UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

FERNANDO MARTINS RIBEIRO

DESENVOLVIMENTO DE DEVICES DRIVERS PARA O RTOS NUTTX NO PADRÃO POSIX

PATO BRANCO
2021

FERNANDO MARTINS RIBEIRO

DESENVOLVIMENTO DE DEVICES DRIVERS PARA O RTOS NUTTX NO PADRÃO POSIX

DEVELOPMENT OF DEVICE DRIVERS FOR THE RTOS NUTTX IN THE POSIX STANDARD

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Gustavo Weber Denardin

PATO BRANCO
2021

Esta licença permite remixe, adaptação e criação a partir do trabalho, para fins não comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

FERNANDO MARTINS RIBEIRO

DESENVOLVIMENTO DE DEVICES DRIVERS PARA O RTOS NUTTX NO PADRÃO POSIX

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 03/dezembro/2021

Gustavo Weber Denardin
Doutorado
Universidade Tecnológica Federal do Paraná

Dalcimar Casanova
Doutorado
Universidade Tecnológica Federal do Paraná

Diogo Ribeiro Vargas
Doutorado
Universidade Tecnológica Federal do Paraná

PATO BRANCO
2021

Dedico este trabalho à minha família, em especial aos meus irmãos Lucas Martins Ribeiro e Eduardo Martins Ribeiro, por me darem o apoio necessário para conseguir realizar esta jornada.

AGRADECIMENTOS

Aos meus pais, Vanderlei M. Ribeiro e Isolde T. Pachoal, por todo o apoio, carinho e por serem exemplos de honestidade, respeito e compreensão.

Aos meus irmãos, Lucas Martins Ribeiro e Eduardo Martins Ribeiro, pela motivação, cuidado, auxílio incondicional e por sempre estarem ao meu lado.

À UTFPR e todos os professores que colaboraram para o meu crescimento acadêmico e profissional.

Ao meu orientador, Prof. Dr Gustavo W. Denardin, pela orientação, oportunidade, dedicação e principalmente, pelo apoio na etapa final deste trabalho.

Aos demais familiares e amigos, por todo o amparo e compreensão durante esta trajetória.

A todos que de alguma forma contribuíram para a realização deste trabalho.

Se não puder voar, corra. Se não puder correr, ande. Se não puder andar, rasteje, mas continue em frente de qualquer jeito.
Martin Luther King Jr.

RESUMO

Este trabalho demonstra o desenvolvimento de um *device driver* para o RTOS Nuttx, utilizando conceitos de portabilidade disponibilizados pela norma POSIX 2013.1. O Nuttx é um sistema operacional de tempo real com compatibilidade POSIX, no entanto, não possui metodologia de desenvolvimento de *driver* definida. Foi confeccionado um processo de desenvolvimento de *driver* do tipo *character device* e demonstrado sua aplicabilidade realizando a utilização do *device driver* em uma aplicação desenvolvida para o Nuttx. Optou-se por desenvolver um *device driver* da classe USB CDC/ACM, com o intuito de utilizar o recurso de emulação de uma porta serial virtual proveniente do componente ACM. Para tanto, fez-se necessária a segmentação do projeto em camadas, que permite abstrair a comunicação entre a controladora USB, o *device driver* implementado na camada de *driver* do Nuttx e a aplicação registrada no diretório apps. Ao concluir o desenvolvimento e o procedimento de configuração do *driver*, foi possível realizar a comunicação com o dispositivo conectado na porta USB disponibilizada no kit de desenvolvimento TM4C1294-Launchpad. A emulação de um terminal virtual ocorreu conforme o planejado, e devido ao desenvolvimento ter sido baseado na norma POSIX que descreve a portabilidade entre sistemas operacionais, o mesmo *device driver* pode ser exportado para outro RTOS POSIX. É possível visualizar que as *system calls* requisitadas na aplicação são processadas pelas funções POSIX desenvolvidas no *device driver*, e a partir disso pode-se concluir que o procedimento do desenvolvimento de *device driver* ocorreu de forma adequada. O trabalho traz contribuições para o desenvolvimento de aplicações e *device drivers* no ambiente do Nuttx. Adicionalmente, os códigos e todos os arquivos de configuração estão disponibilizados como código *open source* no Github à toda comunidade de desenvolvimento.

Palavras-chave: *Device Driver*. Sistemas Embarcados. POSIX. Nuttx.

ABSTRACT

This work demonstrates the development of a device driver for the RTOS Nuttx, using portability concepts provided by the POSIX 2013.1 standard. The Nutty is a real-time operating system with POSIX compatibility, nonetheless, it has no defined driver development methodology. A character device driver development process was made and its applicability using the device driver in an application developed for Nuttx. It was decided to develop a USB CDC/ACM class device driver, to use the emulation feature of a virtual serial port from the ACM component. Therefore, it was necessary to apply the project in layers, which abstract the communication between a USB controller, the device driver implemented in the Nuttx driver layer, and an application registered in the applications directory, which does not belong to the Nuttx core. However, requests to this directory can consume the resources made available by the RTOS. After completing the development and configuration procedure of the driver, it is possible to carry out a communication between the device connected to the USB port provided in the TM4C1294-Launchpad development kit. The application must be created using NuttShell and requested through the command line. The emulation of a virtual terminal went as expected because the development was about the POSIX standard that tests portability between operating systems, the same device driver can be exported to another POSIX RTOS. It is possible to see that the system calls requested in the application are processed by the POSIX functions developed in the device driver. It can conclude that the device driver development procedure is operating as waiting. The work brings contributions to the development of applications and device drivers in the Nuttx environment. As a contribution to the entire development community, the codes and any configuration files are made available as open-source on GitHub.

Keywords: Device Driver. Embedded System. POSIX. Nuttx.

LISTA DE FIGURAS

Figura 1 – Camadas de software 6
Figura 2 – Uma visão dividida do kernel 14
Figura 3 – Arquitetura Tiva 31
Figura 4 – Overview menuconfig do NuttX 33
Figura 5 – Tela inicial do Nuttx. 34
Figura 6 – Arquitetura de um Device Driver no NuttX 36
Figura 7 – Esquema USB CDC/ACM device driver. 41
Figura 8 – Ativando aplicação usbcdc no arquivo de configuração do Nuttx. . . 56
Figura 9 – Builtin Apps 57

LISTA DE QUADROS

Quadro 1 – Símbolos para open() 8
Quadro 2 – Retorno de erros da função posix_devctl() 13

LISTA DE ABREVIATURAS E SIGLAS

ACM	*Abstract Control Model*
API	*Application Program Interface*
CDC	*Communication Device Class*
IOCTL	*Input/Output Control*
MQUEUE	*Named Message Queue Interfaces*
POSIX	*Portable Operating System Interface*
RTOS	*Real Time Operating System*
USB	*Universal Serial Bus*
VFS	*Virtual File System*

LISTA DE ALGORITMOS

2.1 Exemplo de utilização da função open(). 8
2.2 Exemplo de utilização da função read(). 9
2.3 Exemplo de utilização da função write(). 10
2.4 Exemplo de utilização da função write(). 21
2.5 Macros definidas para dev_t 23
2.6 register_chrdev_region() e alloc_chrdev_region(). 23
2.7 Estrutura file_operations. 24
2.8 Exemplo de função read() utilizando o método copy_to_user 25
2.9 Exemplo de função write() utilizando o método copy_from_user 26
2.10 Exemplo de utilização da função ioctl(). 27
2.11 Exemplo de utilização da função ioctl() em user space. 27
2.12 Registro de um character device. 28
3.1 Importação dos cabeçalhos para o template do device driver 35
3.2 Protótipo das funções implementadas do driver 36
3.3 Atribuição dos ponteiros de função na estrutura file_operations 37
3.4 Implementação do método open 37
3.5 Protótipo da função register_driver 38
3.6 Implementação da função teste_driver 38
3.7 Alteração na função tm4c_bringup 38
3.8 Adição de inclusão no arquivo Makefile da board TivaC 39
3.9 Importação dos cabeçalhos para o driver usb cdc/acm 41
3.10 Adição no Make.defs da arquitetura tiva. 42
3.11 Importação dos cabeçalhos utilizados pela controladora usb 43
3.12 Diretivas de compilação utilizadas para configurar a controladora usb . 43
3.13 Mutex utilizados para proteger leitura e escrita 43
3.14 Ativação do modo interrupt handler 43
3.15 Implementação do método cdc_open 43
3.16 Implementação do método Virtual_Comm_Init 45
3.17 Alteração do método USBUARTPrimeTransmit 45
3.18 Implementação do método cdc_read 46
3.19 Implementação do método cdc_write 46
3.20 Implementação do método cdc_ioctl 47
3.21 Implementação do método principal usb_cdc 48
3.22 Adição dos arquivos .c no Makefile do TivaC 48
3.23 Alteração do método tm4c_bringup da board TivaC 48
4.1 Main aplicação usbcdc 51

4.2 usbtask 51
4.3 Utilização do método write na aplicação 52
4.4 Implementação do método usb_terminal_process 52
4.5 Arquivo de configuração Kconfig 55
4.6 Script Make.defs 55
4.7 Script Makefile 55

SUMÁRIO

1 – INTRODUÇÃO **1**

1.1 Objetivos 1

1.1.1 Objetivos específicos 1

1.2 Organização do trabalho 2

2 – REFERENCIAL BIBLIOGRÁFICO **3**

2.1 Sistemas Embarcados 3

2.2 RTOS 4

2.2.1 Núcleo de um RTOS 4

2.3 Portable Operating System Interface 5

2.3.1 POSIX e UNIX 5

2.3.2 Público 6

2.3.3 Propósito 6

2.4 POSIX e o conceito de abstração de hardware por arquivos 7

2.4.1 Descritores de arquivos 7

2.4.2 Abrindo um arquivo 8

2.4.2.1 Exemplo de utilização da função *open()* 8

2.4.3 Lendo de um arquivo 9

2.4.3.1 Exemplo de utilização da função `read()` 9

2.4.4 Escrevendo em um arquivo 9

2.4.4.1 Exemplo de utilização da função `write()` 10

2.4.5 Fechando um arquivo 10

2.5 Controle de dispositivos 10

2.6 POSIX.26 10

2.6.1 Conformidade de implementação 11

2.6.2 Documentação 11

2.6.2.1 Aplicação POSIX.26 estritamente conforme 11

2.6.3 Controle de dispositivos no padrão POSIX.26 12

2.6.3.1 Retorno de `posix_devctl()` 12

2.7 Linux 13

2.7.1 *Kernel* Linux 14

2.8 Nuttx 15

2.8.1 `<nuttx/arch>` 17

2.8.1.1 Resumo dos arquivos 17

2.8.2 `<nuttx/configs>` 17

2.8.2.1 Resumo dos arquivos 17

2.8.3 <nuttx/drivers> . . . 18
2.8.4 <nuttx/include> . . . 18
2.8.5 <nuttx/libs/libc> . . . 18
2.8.6 <nuttx/syscall> . . . 19
2.9 Sistemas de arquivos no Nuttx . . . 19
2.10 Device Drivers . . . 19
2.11 Device Drivers no Linux . . . 20
2.11.1 Construção de módulos . . . 21
2.11.2 *Character Devices* . . . 22
2.11.3 *Major Number* e *Minor Number* . . . 23
2.11.4 Operações de arquivos . . . 24
2.11.4.1 Função *open()* . . . 25
2.11.4.2 Função *release()* . . . 25
2.11.4.3 Troca de dados com *User Space* . . . 25
2.11.4.4 Função *read()* . . . 25
2.11.4.5 Função *write()* . . . 26
2.11.5 Operações avançadas de um *Character Device* . . . 27
2.11.6 Registrando um *Character Device* . . . 28

3 – DESENVOLVIMENTO . . . **30**
3.1 Preparação do ambiente de desenvolvimento . . . 30
3.1.1 Kit de Desenvolvimento Tiva C . . . 30
3.1.2 Pacote lm4flash . . . 31
3.1.3 Putty . . . 31
3.2 Preparação do Ambiente . . . 32
3.3 Configurando o NuttX para o Tiva C . . . 32
3.4 Implementação de um Device Driver genérico no Nuttx . . . 34
3.5 Implementação USB CDC/ACM no Nuttx . . . 39

4 – RESULTADOS OBTIDOS . . . **50**
4.1 Implementação de uma Aplicação Nuttx . . . 50
4.2 Análise dos Resultados . . . 56

5 – CONCLUSÃO . . . **58**
5.1 Trabalhos Futuros . . . 59
5.2 Considerações Finais . . . 59

Referências . . . **60**

1 INTRODUÇÃO

A função de um *device driver* é aceitar requisições abstratas de *software* e cuidar para que as solicitações sejam executadas, permitindo que uma determina aplicação interaja com o periférico. Os *drivers*, em geral, apresentam um conjunto de funções que permite que as aplicações tenham acesso aos recursos oferecidos pelos periféricos. Por esse motivo, todos os sistemas operacionais apresentam determinado padrão na construção de seus *drivers*. Possuindo uma interface padrão entre as aplicações e os dispositivos, o *kernel* pode-se comunicar com qualquer tipo de *driver*, mesmo que desconhecendo-o em tempo de compilação (MORAES; ALMEIDA; SERAPHIM, 2016).

Um *device driver* comumente não pode ser considerado portável em sí, pelo fato de que um *device driver* pode ser visto por meio de camadas e uma dessas camadas é totalmente depende do *hardware*. Entretanto, um aplicativo que utiliza esse *driver* pode ser feito portável se todas as interfaces necessárias forem bem definidas e padronizadas. Dessa forma, é possível afirmar que a função de um *driver* é fornecer mecanismos (o que precisa ser feito) e não políticas (como o programa pode ser usado).

É necessário escrever o código em *kernel space* para acessar o *hardware*, mas a aplicação não deve ser obrigada a aceitar políticas específicas, devido ao fato que cada aplicação possui suas necessidades. O *driver* deve apenas lidar com o *hardware*, deixando as questões de como utilizá-lo para os aplicativos. Nesse caso, é dito que um *driver* é flexível se ele fornece acesso aos seus recursos sem impor limitações (RUBINI; CORBET; HARTMAN, 2005).

A principal vantagem de utilizar um padrão para desenvolvimento de *device drivers* é reutilizar o código existente, sem precisar reescrever o *driver* do início. Por exemplo, para uma série de microcontroladores de determinada fabricante, não é conveniente que o mesmo *driver* seja reescrito para cada microcontrolador. É melhor que o código dependente de *hardware* seja implementado em um módulo separado do código que utiliza e inicializa o *dispositivo*. Dessa forma, a aplicação que utiliza os serviços oferecidos pelo *driver* pode ser utilizada em todos os microcontroladores.

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver uma metodologia para adequação de camadas de abstração de *hardware* de microcontroladores para confecção de *devices drivers* em RTOSs baseados no padrão POSIX.

1.1.1 Objetivos específicos

1. Realizar revisão bibliográfica do modelo de *driver* utilizado no padrão POSIX e no RTOS NuttX;

2. Analisar as camadas de abstração de *hardware* implementadas com a biblioteca TivaWare;
3. Adaptar a camada de abstração de *hardware* de um periférico do microcontrolador TM4C1294 para o padrão POSIX;
4. Descrever uma metodologia de desenvolvimento de *drivers* no NuttX;

1.2 Organização do trabalho

Esse trabalho inicia com uma breve abordagem sobre sistemas embarcados e suas aplicações na Seção 2.1. Em seguida, são abordados os conceitos gerais do padrão POSIX e da extensão POSIX.26 na Seção 2.3 e Seção 2.4 respectivamente. Posteriormente, são abordados conceitos gerais do RTOS NuttX na Seção 2.6. Finalizando o Capítulo 2, a Seção 2.10 aborda o desenvolvimento de *character device* no Linux, visto que a implementação de POSIX em Linux é uma das mais difundidas na literatura.

2 REFERENCIAL BIBLIOGRÁFICO

Este capítulo contém a base teórica utilizada para o desenvolvimento do trabalho e os conceitos necessários para a compreensão do contexto e dos objetivos do que será desenvolvido. Inicialmente abordam-se os sistemas embarcados e suas aplicações, o padrão POSIX aplicado a concepção de *drivers* de dispositivos e o detalhamento do uso de tal padrão no sistema operacional Linux. Em seguida são abordados conceitos sobre o RTOS NuttX, comentando-se diferenças entre o NuttX e Linux e detalhando a metodologia de desenvolvimento de *driver* do tipo *character* no Linux.

2.1 Sistemas Embarcados

A popularização dos microprocessadores iniciou em meados de 1980 e deu início a uma revolução tecnológica. Os processadores que antes eram restritos a computadores de propósito geral, como os de uso corporativo, hoje podem ser encontrados nos dispositivos com as mais diversas funcionalidades e recursos, como: relógios de pulso, celulares, aparelhos de som, televisores, eletrodomésticos e automóveis. A utilização de microprocessadores em dispositivos que não são de propósito geral deu origem ao que é conhecido hoje por Sistemas Embarcados (ZURITA, 2014).

Um sistema embarcado é um sistema computacional completamente encapsulado e dedicado ao dispositivo ou sistema que controla. Diferente de um computador de propósito geral, como um computador de uso pessoal, um sistema embarcado realiza tarefas que possuem requisitos e tarefas predefinidas. Já que o sistema é dedicado a tarefas específicas, o *hardware* não precisa ter o desempenho de um computador de propósito geral, sendo possível otimizá-lo, reduzindo o tamanho, o custo tecnológico e os recursos computacionais (DENARDIN; BARRIQUELO, 2019).

Segundo Denardin e Barriquelo (2019), alguns exemplos de sistemas embarcados são:

a. automotivos: controle de injeção eletrônica, controle de tração, controle de sistemas de frenagem anti-bloqueio (ABS) e *etc*;
b. comunicação: telefones celulares, roteadores, equipamentos de GPS, *etc*;
c. robótica: robôs industriais, drones, *etc*;
d. aeroespacial e militar: sistemas de gerenciamento de voo, controle de armas de fogo, *etc*;
e. controle de processos: processamentos de alimentos, controle de plantas químicas e controle de manufaturas em geral;
f. domésticos: micro-ondas, lavadouras de louça, lavadoura de roupa, *etc*.

O projeto de um sistema embarcado é uma tarefa complexa, por envolver questões de portabilidade, necessidade de apresentar um bom equilíbrio entre consumo e desempenho e entre segurança e confiabilidade (CARRO; WAGNER, 2003). Para o programador é essencial

conhecer os fundamentos de organização e arquitetura do dispositivo.

