, acadêmico, para usá-lo em um processador Intel 80386. Foi inspirado nos sistemas UNIX e no sistema operacional de Tanenbaum, Minix. O Linux difere deste de várias formas, sendo as principais: o suporte a memória virtual de 32 bits (devido ao processador alvo) e o fato de ser de código aberto (BOOTLIN, 2021; SIMMONDS, 2017).

É importante destacar que Linus desenvolveu apenas o núcleo de um sistema operacional e não um completo, com todos os componentes. Para resolver isso, foi necessário usar os

componentes do projeto GNU, especialmente os compiladores (*toolchain*), biblioteca C (*glibc*) e alguns comandos básicos.

Essa característica permanece até hoje, e torna o Linux extremamente portável dependendo do modo de como ele é utilizado: pode ser combinado com o ambiente GNU para formar distros desktop; pode ser combinado com o Android para formar sistemas *mobile* ou ainda pode ser combinado com um ambiente simples baseado em Busybox para criar sistemas embarcados (SIMMONDS, 2017). Essa última é a que será feita neste trabalho, mais precisamente, no capítulo 13 – Construção de um sistema de arquivos do zero.

O projeto foi bem aceito pela comunidade e rapidamente começou a ser usado como núcleo em sistemas operacionais gratuitos, recebendo contribuições individuais ou de grandes companhias (BOOTLIN, 2021). Desde então, o Linux amadureceu em um sistema completo, com robustez, confiabilidade e ótimos recursos que rivalizam com vários sistemas operacionais comerciais (HALLINAN, 2006). Prova disso é o fato de que, em 2019, 74,2% dos servidores web eram alimentados com sistemas Linux, conforme Netcraft (2019); além de ser grandemente aplicado em sistemas embarcados microprocessados (HALLINAN, 2006).

Em se tratando de sistemas embarcados, há diferenças na manipulação do kernel em relação a sistemas desktop e servidores, por exemplo. Nesses últimos, é comum que os desenvolvedores incluam suporte a todo tipo de hardware, ao custo de certa generalização e possivelmente, performance. Por outro lado, em sistemas embarcados há a preocupação em manter o kernel o mais simples possível, devido a maior facilidade para depuração e o processo de desenvolvimento menos custoso, já que criar do zero um sistema embarcado já é difícil por si só (YAGHMOUR et al., 2008).

11.2 FUNCIONALIDADES

Conforme mencionado antes, o kernel Linux é o elemento mais importante de um sistema computacional, pois é responsável por controlá-lo completamente. O kernel é rico em recursos (para todo tipo de aplicação), desempenha várias funções em um sistema e é bastante complexo internamente. Essas atribuições serão detalhadas a seguir.

Em relação a recursos, os mais destacados – em relação a sistemas embarcados – conforme Bootlin (2021), são: portabilidade, ou seja, roda na maioria das arquiteturas; escalabilidade, isto é, funciona desde em supercomputadores até pequenos dispositivos (com no mínimo 4 MB de RAM); possui extensa adequação à padrões – como os do IEEE – e é modulável, ou seja, permite a inclusão apenas do que é necessário.

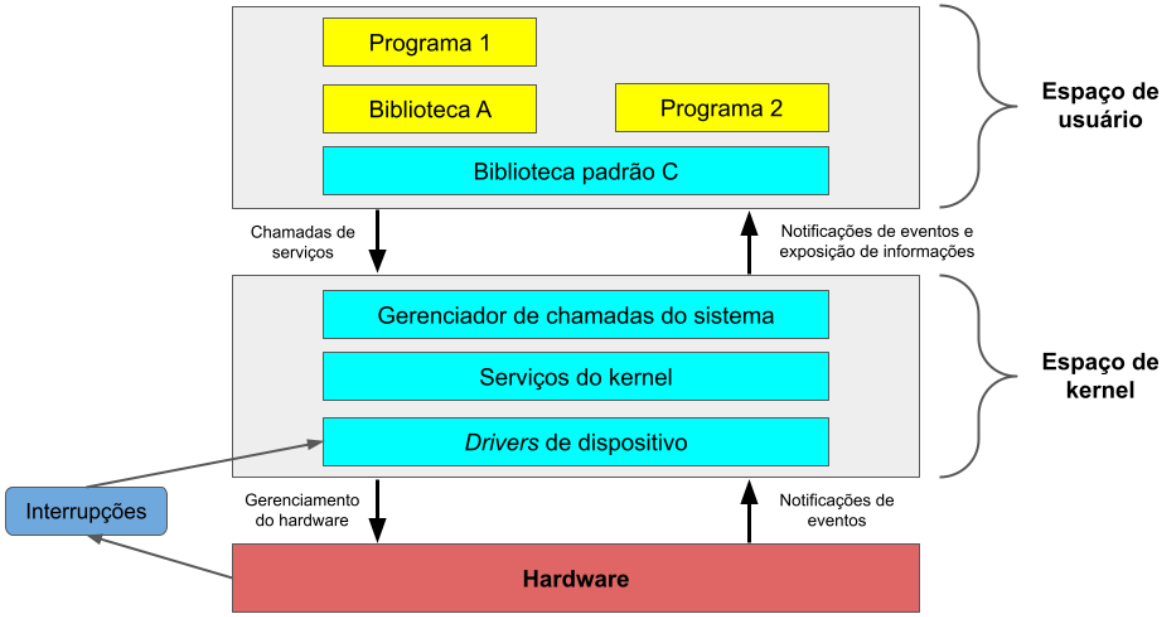
Quanto às funções, o kernel Linux possui diversas dentro de um sistema operacional. Conforme Bootlin (2021), Yaghmour et al. (2008), Simmonds (2017), são três as principais: gerenciar todos os recursos do hardware, gerenciar o acesso das aplicações a ele e fornecer uma API para programas no espaço de usuário. Aquela se refere ao controle de recursos como

CPU, memória e dispositivos de entrada e saída; essa tem a ver com a "multiplexação" de um determinado recurso para várias aplicações – como a interface de rede, por exemplo – e esta diz respeito ao modo pelo qual as aplicações terão acesso ao hardware – que é por bibliotecas compartilhadas e *system calls*.

Por fim, em relação ao seu aspecto interno, o kernel é bastante heterogêneo, sendo construído de diversos subsistemas (kernel services). De acordo com Bootlin (2021), o kernel possui, dentro de si: um gerenciador de memória, um gerenciador de tarefas (ou processos), sistemas de arquivos virtuais, *drivers* de dispositivo, pilhas de protocolos de rede e códigos de baixo nível específicos de cada arquitetura. Em algumas arquiteturas (inclusive a ARM), existe ainda uma especificação da estrutura do hardware na memória durante o processo de inicialização, conhecida como *device tree* (BOOTLIN, 2021; SIMMONDS, 2017).

Além do aspecto de gerenciamento de recursos citado acima, o kernel Linux tem uma função muito importante como intermediário entre aplicações e o hardware. O kernel serve como uma grande interface de aplicações (API), lidando com todas as requisições e escalando as suas prioridades. Abaixo, na figura 30, é mostrado um diagrama esquemático do comportamento do kernel em relação as aplicações.

Figura 30 – Diagrama de blocos do funcionamento de um sistema Linux



Adaptado de Bootlin (2021), Simmonds (2017)

Basicamente, o Linux separa as aplicações em execução em duas classes: as que estão no espaço de usuário e as que estão no espaço de kernel. Espaço de usuário corresponde as aplicações normais (executadas por usuários do sistema, incluindo o *root*), enquanto que o espaço de kernel diz respeito ao kernel em si, além de módulos por ele invocados (geralmente *drivers*

de dispositivo) (SIMMONDS, 2017). Aplicações no espaço de usuário rodam com privilégios baixos de CPU, e precisam de bibliotecas compartilhadas para acessar o espaço de kernel e, assim, o hardware, conforme apresentado na figura 30. O modo como o acesso é feito é por meio de *system calls* (SIMMONDS, 2017).

A camada entre o espaço de usuário e o espaço de kernel é a biblioteca padrão C (*glibc*), a qual traduz rotinas em nível de usuário em chamadas do sistema, *system calls* (BOOTLIN, 2021; SIMMONDS, 2017). *System calls* são funções que permitem acesso direto ao kernel e, consequentemente, ao hardware (KERRISK, 2010). Essas *system calls* são específicas de cada arquitetura, e são gerenciadas pelo gerenciador de chamadas de sistema (*system call handler*), que associa cada chamada ao respectivo subsistema do kernel: chamadas de alocação de memória vão para o gerenciador de memória, chamadas de sistemas de arquivos vão para o gerenciador de sistemas de arquivos, e assim por diante (BOOTLIN, 2021; SIMMONDS, 2017).

Além disso, existem algumas chamadas que exigem um *input* e que serão transferidas a um *driver* de dispositivo, e o próprio hardware pode invocar uma chamada por meio de uma interrupção, que só são gerenciadas pelos *drivers* de dispositivo (no espaço de kernel) e nunca em espaço de usuário (SIMMONDS, 2017). Portanto, a figura 30 representa, em resumo, que tudo aquilo que uma aplicação faz, ela faz mediante o kernel, via *system calls*.

A última das capacidades do kernel, porém não menos importante, é a configurabilidade do kernel, isto é, de escolher quais funcionalidades estarão no executável final durante a fase de compilação (YAGHMOUR et al., 2008).

A etapa de configuração do kernel Linux – que será mostrada neste capítulo – permite que o projetista remova ou adicione suporte a recursos no kernel do seu sistema embarcado. Por exemplo, pode-se remover suporte a diversos sistemas de arquivos que não serão usados, bem como vários protocolos de comunicação em rede, caso o dispositivo embarcado não possua suporte a esses recursos. Por outro lado, é possível adicionar suporte a um periférico (ou protocolo) em específico para atender a demanda do sistema embarcado apenas (YAGHMOUR et al., 2008).

Dependendo da necessidade, muitas funcionalidades podem ser compiladas na forma de módulos dinâmicos, para serem carregadas apenas quando requeridas (YAGHMOUR et al., 2008). A configuração se faz mediante uma interface gráfica, como o *menuconfig*, e é dividida em várias seções, devido a sua complexidade. Essa etapa de configuração via *menuconfig*, bem como as especificidades das seções mais importantes, serão tratadas posteriormente neste capítulo.

11.3 PREPARAÇÃO PARA A COMPILAÇÃO

Antes de iniciar a compilação, é necessário preparar o ambiente para tal, isto é, preparar o código-fonte. Existem, basicamente, duas possibilidades de obtenção do código-fonte do kernel Linux: o SDK e o repositório oficial na internet.

A Texas Instruments entende a necessidade do desenvolvedor de Linux embarcado em precisar adaptar o kernel à sua necessidade, conforme dito na sua documentação (TI, 2019). Devido a isso, o SDK já vem com o código-fonte disponível, pronto para configuração, além de disponibilizar extensa documentação sobre a adaptação desse kernel em seus produtos (*SoCs*). Portanto, neste trabalho será usado o código-fonte do disponibilizado pelo SDK. Na versão do SDK usada neste trabalho (06_00_00_07) a versão do kernel disponível é a 4.19.38.

O código-fonte do kernel Linux está no mesmo lugar em que estava o do U-Boot – na pasta *board-support* do SDK. O procedimento de preparação para a compilação, nesse caso, é apenas clonar a pasta com o código-fonte, já que não foi preciso baixá-lo nem extraí-lo dessa vez. Além disso, deve-se fazer uma pasta de saída – no mesmo diretório – para o resultado da compilação, cujo nome é *kernel_build*.

```
# Clonando a pasta
$ cd ~/texasSDK/board-support
$ cp -r linux-4.19.38+gitAUTOINC+4dae378bbe-g4dae378bbe linux
    ↪ -4.19.38-RASCUNHO

# Criando a pasta de saída
$ mkdir kernel_build
```

11.4 PERSONALIZAÇÃO

O procedimento de compilação desse pacote é quase idêntico aos anteriores. Os passos iniciais são os mesmos: exportam-se as variáveis de ambiente que definem a arquitetura, define-se o caminho de saída, bem como o caminho do código-fonte. Por último, abre-se a pasta desse último para configuração.

```
# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/kernel_build"

# definir o caminho do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/linux-4.19.38-
    ↪ RASCUNHO"

## INÍCIO ##
# abrir a pasta do código-fonte do kernel Linux
cd $SOURCE
```

Como o kernel Linux é gigantesco (em tamanho e opções), é muito difícil, para o desenvolvedor, ter de configurar cada opção manualmente, definindo aquilo que é essencial ou não. Por isso, o SDK da Texas Instruments cede um arquivo de configuração padrão para cada uma de suas famílias de processadores, já otimizado para funcionar corretamente nas suas plataformas.

A regra da Texas Instruments para encontrar a configuração padrão de cada plataforma é observar o número da família de processadores, nesse caso, a *am335x*. Portanto, o nome da configuração é *tisdk_am335x-evm_defconfig*, conforme se observa no console abaixo.

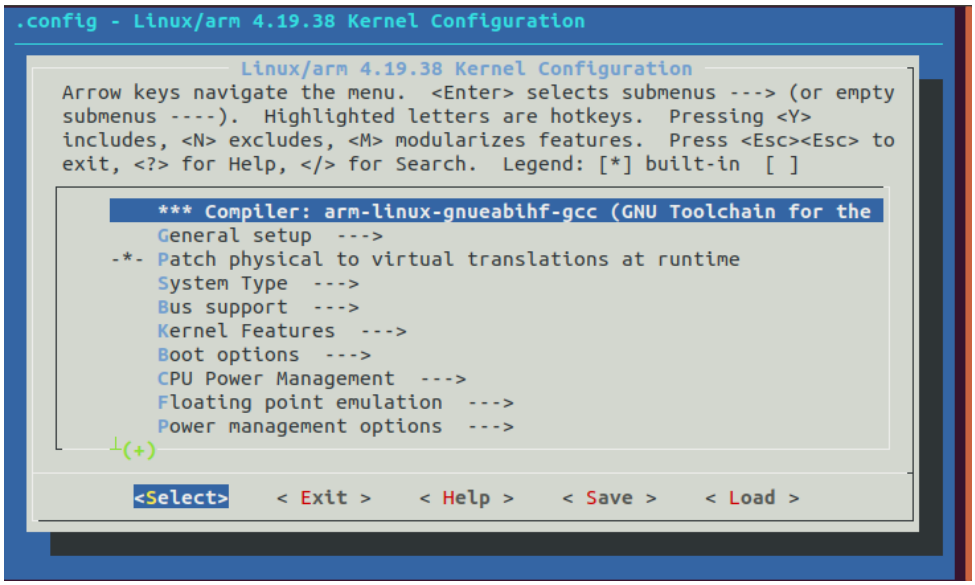
```
1 # selecionar a configuração padrão da BeagleBone Black
2 make tisdk_am335x-evm_defconfig
```

É valido ressaltar que, diferente da compilação anterior (Busybox), a seleção de uma configuração padrão é essencial, já que, diferentemente do Busybox, são milhares as opções de configuração disponíveis no kernel Linux – quase 2000 opções diferentes (LINUXTOPIA, 2010). Além disso, há a vantagem de usar uma versão funcional do kernel, sendo necessária apenas a remoção (ou adição) de recursos conforme a necessidade do sistema embarcado, acelerando, assim, o tempo de desenvolvimento (TI, 2019).

Por fim, pode-se abrir a interface de configuração baseada em *ncurses*, *menuconfig*, para a configuração do kernel, isto é, as funcionalidades que serão compiladas no executável final, além dos módulos, quando necessário. A tela de configuração é mostrada na figura 31.

```
1 $ make menuconfig
```

Figura 31 – Tela de configuração (*menuconfig*) do kernel Linux



Fonte: AUTOR (2021)

Uma das grandes vantagens da utilização de Linux em um sistema embarcado microprocessado é a capacidade de adaptação do kernel às necessidades do projeto, isto é, o kernel dispõe de recursos para adicionar ou retirar suporte de diversos tipos de interfaces, protocolos e recursos (YAGHMOUR et al., 2008). Isso é útil na medida em que o sistema embarcado a ser desenvolvido terá exatamente as funcionalidades que lhe for requerido, tornando-o mais eficiente para a sua tarefa e até mesmo mais seguro, pois determinados recursos não usados podem se tornar brechas de segurança (YAGHMOUR et al., 2008).

Exemplos disso são sistemas embarcados voltados para a aquisição de sinais (em ambientes industriais, em geral). Às vezes, é preciso apenas que o dispositivo acesse a rede para enviar dados para algum servidor remoto, por exemplo. Portanto, não é interessante deixar a maior parte dos protocolos de rede ativos, como o de e-mail, por exemplo (YAGHMOUR et al., 2008).

Tendo isso em vista, a etapa de customização do kernel é dividida em seções, visto que é extensa e complexa. Na versão selecionada para este trabalho, algumas das disponíveis na configuração do kernel são:

- ***General Setup***: Configurações genéricas, como controle de versões e suporte a alguns recursos comuns a qualquer arquitetura (como o *swap* e o modo de descompressão do kernel);

- ***System Type***: Configurações específicas de cada arquitetura de processadores (ARM, neste caso) e família de processadores (Sitara *am335x*, neste caso). Em geral, as opções dessa seção já estão otimizadas para funcionar na plataforma alvo e não devem ser alteradas (salvo casos muito específicos);

- ***Kernel Features***: Recursos do kernel mais próximos do hardware, como multiprocessamento, frequência de *timers*, modo de comunicação com a memória, entre outros. Geralmente essas configurações são editadas apenas em casos muito específicos.

- ***Boot options***: Configurações relativas a passagem do sistema do inicializador para o kernel, principalmente em relação a *device tree*;

- ***CPU Power Management***: Configurações acerca do uso da CPU, naquilo que impacta diretamente no seu consumo de energia, que é a frequência (quando a CPU permite alteração disso em tempo de execução) e o gerenciamento de processos (também chamados de *idle drivers*). Projetistas de sistemas embarcados geralmente fazem uso dessa seção;

- ***Power management options***: Configurações de energia dos periféricos, como a RAM e os dispositivos em geral, e não exclusivas da CPU. Exemplos são suporte a hibernação e *standby*. Essa é outra seção de interesse de desenvolvedores embarcados Linux;

- ***Firmware Drivers***: *Drivers* de dispositivo específicos da arquitetura escolhida (ARM, neste caso);

- ***Memory Management options***: Configurações específicas da ligação entre a memória RAM e o kernel, como alocação e compactação.

- ***Networking support:*** Todas as configurações e protocolos relativos a comunicação entre computadores ou dispositivos (e não somente o Ethernet) estão aqui. Levando em consideração o mundo cada vez mais conectado entre máquinas (IoT), não conceder suporte a rede é praticamente impensável por parte dos desenvolvedores. Ainda assim, é interessante remover todos os protocolos e funcionalidades que são serão usados.

- ***Device drivers***: Todos os tipos de *drivers* de dispositivo disponíveis na versão utilizada no kernel Linux estão aqui. O Linux precisa de *drivers* para toda e qualquer comunicação com o hardware, sejam barramentos, protocolos (SPI e I2C, por exemplo), redes, *clocks* e muitos outros. A maior parte das configurações disponíveis está nessa seção – mais de 60%, segundo Bootlin (2021) – e é aqui também que os desenvolvedores gastam a maior parte do tempo de configuração , definindo todos os recursos que devem ser suportados pelo kernel.

Pode-se perceber que praticamente tudo em um sistema embarcado Linux é configurável, tornando-o ideal para aplicações de pequeno porte – *smartwatches*, por exemplo – a grande porte – sistemas industriais de controle, por exemplo.

Nesse sentido, optou-se por demonstrar, neste trabalho, duas funcionalidades do kernel apenas, já que suas opções são muito extensas e algumas são complexas, e exigem profundo conhecimento de Linux e do hardware. Será mostrada a ausência de dois recursos do kernel: suporte a redes e suporte aos LEDs *onboard*, visto que são fáceis de configurar e demonstrar.

Para desativar o suporte aos protocolos de redes, é preciso desmarcar a opção *Networking support*, na aba geral e, para desativar o suporte aos LEDs, deve-se desmarcar a opção *Leds support*, na seção *Device drivers*.

11.5 COMPILAÇÃO

Feita a personalização, pode-se compilar o kernel Linux. O make, nesse caso, permite que as compilações possam ser feitas separadamente, conforme Yaghmour et al. (2008) e a documentação do SDK (TI, 2019). Assim, apenas o executável do kernel, o zImage, será compilado com o comando abaixo.

```
1 $ make -j9 zImage
```

A compilação demora um pouco, dependendo do número de opções ativadas durante a configuração. Esse executável é o kernel em si, que será carregado para a memória RAM pelo inicializador – U-Boot – no final de sua etapa.

O próximo passo é a compilação do arquivos de árvore de dispositivos. Iniciando com o kernel 3.8, cada placa da Texas Instruments e, consequentemente, cada um dos seus processadores, passou a possuir um arquivo binário de árvore de dispositivos (*device tree binary file*), que é requerido pelo kernel (TI, 2019). Dessa forma, cada placa de desenvolvimento possui o arquivo *.dtb* correto necessário para a sua inicialização. No caso da BeagleBone Black, esse arquivo é compilado com o seguinte comando mostrado abaixo.

```
1 $ make -j9 am335x-boneblack.dtb
```

É importante ressaltar que, caso se esteja em uma situação de desenvolvimento de um dispositivo baseado na placa BeagleBone Black, a configuração das placas podem ser diferentes, por exemplo, pode ser que algumas interfaces (como HDMI e USB) sejam removidas e usadas para outros fins. Nesse caso, a estrutura de dispositivos será diferente da BeagleBone Black, pois as suas conexões entre o processador *am335x* e as interfaces mudam. Portanto, seria necessário a produção de um novo *device tree file* com o auxílio do software *PinMux* (TI, 2019; COLEY, 2014).

A última etapa na compilação do kernel Linux são os módulos. Conforme a documentação do SDK (TI, 2019) e a documentação do kernel (KERNEL, 2016), os módulos são pedaços de código, geralmente *drivers*, que não são compilados junto com o executável do kernel. Isso dá a eles a possibilidade de serem carregados para o kernel em tempo de execução. Para compilá-los, basta executar o comando abaixo. O resultado são arquivos *.ko* (*kernel object*), que são os módulos dinâmicos do kernel.

```
1 $ make -j9 modules
```

11.6 INSTALAÇÃO E DEMONSTRAÇÃO DE USO NA PLACA

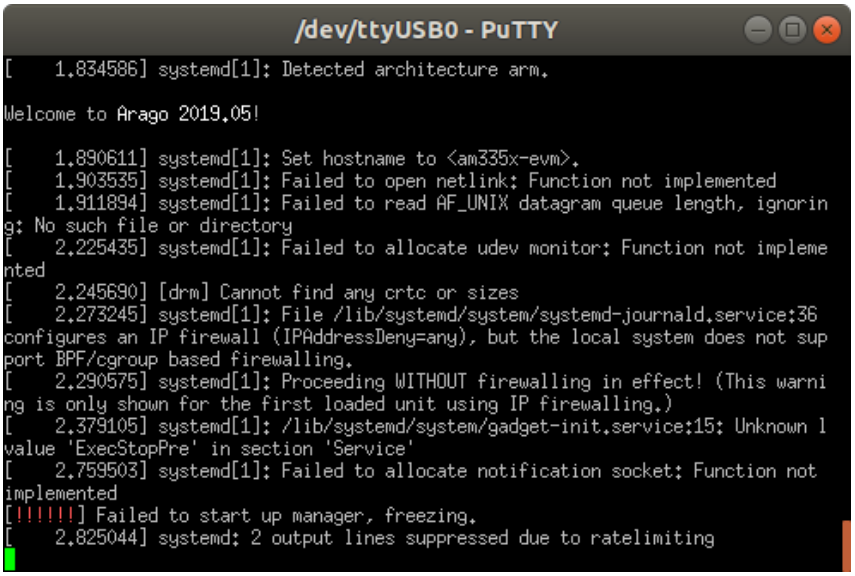
A instalação do Linux no *target* é feita por partes: deve-se transferir o executável do kernel, o arquivo de *device tree* e os módulos. Abaixo é mostrado o comando de instalação (ou transferência) de cada um deles para a pasta de saída *kernel_build* (TI, 2019). Vale destacar que o comando pode ser alterado para a instalação diretamente no *target*, porém, essa medida não foi tomada pois foi necessário fazer alguns ajustes, conforme será descrito a seguir.

```
# instalar kernel e device tree
$ cp arch/arm/boot/zImage arch/arm/boot/dts/am335x-boneblack.dtb
    ↪ $RAIZ

# instalar modulos
$ make INSTALL_MOD_PATH=$RAIZ modules_install
```

Depois desse passo, só restaria copiar esses arquivos para o cartão micro SD do *target*, porém, é necessário fazer algumas alterações no sistema de arquivos do *target*. Devido ao fato de o suporte a interface de redes ter sido completamente removido, muitos *daemons* não conseguiram iniciar utilizando o sistema de arquivos *rootfs* do SDK (o mais completo), logo, a inicialização foi interrompida, conforme se observa na figura 32.

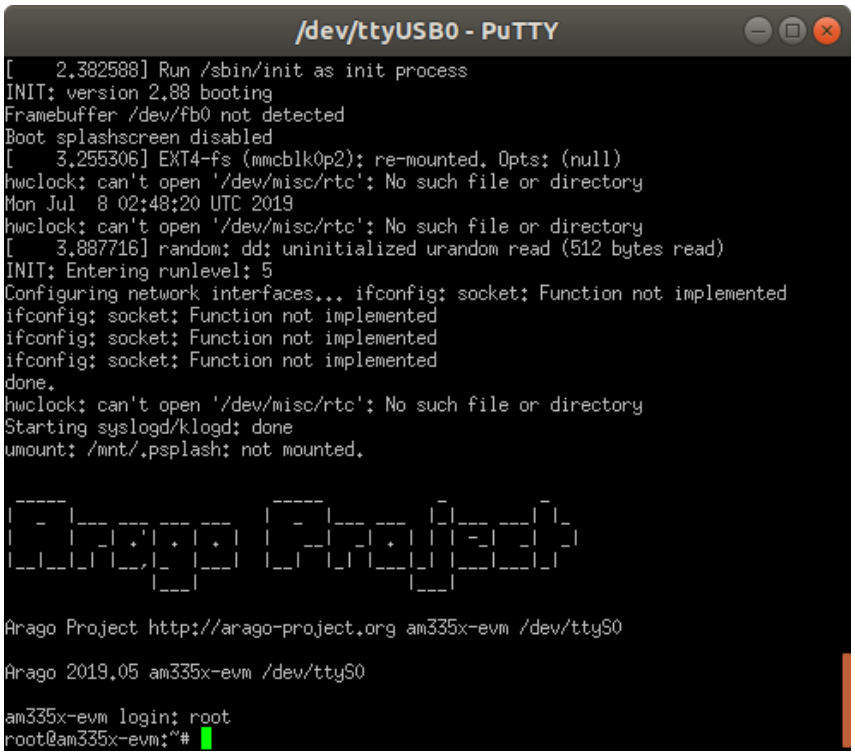
Figura 32 – Inicialização quebrada utilizando o sistema de arquivos *rootfs*



Fonte: AUTOR (2021)

Por esse motivo, foi necessário substituir o sistema de arquivos *rootfs* pelo *tiny*, o menor de todos – e com menos *daemons* – para que fosse possível chegar a tela de login e, assim, o sistema pudesse ser demonstrado. Tendo o novo sistema de arquivos já no cartão de memória, os arquivos podem ser transferidos: os módulos ficam na pasta */lib*, e o kernel e o arquivo de *device tree*, na pasta */boot*. A inicialização do sistema utilizando o kernel modificado pode ser vista na figura 33.

Figura 33 – Inicialização funcional utilizando o sistema de arquivos *tiny*



Fonte: AUTOR (2021)

Com o kernel instalado e o sistema de arquivos funcional, é possível demonstrar as consequências da remoção dos recursos do kernel citados anteriormente. Nesse kernel personalizado, duas funcionalidades tiveram seu suporte removido: as interfaces de rede – primeiro exemplo – e o acesso aos LEDs *onboard* – segundo exemplo.

A opção de interfaces de rede diz respeito a todos os protocolos de comunicação disponíveis em uma dada versão no kernel, sendo os da camada TCP/IP os mais conhecidos. Isso torna fácil a demonstração da falta desse recurso, pois qualquer comando voltado para redes de computadores (como o *ifconfig*) não funcionará, já que usa uma chamada de sistema (*system call*) específica de redes, que fora removida.

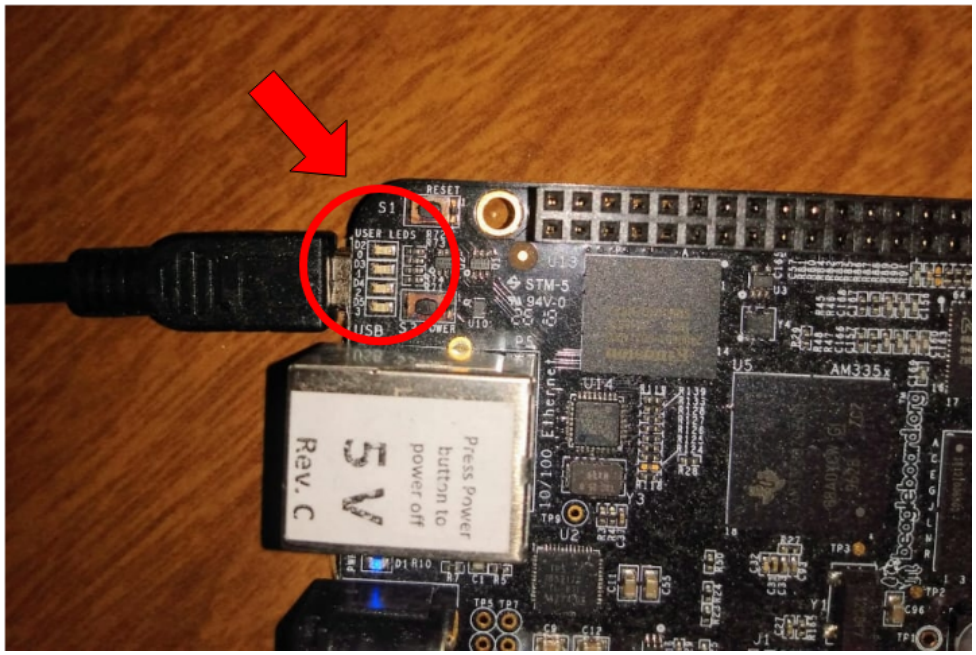
```
1 $ ifconfig
2 ifconfig: /proc/net/dev: No such file or directory
3 ifconfig: socket: function not implemented
```

Como se pode observar no console acima, o comando de configuração de rede, *ifconfig*, não funcionou pois a principal função do sistema operacional que ele usa para funcionar, *socket*, não foi implementada.

O segundo exemplo da ação do kernel no produto final é o suporte aos LEDs *onboard*. O kernel possui alguns recursos para identificar, no hardware, se há LEDs na placa-mãe. Esses LEDs geralmente indicam o estado da CPU e o uso do disco, além de servir de indicativo de que

a placa (ou o computador) está ligado. Como esse suporte foi removido do kernel, os LEDs na placa ficam desligados, conforme é demonstrado na imagem 34.

Figura 34 – LEDs *onboard* apagados



Fonte: AUTOR (2021)

Remover ou adicionar suporte a protocolos e dispositivos que não serão usados são uma ótima prática no desenvolvimento de sistemas embarcados Linux pois, além de poupar recursos, são uma grande barreira defensiva do sistema, já que, em caso de algum ataque hacker ou invasão do sistema, nada poderá ser feito além do que o kernel foi programado para fazer, conforme se observa no primeiro exemplo.

O *script* completo da configuração realizada nesse capítulo é mostrado abaixo, na figura 35.