Além do *hardware*, deve-se considerar o projeto do *software* embarcado, conhecido por *firmware*. O *firmware* é um conjunto de regras que rege o funcionamento do sistema. A essência de um sistema embarcado é conter um *firmware* que controle a sua execução. De nada adianta ter um *hardware* bem estruturado e não ter um *software* bem planejado para controlá-lo (OLIVEIRA; ANDRADE, 2006). Muitos dos sistemas embarcados são sistemas em tempo real e, portanto, o sistema operacional usado nesses sistemas deve ser um sistema operacional de tempo real (RTOS, do inglês *Realtime operating system*).

2.2 RTOS

De acordo com Denardin e Barriquelo (2019), RTOS é uma subclasse de sistemas operacionais destinados a concepção de sistemas computacionais, geralmente embarcados, em que o tempo de resposta a um evento é fixo e deve ser respeitado sempre que possível. São sistemas que possuem requisitos específicos de sequência lógica e de tempo que, se não cumpridos, resultam em falha no sistemas a que se dedicam. É notável ressaltar que tempo real não está associado com velocidade, mas com o cumprimento de prazos de todos os eventos controlados pelo sistema.

Sistemas operacionais de tempo-real estão associados aos requisitos que possuem. Nesse aspecto, é possível dividir em duas categorias: *hard real-time* ou *soft real-time*. Os sistemas *hard real-time* possuem tempos críticos e são intolerantes a atrasos, devendo ser realizados dentro de um intervalo de tempo específico, sob pena de falha na execução das tarefas, o que pode levar a resultados desastrosos. Por outro lado, as tarefas *soft real-time* são mais tolerantes, pois apesar de possuírem um intervalo de tempo máximo para serem executadas, não geram problemas graves no sistema no caso do intervalo de tempo não ser cumprido (BORGES, 2011).

2.2.1 Núcleo de um RTOS

O componente mais importante de um RTOS é o seu núcleo (mais conhecido pelo termo em inglês, *kernel*). O núcleo gerencia os recursos encontrados no sistema embarcado, incluindo o processador, a memória e o temporizador do sistema. As principais funções do núcleo incluem o gerenciamento de tarefas, a sincronização e a comunicação entre tarefas, o gerenciamento de tempo e o gerenciamento de memória (MARWEDEL, 2011). Segundo Denardin e Barriquelo (2019), de acordo com o método de gerenciamento de tarefas utilizado, um núcleo pode ser preemptivo ou não preemptivo.

a. núcleo não preemptivo: Os núcleos não preemptivos também são conhecidos como sistemas cooperativos. Nenhum evento externo pode ocasionar a perda do uso do processador, ou seja, quando uma tarefa ganha o direito de utilizar o processador, nenhuma outra tarefa pode lhe tirar esse recurso. Esse tipo de sistema requer que cada

tarefa desista explicitamente do controle de uso do processador para que outra tarefa seja executada, ou seja, as tarefas cooperam para que todas tenham acesso ao processador. A requisição para desistir voluntariamente do processador é conhecida como *yield* (”ceder”o processador). Para que possua um bom comportamento de multitarefa, essas requisições devem ser realizadas frequentemente por todas as tarefas no sistema.

b. núcleo preemptivo: Nesses sistemas uma tarefa pode perder o controle de uso do processador durante sua execução caso o tempo de execução destinado a tarefa tenha expirado, uma tarefa de maior prioridade esteja pronta para execução, ou devido a qualquer outra regra de operação utilizada pelo sistema. A cada interrupção, exceção ou chamada de sistema, o escalonador pode reavaliar todas as tarefas em estado de pronto para execução e decidir se irá manter ou substituir a tarefa atual em execução.

Enquanto alguns RTOSs são projetados para aplicações embarcadas gerais, outros possuem uma área específica de aplicação. Por exemplo, sistemas operacionais compatíveis com OSEK/VDX possuem foco no controle automotivo. Portanto, o sistema operacional de uma área específica fornece um serviço dedicado que pode ser mais compacto e eficiente que um sistema operacional projetado para várias áreas de aplicações. Alguns RTOSs fornecem uma *Application Programming Interface* (API) padrão, outros disponibilizam um API proprietária. Por exemplo, alguns RTOSs são compatíveis com o padrão POSIX RT (MARWEDEL, 2011).

2.3 Portable Operating System Interface

Portable Operating System Interface 2013.1 (POSIX) é um padrão IEEE que auxilia na compatibilidade e na portabilidade entre sistemas operacionais. Teoricamente, o código fonte compatível com POSIX deve ser perfeitamente portátil. Na prática, a transição de aplicativos geralmente se depara com problemas específicos do sistema. A conformidade com POSIX simplifica a portabilidade de aplicativos.

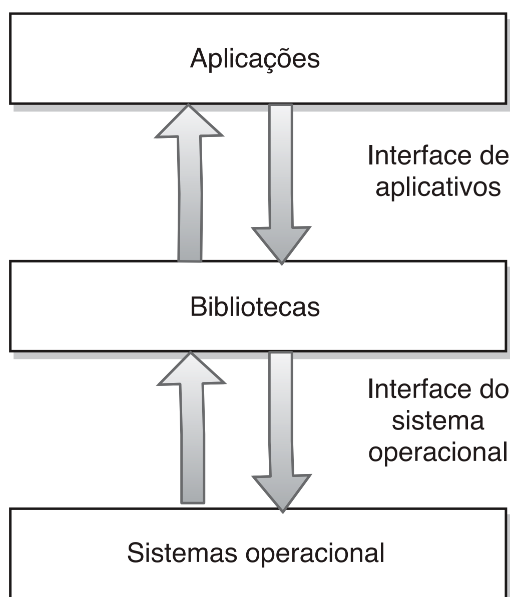
Conceitualmente, o padrão base POSIX descreve um conjunto de serviços fundamentais necessários para a construção eficiente de programas aplicativos. O acesso a esses serviços foi fornecido pela definição de APIs, usando a linguagem C, que estabelece semântica e sintaxe padrão. O objetivo dessas interfaces é permitir que os desenvolvedores de aplicativos possam criar aplicativos portáveis (LEWINE, 1991).

2.3.1 POSIX e UNIX

POSIX é baseado no UNIX System V e no Berkeley UNIX, mas não é um sistema operacional. Em vez disso, POSIX define uma interface entre aplicativos e as bibliotecas. POSIX não versa sobre chamadas de sistemas (*system calls*) ou faz qualquer distinção entre *kernel* e usuário. Os fornecedores podem adaptar variantes do UNIX, ou outro sistema operacional, para fornecer interfaces POSIX. Os aplicativos podem ser movidos de um sistema para outro porque eles veem apenas a interface POSIX e não tem ideia do sistema operacional que está sob a

mesma. Uma implementação consiste em um conjunto de bibliotecas e um sistema operacional. POSIX define apenas a interface entre os aplicativos e as bibliotecas (LEWINE, 1991).

Figura 1 – Camadas de software



Fonte: **traduzido de Lewine (1991,p.78).**

O POSIX define como o aplicativo se comunica com a biblioteca e como a biblioteca e o sistema operacional subjacente se comportam em resposta. Cada implementação pode ter sua própria maneira de dividir o trabalho entre a biblioteca e o sistema operacional.

2.3.2 Público

O público alvo das normas sobre o padrão POSIX são todas as pessoas preocupadas com um sistema operacional padrão baseado no sistema UNIX. Isso inclui pelo menos quatro grupos de pessoas:

a. Pessoas que compram sistemas de *hardware* e *software*;
b. Pessoas gerenciando empresas que estão decidindo sobre futuras direções de computação corporativa;
c. Pessoas que implementam sistemas operacionais;
d. Pessoas que desenvolvem aplicações em que a portabilidade é um objetivo.

2.3.3 Propósito

Vários princípios orientam o desenvolvimento dos padrões POSIX em geral. De acordo com Lewine (1991), o propósito do padrão pode ser descrito com os itens a seguir:

a. Orientado a aplicação: o objetivo básico é promover a portabilidade de programas de aplicativos em ambientes de sistemas UNIX, desenvolvendo um padrão claro, consistente

e não ambíguo para a especificação de interface de um sistema operacional portátil baseado na documentação do sistema UNIX;

b. Interface: os padrões POSIX definem uma interface, não uma implementação. Nenhuma distinção é feita entre as funções de biblioteca e as chamadas do sistema, ambas são referidas como funções. Nenhum detalhe de implementação de qualquer função é fornecido;

c. Fonte: os padrões POSIX foram escritos para que um programa escrito e traduzido para execução em uma implementação em conformidade também pode ser traduzido para execução em outra implementação em conformidade. Portanto, a portabilidade do código-fonte é limitada aos sistemas com os mesmos *drivers* de dispositivos especiais. Entretanto, se esse não for o caso, o uso deste padrão pode melhorar a portabilidade, tornando as partes não portáteis que acessam dispositivos altamente visíveis e uniformemente utilizados. Esse padrão não garante que o código executável (objeto ou binário) será executado sob uma implementação em conformidade diferente daquela para a qual ele foi traduzido, mesmo se o *hardware* subjacente for idêntico;

d. Sem super usuário: instalações e funções de administração do sistema foram excluídas do POSIX, padrões básicos e funções utilizáveis apenas pelo super-usuário não foram inclusas. Entretanto, uma implementação de interface padrão também pode implementar recursos fora do POSIX. O padrão POSIX não se preocupa com restrições de *hardware* ou manutenção do sistema.

2.4 POSIX e o conceito de abstração de hardware por arquivos

Um arquivo de dispositivo é uma abstração de um *device driver* como se fosse um arquivo comum. Neste caso, é possível se comunicar em *user space* com um *device driver* por meio de operações em um arquivo que normalmente disponibilizado em */dev*. Essas operações são representadas pela execução de uma função que gera uma *system call*. Uma *system call* basicamente é uma maneira de as aplicações interagirem com o sistema operacional, solicitando um serviço do *kernel*.

Esta Seção aborda as chamadas básicas dos sistemas POSIX, como `read()`, `write()`, `close()`, `open()`.

2.4.1 Descritores de arquivos

Um descritor de arquivo é um inteiro pequeno não negativo, usado para identificar um arquivo aberto. Seu valor é atribuído em ordem (0,1,2,3..) por processo. O número máximo de descritores de arquivos abertos é limitado, o limite é dado pelo simbolo `OPEN_MAX` no arquivo de cabeçalho `limits.h`.

2.4.2 Abrindo um arquivo

A conexão entre descritores de arquivos e arquivos é definida pelas funções `open()` e `creat()`. A função `open()` é usada para atribuir um descritor de arquivo a um arquivo novo ou existente. A função é definida como:

```
1 int open(const char *path, int oflag, ...);
```

O terceiro argumento é opcional e é chamado de `mode_t`, podendo ser usado para definir os bits de permissão do arquivo quando ele é criado.

O argumento `path` é uma *string* que representa o nome do arquivo a ser aberto. Pode ser um caminho absoluto ou relativo.

O argumento `oflag` é uma OR bit a bit de inclusão dos valores de constantes simbólicas. Podem ser especificados os símbolos a seguir:

Quadro 1 – Símbolos para open().

Símbolo	Descrição
O_RDONLY	Aberto para leitura somente
O_WRONLY	Aberto para escrita somente
O_RDWR	Aberto para leitura e escrita
O_APPEND	Define o deslocamento do arquivo para o final do arquivo antes de cada operação de escrita
O_CREAT	Se o arquivo não existe, permite que ele seja criado.
O_EXCL	Se O_CREAT estiver configurado, faz com que a chamada *open()* falhe se o arquivo já existir.
O_NOCTTY	Se path identifica um terminal, este flag impede que o terminal se torne o terminal de controle para este processo.
O_TRUNC	Esse sinalizador deve ser usado apenas em arquivos comuns abertos para gravação. Isso faz com que o arquivo seja truncado para comprimento zero.

Fonte: **Lewine (1991)**

O programador deve especificar pelo menos um dos três símbolos: `O_RDONLY`, `O_WRONLY` ou `O_RDWR`.

2.4.2.1 Exemplo de utilização da função *open()*

O exemplo a seguir abre um arquivo para escrita, criando o arquivo caso ele não exista. Se o arquivo já existe, o sistema trunca o arquivo para zero bytes.

Listagem 2.1 – Exemplo de utilização da função open().

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define LOCKFILE "/etc/ptmp"
6 ...
7 int pfd; /* Integer for file descriptor returned by open() call. */
8 ...
```

```
9 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR
    ↪ )) == -1)
10 {
11  fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
12  exit(1);
13 }
14 ...
```

2.4.3 Lendo de um arquivo

A única função de baixo nível para ler de um arquivo é a função `read()`. É definida como:

```
1 ssize_t read(int fildes, void *buf, size_t nbyte);
```

A função `read()` lê `nbyte` *bytes* do arquivo aberto em `fildes` no *buffer* `buf` e retorna o número de bytes colocados no *buffer*. Este valor nunca é maior que `nbyte` e será menor que `nbyte` se o arquivo possuir menos bytes disponíveis imediatamente para leitura. O argumento `nbyte` possui o tipo `size_t` no IEEE Std 1003.1-1990. Se houver um erro, o valor de -1 será retornado e `errno` será definido.

2.4.3.1 Exemplo de utilização da função read()

O exemplo a seguir lê dados do arquivo associado ao descritor fd no *buffer* apontado por `buf`.

Listagem 2.2 – Exemplo de utilização da função read().

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 ...
4 char buf[20];
5 size_t nbytes;
6 ssize_t bytes_read;
7 int fd;
8 ...
9 nbytes = sizeof(buf);
10 bytes_read = read(fd, buf, nbytes);
11 ...
```

2.4.4 Escrevendo em um arquivo

A função `write()` executa operação de escrita em um arquivo e é definida como:

```
1 ssize_t write(int fildes, const void *buf, size_t nbyte);
```

A função `write()` escreve `nbyte` *bytes* do *buffer* apontado por `buf` no arquivo aberto em `fildes`. Retorna o número de bytes gravados no arquivo, que pode ser menor que `nbyte` se ocorrer um erro durante a operação de escrita o valor de -1 será retornado.

2.4.4.1 Exemplo de utilização da função `write()`

O exemplo a seguir realiza operação de escrita de dados no *buffer* apontador por `buf` para o arquivo associado ao descritor de arquivo `fildes`.

Listagem 2.3 – Exemplo de utilização da função write().

```
1  #include <sys/types.h>
2  #include <string.h>
3  ...
4  char buf[20];
5  size_t nbytes;
6  ssize_t bytes_written;
7  int fildes;
8  ...
9  strcpy(buf, "This is a test\n");
10 nbytes = strlen(buf);
11
12 bytes_written = write(fildes, buf, nbytes);
13 ...
```

2.4.5 Fechando um arquivo

A função `close()` é utilizada para desalocar o descritor de um arquivo e limpar o arquivo quando o seu uso tiver sido finalizado.

```
1 int close(int fildes);
```

Existem boas razões para chamar explicitamente *close()* para cada arquivo:

i. Arquivos aberto são recursos limitados. É ideal devolvê-los o quantos antes.

ii. É sempre bom ser cautoloso quanto aos erros. Se o programador permitir que a função `exit()` feche os arquivos abertos, os erros serão ignorados.

A função `close()` é o mais portável possível.

2.5 Controle de dispositivos

A função `ioctl()` executa uma variedade de funções de controle nos dispositivos *streams*.

```
1 int ioctl (int fildes, int request, ... / * arg * /);
```

O argumento `fildes` é um descritor de arquivo aberto que se refere a um dispositivo. O argumento `request` seleciona a função de controle a ser executada e depende do dispositivo `stream` que está sendo endereçado. O argumento `arg` representa as informações que podem ser necessárias do dispositivo `stream` específico para executar uma função solicitada.

2.6 POSIX.26

Esse padrão define extensões para o POSIX.1, para suportar a portabilidade de aplicativos no nível código-fonte. Tem como público desenvolvedores de aplicativos e implementadores

de sistemas. O escopo geral deste padrão é definir uma interface de aplicativo portátil para aplicativos com restrições em tempo real que exijam capacidade de controlar dispositivos especiais.

POSIX.26 foi definido exclusivamente no nível de código-fonte. Além disso, embora as interfaces sejam portáteis, alguns dos parâmetros usados por uma implementação podem ter dependências de *hardware* (IEEE STD 1003.26, 2003).

2.6.1 Conformidade de implementação

De acordo com (IEEE STD 1003.26, 2003), uma implementação em conformidade com esse padrão deverá atender à todos os requisitos a seguir:

i. a implementação deve estar em conformidade com POSIX.1;
ii. o sistema deve suportar todas as interfaces definidas na norma. Todas as interfaces suportadas devem suportar o comportamento funcional descrito;
iii. o sistema pode fornecer funções adicionais ou instalações não especificadas em i. ou ii. Extensões fora do padrão devem ser identificadas na documentação do sistema. O documento de conformidade deve definir um ambiente no qual um aplicativo pode ser executado com o comportamento especificado pelo padrão. Em nenhum caso tal ambiente exigirá a modificação de uma aplicação POSIX.26 estritamente conforme.

2.6.2 Documentação

Um documento de conformidade com as seguintes informações deve estar disponível para uma implementação que declare conformidade com esta norma. O documento de conformidade deve ter a estrutura de um anexo ao documento de conformidade POSIX.1 requerido. O documento de conformidade não deve conter informações sobre instalações ou recursos estendidos fora do escopo desta norma.

2.6.2.1 Aplicação POSIX.26 estritamente conforme

Um aplicativo POSIX.26 estritamente em conformidade é um aplicativo que está estritamente conforme definido em POSIX.1, exceto pelo fato de que é permitido usar as interfaces especificadas neste padrão com as restrições de que (IEEE STD 1003.26, 2003). As aplicações que reivindicam conformidade com este padrão devem estar de acordo com os seguinte parâmetros:

i. aceitará qualquer comportamento de implementação que resulte de ações realizadas em áreas descritas neste padrão como definidas pela implementação ou não especificadas, ou quando este padrão indicar que as implementações podem variar;
ii. não executará ações descritas como resultados indefinidos;
iii. para constantes simbólicas, deve-se aceitar qualquer valor no intervalo permitido por POSIX.26, mas não deve confiar em qualquer valor no intervalo que seja maior que os

mínimos listados, ou seja, menor que os máximos listados neste padrão;

iv. não deve usar instalações designadas como obsoletas;

v. é necessário tolerar e permitir a adaptação à presença ou ausência de recursos opcionais definidos no POSIX.1;

vi. para a linguagem de programação C, deve definir _POSIX_26_C_SOURCE como 200312L antes de qualquer cabeçalho ser incluído. Dentro deste padrão, quaisquer restrições impostas a um aplicativo POSIX.26 em conformidade devem restringir um aplicativo POSIX.26 em conformidade estrita;

2.6.3 Controle de dispositivos no padrão POSIX.26

O maior problema com a função `ioctl()` é que o terceiro argumento é ponteiro genérico para um objeto na memória que varia de tamanho e tipo de acordo com o segundo argumento. A interface `posix_devctl()` melhora `ioctl()`, pois permite que o usuário especifique o tamanho do objeto. Um problema secundário com `ioctl()` é que o terceiro argumento é permitido ser interpretado como um inteiro.

A interface `posix_devctl()` definida no padrão POSIX.26 fornece uma alternativa para as implementações de `ioctl()` com uma interface padrão que captura as extensões de `ioctl()`, mas evita várias deficiências, que são discutidas a seguir. A função `posix_devctl()` é definida da seguinte maneira:

```
1 #include <devctl.h>
2
3 int posix_devctl(int fildes,
4       int dcmd,
5       void *restrict dev_data_ptr,
6       size_t nbyte,
7       int *restrict dev_info_ptr
8 );
```

A execução do método fará com que o argumento `dcmd` seja transmitido para o *driver* que está identificado em `fildes`. Se o argumento `dev_data_ptr` não for um ponteiro que aponta para NULL, deverá ser um ponteiro para um *buffer* que é fornecido pela aplicação que invoca o método, e contém os dados a serem transmitidos para o *driver* ou fornece um *buffer* para receber dados do *driver*.

O argumento `dev_info_ptr` fornece a oportunidade de retornar um número inteiro que contém informações adicionais sobre o dispositivo, além de sucesso e falha.

2.6.3.1 Retorno de `posix_devctl()`

Semelhante ao retorno de `ioctl()`, `posix_devctl` deverá retornar zero após uma operação bem sucedida. Caso contrário, um número de erro deve ser retornado para que possa ser interpretado. O valor de retorno em situações de erro é dependente do *driver* e é transmitido por meio do argumento `dev_info_ptr`.

A função `posix_devctl()` deverá falhar nas situações descritas a seguir e retornar os erros correspondentes:

Quadro 2 – Retorno de erros da função posix_devctl().

ERRO	Genitura
EBADF	O argumento `fildes` não é um descritor de arquivo aberto válido
EINTR	A função texttposix_devctl() for interrompida por um sinal
EINVAL	O argumento `nbyte` é negativo ou excede um máximo definido pela implementação ou é menor do que o número mínimo de *bytes* necessários para este comando
EINVAL	Se o argumento transmitido em `dcmd` não possuir corresponde na implementação do *driver*
ENOTTY	O argumento `fildes` não está associado à um arquivo de caractere que aceita operações de controle
EPERM	O processo que realiza a chamada do método não possuí privilégio para executar determinado comando

Fonte: **Lewine (1991)**

Se a função falhar, o efeito desta função que falhou é totalmente dependendo do *driver*. Os dados correspondentes podem ser transferidos, parcialmente transferidos ou não transferidos. Outros erros podem ser detectados, mas os números de erros retornados são dependentes do *driver* e devem estar documentados.

2.7 Linux

Linux começou como um hobby de um estudante finlandês chamado Linus Torvalds em 1991, mas rapidamente se transformou em um sistema operacional avançado e um dos mais utilizados no mundo. Desde o primeiro lançamento sob o processador Intel 386, o *kernel* cresceu gradualmente em complexidade para suportas diversas arquiteturas. Algumas das principais arquiteturas suportadas são: x86, IA64, ARM, PowerPC, Alpha, s390, MIPS e SPARC. Embora o propósito inicial era ser um sistemas operacional para *desktop*, o Linux penetrou no mundo corporativo e dos sistemas embarcados. (VENKATESWARAN, 2008).

O Linux é precedido pelo projeto GNU (GNU é um acrônimo para GNU's Not UNIX), cujo objetivo primário é oferecer um sistema operacional livre, de código aberto e compatível com UNIX. Um sistema operacional GNU é composto por um *kernel* Linux, mas também contém componentes como um compilador de linguagem C, editores de texto, formatadores de texto, clientes de e-mail, interfaces gráficas, bibliotecas, jogos e entre outros. Todos os componentes presente em sistema GNU/Linux são construídos usando software livre.

Originalmente, o *kernel* oficial do projeto GNU era o GNU Hurd, que é uma coleção de processos servidores rodando em cima do Mach. Mach é um micronúcleo desenvolvido na Carnegie Mellon University e depois na Universidade de Utah. O GNU Hurd é um conjunto de servidores que executam sobre o Mach e proveem vários serviços atribuídos ao núcleo do

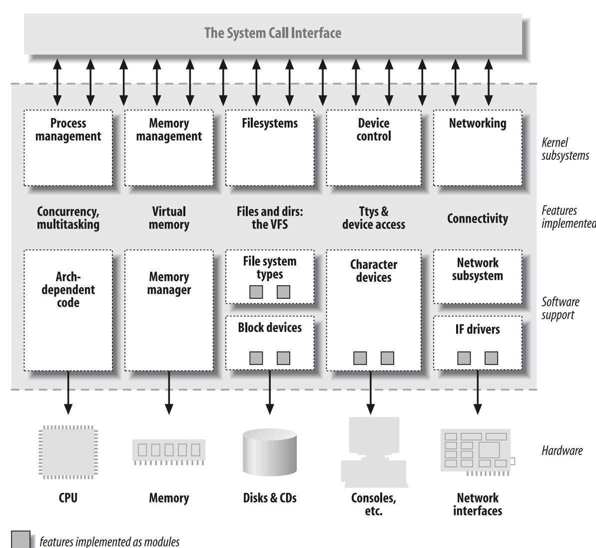
Unix. Entretanto, o Linux foi disponibilizado em 1992, antes da conclusão do GNU Hurd. A combinação do *kernel* Linux com o sistema GNU resultou no sistema completo denominado GNU/Linux (STALLMAN, 2019).

Existem diversas diretrizes sob *softwares* livres, uma delas é denominada como domínio público. Todo *software* liderado sob domínio público não é protegido por direitos autorais e nenhuma restrição é imposta ao seu uso. É possível utilizar gratuitamente, realizar alterações e até mesmo restringir a distribuição de suas fontes modificadas. A Free Software Foundation, principal patrocionadora do GNU, criou o GNU public license (GPL) que impede a possibilidade de intermediários transformarem um software livre em *software* proprietário. Qualquer modificação em um *software* sob a licença GPL deve ser disponibilizada para a comunidade. O *kernel* Linux e a maioria dos componentes de um sistema GNU são liberados sob a licença GPL (VENKATESWARAN, 2008).

2.7.1 *Kernel* Linux

Em um sistema derivado UNIX, vários processos simultâneos são empregados na realização de tarefas distintas. Cada processo exige recursos de sistema, seja na capacidade de processamento, memória, conexão de rede e entre outros recursos. O *kernel* é responsável pelo código que gerencia estas exigências. A despeito que as variações de procedimento entre as diferentes tarefas do *kernel* não possam ser facilmente percebidas, a função do *kernel* pode ser dividida. De acordo com Rubini, Corbet e Hartman (2005) a divisão pode ser realizada da seguinte maneira.

Figura 2 – Uma visão dividida do kernel



Fonte: **adaptado de Rubini, Corbet e Hartman (2005)**

1. Gerenciamento de processo: o *kernel* é responsável pela criação e extinção de processos e lida com sua conexão com dispositivos de entrada e saída. A comunicação entre processos

é fundamental para o funcionamento global do sistemas e também é controlada pelo *kernel*. Além disso, o escalonamento de tarefas, provavelmente a etapa mais importante de todo o sistema operacional, faz parte do processo de gerenciamento.

2. Gerenciamento de memória: a memória de um computador é um recurso muito importante e a política estabelecida para lidar com ela é de suma importância para o funcionamento do sistema. Todas as subdivisões do *kernel* interagem com um subsistema de gerenciamento de memória utilizando um conjunto de chamadas de função, que variam de recursos simples como *malloc* e *free* até funcionalidades mais complexas;
3. Sistemas de arquivo: o Unix é fortemente baseado em um conceito de sistemas de arquivos e quase tudo no Unix pode ser tratado como arquivo. Basicamente sistemas de arquivos são formas diferentes de organizar os dados no meio físico. O *kernel* constrói um sistema de arquivos sob um software sem estrutura e a abstração resultante é constantemente usada por todo o sistema. O Linux oferece suporte a diversos sistemas de arquivos;
4. Controle de dispositivo: com exceção do processador, da memória e de algumas entidades especificas, todas e quaisquer operações de controle são executadas por um código que é específico para um determinado dispositivo. Esse código é chamado de *device driver*. O *kernel* deve possuir um *device driver* para cada periférico de seu sistema, desde o disco rígido até o teclado.
5. Rede: o sistema operacional é responsável pela passagem dos pacotes de dados entre aplicações e interfaces de rede e deve desativar e ativar corretamente os programas que esperam pelos dados na rede. As questões referentes a roteamento e definição de endereço são executadas dentro do *kernel*.

Abordar os aspectos de sistemas de arquivos e controle de dispositivos é um dos principais objetivos deste trabalho.

2.8 Nuttx

Existe uma disponibilidade muito grande de RTOSs no mercado. Entretanto, muitos deles não tiveram continuidade ao seu desenvolvimento. Dos demais RTOSs, a maioria só suporta um microcontrolador ou uma família de microcontroladores. Fazendo uma filtragem mais profunda, nota-se que os poucos RTOSs que suportam múltiplos microcontroladores não possuem todos os recursos que empresas e desenvolvedores necessitam, como: pilhas de protocolos USB, pilhas de protocolos TCP/IP, Wi-Fi, LCD gráfico, SD Card, Sistema de Arquivo FAT, RS485, etc (ASSIS., 2019). Entre esses poucos RTOS que se destacam, existe o NuttX que possui todos os recursos citados e outros, além de suportar várias arquiteturas de microcontroladores e microprocessadores.

O NuttX é um RTOS desenvolvido por Gregory Nutt e disponibilizado pela primeira vez em 2007 sob a licença BSD permissiva. É escalável de microcontroladores de 8 bits à 32 bits, sendo os principais padrões utilizados são POSIX e ANSI. APIs padrões adicionais do Unix e de outros RTOS comuns são adotadas para funcionalidades não disponíveis sob esses padrões,

ou para funcionalidades que não são apropriadas para ambientes profundamente incorporados.

Segundo Nutt (2019), seus objetivos são:

a. *Footprint* pequeno: o NuttX possui um requisito de memória muito pequeno que pode ser atendido em qualquer caso de aplicação; são úteis para sistemas práticos;
b. Conjunto de recursos avançados de SO: o objetivo é fornecer implementações da maioria das interfaces padrão POSIX OS para suportar um ambiente de desenvolvimento rico e multiencadeado para processadores profundamente integrados;
c. Altamente escalável: totalmente escalável de 8 bits até 32 bits. A escalabilidade com o conjunto de recursos avançado é realizada com: muitos arquivos de origem pequenos, links de bibliotecas estáticas altamente configuráveis e uso de símbolos fracos quando disponíveis;
d. Conformidade de padrões: o NuttX visa alcançar um alto grau de conformidade com os padrões. Os principais padrões são POSIX e ANSI. Devido a essa conformidade com os padrões, o software desenvolvido sob outros sistemas operacionais sob esses padrões (como o Linux) deve ser facilmente transportado para o NuttX;
e. Tempo real: totalmente preemptivo; prioridade fixa, *roud-robin* e programação esporádica;
f. Totalmente aberto: licença BSD não restritiva.
g. GNU Toolchains: os toolchains GNU compatíveis são baseados em buildroot e estão disponíveis para download. Fornecem um ambiente de desenvolvimento completo para diversas arquiteturas.

```
|-- nuttx
|    |-- Makefile
|    |-- Documentation/
|    |-- arch/
|    |-- audio/
|    |-- binfmt/
|    |-- boards/
|    |-- crypto/
|    |-- drivers/
|    |-- dummy/
|    |-- graphics/
|    |-- mm/
|    |-- net/
|    |-- openamp/
|    |-- pass1/
|    |-- sched/
|    |-- staging/
|    |-- syscall/
|    |-- tools/
|    |-- video/
|    |-- wireless/
|    |-- fs/
|    |-- include/
|    |-- libs/
|    |-- syscall/
```

2.8.1 <nuttx/arch>

Esse diretório e seus subdiretórios contêm a lógica específica de cada arquitetura. O *port* da placa é definido pelo código específico contido neste diretório. Cada arquitetura deve fornecer um subdiretório <arch-name> sob o <arch/> com as seguintes características:

```
arch-name/
|    |-- include/
|    |    |-- chip-name/
|    |    |-- other-chips/
|    |    |-- arch.h
|    |    |-- irq.h
|    |    |-- types.h
|    |    |-- limites.h
|    |    |-- syscall.h
|    |-- src/
|    |    |-- chip-name/
|    |    |-- Makefile
```

2.8.1.1 Resumo dos arquivos

1. <include/chip-name/>: este subdiretório possui arquivos de cabeçalho específicos do chip;
2. <include/arch.h>: possui todas as definições específicas da arquitetura que possam ser necessárias ao sistema;
3. <include/types.h>: fornece definições de tipos padrão específicos de arquitetura. Deve ser um typedef: `_int8_t, _uint8_t, _int16_t, _uint16_t, _int32_t, _uint32_t`;
4. <include/irq.h> : define as interrupções específicas da arquitetura;
5. <include/syscall.h>: define funções específicas da arquitetura para suportar interrupções ou *syscalls* que podem ser usadas em *user space* com funções NuttX em *kernel space*;
6. <src/chip-name/> : Este diretório contêm arquivo de origem específicos do chip.

2.8.2 <nuttx/configs>

O subdiretório <configs/> contém as informações de configuração para cada placa. Essas configurações junto as configurações específicas em <arch/> completam o *port* do NuttX.

```
board-name/
|    |-- include/
|    |    |-- board.h
|    |-- src/
|    |    |-- Makefile
|    |    |-- (board-specific source files)
```

2.8.2.1 Resumo dos arquivos

1. <include/>: Esse diretório contém arquivos de cabeçalho específicos da placa. Será vinculado como `include/arch/board.h` no momento da configuração e pode ser

incluído via `#include` no arquivo `<arch/board/header.h>`. Esse arquivo de cabeçalho só pode ser incluído por arquivos em `<arch/<arch-name>/include/>` e `<arch/<arch-name>/src/>`;

2. `<src/>`: este diretório contém *drivers* específicos da placa. Será vinculado como `<config>/arch/ <arch-name>/src/board>` no momento da configuração e será integrado ao sistema de compilação.

2.8.3 `<nuttx/drivers>`

Nuttx tem suporte à uma variedade de *device drivers*, incluindo *character device drivers*, *block device drivers* e *drivers* especializados. Este diretório e os diretórios subjacentes possuem os *drivers* independentes de arquitetura.

```
<drivers>/
|   |-- i2c/
|   |    |-- (arquivos de origem do driver de dispositivo I2C)
|   |-- ioexpander/
|   |    |-- (Expansor de I/O e arquivos de origem do driver relacionados
    ↪ ao GPIO)
|   |-- leds/
|   |    |-- (arquivos de origem do driver de dispositivo de LED)
|   |-- net/
|   |    |-- (Arquivos de origem do driver de rede)
|   |-- sensores/
|   |    |-- (Arquivos de origem do driver do sensor)
|   |-- spi/
|   |    |-- (drivers relacionados ao SPI e funções auxiliares)
|   |-- timers/
|   |    |-- (suporte ao driver de dispositivo baseado em timer)
|   |-- usbdev/
|   |    |-- (arquivos de origem do driver de dispositivo USB)
|   |-- usbhost/
|   |    |-- (arquivos de origem do driver host USB)
|   |-- usbmisc/
|   |    |-- (Diversos arquivos de origem do driver USB)
|   |-- usbmonitor/
|   |    |-- (arquivos de origem do monitor USB)
```

2.8.4 `<nuttx/include>`

Este diretório contém os arquivos de cabeçalho do NuttX que são incluídos da seguinte maneira:

```
1 include <stdio.h>
2 include <sys/types.h>
3 ...
```

2.8.5 `<nuttx/libs/libc>`

Este diretório contém uma coleção de funções padrão da `libc` com interfaces customizadas no NuttX. Comumente, é construído uma única biblioteca (`libc.a`). No entanto, se o NuttX for construído como um *kernel* compilado separadamente, o conteúdo deste diretório

será construído como duas bibliotecas: uma para uso em *user space* (`libuc.a`) e outra somente para uso em *kernel space* (`libkc.a`).

Essas bibliotecas de *user space* e *kernel space* juntamente as chamadas de sistema em `<nuttx/syscall>` são necessárias para suportar os dois domínios de proteção diferentes.

2.8.6 `<nuttx/syscall>`

Se o NuttX for construído com o *kernel* compilado separadamente (com `CONFIG_BUILD_PROTECTED=y` ou `CONFIG_BUILD_KERNEL=y`), então o conteúdo deste diretório é construído. Este diretório contém uma interface *syscall* que pode ser usada para comunicação entre os aplicativos em *user space* e o *kernel space*.

2.9 Sistemas de arquivos no Nuttx

O NuttX inclui sistemas de arquivos (VFS, do inglês *Virtual file system*) opcionais e escaláveis. Pode ser omitido, já que o NuttX não depende de qualquer sistema de arquivos. Entretanto, o NuttX pode implementar um VFS que pode ser utilizado para se comunicar com várias entidades utilizando um padrão como: `open()`, `close()`, `read()`, `write()`, etc.

Concordante com outros VFSs, o NuttX suporta pontos de montagem de arquivos, arquivos, diretórios, *device drivers*. O sistema de arquivos do NuttX suporta pseudo sistemas de arquivos, ou seja, sistemas de arquivos que aparecem como mídia normal, mas são apresentados sob controle programático. O Linux, por exemplo, possui `/proc` e os pseudo sistemas de arquivos em `/sys`. Não há mídia física subjacente ao pseudo sistema de arquivos (NUTT, 2019).

O sistema de arquivos raiz do NuttX é sempre um pseudo sistema de arquivos. No Linux o sistema de arquivos raiz deve ser algum dispositivo de bloco físico, então depois de montado o sistema de arquivos é possível montar outros sistemas de arquivos, como `/proc` ou `/sys`. Diferente do Linux, no NuttX o sistema de arquivos raiz é sempre um pseudo sistema de arquivos que não requer nenhum *driver* de bloco subjacente ou dispositivo físico. Então é possível montar o sistema de arquivos real no pseudo sistema de arquivos (NUTT, 2019).