Figura 35 – *script* de compilação e personalização do kernel Linux

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/kernel_build"

# definir o caminho do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/linux-4.19.38-RASCUNHO"

## INÍCIO ##

# abrir a pasta do código-fonte do kernel Linux
cd $SOURCE

# limpar compilações anteriores
make distclean

# selecionar a configuração padrão da BeagleBone Black
make tisdk_am335x-evm_defconfig

# personalizar
make menuconfig

# compilar APENAS o ELF
make -j9 zImage

# compilar device tree
make -j9 am335x-boneblack.dtb

# compilar modulos
make -j9 modules

# instalar kernel e device tree
cp arch/arm/boot/zImage arch/arm/boot/dts/am335x-boneblack.dtb $RAIZ

# instalar modulos
make INSTALL_MOD_PATH=$RAIZ modules_install
```

Fonte: AUTOR (2021)

Parte IV

Construção do sistema de arquivos

12 CONCEITOS IMPORTANTES

Os sistemas operacionais precisam de um sistema de arquivos para suas aplicações. Nesse sentido, os dispositivos baseados em Linux também precisam embarcá-los. No entanto, a configuração de um sistema de arquivos é complexa, e envolve muitos conceitos. Portanto, o objetivo deste capítulo é lançar as bases teóricas para a construção do sistema de arquivos, no capítulo 13.

Logo, este capítulo está estruturado da seguinte forma: conceito de sistema de arquivos – tópico 12.1, importância do sistema de arquivos – tópico 12.2, hierarquia do sistema de arquivos Linux – tópico 12.3 e itens obrigatórios nesse – tópico 12.4.

12.1 O QUE É O SISTEMA DE ARQUIVOS

As aplicações que rodam em um computador precisam armazenar e recuperar informações; geralmente a longo prazo. Nesse caso, há os requisitos de que essas informações precisem ser permanentes, muitos processos devem ser capazes de acessá-las ao mesmo tempo e deve ser possível armazenar uma grande quantidade delas. Além disso, é preciso que haja um controle de como esses dados são escritos nos discos, isto é, onde começam e onde terminam. Para resolver esse problema, os sistemas operacionais usam abstrações para o armazenamento: o conceito de arquivos e sistemas de arquivos (TANENBAUM; BOS, 2016).

O sistema de arquivos é uma forma de organização de um grande volume de dados em algum dispositivo de massa com vista a recuperação posterior, possibilitando ao sistema operacional ler ou gravar os dados sem dificuldade (TANENBAUM; BOS, 2016). O sistema de arquivos é um dos três elementos essenciais de um sistema operacional (junto do kernel e de um interpretador de comandos) e o modo como ele é desenvolvido impacta diretamente no desempenho do sistema embarcado final (SIMMONDS, 2017; YAGHMOUR et al., 2008).

Existem vários tipos diferentes de sistemas de arquivos. Cada um com sua estrutura e lógica particulares, além de velocidade, segurança e entre muitas outras características. Além disso, existem sistemas de arquivos para dispositivos locais, para montagem remota via rede (como o NFS) e até virtuais, como o *procfs* e o *sysfs*, que são comuns no Linux (TANENBAUM; BOS, 2016). Exemplos bastante conhecidos de sistemas de arquivos são o FAT32, NTFS e o ext4.

É comum que os sistemas operacionais suportem mais de um sistema de arquivos, para questões de compatibilidade. Sistemas baseados em UNIX utilizam um sistema de arquivos virtual, ou seja, todos os dispositivos e arquivos parecem existir em uma única hierarquia global de arquivos e pastas – representada por / (barra) ou *root*, no caso do Linux – a qual pode ser composta de diversos sistemas de arquivos (BOOTLIN, 2021).

Um último conceito importante em relação a sistemas de arquivos Linux é: tudo é um arquivo. Isso significa que todos os dispositivos e outros recursos atrelados ao sistema são apresentados como (e possuem comportamento de) arquivos (TANENBAUM; BOS, 2016). Esse conceito é importante para diversas aplicações práticas, e com os sistemas embarcados não é diferente. Os maiores exemplos são os conteúdos dos diretórios */dev*, */proc* e */sys*, que serão abordados com mais detalhes ainda nesse capítulo.

12.2 IMPORTÂNCIA DO SISTEMA DE ARQUIVOS

O sistema de arquivos em um sistema operacional corresponde a um dos organismos mais importantes para o correto funcionamento do todo e, com os sistemas baseados em Linux, não é diferente. A importância do sistema de arquivos se desdobra em várias, tais como: manter a integridade dos dados, ceder um melhor ambiente para as aplicações (e os desenvolvedores delas) e fornecer uma interface para os usuários.

Primeiramente, sem um sistema de arquivos, a integridade dos dados estaria seriamente comprometida, isto é, as informações salvas em um dispositivo de armazenamento seriam um grande corpo de dados, sem informações de onde começam e terminam os dados. Uma das funções do sistema de arquivos é separar os dados em pedaços e dar nomes a cada peça, abstração conhecida como arquivo (TANENBAUM; BOS, 2016).

Ademais, os sistemas Linux, assim como qualquer outro sistema operacional avançado, precisa de uma estrutura de diretórios raiz (*root*) para realizar a maior parte de suas funções. Isso ocorre principalmente pelo fato de que as aplicações que rodam em um sistema Linux – e que dão a ele a sua fama – ficam guardadas no sistema de arquivos, e dele precisam durante a sua execução (HALLINAN, 2006). Por exemplo, as suas aplicações esperam que o sistema de arquivos raiz contenha programas e utilitários para inicializar um sistema; inicializar serviços, como rede e um console do sistema; carregar *drivers* de dispositivos e montar sistemas de arquivos adicionais (HALLINAN, 2006).

De modo estrito, o kernel Linux em si não exige um sistema de arquivos, mas as aplicações e os padrões de desenvolvimento de software, sim. Os programas de espaço do usuário esperam encontrar arquivos com nomes específicos em estruturas de diretório específicas (YAGHMOUR et al., 2008). Tal padrão é conhecido como Padrão de Hierarquia do Sistema de Arquivos (*Filesystem Hierarchy Standard*), e dita normas para a formatação do sistema de arquivos raiz do Linux. Tal conceito será detalhado na próxima seção, 12.3.

Por fim, o último objetivo de um sistema de arquivos é ser receptáculo de uma interface de usuário. Em sistemas desktop, é comum o uso de interfaces gráficas como GNOME, LXDE, entre outras, mas, em um sistema embarcado, muitas vezes apenas o *shell* é habilitado. Por esse motivo, é comum que o sistema de arquivos seja diminuto, contendo apenas o estritamente essencial, além de questões de eficiência e segurança (SIMMONDS, 2017). A construção do

sistema de arquivos será abordada no capítulo 13, e alguns exemplos de interfaces de usuário adicionais (*shell* seguro e interface web) serão tratadas na parte V.

12.3 HIERARQUIA DO SISTEMA DE ARQUIVOS LINUX

Conforme comentado no capítulo anterior, o kernel Linux possui a capacidade de se adaptar a qualquer ambiente, dependendo das ferramentas ao seu redor. Como exemplo, os sistemas de arquivos de uma distro Linux e um smartphone Android são quase completamente diferentes. Isso ocorre pois o kernel não se preocupa com a estrutura de diretórios do sistema no qual ele há de rodar, exceto para o programa *init* (SIMMONDS, 2017).

No entanto, muitos programas esperam encontrar uma estrutura pré-definida, com certos arquivos em certos lugares. Além disso, os desenvolvedores dessas aplicações esperam encontrar uma estrutura similar entre os dispositivos que rodam Linux. Por isso, existe um padrão que rege os sistemas Linux chamado Padrão de Hierarquia do Sistema de Arquivos (*Filesystem Hierarchy Standard*, FHS), definido por vários desenvolvedores do kernel. Esse padrão rege a organização da estrutura raiz dos diretórios, desde pequenos até grandes sistemas e estabelece uma linha mínima de compatibilidade entre kernel e aplicações (SIMMONDS, 2017; HALLINAN, 2006).

O Padrão de Hierarquia do Sistema de Arquivos é um documento que determina a localização de arquivos, pastas e programas instalados no sistema. Ele faz isso especificando princípios e arquivos essenciais para cada área do sistema de arquivos, a partir de casos nos quais se costumava haver conflito. Esse documento é usado principalmente por criadores de distribuições de sistemas operacionais (no caso deste trabalho, embarcadas), mas também pode ser usada por desenvolvedores que querem garantir compatibilidade nos seus produtos (RUSSELL; QUINLAN; YEOH, 2004).

Conforme Russell, Quinlan e Yeoh (2004), o propósito do sistema de arquivos raiz é adequar os seus conteúdos para garantir a inicialização, a restauração, a recuperação e/ou reparo do sistema. A inicialização diz respeito aos serviços (*daemons*), e o suficiente deve estar na raiz nessa etapa. Para a restauração, as ferramentas necessárias para tal devem estar presentes no sistema de arquivos. Por último, para a recuperação e/ou reparo, ferramentas de diagnóstico e reparação do sistema devem estar inclusas. Tudo isso para garantir que o sistema fique tão pequeno quanto possível (RUSSELL; QUINLAN; YEOH, 2004).

Desse modo, os diretórios requeridos na raiz do sistema de arquivos Linux – conforme Russell, Quinlan e Yeoh (2004), assim como suas descrições – conforme Elsner e Lingnau (2015) – são listadas abaixo:

- ***bin***: Comandos utilitários essenciais. Nessa pasta estão os mais importantes comandos (a maioria, de sistema) necessários para: A inicialização (*mount* e *mkdir*, por exemplo), a execução (*ls* e *grep*, por exemplo) e também reparação do sistema. Essa pasta contém

comandos que podem ser usados tanto pelo administrador do sistema quanto pelos usuários, além de serem usados indiretamente por *scripts*;

- ***boot***: Arquivos estáticos do inicializador. Nesse diretório está o kernel Linux em si (vmlinuz), além de outros arquivos requeridos pelo inicializador. No caso deste trabalho, o U-Boot requer apenas o arquivo de device tree;
- ***dev***: Arquivos de dispositivo. Nessa pasta estão os arquivos que funcionam como uma interface entre o *shell* e os *drivers* de dispositivo dentro do kernel. Esses arquivos não são "reais", isto é, são apenas uma representação de um *driver* existente no kernel, baseado na regra de que "tudo é um arquivo" no Linux;
- ***etc***: Arquivos de configuração do sistema. Esse diretório contém os arquivos de configuração da maioria dos programas, assim como do próprio sistema. Além disso, esse diretório possui a sua própria hierarquia, que varia de sistema para sistema.
- ***home***: Diretórios dos usuários comuns. Esse diretório contém os diretórios home de todos os usuários no sistema, exceto o *root*. Embora seja um padrão nos sistemas Linux, sua hierarquia varia conforme a aplicação.
- ***lib***: Bibliotecas compartilhadas essenciais e módulos do kernel. Contém bibliotecas compartilhadas necessárias para iniciar o sistema e executar aplicações. Tais bibliotecas poupam muitos recursos, visto que vários processos usam as mesmas partes básicas, além de que torna mais fácil a depuração. Também ficam nesse local os módulos do kernel – *drivers* de dispositivo, sistemas de arquivos e protocolos de rede.
- ***media***: Ponto de montagem para mídia removível. Este diretório contém subdiretórios que são usados como pontos de montagem para mídia removível, como pendrives, cartões de memória e afins.
- ***mnt***: Ponto de montagem para sistemas de arquivos temporários. Este diretório é fornecido para que o administrador do sistema possa montar temporariamente um sistema de arquivos conforme necessário.
- ***opt***: Pacotes de software de aplicativos complementares. Este diretório é destinado a software de terceiros – pacotes completos preparados por fornecedores que devem ser instaláveis sem entrar em conflito com arquivos de distribuição ou arquivos instalados localmente.
- ***root***: Diretório *home* do usuário *root*. É uma pasta normal e funciona do mesmo modo que as outras da *home*. Pode ser determinado pelo desenvolvedor ou preferência local, mas este é o local padrão recomendado.
- ***sbin***: Binários essenciais do sistema. São utilitários usados para a administração do sistema, assim como os da pasta bin, porém, são voltados para uso exclusivo do usuário *root*.

- ***srv***: Dados para serviços fornecidos pelo sistema. O objetivo principal de especificar isso é para que os usuários possam encontrar a localização dos arquivos de um serviço específico e para que os serviços que requerem uma única árvore para dados possam ser separados do resto do sistema.

- ***tmp***: Arquivos temporários. Esse diretório deve existir para as aplicações que requerem arquivos temporários. As distribuições costumam usar um serviço para limpar esse diretório em cada inicialização. Os programas não devem presumir que quaisquer arquivos ou diretórios nesse local sejam preservados entre as invocações do programa.

- ***usr***: Hierarquia secundária. Esse diretório funciona como uma hierarquia secundária do sistema, só que contendo software não essencial, geralmente executáveis e bibliotecas. Além disso, essa seção é feita para ser compartilhável e apenas de leitura.

- ***var***: Dados variáveis. Esse diretório contém arquivos dinâmicos, isto é, que podem ser alterados por programas em execução. Alguns serviços, como o *syslogd*, usam essa pasta. Essa é mais uma pasta que possui a própria hierarquia.

Existem ainda dois diretórios que, embora não sejam oficiais – segundo o padrão – são largamente utilizados em sistemas Linux. Esses são o *proc* e o *sys*, e são usados como receptáculos para sistemas de arquivos virtuais.

A pasta *proc*é o local onde é montado o *procfs*, um sistema de arquivos que lista todos os processos que estão em execução no kernel, além de outras informações acerca do hardware. Esse é um pseudo sistema de arquivos, devido ao fato de que não existem no disco fisicamente; são apenas abstrações do kernel. Esse é um diretório muito importante, pois cede informações do sistema para ferramentas de análise, como *top* e *ps* (ELSNER; LINGNAU, 2015).

A pasta *sys* é o ambiente do *sysfs*, um sistema de arquivos virtual que organiza todos os *drivers* ativos no kernel. Assim como a anterior, é construída sob demanda do kernel e possui a sua própria hierarquia. Aplicativos que queriam acessar os *drivers* de dispositivos, os acessam por essa pasta (ELSNER; LINGNAU, 2015).

Assim sendo, pode-se verificar que cada um dos diretórios na raiz do sistema de arquivos têm um propósito. No entanto, a maioria deles só possui aplicação prática em sistemas com muitos usuários, como servidores que precisam ser verificados por administradores de sistemas; nem mesmo sistemas desktop fazem uso total desses diretórios. Em sistemas embarcados, esse padrão de hierarquia pode ser vagamente interpretado (YAGHMOUR et al., 2008). A construção dos diretórios para um sistema de arquivos embarcado será abordada no capítulo seguinte, no tópico 13.2.

12.4 ITENS ESSENCIAIS NO SISTEMA DE ARQUIVOS

Baseado na hierarquia do sistema de arquivos, é comum que sistemas baseados em Linux, médios a grandes, contenham diversos utilitários, programas e serviços a mais do que está sendo realmente usado, a fim de estarem preparados para diferentes situações. Porém, sistemas embarcados baseados em Linux seguem a linha oposta: eles carregam consigo apenas o estritamente necessário para o seu funcionamento (YAGHMOUR et al., 2008).

Sendo assim, é preciso garantir uma determinada quantidade de itens que se fazem essenciais a esses sistemas. Não importa se o sistema embarcado será um dispositivo IoT de baixíssimo consumo ou se será um controlador de máquinas industriais; em se tratando de Linux, sempre há um mínimo de objetos que devem estar no sistema de arquivos para dar vida ao dispositivo, embora essa decisão fique, muita das vezes, nas mãos do projetista (YAGHMOUR et al., 2008). Conforme Simmonds (2017), Yaghmour et al. (2008), os itens essenciais em sistemas Linux são:

12.4.1 Bibliotecas de sistema

Em sistemas operacionais, é comum que diferentes aplicações usem partes de códigos comuns, principalmente quando se trata das bibliotecas primordiais. Portanto, para poupar armazenamento, existe a prática de compilação dinamicamente lincada, que é manter, no sistema operacional, as bibliotecas que mais são usadas pela maioria dos processos. Essas bibliotecas são carregadas para a memória RAM uma vez apenas e, como são sempre de leitura, não há problema em várias aplicações as acessarem (TANENBAUM; BOS, 2016).

Manter no sistema de arquivos Linux as principais bibliotecas compartilhadas é crucial para garantir que a maioria das aplicações funcionará sem problemas. Tal fato é verdadeiro para todos os tipos de sistemas Linux (desktop e servidores) e, para os sistemas embarcados, não é diferente.

As principais bibliotecas dos sistemas GNU/Linux pertencem ao pacote *glibc* (Biblioteca GNU C, em tradução livre), que é um projeto que fornece APIs essenciais para sistemas baseados em Linux. Essas APIs fornecem recursos básicos como open, read, write, malloc, printf e outros. A compilação dessa biblioteca será feita para a construção do sistema de arquivos, no tópico 13.5.

12.4.2 Kernel Linux e módulos

Conforme abordado no capítulo 11, o kernel Linux é o primeiro executável carregado para a memória principal durante o processo de inicialização do sistema, e é ele quem controla todo o computador, interfaceando o hardware com os programas. Os módulos do kernel são extensões que podem ser carregadas e removidas da memória em tempo de execução, e servem, dentre outras coisas, para deixar o kernel mais portável. Como já foi demonstrado o processo

de compilação desses itens, a sua instalação no sistema de arquivos será demonstrada no tópico 13.4.

12.4.3 *Device nodes*

Os pontos de dispositivos (*device nodes*) são arquivos especiais, localizados na pasta */dev*, que dão acesso a vários *drivers* de dispositivo (SIMMONDS, 2017). Seguindo a tradição dos sistemas Unix – "tudo é um arquivo" – os dispositivos no sistema são representados como arquivos (com exceção das interfaces de rede) (SIMMONDS, 2017; YAGHMOUR et al., 2008). Embora o kernel não exija a presença de tais arquivos, a maioria das aplicações padrão (como *shells*) o faz para saber quantos e quais dispositivos estão presentes no sistema (YAGHMOUR et al., 2008).

Um *device node* pode ser um dispositivo de bloco (*block device*) ou um dispositivo de caractere (*character device*). Dispositivos de bloco são mídias de armazenamento de massa, tal como pendrives e cartões micro SD. Os dispositivos de caractere, por outro lado, podem representar qualquer outra coisa (novamente, com exceção das interfaces de rede). Um exemplo do primeiro é o HD (ou outro dispositivo de massa) onde está armazenado o sistema, geralmente representado por */dev/sda*; e um exemplo o segundo são as portas seriais – bastante usadas neste trabalho – que podem ser representadas por */dev/ttyS0* ou */dev/ttyUSB0* (SIMMONDS, 2017).

Em sistemas Linux genéricos, gerenciar esses arquivos é uma tarefa complexa, pois os dispositivos costumam mudar de um computador para o outro, desse modo, esses sistemas precisam constantemente rastrear os dispositivos que estão conectados no sistema. Felizmente, esse problema não acomete os sistemas embarcados, pois esses estão sempre com os mesmos dispositivos (YAGHMOUR et al., 2008). Os *device nodes* serão abordados, durante a construção do sistema de arquivos, no tópico 13.6.

12.4.4 Comandos/programas utilitários

Um sistema que possui apenas o kernel, embora seja extremamente leve, não possui função prática alguma. O que faz o sistema Linux utilizável para diferentes situações é o seu conteúdo, representado principalmente por programas utilitários. Esses programas possuem diversas funções no sistema já inciado, seja para análise das condições do mesmo, habilitações para interfaces, entrada e saída de dados, entre outros. Conforme já mencionado, o kernel possui a capacidade de adaptação a diversos ambientes diferentes, porém, em sistemas embarcados é comum o uso dos comandos do pacote GNU/Linux.

Dentre os programas essenciais em um sistema Linux, o *shell* é um dos mais significativos, pois representa a interface básica de uso do sistema. Por mais que existam poucos programas presentes no sistema embarcado (ou até mesmo nenhum), sem um *shell*, não é possível sequer invocar ou interromper o fluxo de programas. O *shell* é a principal forma de iniciar outros programas, e um *script* de *shell* é uma lista de programas a serem executados, com algum

controle de fluxo e um meio de passar informações entre programas (SIMMONDS, 2017). A instalação de comandos utilitários com o Busybox será abordada no tópico 13.3.

12.4.5 Inicializador

A partir do momento que o bootloader entrega o controle do hardware para o kernel Linux, várias ações são tomadas, como a descompactação do kernel e a montagem do sistema de arquivos. Porém, conforme informado anteriormente, o trabalho do kernel é apenas gerenciar os recursos, portanto, a tarefa de inicialização dos outros serviços e softwares essenciais fica a cargo de outro programa, que o kernel apenas invoca antes de terminar a sua inicialização. O nome desse programa é *init* (YAGHMOUR et al., 2008; SIMMONDS, 2017).

O *init* é o programa responsável por invocar os outros programas e serviços essenciais (*daemons*) para o correto funcionamento do sistema. Além disso, também monta outros sistemas de arquivos, inicia interfaces de redes, dispositivos e diversos outros organismos vitais ao funcionamento do sistema (YAGHMOUR et al., 2008; SIMMONDS, 2017).

Existem diversas possibilidades de implementação do *init* em sistemas Linux, porém, três são as mais comuns: o *systemd*, o *System V init* e o Busybox *init*, cada um com um propósito específico, complexidade e flexibilidade (SIMMONDS, 2017).

O *systemd* é definido, conforme a sua documentação (FREEDESKTOP.ORG, 2021), como um "gerenciador de sistema e de serviços". Seu objetivo, desde a sua criação em 2010, é ceder "um conjunto integrado de ferramentas para gerenciar um sistema Linux baseado em um *daemon init*" (SIMMONDS, 2017). Por conta disso, ele contém, além da função de inicialização de programas, capacidade de gerenciar serviços pós-inicialização, além de organização de dispositivos (via *udev*), interface de login e outras funcionalidades (SIMMONDS, 2017; FREEDESKTOP.ORG, 2021).

Atualmente, o *systemd* é considerado o estado da arte em entre os *init*s para Linux, e é o mais usado nas distros para desktop e servidores. No mundo embarcado, seu uso tem crescido, mas limitado a dispositivos de alta complexidade, porém, devido ser extenso e complexo para configurar (SIMMONDS, 2017).

O *systemd* foi desenvolvido para substituir o antigo *System V init*, a fim de atender as novas demandas da atualidade. Ele fornece uma forma mais eficiente de configurar, gerenciar e limitar *scripts* de inicialização. Além disso, provê uma intensa paralelização durante a inicialização, acelerando-a (SIMMONDS, 2017; FREEDESKTOP.ORG, 2021).

O *System V init* data dos primórdios do Linux, pois fora baseado nos antigos *System V* dos sistemas Unix. Esse *init* foi usado nos sistemas Linux até recentemente, quando foi suplantado pelo *systemd* (SIMMONDS, 2017). Embora tenha sido substituído por um software mais moderno, ainda assim, o *System V init* configura como uma opção válida e, em sistemas embarcados, se destaca em dispositivos de média complexidade (SIMMONDS, 2017).

A principal característica do *System V init* é o conceito de níveis de execução (*runlevels*), que permitem que uma coleção de programas seja iniciada ou interrompida de uma vez ao alternar de um nível de execução para outro. Isso permite selecionar prioridades para os programas e serviços na hora da inicialização (SIMMONDS, 2017).

Por fim, o Busybox *init* é uma implementação simplificada do *init* para Linux no utilitário Busybox. Pelo fato de ser pequeno e de fácil configuração, o *init* do Busybox acabou ganhando bastante notoriedade entre os sistemas embarcados, já que geralmente as funções implementadas neles são simples e não precisam de tanto gerenciamento, como o fornecido pelo *systemd*, por exemplo (SIMMONDS, 2017).

Segue abaixo uma tabela comparativa entre os três *inits* – figura 36, que demonstra questões como complexidade, velocidade, tamanho, entre outros. Percebe-se que o *systemd* é o maior e mais complexo, seguido pelo *System V init*. A configuração e a instalação do recurso *init* do Busybox será feita no tópico 13.6.

Figura 36 – Tabela comparativa entre os três principais *inits*

Métrica	BusyBox init	System V init	systemd
Complexidade	Baixa	Média	Alta
Velocidade de inicialização	Rápida	Lenta	Média
Shell requerido	*ash*	*ash* ou *bash*	Nenhum
Número de executáveis	0	4	50
libc	Qualquer uma	Qualquer uma	*glibc*
Tamanho (MB)	0	0.1	34

Adaptado de Simmonds (2017)

13 CONSTRUÇÃO DO SISTEMA DE ARQUIVOS DO ZERO

Um sistema de arquivos construído especificamente para uma aplicação por vezes é necessário, no projeto de um sistema embarcado Linux. Logo, o objetivo deste capítulo é demonstrar a construção de um sistema de arquivos embarcado do zero, aplicando os conceitos apresentados no capítulo anterior.

Isto posto, este capítulo está distribuído em: comentário sobre as imagens do SDK (tópico 13.1), criação da estrutura de diretórios (13.2), instalação do Busybox (13.3), instalação do kernel (13.4), compilação da *glibc* (13.5), configuração do *init* (13.6), comentário sobre os *device nodes* (13.7) e demonstração (13.8).

13.1 SOBRE OS SISTEMAS DE ARQUIVOS DO SDK

Durante a fase de construção do sistema de arquivos de um dispositivo embarcado Linux, duas possibilidades são abertas ao desenvolvedor: construir o seu sistema de arquivos do zero, compilando e instalando os programas e bibliotecas manualmente; ou utilizar algum sistema de arquivos pronto, aproximadamente encaixado nas especificações do projeto.

Quando se utiliza o SDK da Texas Instruments, alguns sistemas de arquivos prontos para a utilização são cedidos. O principal objetivo disso, além de testar e avaliar a placa, é acelerar o *time to market* do projeto, isto é, facilitar o trabalho do desenvolvedor, criando um sistema de arquivos testado e funcional que apenas precise de remoção (ou adição) de recursos para se encaixar nos requisitos do dispositivo (TI, 2019). Dessa forma, muito esforço e, consequentemente, tempo, será poupado.

Sendo assim, quatro sistemas de arquivos prontos vêm junto com o SDK. A TI os preparou pensando em várias situações diferentes que os seus *SoCs* poderiam ser aplicados, isto é, cada um possui uma particularidade. Além disso, eles também possuem tamanhos diferentes, visando aplicações que precisem de mais recursos ou aplicações bastante minimalistas. Desse modo, os quatro sistemas de arquivos cedidos pela Texas Instruments são:

- ***tisdk-rootfs-image:*** tamanho compactado: 893,7 MB, descompactado: 2,3 GB;
- ***tisdk-docker-rootfs-image:*** tamanho compactado: 139,9 MB, descompactado: 666,8 MB;
- ***arago-base-tisdk-image:*** tamanho compactado: 58,5 MB, descompactado: 272,6 MB;
- ***arago-tiny-image:*** tamanho compactado: 10,6 MB, descompactado: 53,4 MB.

O primeiro sistema de arquivos, e o maior de todos, é o *rootfs*. É a imagem completa de um sistema operacional embarcado, contendo diversos comandos e recursos padrão do Linux. Além disso, contém os programas e bibliotecas do SDK, exemplos e outros, sendo o principal deles a interface gráfica *matrix-gui-browser-2.0* (TI, 2019). Pelo fato de ser conter muitos

recursos, é também a imagem mais pesada, tanto em volume quanto em velocidade de execução, portanto, é necessário escolher o que manter para adequar aos requisitos do dispositivo.

A segunda imagem é a *docker*, que, conforme a documentação da TI (2019), foi otimizada para suportar conteinerização via *Docker*. Esse recurso complementa o *namespace* do kernel, utilizando uma API que permite forte isolamento entre processos. Dessa forma, o *Docker* pode servir para isolar serviços no *target*, sendo um ótimo bloco de construção para automatizar sistemas distribuídos, entre outras funcionalidades (TI, 2019).

A terceira imagem é a *arago-base*, ou simplesmente *base*. É um sistema de arquivos genérico, porém, otimizado para sistemas embarcados. O objetivo dele, segundo a TI (2019), é ser um ponto de partida para a adição de pacotes e, consequentemente, para a criação de um sistema de arquivos que atenda às necessidades do projeto. De fato, é o sistema mais leve com mais recursos presentes, pois contém diversos utilitários, serviços e até um servidor web adaptável.

O quarto e último sistema de arquivos é o *tiny*. Como o nome já sugere, essa é a menor de todas as imagens, contento apenas o estritamente necessário para a inicialização e utilização básica do sistema (TI, 2019). Essa imagem é viável quando a anterior não se encaixa nos requisitos do projeto, ou quando o sistema é bastante restrito, necessitando de apenas poucas bibliotecas, ou ainda quando se quer ter mais controle sobre os pacotes instalados, e se quer começar de um sistema de arquivos funcional.

Apesar de serem bastante úteis, nem sempre esses sistemas de arquivos são utilizados. Existem situações nas quais é necessário ter controle absoluto sobre cada biblioteca e programa instalado no sistema embarcado; acerca de questões, por exemplo, de versão e atualização. Além disso, os sistemas de arquivos do SDK possuem a desvantagem de precisarem de remoção (ou adição) de recursos para se encaixarem no projeto (como em uma abordagem de cima para baixo), assim, não se encaixando perfeitamente nos requisitos, pois é comum que sobrem pacotes que não se sabe a função.

Para essas situações, é necessário criar o sistema de arquivos do zero. Embora seja muito mais trabalhoso, esse procedimento garante controle total de pacotes e versões dos softwares instalados no sistema embarcado, o que pode ser importante dependendo das regras de negócio. Ademais, é importante demonstrar esse processo para servir de referência para outros, conforme os objetivos deste trabalho. Esse sistema de arquivos só será usado nesse capítulo, pois nos outros (anteriores e posteriores) foram usados os do SDK, devido a sua praticidade.

13.2 CRIAÇÃO DAS PASTAS E CONFIGURAÇÃO DAS PERMISSÕES

A hierarquia do sistema de arquivos é importante para os sistemas Linux, tanto para execução de serviços quanto para mantê-los. Porém, diferentes sistemas computacionais (desde desktop até mainframes) nem sempre seguem essa regra, e com os sistemas embarcados não é

diferente (YAGHMOUR et al., 2008). Baseado em Yaghmour et al. (2008), Hallinan (2006), os diretórios essenciais que serão criados nessa abordagem são: */bin, /boot, /dev, /etc, /lib, /proc, /sbin, /sys*, e */usr*.

Como o sistema de arquivos está sendo construído do zero, as pastas da raiz devem assim ser também construídas. Pode-se fazer isso diretamente no cartão de memória já montado no sistema (na partição *rootfs*). Porém, é uma boa prática fazer uma pasta para testes no *host* primeiro, para, depois de completo. ser transferido para o cartão de memória. Tal prática é conhecida como diretório de teste e é recomendada por Simmonds (2017).

Assim sendo, será criada uma pasta dentro de */texasSDK/board-support* chamada *rootfs*, e lá serão criadas as pastas essenciais do sistema de arquivos embarcado. A sequência de operações, que também pode ser feita via GUI, é mostrada no console abaixo.

```
# Criação da pasta "espelho" do sistema de arquivos do target
$ cd ~/texasSDK/board-support
$ mkdir rootfs && cd rootfs

# Criação dos diretórios essenciais
$ mkdir bin boot dev etc lib proc sbin sys usr
$ mkdir usr/bin usr/lib usr/sbin

# Modificação das permissões em /proc e /sys
$ chmod 555 proc sys
```

Além disso, é necessário alterar as permissões das pastas *proc* e *sys*. Devido essas pastas serem os locais nos quais serão montados os sistemas virtuais *procfs* e *sysfs*, vitais para monitoramento e utilização do sistema, elas também deve ter permissões diferenciadas das outras. É necessário garantir que nenhum usuário possa escrever (gravar) nessas pastas, apenas ler e executar, para garantir a estabilidade do sistema. Logo, alteram-se as permissões com o comando chmod, conforme mostrado no console acima.

13.3 INSTALAÇÃO DOS COMANDOS UTILITÁRIOS COM O BUSYBOX

A etapa seguinte na construção do sistema de arquivos é populá-lo com comandos básicos. Os comandos do Linux, são, em sua maioria, executáveis escritos em C, cada um com sua árvore de código-fonte e seu *Makefile*. Escolher os comandos necessários para o sistema de arquivos, baixá-los da internet e compilá-los um a um seria uma tarefa extenuante para o desenvolvedor, além de gerar dezenas de megabytes que, dependendo da situação, poderia fugir do escopo do projeto (YAGHMOUR et al., 2008).

Para resolver essa situação, os projetistas de sistemas embarcados Linux fazem uso massivo da ferramenta Busybox, que produz vários executáveis em um, de uma vez. A meta

nessa etapa seria fazer a compilação desse utilitário, porém, já foi feita no capítulo 10, portanto, o produto da compilação será apenas transferido para o sistema de arquivos que está sendo produzido.

A estrutura de diretório do Busybox está conforme a construída na seção anterior, logo, basta copiá-la, e o sistema de arquivos em *rootfs* será populado automaticamente. O Busybox possui itens nas pastas *bin, sbin, usr/bin* e *usr/sbin*.

```
$ cd ../busybox_build/
$ cp -r . ../rootfs/ && cd ../rootfs
```

13.4 INSTALAÇÃO DO KERNEL LINUX E DOS MÓDULOS

Após a transferência dos comandos utilitários, é o momento de transferir o kernel e os seus módulos, compilados no capítulo 11. Novamente, não é necessário compilar nessa etapa, visto que já fora feito.

Considerando que os produtos da compilação estão na pasta *kernel_build*, a transferência pode ser feita para a pasta *rootfs* a partir dos comandos mostrados abaixo. O kernel e o arquivo de *device tree* devem ser copiados para a pasta */boot* e os módulos, para a pasta */lib*.

```
# Transferência dos kernel e do arquivo de device tree
$ cd ../kernel_build/
$ cp am335x-boneblack.dtb zImage ../rootfs/boot/

# Transferência dos módulos
$ cp -r lib ../rootfs/ && cd ../rootfs
```

13.5 COMPILAÇÃO DAS BIBLIOTECAS DE SISTEMA: *GLIBC*

Nesse momento, é a vez de compilar e transferir as bibliotecas compartilhadas essenciais do sistema: a biblioteca padrão C do pacote GNU, *glibc*. Esse projeto provê as bibliotecas centrais dos sistemas baseados em GNU/Linux, assim como outros baseados no kernel Linux. Suporta diferentes arquiteturas de sistema (ou seja, permite compilação cruzada), segue o padrão ISO C e foi desenvolvido para ser retrocompatível, portável e de alta performance – o que o torna perfeito, também, para aplicações embarcadas (GNU, 2020).

O processo de compilação é semelhante ao dos pacotes já compilados neste trabalho. Primeiramente, acessa-se o repositório oficial dos códigos-fontes e faz-se o download da versão mais atualizada (durante o desenvolvimento deste trabalho, foi usada a versão 2.32 da *glibc*). Após isso, extrai-se e clona-se o diretório como boa prática. A sequência de passos é mostrada no console abaixo.

```
# Download do código-fonte
$ cd ~/Downloads/
$ wget https://ftp.gnu.org/gnu/libc/glibc-2.32.tar.xz
# Extração
$ tar -xf glibc-2.32.tar.xz
# Movendo o código-fonte para a pasta "board-support"
$ mv glibc-2.32 ~/texasSDK/board-support
# Clonando a pasta
$ cd ~/texasSDK/board-support
$ cp -r glibc-2.32 glibc-2.32-RASCUNHO
```

Tendo o código-fonte disponível, pode-se passar para a etapa de compilação. A *glibc* é extensa, e a sua compilação pode demorar até mais de uma hora em computadores fracos. Além disso, ela possui suporte a diversas configurações, mas essas são feitas via um *script* chamado *configure*, e não pelo *menuconfig*. No caso deste trabalho, a compilação básica – com as opções padrão – será feita.

A compilação da *glibc* destoa de outras compilações realizadas neste trabalho, visto que algumas opções são específicas, embora a sequência de passos seja a mesma. À priori, durante a exportação das variáveis de ambiente, são necessárias mais algumas além das básicas ARCH e CROSS_COMPILE, conforme se observa no console abaixo. Uma pasta foi criada para a saída da compilação com o nome *glibc_build*.

```
# definir a arquitetura
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ export CC=arm-linux-gnueabihf-gcc
$ export RANLIB=arm-linux-gnueabihf-ranlib
$ export STRIP=arm-linux-gnueabihf-strip

# definir o caminho pasta de saída
$ export RAIZ="/home/felipe/texasSDK/board-support/glibc_build"
```

Além disso, diferentemente de outras bibliotecas e pacotes de software, a *glibc* não permite configuração de dentro do seu diretório; é necessário iniciar a compilação diretamente do diretório de saída, *glibc_build*. Portanto, encaminha-se até a pasta de saída, e de lá configura-se e inicia-se a compilação.

```
# abrir a pasta de saída
$ cd $RAIZ

# configurando a compilação com ./configure
$ /home/felipe/texasSDK/board-support/glibc-2.32-RASCUNHO/./configure \
  --prefix= \
  --host=arm-linux-gnueabihf \

# compilar
$ make -j9
```

A instalação do pacote pode prosseguir com o comando abaixo.

```
$ make -j9 install DESTDIR=$RAIZ
```

As bibliotecas compartilhadas produzidas pela *glibc* ficam na pasta *$RAIZ/lib*. São várias, para diversas aplicações de sistema. Porém, conforme aponta Yaghmour et al. (2008), apenas algumas são essenciais, visto que a maioria é raramente usada. Dessa forma, as principais bibliotecas do pacote *glibc* são:

- ***libc***: A principal das bibliotecas; contém as principais funções C;
- ***libcrypt***: Necessária para a maioria das aplicações que envolvem autenticação;
- ***libdl***: Necessária para aplicações que usem funções como *dlopen()*;
- ***libm***: Necessária para funções matemáticas;
- ***libpthread***: Necessária para programação com *threads*;
- ***libresolv***: Necessária para resolução de nomes;
- ***libutil***: Necessária para conexão com terminais;
- ***ld***: Esse componente não é exatamente uma biblioteca, mas sim o *linker* oficial da *glibc*. Todo executável ELF no sistema o procura para carregar as suas bibliotecas vinculadas dinamicamente.

Além das bibliotecas em si, observa-se que existem links simbólicos para elas, geralmente com um número no final (*.so.6*). Tais links servem para questões de retrocompatibilidade (YAGHMOUR et al., 2008).

Dessa forma, as bibliotecas e seus links simbólicos podem ser transferidos para a pasta *rootfs*. Um *script* para facilitar essa tarefa foi cedido por Yaghmour et al. (2008) e se encontra adaptado abaixo. O resultado dessa transferência é mostrado na figura 37.

```
$ cd ${RAIZ}/lib
$ for file in libc libcrypt libdl libm libpthread libresolv libutil
> do
> cp $file-*.so ~/texasSDK/board-support/rootfs/lib
> cp -d $file.so.[*0-9] ~/texasSDK/board-support/rootfs/lib
> done
$ cp -d ld*.so* ~/texasSDK/board-support/rootfs/lib
```

Figura 37 – Pasta */lib* antes do *stripping*

Fonte: AUTOR (2021)

Após a cópia, percebe-se que o peso das bibliotecas somado é grande, o que pode ser um problema para o dispositivo final. Por isso, existe uma ferramenta na toolchain (pacote do compilador cruzado) que permite reduzir o tamanho dessas bibliotecas removendo símbolos de depuração (YAGHMOUR et al., 2008). O comando de *stripping* segue abaixo e o resultado é mostrado na figura 38.

```
$ arm-linux-gnueabihf-strip ~/texasSDK/board-support/rootfs/lib/*.so
```

Figura 38 – Pasta */lib* depois do *stripping*

Fonte: AUTOR (2021)

Vale ressaltar que é melhor fazer isso depois de copiadas as bibliotecas, pois talvez seja necessário copiá-las novamente (YAGHMOUR et al., 2008). Além disso, não se deve reduzir os módulos do Linux, pois alguns símbolos são exigidos pelo carregador de módulo, logo, o módulo não será carregado se eles forem removidos (SIMMONDS, 2017).

Por fim, o *script* completo da compilação feita nessa seção é apresentado na figura 39.

Figura 39 – *script* de compilação da *glibc*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
export CC=arm-linux-gnueabihf-gcc
export RANLIB=arm-linux-gnueabihf-ranlib
export STRIP=arm-linux-gnueabihf-strip

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/glibc_build"

## INÍCIO ##

# abrir a pasta de saída
cd $RAIZ

# configurando a compilação com ./configure
/home/felipe/texasSDK/board-support/glibc-2.32-RASCUNHO/./configure \
        --prefix= \
        --host=arm-linux-gnueabihf

# limpar compilações anteriores
make distclean

# compilar
make -j9

# instalar
make -j9 install DESTDIR=$RAIZ
```

Fonte: AUTOR (2021)

13.6 CONFIGURAÇÃO DO BUSYBOX *INIT*

Conforme descrito no capítulo anterior, existem três utilitários principais com a função de inicializadores no Linux. Porém, é visível que dois deles, o *systemd* e o *System V init*, são, na maioria das vezes, complexos demais para sistemas embarcados, visto que têm como objetivo grandes sistemas geralmente. Além disso, sua configuração manual é consideravelmente complexa, o que foge do escopo dessa abordagem. Por isso, foi escolhido para esse sistema de arquivos mínimo o *init* do Busybox.

Assim como todos os outros comandos básicos cedidos pelo Busybox, o *init* também está contido dentro do executável, sendo acessível por link simbólico (presente na pasta */sbin/init*). Esse *init* é particularmente bem adaptado para sistemas embarcados, visto que provê a funcionalidade necessária sem adicionar mais peso ao sistema (ou mais complexidade). No entanto, nem sempre esse recurso será suficiente, por exemplo, em sistemas embarcados mais complexos que precisem de *runlevels* (YAGHMOUR et al., 2008).