2.10 Device Drivers

Uma das principais funções de um sistema operacional é controlar todos os periféricos de um computador, tratar erros, interceptar interrupções e fornecer uma interface entre o dispositivo e as aplicações. A função de um *device driver* é aceitar as requisições abstratas do *software* e cuidar para que a solicitação seja executada, permitindo a interação entre o *software* e os periféricos (MORAES; ALMEIDA; SERAPHIM, 2016).

Devices drivers são bibliotecas de *software* responsáveis pela inicialização e gerenciamento de dispositivos. Portanto, é o elo de ligação entre o *hardware* e o sistema operacional, *middleware* e as camadas de aplicação (DENARDIN; BARRIQUELO, 2019). Quando o *kernel*

reconhece que uma determinada ação é necessária a partir do dispositivo, ele chama a rotina de tratamento do *driver*, que passa o controle do processo de usuário para a rotina do *driver*. O controle é retomado para o processo de usuário quando a rotina do *driver* é concluída.

De acordo com Saraswat (2010), um *device driver* fornece os seguintes recursos:

a. Um conjunto de rotinas que se comunicam com um dispositivo de *hardware* e fornecem uma interface uniforme ao *kernel* do sistema operacional.
b. Um componente que pode ser integrado ou removido do sistema operacional dinamicamente.
c. Gerenciamento do fluxo de dados e controle entre programas de usuários e um dispositivo periférico.
d. Uma seção definida pelo *kernel* que permite a um dispositivo aparecer como um *dev* para o restante do sistema.

Um *device driver* simplifica a tarefa da aplicação, atuando como um tradutor entre o dispositivo e as aplicações. O código de alto nível das aplicações pode ser escrito independente do dispositivo que será utilizado (MORAES; ALMEIDA; SERAPHIM, 2016).

2.11 Device Drivers no Linux

Uma das principais funcionalidades do *kernel* é prover um mecanismo de acesso aos dispositivos para as bibliotecas e aplicações. Segundo Rubini, Corbet e Hartman (2005), no Linux, o acesso ao *hardware* é exportado para as aplicações por intermédio de três classes de dispositivos:

a. *Character device*: pode ser acessado como um arquivo, e um *driver* de caractere é responsável pela execução deste *device*. Esse *driver* comumente é acessado por meio de função como `open()`, `read()`, `write()`, `close()`, que são *system calls* POSIX. Um character device pode ser acessado pelo ponto de interconexão (nós) de um arquivo de sistema, como `dev/tty1` e `dev/lp1`.
b. *Block device*: pode armazenar um sistema de arquivos, da mesma forma que um disco. Os *character device* e *block device* diferem apenas quanto à forma pela qual os dados são gerenciados internamente pelo kernel. É acessado por meio de um nó do sistema de arquivos.
c. *Network device*: é representado por uma interface de rede física ou *software*, que é responsável por enviar e receber pacotes de dados por meio de uma camada de rede do *kernel*. Não possui um arquivo em `/dev` e sua comunicação obtida através de uma API específica.

O Linux pode carregar cada tipo de dispositivo na forma de um módulo, permitindo que os usuários experimentem um *hardware* novo em tempo de execução (RUBINI; CORBET; HARTMAN, 2005). O Linux é um *kernel* monolítico, mas internamente é bem modular. Cada funcionalidade é abstraída em um módulo, por isso, permite um sistema de configuração para adicionar ou remover determinada funcionalidade.

Além disso, é possível compilar separadamente uma funcionalidade, e carregar o arquivo gerado em tempo de execução. Um *device driver* pode ser compilado de forma integrada ao *kernel* (*built-in*) ou como um módulo do *kernel* em tempo de execução (PRADO, 2016). Essa característica de carregar o módulo dinamicamente ajuda a diminuir o tempo de desenvolvimento, pois, é possível testar as seguintes versões do *driver* sem passar pelo extenso ciclo de desligar ou reinicializar (RUBINI; CORBET; HARTMAN, 2005).

2.11.1 Construção de módulos

É importante enfatizar as diferenças entre um módulo e um aplicativo. Enquanto um aplicativo executa determinada tarefa do começo ao fim, um módulo faz seu próprio registro para atender requisições futuras e sua tarefa termina de imediato, ou seja, o ponto de entrada do módulo é para preparar para uma chamada posterior das funções do módulo. O segundo ponto de entrada de um módulo é chamado logo antes do módulo ser descarregado (RUBINI; CORBET; HARTMAN, 2005).

De acordo com Prado (2016), as principais características de módulo para o desenvolvimento são:

i. Módulos tornam o desenvolvimento em *kernel space* mais fácil, já que o *target* não precisa ser reiniciado cada vez que um módulo é carregado;
ii. Ajuda a manter a imagem do *kernel* bem pequena;
iii. Só ocupa memória enquanto estiver carregado;
iv. O tempo de *boot* do *kernel* fica menor.
v. Deve-se ficar atento, pois, os módulos rodam em *kernel space*. Uma vez carregados, o módulo possui total controle do sistema, por isso só podem ser carregados como *root*.

O código a seguir exemplifica a construção de um módulo genérico.

Listagem 2.4 – Exemplo de utilização da função write().

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3
4  /* module initialization */
5  static int __init mymodule_init(void)
6  {
7   printk("My module initialized.\n");
8   return 0;
9  }
10
11 /* module exit */
12 static void __exit mymodule_exit(void)
13 {
14  printk("Exiting my module.\n");
15 }
16
17 module_init(mymodule_init);
18 module_exit(mymodule_exit);
19
20 MODULE_LICENSE("GPL");
```

A macro `module_init()` declara a função de inicialização que será chamada ao carregar o módulo para a memória. É responsável por inicializar o módulo e é removida da memória assim que executada. A macro retorna o valor zero para execução concluída e um número negativo em caso de erro. A macro `module_exit()` declara a função de saída, que será chamada assim que o módulo for descarregado da memória (RUBINI; CORBET; HARTMAN, 2005).

Além das funções anteriores, é possível declarar informações dos módulos usando as seguintes macros:

a. `MODULE_LICENSE()`: declara a licença do módulo;
b. `MODULE_DESCRIPTION()`: mensagem descritiva do módulo;
c. `MODULE_AUTHOR()`: autor do módulo;
d. `MODULE_VERSION()`: versão do módulo;
e. `MODULE_PARM_DESC()`: descrição dos parâmetros recebidos pelo módulo.

2.11.2 *Character Devices*

Com exceção dos *drivers* para dispositivos de armazenamento, a maioria dos *device drivers* são implementados como um *character device*. Um *character device* é acessado por meio dos nós do sistema de arquivos, geralmente, localizados no diretório `/dev` (PRADO, 2016). De acordo com (RUBINI; CORBET; HARTMAN, 2005), cada arquivo de dispositivo possui três informações básica, que identificam o dispositivo ao qual o arquivo pertence:

a. Tipo: character device ou block device;
b. Major number: categoria do dispositivo;
c. Minor number: identificador do dispositivo.

Os arquivos de dispositivos são arquivos especiais e que são identificados pelo caractere "c"na primeira coluna de saída do comando *ls -l /dev*. Os dispositivos de bloco também aparecem na saída do comando em `/dev`, mas são identificados pelo caractere "b". A seguir, um exemplo de saída do comando *ls -l /dev*.

```
crw-rw-rw-  1 root        tty        5,   0 jun 27 13:51 tty
crw--w----  1 root        tty        4,   0 jun 27 13:51 tty0
crw--w----  1 Debian-gdm  tty        4,   1 jun 27 13:51 tty1
crw--w----  1 root        tty        4,  10 jun 27 13:51 tty10
crw-rw----  1 root        tty        7, 133 jun 27 13:51 vcsa5
crw-rw----  1 root        tty        7, 134 jun 27 13:51 vcsa6
crw-------  1 root        root     249,   0 jun 28 13:51 hidraw0
crw-------  1 root        root     249,   1 jun 28 13:51 hidraw1
```

É possível ver dois números separados por uma vírgula nas entradas do arquivo de dispositivo antes da data da última modificação. Esses são os *major number* e *minor number* para o dispositivo específico. O *major number* identifica o *driver* associado ao dispositivo. Por exemplo, `/dev/vcsa4` e `/dev/vcsa5` são gerenciados pelo o *driver* de *major number* 7, enquanto `/dev/tty0`, `/dev/tty1` e `/dev/tty10` são gerenciados pelo *driver* 4. O *kernel* utiliza o *major number* para associar o *driver* ao dispositivo adequado.

O *minor number* só é utilizado pelo *device driver*. É comum que um *driver* possua vários dispositivos, o *minor number* garante uma forma de fazer a distinção entre os dispositivos do *driver* (RUBINI; CORBET; HARTMAN, 2005).

2.11.3 *Major Number* e *Minor Number*

O primeiro passo para desenvolver um *driver* de *character device* ou *block device* é registrar um *device number* para o *device driver*. O *device number* é composto pelos números *major number* e *minor number*. O *kernel* armazena informações de *device number* no tipo de dados `dev_t` (PRADO, 2016).

O tipo de dados `dev_t` está definido em `<Linux/types.h>` e atualmente é representado com 32 *bits*, dos quais os doze bits mais significativos representam o *major number* e os vinte bits menos significativos representam o *minor number* (PRADO, 2016). Algumas macros são disponibilizadas para gerenciar as variáveis do tipo `dev_t`:

Listagem 2.5 – Macros definidas para dev_t

```
1 /* creating a device number */
2 dev_t mydev = MKDEV(major, minor);
3
4 /* extracting major number */
5 MAJOR(mydev);
6
7 /* extracting minor number */
8 MINOR(mydev);
```

Adicionar um novo *driver* no sistema significa atribuir um *major number* a ele. A atribuição de um *major number* deve ser feita na inicialização do módulo, e esta atribuição pode ser feita estaticamente ou dinamicamente (RUBINI; CORBET; HARTMAN, 2005). É possível registrar estaticamente por meio da função `register_chrdev_region()`, como apresentado no *algoritmo* a seguir.

Entretanto, pode acontecer o caso em que o programador não sabe todos os dispositivos que estão presente no sistema, pode haver conflito ao tentar alocar estaticamente um *major number* que já está sendo utilizado por outro dispositivo. Neste caso, é aconselhável registrar o *major number* e *minor number* dinamicamente com a função `alloc_chrdev_region()` (RUBINI; CORBET; HARTMAN, 2005).

Listagem 2.6 – register_chrdev_region() e alloc_chrdev_region().

```
1 #include <linux/fs.h>
2
3 /* allocate device number statically */
4 int register_chrdev_region(dev_t from,
5 unsigned count,
6 const char *name);
7
8 /* allocate device number dynamically */
9 int alloc_chrdev_region(dev_t *dev,
10 unsigned baseminor,
11 unsigned count,
12 const char *name);
```

```
13
14 /* example with register_chrdev_region() */
15 static dev_t mydriver_dev = MKDEV(202, 128);
16
17 if (register_chrdev_region(mydriver_dev, 4,  "mydriver")) {
18  pr_err("Failed␣to␣allocate␣device␣number\n");
19  [...]
20 }
21 /* example with alloc_chrdev_region()*/
22 static dev_t mydriver_dev;
23 if (alloc_chrdev_region(&mydriver_dev, 0, 4, "mydriver")) {
24  pr_err("Failed␣to␣allocate␣device␣number\n");
25  [...]
26 }
```

A desvantagem da atribuição dinâmica é que não é possível criar os nós de dispositivos antecipadamente, pois não existe garantias de que o major number atribuído ao dispositivo seja sempre o mesmo. No entanto, o problema não se resume a isso, já que todos os dispositivos registrados estão visíveis em `/proc/devices`, logo, é possível fazer um *script* que lê o *major number* atribuído ao dispositivo (RUBINI; CORBET; HARTMAN, 2005). O *major number* deve ser liberado quando o módulo é descarregado do sistema. Para liberar, utilizamos a função a seguir:

```
1 int unregister_chrdev(unsigned int major, constar char *name);
```

Esta função deve ser executa com a função `module_exit()`. O primeiro argumento represeta o *major number* que está sendo liberado e o segundo argumento representa o nome do dispositivo associado. O *kernel* compara os argumentos com o nome e *major number* registrados do dispositivo: se forem diferentes, a função retorna -EINVAL (RUBINI; CORBET; HARTMAN, 2005).

2.11.4 Operações de arquivos

Um dispositivo é identificado internamente por uma estrutura do tipo *file* e o *kernel* usa a estrutura `file_operations` para acessar as funções do *driver*. A estrutura `file_operations` (`fops`) é dada por uma tabela de ponteiros de função e contém todas as operações implementadas pelo *driver*.

Listagem 2.7 – Estrutura file_operations.

```
1 #include <linux/fs.h>
2 static struct file_operations mydriver_fops{
3  .owner    = THIS_MODULE,
4  .open     = mydriver_open,
5  .release  = mydriver_release,
6  .read     = mydriver_read,
7  .write    = mydriver_write,
8  .ioctl   = mydriver_ioctl,
9 };
```

Cada entrada na tabela indica a função definida pelo *driver* para lidar com a operação solicitada. Como ela é genérica para todos os arquivos gerenciados pelo *kernel*, nem todas as operações definidas nesta estrutura são necessárias para um *device driver* de caractere.

A estrutura `file` representa um arquivo aberto. É criada pelo *kernel* em `open` e passada para qualquer função que opere no arquivo. Assim que o arquivo é fechado, o *kernel* libera a estrutura `file` (RUBINI; CORBET; HARTMAN, 2005).

2.11.4.1 Função *open()*

O método `open()` é fornecido para um *driver* para realizar qualquer inicialização na preparação das operações posteriores. Segundo (RUBINI; CORBET; HARTMAN, 2005), as principais características do método `open()` são:

a. é chamado quando o arquivo de dispositivo é aberto;
b. uma estrutura do tipo `file` é criada toda vez que um arquivo é aberto, e armazena informações como a posições corrente do arquivo, modo de abertura, etc;
c. a estrutura `inode` é a representação única do arquivo no sistema;
d. checa erros específicos do dispositivo;
e. inicializa o dispositivo, caso ele seja aberto pela primeira vez;

2.11.4.2 Função *release()*

O método `release()` é simples, é exatamente o oposto de `open()`, e deve ser chamado quando o arquivo de dispositivo é fechado.

2.11.4.3 Troca de dados com *User Space*

Não é possível acessar *buffers* presentes no *user space* por meio de um código presente no *kernel space*, portanto, para manter o código seguro e portável, o *driver* precisa usar funções específicas para trocas dados entre *user space* e *kernel space* (PRADO, 2016).

Para copiar o conteúdo de um buffer presente em *user space*, a função `read()` correspondente do *driver* deve utilizar o método `copy_to_user()`. A função `write()` correspondente do *driver* faz o inverso utilizando o método `copy_from_user()`. Abaixo será identificado a utilização dos métodos com as funções `read()` e `write()` respectivamente (RUBINI; CORBET; HARTMAN, 2005).

2.11.4.4 Função *read()*

A função `read()` é chamada quando é realizada uma operação de leitura no arquivo de dispositivo. O *driver* deverá:

i. Ler até `sz` *bytes* do dispositivo e salvar no buffer `buf`;
ii. Atualizar a posição atual do arquivo na variável `off` (opcional);
iii. Retornar a quantidade de bytes lidos.

Listagem 2.8 – Exemplo de função read() utilizando o método copy_to_user

```
1 static ssize_t mydriver_read(struct file *file, char __user *buf, size_t
   ↪ count, loff_t *ppos)
```

```
2 {
3  int transfer_size, qtd;
4  char bufrx[256];
5
6  transfer_size = min_t(int, sizeof(bufrx), count);
7
8  qtd = read_device(bufrx, transfer_size);
9
10  if (copy_to_user(buf, bufrx, qtd)) {
11   return -EFAULT;
12  } else {
13   return qtd;
14  }
15 }
```

O valor de retorno da função `read()` é identificado de diferentes maneiras. Se o valor for igual ao argumento `sz`, significa que o número requisitado de *bytes* foi transferido com sucesso. Se o retorno for positivo, entretanto, menor do que `sz`, significa os dados foram transferidos parcialmente. Isto pode ocorrer por uma série de motivos. Se o retorno for igual a zero, é interpretado como fim de arquivo. Se o retorno for negativo, significa que há um erro. O tipo de erro é especificado de acordo com o valor por meio de `<linux/errno.h>` (RUBINI; CORBET; HARTMAN, 2005).

2.11.4.5 Função *write()*

A função `write()` é chamada quando é realizada uma operação de escrita no arquivo de dispositivo. Suas principais características são:

i. Ler `sz` bytes do buffer `buf` e escrever no dispositivo;
ii. Atualizar a posição do arquivo na variável `off` (opcional);
iii. Retornar a quantidade de bytes escritos no dispositivo.

Listagem 2.9 – Exemplo de função write() utilizando o método copy_from_user

```
1 static ssize_t mydriver_write(struct file *file, const char __user *buf,
    ↪              size_t count, loff_t *ppos)
2 {
3   int transfer_size;
4   char buftx[256];
5
6   transfer_size = min_t(int, sizeof(buftx), count);
7
8   if (copy_from_user(buftx, buf, transfer_size)) {
9    return -EFAULT;
10   } else {
11    write_device(buftx, transfer_size);
12    return transfer_size;
13   }
14 }
```

O método `write()` interpreta o retorno da função de maneira semelhante ao método `read()`. Se o valor for igual a `sz`, significa que o número requisitados de *bytes* foi transferido. Se o retorno for positivo, porém, menor do que `sz`, significa que os dados foram transferidos parcialmente. Se o retorno for igual a zero, nada foi gravado. Este resultado não é um erro e não possui motivo para retornar um código de erro. Logo, a biblioteca padrão tenta chamar o

método `write()` novamente. Um retorno negativo representa a ocorrência de um erro (RUBINI; CORBET; HARTMAN, 2005).

Entretanto, dependendo da quantidade de dados e do fluxo de comunicação, o uso das funções de troca de dados entre *user space* e *kernel space* pode impactar na performance do sistema. Para estes casos, existem soluções alternativas, em que não é necessário realizar a cópia dos *buffers*. Pode-se implementar o método `mmap()`, que permite que um código rodando em *user space* tenha acesso direto a memória.

2.11.5 Operações avançadas de um *Character Device*

Esta operação esta associada a *system call* `ioctl()`, e permite estender as capacidades do *driver* além da API de `read()` e `write()`. A forma mais comum de executar operações de controle por meio de um *device driver* é implementada pelo método `ioctl()`. A *system call* `ioctl()` oferece um ponto de entrada específico do *device* para que o *driver* emita comandos. Geralmente, operações de controle não podem ser visualizadas na abstração de `read()/write()`. Por exemplo, os dados que são gravados por meio de uma porta serial são transmitidos pela porta e não é possível alterar a taxa de transmissão dos dados ao realizar a operação de escrita no dispositivo. Esta é a função do método `ioctl()`: controlar o canal entrada e saída (RUBINI; CORBET; HARTMAN, 2005).

O comando a ser executado é transmitido por meio do argumento `cmd` e pode-se ser necessário utilizar o argumento `arg` para passar informações adicionais necessárias para execução do comando. O algoritmo seguinte, exemplifica a utilização da função `ioctl()`.

Listagem 2.10 – Exemplo de utilização da função ioctl().

```
1  static long phantom_ioctl(struct file *file, unsigned int cmd, unsigned
       ↪ long arg)
2  {
3   struct phm_reg r;
4   void __user *argp = (void __user *)arg;
5
6   switch(cmd) {
7   case PHN_SET_REG:
8    if (copy_from_user(&r, argp, sizeof(r)))
9     return -EFAULT;
10  /* do something */
11  break;
12
13  case PHN_GET_REG:
14   if(copy_to_user(argp, &r, sizeof(r)))
15    return -EFAULT;
16  /* do something */
17  break;
18
19  default:
20   return -ENOTTY;
21  }
22  return 0;
23 }
```

Listagem 2.11 – Exemplo de utilização da função ioctl() em user space.

```
int main(void)
{
 int fd, ret;
 struct phm_reg reg;

 fd = open("/dev/phantom");
 assert(fd > 0);

 reg.field1 = 30;
 reg.field2 = 20;

 ret = ioctl(fd, PHN_SET_REG, &reg);
 assert(ret == 0);

 return 0;
}
```

A maioria das funções do *kernel* retorna `-EINVAL` quando o número do comando não corresponde a uma operação válida. O POSIX, no entanto, afirma que, se um comando de `ioctl()` que não corresponde à uma operação válida, então, o método deve retornar `-ENOTTY` (RUBINI; CORBET; HARTMAN, 2005).

2.11.6 Registrando um *Character Device*

Um *character device* é representado pelo *kernel* por meio de uma estrutura denominada `cdev`. Para registrar o *device*, é necessário declarar uma estrutura global do tipo `cdev` e inicializá-la utilizando o método `cdev_init()`. Em seguida, basta adicionar o *device* ao sistema com a função `cdev_add()` (PRADO, 2016).

Listagem 2.12 – Registro de um character device.

```
#include <linux/cdev.h>

static struct cdev mydriver_cdev;

static int __init mydriver_init(void)
{
 [...]
 cdev_init(&mydriver_cdev, &mydriver_fops);

 if (cdev_add(&mydriver_cdev, mydriver_dev, 1)) {
  pr_err("Char driver registration failed\n");
  [...]
 }
 [...]
}
```

Posteriormente à chamada de `cdev_add()`, o *kernel* associará o *major/minor number* registrado com as operações de arquivo definidas e o dispositivo está pronto para uso.

Com o registro do *device driver* realizado pelo *kernel*, o device ficará disponibilizado no diretório */dev* e poderá ser utilizado por qualquer aplicação que utilize os conceitos de abstração de devices drivers por meio de operações em arquivos.

As operações registradas sobre um *device driver* no linux, podem, posteriormente serem comparadas à implementação do driver deste trabalho.

3 DESENVOLVIMENTO

Este capítulo apresenta o desenvolvimento do trabalho proposto, iniciando com a preparação do ambiente de desenvolvimento. Em seguida, a metodologia para adequação de camadas de abstração de hardware de microcontroladores para *device drivers* proposta é segmentada em duas etapas. A primeira apresenta uma metodologia para criar um *template* de um *device driver* do tipo caracter *device* no Nuttx. Em sequência, como implementar um *driver* da classe USB CDC/ACM sobre o *template* desenvolvido. E por fim, a implementação de uma aplicação no Nuttx que utiliza as funções do *device driver* implementado.

3.1 Preparação do ambiente de desenvolvimento

O primeiro passo para começar a o desenvolvimento no `Nuttx` é realizar a instalação de um conjunto de ferramentas necessárias para arquitetura com a qual você trabalhará e, finalmente, realizar o download do código fonte.

O conjunto de ferramentas varia de acordo com o sistema operacional que está utilizando. Nesse caso, será demonstrado o método de instalação para distribuições Linux baseados no Debian. Para desenvolvimento deste projeto as seguintes ferramentas foram utilizadas:

a. NuttX 10.0.1 LTS;
b. *Kit* de desenvolvimentoTiva C Series TM4C1294 Connected;
c. API TivaWare;
d. *Toolchain* para desenvolvimento de aplicação no NuttX.
e. Ubuntu 20.04.2 LTS;
f. lm4flash;
g. Visual Studio Code 1.54.3;
h. Cabo auxiliar micro-USB/A;
i. Putty 0.73.

3.1.1 Kit de Desenvolvimento Tiva C

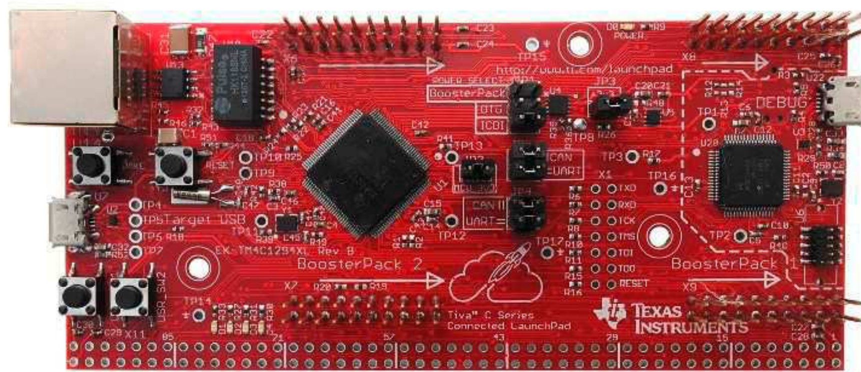
O kit de desenvolvimento TM4C1294 Connected LaunchPad é uma plataforma de desenvolvimento de baixo custo para microcontroladores baseados em ARM Cortex-M4. O design do Connected Launchpad destaca TM4C1294NCPDT MCU com seu 10/100 Ethernet MAC e PHY no *chip*, USB 2.0, módulo de hibernação, modulação por largura de pulso de controle de movimento e diversos modos de conectividade serial simultânea.

Os recursos disponíveis no TM4C1294 Connected LaunchPad são:

a. CPU ARM Cortex-M4 de 120 MHz e 32 bits;
b. Flash de 1 MB, SRAM de 256 KB, EEPROM de 6 KB;

c. Ethernet 10/100 integrado MAC + PHY;
d. ADCs 2MSPS duplos de 12 bits, PWMs de controle de movimento;
e. Host, dispositivo e OTG de alta velocidade USB 2.0;
f. Locais de conexão BoosterPack XL empilhaveis e duplos;
g. Dois módulos de rede de área do controlador (CAN);
h. Interface de depuração on-board, in-circuit (ICDI);
i. uporte para várias cadeias de ferramentas de desenvolvimento: CCS , Keil, IAR e GCC;
j. Dezenas de exemplos de aplicativos fornecidos com o TivaWare SDK;

Figura 3 – Arquitetura Tiva



Fonte: **Texas Instruments.**

3.1.2 Pacote lm4flash

O pacote `lm4flash` contém uma ferramenta que grava no Tiva C o arquivo binário da aplicação gerado pelo objcopy do GCC. Possuí procedimento de instalação simples e pode ser instalado via terminal utilizando o seguinte comando.

```
$ sudo apt-get install lm4flash
```

3.1.3 Putty

A ferramenta Putty é um software de emulação de terminal de código livre. Suporta diversos protocolos, inclusive, comunicação serial, da qual foi utilizada para conectar ao NuttX instalado no TIVA C e, também, para realizar os testes com o *driver* desenvolvido na porta usb do dispositivo.

Para realizar a instalação do Putty, execute o seguinte comando:

```
$ sudo apt-get install putty
```

3.2 Preparação do Ambiente

Como explicado anteriormente, a preparação do ambiente é diretamente ligada à arquitetura de desenvolvimento, os passos seguintes exemplificam o procedimento de instalação das ferramentas listadas anteriormente.

Para baixar a toolchain de desenvolvimento, execute os seguintes comandos:

```
$ sudo apt install \
bison flex gettext texinfo libncurses5-dev libncursesw5-dev \
gperf automake libtool pkg-config build-essential gperf genromfs \
libgmp-dev libmpc-dev libmpfr-dev libisl-dev binutils-dev libelf-dev \
libexpat-dev gcc-multilib g++-multilib picocom u-boot-tools util-linux
```

O Nuttx utiliza o Kconfig, que é exposto por meio de uma série de interfaces interativas baseadas em menu. Para instalar, execute o comando a seguir:

```
$ sudo apt install kconfig-frontends
```

Alguns sistemas operacionais, como o Linux, distribuem conjuntos de ferramentas para várias arquiteturas. Essa é geralmente uma escolha fácil, mas deve-se estar ciente de que em alguns casos a versão oferecida pelo sistema operacional utilizado pode ter problemas e pode ser melhor usar uma versão amplamente utilizada de outra fonte.

O exemplo a seguir mostra como instalar um conjunto de ferramentas para a arquitetura ARM:

```
$ apt install gcc-arm-none-eabi binutils-arm-none-eabi
```

E por último, basta realizar o download do Nuttx diretamente do repositório oficial no Github. O Nuttx é desenvolvido ativamente no GitHub e existem dois repositórios principais, `nuttx` e `apps`. Para realizar o *download* de ambos, utilize os seguintes comandos.

```
$ mkdir nuttx
$ cd nuttx
$ git clone https://github.com/apache/incubator-nuttx.git nuttx
$ git clone https://github.com/apache/incubator-nuttx-apps apps
```

Neste momento, os pré-requisitos do nuttx foram instalados e o código fonte foi clonado, basta compilar o código fonte em um binário executável que pode ser transmitido para placa de desenvolvimento.

3.3 Configurando o NuttX para o Tiva C

A primeira etapa é inicializar a configuração do Nuttx para uma plataforma específica, com base em uma pré-configuração existente. Para listar todas as configurações com suporte, execute:

```
$ cd nuttx
$ ./tools/configure.sh -L | less
```

É possível ver que todas as placas suportam a configuração padrão do NuttShell, que é um bom ponto de partida. O NuttShell é um *shell system* que oferece suporte à um extenso

conjunto de comandos, scripts e, também, a capacidade de executar seus próprios aplicativos como embutidos, partindo do mesmo binário do Nuttx.

Para escolher uma configuração, basta indicar a plataforma host, no seguinte formato:

```
$ ./tools/configure.sh <board name>:<board configuration>
```

Nesse caso, para utilizar o tm4c1294-launchpad como host, utilize:

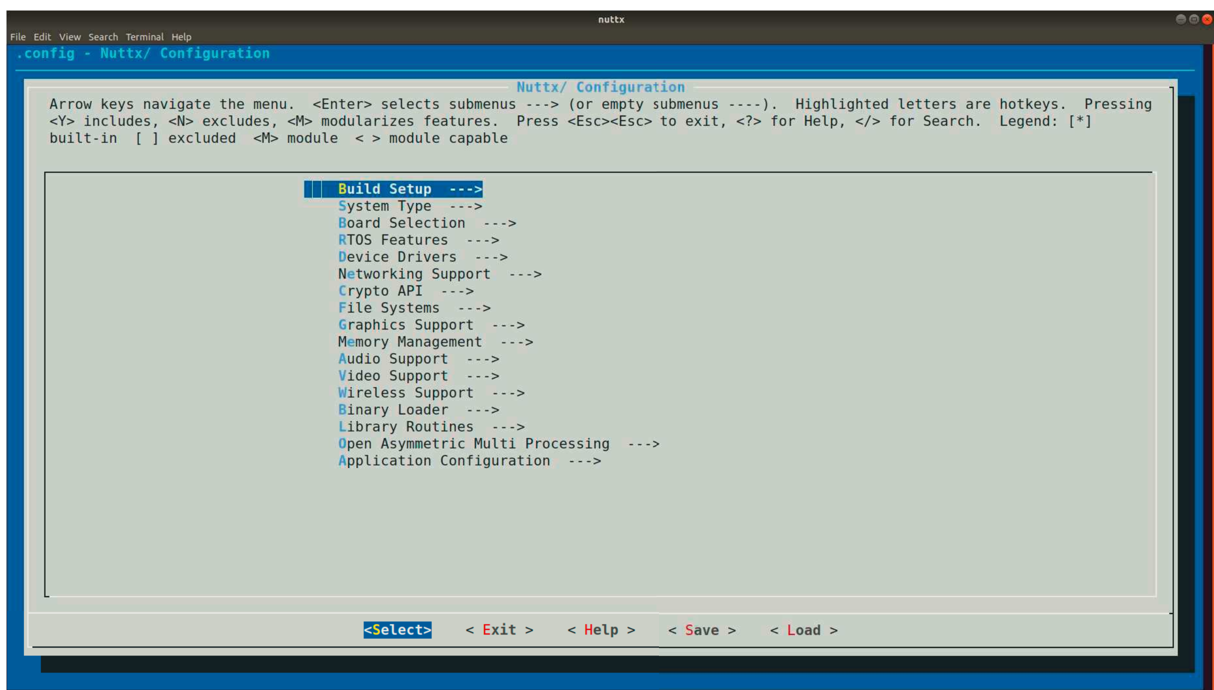
```
$ cd nuttx
$ ./tools/configure.sh -l tm4c1294-launchpad:nsh
```

Feito isso, o conjunto de configurações iniciais está pronto e posteriormente devem ser personalizadas para que as aplicações e *drivers* possam ser utilizados. Para personalizar a configuração do Nuttx para a placa em uso, deve-se utilizar a ferramenta menuconfig que disponibiliza de maneira visual o gerenciamento de configurações do Nuttx. O menuconfig config pode ser acessado utilizando o comando make no terminal, como indica a listagem a seguir.

```
$ cd nuttx
$ make menuconfig
```

O comando irá exibir um menu de configuração simples e iterativo pelas setas do teclado. O menuconfig deverá ser exibido de acordo com a Figura 4.

Figura 4 – Overview menuconfig do NuttX



Fonte: **Própria.**

Finalmente, pode-se realizar o processo de compilação do Nuttx. Para fazer isso, deve-se executar o comando *make* no diretório raiz do Nuttx. A listagem à seguir exemplifica a utilização do comando para realizar o procedimento de compilação.

```
$ cd nuttx/
$ make -j
```

Se nenhum erro for encontrado, a compilação será concluída e a saída será um arquivo binário chamado **nuttx.bin** que pode ser enviado para a placa.

Para gravar o binário na placa de destino, se utiliza a ferramenta lm4flash. Para utilizar o lm4flash, deve-se comando no diretório raiz do nuttx. A listagem a seguir, exemplifica esse procedimento.

```
$ cd nuttx/
$ lm4flash nuttx.bin
```

Após gravar o binário no dispositivo final, faz-se necessário a utilização de um emulador de terminal para conectar no Nuttx que já está rodando no Tiva C. Neste caso, optou-se por utilizar a ferramenta `Putty`. Normalmente, basta se conectar no dev `/dev/ttyACM0`. Após realizar conexão, a seguinte mensagem será exibida no console:

Figura 5 – Tela inicial do Nuttx.

Fonte: **Própria.**

Nesse ponto, o Nuttx pode executar direto na plataforma de destino, no entanto, sem nenhuma configuração, aplicação ou *driver* adicionado.

3.4 Implementação de um Device Driver genérico no Nuttx

Desenvolver um *Device Driver* para o Nuttx não é uma tarefa simples, visto que não existem tutoriais oficiais para desenvolvimento de *drivers*, apenas documentos que explicam o funcionamento das *API's* disponíveis para utilização no âmbito dos *drivers*. Sabendo disso,

optou-se por desenvolver um *template* que tem a finalidade de facilitar o entendimento e posicionamento de cada componente do *driver*. Posteriormente, o mesmo *template* é utilizado para a implementação do *driver* USB da classe CDC/ACM.

O primeiro passo do desenvolvimento é entender a função de cada componente do *driver*, por isso faz-se necessário o entendimento das camadas presentes entre o processo de registro e utilização do *driver*. O Nuttx utiliza o conceito de *upper half* e *lower half*.

1. Camada *Upper Half*: realiza o registro do *device driver* utilizando uma chamada do tipo *register_driver()*;
2. Camada *Lower Half*: realiza a comunicação com a camada *upper half* por meio de *callbacks*.

O suporte de *device drivers* depende de pseudo sistema de arquivos que é ativado por padrão.

Na base da arquitetura do *device driver*, possuímos o microcontrolador e seus períficos que juntos constituem o hardware. Em sequência, temos a camada denominada de *lower half* em que ficam situados os códigos específicos de cada microcontrolador, por exemplo: os componentes do *drivers* que diz respeito aos periféricos de um microcontrolador, como a configuração e uso de um registradores do periférico. No processo de inicilização, o Nuttx irá inicializar os *drivers* específicos de cada periférico e em sequência irá associá-los aos *drivers* da camada *upper half*. Nessa camada, ficam registrados todo e qualquer código que interaja diretamente com os períficos do microcontrolador, etapa conhecida como implementação de baixo nível.

Acima da camada *lower half*, existe a camada *upper half* em que ficam localizados os *drivers* genéricos, tais como: Serial, SD/MMC, Network, CAN. O *driver* genérico fica responsável de criar um device de unidade no sistema de arquivos disponível. Na camada *upper half* são implementadas as funções principais de acordo com a norma POSIX, sendo elas: *open*, *read*, *write*, *close*, *ioctl*. Nessas funções, devemos escrever o código que será utilizados via *system calls* pelos aplicativos na camada de aplicação. Quando um *device driver* é carregado pelo sistema, seu acesso é disponibilizado no caminho `/dev`. As aplicações realizam operações comuns de arquivos diretamente no *device driver*, que é abstraído como um arquivo.

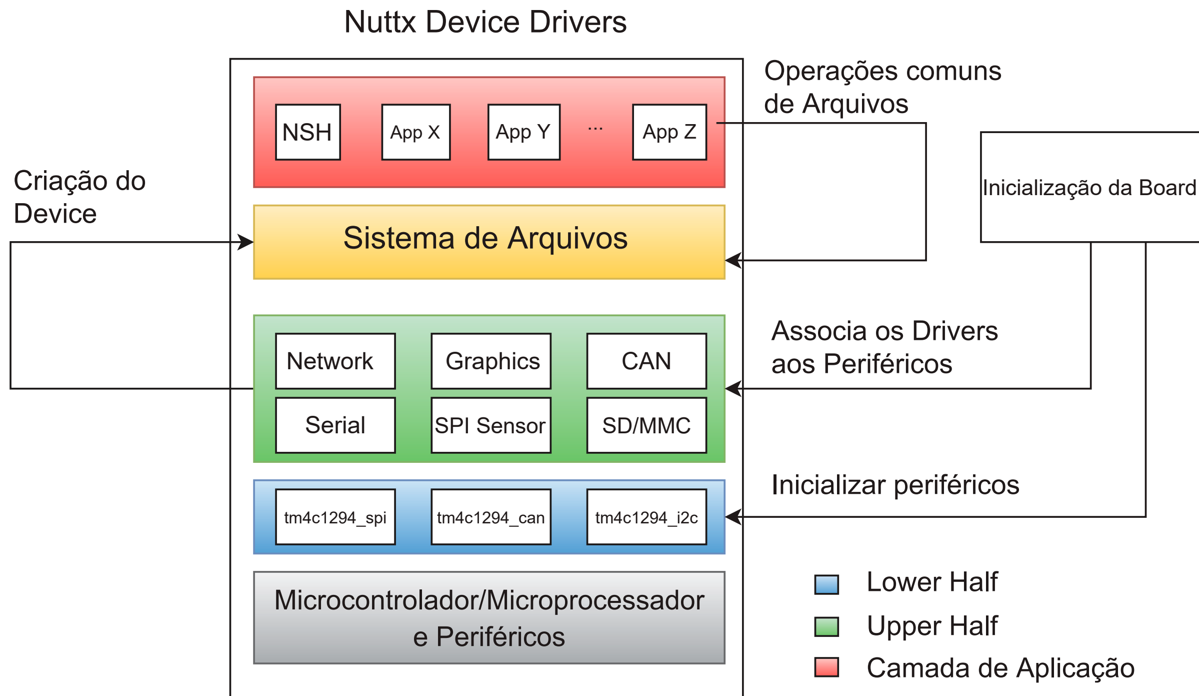
A Figura 6 demonstra um processo simplificado da arquitetura de um *device driver* no Nuttx e segmenta as camadas *lower half*, *upper half* e a camada de aplicação.

A primeira etapa da construção do *template* é definir quais serão os protótipos e estruturas necessárias para o desenvolvimento das funções POSIX que posteriormente serão chamadas via *system calls*. Inicialmente deve-se incluir os cabeçalhos mínimos para uma configuração inicial do *driver*, de acordo com a listagem 3.1.

Listagem 3.1 – Importação dos cabeçalhos para o template do device driver