O Busybox *init* funciona como o *System V init*, só que de modo mais simples: ele observa o conteúdo de um arquivo, */etc/inittab*, que possui um formato específico para ser analisado pelo

init. Esse arquivo separa os comandos em linhas, e cada comando é separado por dois pontos, baseado no formato apresentado abaixo.

```
1 id:runlevel:action:process
```

O campo *id* é o dispositivo serial que o comando será executado; se deixado vazio, será usada a serial padrão. O campo *runlevel* é ignorado, portanto, pode-se deixá-lo em branco. O campo *process* é o comando a ser executado, com seus parâmetros, se houver. Por fim, o campo *action* refere-se ao modo como o processo há de ser iniciado, conforme as oito possibilidades apresentadas abaixo (YAGHMOUR et al., 2008; SIMMONDS, 2017):

- ***sysinit***: O comando é executado antes de todos os outros;
- ***respawn***: Inicia o programa e o reinicia, caso ele termine. Usado para executar comandos como *daemons*;
- ***askfirst***: Semelhante ao anterior, porém requer uma entrada do usuário para reiniciar;
- ***wait***: Inicia o programa e espera pelo seu término;
- ***once***: Inicia o programa e não tenta reiniciá-lo, caso termine;
- ***ctrlaltdel***: Inicia quanto o sinal SIGINT for acionado (geralmente pelas teclas Ctrl + Alt + Del);
- ***shutdown***: Executa o comando quando o sistema estiver desligando;
- ***restart***: Executa o processo quando o *init* reinciar; geralmente é o próprio *init*.

Durante a inicialização, o *init* do Busybox executa as seguintes ações na seguinte ordem, segundo Yaghmour et al. (2008):

1. Configura manipuladores de sinal para *init*;
2. Inicializa o console. Por padrão, ele usa o dispositivo especificado pelo kernel. Se nenhum for cedido, ele tenta usar o */dev/console*;
3. Analisa o arquivo *inittab*, em /etc/*inittab*. Se nenhum for cedido, ele usa uma configuração padrão;
4. Executa o *script* de inicialização padrão, */etc/init.d/rcS*;
5. Executa os comandos do *inittab* que forem de espera (*wait*);
6. Executa os comandos do *inittab* que só rodam uma vez (*once*);

Após esses passos, o *init* continua executando eternamente, cuidando das seguintes tarefas:

1. Executa os comandos do *inittab* que forem de reexecutados (*respawn*);
2. Executa os comandos do *inittab* que devem ser solicitados primeiro (*askfirst*);
3. Aguarda os processos filhos encerrarem.

O primeiro passo da inicialização, após o console, é analisar o arquivo *inittab*. Esse arquivo serve, principalmente, para configurar ações padrão para reinicialização, chamada de serviços, entre outros. Caso não seja cedido, o *init* segue a configuração padrão mostrada no

console abaixo. Como essa é completa o bastante para a demonstração deste capítulo, não será criado um arquivo *inittab*.

```
# Configuração padrão do inittab - Busybox init

::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
::restart:/sbin/init
tty2::askfirst:/bin/sh
tty3::askfirst:/bin/sh
tty4::askfirst:/bin/sh
```

O primeiro passo do *inittab* é executar o *script rcS*, que se encontra em *etc/init.d*. Portanto, cria-se esse diretório, cria-se o arquivo e configura-o com permissão de execução. Os passos em console são demonstrados abaixo.

```
$ cd ~/texasSDK/board-support/rootfs
$ mkdir etc/init.d && cd etc/init.d
$ touch rcS
$ chmod +x rcS
```

O próximo passo é editar esse arquivo para definir quais comandos serão executados na inicialização. Essa etapa depende muito de cada projeto, e o desenvolvedor é livre para adicionar diversos tipos de comandos diferentes. Porém, existem alguns que são, de certo modo, padrão para esse arquivo, sendo uma boa prática inseri-los. Esses comandos são: a montagem dos sistemas de arquivos virtuais *procfs* e *sysfs*, a inicialização dos *loggers* de sistema e de kernel – *syslogd* e *klogd* – e a configuração da interface *loopback* de rede; conforme demonstrado na figura 40. Vale destacar que, como o kernel não está com a interface de rede habilitada, os comandos ifconfig e o syslogd foram comentados.

Figura 40 – Arquivo *etc/init.d/rcS*

```
#!/bin/sh

echo "Montando procfs"
mount -t proc /proc /proc

echo "Montando sysfs"
mount -t sysfs sysfs /sys

echo "Iniciando os loggers do sistema"
# syslogd
klogd

# echo "Configurando a interface loopback"
# ifconfig lo 127.0.0.1
```

Fonte: AUTOR (2021)

13.7 ACERCA DOS *DEVICE NODES*

Conforme abordado no capítulo anterior, os *device nodes* são os dispositivos presentes no sistema Linux, representados na forma de arquivos. Cada dispositivo possui um arquivo, e montar todos esses arquivos manualmente, dependendo da quantidade de arquivos presentes no sistema, pode ser inviável.

No entanto, o kernel possui uma funcionalidade, chamada *devtmpfs*, que monta os *device nodes* sobre demanda, automaticamente. Ele é um pseudo sistema de arquivos que é montado no momento da inicialização do kernel, e preenche a pasta */dev* com os dispositivos conhecidos pelo kernel até o momento (SIMMONDS, 2017).

Se, no momento da configuração do kernel (tópico 11.4) for selecionada a opção "*Automount devtmpfs at /dev, after the kernel mounted the rootfs*", que equivale a CONFIG_DEVTMPFS_MOUNT no arquivo .config, então esses dispositivos serão montados automaticamente. Na configuração padrão do SDK TI essa opção já vem ativada e, como não foi alterada, os *device nodes* aparecerão automaticamente.

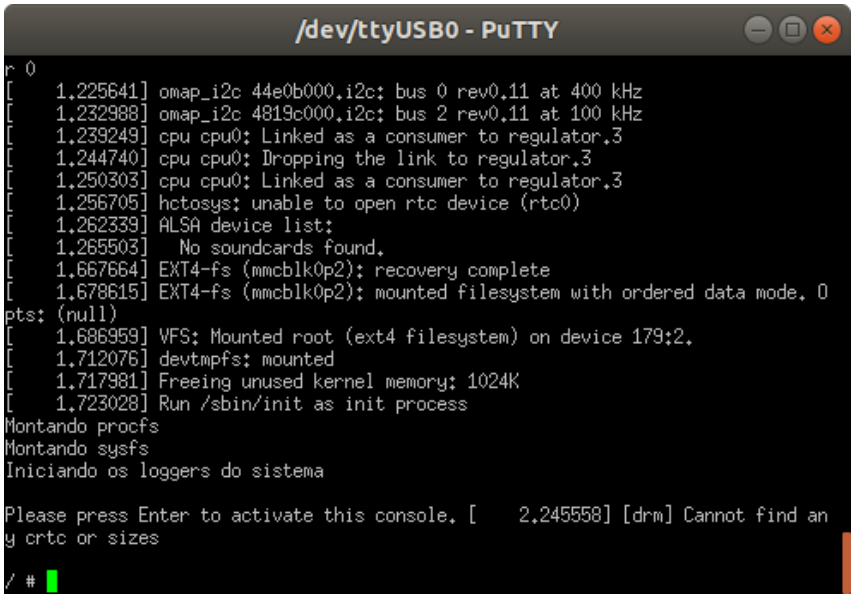
13.8 DEMONSTRAÇÃO

Por fim, o sistema de arquivos está completo. O próximo passo é transferir para o cartão micro SD o conteúdo da pasta de teste, *rootfs*. É necessário, primeiro, limpar da partição o sistema de arquivos antigo para, então, instalar o novo. Esses passos são feitos conforme os comandos abaixo.

```
1 $ sudo rm -r /media/felipe/rootfs
2 $ sudo cp -r ~/texasSDK/board-support/rootfs/. /media/felipe/rootfs
```

A tela de boot do sistema construído é apresentada na figura 41. Observa-se o final das informações de log do kernel – referentes a sua inicialização – e as informações do *init*, presentes no arquivo *etc/init.d/rcS*.

Figura 41 – Tela de *boot* do sistema montado do zero



Fonte: AUTOR (2021)

Conforme foi configurado, os sistemas de arquivos virtuais, isto é, o *sysfs* e o *procfs*, além do sistema de dispositivos, *devtmpfs*, foram todos montados. A listagem das suas pastas é apresentada nos consoles abaixo.

```
$ ls sys
block      class      devices    fs         module
bus        dev        firmware   kernel     power
```

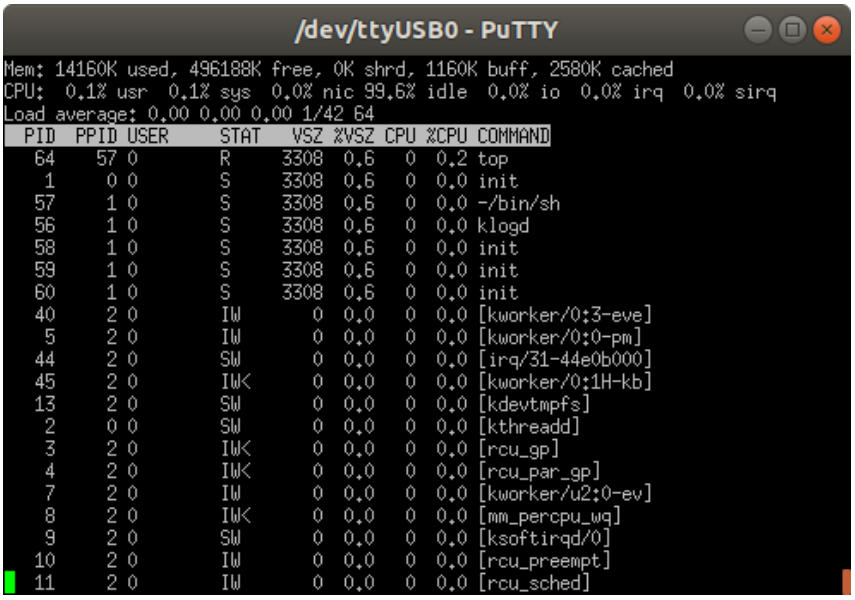
```
$ ls proc
1              41             8              interrupts     sched_debug
10             42             9              iomem          schedstat
11             43             asound         ioports        self
12             44             buddyinfo      irq            softirqs
13             45             bus            kallsyms       stat
14             46             cgroups        key-users      sys
16             47             cmdline        keys           sysrq-trigger
(...)
```

```
$ ls dev
autofs              ptyv7               ttyd6
console             ptyv8               ttyd7
cpu_dma_latency     ptyv9               ttyd8
dri                 ptyva               ttyd9
full                ptyvb               ttyda
gpiochip0           ptyvc               ttydb
(...)
```

Além disso, os comandos utilitários foram igualmente instalados com a ferramenta Busybox. Isso significa que, mesmo com um sistema de arquivos básico e diminuto, diversas operações são possíveis, como manipulação de arquivos – com o *touch, cp, vi, less* e muitos outros; montagem de sistemas de arquivos – com *mount*; configuração de rede – com o *netstat, ifconfig* e *ping* (embora a rede esteja desabilitada nesse kernel); além de muitos outros comandos. Desse modo, embora não haja nenhuma aplicação customizada nesse sistema, o que foi construído já é o suficiente como base para a maioria das aplicações.

O último detalhe a ser destacado nesse sistema de arquivos mínimo, é em relação ao consumo de recursos, ou seja, quantos processos estão em execução. Como não foi configurado quase nada para rodar, é esperado que o processador esteja praticamente vazio, contendo apenas o kernel em si e o que foi iniciado, que é o *init*, o *klogd* e o *shell*. Tal demonstração, feita com o auxílio da ferramenta *top*, pode ser vista na figura 42.

Figura 42 – Utilização dos recursos computacionais com o sistema de arquivos mínimo



Fonte: AUTOR (2021)

Nota-se que o processador está sendo pouquíssimo usado (em 0,1% da capacidade, apenas), além do fato de que a maior parte da memória está livre. Um fato interessante é a memória RAM que está sendo usada no momento é aproximadamente 14 MB. Esse tamanho é o mesmo do kernel descompactado, indicando, portanto, que nenhum processo relevante o bastante está sendo executado além do kernel.

Parte V

Configuração das interfaces de utilização

14 HABILITAÇÃO DO SSH COMO *SHELL* REMOTO

A hiperconectividade dos dispositivos IoT, juntamente com vantagens, trouxe riscos. É mandatório que o projetista leve em consideração bem embarcar soluções de segurança em seus dispositivos. Assim, este capítulo contextualiza a questão da segurança em IoT, a partir da configuração e uso do *OpenSSH* para sistemas embarcados.

Desse modo, este capítulo está subdividido em: conceito do SSH – tópico 14.1, importância – tópico 14.2, preparação para a compilação – tópico 14.3, compilação das dependências – tópico 14.4, compilação e instalação do *OpenSSH* – tópico 14.5 e utilização do *ssh* e do *scp* – tópicos 14.6 e 14.7.

14.1 CONCEITO

Nos anos 1990, houve a criação e advento da internet. Logo, surgiu a possibilidade (e necessidade) de conectar computadores em todo o mundo, trocar dados e arquivos e até logar-se remotamente em uma máquina; tudo isso a partir de contas pessoais, como as de usuário e de e-mail, por exemplo. Desse modo, diversos protocolos e programas surgiram para sanar esses problemas, tais quais os conhecidos TELNET e FTP, para conexão remota e transferência de arquivos, respectivamente (BARRETT; SILVERMAN, 2001).

No entanto, a internet logo se mostrou um ambiente inseguro. Um arquivo contendo dados sensíveis poderia ser interceptado e lido, ou até alterado, comprometendo sua integridade. Além disso, pela fata de autenticação, pessoas não autorizadas poderiam logar remotamente em uma máquina. Todos esses problemas aconteciam principalmente pela falta de criptografia, pois os dados – inclusive senhas – eram enviados como texto simples diretamente (BARRETT; SILVERMAN, 2001).

Na época, nenhum dos utilitários existentes possuíam recursos para evitar essas situações, pois careciam do principal elemento para navegação em redes, globais e locais – segurança. Por isso, surgiu a necessidade de uma solução de corrigisse os problemas citados em um único utilitário, de modo transparente e intuitivo – o protocolo SSH (BARRETT; SILVERMAN, 2001).

O protocolo SSH (*Secure Shell*) é um método para garantir um login remoto seguro de um computador para outro, mesmo sob redes inseguras. Isso é obtido a partir de uma forte autenticação, além da proteção da comunicação usando uma forte criptografia. Atualmente, é a principal ferramenta para login remoto e transferência de arquivos – utilizando os protocolos SSH e SFTP, respectivamente – suplantando os antigos TELNET e FTP (BARRETT; SILVERMAN, 2001; YLONEN; LONVICK et al., 2006).

O funcionamento básico do SSH é proteger a comunicação: sempre que os dados são

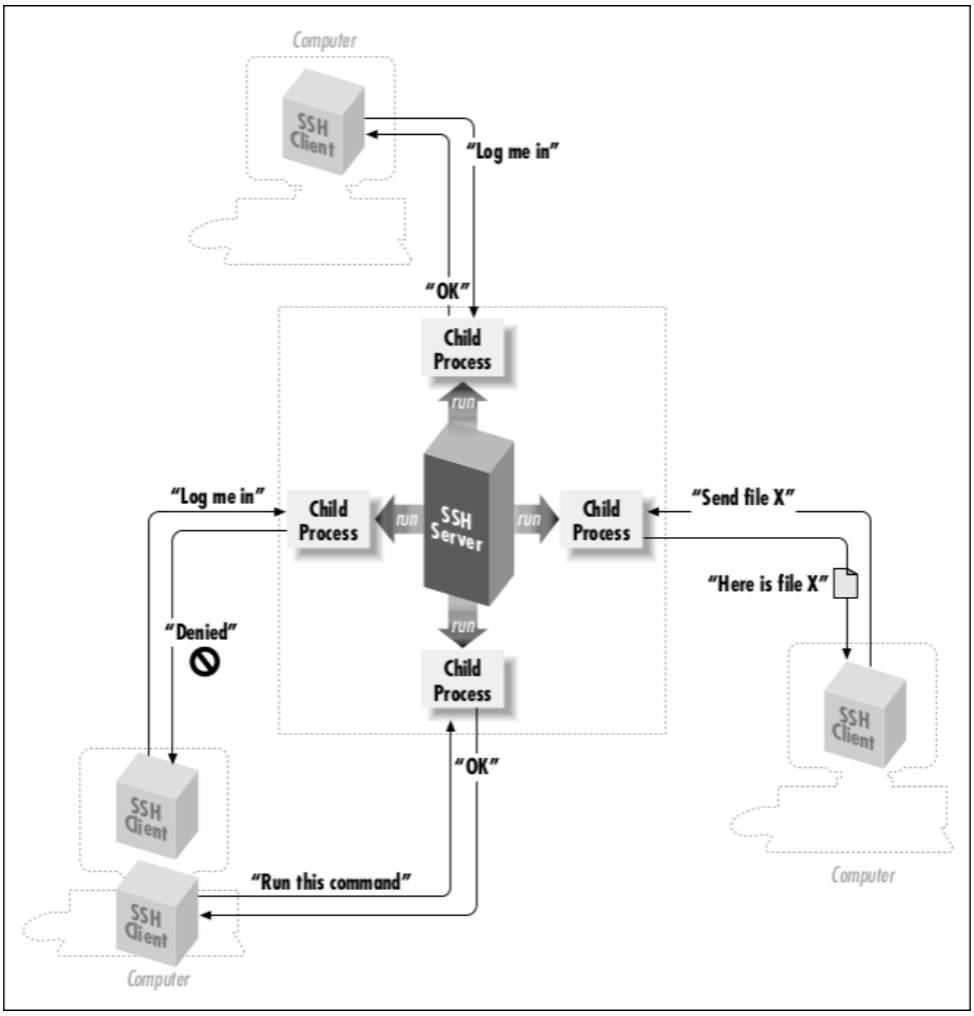
enviados por um computador para a rede, o SSH os criptografa automaticamente; quando chegam ao destinatário, são descriptografados. O resultado é uma proteção transparente: os usuários podem trabalhar e se conectar normalmente, pois as suas comunicações estão protegidas via algoritmos seguros e modernos de criptografia (BARRETT; SILVERMAN, 2001).

O SSH se tornou bastante popular, e dois motivos para isso são destacados. Primeiro, o fato de ser baseado em software permite que seja agregada segurança em qualquer dispositivo. Segundo, é um recurso bastante poderoso com diversas funcionalidades. Essas características permitem que o SSH seja aplicado em todos os ramos da computação, desde data centers até sistemas embarcados (BARRETT; SILVERMAN, 2001).

É válido destacar que o SSH é um protocolo – não um produto, ou seja, uma regra de determina como se deve comunicar computadores em redes inseguras (BARRETT; SILVERMAN, 2001). Tal protocolo está especificado na documentação oficial, escrita por Tatu Ylonen (YLONEN; LONVICK et al., 2006). Dessa forma, existem diversas implementações diferentes desse protocolo representadas por programas, pagos ou gratuitos; com códigos fechados ou abertos. A principal ferramenta open source e gratuita é o *OpenSSH*.

O SSH é, basicamente, uma estrutura cliente-servidor, conforme mostrado na figura 43 abaixo. Esse exemplo representa uma situação comum em sistemas de servidores, no qual um ou mais computadores podem se conectar ao servidor. Isso é possível pois o SSH possui duas funcionalidades: o modo servidor, que é um processo em execução na máquina que há de ser conectada; e o modo cliente, que faz a conexão remota (BARRETT; SILVERMAN, 2001).

Figura 43 – Arquitetura da conexão SSH



Fonte: (BARRETT; SILVERMAN, 2001)

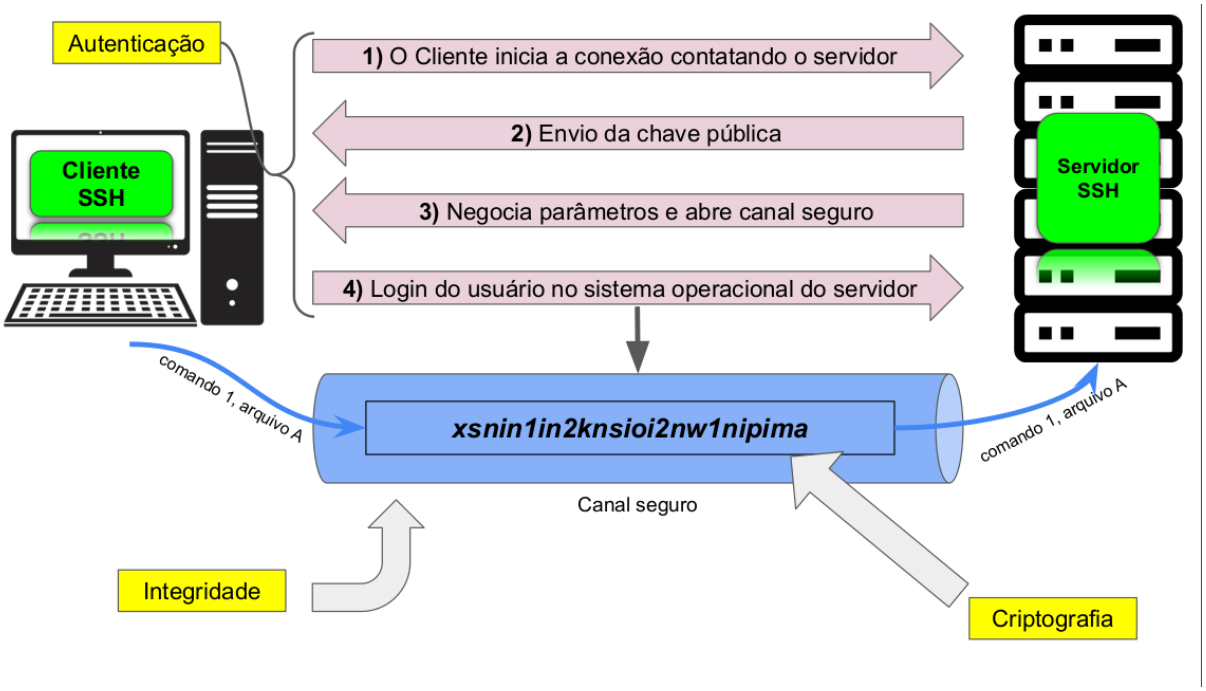
Um programa de servidor SSH, normalmente instalado e executado por um administrador de sistema, aceita ou rejeita conexões de entrada dos computadores clientes. Os usuários, por sua vez, executam programas clientes SSH para fazer solicitações ao servidor SSH – geralmente login, comandos e arquivos. Portanto, o protocolo SSH estabiliza a conexão do cliente com o servidor mediante seu programa utilitário (BARRETT; SILVERMAN, 2001).

O funcionamento do SSH é baseado nos princípios de autenticação, integridade e criptografa. Em síntese, o cliente SSH conduz o processo de configuração da conexão e usa criptografia de chave pública para verificar a identidade do servidor SSH. Após a fase de configuração, o protocolo SSH usa criptografia simétrica forte e algoritmos de *hash* para garantir a privacidade e integridade dos dados trocados entre o cliente e o servidor (YLONEN; LONVICK et al., 2006).

O diagrama simplificado do funcionamento do protocolo é mostrado na figura 44. Primeiramente, é feita a autenticação, que serve para determinar, com segurança, a identidade de alguém. Quando o usuário tenta se conectar remotamente a um computador, o SSH solicita a sua identidade digital (chave); se for aprovada, o SSH permite a conexão, senão, a rejeita

(BARRETT; SILVERMAN, 2001; YLONEN; LONVICK et al., 2006).

Figura 44 – Diagrama do funcionamento do protocolo SSH



Adaptado de Barrett e Silverman (2001), Ylonen, Lonvick et al. (2006)

Uma vez permitida a conexão, é aberto um canal seguro para a conexão, protegido de ponta a ponta com criptografia. Essa proteção garante que os dados sejam ilegíveis para invasores ou outros membros da rede, exceto, apenas, para o destinatário. Além disso, o canal seguro garante integridade dos dados, pois o SSH possui métodos para detectar modificações nos dados em trânsito (BARRETT; SILVERMAN, 2001; YLONEN; LONVICK et al., 2006).

Dessa forma, as principais funcionalidades do SSH, conforme Barrett e Silverman (2001), são:

- **Logins remotos seguros:** Verificação de identidade antes de estabelecer a conexão, baseada em rigorosos e modernos algoritmos;
- **Transferência segura de arquivos:** Basicamente a mesma funcionalidade de protocolos legados – como o FTP – porém, mais seguro e confiável, sendo baseado no SFTP;
- **Execução remota de comandos segura:** Controle remoto de computadores por linha de comando, porém, com a segurança característica do protocolo SSH;
- **Controle de acesso:** Capacidade de ceder a determinados usuários apenas certas permissões, tal como em um controlador de domínio.

14.2 IMPORTÂNCIA PARA SISTEMAS EMBARCADOS LINUX

Atualmente, há uma ampliação na adoção de dispositivos inteligentes conectados à internet – os IoT (Internet das Coisas). Esses objetos – que são sistemas embarcados – possuem aplicações em diversas áreas, com uso em, por exemplo: residências, com a automação residencial; indústrias, com a atualização do maquinário; além do uso extensivo em cidades inteligentes. Nesse sentido, o uso desses dispositivos embarcados tende a elevar a conectividade a um outro nível (SSH, 2021a).

No entanto, a hiperconectividade desses sistemas embarcados, sobretudo utilizando a internet como meio principal, implica em sérios riscos de segurança. Ameaças de ataques remotos, controle por pessoas não autorizadas, instalação de softwares maliciosos e espionagem são apenas alguns desses exemplos, o que torna a segurança um fator importantíssimo para o desenvolvimento de dispositivos IoT (SSH, 2021a).

Um dos maiores problemas na construção de dispositivos conectados é que geralmente são muito mal projetados do ponto de vista da segurança. Além disso, até certo tempo atrás, era comum que produtos embarcados carregassem senhas padrão consigo, senha essa compartilhada por todos os dispositivos. Todas essas características são falhas gravas de design que permitem a terceiros obter total controle de milhares de dispositivos, de uma só vez (SSH, 2021a).

Devido a grande importância da IoT para a sociedade, várias medidas foram pensadas pela indústria para mitigar esses problemas. Dentre essas medidas, as duas principais são destacadas: regulações e certificações para os produtos, que garantam um nível mínimo de segurança; e soluções técnicas – como ferramentas pré-instaladas – que fortaleçam a segurança nos dispositivos embarcados (SSH, 2021a).

Os sistemas embarcados, por padrão, são limitados em recursos computacionais, o que implica em carência de proteção contra ataques cibernéticos por si próprios. No entanto, quando baseados em Linux, os sistemas embarcados podem tirar vantagem do acesso seguro à rede característico do sistema operacional – além de outros recursos de segurança no kernel. Além disso, *scripts* e técnicas comumente aplicadas em servidores Linux podem ser transplantadas para esse ambiente, melhorando o nível de segurança de acesso à rede desses dispositivos (MARINUSHKIN, 2015).

A principal ferramenta, em Linux, para garantir a segurança no acesso à redes inseguras é o SSH. No caso de sistemas embarcados, o SSH pode ser usado como servidor para controle e administração remotos, a partir do uso dos protocolos SSH e SFTP, no login seguro e na transferência de arquivos segura, respectivamente (MARINUSHKIN, 2015).

O acesso remoto usando os protocolos SSH e SFTP representam um grande benefício no uso de Linux em sistemas embarcados, pois a necessidade de gerenciamento remoto dos dispositivos IoT é frequente, para análise do estado do mesmo e atualizações de software, por exemplo (MARINUSHKIN, 2015). A principal área de aplicação desse princípio e, consequente,

da ferramenta SSH, é no ambiente industrial, mas pode ser aplicada em outros ambientes, como nas cidades inteligentes, saúde e agricultura (LUETH, 2020).

Devido ao fato do protocolo SSH ser um padrão IFTF, existem várias implementações (programas) diferentes para tal, tanto pagas quanto gratuitas. As duas principais gratuitas são o *OpenSSH* – a canônica *open source* – e o *Dropbear* (YAGHMOUR et al., 2008; MARINUSHKIN, 2015).

O *Dropbear* é uma implementação simplificada do protocolo, voltada para sistemas embarcados. No entanto, ele carece de várias funcionalidades, e não possui algoritmos fortes de proteção. Por outro lado, o *OpenSSH*, pelo fato de ser a implementação oficial do protocolo, é muito mais testada e madura, além de usar como base a biblioteca *OpenSSL*, outro grande recurso de segurança (YAGHMOUR et al., 2008; MARINUSHKIN, 2015). Por esse motivo, neste capítulo será demonstrada a configuração do utilitário *OpenSSH*.

14.3 PREPARAÇÃO PARA A COMPILAÇÃO DO *OPENSSH*

O *OpenSSH* é a principal ferramenta de conectividade para login remoto utilizando o protocolo SSH. Isso se deve ao fato dele oferecer um grande conjunto de recursos de tunelamento seguro, vários métodos de autenticação e opções de configuração sofisticadas, protegendo o serviço em questão contra ataques (SSH, 2021b).

O *OpenSSH* é uma suíte contendo várias ferramentas para administração segura de sistemas, transferência de arquivos e outras comunicações. Essas funcionalidades são oferecidas pelos principais executáveis do pacote: *ssh*, *scp* e *sftp* (SSH, 2021b). Nesse capítulo, serão demonstrados os dois primeiros.

Devido a sua complexidade e robustez, o *OpenSSH* possui dependências; bibliotecas que contém partes das suas funcionalidades. O *OpenSSH* usa como base duas bibliotecas legadas dos sistemas Unix: a *zlib* e a *OpenSSL*, a primeira, para compressão dos dados em trânsito; a segunda, para criação de um canal seguro conforme os padrões SSL. Portanto, antes de compilar o *OpenSSH*, é necessário compilar primeiramente as suas dependências.

É notável o grande número de pacotes de software a serem compilados neste capítulo, o que leva a necessidade de maior organização da compilação, devido ao maior número de arquivos. Portanto, diferentemente das outras compilações feitas até então, que foram lançadas na pasta *board-support* do SDK, dessa vez será feita uma pasta somente para essas compilações – das dependências e do utilitário – além dos arquivos de configuração e compilação. Isso será feito para tornar o ambiente mais organizado e fácil de controlar, já que o processo de compilação dos mesmos também é extenso.

Sendo assim, será feita uma pasta chamada *ssh* dentro do diretório *board-support*. Para essa pasta serão movidos os códigos-fonte, nela será feita a compilação dos três e também serão salvos os *scripts* de compilação de cada um. O processo de criação da pasta é mostrado no

console abaixo.

```
# Criando a pasta "ssh" dentro de board-support
$ mkdir ~/texasSDK/board-support/ssh
```

Uma vez criado o diretório especial, pode-se baixar os códigos-fontes do *zlib*, do *OpenSSL* e do *OpenSSH*. Os três pacotes podem ser baixados diretamente do site oficial de cada um. No console abaixo é demonstrado o método descrito.

```
# Download dos códigos-fonte
$ cd ~/Downloads/

# zlib
$ wget https://zlib.net/zlib-1.2.11.tar.gz
# OpenSSL
$ wget https://www.openssl.org/source/openssl-1.1.1j.tar.gz
# OpenSSH
$ wget https://openbsd.c3sl.ufpr.br/pub/OpenBSD/OpenSSH/portable/
    ↪ openssh-8.4p1.tar.gz
```

Após o download, prossegue-se com a descompactação e transferência para a pasta de desenvolvimento, seguindo os passos já feitos em outros capítulos. Dessa forma, o processo de compilação do *OpenSSH* pode iniciar, começando pelas dependências. O procedimento é demonstrado abaixo.

```
# Extração
$ tar -xf zlib-1.2.11.tar.gz
$ tar -xf openssl-1.1.1j.tar.gz
$ tar -xf openssh-8.4p1.tar.gz

# Movendo os códigos-fonte para a pasta "board-support/ssh"
$ mv zlib-1.2.11 ~/texasSDK/board-support/ssh
$ mv openssl-1.1.1j ~/texasSDK/board-support/ssh
$ mv openssh-8.4p1 ~/texasSDK/board-support/ssh
```

14.4 COMPILAÇÃO DAS DEPENDÊNCIAS: *ZLIB* E *OPENSSL*

A primeira dependência a ser compilada será o *zlib*. Primeiramente é criado um *script* de compilação, cujo nome é *build_zlib.sh*, conforme a prática adotada neste trabalho. O *zlib* é uma biblioteca simples, portanto, os passos de compilação são os mesmos passos básicos de qualquer pacote de software: declaração das variáveis de ambiente, configuração, compilação e instalação. A única especificidade na compilação do *zlib* é a declaração de duas variáveis de arquitetura a mais: CC e AR. O *script* completo da compilação do *zlib* é apresentado na figura 45.

Figura 45 – *script* de compilação da *zlib*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
export CC=arm-linux-gnueabihf-gcc
export AR=arm-linux-gnueabihf-ar

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/ssh/zlib_build"

# definir o caminho dos códigos-fontes
export SOURCE="/home/felipe/texasSDK/board-support/ssh/zlib-1.2.11"

## INÍCIO ##

# abrir a pasta do código-fonte zlib
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar
./configure --prefix=$RAIZ

# compilar
make -j9

# instalar
make -j9 install
```

Fonte: AUTOR (2021)

Na pasta *zlib_build* são lançados os produtos da compilação, demonstrados no console abaixo com o auxílio do comando *tree*. Observa-se que são produzidas três pastas: *include*, *lib* e *share*, para arquivos de cabeçalho, bibliotecas compartilhadas e manuais, respectivamente. Para a compilação (e utilização) do *OpenSSH*, apenas o conteúdo da pasta *lib* será usado.

```
$ cd ~/texasSDK/board-support/ssh/zlib_build && tree
.
|---- include
|    |---- zconf.h
|    +---- zlib.h
|---- lib
|    |---- libz.a
|    |---- libz.so -> libz.so.1.2.11
|    |---- libz.so.1 -> libz.so.1.2.11
|    |---- libz.so.1.2.11
|    +---- pkgconfig
|         +---- zlib.pc
+---- share
     +---- man
          +---- man3
               +---- zlib.3

6 directories, 8 files
```

Após isso, passa-se à compilação da segunda dependência: *OpenSSL*. Mais uma vez, o procedimento é idêntico as compilações anteriores. A única peculiaridade é no momento da configuração: o *OpenSSL* possui um *script* de configuração diferente dos demais, visto que possui suporte a diferentes tipos de arquiteturas e sistemas operacionais e, para cada um, há uma opção específica. No caso dessa aplicação, escolheu-se a *linux-armv4*. O *script* completo da compilação do *OpenSSL* é apresentado na figura 46.

Figura 46 – *script* de compilação da *OpenSSL*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/ssh/openssl_build"

# definir o caminho dos códigos-fontes
export SOURCE="/home/felipe/texasSDK/board-support/ssh/openssl-1.1.1j"

## INÍCIO ##

# abrir a pasta do código-fonte OpenSSL
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar (perceber o Configure iniciando com maiúscula)
./Configure linux-armv4 \
        --prefix=$RAIZ \
        --openssldir=ssl shared

# compilar
make -j9

# instalar
make -j9 install
```

Fonte: AUTOR (2021)

Os produtos da compilação (mais extensa, nesse caso) são salvos na pasta *OpenSSL_build*. Os diretórios produzidos são visualizados abaixo, de forma resumida, com o comando tree. São produzidas as pastas *bin, include, lib, share* e *ssl*. Muitos arquivos de cabeçalho e documentações são produzidos, porém, somente os executáveis (na pasta *bin*) e as bibliotecas compartilhadas (na pasta *lib*) são requeridas para a compilação cruzada do *OpenSSH*.

```
$ cd ~/texasSDK/board-support/ssh/openssl_build && tree -L 2
.
|---- bin
|    |---- c_rehash
|    +---- openssl
|---- include
|    +---- openssl
|---- lib
|    |---- engines-1.1
|    |---- libcrypto.a
|    |---- libcrypto.so -> libcrypto.so.1.1
|    |---- libcrypto.so.1.1
|    |---- libssl.a
|    |---- libssl.so -> libssl.so.1.1
|    |---- libssl.so.1.1
|    +---- pkgconfig
|---- share
|    |---- doc
|    +---- man
+---- ssl
     |---- openssl.cnf
     |---- openssl.cnf.dist
     +---- (...)

13 directories, 12 files
```

14.5 COMPILAÇÃO E INSTALAÇÃO DO UTILITÁRIO *OPENSSH* NO *TARGET*

Com o trabalho com as dependências concluído, pode-se, enfim, iniciar a configuração do utilitário *OpenSSH*. Devido a complexidade dessa etapa, esse tópico será dividido em três partes: a compilação do *OpenSSH*, a transferência dos arquivos para o cartão micro SD e a instalação da ferramenta no *target*.

Em primeiro lugar, é necessário compilar o *OpenSSH* para obter os executáveis, que são as ferramentas *ssh, scp*, e outras. No entanto, é válido destacar que a sua compilação não foi completamente projetada para outras plataformas, ou seja, o utilitário por vezes é hostil à compilação cruzada. Embora esse problema fosse mais presente em versões anteriores (foi usada a versão 8.4p1), ainda assim, é necessário atenção no procedimento, sobretudo no final.

Durante a compilação convencional, o *make*, no momento da instalação (*make install*) tenta executar o programa compilado para geração de chaves – o *ssh-keygen*. Porém, como fora

compilado para uma arquitetura diferente, o mesmo não será executado, acionando um erro.

Dessa forma, o passo da compilação do *OpenSSH* precisa ser alterado no final: após a compilação, com o comando *make -j9*, é necessário fazer a instalação manualmente, sem o auxílio do comando *make install*. Além disso, as chaves do SSH precisam ser geradas também manualmente, o que pode ser feito como o comando *ssh-keygen* do *host*, que vem com o SSH instalado por padrão. O *script* completo da compilação do *OpenSSH* é apresentado na figura 47.

Figura 47 – *script* de compilação do *OpenSSH*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
export CC=arm-linux-gnueabihf-gcc
export AR=arm-linux-gnueabihf-ar

# definir o caminho da pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/ssh/openssh_build"

# definir o caminho dos códigos-fontes
export SOURCE="/home/felipe/texasSDK/board-support/ssh/openssh-8.4p1"

# definir o caminho da compilação do zlib
export SSL_BUILD="/home/felipe/texasSDK/board-support/ssh/openssl_build"

# definir o caminho da compilação do openssl
export ZLIB_BUILD="/home/felipe/texasSDK/board-support/ssh/zlib_build"


## INÍCIO ##

# abrir a pasta do código-fonte OpenSSH
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar
./configure \
        --host=arm-linux-gnueabihf \
        --with-libs \
        --with-zlib=$ZLIB_BUILD \
        --with-ssl-dir=$SSL_BUILD \
        --disable-etc-default-login

# compilar
make -j9

# gerar chaves
ssh-keygen -t rsa -f ssh_host_rsa_key -N ""
ssh-keygen -t dsa -f ssh_host_dsa_key -N ""
```

Fonte: AUTOR (2021)

Em segundo lugar, é necessário transferir o resultado da compilação para o *target*. Nesse caso são vários arquivos: executáveis, bibliotecas compartilhadas e arquivos de configuração. Por conta disso, aliado ao fato de ser um processo manual – pela falta do *make install* – é uma

boa prática separar, em um outro *script*, a função de separar e copiar os arquivos para o destino, principalmente para organização. Portanto, foi criado o *script transfer.sh*, contendo os comandos de cópia para cada arquivo necessário.

O primeiro passo é declarar, como variáveis do *shell*, os locais de cada diretório a ser usado – as dependências, o utilitário e o destino, a fim de facilitar a escrita do *script*. Nesse caso, foi criada uma pasta de destino dos arquivos, chamada de *transferencia-teste*, para servir como a pasta "*build*" das compilações anteriores. Vale destacar que o alvo poderia ser diretamente o cartão de memória, porém, optou-se por uma cópia no *host*. A demonstração desse passo é mostrada abaixo.

```
SOURCE="/home/felipe/texasSDK/board-support/ssh/openssh-8.4p1"
SSL_BUILD="/home/felipe/texasSDK/board-support/ssh/openssl_build"
ZLIB_BUILD="/home/felipe/texasSDK/board-support/ssh/zlib_build"
ROOTFS="/home/felipe/texasSDK/board-support/ssh/transferencia-teste"
```

Após isso, montam-se, no destino, as pastas nas quais serão copiados os arquivos. O procedimento é feito com o comando *mkdir*. Isso é feito para que o *script* não para caso as pastas não existam e, caso existam, o comando passará adiante. Esse passo é mostrado no console abaixo.

```
mkdir $ROOTFS/usr/
mkdir $ROOTFS/usr/{lib,bin,sbin,libexec}
mkdir $ROOTFS/usr/local/
mkdir $ROOTFS/usr/local/{bin,etc}
```

Uma vez que as pastas estejam prontas, pode-se começar a transferência, inicialmente, pelas dependências. Embora sejam gerados vários arquivos, apenas alguns são necessários, conforme se observa no *shell* abaixo.

```
# copiar zlib
cp -a $ZLIB_BUILD/lib/libz*.so* $ROOTFS/usr/lib

# copiar openssl
cp -a $SSL_BUILD/lib/lib*.so* $ROOTFS/usr/lib
cp $SSL_BUILD/bin/c_rehash $ROOTFS/usr/bin
cp $SSL_BUILD/bin/openssl $ROOTFS/usr/bin
```

Por fim, os arquivos do *OpenSSH* podem ser copiados. Os executáveis, bibliotecas e arquivos de configuração são transferidos, cada um para sua pasta. Vale destacar que a estrutura de diretórios obedecida foi a da pasta */usr*, conforme o padrão de instalação do *make*. Esse passo é mostrado abaixo, e o *script* completo da transferência do *OpenSSH* é apresentado na figura 48.

```
cp $SOURCE/sshd $ROOTFS/usr/sbin
cp $SOURCE/{scp,sftp,ssh,ssh-add,ssh-agent,ssh-keygen,ssh-keyscan} \
$ROOTFS/usr/local/bin/
cp $SOURCE/{sftp-server,ssh-keysign,moduli} $ROOTFS/usr/libexec/
cp $SOURCE/{sshd_config,ssh_config} $ROOTFS/usr/local/etc/
cp $SOURCE/ssh_host_* $ROOTFS/usr/local/etc/
```

Figura 48 – *script* de transferência do *OpenSSH*

```
# definição de variáveis de ambiente

SOURCE="/home/felipe/texasSDK/board-support/ssh/openssh-8.4p1"
SSL_BUILD="/home/felipe/texasSDK/board-support/ssh/openssl_build"
ZLIB_BUILD="/home/felipe/texasSDK/board-support/ssh/zlib_build"
ROOTFS="/home/felipe/texasSDK/board-support/ssh/transferencia-teste"


# construindo pastas alvo, caso não existam
mkdir $ROOTFS/usr/
mkdir $ROOTFS/usr/{lib,bin,sbin,libexec}
mkdir $ROOTFS/usr/local/
mkdir $ROOTFS/usr/local/{bin,etc}

# copiar zlib
cp -a $ZLIB_BUILD/lib/libz*.so* $ROOTFS/usr/lib

# copiar openssl
cp -a $SSL_BUILD/lib/lib*.so* $ROOTFS/usr/lib
cp $SSL_BUILD/bin/c_rehash $ROOTFS/usr/bin
cp $SSL_BUILD/bin/openssl $ROOTFS/usr/bin

# COPIAR SSH
cp $SOURCE/sshd $ROOTFS/usr/sbin
cp $SOURCE/{scp,sftp,ssh,ssh-add,ssh-agent,ssh-keygen,ssh-keyscan} \
$ROOTFS/usr/local/bin/
cp $SOURCE/{sftp-server,ssh-keysign,moduli} $ROOTFS/usr/libexec/
cp $SOURCE/{sshd_config,ssh_config} $ROOTFS/usr/local/etc/
cp $SOURCE/ssh_host_* $ROOTFS/usr/local/etc/
```

Fonte: AUTOR (2021)

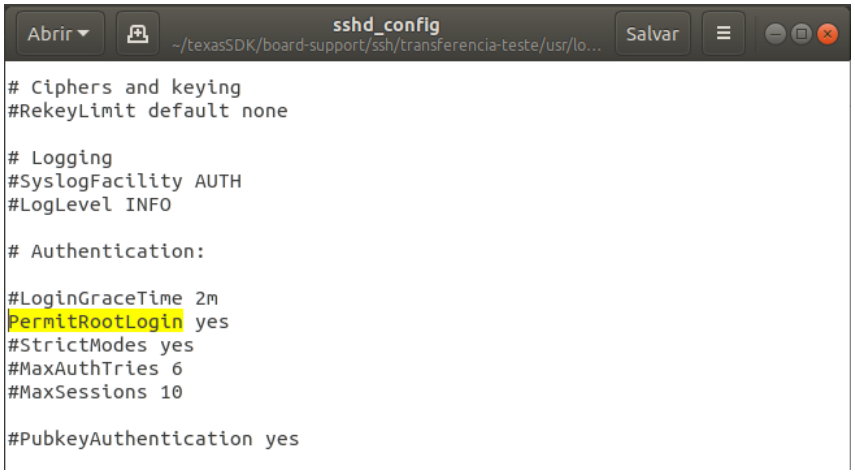
Em terceiro lugar, é necessário instalar o *OpenSSH* no *target*. Embora nas compilações anteriores os pacotes eram instalados pelo simples ato de cópia, nesse caso e no do próximo capítulo, é necessário executar alguns comandos dentro da placa, ou seja, a instalação possui uma segunda etapa, dessa vez no dispositivo alvo.

O processo de instalação de um utilitário como o *OpenSSH*, devido a sua complexidade (e necessidade), possui muitos passos, como habilitar o serviço no *systemd* para que o *ssh* seja sempre iniciado junto com o sistema, por exemplo. Esse tipo de tarefa costuma ser feita pelo gerenciador de pacotes do sistema. Como tal recurso não está disponível aqui, optou-se por fazer uma instalação mais simples, apenas para demonstração.

O passo essencial para instalar o *OpenSSH* é criar um usuário específico para esse utilitário, chamado “ssh”. Esse é um usuário especial, que não serve para login; é usado para prevenir ataques por conta de privilégios, por exemplo, do *root*. Programas que fazem uso de conexão remota costumam fazer uso desse recurso, sobretudo servidores web. Portanto, o objetivo é criar um usuário separado para o *ssh*.

Porém, antes de transferir, é necessário alterar o arquivo de configuração do *daemon* SSH, o *sshd_config*. Esse arquivo vem configurado por padrão para impedir o login do usuário *root* – mesmo com a senha certa – por segurança. Para evitar a criação de outro usuário – já que o *target* só possui o *root*, é necessário corrigir a opção *PermitRootLogin* para *yes*. O resultado fica conforme a imagem 7.

Figura 49 – Arquivo *sshd_config* corrigido



Fonte: AUTOR (2021)

Depois de transferidos os arquivos e com a placa já iniciada, cria-se o usuário “ssh”. Primeiramente, cria-se o diretório *home* do mesmo, para, enfim, utilizar os comandos *groupadd* e *useradd* para a criação do grupo e usuário “ssh”, respectivamente. Para facilitar esse processo, foi feito um *script* – *install_ssh.sh* – que é apresentado na figura 8.

Figura 50 – *script* de instalação do *OpenSSH*

```
mkdir /var/empty
mkdir /var/empty/sshd
groupadd -g 35 sshd
useradd -u 35 -g 35 -c "sshd privsep" -d /var/empty/sshd -s /sbin/
```

Fonte: AUTOR (2021)

14.6 UTILIZAÇÃO DO SSH COMO *SHELL* REMOTO

Com a instalação do *OpenSSH* concluída, pode-se passar para a demonstração das suas ferramentas; em especifico, a de login remoto e a de transferência de arquivos. Nesse tópico, será demonstrada a configuração do *host* e do *target* para conexão via Ethernet, a inicialização do *daemon ssh* e a conexão remota, além do uso remoto em si.

A conexão SSH funciona sobre as camadas TCP/IP, ou seja, é necessário garantir que há comunicação de rede entre os dois pontos antes de iniciar a ferramenta. Portanto, antes de iniciar o *daemon ssh* ou sequer configurar o IP em cada dispositivo, é necessário garantir a comunicação Ethernet no meio físico.

A placa BeagleBone Black possui apenas a interface de rede cabeda disponível, o que requer um cabo par trançado para a sua conexão. Tal cabo fora listado nos materiais necessários, na Metodologia (tópico 3.1). Até o momento, esse cabo não fora utilizado, pois não houve a necessidade de comunicação via rede entre *host* e *target*. Porém, agora que uma aplicação exige essa conexão (assim como as próximas), é necessário utilizá-lo.

Com a placa já ligada e o sistema já iniciado, conecta-se o cabo no *target* e, logo em seguida, ao *host*. Os LEDs da interface de rede da placa, um verde e um amarelo, devem acender, indicando que a conexão de nível físico – *Fast Ethernet* – está estabilizada.

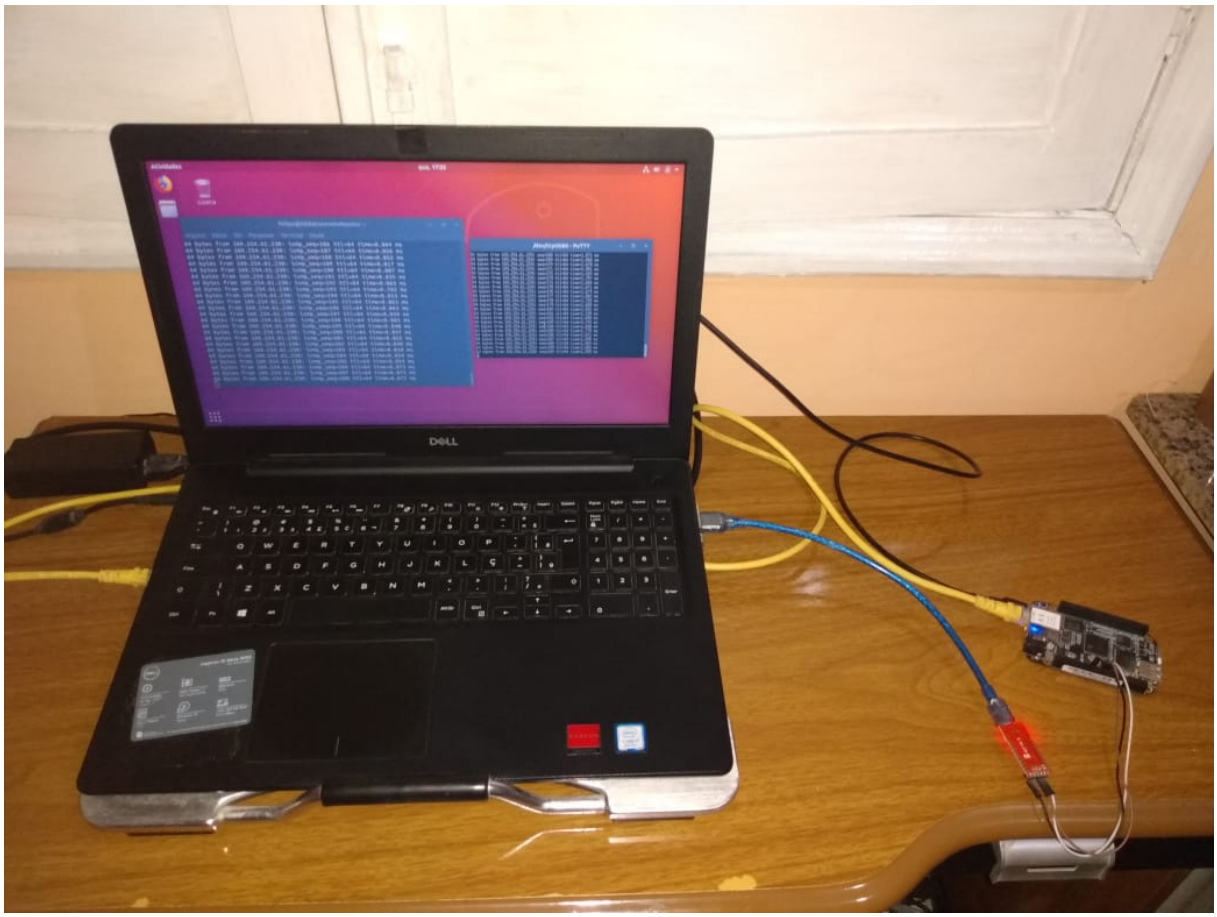
Com o cabo conectado, é necessário configurar o IP de cada ponto da conexão. Como o *host* não possui um servidor DHCP na interface cabeada, é preciso habilitar o IP e máscara de ambos manualmente com o comando *ip*. Para o *target*, o comando é o mostrado logo abaixo, no primeiro console; para o host, o comando é o mostrado logo a seguir, no segundo console.

```
1 $ ip ad add 169.254.61.230/16 dev eth0
```

```
1 $ sudo ip ad add 169.254.61.229/16 dev enp2s0
```

Desse modo, a conexão Ethernet, tanto física quanto virtual, está configurada. É possível testar a conectividade com o comando *ping*. A organização da bancada de trabalho, dessa vez completa, é apresentada na figura 51.

Figura 51 – Bancada de desenvolvimento com o cabo Ethernet



Fonte: AUTOR (2021)

Assim, usando a interface serial, inicia-se o SSH no *target*. A maioria dos programas utilitários em Linux precisa de um *script* (geralmente do *systemd*) para ser iniciado como um *daemon*, pois, se iniciados diretamente (isto é, pelos seus executáveis), esses tendem a travar o *shell*. Esses *scripts*, então, os executam como *threads*, com o uso do parâmetro "&" (ou outros recursos).

O *OpenSSH* já contém um mecanismo para iniciar o ssh como um *daemon*, é o executável *sshd*. Para iniciá-lo, é necessário inserir o seu caminho completo – no caso, */usr/sbin/sshd*. Esse executável possui diversos parâmetros opcionais, os quais constam na documentação. Para esse trabalho, utilizou-se apenas para definir o arquivo de configuração (*sshd_config*) e a chave pública RSA (*ssh_host_rsa_key*). A demonstração desse comando se encontra abaixo.

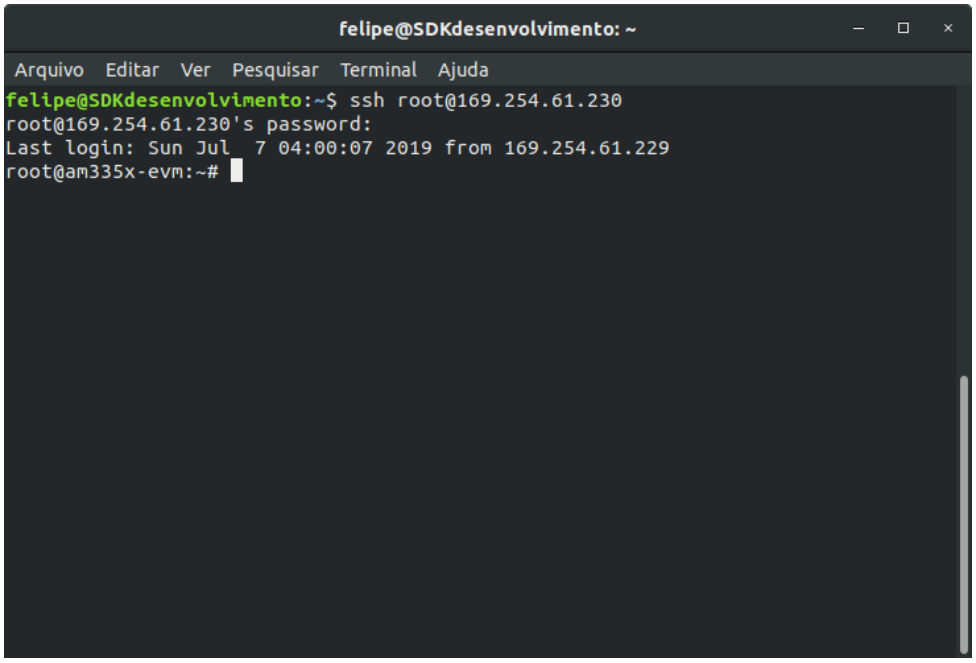
```
1 $ /usr/sbin/sshd -f /usr/local/etc/sshd_config -h /usr/local/etc/
    ↪ ssh_host_rsa_key
```

Com isso, o *host* pode, enfim, conectar-se ao *target* utilizando o SSH. O comando de conexão obedece ao seguinte padrão: *ssh usuário@IP*, sendo "usuário" o nome do usuário pelo qual se deseja conectar. Como a placa não possui outros usuários além do *root* e, como a conexão

usando o *root* foi liberada anteriormente, pode-se conectar ao *target* utilizando o comando abaixo. Se tudo ocorrer bem, a interface deve ser como a mostrada na figura 52.

```
$ ssh root@169.254.61.230
```

Figura 52 – Conexão SSH entre *host* e *target* sucedida



Fonte: AUTOR (2021)

Uma vez que o login remoto tenha sido feito, qualquer comando que esteja dentro da autoridade do usuário logado pode ser executado. Nesse caso, como foi feita a conexão usando o *root*, qualquer comando pode ser dado. Para fins de demonstração, o comando uname -a prova que o terminal está na placa, conforme mostrado abaixo.

```
root@am335x-evm:~# uname -a
Linux am335x-evm 4.19.38-g4dae378bbe #1 PREEMPT Sun Jul 7 04:39:33
    ↪ UTC 2019 armv7l GNU/Linux
```

Praticamente qualquer comando pode ser executado nesse estado, o que prova o potencial do SSH como ferramenta para gerenciamento remoto de sistemas embarcados. Apesar disso, não é recomendável permitir o acesso do *root*, devido ao seu poder total sobre o sistema; antes, é uma boa prática construir usuários para cada necessidade, com permissões específicas apenas para a sua tarefa. Por fim, a conexão é encerrada com o comando abaixo.

```
root@am335x-evm:~# logout
Connection to 169.254.61.230 closed.
```

14.7 TRANSFERÊNCIA DE ARQUIVOS UTILIZANDO *SCP*

Além do login remoto, por vezes é necessário transferir arquivos para o *target* de forma rápida e segura, seja para testes ou atualizações. Não importa se a placa estiver no laboratório ou se estiver em campo; o melhor método para fazer isso é por meio do protocolo SFTP, presente no utilitário *OpenSSH* nos comandos *sftp* (implementação direta) e *scp* (mais simplificado).

A fim de testar essa funcionalidade, será criado um arquivo simples de texto no *host* para ser transferido ao *target*. Primeiramente, no *host* abre-se um terminal e cria-se um arquivo *test.txt*, e insere-se o conteúdo "teste scp" nele. Tal procedimento pode ser feito a partir do primeiro dos comandos no console abaixo. Após, percebe-se que o processo foi sucedido a partir do comando *cat*.

```
$ echo teste scp > test.txt
$ cat test.txt
teste scp
```

O próximo passo é transferir esse arquivo. O padrão do comando *scp* é: *scp arquivo usuário@IP:diretório*. Nesse caso, diretório diz respeito ao local onde o arquivo será escrito no destino, referente ao sistema de arquivos do mesmo.

É válido ressaltar que, assim como no anterior, o *scp* leva em conta o usuário que está executando a transferência; nesse caso, a cópia só funcionará se o respectivo usuário logado tiver permissão de escrita no diretório alvo. Como se está usando o *root*, a escrita pode ser feita em qualquer local do sistema de arquivos. Sendo assim, a transferência ocorre pelo comando abaixo.

```
$ scp test.txt root@169.254.61.230:/home
root@169.254.61.230's password:
test.txt                                      100%   10     4.4KB/s   00:00
```

Para demonstrar a transferência, acessa-se a placa pela interface serial. Como o arquivo foi salvo na pasta home, os comandos *ls* e *cat* demonstram o resultado positivo do teste. Vale ressaltar que um arquivo transferido pelo comando *scp* possui como proprietário aquele que fora usado para login, no caso, o *root*. A demonstração do arquivo presente no *target* está no console abaixo.

```
$ cd /home
$ ls -l
drwx------    3 root     root          4096 Jul  7 03:11 root
-rw-r--r--    1 root     root            10 Jul  7 03:17 test.txt
$ cat test.txt
teste scp
```

15 CONSTRUÇÃO DE UMA INTERFACE WEB UTILIZANDO *LIGHTTPD*

É vital que o desenvolvedor de sistemas embarcados Linux considere de que modo a interação com o usuário será feita no seu dispositivo. Existem diversos meios, e as interfaces web se popularizaram muito nos últimos tempos. Dessa forma, o objetivo deste capítulo é mostrar a necessidade de um servidor web em um sistema embarcado Linux e demonstrar uma aplicação de configuração e monitoramento remotos como exemplo.

Assim sendo, este capítulo contém: contextualização sobre interfaces web (tópico 15.1), estrutura de uma aplicação web embarcada (15.2), comentário sobre servidores web (15.3), preparação para a compilação do *lighttpd* (15.4), compilação e instalação (15.5) e, por último, demonstração de uma página exemplo (15.6).

15.1 INTERFACES WEB EM SISTEMAS EMBARCADOS

Com o avanço da maturação da internet (e das redes TCP/IP) como tecnologia de comunicação, modelos antigos, tais como alguns protocolos industriais – caros e de alta especialidade – foram, aos poucos, sendo substituídos pelos padrões Ethernet e Wi-Fi (LIU; CHENG, 2010). Na mesma medida, *browsers* web se tornaram a interface de usuário da maioria das aplicações, devido a facilidade de desenvolvimento, além de ceder uma interface gráfica para vários clientes ao mesmo tempo, sem necessidade de instalação (JU; CHOI; HONG, 2000). Nesse sentido, um número crescente de tecnologias da Web também podem ser aplicadas ao gerenciamento de elementos de rede (JU; CHOI; HONG, 2000).

Assim sendo, elementos de uma rede podem ser gerenciados, configurados e monitorados por meio de uma GUI, usando um navegador da web. Isso pode ser obtido instalando um software servidor web no equipamento ou máquina que se queira gerenciar e desenvolvendo uma página web utilizando HTML, gráficos e outros elementos comuns a *browsers*. Desse modo, as informações podem ser trocadas por meio do protocolo HTTP; isto é, requerendo páginas e enviando formulários (JU; CHOI; HONG, 2000).

Por outro lado, no mundo dos sistemas embarcados, a evolução das capacidades e das aplicações dos dispositivos aumentou a complexidade de controle dos mesmos. No entanto, interfaces de controle de sistemas embarcados geralmente são difíceis de construir, o que também aumenta o custo. Devido a isso, dispositivos complexos saem de fábrica com pobres e sobrecarregadas interfaces. Além disso, a maioria dos sistemas embarcados não pode contar com *displays* – que são mais fáceis para o usuário – por exemplo. A necessidade de interfaces mais informativas e intuitivas, portanto, se faz necessária (FILIBELI; OZKASAP; CIVANLAR, 2007).

Nesse sentido, a simplicidade no uso dos padrões TCP/IP, atrelados à facilidade cedida

pela tecnologia web permite que sejam desenvolvidos sistemas embarcados com servidores web embutidos. Essa prática permite que esses dispositivos possam ser gerenciados pela rede, a partir de uma interface de usuário baseada em GUI que, por sua vez, é amigável, barata e multiplataforma (JU; CHOI; HONG, 2000).

Uma prova disso é o fato de que uma das maiores tendências em sistemas embarcados habilitados para rede é a inclusão de um servidor web (HTTP). Isso ocorre pois as aplicações básicas de tal solução – gerenciamento remoto e visualização gráfica de dados – são ainda mais importantes no desenvolvimento e utilização da maior parte dos sistemas embarcados, potencializando, assim, os benefícios do uso dessa solução nos mesmos (YAGHMOUR et al., 2008; JU; CHOI; HONG, 2000).

Logo, cada dispositivo pode fornecer sua própria interface como uma página web na rede. Essas páginas podem conter imagens, áudios, textos, e muitos outros recursos. Se combinados, esses podem ser usados para uma infinidade de funcionalidades e configurações, como checagem do status, depuração dos pinos, diagnósticos e até atualizações remotas. Além disso, essa solução permite que os sistemas embarcados possam ser gerenciados remotamente, de qualquer dispositivo que contenha um navegador web, facilitando a utilização dos mesmos e promovendo cada vez mais a Internet das Coisas (FILIBELI; OZKASAP; CIVANLAR, 2007; YAGHMOUR et al., 2008).

O uso de servidores web em sistemas embarcados traz consigo diversas vantagens, principalmente relacionadas a economia em gastos com desenvolvimento e manutenção. Basicamente, é possível enquadrá-las em: independência da aplicação, gerenciamento multiplataforma, utilização mais acessível do dispositivo, facilidade do desenvolvimento, economia com gastos e benefícios para usuários e desenvolvedores.

Ao embutir uma GUI no próprio dispositivo, o mesmo torna-se independente; isto é, essa solução dispensa a necessidade de instalação de um software cliente na máquina do usuário, visto que o acesso se faz mediante um navegador web comum. Além disso há, também, vantagens em relação ao versionamento, visto que não há riscos de o usuário possuir uma versão anterior da aplicação que não suporte todos os recursos. Consequentemente, qualquer atualização futura na aplicação precisa ocorrer apenas do lado do sistema embarcado (JU; CHOI; HONG, 2000).

Há, além dessas vantagens, a utilização de uma interface gráfica multiplataforma, isto é, que independe de sistema operacional – software – ou arquitetura computacional – hardware – para ser operada. Isso é possível pois as interfaces web são acessadas por navegadores web, disponíveis em praticamente todos os computadores (JU; CHOI; HONG, 2000; LIU; CHENG, 2010).

Ao embutir um servidor web em um sistema embarcado, arquivos e aplicações tornam-se acessíveis pela rede – tal qual no SSH. No entanto, a interface web embarcada possui como vantagem a limitação de recursos e opções configuráveis pelo usuário, ou seja, usuários não

especializados em console podem ajustar dispositivos com o auxílio de uma interface gráfica. Um exemplo disso são os roteadores, que possuem uma página de configuração pela qual se ajustam parâmetros de rede. Tal interface permite que sejam feitas apenas configurações permitidas no aparelho, limitando falhas (JU; CHOI; HONG, 2000).

Pelo fato de ser baseada em uma linguagem de marcação de hipertexto (HTML), a página pode ser mais facilmente desenvolvida, dado que sua interface pode ser prototipada com facilidade, e a atualização da mesma pode ser feita com a importação de poucos arquivos, visto que os mesmos (HTML, CSS E Javascript) são renderizados no navegador (JU; CHOI; HONG, 2000).

Além disso, a solução web em questão pode poupar o dispositivo de vir com um *display* (LCD, OLED, HDMI, etc.) embarcado. Isso é vantajoso na medida em que GUIs para sistemas operacionais são complexas para desenvolver, e geralmente consomem muitos recursos computacionais (tal qual o GNOME, por exemplo). Ademais, o uso de *displays* acarreta um alto consumo de corrente para um sistema embarcado; cerca de 25% do consumo total (PRAKASH; SHIN, 2013). Dessa forma, uma interface web embarcada gera economia de gastos no desenvolvimento e no consumo de energia.

Por fim, interfaces web embarcadas oferecem benefícios tanto para usuários quanto para desenvolvedores. Para esses, o acesso e o uso é mais fácil e permite aquisições de informações em tempo real. Para estes, o uso de uma tecnologia web pode poupar muito tempo de desenvolvimento, visto que há bastante documentação disponível, o desenvolvimento web está bastante maduro e presente no mercado a anos, os *frameworks* web são robustos e as páginas em si são mais simples de construir do que interfaces de aplicações desktop, por exemplo (JU; CHOI; HONG, 2000).

15.2 ESTRUTURA DE UMA APLICAÇÃO WEB EMBARCADA

Para demonstrar a construção de uma interface web para um sistema embarcado – etapa essa que envolve desde a compilação do software servidor até a demonstração – é necessário, primeiramente, demonstrar a arquitetura de uma solução web para Linux embarcado. Porém, antes de fazê-lo, é importante definir os conceitos de servidor web e servidor de aplicação, e como eles se complementam para ceder a funcionalidade necessária a interface. Após isso, é possível comentar a estrutura de uma aplicação web embarcada, que possui as suas particularidades em relação a soluções web comuns.

Um servidor web é um software instalado no lado “servidor” do conjunto, responsável por aceitar requisições de clientes (representados pelos *browsers*) e enviar documentos estáticos do site (arquivos HTML, imagens, planilhas, entre outros), tudo mediante o protocolo HTTP (REALTIMELOGIC, 2021). Os principais exemplos disponíveis atualmente no mercado são os servidores *Apache*, *Ngnix*, *lighttpd* e *Boa*.

Embora a web atualmente seja bastante complexa, com páginas dinâmicas, com gráficos e *streaming* de media, tudo o que o servidor web pode fazer é responder ao cliente com o HTTP. Pelo fato de o protocolo HTTP ser considerado sem estado (*stateless*), não há inteligência nas operações, isto é: o servidor web não pode tomar decisões e nem executar programas. Além disso, o mesmo está limitado a interfaces estáticas, já que precisaria que outro *script* o fizesse (REALTIMELOGIC, 2021).

Em sites antigos, era comum a utilização do CGI (*Common Gateway Interface*) para mitigar esse problema. Sua utilização era simples:uma requisição GET ou POST executava um *script* no servidor tomando os dados recebidos como parâmetros. Esse método é inapropriado para sistemas embarcados, visto que é lento e pesado, além de abrir diversas brechas de segurança (REALTIMELOGIC, 2021).

Além desses impasses, há o fato de que uma interface web para um sistema embarcado geralmente visa o controle remoto do mesmo, o que significa permitir a funcionalidade web afetar o funcionamento do dispositivo, lidando diretamente com o sistema operacional. Como um servidor web não tem tal capacidade nativa, seria necessário desenvolver uma solução semelhante ao CGI, que receba requisições do servidor web e execute programas separadamente, cada um para uma função específica. Esse modelo, devido a sua complexidade, é praticamente inviável no desenvolvimento de um sistema embarcado (REALTIMELOGIC, 2021).

Para lidar com esses entraves, são utilizados os servidores de aplicação web. Eles funcionam ao lado dos servidores web, expandindo-lhes as capacidades, pelo fato de serem construídos sobre uma linguagem de programação. Em síntese, eles permitem decisões em tempo real, construção da regra de negócio, além de criação de páginas dinâmicas (REALTIMELOGIC, 2021). As principais linguagens utilizadas para a construção de servidores de aplicação web são Javascript, Python e C#.

Os servidores de aplicação resolvem o problema dos servidores web, pois cedem recursos que permitem operações complexas sobre a mesma camada HTTP. Além disso, há a vantagem do uso de *frameworks*, que a maioria das linguagens fazem. Esses poupam muito tempo e recursos, pois já contam com diversas bibliotecas. Dessa forma, o trabalho em baixo nível já está pronto, logo, o desenvolvimento pode focar na adaptação do *framework* (REALTIMELOGIC, 2021).

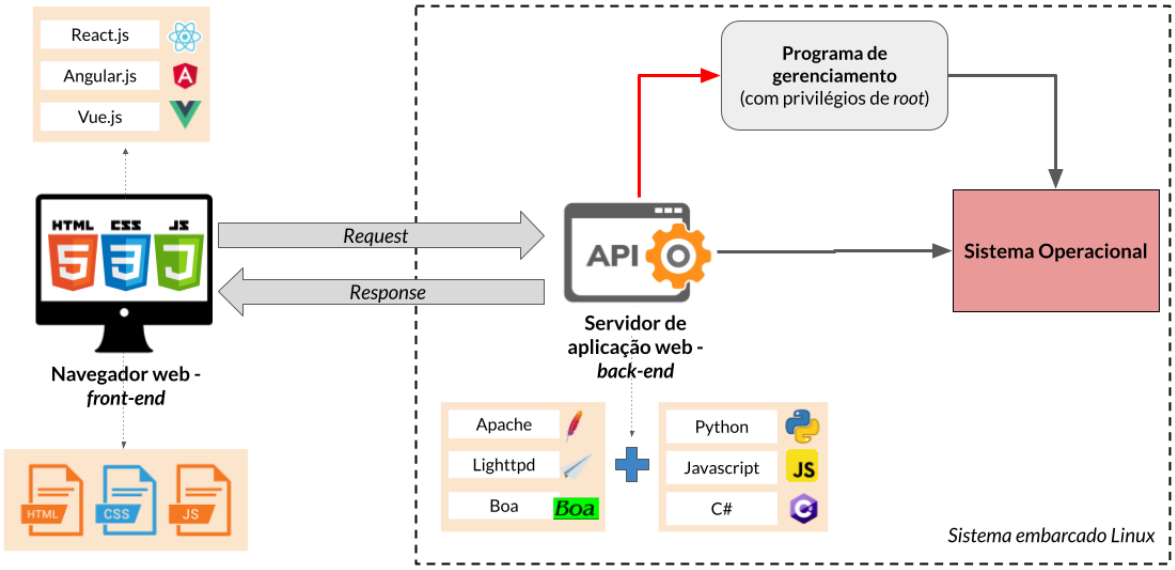
Portanto, as aplicações web atuais – que são a junção da interface de usuário e da regra de negócio – são constituídas, principalmente, de servidores web – para responder requisições via HTTP – e servidores de aplicação – para lidar com processamento e funcionalidades. No caso de arquiteturas (isto é, soluções) para sites e empresas, geralmente há, ainda, a adição de um banco de dados. Porém, no caso de sistemas embarcados, o objetivo é gerenciar o dispositivo remotamente, configurando parâmetros do sistema operacional. Dessa forma, mais elementos devem ser adicionados a essa arquitetura.

Embora existam arquiteturas e modelos de soluções para a web – principalmente em

ambientes desktop – não há uma solução definitiva, pois cada caso é único. Isso é ainda mais visível em ambientes embarcados, nos quais as necessidades mudam a cada projeto. Logo, é esperado que hajam diversos modelos de aplicações web para os mesmos.

Porém, levando em consideração modelos modernos de desenvolvimento web, assim como requisitos comuns a diversos sistemas embarcados microprocessados, é possível chegar a um consenso de arquitetura web embarcada. Tal modelo é apresentado na figura 53, e comentado logo em seguida.

Figura 53 – Diagrama da arquitetura de uma aplicação web embarcada



Adaptado de RealTimeLogic (2021), Ju, Choi e Hong (2000), Wang et al. (2020)

A primeira parte de uma aplicação web é o *front-end*. É representado pela página em si, que é renderizada no navegador web. Essa página contém uma interface gráfica para utilização que, em sistemas embarcados, geralmente serve para configuração de parâmetros do dispositivo, além de obtenção de dados do sistema em tempo real, como sensores. O modo de desenvolvimento do *front-end* é geralmente baseado em uma página única, também chamada de SPA (*Single Page Aplication*).

No lado do sistema embarcado, o servidor de aplicação web lida com as requisições, tratando das chamadas mais baixas do protocolo HTTP – *requests* e *responses* – e lidando com processamento e controle, funcionalidade essa que faz com que o mesmo seja chamado de *back-end*.

Como a função básica de uma interface web embarcada é a configuração remota, é esperado que por meio da mesma possam-se alterar parâmetros do sistema operacional, críticos ou não, a partir do servidor de aplicação (que é construído em uma linguagem de programação). No entanto, algumas configurações (como a de rede) precisam de privilégios de *root* para serem alteradas, visto que estão associadas ao kernel.

Nesse caso, é uma boa prática criar um programa gerenciador com privilégios de *root*, que funciona como um intermediário para o sistema operacional em configurações críticas, tais como as de IP e outros parâmetros do kernel. Esse programa, que pode ser desenvolvido em qualquer linguagem – mas preferencialmente em C, C++ ou *shell* – é executado na inicialização do sistema, funcionando como um *daemon*.

Dessa forma, a arquitetura proposta combina soluções modernas, pois usa métodos e *frameworks* da web atual (e não antigos e inseguros, como o CGI); segurança, pois contém camadas de proteção e boas práticas de desenvolvimento; além de performance, pois foi feita visando as necessidades mais comuns relacionadas a sistemas embarcados Linux.

15.3 SERVIDORES WEB E O *LIGHTTPD*

As interfaces web de gerenciamento podem ser encontradas em vários tipos de sistemas embarcados, especialmente nos microcontrolados e nos microprocessados. No caso dos primeiros, existe uma grande restrição no que diz respeito aos recursos que podem ser usados pela aplicação web, logo, as possibilidades de configuração são, em geral, limitadas. Por outro lado, os últimos, pelo fato de contarem com um sistema operacional, permitem a instalação de um servidor web completo, poupando o desenvolvedor de detalhes de implementação e gerenciando os recursos computacionais – principalmente de rede – de forma mais eficiente.

No caso deste trabalho, o uso de Linux como sistema operacional permite que programas servidores web tradicionais possam ser utilizados praticamente da mesma forma que seriam utilizados em ambientes desktop e servidores, por exemplo. Além disso, a BeagleBone Black, como uma placa de desenvolvimento com recursos robustos, permite que sejam implementadas diversas funcionalidades na aplicação web, sem perda de desempenho.

No entanto, é válido destacar que nem todos os servidores web se enquadram nesse cenário. Alguns sistemas embarcados possuem recursos muito limitados, logo, precisam de um servidor web bastante leve e direcionado apenas a responder requisições básicas. Nesse sentido, existem, basicamente, dois tipos principais de servidores web para Linux embarcado: os de grande porte (de servidores ou datacenters) e os de pequeno porte (para sistemas embarcados).

Os servidores web de grande porte são assim chamados pois são projetados para lidar com muitas requisições ao mesmo tempo; às vezes milhares. Além disso, esses softwares também contam com diversos recursos de segurança e autenticação. Por conta disso, geralmente eles são aplicados em máquinas robustas, como *datacenters*. Os principais representantes dessa categoria são o *Apache* e o *Ngnix*.

No entanto, esses servidores, na maioria das situações, não se enquadram nos requisitos de um sistema embarcado. Tomando como exemplo o Apache, dois problemas são destacados: o mesmo é conhecido como um software bastante extenso e pesado, mesmo para computadores normais. Além disso, ele não é projetado para compilação cruzada, logo, a sua adaptação e

configuração em um sistema embarcado seria muito custosa (YAGHMOUR et al., 2008).

Os servidores web de pequeno porte, por outro lado, são utilizados em sistemas computacionais pouco ou muito limitados, com necessidades também restritas. Essas necessidades geralmente estão em torno de servir apenas um pequeno número de clientes e de fornecer interfaces simples. Por conta disso, eles se popularizaram no desenvolvimento de sistemas embarcados, já que esses geralmente são os requisitos dos mesmos (YAGHMOUR et al., 2008). Dois dos principais exemplos dessa categoria são o *Boa* e o *thttpd*.

Apesar disso, existem situações nas quais essa categoria de servidores não se enquadra perfeitamente. Por exemplo, o *Boa* não possui suporte a um servidor de aplicação, ficando limitado ao CGI. Além disso, ele não possui nenhuma função de autenticação, perdendo segurança (DOOLITTLE; NELSON, 2000; YAGHMOUR et al., 2008). Essas e outras características podem limitar a aplicação do mesmo em um roteador, por exemplo, que faz uso dessas funcionalidades em geral.

Assim, é preciso, para a situação proposta neste trabalho, de um servidor web que contenha as melhores partes das duas categorias citadas. Um exemplo que se encaixa perfeitamente nesses quesitos é o *lighttpd*. Esse servidor web, voltado para alta performance, implementa as funções básicas de segurança e leveza, isto é, ele implementa funcionalidades de servidores robustos, como autenticação, porém, é bastante leve, de fácil configuração e favorável à compilação cruzada (LIGHTTPD, 2020).

O *lighttpd* já possui um uso extenso em sistemas embarcados devido a seu baixo consumo de recursos (LINUX.COM, 2015). Uma prova disso é o sistema de arquivos *rootfs* do SDK, que utiliza o *lighttpd* como servidor web padrão para sua página de configuração. Nesse sentido, o servidor web escolhido para esse trabalho e que será abordado nesse capítulo será o *lighttpd*. Será demonstrada a sua compilação, instalação e configuração no *target*. No fim, uma página web será demonstrada como exemplo de interface de configuração, porém, não será implementado o servidor de aplicação nem a funcionalidade de gerenciamento; apenas o *front-end* será mostrado.

15.4 PREPARAÇÃO PARA A COMPILAÇÃO

Uma das vantagens do *lighttpd* é a sua portabilidade: ele é de fácil compilação cruzada se comparado a outros servidores web de grande porte, fator esse que aumenta o seu potencial de uso em sistemas embarcados Linux. Embora seja baseado em módulos (como o Apache), que são bibliotecas separadas que aumentam as suas funcionalidades, nenhuma é realmente obrigatória para a sua compilação, o que o torna ajustável para as necessidades do dispositivo.

Apesar disso, é altamente recomendável pela documentação (LIGHTTPD, 2020) que a compilação seja feita juntamente da biblioteca *pcre*. Essa biblioteca fornece um conjunto de funções que implementam correspondência de padrão de expressão regular usando a mesma sintaxe e semântica do Perl 5, e é usada por vários projetos abertos (HAZEL, 2021). No *lighttpd*,

o *pcre* torna possível a edição de arquivos de configuração conforme o estilo *regex lighttpd*, de modo que é praticamente uma dependência obrigatória. Embora existam outras recomendações, como a *zlib* para compressão e a *libssl* para segurança, foi escolhido seguir o caminho mais simples para esse trabalho, apenas para demonstração.

Assim sendo, é preciso baixar o *pcre* e o *lighttpd* de seus sites oficiais. Para esse trabalho, foram usadas as versões 8.44 para o *pcre* e 1.4.59 para o textitlighttpd. Os objetos são salvos na pasta de *downloads* por padrão. Tais *downloads* são demonstrados no console abaixo.