```
1 #include <nuttx/config.h>
2 #include <nuttx/board.h>
3 #include "tiva_gpio.h"
4 #include "tm4c1294-launchpad.h"
```

Figura 6 – Arquitetura de um Device Driver no NuttX



Fonte: **Própria.**

Optou-se por realizar a implementação das funções principais e mais utilizadas sobre o padrão POSIX, sendo elas, descritas na listagem 3.2.

Listagem 3.2 – Protótipo das funções implementadas do driver

```
/*
* Protótipos e Estruturas
*/

typedef FAR struct file file_t;

static int cdc_open(file_t *filep);
static int cdc_close(file_t *filep);
static ssize_t cdc_read(file_t *filep, void *buf, size_t buflen);
static ssize_t cdc_write(file_t *filep, void *buf, size_t buflen);
static int cdc_ioctl(file_t *filep, int request, unsigned long arg);
```

No Nuttx assim como no Linux, todo *character device* deve possuir a implementação de uma `struct file_operations`, que contêm os ponteiros de função apontando para as definições das funções implementadas do *driver*. Cada campo da estrutura corresponde a uma função definida pelo *driver* para tratar uma operação solicitada da aplicação. Qualquer membro da estrutura que não for atribuído, será declarado como `NULL` pelo compilador `GCC`. Comumente, um ponteiro para uma estrutura file_operations é denominado como `fops`. A listagem 3.3, exemplifica a implementação da estrutura para o *template* do *driver*.

Listagem 3.3 – Atribuição dos ponteiros de função na estrutura file_operations

```
1 static const struct file_operations fops = {
2  driver_open,   /* open  */
3  driver_close,  /* close  */
4  driver_read,   /* read  */
5  driver_write,  /* write  */
6  0,        /* seek  */
7  driver_ioctl,  /* ioctl  */
8 };
```

Em sequência, com base nas funções declaradas na estrutura *file_operations*, deve-se desenvolver a funcionalidade de cada uma das funções respeitando o tipo, parâmetros de entrada e retorno com base no padrão POSIX. A listagem 3.4, deixa de exemplo um *template* que pode ser utilizado para qualquer *driver*, bastando realizar a implementação das funcionalidades que cada método irá disponibilizar para a camada de aplicação.

Listagem 3.4 – Implementação do método open

```
1 /***************************************************************
2 * Test: POSIX Functions
3 ***************************************************************/
4 static int driver_open(file_t *filep)
5 {
6   return OK;
7 }
8 
9 static int driver_close(file_t *filep)
10 {
11   return OK;
12 }
13 
14 static ssize_t driver_read(file_t *filep, void *buf, size_t buflen)
15 {
16   // Read Device
17 }
18 
19 static ssize_t driver_write(file_t *filep, void *buf, size_t buflen)
20 {
21   // Write Device
22 }
23 
24 static int driver_ioctl(file_t *filep, int request, unsigned long arg){
25   switch(request)
26   {
27    case 1:
28    {
29     ...
30    }
31    break;
32    case 2:
33    {
34     ...
35    }
36    break;
37    case 3:
38    {
39     ...
40    }
41   }
```

```
42   return 0;
43 }
```

Para registrar um device driver no Nuttx, devemos utilizar a função *register_driver*, que fica responsável por subir o *driver* no processo de inicialização do Nuttx. A listagem 3.5 demonstra a declaração da função e seus parâmetros.

Listagem 3.5 – Protótipo da função register_driver

```
1 int register_driver(const char *path, const struct file_operations *fops
    ↪ , mode_t mode, void *priv);
```

1. *path*: nome do *device*;
2. *file_operations*: contêm os ponteiros de função definidas pelo *driver* que executas as operações no dispositivo;
3. *mode*: Na especificação POSIX, este parâmetro fornece o nível de permissão sob o *device*;
4. *priv*: ponteiro para uma estrutura adicional que pode ser utilizada pelo *driver*.

Ao realizar o procedimento descrito anteriormente teremos o *template* mínimo necessário para carregar um *driver* no Nuttx. A seguir, deve-se implementar uma função que será chamada no momento de inicialização do sistema operacional, sendo que essa função será responsável pela inicialização e registro do device *driver*. Na listagem 3.6 apresenta-se um exemplo de implementação da função principal de registro do *driver*.

Listagem 3.6 – Implementação da função teste_driver

```
1 void test_driver(void){
2  int aux;
3
4  aux = register_driver("/dev/test_driver", &fo_driver, 0644, NULL);
5  if (aux < 0)
6  {
7   printf("\n test_driver driver register_driver failed: %d\n", reg);
8  }
9  printf("\n test_driver driver initialized successfully!\n");
10 }
```

Por fim, é necessário garantir que o sistema operacional execute a função principal durante seu processo de inicialização e registre o *device driver* no caminho /dev sistema de arquivos. Para realizar esse procedimento, deve-se executar a função `test_driver` dentro da função *bringup* do tivac, que é responsável de registrar os *drivers* básicos da placa em uso. A listagem 3.23 apresenta a inclusão do *device driver* desenvolvido na função que carrega os *drivers* básicos de uma placa, neste caso, as placas da família de microcontroladores TM4C da *Texas Instruments*.

Listagem 3.7 – Alteração na função tm4c_bringup

```
1 /*******************************************************************
2 * Public Functions
3 ********************************************************************/
4
5 /*******************************************************************
```

```
* Name: tm4c_bringup
*
* Description:
*   Bring up board features
*
*******************************************************************/

int tm4c_bringup(void)
{
 int ret;
 test_driver();


 /* Register I2C drivers on behalf of the I2C tool */

 tm4c_i2ctool();
 ...
}
```

Faz-se necessário a declaração do .c contêndo a função principal do *driver* no arquivo de compilação (`Makefile`) da placa em uso. Nesse caso, no mesmo diretório do *driver*, adicione o arquivo `test_driver.c` ao parâmetro `CSRCS` sem a exclusão dos demais, assim como apresentado na listagem 3.8.

Listagem 3.8 – Adição de inclusão no arquivo Makefile da board TivaC

```
include $(TOPDIR)/Make.defs

CSRCS = tm4c_boot.c tm4c_bringup.c
CSRCS += test_driver.c
```

A partir deste momento o *driver* está pronto para ser compilado e registrado pelo Nuttx. Para realizar o teste, basta compilar o Nuttx na raiz do projeto `nuttx/` e gravar o binário utilizando a ferramenta lm4flash, conforme processo descrito na Seção 3.2.

3.5 Implementação USB CDC/ACM no Nuttx

Para exemplificar a implementação de um *device driver* para um periférico utilizando o *template* descrito anteriormente e, demonstrar o uso desse *device driver*, optou-se por desenvolver um *device driver* do tipo *character device* para uma das classes de dispositivos USB existentes e suportadas pelo microcontrolador utilizado. O NuttX possui uma implementação de controladora USB *host* para algumas plataformas de hardware em que suporta dispositivos da classe USB *mass storage* e HID *keyboard*. Já a implementação de USB *device* está disponível para diversas plataformas de *hardware*, suportando as classes USB *mass storage*, CDC/ACM *serial*, HID *keyboard* e HID *mouse*.

As interfaces seriais costumam ser a opção mais fácil ao trabalhar com microcontroladores para transferir caracteres, bastando conectar os pinos de serial GND e TX do microcontrolador a um conversor USB para serial. O RX só é necessário quando é desejado enviar caracteres para o microcontrolador. Embora seja um procedimento bem simples, existe a necessidade de utilizar um componente de hardware extra, nesse caso o adaptador USB para serial, torna-se um incomôdo. Se o microcontrolador tem uma interface USB funcional e uma

pilha USB, é bem provável que existam alternativas como o fornecimento de um dispositivo serial virtual via USB. Dessa forma, o único componente extra será um cabo USB que ficaria conectado entre o microcontrolador e o computador.

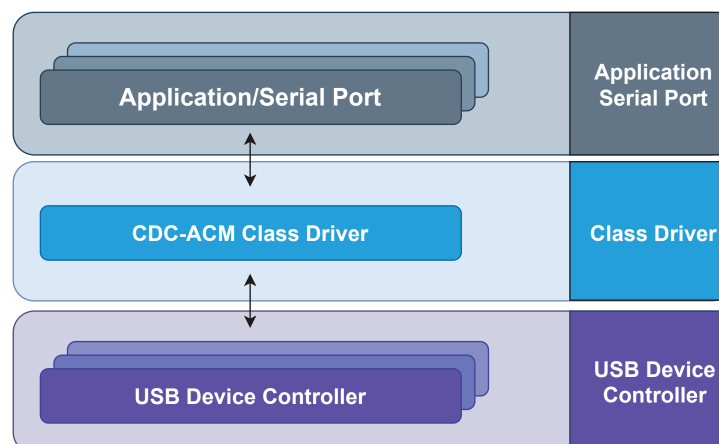
Uma escolha comum nesse cenário é fornecer um dispositivo em conformidade com o padrão USB *Communication Device Class* (CDC), especificamente o modelo de controle *Abstract Control Model* (ACM). Sabendo disso, optou-se por escolher essa classe para realizar a implementação no *template* do *device driver* para o NuttX desenvolvido anteriormente.

A Classe USB CDC/ACM é um protocolo documentado publicamente independente de fornecedor ou instituição privada e que pode ser utilizado para emular portas seriais por meio do protocolo USB. A USB CDC é uma classe que é composta de dispositivos de telecomunicação, dispositivos de rede como modems ASDL, adaptadores de Ethernet, *hubs* e etc. Tem a finalidade de especificar vários modelos para oferecer suporte a diferentes tipos de dispositivos de comunicação. Já o ACM é definido para oferecer suporte a dispositivos de modem legado e uma das vantagens do ACM é emulação de uma portal serial a partir de um dispositivo USB, facilitando o desenvolvimento de aplicações, fornecendo compatibilidade com dispositivos legados baseado em RS-232, tal abordagem garante a abstração da comunicação USB para a aplicação.

Nesse trabalho, será utilizado a característica do ACM de emular uma porta serial que normalmente é conhecida como Virtual COM port. Com a porta de comunicação disponível, é necessário conectar um cabo USB no dispositivo disponível em /dev/ttyACMx no ambiente linux.

A arquitetura da solução USB CDC/ACM para o Nuttx está demonstrada na Figura 7. O esquema está segmentado em três camadas simples, a camada da controladora USB, a camada da implementação do *driver* e a camada de aplicação.

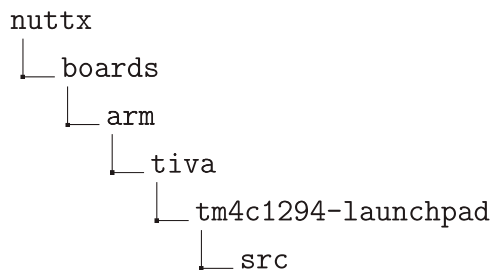
Figura 7 – Esquema USB CDC/ACM device driver.



Fonte: **Própria.**

Com o esquema definido na Figura 7, deve-se implementar a abstração apresentada em camadas no *template* construído anteriormente.

O primeiro passo é adicionar o *template* para o diretório correto, o arquivo principal com extensão `.c` deve ser inserido no diretório `nuttx/boards/$arch/$board/src`, em que os subdiretórios definidos com $ devem ser substituídos pela placa utilizada para desenvolvimento. No presente trabalho foi utilizado o seguinte diretório: `nuttx/boards/arm/tiva/tm4c1294-launchpad/src`.



Em sequência, podemos realizar a inclusão dos arquivos de cabeçalhos que serão utilizados pelo código principal do *driver*. Note que alguns cabeçalhos pertecem a duas biblitoecas estáticas que inicialmente não estão integradas ao Nuutx.

Listagem 3.9 – Importação dos cabeçalhos para o driver usb cdc/acm

```
/*****************************************************************
* Includes
*****************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <debug.h>
#include <errno.h>
#include <sched.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <inttypes.h>
#include <fcntl.h>

#include <nuttx/config.h>
#include <nuttx/board.h>

#include "tiva_gpio.h"
#include "tm4c1294-launchpad.h"
#include <mqueue.h>

#include <arch/board/inc/hw_ints.h>
#include <arch/board/inc/hw_memmap.h>
#include <arch/board/inc/hw_types.h>
#include <arch/board/inc/hw_uart.h>

```

```
32 #include <arch/board/driverlib/rom.h>
33 #include <arch/board/driverlib/rom_map.h>
34 #include <arch/board/driverlib/pin_map.h>
35 #include <arch/board/driverlib/timer.h>
36 #include <arch/board/driverlib/sysctl.h>
37 #include <arch/board/driverlib/interrupt.h>
38 #include <arch/board/driverlib/watchdog.h>
39 #include <arch/board/driverlib/gpio.h>
40 #include <arch/board/driverlib/uart.h>
41 #include <arch/board/driverlib/usb.h>
42 #include <arch/board/driverlib/rom.h>
43 #include <arch/board/driverlib/debug.h>
44
45 #include <arch/board/usblib/usblib.h>
46 #include <arch/board/usblib/usbcdc.h>
47 #include <arch/board/usblib/usb-ids.h>
48 #include <arch/board/usblib/device/usbdevice.h>
49 #include <arch/board/usblib/device/usbdcdc.h>
```

O microcontrolador TivaC possuí algumas API disponíveis para interagir com os periféricos, que são disponibilizadas como bibliotecas estáticas. Nesse projeto, optou-se por utilizar duas bibliotecas estáticas externas, a *usblib* e *driverlib*. Para adicionar estas bibliotecas no Nuttx, deve-se incluir os arquivos .a no diretório `$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/include/`, em nosso caso os arquvos ficaram posicionados nos seguintes diretórios:

a. `$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/include/driverlib`;

b. `$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/include/usblib`

Em seguida, deve-se utilizar o recurso disponível denonimado *EXTRA_LIBS*, o sistema de compilação do Nuttx suporta duas chamadas de variáveis especiais relacionadas ao Makefile, a *EXTRA_LIBS* e *EXTRA_LIBPATHS*. Então, faz-se necessário adicionar os *paths* correspondentes de cada biblioteca para os atributos *EXTRA_LIBPATHS* e indicar a biblioteca corresponde no parâmetro *EXTRA_LIBS*. Adicionou-se as seguintes linhas no arquivo de configuração Make.defs da arquitetura correspondente da placa de desenvolvimento. O arquivo está presente no diretório: `$(TOPDIR)/arch/arm/src/tiva/Make.defs`. A listagem 3.10 demonstra como as bibliotecas estáticas ser incluídas no arquivo *Make.defs*.

Listagem 3.10 – Adição no Make.defs da arquitetura tiva.