```
# Download dos códigos-fonte
$ cd ~/Downloads/

# pcre
$ wget https://ftp.pcre.org/pub/pcre/pcre-8.44.tar.gz
# lighttpd
$ wget https://download.lighttpd.net/lighttpd/releases-1.4.x/lighttpd
    ↪ -1.4.59.tar.gz
```

Após isso, é possível prosseguir com o procedimento de preparação para a compilação: descompactação dos códigos-fonte, transferência para a pasta de desenvolvimento e clonagem das pastas. No caso desse capítulo, como o *pcre* é bastante simples, sua compilação é bastante direta, portanto, a clonagem será feita apenas no *lighttpd*. O procedimento descrito está demonstrado no console abaixo.

```
# Extração
$ tar -xf pcre-8.44.tar.gz
$ tar -xf lighttpd-1.4.59.tar.gz

# Movendo os códigos-fonte para a pasta "board-support"
$ mv pcre-8.44.tar.gz ~/texasSDK/board-support/
$ mv lighttpd-1.4.59.tar.gz ~/texasSDK/board-support/

# Clonando apenas a pasta lighttpd
$ cd ~/texasSDK/board-support/
$ cp lighttpd-1.4.59.tar.gz lighttpd-1.4.59-RASCUNHO.tar.gz
```

15.5 COMPILAÇÃO E INSTALAÇÃO DA DEPENDÊNCIA *PCRE* E DO *LIGHTTPD*

As dependências de um software devem ser compiladas antes do mesmo; portanto, é preciso compilar, primeiramente, o *pcre*. A sua compilação é direta, dado que o *pcre* é um pacote simples. Assim, os passos da compilação seguem exatamente o padrão de compilação cruzada, sem destoar: declaração das variáveis de ambiente, configuração, compilação e instalação. Além

disso, foi feito um *script* para automatizar esse processo, chamado *build_pcre.sh*. Dessa forna, o *script* completo da compilação do *pcre* é apresentado na figura 54.

Figura 54 – *script* de compilação do *pcre*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho da pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/pcre_build"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/pcre-8.44"

## INÍCIO ##

# abrir a pasta do código-fonte pcre
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar
./configure \
        --prefix=$RAIZ \
        --host=arm-linux-gnueabihf \

# compilar
make -j9

# instalar
make -j9 install
```

Fonte: AUTOR (2021)

Após a instalação, os itens produzidos pela compilação são lançados na pasta *pcre_build*, feita de antemão. Como esperado, as pastas produzidas seguem o padrão da pasta *usr*. No caso do *pcre*, são geradas as pastas *bin, include, lib* e *share*. Embora o uso prático deste pacote no *target* esteja restringido apenas as bibliotecas compartilhadas, o conteúdo das três primeiras pastas será usado para a compilação do *lighttpd*. Abaixo está exposta a estrutura simplificada da pasta *pcre_build*.

```
$ cd ~/texasSDK/board-support/pcre_build && tree -L 2
.
|---- bin
|    |---- pcre-config
|    |---- pcregrep
|    +---- pcretest
|---- include
|    |---- pcrecpparg.h
|    |---- pcrecpp.h
|    |---- pcre.h
|    |---- (...)
|---- lib
|    |---- libpcre.a
|    |---- libpcrecpp.a
|    |---- libpcrecpp.la
|    |---- libpcre.la
|    |---- (...)
|    +---- pkgconfig
+---- share
     |---- doc
     +---- man

7 directories, 24 files
```

A compilação do *lighttpd* também ocorre sem imprevistos: o procedimento segue o padrão, sendo o momento da configuração da compilação o único que requer mais atenção. No entanto, pelo fato de ser um servidor web, o *lighttpd* precisa de alguns arquivos adicionais para o seu pleno funcionamento, arquivos esses que não são produzidos junto a compilação. Além disso, também precisa de um usuário separado e uma estrutura de diretórios para tal, onde ficarão as páginas estáticas. Por esse motivo, antes de iniciar a compilação, é necessário criar esse ambiente necessário ao funcionamento do *lighttpd* no *target*.

Basicamente, o *lighttpd* precisa de um arquivo de configuração como base para iniciar o *daemon*, além de um usuário separado, por questões de segurança. Por padrão, esses se encontram nas pastas */etc* e */var* do sistema, respectivamente. Portanto, para simplificar a transferência posterior, será criada uma estrutura de arquivos conforme a hierarquia padrão dentro da pasta de compilação *lighttpd_build*.

```
$ cd ~/texasSDK/board-support/lighttpd_build
$ mkdir etc usr var
```

Primeiro, é preciso criar o arquivo de configuração do *lighttpd*, *lighttpd.conf*. Neste

arquivo estão descritos parâmetros para a inicialização do *daemon*, como pasta padrão do usuário *lighttpd*, porta padrão, módulos a serem carregados e tipos de arquivos a serem processados pelo mesmo. Esse arquivo é um pouco complexo de se escrever do zero – especialmente quando se possuem muitos requisitos para o servidor web – porém, podem ser facilmente encontrados modelos dele na internet. Abaixo, na figura 55, é apresentada a versão simplificada do arquivo *lighttpd.conf* usado. Esse arquivo é salvo na pasta *etc/lighttpd/*.

Figura 55 – Arquivo de configuração *lighttpd.conf*

```
server.document-root = "/var/www/html"

server.port = 80
server.username = "lighttpd"
server.groupname = "lighttpd"
server.tag ="lighttpd"

server.modules              = (
        "mod_access",
       (...)
)

# mimetype mapping
mimetype.assign             = (
  ".pdf"          =>      "application/pdf",
  (...)
 )
index-file.names = ( "index.html", "index.cgi", "index.php" )


alias.url += ( "/style" => "/var/www/style/" )
alias.url += ( "/aplication" => "/var/www/aplication/" )
alias.url += ( "/images" => "/var/www/images/" )
```

Fonte: AUTOR (2021)

O próximo passo é criar a estrutura de diretório do usuário *lighttpd*. Assim como no *OpenSSH*, o *lighttpd* precisa de um usuário separado para iniciar. Para poupar trabalho no *target*, é uma boa prática criar a sua pasta *home* no *host*. Por padrão, servidores web usam a pasta */var/www/* como *home* dos seus usuários, portanto, a mesma será criada.

Além disso, é necessário criar o sistema de arquivos do servidor web, isto é, a estrutura que comportará a página estática e os seus arquivos *(.html, .css* e *.js*). Essa estrutura geralmente se apresenta na seguinte forma: a pasta *aplication* para os *scripts* Javascript, a pasta *style* para os arquivos CSS, a pasta *images* para as imagens e a pasta *html* para a página em si. Esses arquivos foram desenvolvidos separadamente e transferidos para essas subpastas. A demonstração desses passos está demonstrada no console abaixo.

```
1 $ mkdir var/www && cd var/www
2 $ mkdir aplication html style images
```

Com a estrutura de diretórios e os arquivos prontos, pode-se começar a configuração do *lighttpd*, no subdiretório *usr*. Apenas em duas situações a compilação difere do padrão. A primeira é na exportação dos caminhos das pastas a serem utilizadas no *script*: é necessária a adição do caminho da compilação da dependência *pcre*, que será utilizada. Abaixo consta o console desse procedimento; nota-se o "*/usr*" no final da variável RAIZ, indicando que a compilação será realizada dentro dessa subpasta.

```
# definir o caminho da pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/lighttpd_build/usr"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/lighttpd-1.4.59-
    ↪ RASCUNHO"

# definir o caminho do pcre
export PCRE_BUILD="/home/felipe/texasSDK/board-support/pcre_build"
```

A segunda especificidade dessa compilação consiste na sua configuração. O *lighttpd* precisa que alguns parâmetros sejam explicitamente declaradas no seu ajuste, caso contrário a sua compilação falhará. Nesse sentido, opções como suporte a *zlib*, *bzip2* o protocolo IPv6 precisam ser desabilitadas, e os executáveis e bibliotecas dinâmicas, habilitados. Além disso, há a configuração do *pcre* para a compilação, feita por meio das três últimas variáveis mostradas na configuração abaixo.

```
./configure \
  --prefix=$RAIZ \
  --host=arm-linux-gnueabihf \
  --enable-shared \
  --without-zlib \
  --without-bzip2 \
  --disable-ipv6 \
  PCRECONFIG=$PCRE_BUILD/bin/pcre-config \
  PCRE_LIB=$PCRE_BUILD/lib/libpcre.a \
  CFLAGS="$CFLAGS -DHAVE_PCRE_H=1 -DHAVE_LIBPCRE=1 -I$PCRE_BUILD/
    ↪ include"
```

Por fim, a compilação e a instalação podem ser feitas com o comando *make*. O *script* completo da compilação do *lighttpd* é apresentado na figura 56.

Figura 56 – *script* de compilação do *lighttpd*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho da pasta de saída (OBS.: outros itens foram adicionados na
pasta build, por isso o /usr no final)
export RAIZ="/home/felipe/texasSDK/board-support/lighttpd_build/usr"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/lighttpd-1.4.59-RASCUNHO"

# definir o caminho do pcre
export PCRE_BUILD="/home/felipe/texasSDK/board-support/pcre_build"

## INÍCIO ##

# abrir a pasta do código-fonte lighttpd
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar
#./configure --help
./configure \
        --prefix=$RAIZ \
        --host=arm-linux-gnueabihf \
        --enable-shared \
        --without-zlib \
        --without-bzip2 \
        --disable-ipv6 \
        PCRECONFIG=$PCRE_BUILD/bin/pcre-config \
        PCRE_LIB=$PCRE_BUILD/lib/libpcre.a \
        CFLAGS="$CFLAGS -DHAVE_PCRE_H=1 -DHAVE_LIBPCRE=1 -I$PCRE_BUILD/include"

# compilar
make -j9

# instalar
make -j9 install
```

Fonte: AUTOR (2021)

A instalação é, enfim, salva na pasta *lighttpd_build/usr*. Os principais itens produzidos são os executáveis, na pasta *sbin*: *lighttpd* (o servidor web em si) e *lighttpd-angel* (um analisador de arquivos de configuração), além das bibliotecas compartilhadas, na pasta *lib*. Nesse caso, essas bibliotecas são os módulos do *lighttpd*, que aumentam as suas capacidades, como autenticação, páginas dinâmicas, protocolos variados, entre outros. A estrutura dos arquivos produzidos é apresentada no console abaixo, de forma reduzida.

```
$ cd ~/texasSDK/board-support/lighttpd_build && tree -L 3
.
|---- etc
|    +---- lighttpd
|         +---- lighttpd.conf
|---- usr
|    |---- lib
|    |    |---- mod_access.la
|    |    |---- mod_accesslog.la
|    |    |---- mod_accesslog.so
|    |    |---- (...)
|    |---- sbin
|    |    |---- lighttpd
|    |    +---- lighttpd-angel
|    +---- share
|    +---- man
+---- var
     +---- www
          |---- aplication
          |---- html
          |---- images
          +---- style

13 directories, 71 files
```

Com a dependência compilada e os arquivos do servidor web prontos, é possível fazer a transferência para o *target*. No caso do *pcre*, apenas as pastas *bin* e lib precisam ser copiadas para o diretório */usr* e, no caso do *lighttpd*, a estrutura já foi feita visando a cópia diretamente para a raiz, assim as pastas serão mescladas. É importante frisar que o sistema de arquivos escolhido foi o *arago-base*, visto que o mais completo – *rootfs* – já possui uma instalação do *lighttpd*, o que causaria conflito.

Assim como no *OpenSSH*, são necessários passos adicionais no *target* para completar a instalação do utilitário. O primeiro passo é criar um grupo e um usuário específicos para o *lighttpd*, utilizando a pasta construída anteriormente (*/var/www/*) como *home*. O segundo passo é alterar o dono do arquivo *lighttpd*.conf e o grupo do mesmo para o recém criado *lighttpd* com o auxílio do comando *chown*. Para facilitar esse processo, foi feito um *script* – *install_lighttpd.sh* – que é apresentado na figura 57.

Figura 57 – *script* de instalação do *lighttpd*

```
# Criar usuário e grupo lighttpd
groupadd lighttpd
useradd -g lighttpd -d /var/www/html -s /sbin/nologin lighttpd

# Alterar o dono do arquivo e o grupo
chown lighttpd:root /etc/lighttpd/lighttpd.conf
```

Fonte: AUTOR (2021)

15.6 EXEMPLO: PÁGINA WEB DE CONFIGURAÇÃO E MONITORAMENTO

Para exemplificar o uso do servidor web e, baseado nas principais necessidades de sistemas embarcados, decidiu-se criar um modelo de interface web contendo exemplos de opções de gerenciamento e análise, isto é, foi criada uma página estática (*front-end*) com opções de configuração funções do sistema, monitoramento de variáveis e sensores e análise de *logs*. O objetivo disso é mostrar algumas das muitas possibilidades aliadas ao uso de um servidor web em um sistema embarcado, e o quanto ele pode torná-lo mais robusto e independente.

Para que a página possa ser acessada pelo *host*, o servidor web precisa ser iniciado no *target*. Considerando que a interface de rede já esteja habilitada (conforme procedimento do capítulo anterior), é possível iniciar o servidor web com o comando abaixo. Percebe-se o caractere "&" no final, indicando que esse processo será aberto em paralelo, dado que ele ocupa o *shell*.

```
1 $ lighttpd -D -f /etc/lighttpd/lighttpd.conf -m /usr/lib &
```

Feito isso, testa-se o acesso inserindo-se o IP da placa no navegador web. A porta definida na configuração foi a 80 – o padrão HTTP – logo, não é preciso explicitá-la. Se o procedimento for bem sucedido, a interface deve ser carregada no navegador. Abaixo, na imagem 58, é mostrada a tela de apresentação da interface web embarcada.

Figura 58 – Interface web: Tela de apresentação

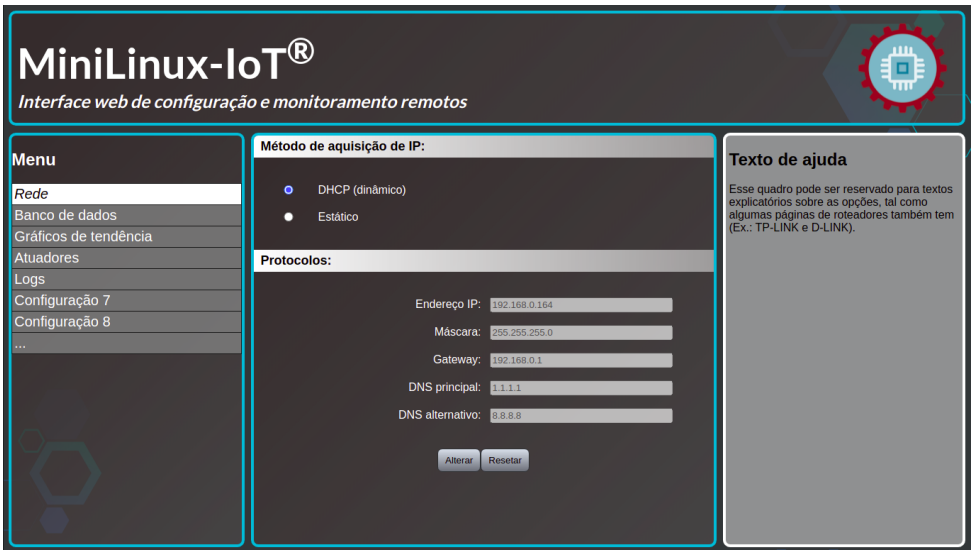
Fonte: AUTOR (2021)

Essa interface foi diretamente baseada nas páginas de configuração dos roteadores de rede mais conhecidos. O motivo disso é que os roteadores são os principais representantes dos sistemas embarcados com interface de configuração e monitoramento remotos via web. Portanto, o design dos mesmos fora aproveitado e adaptado as necessidades que hão de ser expostas.

A primeira página, apresentada na figura 58, corresponde a tela de apresentação, com fins de ambientação do usuário a interface. Na parte superior, foi colocado um nome fictício para o equipamento em questão – a esquerda – e uma logomarca do mesmo, também fictícia – a direita. Na barra lateral esquerda está o menu, com todas as opções de configurações disponíveis. Na área central serão carregadas as telas de configuração, dependendo da opção escolhida no menu. Por fim, na lateral direita, está uma área separada para um texto de ajuda, que alguns equipamentos possuem.

Ao clicar no primeiro item, o usuário é levado a primeira opção de configuração – rede – conforme a figura 59. Essencial em equipamentos de rede, a capacidade de configurar a interface de rede por via gráfica é um recurso muito útil em sistemas embarcados Linux, dado a função vital da interconectividade nos dispositivos atualmente. Foram colocadas as principais opções de rede configuráveis na página, levando em consideração apenas uma placa de rede: a obtenção dinâmica ou estática do IP e, caso escolhida a segunda opção, a definição do IP, máscara, *Gateway*, e os endereços DNS.

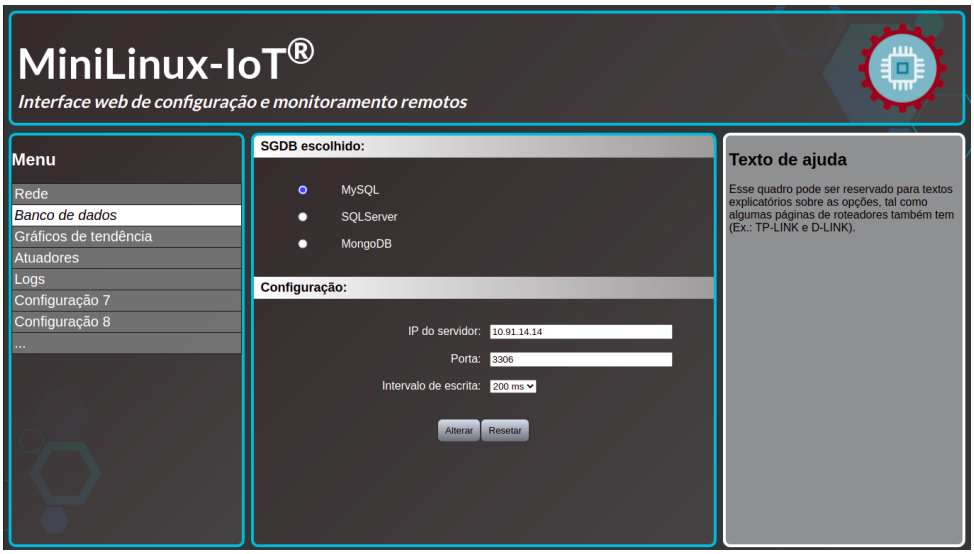
Figura 59 – Interface web: Configurações de rede



Fonte: AUTOR (2021)

Na segunda opção, na figura 60, é possível configurar detalhes de escrita de dados do dispositivo para um dos principais sistemas de banco de dados disponíveis atualmente. Supondo que exista uma aplicação embarcada que faça a escrita de dados de sensores em um (ou vários) banco(s) de dados, por exemplo, é possível alterar parâmetros da aplicação em tempo de execução, tais como o IP do servidor de banco de dados, a porta padrão e até a taxa de escrita de dados (em caso de atualização, por exemplo). O objetivo desse exemplo é mostrar que é possível utilizar uma interface web para alterar parâmetros de uma aplicação embarcada, qualquer que seja (e não somente a descrita nesse exemplo).

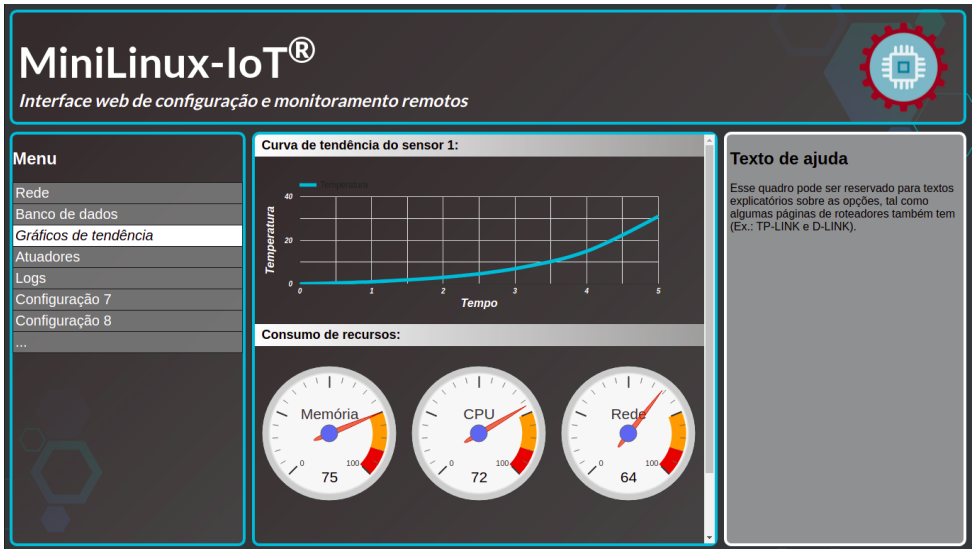
Figura 60 – Interface web: Configurações dos bancos de dados



Fonte: AUTOR (2021)

A terceira opção – figura 61 – demonstra a potencial para monitoramento remoto fornecido por uma interface web. Se o sistema embarcado for de aquisição de dados, e se os mesmos não forem complexos, é possível montar gráficos de valores (passado) e tendência (futuro). Além disso, a própria placa pode ser monitorada de forma gráfica remotamente, isto é, o seu consumo de processamento, memória e outros recursos. Essas fontes permitem a análise do estado de um sistema embarcado de forma muito mais rápida. Além dos exemplos expostos na imagem, outros podem ser adicionados, dependendo da necessidade.

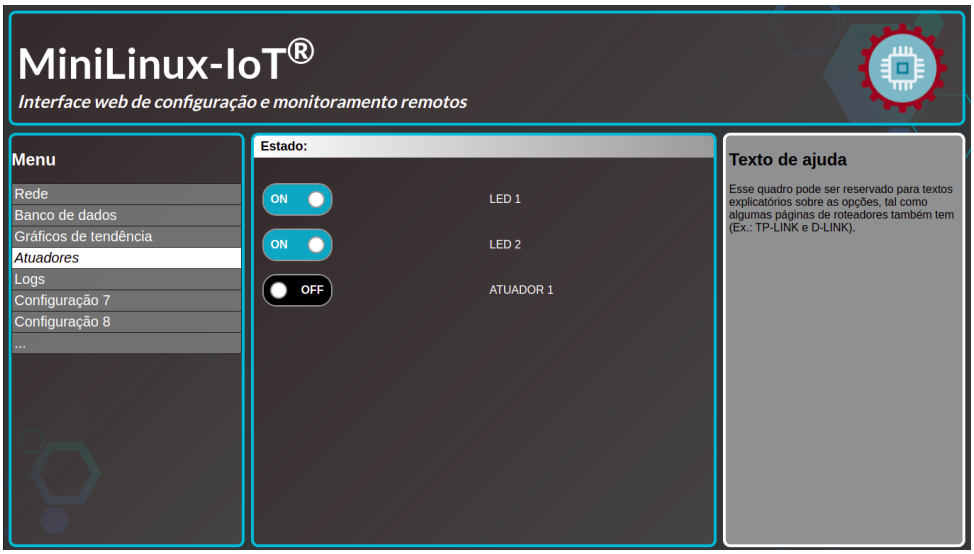
Figura 61 – Interface web: Gráfico de tendência e monitor de recursos



Fonte: AUTOR (2021)

Uma interface web também pode servir para controle ativo, e não somente análise passiva. Se atuadores estiverem conectados ao dispositivo, é possível, com a devida programação, fazer o controle remoto dos mesmos via interface gráfica web. Abaixo, na figura 62, foram listados três exemplos simples de atuadores, mas que poderiam ser expandidos para motores e braços robóticos por exemplo – tudo depende do objetivo do sistema embarcado. Nota-se que é possível definir o seu estado (ligado e desligado), mas também poderiam ser definidos controles mais precisos, como velocidade.

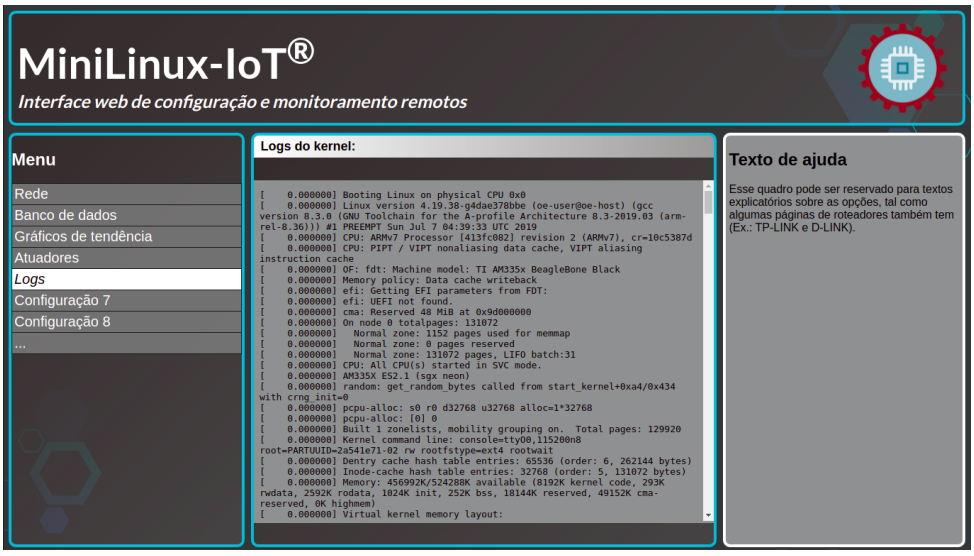
Figura 62 – Interface web: Controle de atuadores



Fonte: AUTOR (2021)

A última das opções exemplificadas é a de visualização de *logs* do sistema, apresentada na figura 63. Por vezes, é preciso analisar os *logs* gerados pelo sistema e pelas aplicações, geralmente para localizar erros. Nesse sentido, uma área exclusiva da interface web, com os principais *logs* já carregados, pode ser útil. Nessa imagem, é mostrada o *log* de inicialização do kernel, mas outros podem ser adicionados.

Figura 63 – Interface web: *Logs* do sistema



Fonte: AUTOR (2021)

As últimas duas configurações – Configuração 7 e 8 – demonstram que existem muito mais possibilidades que podem ser agregadas a uma interface web embarcada, tudo depende dos requisitos do projeto. Esse modelo serve para demonstrar algumas das mais comuns, e como elas podem ser adaptadas a várias situações diferentes.

Parte VI

Programação em Linux embarcado utilizando C++

16 APLICAÇÃO: SOFTWARE DE AQUISIÇÃO DE DADOS PARA UM SERVIDOR MYSQL

Um sistema embarcado Linux não subsiste apenas com o seu sistema operacional básico; é preciso que haja uma aplicação (software) que lhe conceda utilidade prática. Dessa forma, o objetivo deste capítulo é desenvolver e demonstrar uma aplicação embarcada simples, voltada a IoT industrial, demonstrando a importância desse tipo de desenvolvimento de software e dessa etapa no desenvolvimento de um sistema embarcado.

Em vista disso, este capítulo está segmentado da seguinte maneira: contextualização da aplicação – tópico 16.1, diagrama da aplicação – tópico 16.2, desenvolvimento do código – tópico 16.3 e demonstração – tópico 16.4.

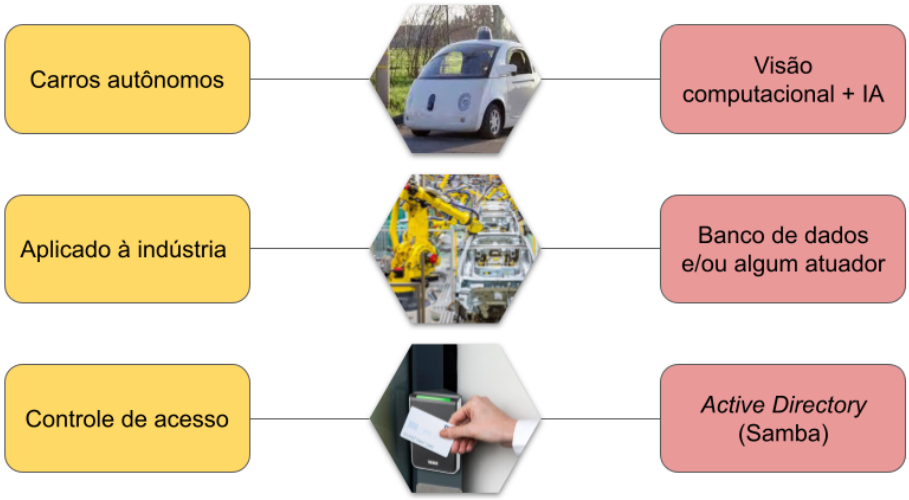
16.1 A NECESSIDADE DA AQUISIÇÃO DE DADOS PARA A IOT INDUSTRIAL

Durante o desenvolvimento de um sistema embarcado baseado em Linux, os primeiros passos – e mais essenciais – giram em torno da configuração e instalação dos itens obrigatórios – kernel, *shell* e sistema de arquivos – abordados na parte IV deste trabalho. Esses itens são a base do sistema operacional, e permitem a utilização pelo usuário e pelas aplicações, conforme já fora exposto.

Conforme o ciclo de vida do desenvolvimento de um sistema Linux embarcado, apresentado no tópico 5.1, a etapa que vem após a configuração do sistema operacional é o desenvolvimento de aplicações embarcadas, que darão ao mesmo a sua singularidade, isto é, que os tornarão diferentes um do outro, especificando-os cada um para sua tarefa – que é o objetivo de um sistema embarcado. Alguns softwares utilitários podem auxiliar o desenvolvedor a alcançar esse objetivo – conforme apresentado na parte V – porém, quase sempre é necessário que um software novo seja desenvolvido com o uso de uma linguagem de programação.

Para demonstrar a necessidade do desenvolvimento de aplicações embarcadas customizadas, basta analisar os diferentes tipos de sistemas embarcados existentes. A figura 64 apresenta alguns exemplos de sistemas embarcados da vida real, e que tipo de softwares rodariam neles.

Figura 64 – Exemplos de aplicações embarcadas



Fonte: AUTOR (2021)

Conforme a figura acima, um carro autônomo teria, provavelmente, um software de visão computacional que usaria diversas bibliotecas de inteligência artificial para tornar possível a sua direção. Se aplicado a indústria, um sistema embarcado serviria ou para monitorar (a partir da aquisição de sensores) ou para controlar (a partir de atuadores) algo. Outro exemplo é quando um dispositivo é usado para controle de acesso, o qual deverá sincronizar dados com algum servidor de autenticação – como o *Active Directory*, por exemplo. Nota-se que cada um deles é voltado para uma necessidade especifica, e isso só é possível graças a presença de softwares específicos desenvolvidos para os seus respectivos fins.

No entanto, o que foi feito até a parte IV permite apenas a utilização mínima do sistema, ou seja, não há nenhuma aplicação customizada que caracterize o sistema embarcado teórico em questão como de um tipo ou outro, ou com um foco específico; não há funcionalidade prática alguma envolvida. Mesmo com os softwares utilitários instalados na parte V – o *OpenSSH* e o *lighttpd* – ainda assim, não há uma funcionalidade prática aplicada no sistema embarcado em questão.

Embora esse trabalho seja uma tutorial que tome como base uma placa de desenvolvimento – isto é, sem um foco prático especifico – ainda assim, é interessante abordar a questão do desenvolvimento de software embarcado, tomando como base uma área de atuação real (embora não seja o objetivo produzir um dispositivo embarcado final). Para tanto, escolheu-se a área de IoT industrial, pois está intimamente ligada com o que já foi visto até aqui – especialmente na parte V.

A IoT industrial (ou simplesmente IIoT) é uma subárea da IoT. Com a evolução da indústria, houve o advento da quarta revolução industrial, ou Indústria 4.0, que tem como objetivo melhorar a eficiência do processo industrial utilizando soluções tecnológicas, a fim de criar operações responsivas e interconectadas (INTEL, 2021; BUTUN, 2020). Portanto,

surgiu a necessidade de melhor gerir as aplicações industriais, que demandam um alto nível de confiabilidade, já que falhas acarretam ou em altos prejuízos ou em riscos de vida (BUTUN, 2020).

Nesse sentido, a IIoT pode ser definida como a integração de sensores e dispositivos embarcados, *middleware*, *back-end* e *cloud* para melhor gestão das operações e ativos da empresa (BUTUN, 2020). Esse área da internet das coisas reúne máquinas e dispositivos interconectados, computação em nuvem (em geral) e estudo analítico (processamento de dados) para melhorar a performance e a produtividade dos processos industriais (AWS, 2021). Os dispositivos de IIoT variam de minúsculos sistema embarcados ambientais a complexos robôs industriais (HPE, 2021).

O conceito de IIoT nasceu praticamente junto ao de indústria 4.0, pois a aplicação da conectividade e inteligência do maquinário para melhoria da produção se faz com a implementação de dispositivos embarcados (BUTUN, 2020). Nesses dispositivos residem, por exemplo, a Inteligência Artificial, a robótica e a computação de borda, que empresas estão usando – desde o fornecimento até a linha de produção – para tomar decisões informadas e oportunas (INTEL, 2021). Dessa forma, os sistemas embarcados industriais podem ser classificados em duas grandes subáreas: para controle de máquinas e para monitoramento de máquinas (MOKEY, 2018).

Em relação a primeira, os dispositivos embarcados são usados para automatizar diversas tarefas na planta industrial, como controle de CNCs, controle de velocidade de esteiras, além de ajustes automáticos em outros equipamentos. O controle da máquina é um recurso de automação que permite aos fabricantes ajudar a melhorar a qualidade geral do produto, pois os dispositivos embarcados podem ser integrados em sistemas de controles preexistentes (MOKEY, 2018).

Em relação a segunda, os sistemas embarcados industriais revisam a condição de equipamentos em tempo real, medindo potência, pressão, temperatura, vibração, taxa de fluxo e muitos outros parâmetros. Esses dispositivos embarcados podem fazer uso de *backbones* de rede já existentes para enviar dados a um servidor local ou na nuvem (MOKEY, 2018; BUTUN, 2020). Neste capítulo, o projeto de software desenvolvido dará foco a essa área.

Nesse sentido, uma das principais características da IIoT são a geração intensiva de dados analógicos. É comum que em uma planta industrial hajam diversas variáveis a serem medidas, portanto, sistemas embarcados industriais possuem sensores que geram uma grande quantidade de dados analógicos que precisam ser tratados, analisados e, principalmente, armazenados mediante um sistema gerenciador de banco de dados (BUTUN, 2020). Esses banco de dados, então, analisam os dados e disparam eventos, conforme a necessidade (HPE, 2021).

Os dados obtidos mediante a aquisição são agregados e analisados, gerando relatórios e informações úteis para a empresa (MOKEY, 2018). Esses dados são vitais para a manutenção preditiva, que auxilia a detecção de defeitos antes que afetem a produção, aumentando a vida útil dos mesmos e a segurança dos operadores (INTEL, 2021; AWS, 2021). Logo, os sistemas

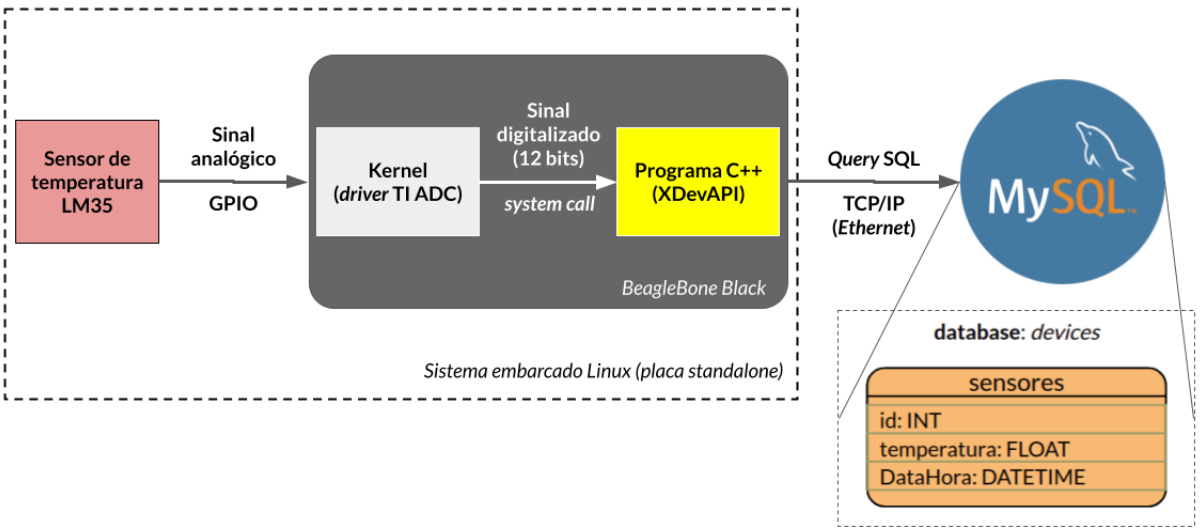
embarcados para IIoT podem realizar o monitoramento de máquinas para medir desempenho, melhorar a produtividade e otimizar recursos (MOKEY, 2018).

À luz dessa contextualização, o objetivo desse capítulo é demonstrar o desenvolvimento de uma aplicação embarcada de aquisição de dados, isto é, será construído um software que lê dados de um sensor em um determinado intervalo de tempo e enviará essa leitura a um servidor de banco de dados. Para tanto, será usado, com fins de exemplo, um sensor de temperatura LM35 e um servidor MySQL – no *host*. O programa será desenvolvido para a arquitetura alvo ARM, usando a linguagem C++.

16.2 DIAGRAMA DA APLICAÇÃO

A aplicação embarcada que há de ser desenvolvida tem como objetivo ler o valor de um sensor (no caso, de temperatura) e armazená-lo em um banco de dados. Para tanto, é necessário um software embarcado na placa que faça a aquisição desse dado analógico, trate-o e o envie ao banco por meio de uma *query*. Esse software deve rodar como um serviço, fazendo a leitura periodicamente. Desse modo, a aplicação de aquisição de dados embarcada pode ser divida em partes: o sensor, a placa (com o Linux e o programa em si) e o banco de dados. O diagrama da aplicação é apresentado na figura 65, e comentado logo em seguida.

Figura 65 – Diagrama da aplicação de aquisição de dados



Fonte: AUTOR (2021)

Primeiramente, é feita a aquisição de uma leitura do sensor de temperatura – LM35 – por parte da BeagleBone Black. Esse sensor fica conectado à placa por meio da GPIO da mesma, em uma porta analógica específica. Como o ambiente é microprocessado, é necessário um *driver* para que o sistema operacional alcance a GPIO da placa, que no caso, é o *driver* ADC da Texas

Instruments. Esse *driver* é habilitado na compilação do kernel (apresentada no capítulo 11) e vem ativado por padrão.

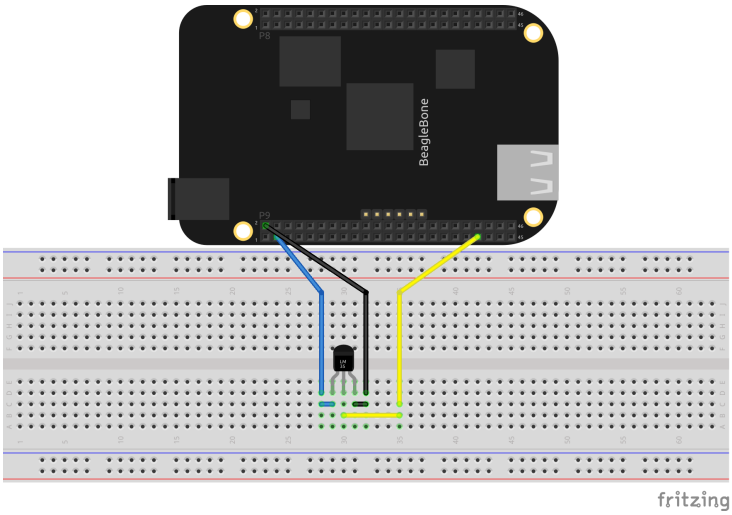
Já na placa, a leitura é tratada pelo *driver* ADC para um valor digitalizado que, conforme a resolução da BeagleBone Black – que é de 12 bits – varia entre 0 e 4095. Essa leitura é, então, recebida pelo programa de aquisição, escrito em C++, que fica rodando permanentemente, como um serviço.

O papel do software de aquisição é estabilizar uma conexão com o banco de dados e inserir a leitura em uma *query* SQL, além de um número de identificação (que determina o número da placa), e a hora em que foi realizada a *query*. É necessária a utilização de uma linguagem de programação pois o acesso ao banco de dados é feito por meio de um conector, disponibilizado oficialmente pelos desenvolvedores do banco de dados para um seleto número de linguagens. No caso do MySQL, a API usada para a conexão é a XDevAPI.

Por fim, a *query* é feita ao banco de dados mediante a interface de rede padrão da placa – Ethernet. Os dados são armazenados em uma *database* feita de antemão para esse fim. Deu se o nome da *database* de *devices*, e foi criada uma tabela chamada sensores, nos quais os dados serão salvos: o id da placa (número inteiro), o valor da temperatura (número *float*) e a data e a hora do registro (valor *datetime*).

O esquema físico de montagem do projeto foi feito em uma protoboard, conforme a imagem 66, montada com o auxílio do software *Fritzing*. Nela, observa-se a BeagleBone Black e o sensor de temperatura LM35 conectado a ela, por meio da GPIO. Nesse caso, foi utilizado o barramento P9 da GPIO, com o sensor sendo alimentado diretamente pela placa (com 3,3 V) e o pino de sinal, conectado ao pino 39 (ou AIN0), o primeiro pino analógico da placa. Vale ressaltar que, em uma situação real, o sensor e os componentes da BeagleBone Black estariam na mesma placa *standalone*.

Figura 66 – Esquema de ligação dos componentes na *protoboard*



Fonte: AUTOR (2021)

16.3 DESENVOLVIMENTO DO CÓDIGO

O software embarcado de aquisição de dados é um programa utilitário, desenvolvido na linguagem C++. Embora existam várias linguagens de programação para uso no Linux (inclusive embarcado, na arquitetura ARM), optou-se pelo C++ por conta de algumas vantagens que o seu uso traz: é mais nativa do Linux, é rápida e moderna.

A linguagem C++ já é usada extensivamente em programas utilitários nas distribuições Linux, especialmente devido ao suporte dado pelo pacote GNU, no compilador *gcc*. Além disso, pelo fato de ser compilada e de ter sido baseada em C, o C++ possui grande performance e leveza, fatores cruciais em sistemas embarcados. Por fim, o C++ ainda combina paradigmas modernos de programação, tal qual a orientação a objetos, sem deixar de lado a velocidade, sendo de maior facilidade de programação do que o C.

Antes de iniciar o desenvolvimento, é necessário criar um ambiente para tal, de modo que o código, os arquivos de compilação e o executável estejam organizados. Assim, optou-se por criar um diretório dentro do SDK para comportar essa compilação. Foi criada uma pasta chamada *codigos*, e uma subpasta chamada *aplicacao_final_mysql_BBB*, na qual o projeto estará. Esse procedimento está demonstrado nos comandos abaixo.

```
1 $ cd ~/texasSDK
2 $ mkdir codigos
3 $ mkdir codigos/aplicacao_final_mysql_BBB
```

Após isso, é necessário criar os arquivos básicos: o código-fonte, o arquivo de configuração e a pasta de compilação. Para esse desenvolvimento, decidiu-se por utilizar o *cmake* para a compilação, visto que o conector para o MySQL o utiliza como padrão e também pelo fato de ele facilitar a programação. O *cmake* é um utilitário para a compilação de pacotes – assim como o *make* – porém, além do fato de ser multiplataforma, ele gera *Makefiles*, e não executáveis diretamente. Os arquivos e a pasta foram criados conforme o console abaixo.

```
1 $ cd ~/texasSDK/codigos/aplicacao_final_mysql_BBB
2 $ mkdir build
3 $ touch main.cpp
4 $ touch CMakeLists.txt
```

Neste momento, pode-se iniciar o desenvolvimento do código. Usando uma IDE, abre-se o arquivo *main.cpp*. O primeiro passo é incluir as bibliotecas que contém os objetos que serão utilizados pelo programa. A mais importante delas é o conector para o MySQL, representado pela API XDevAPI. Os outros servem para as outras funções do código. Por fim, habilitam-se os *namespaces*: *mysqlx* e *std*.

```
#include <iostream>
#include <mysqlx/xdevapi.h>
#include <fstream>
#include <string>
#include <sstream>
#include <unistd.h>

using namespace mysqlx;
using namespace std;
```

Dentro da função *main*, o primeiro passo é declarar as variáveis. No caso desse programa, apenas duas são necessárias: uma para armazenar o valor digitalizado do sensor (do tipo *int*) e outra para conter o valor calculado da temperatura (do tipo *float*).

```
//Variáveis
int value;
float temp;
```

O programa inicia de fato quando é feita a conexão com o MySQL. Por meio da classe *Session* da XDevAPI, é possível abrir esses parâmetros com as informações de IP do servidor, porta, usuário e senha. Nesse caso, coloca-se o IP do *host* no qual fora instalado o servidor MySQL anteriormente; a porta padrão da API (que é diferente da porta do MySQL); além de um usuário e senha definidos de antemão. Após isso, seleciona-se a *database* definida para esse propósito – *devices*.

```
//Conectar ao MySQL
Session mySession("169.254.61.229", 33060, "device1",
↪  "12345678");
cout << "Conectado ao MySQL" << endl;

//Acessar a database
mySession.sql("USE devices").execute();
```

Vale ressaltar que é uma boa prática não inserir o login do usuário *root* do MySQL em projetos finais. Ao invés, é recomendável em situações do tipo fazer um usuário separado apenas para o dispositivo em questão, e limitar a quantidade de privilégios do mesmo. Isso para que, caso o dispositivo seja hackeado, o invasor não tenha acesso ao banco além dos privilégios do

usuário. Essa boa prática foi feita nesse desenvolvimento, por meio do usuário *device1*, que representa o primeiro de vários dispositivos (se houvessem).

Com a aplicação conectada ao SGBD, pode-se começar a ler os dados do sensor. Dentro de um *loop* infinito, primeiramente o valor bruto digitalizado é lido por meio da função *readAnalog*. Essa função abre uma estrutura de *stream* de arquivos para acessar o *driver* ADC do sistema, que se apresenta na forma de arquivo. O valor é salvo na variável *value*.

```
//Dentro do loop
value = readAnalog(0);
```

Após isso, calcula-se o valor da temperatura baseado na resolução da placa e nas especificações do sensor, que descrevem um aumento de 10 mV para cada grau a mais. Esse cálculo é realizado pela função *calcTemp*, e é salvo na variável *temp*.

```
//Dentro do loop
temp = calcTemp(value);
```

Antes de poder enviar a leitura para o banco de dados, é necessário fazer uma adaptação no mesmo. Nesse código, optou-se por utilizar a função *sql.execute()* da API, que usa apenas strings para realizar a *query*. Dessa forma, é preciso fazer um *cast* no valor *float* para *string*, por meio da função *castdouble2string*.

```
//Dentro do loop
std::string s = castdouble2string(temp);
```

O valor da temperatura pode, enfim, ser enviado ao banco de dados por meio de uma *query* SQL. Conforme especificado no diagrama, mais dois valores são enviados juntamente com a temperatura: o id do dispositivo e a hora do recebimento. Isso foi feito para demonstrar algumas das possibilidades de uma aplicação de aquisição. A identificação do dispositivo é útil quando se tem vários em uma planta industrial, e se precisa fazer um filtro (ou relatório). A hora da aquisição obedece o mesmo princípio. No final, fecha-se o ciclo com um *delay* de meio segundo antes de uma nova aquisição.

```
//Dentro do loop
mySession.sql(
        "INSERT INTO sensores(id, temperatura, DataHora)"
         " VALUES (1, " + s + ", NOW())").execute();
 delay(0.5);
```

O código completo desenvolvido, junto das declarações das funções utilizadas, é apresentado na figura 67.

Figura 67 – Código-fonte da aplicação - arquivo *main.cpp*

```
#include <iostream>
#include <mysqlx/xdevapi.h>
#include <fstream>
#include <string>
#include <sstream>
#include <unistd.h>

using namespace mysqlx;
using namespace std;

#define ADC_PATH "/sys/bus/iio/devices/iio:device0/in_voltage"

std::string castdouble2string(double d);
int readAnalog(int number);
float calcTemp(int analogValue);
void delay(float tempo);


int main(){
    cout << "Iniciando software de aquisição de dados" << endl;

    //Variáveis
    int value;
    float temp;

    //Conectar ao MySQL
    Session mySession("169.254.61.229", 33060, "device1", "12345678");
    cout << "Conectado ao MySQL" << endl;

    //Acessar a database
    mySession.sql("USE devices").execute();


    while (1){
        value = readAnalog(0);
        temp = calcTemp(value);
        std::string s = castdouble2string(temp);
        mySession.sql(
            "INSERT INTO sensores(id, temperatura, DataHora)"
            " VALUES (1, " + s + ", NOW())").execute();
        delay(0.5);
    }

    return 0;
}
```

```
std::string castdouble2string(double d){
    std::string s;
    ostringstream stemp;

    stemp << d;
    s = stemp.str();

    return s;
}

int readAnalog(int number){
   stringstream ss;
   ss << ADC_PATH << number << "_raw";
   fstream fs;
   fs.open(ss.str().c_str(), fstream::in);
   fs >> number;
   fs.close();
   return number;
}

float calcTemp(int analogValue){
    float temp;
    temp = (analogValue*1.8)/4096;
    temp = temp/0.01;
    return temp;
}

void delay(float tempo){
    usleep(tempo * 1000000);
}
```

Fonte: AUTOR (2021)

Com o código pronto, pode-se compilá-lo. A compilação é feita com o utilitário *cmake*, que precisa receber parâmetros para produzir o *Makefile* da aplicação. Utilizando o arquivo *cmakeLists.txt*, pode-se definir esses parâmetros de forma mais fácil. A configuração para a compilação do programa está na figura 68, mostrada abaixo, com a sua explicação logo em seguida.

Figura 68 – Arquivo de configuração da compilação - *cmakeLists.txt*

```
cmake_minimum_required(VERSION 3.10)
project(aplicacao_final_mysql_BBB)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_BUILD_TYPE Debug)

include_directories(/home/felipe/Downloads/mysql-connector-c++-8.0.22-src/
teste_ARM/include)
link_directories(/usr/lib/mysqlcppconn8_ARM)

set(PROJECT_LINK_LIBS libmysqlcppconn8.so)
add_executable(sensorDataMySQL main.cpp)
target_link_libraries(sensorDataMySQL ${PROJECT_LINK_LIBS})

SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
```

Fonte: AUTOR (2021)

As primeiras linhas do arquivo apresentado acima são comuns a diversos projetos, pois representam os parâmetros gerais do *cmake*. É preciso estabelecer a versão mínima do *cmake* a se utilizar e o nome do projeto. Além disso, no caso dessa aplicação, é necessário estabelecer a versão do C++ a ser utilizada, pois é requerida pelo conector MySQL a versão 11 para a compilação ser bem sucedida. Por fim, o executável será produzido com símbolos de depuração, definidos no parâmetro CMAKE_BUILD_TYPE. Esses símbolos são vitais para a depuração e serão utilizados no próximo capítulo.

O próximo passo é definir os diretórios do arquivo de cabeçalho e da biblioteca compartilhada do conector MySQL. Esses arquivos estão presentes no caminho especificado, pois foram compilados de antemão para essa situação. Vale destacar que é uma boa prática manter as bibliotecas compartilhadas do conector MySQL em uma subpasta do diretório */usr/lib*, pois é lá que serão elas serão buscadas dentro do *target*.

Com os caminhos das dependências definidos, basta estabelecer o nome da biblioteca compartilhada requerida (*libmysqlcppconn8.so*), determinar o nome do executável final e os códigos-fonte – no caso, apenas um – e lincá-los. O último e mais importante passo nessa compilação é definir os compiladores cruzados, pois sem eles não é possível produzir um executável para a arquitetura alvo.

Para compilar o programa, é preciso estar, no *shell*, em um diretório diferente do que está a configuração. Por padrão, essa compilação se faz em uma subpasta dentro do projeto – *build* – já construída. Portanto, acessa-se essa pasta e executam-se os seguintes comandos: *cmake*, para que o utilitário construa o *Makefile* da compilação; e *make*, para que o executável final seja compilado. Esse processo está definido no console abaixo.

```
# Encaminha-se para a pasta de compilação
$ cd build
# Produz-se o Makefile
$ cmake ..
# Compila-se o programa
$ make
```

A estrutura dos arquivos produzidos é conforme a produzida pelo comando *tree* abaixo. Nota-se o *Makefile* e o executável, de nome *sensorDataMySQL*. Esse executável deverá ser transferido para a placa, juntamente com a biblioteca compartilhada do conector MySQL, pois o programa foi compilado dinamicamente.

```
$ tree -L 1
.
|---- CMakeCache.txt
|---- CMakeFiles
|---- cmake_install.cmake
|---- Makefile
+---- sensorDataMySQL

1 directory, 5 files
```

Para a instalação no *target*, pode-se aproveitar o comando *scp* de transferência pela rede, compilado anteriormente. Primeiramente transfere-se a biblioteca compartilhada e depois o executável. Para o primeiro, copia-se a pasta *mysqlcppconn8_ARM*, em */usr/lib*. Para o segundo, cria-se uma pasta na */home*, com o nome do projeto. É uma boa prática construir um link simbólico do executável para a pasta */usr/bin*, para que assim seja acessível de qualquer lugar do sistema, como um comando utilitário. O procedimento descrito está demonstrado nos consoles abaixo.

```
# NO TARGET
# Criação das pasta do projeto na pasta /home
$ mkdir /home/aplicacao_final_mysql_BBB

# Após o executável ter sido transferido, cria-se o link simbólico
    ↪ para /usr/bin
$ ln -s /home/aplicacao_final_mysql_BBB/sensorDataMySQL /usr/bin/
    ↪ sensorDataMySQL
```

```
# NO HOST
# Transferência da biblioteca libmysqlcppconn8.so
$ scp -r /usr/lib/mysqlcppconn8_ARM/ root@169.254.61.230:/usr/lib/

#Transferência da aplicação de aquisição
scp sensorDataMySQL root@169.254.61.230:/home/
    ↪ aplicacao_final_mysql_BBB
```

16.4 DEMONSTRAÇÃO

Com o software devidamente instalado e com o link simbólico na pasta */usr/bin* estabelecido, é possível iniciá-lo. O comando é invocado no *shell*, e imprime o andamento da conexão, conforme mostrado no console abaixo. Como existe um laço infinito no código, o mesmo permanece rodando até ser interrompido manualmente.

```
$ sensorDataMySQL
Iniciando software de aquisição de dados
Conectado ao MySQL

# O programa ficará rodando, aquistando dados até ser interrompido
```

O funcionamento natural do programa é enviar uma *query* a cada meio segundo. É possível analisar esse comportamento por meio do cliente do MySQL – um console usado para acessar o banco – instalado no *host*. O comando abaixo seleciona, da tabela *sensores*, os dez últimos registros. Nota-se a estrutura definida anteriormente, na figura 65, assim como as valores definidos no código-fonte, indicando que o programa foi desenvolvido corretamente.

```
mysql> use devices;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from sensores order by DataHora desc limit 10;
+----+-------------+---------------------+
| id | temperatura | DataHora            |
+----+-------------+---------------------+
|  1 |     34.9365 | 2021-06-01 16:43:49 |
|  1 |     34.9805 | 2021-06-01 16:43:49 |
|  1 |     34.9365 | 2021-06-01 16:43:48 |
|  1 |     34.8486 | 2021-06-01 16:43:47 |
|  1 |     34.8926 | 2021-06-01 16:43:47 |
|  1 |     34.8926 | 2021-06-01 16:34:29 |
```

```
|  1 |      34.9805 | 2021-06-01 16:34:29 |
|  1 |      34.9365 | 2021-06-01 16:34:28 |
|  1 |      34.9365 | 2021-06-01 16:34:27 |
|  1 |      35.0244 | 2021-06-01 16:34:27 |
+----+--------------+---------------------+
10 rows in set (0,01 sec)
```

Dessa forma, a aplicação de aquisição cumpre o seu objetivo. Ela é apenas um exemplo, portanto, foi planejada de forma simples, apenas para demonstração. É notável que, em uma situação real, um software como esse seria muito mais robusto, podendo ser ampliado para fazer a leitura de diversos sensores de uma vez só, por exemplo. Também poderiam ser utilizados vários SGBDs diferentes, como o SQLServer e até mesmo um NoSQL, como o MongoDB, todos ao mesmo tempo (embora, nesse caso, fosse melhor separá-los em executáveis diferentes).

Todas essas capacidades de expansão (e o próprio exemplo desenvolvido) demonstram, mais uma vez, o potencial dos sistemas embarcados microprocessados para propósitos complexos. Isso pois, na situação hipotética de um aumento de sensores ou de bancos de dados na aplicação citada acima, o sistema operacional trataria de alocar os recursos computacionais para os processos, ou seja, o programador não precisaria se preocupar com esses detalhes.

Comparado a um ambiente microcontrolado, no qual o desenvolvedor precisa lidar com todos os detalhes da implementação – além de geralmente serem *single thread* – os sistemas embarcados microprocessados possuem uma enorme vantagem – agora comprovada – para embarcar aplicações que precisem de um ambiente computacional complexo para rodarem, como as industriais, por exemplo. O uso de Linux é outro fator que potencializa esses benefícios, pois, com apenas algumas adaptações, aplicações para desktop podem ser portadas para ARM, como foi o caso do software desenvolvido nesse capítulo.

17 DEPURAÇÃO: USO DO GDB SERVER PARA ANÁLISE REMOTA DE SOFTWARE

A testagem é um processo que faz parte da produção de qualquer produto. Em sistemas embarcados baseados em Linux, garantir um meio de testar o dispositivo faz parte do ciclo de vida do projeto. Dessa forma, o objetivo deste capítulo é mostrar a necessidade da depuração remota em sistemas embarcados Linux (em desenvolvimento ou já produzidos) e demonstrar, como exemplo, a depuração de um software embarcado.

Assim sendo, este capítulo apresenta: a importância da depuração remota (tópico 17.1), o uso do GDB Server de uma imagem do SDK (17.2), a depuração remota da aplicação do capítulo anterior (17.3) e, por último, uma menção a ferramenta *Code Composer* (17.4).

17.1 A IMPORTÂNCIA DA DEPURAÇÃO REMOTA PARA SISTEMAS EMBARCADOS

Na área de engenharia de software, existe um conceito – o processo de software – que auxilia desenvolvedores a organizar e dirigir o desenvolvimento de um projeto de software. Um processo de software, segundo Sommerville (2011), é "um conjunto de atividades relacionadas que levam à produção de um produto de software". Esse produto pode variar, conforme o autor, desde programas singulares (desenvolvidos em uma linguagem) até aplicações robustas, com muitas camadas.

Pelo fato de o desenvolvimento de software embarcado ser muito semelhante ao desenvolvimento convencional, esse princípio também é seguido pelos desenvolvedores para sistemas embarcados, embora o produto de software seja mais próximo, em geral, do primeiro exemplo – conforme a aplicação exemplificada no capítulo anterior.

De acordo com Sommerville (2011), um processo de software deve conter, pelo menos, quatro atividades: especificação, projeto e implementação, validação e evolução. Em relação aos sistemas embarcados, pode-se dizer que a validação é uma das atividades mais importantes, pois o software embarcado deve ser muito bem testado para que o seu pleno funcionamento possa ser garantido. Isso é essencial na medida em que softwares embarcados geralmente estão associados a tarefas críticas ou ligados a vida de alguém.

Apesar de todos os testes e validações, que devem ser realizadas na etapa de desenvolvimento, podem ocorrer situações nas quais o software embarcado apresenta falhas ou comportamentos inesperados após a sua produção (se for comercializável) ou instalação (se for único para um propósito) (SIMMONDS, 2017). Nesses casos, é muito difícil fazer a depuração localmente no dispositivo, pois o mesmo pode estar inacessível fisicamente, além da limitação computacional característica dos mesmos (YAGHMOUR et al., 2008). Isso significa que, diferentemente de softwares normais, fazer depuração e atualização em sistemas embarcados não é

um procedimento trivial.

É sabido que falhas podem acometer softwares embarcados, tanto na fase de desenvolvimento, quanto na utilização. Habilitar um meio de identificar e corrigir possíveis erros faz parte do processo de desenvolvimento de um sistema embarcado (HALLINAN, 2006; SIMMONDS, 2017). Para isso, existem diversas técnicas e ferramentas para analisar problemas em programas. As principais estão relacionadas a análise estática e dinâmica, revisão de código, *tracing* e depuração interativa (SIMMONDS, 2017). Em sistemas embarcados Linux, a principal ferramenta utilizada é o GDB (YAGHMOUR et al., 2008; HALLINAN, 2006).

O utilitário GDB (ou GNU *debugger*) é um depurador simbólico, criado em 1986 por Richard Stallman (YAGHMOUR et al., 2008). Em suma, o propósito do GDB é permitir a visualização do comportamento do programa durante a sua execução. Isso permite averiguar os motivos que levaram um software a travar, por exemplo (STALLMAN et al., 2002; SIMMONDS, 2017). Suas capacidades evoluíram com o tempo para incluir depuração de código de baixo nível (como o próprio kernel) e suporte a várias arquiteturas (HALLINAN, 2006).

O GDB foi projetado para linguagens compiladas, especialmente C e C++ – padrões do *gcc* – apesar de já suportar outras atualmente (SIMMONDS, 2017). É uma ferramenta poderosa e flexível, com muitas possibilidades de configuração, e já está a tanto tempo no mercado que outras distribuições embarcadas não Linux também o usam como padrão (YAGHMOUR et al., 2008).

Apesar de toda a capacidade de depuração embarcada e todos os benefícios da ferramenta, é inviável utilizar o GDB diretamente em um sistema embarcado, visto que o tamanho do mesmo é grande para sistemas embarcados, conforme cita Yaghmour et al. (2008). Nesse caso, é preciso depurar o software remotamente. Para isso, o pacote GDB conta com o utilitário GDB Server, feito para esse fim.

Pelo fato de o kernel Linux implementar a *system call ptrace()*, o GDB não é necessariamente obrigatório. Antes, é possível aplicar uma versão mais leve do mesmo – o GDB Server – que rode no *target* executando comandos recebidos do *host* remoto, no qual permanece a parte mais pesada (YAGHMOUR et al., 2008). Seu funcionamento é baseado tanto em rede TCP/IP quanto em serial, embora o primeiro seja preferível.

O recurso do GDB Server não é usado somente em situações pós-produção, mas também durante o desenvolvimento, devido a sua praticidade. Por isso, aplicações embarcadas podem ser depuradas sem que o GDB completo esteja embarcado nas mesmas, tendo como vantagem maior eficiência no uso de recursos e a utilização da rede para depurar, o que pode ser feito a distância.

Portanto, o objetivo deste capítulo é realizar a depuração remota de uma aplicação utilizando o GDB Server. Serão demonstradas a conexão e a depuração, isto é, o uso do console do GDB para checar o estado do programa em tempo de execução. Será usado o software de aquisição desenvolvido no capítulo 16 como exemplo, visto que foi projetado visando uma

aplicação real – a IoT industrial.

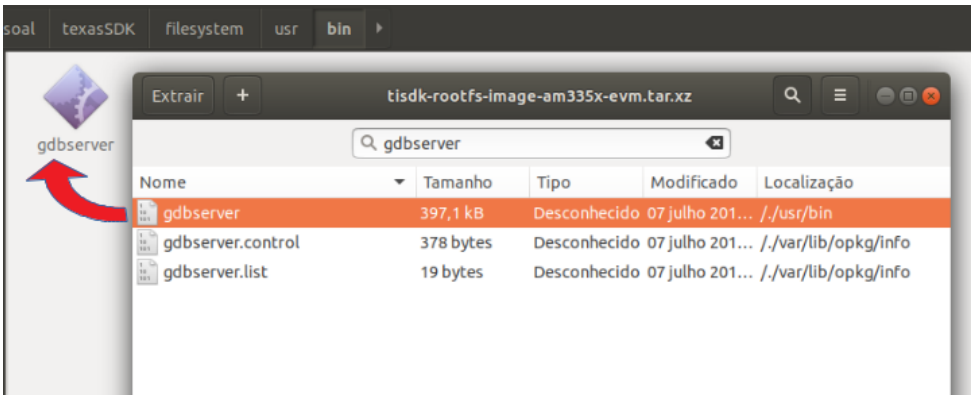
17.2 USO DO GDB SERVER DO SISTEMA DE ARQUIVOS DA TEXAS INSTRUMENTS

O GDB Server é um utilitário do pacote GDB. O GDB, pelo fato de fazer parte do pacote GNU, pode ser obtido do seu site oficial, configurado e compilado. Porém, diferentemente das outras ferramentas e utilitários abordados neste trabalho, o GDB não será compilado; ao invés, será usada a versão pronta do GDB Server, que vêm junto do sistema de arquivos *rootfs* da Texas Instruments. Essa medida será tomada para poupar trabalho, demonstrando, mais uma vez, o potencial do SDK em reduzir o tempo de desenvolvimento do dispositivo.

No entanto, o sistema de arquivos que vem sendo usado é o *arago-base*, não o *rootfs*, ou seja, o GDB Server não se encontra nele. Portanto, é necessário descompactar apenas o executável para transferi-lo ao *target*. As imagens dos sistemas de arquivos do SDK se encontram em */texasSDK/filesystem*.

Para extrair apenas o GDB Server da imagem do sistema de arquivos, é recomendável usar a ferramenta gráfica padrão do Ubuntu: o Gerenciador de Compactação. Procura-se pelo executável, que se encontra em */usr/bin* e faz-se a extração pelo ato de clicar e arrastar. O resultado é mostrado na figura 69.

Figura 69 – Extração do GDB Server da imagem *rootfs* do SDK



Fonte: AUTOR (2021)

Após isso, pode-se transferi-lo para o *target* pela rede, mediante o *scp*. O local alvo do objeto é na pasta */usr/bin*, do mesmo modo que era no outro sistema de arquivos. O GDB Server possui como dependência apenas as bibliotecas padrão da *libc*, portanto, não há problema em transferir apenas o executável. Os comandos utilizados nessa etapa são mostrados abaixo.

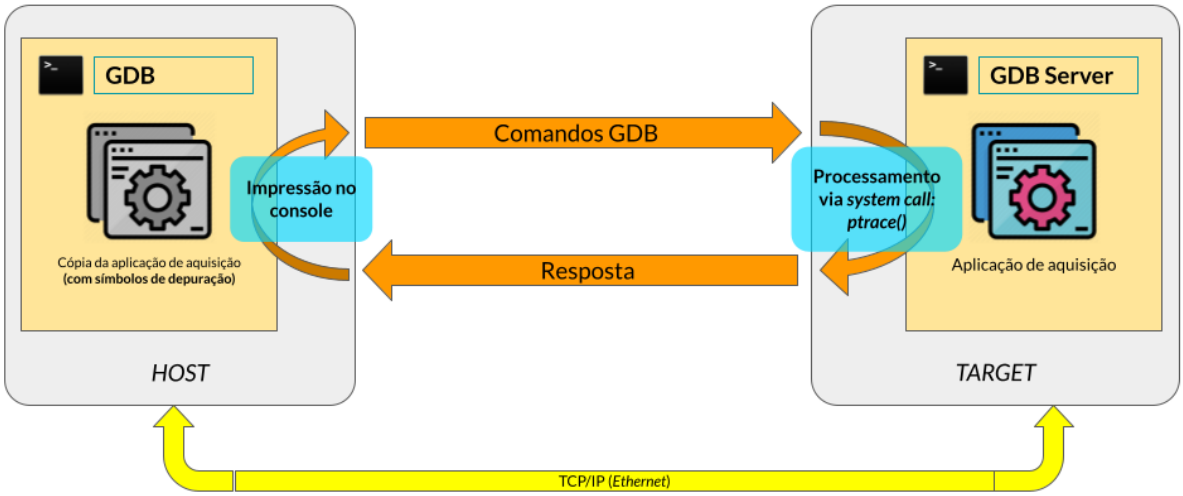
```
1 $ cd ~/texasSDK/filesystem/usr/bin
2 $ scp gdbserver root@169.254.61.230:/usr/bin
```

17.3 DEPURAÇÃO REMOTA DO SOFTWARE DE AQUISIÇÃO DE DADOS

Para depurar um programa localmente, é necessário iniciá-lo como um parâmetro do GDB. Dessa forma, o GDB controlará a execução do mesmo e, por meio do console especial do GDB, é possível controlar o fluxo do programa: parar, continuar, mostrar o valor de variáveis, entre muitas outras opções. O comando deve estar no seguinte formato: gdb nomeDoPrograma (STALLMAN et al., 2002).

No entanto, em sistemas embarcados, a depuração é feita remotamente, por meio do GDB Server. Nesse caso, o procedimento envolve um conjunto entre *host* e *target*: o primeiro – com o GDB completo – fornece a interface para o usuário e envia os comandos; o segundo – com o GDB Server – recebe os comandos e controla a execução do software (STALLMAN et al., 2002; SIMMONDS, 2017). O diagrama da figura 70 ilustra esse processo, que será descrito com mais detalhes logo a seguir.

Figura 70 – Diagrama do funcionamento do GDB Server



Fonte: AUTOR (2021)

Para iniciar a compilação, é necessário que haja, além do executável no *target*, uma cópia no *host*, para que o GDB possa acompanhar a depuração. Sendo assim, são iniciados, tanto o GDB Server quanto o GDB, com os parâmetros de conexão via rede, para que possam fazer a depuração conjuntamente. No caso de sistemas embarcados, o GDB a ser utilizado é para a arquitetura alvo – ARM, nesse caso – que faz parte do kit de desenvolvimento.

Com o GDB iniciado e conectado ao GDB Server da placa, pode-se usar a interface de console para executar comandos na aplicação, como se estivesse presente localmente. Esses comandos são enviados via rede, e executados remotamente pelo GDB Server. A resposta, que também vem por meio dele, é postada no console do *host*. Dessa forma, o ciclo se repete, até a finalização da depuração.

É valido ressaltar que, para a depuração ser possível, é preciso compilar o programa no modo *debug*, isto é, com os símbolos de depuração. Esses símbolos serão lidos pelo GDB para acompanhar o fluxo do programa. No gcc, isso se faz por meio do parâmetro *-g*, porém, no caso da aplicação em questão, que foi compilada usando *cmake*, o modo de depuração é habilitado pelo parâmetro CMAKE_BUILD_TYPE, conforme configurado no capítulo anterior.

Porém, no caso da depuração remota, na qual há dois executáveis em questão, não é obrigatório que a versão do *target* esteja com os símbolos de depuração; somente o *host* deve conter. Isso é muito útil para sistemas reais, pois os símbolos aumentam consideravelmente o tamanho do software, logo, pode-se deixar a versão de *debug* com os desenvolvedores para ser usada apenas quando necessário.

Com todos os itens instalados e prontos, é possível iniciar a depuração remota. Considerando a interface de rede já configurada, iniciam-se, no *host* e no *target*, as suas versões do GDB sobre o software de aquisição. Começando pelo *target*, o comando é o apresentado abaixo. Nota-se o IP do *host* e a porta especificada. Quanto ao IP, não é obrigatório, porém, é uma boa prática para limitar o número de conexões. Quanto a porta, qualquer uma pode ser especificada (desde que não esteja sendo usada por outra aplicação).