```
1 EXTRA_LIBPATHS += -L '$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/
    ↪ include/driverlib/gcc/'
2
3 EXTRA_LIBPATHS += -L '$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/
    ↪ include/usblib/gcc/'
4
5 EXTRA_LIBS += -ldriver
6 EXTRA_LIBS += -lusb
```

Deve-se observar que os arquivos das bibliotecas são nomeados como libdriver.a e libusb.a, mas para realizar o input no parâmetro *EXTRA_LIBS* utilizamos a abreviação da maneira exemplificada na listagem acima.

Listagem 3.11 – Importação dos cabeçalhos utilizados pela controladora usb

```
1 #include "drivers/buttons.h"
2 #include "drivers/pinout.h"
3 #include "USB/usb_serial_structs.h"
4 #include "USB/cdc/virtual_com.h"
```

As diretivas de compilação utilizadas no projeto, são descritas na listagem 3.12. São diretivas que serão utilizadas para configuração do *driver* usb por parte da controladora. Já a diretiva `QUEUE_NAME` representa o nome de uma queue que utilizamos para realizar a comunicação do *driver* com a controladora USB.

Listagem 3.12 – Diretivas de compilação utilizadas para configurar a controladora usb

```
1 #define GPIO_PD6_USB0EPEN       0x00031805
2 #define SEM_PRIO_NONE             0
3 #define SEM_PRIO_INHERIT          1
4 #define SEM_PRIO_PROTECT          2
5
6 #define QUEUE_NAME "/usb_queue"
```

Para realizar a proteção do recurso que nesse caso será o *buffer* leitura ou escrita, declaramos dois `mutex` para realizar essa proteção. A declaração é simples e ambos serão inicilizados na inicialiazação do *driver*. A listagem 3.13 demonstra a implementação do recurso de proteção.

Listagem 3.13 – Mutex utilizados para proteger leitura e escrita

```
1 FAR sem_t sRead;
2 FAR sem_t sWrite;
```

Neste momento, é necessário ativar o modo *interrupt handler* para o device. Que posteriormente será utilizado na função `open` no momento de realizar attach nas interrupções da USB.

Listagem 3.14 – Ativação do modo interrupt handler

```
1 static int tiva_usb_int(int irq, FAR void *context, FAR void *arg)
2 {
3  USB0DeviceIntHandler();
4  return 0;
5 }
```

Posteriormente, o arquivo principal .c possuí tudo que é preciso para implementação das funções POSIX. A primeira função à ser implementada é o método *open*, que nesse caso tem a finalidade de inicializar e configurar os pinos do TivaC para utilizar a placa em modo dispositivo USB e, em seguida, inicializar os *buffers* de comunicação utilizados pela porta virtual emulada. A seguir, a implementação do método *open* na listagem 3.15.

Listagem 3.15 – Implementação do método cdc_open

```
1 static int cdc_open(file_t *filep)
2 {
3   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
4  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
```

```
5   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
6   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);
7   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);
8
9   //
10  // Configure the device pins.
11  //
12  MAP_GPIOPinConfigure(GPIO_PD6_USB0EPEN);
13  MAP_GPIOPinTypeUSBAnalog(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
14  MAP_GPIOPinTypeUSBDigital(GPIO_PORTD_BASE, GPIO_PIN_6);
15  MAP_GPIOPinTypeUSBAnalog(GPIO_PORTL_BASE, GPIO_PIN_6 | GPIO_PIN_7);
16  MAP_GPIOPinTypeGPIOInput(GPIO_PORTQ_BASE, GPIO_PIN_4);
17
18  g_bUSBConfigured = false;
19  irq_attach(INT_USB0_TM4C129, tiva_usb_int, NULL);
20
21  USBBufferInit(&g_sTxBuffer);
22  USBBufferInit(&g_sRxBuffer);
23
24  USBStackModeSet(0, eUSBModeForceDevice, 0);
25  USBDCDCInit(0, &g_sCDCDevice);
26  Virtual_Comm_Init();
27 }
```

Não é possível realizar a leitura dos *buffers* diretamente da função *read* do *driver*. Faz-se necessário a implementação de algum método de comunicação entre o *driver* e os *buffers* da controladora USB.

A função `Virtual_Comm_Init()` chamada no método `open` tem a finalidade de inicializar a *queue* utilizada para comunicação entre o *buffer* da controladora usb e o *device driver* USB CDC/ACM.

O Nuttx suporta POSIX *Named Message Queue Interfaces* que serve para comunicação entre tarefas no Nuttx. Optou-se por utilizar esse recurso, visto que sua utilização é muito semelhante a utilização do *driver* que estamos implementando, devido a interface POSIX.

Para inicializar uma *queue* utilizando a *mqueue*, é necessário inicializar e atribuir um estrutura do tipo `mq_attr`, que possuí os seguintes parâmetros:

a. mq_maxmsg;

b. mq_msgsize.

Note que existem outros dois parâmetros, mas não é necessário utilizá-los porque serão ignorados e setados como parâmetos default.

Em seguida, utilizamos a função `mq_open` para criar um descritor de *queue* com os seguintes parâmetros de entrada:

a. mqName: QUEUE_NAME;

b. oflags: 0_CREAT;

c. mode: 0644;

d. mq_attr: attr.

Em seguida, é requisitado o método `mq_close` da fila recém criada. Vale ressaltar que ao invocar o método *close* o `QUEUE_name` não perde o vínculo com a fila criada, para excluir o vínculo do descritor de fila com a diretiva `QUEUE_NAME` deve-se utilizar o método `mq_unlink`.

Listagem 3.16 – Implementação do método Virtual_Comm_Init

```
void Virtual_Comm_Init(void)
{
 struct mq_attr attr;
 mqd_t mq;

 attr.mq_flags = 0;
 attr.mq_maxmsg = 64;
 attr.mq_msgsize = 1;
 attr.mq_curmsgs = 0;

 mq = mq_open(QUEUE_NAME, O_CREAT, 0644, &attr);
 mq_close(mq);
}
```

O metódo seguinte `USBUARTPrimeTransmit` é implementado no arquivo principal da `virtual_comm.c` e deve ser modificado para transmitir os *bytes* lidos do `g_sRxBuffer`. Após realizar a leitura dos *bytes* transmitidos pela porta serial virtual, é necessário enviar os mesmos *bytes* lidos para a *queue* com a função `mq_send()`, com comportamento similar a função *write*. Em seguida, os caracteres lidos devem ser enviados para o *buffer* de transmissão para que sejam visualizados no terminal que estará com *device* aberto.

Listagem 3.17 – Alteração do método USBUARTPrimeTransmit

```
void USBUARTPrimeTransmit(void)
{
 mqd_t  mq;
 unsigned long ulRead;
 unsigned long read_bytes;
 unsigned char ucChar[32];

 ulRead = USBBufferRead((tUSBBuffer *)&g_sRxBuffer, (unsigned char *)&
    ↪ ucChar, 32);

 uint32_t written = 0;
 int status = 0;
 if(ulRead)
 {
  read_bytes = 0;
  do
  {
   mq = mq_open(QUEUE_NAME, O_EXCL | O_RDWR);
   status = mq_send(mq,&ucChar[read_bytes],1,0);
   mq_close(mq);

   if(status == -1){
    perror("mq_send failure on mqfd");
   }
   written = USBBufferWrite((tUSBBuffer *)&g_sTxBuffer, &ucChar[
    ↪ read_bytes], 1);

   read_bytes++;
   ulRead--;
  }while(ulRead);
 }
}
```

Para o método `read` é necessário implementar a comunicação por meio de uma *queue* que armazena os elementos lidos no *buffer* por meio da porta serial emulada. Primeiro, antes de realizar qualquer procedimento de leitura, deve-se utilizar o método *nxsem_wait* com o *muttex*

para proteção de recurso no momento de leitura. Posteriormente, utiliza-se a função `mq_open` e em seguida, é necessário chamar a função `mq_receive` que também possuí interfaces POSIX, então é de conhecimento que o funcionamento é semelhante as função *read* do *device driver* desse trabalho. Nesse caso, da mesma maneira que são informados os parâmetros para a função *read* do *driver* usbcdc, é necessário transmitir os seguintes parâmetros para a *mq_receive*:

a. mq: o descritor de arquivos da *queue*;
b. buf: o *buffer* a ser armazenado a leitura;
c. buflen: o número de *bytes* a serem lidos;
d. NULL:

Após realizar a leitura na *queue* de maneira sucedida, é necessário chamar a função *mq_close* para finalizar o descritor de arquivos da *queue* atribuído a variável `mq`. Posteriormente, utiliza-se a função *nxsem_post* para liberar o recurso. Em seguida, a listagem 3.18 demonstra a implementação do método descrito anteriormente.

Listagem 3.18 – Implementação do método cdc_read

```
1  static ssize_t cdc_read(file_t *filep, void *buf, size_t buflen)
2  {
3   ssize_t len = 0;
4   mqd_t  mq;
5   ssize_t ulRead = 0;
6
7   if(buf == NULL || buflen < 1)
8   {
9    return -EINVAL;
10  }
11
12  (void)nxsem_wait(&sRead);
13  mq = mq_open(QUEUE_NAME, O_RDONLY | O_NONBLOCK);
14  ulRead = mq_receive(mq, buf, buflen, NULL);
15  mq_close(mq);
16  (void)nxsem_post(&sRead);
17
18  return ulRead;
19 }
```

O método `write` por sua vez, tem a finalidade receber um *buffer* com uma certa de quantidade de caracteres a serem escritos na portal serial virtual. Mas nesse caso não se faz necessário a utilização de uma *queue* para intermediar a comunicação entre *driver* e *buffer* da controladora USB CDC. Optou-se por escrever os caracteres recebidos diretamente no *buffer* de transmissão da controladora. Foi escolhido esse método porque não há uma maneira convencional de realizar um acionamento e identificar que os caracteres devem ser enviados, em vez disso, pode-se simplesmente escrevê-los no *buffer* de transmissão que será gerado uma interrupção e os mesmos serão disponíbilizados no terminal virtual. A listagem 3.19 demonstra a implementação da função *cdc_write*.

Listagem 3.19 – Implementação do método cdc_write

```
1  static ssize_t cdc_write(file_t *filep, void *buf, size_t buflen)
2  {
3   unsigned char buf_aux[buflen];
4   ssize_t nbytes = 0;
```

```
5
6  char carac = ((char *) buf) [0];
7
8  (void)nxsem_wait(sbWrite);
9  uint32_t written = 0;
10
11 for(int i = 0; i < buflen; i++){
12  buf_aux[i] = ((char *) buf) [i];
13  nbytes += USBBufferWrite((tUSBBuffer *)&g_sTxBuffer, &buf_aux[i], 1);
14 }
15
16 (void)nxsem_post(&sWrite);
17
18 return nbytes;
19 }
```

Listagem 3.20 – Implementação do método cdc_ioctl

```
1 static int cdc_ioctl(file_t *filep, int request, unsigned long arg)
2 {
3  switch(request)
4  {
5   case 1:
6   {
7    tLineCoding psLineCoding;
8
9    psLineCoding.ui8Databits = 7;
10   psLineCoding.ui8Parity = USB_CDC_PARITY_ODD;
11   psLineCoding.ui8Stop = USB_CDC_STOP_BITS_1;
12
13   uint32_t ui32Event = USBD_CDC_EVENT_SET_LINE_CODING;
14   uint32_t ui32MsgValue = 0;
15   uint32_t ret = ControlHandler(0, ui32Event, ui32MsgValue, &
      ↪ psLineCoding);
16
17   if(!ret){
18    printf("\nError to use SetLineCoding\n");
19   }
20
21   tLineCoding rsLineCoding;
22
23   GetLineCoding(&rsLineCoding);
24  }
25  break;
26  case 2:
27  {
28   tLineCoding psLineCoding;
29
30   GetLineCoding(&psLineCoding);
31  }
32  break;
33  }
34 return 0;
35 }
```

E por fim, a implementação da função `up_usbcdc` que fica responsável de inicializar os *mutex* utilizados nos métodos *read* e *write*, e também, de registrar o *device driver* em tempo de compilação utilizando o método `register_driver` abordando anteriormente.

Listagem 3.21 – Implementação do método principal usb_cdc

```
1 /****************************************************************
2 * Initialize device, add /dev/usb0
```

```
3 *******************************************************/
4
5 void up_usbcdc(void){
6
7  int ret = 0;
8  ret = nxsem_init(&sRead,0,0);
9  (void)nxsem_post(&sRead);
10
11  int ret2 = 0;
12  ret2 = nxsem_init(&bWrite,0,0);
13  (void)nxsem_post(&bWrite);
14
15  //sem_init(&sem, 0, 0);
16  sem_setprotocol(&sRead, SEM_PRIO_NONE);
17  sem_setprotocol(&bWrite, SEM_PRIO_NONE);
18
19  usbtask();
20
21  int reg;
22
23  reg = register_driver("/dev/usb0", &usb_cdc_ops, 0644, NULL);
24  if (reg < 0)
25  {
26   printf("\nusb_cdc driver register_driver failed: %d\n", reg);
27  }
28  printf("\nusb_cdc driver initialized successfully!\n");
29 }
```

Com a finalização do desenvolvimento dos métodos principais do *driver*, basta realizar a implementação das configurações necessárias para que o *driver* seja compilado e possa ser executado pelo Nuttx em seu *runtime*. Para realizar a inclusão do *driver* recém desenvolvido, localize o arquivo Makefile principal do TivaC, que fica no diretório: `$(TOPDIR)/boards/arm/tiva/tm4c1294-launchpad/`, e em seguida, deve-se atribuir a variável *CSRCS* os arquivos que serão compilados.

Listagem 3.22 – Adição dos arquivos .c no Makefile do TivaC

```
1 CSRCS = tm4c_boot.c tm4c_bringup.c
2 CSRCS += test_driver.c
3 CSRCS += USB/usb_serial_structs.c
4 CSRCS += USB/cdc/virtual_com.c
```

Com o arquivo indicado para ser compilados junto aos demais .c do projeto, deve-se chamar a função principal do *driver* responsável de realizar o registro do mesmo. Optou-se por invocar o método principal na função `tm4c_bringup` que é responsável por inicializar as ferramentas do TivaC, tais como spi, i2c, usb, entre outros. Basta chamar a função principal no escopo da função, como indica a listagem 3.23.

Listagem 3.23 – Alteração do método tm4c_bringup da board TivaC

```
1 /*******************************************************
2 * Name: tm4c_bringup
3 * Description:
4 *   Bring up board features
5 *******************************************************/
6
7 int tm4c_bringup(void)
8 {
9  int ret;
```

```
10
11   tm4c_i2ctool();
12   up_usbcdc();
13   ...
14 }
```

Por fim, quando o Nuttx executar a função *tm4c_bringup*, o *driver* será registrado e o device correspondente pode ser encontrado no caminho */dev*.

Neste capítulo foi definido uma metodologia de criação de *device drivers* do tipo *character device* para o Nuttx. Em sequência, abordou-se um exemplo de confecção de um *device driver* utilizando a metodologia apresentada. Posteriormente, no próximo capítulo será apresentado um exemplo de aplicação utilizando o *device driver* implementado anteriormente.

4 RESULTADOS OBTIDOS

Este capítulo apresenta a implementação de uma aplicação utilizando o *device driver* desenvolvido no Capítulo 3. E por fim, a análise dos resultados obtidos utilizando o *device driver*.

4.1 Implementação de uma Aplicação Nuttx

O diretório apps/ fornece várias aplicações encontradas em subdiretórios. Essas aplicações não fazem parte do core do Nuttx, mas são desenvolvidas para ajudar os desenvolvedores que desejam implementar suas próprias aplicações. O diretório apps/ é uma extensão ao *core* do Nuttx que pode ser utilizada mas não é essencial para o funcionamento do RTOS.

O diretório de aplicações padrão do Nuttx deve ser nomeado como *apps/*. Esse diretório deve aparecer na árvores de diretórios no mesmo nível do diretório principal do Nuttx.

Se o diretório for nomeado corretamente, o Nuttx será capaz de encontrar o diretório de aplicações. Se o diretório possuir uma localização diferente da indicada, é necessário definir esse nova localização para o sistema de compilação do Nuttx.

Para executar uma aplicação desenvolvida via linha de comando, se faz necessário a utilização do recurso de aplicações integradas, que é um método de requisitar aplicações personalizadas por meio de linha de comando utilizando o NuttShell(NSH). O NSH oferece suporte a um método contínuo de invocação dos aplicativos, quando a seguinte opção for habilitada no arquivo de configuração do Nuttx: CONFIG_NSH_BUILTIN_APPS=y

As aplicações registradas no diretório apps/ estarão acessíveis por meio de linha comando no NSH. É possível digitar `help` para visualizar a lista de aplicações registradas no Nuttx.

O primeiro momento de desenvolvimento da aplicação se dá na escolha do diretório em que ficará registrado. No Nuttx, uma aplicação pode estar registrada como um exemplo, um utilitário do sistema ou apenas um comando. A diferença entre as três opções para o Nuttx do ponto de vista de compilação é a pasta em que irão ficar localizados os arquivos da aplicação. Para utilizar o driver desenvolvido no Capítulo 3, optou-se por desenvolver uma aplicação do tipo exemplo e que todo e qualquer arquivo utilizado nessa aplicação pode ser reutilizada para os outros tipos de aplicações.

Basicamente, uma aplicação requer quatro arquivos principais para que seja possível compilar, configurar e registrar a aplicação no RTOS. Os arquivos principais são:

a. app.c;
b. Kconfig;

c. Make.defs;

d. Makefile.

O item a. pode ser um conjunto de arquivos ou um único arquivo, e o mesmo pode conter acesso aos arquivos de cabeçalhos do *device driver*, bibliotecas C/C++, bibliotecas do Nuttx e qualquer outra biblioteca e arquivos presentes no projeto.

O arquivo `Kconfig` serve para indicar quais serão as opções que serão exibidas no menuconfig do Nuttx, tais como descrição, nome e dependências.

O arquivo `Make.defs` adiciona a aplicação quando ela é selecionada no menuconfig de configuração ou incluída diretamente no arquivo de configuração principal do Nuttx.

E por fim, o arquivo `Makefile` serve para compilar a aplicação e indicar quais serão as opções utilizadas na aplicação e no sistema quando ela for executada no NuttShell.

Com os arquivos criados, o próximo passo é criar o diretório em que irá ficar a aplicação no diretório apps. Para esta aplicação deve-se criar uma pasta chamada usbcdc no diretório *apps/examples/*. Em seguida, é necessário criar o arquivo principal da aplicação, nesse caso, foi escolhido nomear o arquivo como `usbcdc.c`.

O *script* usbcdc.c implementa dentre alguns métodos, a *main* que é responsável por inicializar a aplicação e requisitar os métodos que farão a utilização das funções do *device driver*.

Listagem 4.1 – Main aplicação usbcdc

```
1  #ifdef CONFIG_BUILD_KERNEL
2  int main(int argc, FAR char *argv[])
3  #else
4  int usbcdc_main(int argc, char *argv[])
5  #endif
6  {
7   printf("usbcdc app works!!\n");
8
9   int ret = 0;
10  ret = usbtask();
11
12  return 0;
13 }
```

A chamada da função `usbtask` na *main*, refere-se ao método responsável por implementar a inicialização do terminal e um *loop* infinito que fica lendo e escrevendo caracteres na portal virtual serial por meio das funções desenvolvidas no *device driver*.

Listagem 4.2 – usbtask

```
1  static int usbtask()
2  {
3   usb_terminal_init();
4   (void)usb_terminal_add_cmd((command_t*)&usb_ver_cmd);
5
6   while(1)
7   {
8    /* Call the application task */
9    usb_terminal_process();
10  }
11 }
```

O método `printf_usb_app` é um bom exemplo de utilização do *device driver*. Basicamente, o método recebe uma *string* como parâmetro de entrada que será escrita no *buffer* da portal serial virtual. Realiza-se a abertura do *device* com a função *open* utilizando a flag que indica somente leitura. Em seguida, é necessário enviar os caracteres da *string* para o *buffer* da portal serial virtual utilizando a função *write*. A listagem 4.3 demonstra a implementação do método descrito acima.

Listagem 4.3 – Utilização do método write na aplicação