```
$ gdbserver 169.254.61.229:10000 sensorDataMySQL
Process sensorDataMySQL created; pid = 804
Listening on port 10000
```

Logo após, inicia-se o depurador GDB no *host*. À princípio, ele apenas iniciará o GDB, não a aplicação; para isso é necessário especificar o comando que determina o que ele deve fazer. Nesse caso, o comando é o de conexão ao *target*, feito por meio do comando *target remote*, mostrado abaixo. Nota-se a especificação do IP do *target* ao qual o GDB deve ser conectar, além da porta especificada.

```
$ arm-linux-gnueabihf-gdb sensorDataMySQL
GNU gdb (GDB) 8.2
Copyright (C) 2018 Free Software Foundation, Inc.
(...)
(gdb) target remote 169.254.61.230:10000
Remote debugging using 169.254.61.230:10000
Reading /lib/ld-linux-armhf.so.3 from remote target...
warning: File transfers from remote targets can be slow. Use "set
    ↪ sysroot" to access files locally instead.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading symbols from target:/lib/ld-linux-armhf.so.3...done.
0xb6fcea00 in _start () from target:/lib/ld-linux-armhf.so.3
(gdb)
```

O GDB Server reage, confirmando a depuração remota vindo do *host*, conforme o comando abaixo. Deste ponto em diante, a depuração será feita completamente pelo *host*. O uso do GDB se faz mediante comandos próprios da ferramenta, que estão especificados no manual oficial. No caso dessa depuração, apenas alguns serão executados, visando a análise completa do código.

```
Remote debugging from host 169.254.61.229
```

Durante a depuração, é uma boa prática especificar alguns parâmetros inicialmente, dentre os quais estão: o retorno de informações (ou a verbosidade) do programa, assim como alguns pontos de parada. O primeiro é útil para se abstrair o máximo de informações sobre a execução (quando se quer encontrar erros, por exemplo), enquanto que o segundo é útil para marcar pontos de referência no fluo do código já que o mesmo flui de uma só vez.

No GDB, esses parâmetros são configurados pelos comandos *set vebose on* e *break*. No caso desse, é possível especificar desde uma linha do código quanto uma função. Para esse código, é interessante criar um ponto de parada no *main*. Sendo assim, os comandos são executados conforme o console do GDB abaixo.

```
(gdb) set verbose on
Current language:  auto
The current source language is "auto; currently c".
(gdb) break main
Breakpoint 1 at 0x17548: file /home/felipe/texasSDK/codigos/
    ↪ aplicacao_final_mysql_BBB/main.cpp, line 20.
```

Para executar o código e ver o seu funcionamento, existem dois comandos principais: o *run* e o *continue*. O *run* percorre o programa inteiro sem parar, como se estivesse sendo executado diretamente, só que com as informações de *debug* sendo exibidas. O *continue*, por outro lado, percorre o programa até o próximo break, caso algum tenha sido especificado. No caso dessa depuração, foi utilizado o *continue*, pois se almeja fazer uma verificação mais detalhada.

Ao se utilizar o *continue* pela primeira vez, observa-se que o primeiro passo de um programa é acessar as bibliotecas compartilhadas, já que fora compilado dinamicamente. Após isso, com outro continue, o programa prossegue até o *main*, onde foi definido um *break* anteriormente.

```
(gdb) continue
Continuing.
Reading /usr/lib/mysqlcppconn8_ARM/libmysqlcppconn8.so.2 from remote
    ↪ target...
Reading /usr/lib/libstdc++.so.6 from remote target...
Reading /lib/libm.so.6 from remote target...
(..)
Reading symbols from target:/usr/lib/mysqlcppconn8_ARM/
    ↪ libmysqlcppconn8.so.2...done.
Reading symbols from target:/usr/lib/libstdc++.so.6...done.
Reading symbols from target:/lib/libm.so.6...done.
(..)

Program received signal SIGILL, Illegal instruction.
0xb6504bc8 in _armv7_tick () from target:/usr/lib/libcrypto.so.1.1
```

```
(gdb) continue
Continuing.

Breakpoint 1, main ()
    at /home/felipe/texasSDK/codigos/aplicacao_final_mysql_BBB/main.
    ↪ cpp:20
warning: Source file is more recent than executable.
20    cout << "Iniciando software de aquisição de dados" << endl;
Current language:  auto
The current source language is "auto; currently c++".
```

Com o GDB dentro do *main*, é mais vantajoso executar o código passo a passo, para observar o seu funcionamento detalhadamente; caso algum erro aconteça em um comando (ou linha), o mesmo aparecerá para o desenvolvedor. Nesse sentido, dois são os principais comandos para essa tarefa: o *step* e o *next*.

O *step* flui linha a linha do código e, caso alguma função seja encontrada, o *step* entrará nela e analisará a sua implementação também, adicionando uma alta carga para a depuração, causando demoras em algumas situações. O *next*, por outro lado, não percorre as declarações das funções, ficando apenas com a sequência do código principal. Para esse exemplo, o *next* será usado, já que o *step* só é usado para análises muito profundas.

Dessa forma, prossegue-se com a execução do programa com o comando *next*. Observa-se que, conforme o GDB passa por *couts* dentro do código, o programa reage no *target*, já que ele está sendo executado de fato; só que passo a passo. Os consoles do *host* e do *target* são apresentados abaixo. Observa-se que, após a *query* SQL – na linha 40 – o programa retorna a

leitura do sensor pela função *readAnalog* – na linha 35 – pois o programa está em *loop* neste ponto.

```
(gdb) next
27    Session mySession("169.254.61.229", 33060, "device1", "12345678");
(gdb) next
28    cout << "Conectado ao MySQL" << endl;
(gdb) next
31    mySession.sql("USE devices").execute();
(gdb) next
35      value = readAnalog(0);
(gdb) next
36      temp = calcTemp(value);
(gdb) next
37      std::string s = castdouble2string(temp);
(gdb) next
38      mySession.sql(
(gdb) next
39        "INSERT INTO sensores(id, temperatura, DataHora)"
(gdb) next
37      std::string s = castdouble2string(temp);
(gdb) next
40        " VALUES (1, " + s + ", NOW())").execute();
(gdb) next
35      value = readAnalog(0);
```

```
Iniciando software de aquisição de dados
Conectado ao MySQL
```

A última análise a ser feita é a das variáveis durante a execução de um programa. É muito importante saber que valor as variáveis estão recebendo, para se certificar que o software retornará o resultado correto. Para isso, o GDB possui o comando *print*, que retorna o valor atual de uma variável. Como exemplo, foi verificado o valor da temperatura recém calculado, antes que fosse inserido no banco de dados.

```
(gdb) print temp
Reading in symbols for libc-start.c...done.
$1 = 30.3662128
```

Por fim, o programa pode ser encerrado pelo comando *kill*, que termina o processo. Para encerrar o GDB, usa-se o comando *quit*, conforme o console mostrado abaixo.

```
(gdb) kill
Kill the program being debugged? (y or n) y
[Inferior 1 (process 804) killed]
(gdb) quit
```

17.4 UMA NOTA ACERCA DO *CODE COMPOSER*

Embora o GDB seja uma excelente ferramenta para depuração, é notável que o seu uso pelo *shell* é pouco prático. É preciso seguir uma sequência de comandos e a visualização fica limitada a uma linha por vez, o que inibe a análise do programa como um todo, principalmente as suas variáveis.

Uma experiência completa e mais eficiente de depuração poderia ser obtida mediante uma interface gráfica, para que o desenvolvedor possa observar o comportamento de todo o programa de uma vez. Felizmente, o GDB permite ser controlado por outros programas, a partir de uma estrutura de interface de máquina nativa (SIMMONDS, 2017).

Nesse sentido, existem diversas ferramentas, com interfaces gráficas amigáveis, que permitem uma melhor depuração por parte do desenvolvedor. Essas ferramentas escondem grande parte da complexidade do GDB, e fornecem mecanismos para definir pontos de interrupção, visualizar variáveis e cuidar de outras tarefas de modo mais simples (YAGHMOUR et al., 2008; SIMMONDS, 2017). Para essa finalidade, a ferramenta mais utilizada é a IDE Eclipse (YAGHMOUR et al., 2008).

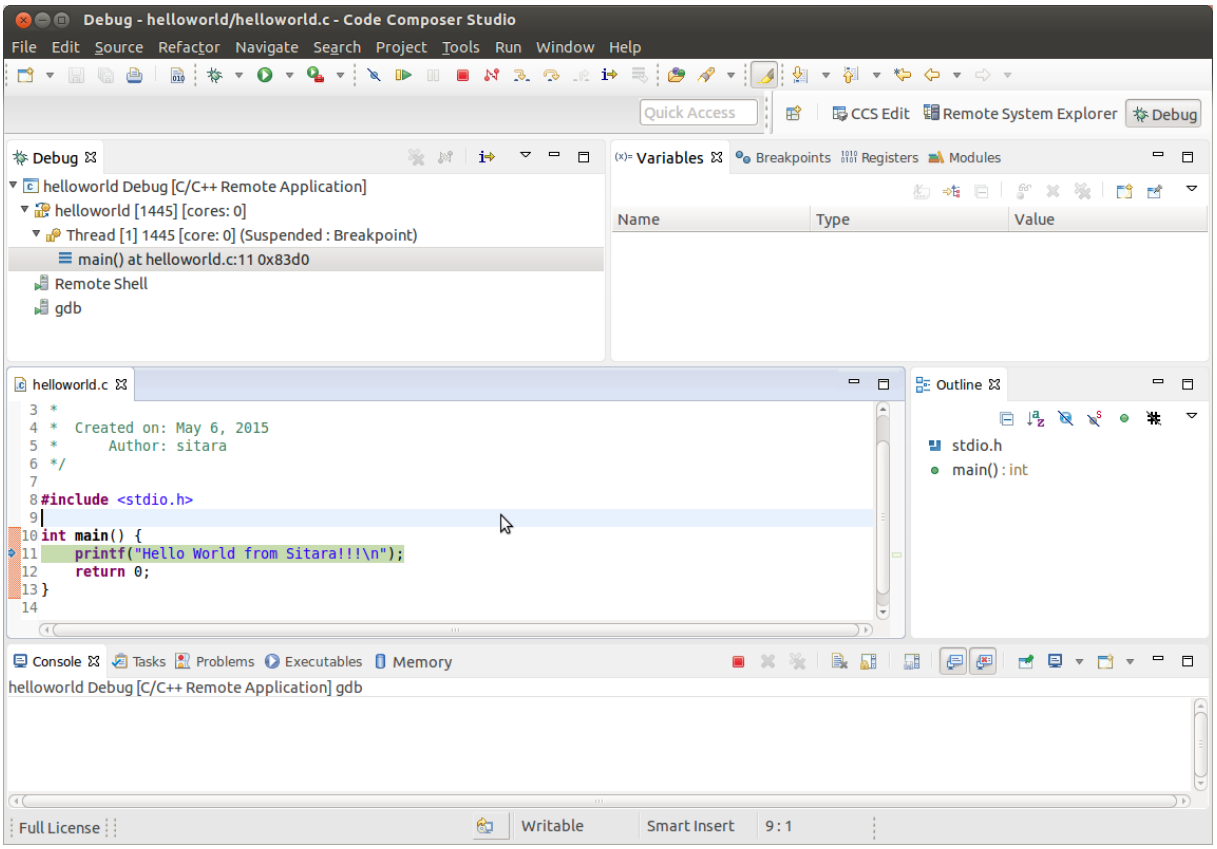
No entanto, para o desenvolvimento utilizando o *SoC am335x*, é possível, juntamente com o SDK, instalar uma IDE exclusiva da Texas Instruments: o *Code Composer Studio*. Essa IDE, que utiliza o Eclipse como *back-end*, foi projetada para ser uma ferramenta de desenvolvimento para a maioria dos *SoCs*, microcontroladores e dispositivos da Texas Instruments (TI, 2019).

O *Code Composer* compreende um conjunto de ferramentas usadas para desenvolver e depurar sistemas embarcados, tanto em hardware quanto em software. Por meio dessa IDE, é possível fazer desde a configuração do kernel, o desenvolvimento de aplicações embarcadas, conexão via SSH e até depuração remota, alvo deste capítulo (TI, 2019). Dessa forma, o *Code Composer* é a principal ferramenta de desenvolvimento do SDK, visto que possui uma interface gráfica intuitiva que cobre praticamente todas as etapas do desenvolvimento.

Apesar dessas vantagens, a instalação dessa IDE é opcional, e não foi coberta nesse trabalho devido a alguns motivos. Primeiro, a sua instalação é longa e está toda especificada na documentação, dispensando outra abordagem. Além disso, preferiu-se, no decorrer deste trabalho, por mostrar a configuração via console da maior parte dos itens, pelo fato de ser mais portável: o *script shell* pode ser melhor demonstrado (por ser estruturado) e portado (por ser um arquivo, em geral).

A despeito dessa decisão, é válido mencionar o *Code Composer* como uma excelente ferramenta de desenvolvimento embarcado, sobretudo quanto à questão da depuração remota, nativa na ferramenta. Essa fornece, em tempo real, o estado do programa, conforme a sua execução. Utilizando os símbolos e o código-fonte, é possível analisar exatamente onde está o erro no programa, de uma forma muito mais rápida. Um exemplo de uso do *Code Composer* para depuração remota é apresentado na figura 71.

Figura 71 – Interface do *Code Composer*: depuração remota



Fonte: (TI, 2019)

Parte VII

Aplicação real

18 PLATAFORMA DE AQUISIÇÃO DE DADOS PARA O SISTEMA DE MONITORAMENTO DE MÁQUINAS DA ELETRONORTE: SIMMEDAQ

Existem diversos sistemas embarcados atualmente que usam Linux como núcleo. Desde equipamentos do dia a dia quanto dispositivos específicos para aplicações robustas, os exemplos são vários. Nesse sentido, é lícito detalhar um sistema embarcado real, afim de verificar a aplicação do conhecimento exposto ao longo deste trabalho. O objetivo deste capítulo é exemplificar todo o desenvolvimento (e a importância dele) com um projeto real: a plataforma de aquisição de dados da Eletronorte - SIMMEDAQ.

À vista disso, este capítulo está separado em: introdução ao sistema de monitoramento de maquinas da Eletronorte – tópico 18.1, objetivo da plataforma – tópico 18.2 e ciclo de vida do projeto – tópico 18.3.

18.1 INTRODUÇÃO: O SIMME E A MANUTENÇÃO PREDITIVA

O recurso da energia elétrica, segundo a ANEEL (2021), “é um insumo essencial à sociedade, indispensável ao desenvolvimento socioeconômico das nações”. Nesse sentido, o setor elétrico nacional é um segmento fundamental da economia, pois garante o consumo de energia por empresas e pela sociedade (ENGIE, 2020; ANEEL, 2021).

O setor elétrico no Brasil é dividido em: geração, transmissão, distribuição e comercialização. Resumidamente, as geradoras produzem a energia, que é transportada para as subestações. Essa energia é distribuída para as casas e as empresas, por meio das comercializadoras (ENGIE, 2020; ANEEL, 2021).

Nessa linha, as Centrais Elétricas do Norte do Brasil – Eletronorte – é uma concessionária de serviço público de energia elétrica. Criada em 1973, essa empresa é uma das maiores geradoras e transmissoras do Brasil. Sua área de atuação corresponde a Amazônia brasileira, Centro Oeste e o estado do Maranhão no Nordeste (SENA, 2020).

Tomando como referência a região amazônica, foram constatados vastos problemas de logística. Esses problemas tornam a manutenção das máquinas das instalações da empresa uma tarefa muito árdua, amplificando o custo da mesma. Somado a isso, está o fato de existem grandes máquinas rotativas nas usinas, que naturalmente possuem um custo de manutenção corretiva elevadíssima, pois possuem grandes equipamentos, e movem grandes forças de trabalho (SENA, 2020).

Esses fatores levantaram a necessidade da criação de um sistema de análise da saúde das máquinas, de tal modo que a manutenção passasse a ser preditiva, ou seja, atuada apenas quando necessário, maximizando o tempo de vida dos equipamentos. Esse sistema deveria ser baseado em medições – para indicar o estado dos componentes das máquinas – que, em conjunto

de modelagens matemáticas, permitiriam a análise completa da máquina, prevendo o momento da intervenção e quais componentes precisariam de reparo (SENA, 2020).

Dessa forma, em 2007 foi criado o SIMME – Sistema de Monitoramento de Máquinas e Equipamentos – empregado em toda a Eletronorte. Esse sistema consiste em sensores instalados nas máquinas e conectados a CPUs industriais. Esses computadores, com um software de aquisição instalado, enviam os dados – pela rede de fibra óptica da empresa – para bancos de dados, que são acessados posteriormente pelos funcionários, por meio de softwares clientes (SENA, 2020).

O SIMME foi um dos primeiros sistemas desenvolvidos para monitoramento preditivo on-line, sendo um dos pioneiros em âmbito mundial. Além disso, pelo fato de ter sido desenvolvido completamente pela Eletronorte, ele não depende de ferramentas de terceiros para funcionar (SENA, 2020).

O SIMME foi desenvolvido com o propósito de auxiliar os especialistas das plantas monitoradas na análise das máquinas, a partir de uma série de ferramentas. Essas ferramentas consistem, basicamente, em gráficos de tendência, formas de onda e eventos (SENA, 2020).

O processo de aquisição de sinais é feito mediante o uso de computadores industriais, que possuem capacidade de manutenção remota e alto poder de processamento. Um modelo está representado na figura 72. No entanto, com o passar do tempo, alguns problemas começaram a aparecer, dentre os quais: rápida obsolência do hardware e do software; longas atualizações e mal contatos na placa-mãe, devido a vibração, poeira e umidade características das instalações (SENA, 2020).

Figura 72 – Modelo de computador industrial

Fonte: AUTOR (2019)

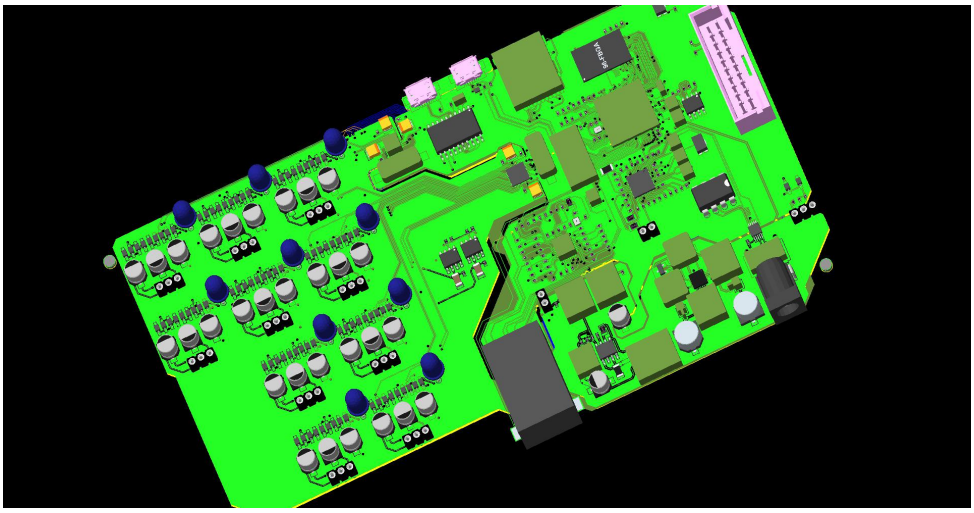
Todos esses problemas implicavam em manutenções frequentes no sistema, gerando

prejuízo. Portanto, para solucionar essa adversidade, foi projetada uma família de dispositivos para a aquisição de dados. Esses sistemas embarcados deveriam substituir os antigos computadores industriais e ser projetados especificamente para a necessidade, evitando problemas de obsolescência e incompatibilidade. À essa família de dispositivos embarcados foi dado o nome de SIMMEDAQ (SENA, 2020).

18.2 OBJETIVO DA PLATAFORMA

A plataforma SIMMEDAQ consiste em uma série de placas concebidas para substituir os computadores industriais. São sistemas embarcados microprocessados, que utilizam o kernel Linux personalizado como núcleo, e foram baseados na placa de desenvolvimento BeagleBone Black. Esses instrumentos foram projetados para operar em usinas hidrelétricas, ou seja, estarão sujeitos a todo o tipo de intempérie característica, como umidade, vapor, calor e difícil acesso (SENA, 2020). Na figura 73 é mostrada a renderização 3D do primeiro modelo do SIMMEDAQ construído.

Figura 73 – Visualização 3D da placa de circuito impresso

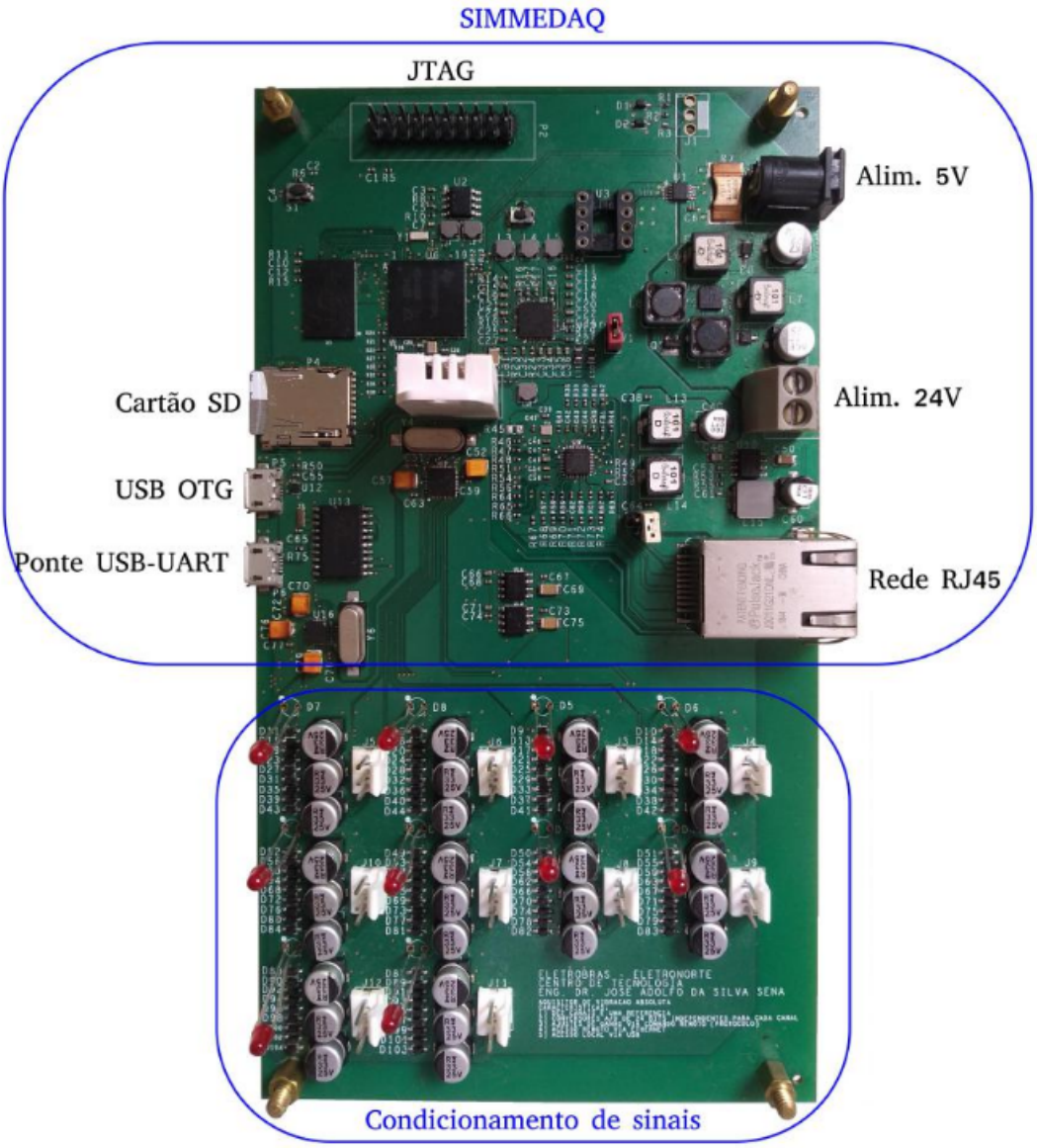


Fonte: (SENA, 2020)

O funcionamento do sistema embarcado SIMMEDAQ pode ser resumido da seguinte forma: sensores são conectados ao circuito de condicionamento de sinais. Esse circuito está conectado ao processador embarcado por meio do protocolo SPI. Na parte computacional, o software de aquisição lê os dados dos sensores por meio de um *driver*. Esse software envia os dados a um servidor de banco de dados, para ser lido posteriormente.

Na figura 74 está representada a placa pronta, assim como a separação entre as suas principais partes: a computacional e a de condicionamento. Para lidar com a aquisição de dados de vários equipamentos diferentes, esses dispositivos contam com o mesmo circuito computacional, mas com circuitos condicionadores de sinal diferentes, isto é, adaptados as máquinas que irão monitorar.

Figura 74 – Face superior da placa com as áreas de processamento e condicionamento destacadas



Fonte: (SENA, 2020)

É válido destacar um detalhe no projeto deste dispositivo. Para a construção do seu hardware, a placa de desenvolvimento BeagleBone Black foi tomada como base. Apesar de existir outra placa mais potente no mercado – a Raspberry Pi – não é possível desenvolver um sistema embarcado a partir dela, visto que seu hardware é fechado e o seu *SoC* não está disponível para venda; o seu objetivo é ser um computador de placa única (SBC – *Single Board Computer*) e não uma placa de desenvolvimento. A BeagleBone Black, por outro lado, além de ser *open hardware*, possui vasta documentação e suporte.

O desenvolvimento de um sistema embarcado para auxiliar a manutenção preditiva de equipamentos representa um grande passo na modernização do setor elétrico brasileiro. Existem diversas vantagens no desenvolvimento desse projeto, tais quais a redução drástica de custos e a

criação de uma patente nacional.

A substituição de grandes e caros computadores industriais por sistemas embarcados microprocessados baseados em Linux representa um alto ganho monetário. Isso porque esses dispensam a necessidade de licença paga de software. Além disso, há muito menos gastos com manutenção do sistema de monitoramento e maior economia de energia, visto que os dispositivos são projetados especificamente com o fim de monitoramento e aquisição (SENA, 2020).

Somando a esse fato, há o ganho intelectual para a industria do país. Esse projeto representa um desenvolvimento de alto nível, inédito no Brasil, pois é um sistema embarcado de alta complexidade – tanto em hardware quanto em software – com um firmware personalizado que usa como núcleo o kernel Linux. Sendo assim, a criação de uma patente nacional com tamanha tecnologia representa o grande potencial da academia e da indústria brasileira na criação de instrumentos computacionais avançados (SENA, 2020).

18.3 CICLO DE VIDA DO PROJETO

A placa SIMMEDAQ e o seu firmware foram desenvolvidos entre os anos de 2019 a 2021, na mesma época do estágio do autor deste trabalho na Eletronorte, no Laboratório Central. Dessa forma, este trabalho foi diretamente baseado nessa plataforma e, portanto, possuem muitas semelhanças. O ciclo de vida de um projeto de sistema embarcado Linux, exposto nas partes 2 a 6 deste trabalho, foi também baseado no desenvolvimento desse dispositivo, logo, representa, de modo genérico, as etapas do desenvolvimento do sistema embarcado de aquisição. As semelhanças no desenvolvimento desse dispositivo com este trabalho serão expostas a seguir.

A parte 2 – Preparação do ambiente – demonstra a instalação do sistema operacional e do SDK no *host*, assim como a configuração do mesmo para o desenvolvimento. Pelo fato de a SIMMEDAQ usar um processador da família *am335x*, também foi feito uso, nesse projeto, do SDK e das ferramentas da Texas Instruments, fato esse que facilitou grandemente o desenvolvimento.

A parte 3 – Configuração dos softwares essenciais – representa a personalização dos três executáveis principais para um sistema embarcado baseado em Linux – o *bootloader*, o kernel e os comandos utilitários do *shell*. O kernel – principal desses – determina a performance e o nível de customização do sistema embarcado. Assim, a plataforma SIMMEDAQ possui como principal elemento o kernel Linux, que fora customizado para atender exatamente aos requisitos do projeto, eliminando incompatibilidades e elevando a qualidade.

A parte 4 – Construção do sistema de arquivos – mostrou o processo de criação de um sistema de arquivos Linux do zero, visando a aplicação em sistemas embarcados. Devido ao fato de o projeto da SIMMEDAQ possuir seus requisitos, as imagens prontas do SDK não se enquadravam, logo, era necessário construir um sistema de arquivos específico, contando apenas com os itens necessários, visando maior controle do projeto.

A parte 5 – Configuração das interfaces de utilização – descreveu a instalação de dois

dos principais utilitários para sistemas embarcados: o *OpenSSH* e o *lighttpd*. Essas interfaces de utilização são muito comuns em ambientes industriais e, com a plataforma SIMMEDAQ, não é diferente. Faz parte dos requisitos do projeto a configuração remota, tanto por console quanto por interface gráfica. Nesse sentido, ambos completam essas necessidades.

A parte 6 – Programação em Linux embarcado utilizando C++ – mostra o desenvolvimento de uma aplicação embarcada utilizando uma linguagem de programação, assim como a sua depuração remota. Considerando que a aquisição de dados dos sensores é o principal objetivo da plataforma SIMMEDAQ, também fora desenvolvida uma aplicação de aquisição – embora muito mais complexa. Além disso, a depuração remota também foi habilitada, visto que o dispositivo estará em locais de difícil acesso – em máquinas rotativas.

Este trabalho é, na verdade, o fruto do desenvolvimento dessa plataforma de aquisição, à ser usada nas usinas hidrelétricas para monitoramento preditivo, aumentando a eficiência do processo de geração de energia elétrica. Esse projeto é uma patente pertencente a uma empresa publica - Eletronorte – que se traduz em grandes ganhos à sociedade, seja em redução de custos quanto em desenvolvimento tecnológico. Portanto, é perceptível a conformidade deste trabalho com o desenvolvimento de um sistema embarcado real.

19 CONSIDERAÇÕES FINAIS

Este trabalho consiste em uma documentação do processo de desenvolvimento de um sistema embarcado baseado em Linux. Foram expostos conceitos específicos da área, foram feitas contextualizações sobre as etapas do desenvolvimento e, mais importante, foram feitas demonstrações práticas em um ambiente de desenvolvimento, utilizando uma placa aberta e com rica documentação, pronta para a construção de produtos finais.

Dessa forma, pode-se dizer que o objetivo inicial – desenvolver uma documentação contendo o passo a passo do desenvolvimento de um sistema embarcado Linux, cobrindo teoria e prática – foi completamente concluído. Desde a introdução a área de Linux embarcado, passando pela preparação do ambiente de desenvolvimento até a maturação do projeto, para finalizar com um exemplo real, todos os itens expressos nos objetivos específicos foram cumpridos à risca.

Durante o desenvolvimento deste trabalho, procurou-se associar a teoria à prática da maneira mais eficiente possível: cada conceito possuiu sua devida relevância e demonstração na placa BeagleBone Black. Isso visando a construção de um texto mais otimizado quanto possível, visto que esse conteúdo é, naturalmente, muito extenso e complexo. Essa otimização foi feita principalmente para aqueles que desejam adentrar no mundo de Linux embarcado.

Como é perceptível, esse trabalho foi massivamente baseado nas documentações e livros base apresentados no tópico 3.5 – Procedimento de elaboração da documentação. Porém, outros materiais foram também usados, e mais de um autor foi usado quando possível.

Além disso, foram adicionados outros conceitos que os livros base não abordavam. Levando em consideração que a maior parte deles foi feito antes de 2015 (ou seja, já são um pouco antigos), procurou-se atualizar o conhecimento onde foi possível, agregando novos conceitos e fazendo novas contextualizações, adaptadas ao mundo atual, fortemente preenchido por tecnologias ubíquas, sobretudo IoT.

Documentações técnicas de conceitos complexos da área de tecnologia geralmente são escassas – especialmente na língua portuguesa. Sendo assim, o presente trabalho representa uma grande contribuição aos materiais nacionais, visto que não há uma documentação que aborde, da maneira como foi feito, a área de sistemas embarcados baseados em Linux (inclusive internacionalmente). Ademais, uma abordagem como essa, com foco em IoT industrial é ainda mais escassa, o que eleva a relevância do mesmo para uma referência única.

A principal dificuldade encontrada durante o desenvolvimento deste trabalho foi a questão das referências, visto que são poucas. Para se ir além em uma área que está na fronteira do conhecimento, foi preciso buscar outras referências inter-relacionadas, principalmente nos assuntos chave: arquitetura de computadores, redes e Linux.

Nessa linha, ainda há a dificuldade natural da área de Linux embarcado. Para quem vai desenvolver um projeto ou escrever uma documentação, é preciso que possua domínio, pelo menos a nível intermediário, de arquitetura de computadores, sistemas embarcados, eletrônica (analógica e digital), redes e interconexão, programação (especialmente na linguagem C) e, acima de tudo, conhecimento em Linux – nesse caso, o kernel. Por conta disso, um grande estudo e, consequentemente, aprendizado, foi exigido do autor para a confecção deste trabalho.

Como qualquer documentação da área de tecnologia, sempre é possível melhorar e atualizar, principalmente por causa da intensa mutação que a mesma sobre constantemente – novas tecnologias e conceitos são criados sem parar. Por conta disso, este trabalho ainda possui muitos pontos a expandir, como: desenvolvimento de *drivers* de dispositivo e multiplexação dos pinos do processador *am335x* utilizando *PinMux*.

No entanto, tal medida não é interessante. O motivo disso é que a área de sistemas embarcados, especialmente os relacionados a Linux, possuem milhares de conceitos e aplicações diferentes, de modo que podem ser indefinidamente expandidos, tanto em extensão quanto em profundidade. Este trabalho foi feito de modo otimizado, ou seja, o passo a passo foi condensado na forma de partes, contendo a quantidade ideal de informações para o conhecimento da área.

Seria mais interessante, ao invés disso, usar o conhecimento apresentado na construção de produtos reais, ou seja, priorizar a prática em detrimento da teoria. Considerando a grande demanda que a sociedade tem por soluções tecnológicas – especialmente baseadas em IoT – é muito mais vantajoso, no momento, utilizar esse conhecimento em situações práticas do que produzir mais documentações. Dessa forma, haveria um grande ganho para todos os setores, principalmente para a indústria nacional.

REFERÊNCIAS

ANDERSEN, E.; LANDLEY, R. *BusyBox: The Swiss Army Knife of Embedded Linux*. Denys Vlasenko, 2012. Acesso em: 15 jun. 2021. Disponível em: <https://busybox.net/about.html>. Citado 4 vezes nas páginas 97, 98, 99 e 100.

ANEEL. *Saiba mais sobre o setor elétrico brasileiro*. AGÊNCIA NACIONAL DE ENERGIA ELÉTRICA, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://www.aneel.gov.br/home?p_p_id=101&p_p_lifecycle=0&p_p_state=maximized&p_p_mode=view&_101_struts_action=%2Fasset_publisher%2Fview_content&_101_returnToFullPageURL=%2F&_101_assetEntryId=14476909&_101_type=content&_101_groupId=654800&_101_urlTitle=faq&inheritRedirect=true>. Citado na página 215.

AWS. *Internet das Coisas Industrial*. Amazon Web Services, Inc., 2021. Acesso em: 17 jun. 2021. Disponível em: <https://aws.amazon.com/pt/iot/solutions/industrial-iot/>. Citado na página 191.

BARRETT, D. J.; SILVERMAN, R. E. *SSH, the Secure Shell: The Definitive Guide*. [S.l.]: O'Reilly I& Associates, Inc., 2001. Citado 4 vezes nas páginas 147, 148, 149 e 150.

BOOTLIN. *Embedded Linux system development*. Bootlin, 2021. Acesso em: 16 jun. 2021. Disponível em: <https://bootlin.com/doc/training/embedded-linux/embedded-linux-slides.pdf>. Citado 6 vezes nas páginas 105, 106, 107, 108, 112 e 121.

BUTUN, I. *Industrial IoT: Challenges, Design Principles, Applications, and Security*. [S.l.]: Springer, 2020. Citado 3 vezes nas páginas 45, 190 e 191.

COLEY, G. *Beaglebone black system reference manual - Revision C. 1*. 12500 TI Blvd. Dallas, Tx 75243, 2014. Citado 8 vezes nas páginas 31, 44, 45, 55, 77, 79, 80 e 113.

COMPONENTS101. *Difference between Microprocessor and Microcontroller*. Components101, 2019. Acesso em: 20 out. 2020. Disponível em: <https://components101.com/articles/difference-between-microprocessor-and-microcontroller>. Citado 2 vezes nas páginas 24 e 30.

DENK, W.; ZUNDEL, D. *The DENX U-Boot and Linux Guide (DULG) for canyonlands*. DENX Software Engineering, 2011. Acesso em: 15 jun. 2021. Disponível em: <http://www.denx.de/wiki/bin/view/DULG/Manual>. Citado 7 vezes nas páginas 84, 86, 87, 89, 90, 91 e 92.

DOOLITTLE, L.; NELSON, J. *Boa Webserver*. boa.org, 2000. Acesso em: 17 jun. 2021. Disponível em: <http://www.boa.org/documentation/boa-3.html#ss3.2>. Citado na página 173.

ELSNER, T.; LINGNAU, A. *Introduction to Linux for Users and Administrators*. [S.l.]: tuxcademy, 2015. Citado 2 vezes nas páginas 123 e 125.

EMBARCADOS, E. *Sistema Embarcado – O que é? Qual a sua importância?* Embarcados, 2013. Acesso em: 29 set. 2020. Disponível em: <https://www.embarcados.com.br/sistema-embarcado/>. Citado na página 23.

EMBEDDEDRELATED. *EmbeddedRelated Home Page*. EmbeddedRelated, 2020. Acesso em: 20 out. 2020. Disponível em: <https://www.embeddedrelated.com/>. Citado na página 32.

ENGIE. *Você sabe como funciona o setor elétrico no Brasil?* Engie, 2020. Acesso em: 17 jun. 2021. Disponível em: <https://www.alemdaenergia.com.br/voce-sabe-como-funciona-o-setor-eletrico-no-brasil/>. Citado na página 215.

FILHO, R. d. S. A. *Customização da Plataforma Android*. 2013. Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação, Universidade do Sul de Santa Catarina, Brasil. Citado na página 33.

FILIBELI, M. C.; OZKASAP, O.; CIVANLAR, M. R. Embedded web server-based home appliance networks. *Journal of Network and Computer Applications*, Elsevier, v. 30, n. 2, p. 499–514, 2007. Citado 2 vezes nas páginas 167 e 168.

FILIPEFLOP. *O que são Placas de Desenvolvimento?* FilipeFlop Componentes Eletrônicos, 2020. Acesso em: 30 nov. 2020. Disponível em: <https://www.filipeflop.com/categoria/placas-de-desenvolvimento/>. Citado na página 31.

FREEDESKTOP.ORG. *systemd System and Service Manager*. Tfreedesktop.org, 2021. Acesso em: 16 jun. 2021. Disponível em: <https://www.freedesktop.org/wiki/Software/systemd/>. Citado na página 128.

GARCIA, F. D. *Introdução aos sistemas embarcados e microcontroladores*. Embarcados, 2018. Acesso em: 29 set. 2020. Disponível em: <https://www.embarcados.com.br/sistemas-embarcados-e-microcontroladores/>. Citado na página 23.

GNU. *GNU Bison*. Free Software Foundation, Inc., 2014. Acesso em: 14 jun. 2021. Disponível em: <https://www.gnu.org/software/bison/>. Citado na página 69.

GNU. *The GNU C Library*. Free Software Foundation, Inc., 2020. Acesso em: 16 jun. 2021. Disponível em: <https://www.gnu.org/software/libc/manual/>. Citado na página 134.

HALLINAN, C. *Embedded Linux primer: a practical real-world approach*. [S.l.]: Prentice Hall, 2006. Citado 16 vezes nas páginas 32, 46, 47, 48, 50, 51, 83, 84, 97, 98, 106, 122, 123, 133, 204 e 231.

HAZEL, P. *PCRE - Perl Compatible Regular Expressions*. pcre.org, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://www.pcre.org/>. Citado na página 173.

HE, N.; QIAN, Y.; HUANG, H.-w. Experience of teaching embedded systems design with beaglebone black board. In: IEEE. *2016 IEEE International Conference on Electro Information Technology (EIT)*. [S.l.], 2016. p. 0217–0220. Citado na página 33.

HEATH, S. *Embedded Systems Design*. [S.l.]: Newnes, 2003. ISBN 0 7506 5546 1. Citado 3 vezes nas páginas 23, 29 e 30.

HPE. *O que é a Internet das Coisas Industrial (IIoT)?* Hewlett Packard Enterprise Development LP, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://www.hpe.com/br/pt/what-is/industrial-iot.html>. Citado na página 191.

INTEL. *Tecnologia de automação e IoT industrial (IIoT)*. Intel Corporation, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://www.intel.com.br/content/www/br/pt/internet-of-things/industrial-iot/overview.html>. Citado 2 vezes nas páginas 190 e 191.

JU, H.-T.; CHOI, M.-J.; HONG, J. W. An efficient and lightweight embedded web server for web-based network element management. *International Journal of Network Management*, Wiley Online Library, v. 10, n. 5, p. 261–275, 2000. Citado 4 vezes nas páginas 167, 168, 169 e 171.

KERNEL. *Welcome to The Linux Kernel's documentation*. The kernel development community, 2016. Acesso em: 16 jun. 2021. Disponível em: <https://www.kernel.org/doc/html/v4.10/index.html>. Citado na página 113.

KERRISK, M. *The Linux programming interface: a Linux and UNIX system programming handbook*. [S.l.]: No Starch Press, 2010. Citado na página 108.

LANDLEY, R. *Introduction to cross-compiling for Linux*. landley.net, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://landley.net/writing/docs/cross-compiling.html>. Citado na página 231.

LIGHTTPD. *Lighttpd Documentation*. Redime, 2020. Acesso em: 17 jun. 2021. Disponível em: <https://redmine.lighttpd.net/projects/lighttpd/wiki>. Citado na página 173.

LINUX.COM. *Which Light Weight, Open Source Web Server is Right for You?* Linux.com, 2015. Acesso em: 17 jun. 2021. Disponível em: <https://www.linux.com/news/which-light-weight-open-source-web-server-right-you/>. Citado na página 173.

LINUXTOPIA. *Chapter 5. Configuring and Building*. Linuxtopia, 2010. Acesso em: 16 jun. 2021. Disponível em: <https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/ch05.html>. Citado na página 110.

LIU, Y.; CHENG, X. Design and implementation of embedded web server based on arm and linux. In: IEEE. *2010 The 2nd International Conference on Industrial Mechatronics and Automation*. [S.l.], 2010. v. 2, p. 316–319. Citado 2 vezes nas páginas 167 e 168.

LU, Y.; WANG, D.; ZHU, S. Design of an embedded multi-antenna satellite data acquisition system based on arm-linux. In: IEEE. *2018 Chinese Control And Decision Conference (CCDC)*. [S.l.], 2018. p. 4326–4331. Citado na página 33.

LUETH, K. L. *Top 10 IoT applications in 2020*. IoT Analytics GmbH, 2020. Acesso em: 16 jun. 2021. Disponível em: <https://iot-analytics.com/top-10-iot-applications-in-2020/>. Citado na página 152.

MACIEL, F. *O que é Linux Embarcado*. Blog do Software Livre, 2014. Acesso em: 02 dez. 2020. Disponível em: <https://softwarelivre.blog.br/2014/05/24/o-que-e-linux-embarcado/>. Citado 2 vezes nas páginas 25 e 31.

MADEIRA, D. *A Revolução das Placas de Desenvolvimento*. Portal Vida de Silício, 2018. Acesso em: 30 nov. 2020. Disponível em: <https://portal.vidadesilicio.com.br/revolucao-das-placas-de-desenvolvimento/>. Citado 2 vezes nas páginas 25 e 31.

MARINUSHKIN, K. *Improving security for network-connected, Linux-based systems*. OpenSystems Media, 2015. Acesso em: 16 jun. 2021. Disponível em: <https://www.embeddedcomputing.com/technology/security/network-security/improving-security-for-network-connected-linux-based-systems>. Citado 2 vezes nas páginas 151 e 152.

MISCHIE, S.; PAZSITKA, R. et al. On using sitara am335x starter kit to achieve basic applications based on linux operating system. In: *Proc. of 12th WSEAS Conference on Circuits, Systems, Electronics, Control and Signal Conference*. [S.l.: s.n.], 2013. p. 218–223. Citado na página 33.

MOKEY, S. *Using Embedded Systems in Industrial Automation Applications*. Digitronik Labs, 2018. Acesso em: 17 jun. 2021. Disponível em: <https://www.digitroniklabs.com/blog/using-embedded-systems-in-industrial-automation-applications/>. Citado 2 vezes nas páginas 191 e 192.

MOSNEANG, C.; MISCHIE, S.; PAZSITKA, R. Integrating the accelerometer of the am335x sitara starter kit in a qt application. In: IEEE. *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. [S.l.], 2014. p. 225–229. Citado na página 33.

MUñOZ, R. *Canal Embedded Linux*. Youtube, 2019. Acesso em: 20 out. 2020. Disponível em: <https://www.youtube.com/c/EmbeddedLinux/about>. Citado na página 32.

NETCRAFT. *August 2019 Web Server Survey*. Netcraft Ltd, 2019. Acesso em: 16 jun. 2021. Disponível em: <https://news.netcraft.com/archives/2019/08/15/august-2019-web-server-survey.html>. Citado na página 106.

OSBORN, L. *Using U-boot as production test strategy — really?* ASSET InterTech, Inc., 2018. Acesso em: 15 jun. 2021. Disponível em: <https://www.asset-intertech.com/resources/blog/2018/11/using-u-boot-as-production-test-strategy-really/>. Citado na página 91.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface, (Rev. ed. of: Computer organization and design/John L. Hennessy, David A. Patterson. 1998.)*. [S.l.]: Elsevier, Inc, 2012. Citado na página 23.

PRADO, S. *Como se tornar um desenvolvedor de Linux embarcado*. Blog do Sergio Prado, 2011. Acesso em: 30 nov. 2020. Disponível em: <https://sergioprado.org/como-se-tornar-um-desenvolvedor-de-linux-embarcado/>. Citado 2 vezes nas páginas 24 e 25.

PRADO, S. *Sobre Sérgio Prado*. Blog do Sergio Prado, 2020. Acesso em: 20 out. 2020. Disponível em: <https://sergioprado.org/sobre/>. Citado na página 32.

PRAKASH, P.; SHIN, K. Power optimization techniques for energy-efficient systems. *Electronic Engineering & Product World*, p. 09, 2013. Citado 2 vezes nas páginas 45 e 169.

REALTIMELOGIC. *What is an Embedded Application Server?* Real Time Logic, 2021. Acesso em: 17 jun. 2021. Disponível em: <https://realtimelogic.com/articles/What-is-an-Embedded-Application-Server>. Citado 3 vezes nas páginas 169, 170 e 171.

REIS, F. d. *Introdução aos Sistemas Embarcados*. Bóson Treinamentos, 2015. Acesso em: 29 set. 2020. Disponível em: <http://www.bosontreinamentos.com.br/eletronica/eletronica-geral/introducao-aos-sistemas-embarcados/>. Citado na página 23.

RUSSELL, R.; QUINLAN, D.; YEOH, C. Filesystem hierarchy standard. *V2*, v. 3, p. 29, 2004. Citado na página 123.

SAMPAIO, C. *ARM para hobbyistas – Parte 1: Placas de desenvolvimento*. Embarcados, 2014. Acesso em: 30 nov. 2020. Disponível em: <https://www.embarcados.com.br/arm-para-hobbyistas-parte-1/>. Citado na página 31.

SANTANA, O. *Cross-compiling fácil fácil*. osantana.me, 2008. Acesso em: 17 jun. 2021. Disponível em: <https://osantana.me/cross-compiling-facil-facil/>. Citado na página 231.

SANTOS, R.; PERESTRELO, L. M. C. *Beaglebone for dummies*. [S.l.]: John Wiley & Sons, 2015. Citado na página 31.

SCHILDT, H. *C completo e total*. [S.l.]: Makron, 1997. Citado na página 231.

SENA, J. A. da S. *Reltório do desenvolvimento do SIMME 4.0*. Belém: Centrais Elétricas do Norte do Brasil S/A, 2020. Citado 5 vezes nas páginas 215, 216, 217, 218 e 219.

SIMMONDS, C. *Mastering Embedded Linux Programming*. 2. ed. Birmingham, Inglaterra: Packt Publishing Ltd, 2017. ISBN 978-1-78728-328-2. Citado 33 vezes nas páginas 32, 45, 46, 47, 48, 49, 51, 55, 84, 86, 88, 91, 97, 98, 105, 106, 107, 108, 121, 122, 123, 126, 127, 128, 129, 133, 138, 140, 142, 203, 204, 206 e 211.

SOMMERVILLE, I. *Engenharia de Software*. 4. ed. São Paulo: Pearson Pratice Hall, 2011. Citado na página 203.

SOUZA, R. *Sistemas Operacionais Embarcados*. Medium, 2018. Acesso em: 29 set. 2020. Disponível em: <https://medium.com/@rafaelasouza_26759/sistemas-operacionais-embarcados-46eedef42acf>. Citado 2 vezes nas páginas 23 e 24.

SSH. *IoT Security*. SSH.COM, 2021. Acesso em: 16 jun. 2021. Disponível em: <https://www.ssh.com/academy/iot>. Citado na página 151.

SSH. *OpenSSH: SSH key management needs attention*. SSH.COM, 2021. Acesso em: 16 jun. 2021. Disponível em: <https://www.ssh.com/academy/ssh/openssh>. Citado na página 152.

SSH. *PuTTY Home - Free Downloads, Tutorials, and How-Tos*. SSH.COM, 2021. Acesso em: 14 jun. 2021. Disponível em: <https://www.ssh.com/academy/ssh/putty>. Citado na página 70.

STALLINGS, W. *Arquitetura e Organização de Computadores*. 8. ed. São Paulo: Pearson Pratice Hall, 2010. ISBN 978-85-7605-564-8. Citado 3 vezes nas páginas 23, 29 e 30.

STALLMAN, R. et al. *Debugging with GDB*. Free Software Foundation, 2002. Acesso em: 17 jun. 2021. Disponível em: <https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_toc.html>. Citado 2 vezes nas páginas 204 e 206.

TANENBAUM, A. S.; BOS, H. *Sistemas Operacionais Modernos*. 4. ed. São Paulo: Pearson Education do Brasil, 2016. ISBN 978-85-4301-818-8. Citado 6 vezes nas páginas 30, 31, 105, 121, 122 e 126.

TI. *Optimizing Linux Boot Time*. Texas Instruments Incorporated, 2012. Acesso em: 15 jun. 2021. Disponível em: <https://training.ti.com/system/files/docs/sitara-boot-camp-04-optimizing-linux-boot-time-presentation.pdf>. Citado 2 vezes nas páginas 93 e 94.

TI. *Processor SDK Linux Software Developer's Guide*. Texas Instruments Incorporated, 2019. Acesso em: 13 jun. 2021. Disponível em: <http://software-dl.ti.com/processor-sdk-linux/esd/docs/06_00_00_07/linux/index.html>. Citado 29 vezes nas páginas 44, 45, 46, 50, 55, 56, 61, 63, 64, 65, 66, 68, 73, 74, 76, 79, 85, 87, 88, 89, 94, 109, 110, 112, 113, 131, 132, 211 e 212.

VAHID, F.; GIVARGIS, T. Embedded system design: A unified hardware/software approach. *Department of Computer Science and Engineering University of California*, 1999. Citado 3 vezes nas páginas 23, 29 e 30.

WANG, E. et al. Ewvhunter: Grey-box fuzzing with knowledge guide on embedded web front-ends. *Applied Sciences*, Multidisciplinary Digital Publishing Institute, v. 10, n. 11, p. 4015, 2020. Citado na página 171.

WOLF, M. *Computers as components: principles of embedded computing system design*. [S.l.]: Elsevier, 2012. Citado 2 vezes nas páginas 29 e 30.

YAGHMOUR, K. et al. *Building embedded Linux systems*. 2. ed. [S.l.]: O'Reilly Media, 2008. ISBN 978-0-596-52968-0. Citado 48 vezes nas páginas 24, 25, 32, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 83, 84, 85, 86, 88, 89, 90, 91, 97, 98, 105, 106, 108, 111, 112, 121, 122, 125, 126, 127, 128, 133, 136, 137, 138, 139, 140, 152, 168, 173, 203, 204, 211 e 231.

YLONEN, T.; LONVICK, C. et al. *The secure shell (SSH) protocol architecture*. [S.l.]: RFC 4251, January, 2006. Citado 4 vezes nas páginas 147, 148, 149 e 150.

Apêndices

APÊNDICE A – COMPILAÇÃO CRUZADA

Ao longo do desenvolvimento de um sistema embarcado baseado em Linux, diversos softwares – padrão e personalizados – precisarão ser compilados para o *target* e, neste trabalho não é diferente. Desde o próprio kernel Linux até aplicativos customizados, esse conhecimento é usado em praticamente todos os capítulos. Portanto, é necessário expor esse conceito, já que a compilação para arquiteturas embarcadas se faz de modo específico. Vale ressaltar que todas as compilações cruzadas neste trabalho tomam como base esse apêndice.

Programas de computador desenvolvidos com o uso de uma linguagem compilada (como C e C++, por exemplo) precisam de um compilador para serem executados. Um compilador é o programa que converte o código-fonte em um arquivo objeto (ou código de máquina), e tal processo é denominado de compilação (SCHILDT, 1997). São feitas diversas análises, filtros de erros e união (lincagem) de diversos arquivos fontes. Quando feita visando a mesma arquitetura na qual fora produzida – por exemplo, programas para desktop x86 desenvolvidos em outras máquinas do mesmo tipo – a compilação é do tipo direta.

No entanto, existem situações em que não é possível fazer a compilação diretamente na arquitetura alvo. Isso ocorre principalmente pelo fato de que o processador é limitado demais para executar o processo de compilação (que geralmente requer uma certa potência de hardware). Casos assim são os dos sistemas embarcados. Quando é necessário fazer uma compilação em um dispositivo (por exemplo, x86_64) visando a execução em outro (por exemplo, ARM), então a compilação é do tipo cruzada (HALLINAN, 2006; YAGHMOUR et al., 2008).

Fazer a compilação de grandes pacotes de software de modo manual – isto é, sem o auxílio de um gerenciador de pacotes como o *dpkg* ou o *apt* – é um processo trabalhoso, visto que são diversas as opções que precisam ser configuradas. Na compilação cruzada, esse processo é ainda mais trabalhoso, visto que, dependendo do software a ser compilado, diferentes adaptações terão de ser feitas, pois os pacotes nem sempre se ajustam à arquitetura alvo; ou seja, cada caso é um caso (LANDLEY, 2021; SANTANA, 2008).

Apesar disso, existem uma certa sequência de passos que podem ser executados em praticamente todos os softwares (pelo menos, os que serão compilados ao longo deste trabalho), bastando algumas adaptações pontuais. O principal da compilação cruzada de pacotes de software é declarar para qual arquitetura o mesmo será compilado e observar o processo de compilação e o resultado final para saber se tudo deu certo, além de testá-lo, obviamente.

Para demonstrar esse processo, será feita a compilação cruzada de um pacote simples, o GNU *nano*. Esse programa é um editor de texto desenvolvido pelo GNU software. É um pouco mais robusto do que seus antecessores, como o *vi* e *vim*, ambos de console também. Esse programa é excelente para a demonstração, visto que é pequeno, porém configurável e também

possui dependências, itens que quase sempre grandes programas também possuem.

Para começar, é necessário obter o código-fonte. Praticamente todos os softwares para Linux possuem seus fontes disponíveis na internet, geralmente no *Github*, bastando ao desenvolvedor baixar o item compactado. Isso é vantajoso na medida que o desenvolvedor sempre terá acesso a versão mais recente do programa, principalmente em relação à programas do pacote GNU, que são atualizados com frequência.

Durante o desenvolvimento deste trabalho, a maioria dos pacotes serão baixados da internet, porém em outras ocasiões (no caso do U-Boot e do kernel Linux) será usada a versão cedida pelo SDK da Texas Instruments. Apesar de, atualmente, existirem versões mais atualizadas do mesmo, decidiu-se por isso para manter o desenvolvimento de acordo com a documentação disponível.

O download dos programas quase sempre se faz diretamente do site oficial dos mesmos. É recomendável utilizar a saída gráfica, porém, caso se precise utilizar o console, o comando *wget* pode ser utilizado para baixar o pacote. Quando disponível no *Github* (ou quando se está contribuindo para o desenvolvimento), o download pode ser feito pelo comando *git clone*. Abaixo é mostrado um exemplo de cada. O pacote costuma vir compactado, no formato *.tar.xz* ou *.tar.gz*.

```
1 $ wget https://www.nano-editor.org/dist/v5/nano-5.6.1.tar.xz
2 $ git clone git://github.com/madnight/nano.git
```

Após o download (considerando que tenha sido salvo na pasta */Downloads*), é importante mover o pacote para uma pasta isolada, específica para o desenvolvimento. Neste trabalho, resolveu-se usar como padrão uma do SDK, */texasSDK/board-support*, visto que é nesse local que estão os fontes que já vem no SDK. Portanto o pacote baixado será descompactado e movido com os comandos abaixo.

```
1 $ tar -xf nano-5.6.1.tar.xz
2 $ mv nano-5.6.1 ~/texasSDK/board-support/
```

Além disso, é uma boa prática clonar a pasta dos códigos-fonte, para o caso de se algum arquivo corromper ou der conflito durante a compilação, pode-se apagar a pasta e restaurar o seu conteúdo original. Para isso, usa-se os comando abaixo.

```
1 $ cd ~/texasSDK/board-support
2 $ cp -r nano-5.6.1/ nano-5.6.1-RASCUNHO
```

Neste momento, o desenvolvimento está pronto para ser iniciado. É comum que a sequência de comandos a serem demonstrados a seguir sejam executados de uma só vez, a partir de um arquivo executável do *shell script* do Linux, geralmente chamado de *build.sh*. Porém, a fim de demonstrar passo a passo o processo de compilação cruzada, os comandos relativos a compilação do *nano* serão listados e comentados um a um e, no final, será mostrado o *script* de

compilação completo. Essa prática foi também adotada em todas as outras compilações feitas nesse trabalho.

O primeiro passo é exportar as variáveis de ambiente que definem a arquitetura alvo. Durante o processo de compilação cruzada, é necessário informar que se deseja compilar para outra arquitetura. Esse passo é executado declarando-se variáveis de ambiente – ARCH e CROSS_COMPILE, em geral – que contêm as diretivas para a arquitetura alvo e compilador cruzado, respectivamente. Alguns softwares podem exigir ainda outras variáveis de ambiente adicionais, como CC e AR.

No *shell* Linux, é possível declarar variáveis, assim como em qualquer linguagem, e são chamadas de variáveis de ambiente. É possível guardar strings, números e até comandos inteiros para serem usados posteriormente. O comando *export* age como uma extensão desse. Esse comando, que já vem embutido no *shell*, garante que variáveis e funções sejam passadas aos processos filhos, o que é essencial na compilação cruzada, pois o comando *make*, responsável pela compilação, é um processo filho que usará as variáveis de arquitetura.

É comum que manuais e livros apresentem a declaração dessas variáveis diretamente na hora da compilação, como em *make CROSS_COMPILE=arm-linux-gnueabihf-*. Porém, neste trabalho será seguida uma boa prática, que é a declaração antecipada dessas variáveis de ambiente, como se segue abaixo. Assim, o comando de compilação, *make*, poderá ser executado de forma bem mais simples.

```
1 $ export ARCH=arm
2 $ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Além dessas variáveis, é necessário, também, declarar o local da pasta de saída do produto da compilação – que pode ser um executável, bibliotecas compartilhadas, códigos-fonte, manuais ou todos esses juntos – além do local do código-fonte. Embora nem sempre usados, é importante para manter a referência, no *script* de compilação, dos locais das pastas, caso a compilação precise ser feita em outro computador.

O local da pasta de saída é declarado como RAIZ, e o do código-fonte, como SOURCE. Caso não seja especificado o caminho de saída no momento da compilação, o produto será instalado na pasta do código-fonte. Os comandos são demonstrados abaixo.

```
1 $ export RAIZ="/home/felipe/texasSDK/board-support/nano_build"
2 $ export SOURCE="/home/felipe/texasSDK/board-support/nano-5.6.1-
    ↪ RASCUNHO"
```

Uma vez declaradas as variáveis de ambiente, pode-se iniciar a configuração do software para a compilação. Para iniciar a configuração, é necessário acessar o local do código fonte, já que o *script* de configuração e de compilação devem ser executados de lá (salvos casos muito específicos). A pasta pode ser acessada com o comando *cd*, conforme o console abaixo.

```
$ cd $SOURCE
```

Como essa é a primeira tentativa de compilação, os arquivos de configuração estão zerados, ou seja, não possuem nenhuma informação acerca da máquina, pois não foram modificados desde que foram extraídos. Mas, caso já tivesse sido feita uma compilação (ou pelo menos uma tentativa) e se quisesse compilar novamente, é proveitoso limpar a configuração anterior, para evitar conflitos entre arquivos ou resultados inesperados. Tal comando se faz com o *make distclean*, demonstrado abaixo.

```
$ make distclean
```

Assim, pode-se configurar o programa. Essa etapa é diferente dependendo do software; os mais básicos costumam fazer uso apenas de um *script* chamado configure, enquanto que outros mais complexos fazem uso de uma interface de configuração baseada em ncurses, *menuconfig*. Por exemplo, no caso do *nano*, a configuração é por configure, no caso do U-Boot, kernel e o Busybox, é por *menuconfig* (todos pertencentes à Parte III – Configuração dos softwares essenciais).

Em geral, quando se usa o *script* configure, dois parâmetros são fundamentais: o diretório de saída e a arquitetura alvo. Outros podem ser configurados conforme as necessidades do projetista, e variam em cada caso; geralmente estão associados à adição ou não de funcionalidades do programa na versão final, ou bibliotecas que as fornecem.

Diferentes programas possuem diferentes dependências e, caso a máquina alvo não as possuam, é necessário compilá-las também, do mesmo modo que o programa principal, ou seja, a compilação por vezes é recursiva também. Por esse motivo, a compilação cruzada tende a ser extenuante pois, dependendo do número de dependências, há a necessidade de compilar todas manualmente, sem o auxílio de um gerenciador de pacotes.

Por exemplo, na configuração do *nano* é possível desabilitar o suporte a colorização da sintaxe conforme o tipo de arquivo (*.c, .php, .sh*, etc.). Essa funcionalidade vem de uma biblioteca compartilhada que é uma dependência do programa – *libmagic*. Como o *target* em questão, não possui essa biblioteca, o programa não funcionará a menos que essa funcionalidade seja desabilitada no momento da compilação. Isso se faz pelo parâmetro disable-libmagic, conforme apresentado no console abaixo.

```
$ ./configure \
  --prefix=$RAIZ \
  --host=arm-linux-gnueabihf \
  --disable-libmagic
```

Desse modo, tudo está pronto para a compilação. Apenas o comando *make* seria necessário, porém, existe uma prática que é muito útil em desenvolvimentos que é o uso do parâmetro *-j*.

Esse parâmetro montra *threads* de compilação, baseado no número de núcleos que o processador tem. Em situações normais, o *make* usa apenas um núcleo (não importa o tamanho da compilação – grande ou pequena), mas, com isso, há um uso bem mais eficiente do processador, reduzindo, assim, o tempo de compilação.

Para usar esse parâmetro é necessário saber o número de núcleos que o processador tem, o que pode ser obtido com o comando *nproc*. No caso deste desenvolvimento, o *host* usado possui 8. Depois, soma-se +1 ao número, pois, caso alguma *thread* acabe antes das outras, um núcleo não ficará desocupado. Assim, a compilação é executada com o comando mostrado abaixo.

```
$ make -j9
```

Para finalizar a compilação, é necessário executar o comando *make install*, que instala os produtos da compilação na pasta desejada.

```
$ make -j9 install
```

O padrão da instalação de softwares é conforme a hierarquia da pasta */usr* do Linux, isto é, a estrutura é normalmente dividida nas seguintes pastas: *bin* e *sbin* (para executáveis), *include* (para códigos-fonte), *lib* (para bibliotecas compartilhadas) e *share* (para manuais). Diferentes programas produzem uma ou outra pasta, ou todas elas quando necessário. No caso do *nano*, apenas as pastas *bin* e *share* foram produzidas, conforme se observa na estrutura do diretório *nano_build*, representada com o comando *tree*:

```
$ cd $RAIZ
$ tree
.
|---- bin
|    |---- nano
|    +---- rnano -> nano
+---- share
     |---- doc
     |---- info
     |---- locale
     |---- man
     +---- nano
```

Ademais, é importante verificar se o programa foi devidamente compilado para ARM. Pode-se averiguar isso facilmente pelo comando *file*, que analisa a natureza de um arquivo. Se a saída do comando corresponder a um executável para ARM, com o *linker* apontando para */lib/ld-linux-armhf.so.3*, significa que a compilação foi bem sucedida.

```
$ cd bin
$ file nano
nano: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
    ↪ dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for
    ↪ GNU/Linux 3.2.0, with debug_info, not stripped
```

Dessa forma, o programa está pronto para ser transferido para o *target*. No entanto, é válido um comentário acerca do que transferir. Em compilações diretas, diversos arquivos são produzidos além do essencial – como os arquivos de cabeçalho e documentações, visto que podem servir de referência no futuro. No caso de sistemas embarcados, porém, somente o executável (ou a biblioteca, dependendo do que se está compilando) precisa ir, visto que apenas o estritamente essencial é necessário. Portanto, será transferido apenas o executável *nano* nesse caso, que se encontra na pasta bin, para o seu respectivo local no *target*, na pasta */usr/bin*.

```
$ cp nano /media/felipe/rootfs/usr/bin/
```

Por fim, com o executável já no *target*, basta testá-lo. Essa etapa também é importante, pois trata-se do teste final do software compilado a fim de saber se tudo deu certo e se há compatibilidade afinal. Como foi inserido em um local pertencente ao PATH, o programa pode ser executado de qualquer lugar do sistema de arquivos. Além disso, como se trata de um programa de edição e visualização de arquivos de texto, sua demonstração é melhor visualizada quando utilizado sobre um arquivo código-fonte; portanto, será usado um arquivo de cabeçalho que se encontra no sistema de arquivos *rootfs* do SDK como exemplo - */usr/include/locale.h*. O resultado disso é mostrado na figura 75, com os comandos mostrados logo abaixo.

```
$ cd /usr/include
$ nano locale.h
```

Figura 75 – GNU *nano* funcional no *target* ARM após compilação cruzada

Fonte: AUTOR (2021)

O *script* completo da configuração realizada nesse capítulo é mostrado abaixo, na figura 76.

Figura 76 – *script* de compilação e personalização do *nano*

```
## VARIÁVEIS DE AMBIENTE ##

# definir a arquitetura
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-

# definir o caminho da pasta de saída
export RAIZ="/home/felipe/texasSDK/board-support/nano_build"

# definir o caminho da pasta do código-fonte
export SOURCE="/home/felipe/texasSDK/board-support/nano-5.6.1-RASCUNHO"

## INÍCIO ##

# abrir a pasta do código-fonte nano
cd $SOURCE

# limpar compilações anteriores
make distclean

# configurar
#./configure --help
./configure \
        --prefix=$RAIZ \
        --host=arm-linux-gnueabihf \
        --disable-libmagic

# compilar
make -j9

# instalar
make -j9 install
```

Fonte: AUTOR (2021)