```
1  void printf_usb_app(char *s)
2  {
3   int fd = open("/dev/usb0", O_WRONLY);
4
5   uint32_t count = 0;
6   uint32_t written = 0;
7   uint32_t char_to_written = 0;
8   uint32_t written_char = 0;
9
10  char *string = s;
11
12  while(*string)
13  {
14   count++;
15   string++;
16  }
17
18  char_to_written = count;
19
20  do
21  {
22   written = write(fd, (unsigned char *)&s[written_char], char_to_written
      ↪ );
23
24   written_char += written;
25   char_to_written -= written;
26  }while(written_char != count);
27
28  close(fd);
29 }
```

O método `usb_terminal_process` realiza a leitura dos caracteres individuais lidos do *device* /dev/usb0 e em seguida analisa se é um comando válido ou não. A leitura ocorre utilizando a função *read* implementada anteriormente.

Listagem 4.4 – Implementação do método usb_terminal_process

```
1  void usb_terminal_process(void)
2  {
3   char key_detected = 0;
4
5   while(1)
6   {
7    int fd;
8    fd = open("/dev/usb0", O_RDWR);
9    unsigned char data;
10   ssize_t bytes_read;
11   char c;
12   fflush(data);
13
14   bytes_read = read(fd, &data, 1);
```

```
15  c = (char)data;
16
17  if(bytes_read == 1)
18  {
19   if ((c != '\n') && (c != '\r'))
20   {
21    // If not backspace
22    if (c != 0x7F)
23    {
24     if (USBSilentMode == FALSE)
25     {
26      // If not up key
27      if (c != '\033')
28      {
29       // if not 2 chars that came together with up key
30       if (!key_detected){
31        //putchar_usb_app(c);
32       }
33      }else
34      {
35       // if up key, prepare to discard next two chars
36       // and print the last command, doing it the current command
37       key_detected = 2;
38       // erase last chars in case of up key pressed
39       while(usb_cmd_line_ndx)
40       {
41        putchar_usb_app(0x7F);
42        usb_cmd_line_ndx--;
43       }
44       printf_usb_app(last_cmd_line);
45       char *cp = last_cmd_line;
46       usb_cmd_line_ndx = 0;
47       while(*cp)
48       {
49        usb_cmd_line[usb_cmd_line_ndx] = last_cmd_line[usb_cmd_line_ndx
    ↪ ];
50        usb_cmd_line_ndx++;
51        cp++;
52       }
53       usb_cmd_line[usb_cmd_line_ndx] = '\0';
54      }
55     }
56    }else
57    {
58     if (usb_cmd_line_ndx)
59     {
60      putchar_usb_app(c);
61     }
62    }
63   }
64
65   /* Execute command if enter is received, or usb_cmd_line is full. */
66   // if ((c=='\r') || (usb_cmd_line_ndx == sizeof(usb_cmd_line)-2))
67   if(c=='\r')
68   {
69    int usb_start = usb_skipp_space(usb_cmd_line, 0);
70    int usb_end = usb_find_word(usb_cmd_line, usb_start);
71
72    // copy last command
73    int cpi = 0;
74    while(cpi<usb_cmd_line_ndx)
75    {
76     last_cmd_line[cpi] = usb_cmd_line[usb_start+cpi];
77     cpi++;
78    }
```

```
79      last_cmd_line[cpi] = '\0';
80
81      int usb_x;
82
83      /* Separate command string from parameters, and close
84      parameters string. */
85      usb_cmd_line[usb_end] = usb_cmd_line[usb_cmd_line_ndx] = '\0';
86
87      /* Identify command. */
88      usb_x = usb_find_command(usb_cmd_line + usb_start);
89
90      /* Command not found. */
91      if (usb_x == -1)
92      {
93       printf_usb_app("\nUnknown command!\r\n");
94      }
95      else
96      {
97       (*usb_cmds[usb_x]->func)(usb_cmd_line+usb_end + 1);
98      }
99      usb_cmd_line_ndx=0;
100     usb_print_prompt();
101     USBSetSilentMode(FALSE);
102    }
103    else
104    { /* Put character to usb_cmd_line. */
105     if (c=='\b')
106    {
107
108    if (usb_cmd_line_ndx > 0)
109    {
110     usb_cmd_line[usb_cmd_line_ndx] = '\0';
111     usb_cmd_line_ndx--;
112    }
113   }
114   else if(c=='\n'){
115    continue;
116   }
117   else
118   {
119    if (c == 0x7F)
120    {
121     if (usb_cmd_line_ndx)
122     {
123      usb_cmd_line[usb_cmd_line_ndx]=0;
124      usb_cmd_line_ndx--;
125      usb_cmd_line[usb_cmd_line_ndx]=0;
126     }
127    }else
128    {
129     // Only accept chars different to up key and its two next chars
130     if (c != '\033')
131     {
132      if (!key_detected)
133      {
134       usb_cmd_line[usb_cmd_line_ndx++] = c;
135      }else
136      {
137       key_detected--;
138      }
139     }
140    }
141   }
142   }
143   }
```

```
144    close(fd);
145  }
146 }
```

A configuração do arquivo `Kconfig`, define quais serão as opções visualizadas na ferramenta `menuconfig` do Nuttx, e podem ser visualizadas na Figura 8. Basicamente, consiste na descrição da aplicação e na configuração dos elementos básicos como *priority* e *stacksize*. Optamos por utilizar valores padrões de 100 para *priority* e 2048 bytes para o parâmetro *stacksize*. A listagem 4.5 demonstra como configurar o *script* Kconfig.

Listagem 4.5 – Arquivo de configuração Kconfig

```
1 config EXAMPLES_USBCDC
2 tristate "\"USB CDC!\" example"
3 default n
4 ---help---
5 Enable the \"Hello, USBCDC!\" example
6
7 if EXAMPLES_USBCDC
8
9 config EXAMPLES_USBCDC_PROGNAME
10 string "USB_CDC"
11 default "USB_CDC"
12 ---help---
13 This is the name of the program that will be use when the NSH ELF
14 program is installed.
15
16 config EXAMPLES_USBCDC_PRIORITY
17 int "USBCDC task priority"
18 default 100
19
20 config EXAMPLES_USBCDC_STACKSIZE
21 int "USBCDC stack size"
22 default 2048
23
24 endif
```

Em sequência, o *script* `Make.defs` irá adicionar a aplicação ao conjunto de aplicações configuradas que serão compiladas quando o processo de compilação for executado. Basta utilizar o padrão para inclusão da aplicação desenvolvida de acordo com a listagem abaixo.

Listagem 4.6 – Script Make.defs

```
1 ifneq ($(CONFIG_EXAMPLES_USBCDC),)
2 CONFIGURED_APPS += $(APPDIR)/examples/usbcdc
3 endif
```

E por fim, a construção do *script* `Makefile` é necessária para compilar corretamente a aplicação indicando quais são os arquivos .c que serão incluídos. O parâmetro `EXTRADEFINES` deve-se utilizado como indica a listagem 4.7 para compilar a aplicação para a placa em uso.

Listagem 4.7 – Script Makefile

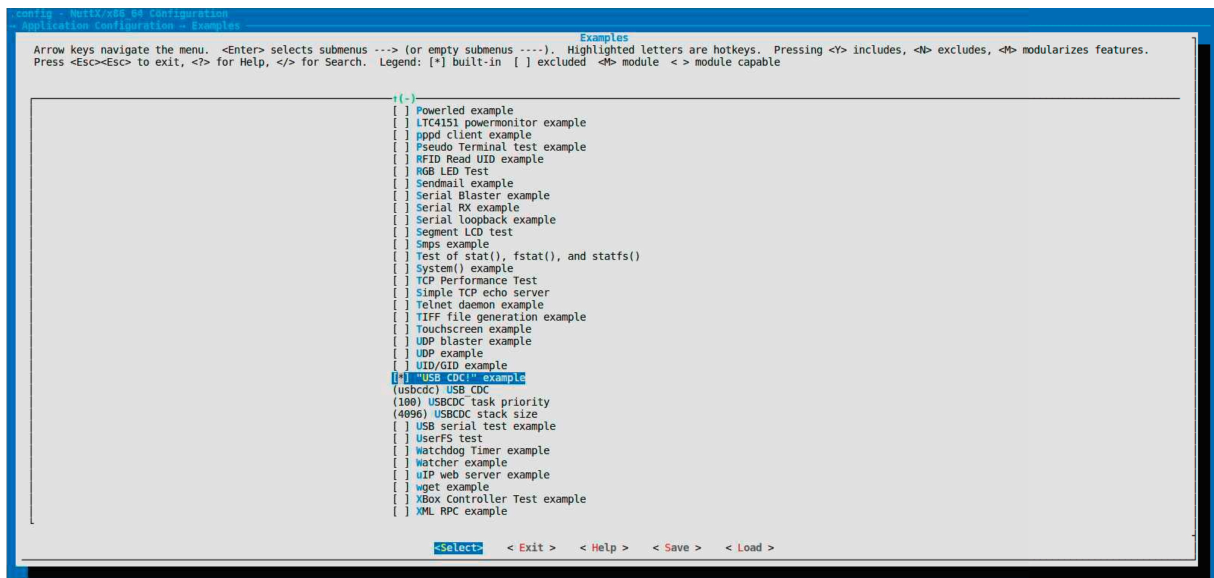
```
1 include $(APPDIR)/Make.defs
2
3 EXTRADEFINES += -DPART_TM4C1294NCPDT -DTARGET_IS_TM4C129_RA0 -Dgcc
4
5 # usbcdc built-in application info
6
```

```
7  PROGNAME   = $(CONFIG_EXAMPLES_USBCDC_PROGNAME)
8  PRIORITY   = $(CONFIG_EXAMPLES_USBCDC_PRIORITY)
9  STACKSIZE  = $(CONFIG_EXAMPLES_USBCDC_STACKSIZE)
10 MODULE     = $(CONFIG_EXAMPLES_USBCDC)
11
12 MAINSRC = usbcdc.c
13 CSRCS = drivers/buttons.c drivers/pinout.c USB/usb_serial_structs.c
14 CSRCS += USB/cdc/usb_terminal.c USB/cdc/usb_terminal_commands.c
15
16 include $(APPDIR)/Application.mk
```

Após realizar o procedimento de *build* do Nuttx, é necessário atualizar o arquivo de configuração para incluir a aplicação no processo de compilação. Para realizar este procedimento, deve-se o comando `make menuconfig` e realizar a inclusão da configuração no caminho:

Application Configuration > Examples

Figura 8 – Ativando aplicação usbcdc no arquivo de configuração do Nuttx.



Fonte: **Própria.**

Após ativar a aplicação no *menuconfig* do Nuttx, basta realizar o procedimento padrão de compilação do Nuttx que a aplicação estará disponível no NuttShell. Para visualizar as aplicações disponíveis no Nuttx, execute o comando `help`. A aplicação deve estar incluída na lista *Builtin Apps*, como indica a Figura 9.

Executando a aplicação pelo `NuttShell`, basta conectar no *device* disponibilizado no computador *host*, geralmente disponível no caminho */dev* com o nome do *device* no formato */dev/ttyACMx*. A conexão pode ser realizada pelo *software* `Putty`. Em sequência, algumas sequências de caracteres podem ser enviados para demonstrar o funcionamento do *driver*.

Figura 9 – Builtin Apps

Fonte: **Própria.**

4.2 Análise dos Resultados

O *device driver* funciona como determinado, e o mesmo pode ser utilizado por qualquer aplicação que tem o intuito de comunicar sequências de caracteres por meio da portal serial virtual. Por outro lado, o *driver* não apresenta robustez, visto que constantemente, a aplicação deve abrir descritores de arquivos para realizar a comunicação. A cada *system call*, a implementação do *driver* comunica com a *queue* utilizada para a comunicação com a controladora, e esse processo também abre descritores de *queue*.

O constante fluxo de operações *open* e *close* no *device* é um problema, causando o aumento de latência de entrada e saída de caracteres, uma vez que a aplicação bloqueia até que os dados sejam lidos e retirados da *queue*.

Naturalmente, é esperado que o *device driver* implementado com as interfaces POSIX seja um pouco inferior no desempenho comparado com um *device driver* sem a implementação da interface POSIX, visto que existe uma camada adicional entre a chamada das funções POSIX e as funções implementadas no *driver*. Existe a necessidade de analisar o custo benefício entre o tempo e custo do projeto. O desenvolvimento do código com princípios de portabilidade se faz necessário afim de economizar a reescrita do mesmo *device driver* para ambientes diferentes.

5 CONCLUSÃO

Este trabalho apresentou uma abordagem para desenvolver *device drivers* do tipo *character device* no RTOS Nuttx, utilizando princípios de desenvolvimento com foco em portabilidade entre sistemas operacionais. O desenvolvimento foi segmentado em três etapas, que unidas constituem todas camadas necessárias para realizar o registro e utilização de um *device driver*.

Na primeira etapa de trabalho foi desenvolvido os arquivos principais do device driver na camada *upper half*. Constituem nessa camada, a implementação das funções principais do *driver*, a controladora USB que realiza o controle dos *buffers* de leitura e escrita na porta serial virtual e a comunicação entre a controladora e as funções POSIX. Em seguida, foram realizadas todas as configurações necessárias nos arquivos de compilação e de configuração da placa utilizada no `menuconfig` do Nuttx. Também se fez necessário o desenvolvimento de uma aplicação na camada de aplicações do Nuttx externa ao core, com a finalidade de demonstrar a utilização do *device driver*.

Constatou-se que o principal problema do desenvolvimento de *device drivers* para o Nuttx é a complexidade e a falta de documentação. Não existem maneiras convencionais de *debug* no Nuttx, o que acaba dificultando o processo de desenvolvimento de *devices drivers*, visto que toda e qualquer modificação precisa ser compilada e testada com chamadas da função *printf*, que só é possível ser utilizada após realizar uma determinada configuração no arquivo principal de configurações do Nuttx. Fica evidente que a quantidade de alterações no arquivo de configuração que devem ser realizadas, dificulta o desenvolvimento, porque as configurações não possuem descrições claras do seu objetivo. Outro ponto, a quantidade de arquivos de compilação (Makefile) que são necessárias alterações. Alguns procedimentos sugeridos na documentação oficial não funcionam como esperado em praticado, tendo que realizar alternativas estabelecendo um *workaround* no projeto.

A busca pela semelhança com sistemas embarcados baseados em Linux, acaba por se tornar uma vantagem tanto quanto desvantagem. Entre os pontos positivos estão, a portabilidade e compatibilidade com a norma POSIX, árvore de diretórios semelhantes ao do Linux e o *layout* de *device driver* semelhantes ao do linux. E entre os pontos negativos estão a complexidade na configuração de arquivos do sistema e muitas modificações para tornar a portabilidade viável.

Na teoria sabemos que a utilização da norma POSIX estabelece regras para manter a portabilidade de componentes entre sistemas operacionais baseados no mesmo padrão. No entanto, para fazer a portabilidade de um *device driver* desenvolvido em outro sistema operacional baseado na compatibilidade da norma POSIX, são necessários uma série de modificações e configurações. Um *device driver* simples precisa de uma série de adaptações e inclusões nas configurações do Nuttx para que o mesmo seja compilado e possa ser executado.

5.1 Trabalhos Futuros

Para trabalhos futuros, sugere-se o desenvolvimento da controladora USB na camada *lower half* do Nuttx, para que a *stack* USB utilizada no device driver da USB CDC/ACM, seja a do Nuttx e não a do TM4C1294. Procedimento do qual é necessário a implementação de baixo nível que configura e comunica diretamente com o periférico.

5.2 Considerações Finais

O resultado obtido com esse trabalho permite a criação do ciclo completo de um *device driver*, desde o registro do *driver* abstraído como *device*, até sua utilização na camada de aplicação. O objetivo do trabalho foi atingido, e a sequência de passos definida pode ser utilizada para criar qualquer *driver* do tipo *character device*.

O código do projeto estará disponível como código aberto, com o intuito de contribuir com a comunidade de desenvolvimento de *software open-source*. Ficará disponibilizado no sistema de versionamento GitHub, no diretório remoto ”`fernandomartinsrib/device_driver_nx`”.

Referências

ASSIS., A. C. de. **Conheça o RTOS NuttX.**, 2019. Disponível em: <http://nuttx.org/doku.php?id=presentations:discover-nuttx/>. Acesso em: 10 mai. 2019.

CARRO, L.; WAGNER, F. **Capítulo 2 das Jornadas de atualização em informática**. 1. ed. Campinas: Sociedade Brasileira de Computação, 2003.

DENARDIN, G. W.; BARRIQUELO, C. H. **Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados**. 1. ed. São Paulo: Editora Edgard Blucher Ltda, 2019.

LEWINE, D. A. **POSIX programmer's guide**. 1. ed. Sebastopol CA: O'Reilly Associates, 1991.

MARWEDEL, P. **Embedded system design**. 2. ed. Springer, 2011.

MORAES, C. H. V.; ALMEIDA, R. M. A.; SERAPHIM, T. F. P. **Programação de sistemas embarcados: desenvolvendo software para microcontroladores em lingugagem C**. 1. ed. São Paulo: Editora Elsivier Ltda, 2016.

NUTT, G. **NuttX RTOS Porting Guide.**, 2019. Disponível em: <http://nuttx.org/doku.php?id=documentation:portingguide>. Acesso em: 10 jun. 2019.

OLIVEIRA, A. S.; ANDRADE, F. S. **Sistemas embarcados hardware e firmware na pratica**. Erica, 2006.

PRADO, S. **Linux device drivers.**, 2016. Disponível em: <http://e-labworks.com/treinamentos/drivers/source>. Acesso em: 20 jun. 2019.

RUBINI, A.; CORBET, J.; HARTMAN, G. K. **Linux device drivers**. 3. ed. O'Reilly, 2005.

SARASWAT, P. K. **User space device drivers i introduction and implementation using VGAlib library.**, 2010. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.1805&rep=rep1&type=pdf>. Acesso em: 16 jun. 2019.

STALLMAN, R. **Linux and the GNU System.**, 2019. Disponível em: <https://www.gnu.org/gnu/linux-and-gnu.en.html>. Acesso em: 3 jul. 2019.

VENKATESWARAN, S. **Essential linux device drivers**. 1. ed. Prentice Hall, 2008.

ZURITA, M. **Projeto de sistemas embarcados.**, 2014. Disponível em: <https://www.researchgate.net/publication/267298521_Projeto_de_Sistemas_Embarcados>. Acesso em: 02 abr. 2019.