Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia Eletrônica

Desenvolvimento de *drivers* de dispositivo em *Linux* embarcado para os módulos de *hardware* RC522 e LCD HD44780

Autor: Júlio César Schneider Martins
Orientador: Prof. Dr. Diogo Caetano Garcia

Brasília, DF
2024

Júlio César Schneider Martins

Desenvolvimento de *drivers* de dispositivo em *Linux* embarcado para os módulos de *hardware* RC522 e LCD HD44780

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Diogo Caetano Garcia

Brasília, DF

2024

Júlio César Schneider Martins

Desenvolvimento de *drivers* de dispositivo em *Linux* embarcado para os módulos de *hardware* RC522 e LCD HD44780/ Júlio César Schneider Martins. – Brasília, DF, 2024-

102 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Diogo Caetano Garcia

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2024.

1. Device drivers. 2. Linux embarcado. I. Prof. Dr. Diogo Caetano Garcia. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Desenvolvimento de *drivers* de dispositivo em *Linux* embarcado para os módulos de *hardware* RC522 e LCD HD44780

CDU 02:141:005.6

Júlio César Schneider Martins

Desenvolvimento de *drivers* de dispositivo em *Linux* embarcado para os módulos de *hardware* RC522 e LCD HD44780

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 16 de setembro de 2024 – Data da aprovação do trabalho:

Prof. Dr. Diogo Caetano Garcia
Orientador

Prof. Dr. Edson Mintsu Hung
Convidado 1

Prof. Dr. Tiago Alves da Fonseca
Convidado 2

Brasília, DF
2024

Dedico este trabalho aos meus pais, Carlson Jorge Martins da Silva e Lucimara Schneider Martins, que proveram o apoio que o possibilitou. Também a Alexandra Elbakyan, criadora da ferramenta Sci-Hub, que teve grande importância neste trabalho, por uma ciência livre e aberta.

Resumo

Esse trabalho buscou estudar e desenvolver *drivers* de dispositivo, desenvolvidos para módulos de *hardware* no contexto de sistema embarcados, especificamente para sistemas operacionais Linux. Foi realizada a análise da bibliografia sobre o que são *drivers* de dispositivo, e quando devem ser utilizados, ainda indicando as dificuldade no desenvolvimento de *drivers* de dispositivo. Nesse trabalho também foi realizado o desenvolvimento de *drivers* de dispositivo para *displays* com controlador compatível ao controlador LCD Hitachi HD 44780 e para a placa de comunicação RFID RC522, foi identificado o correto funcionamento a partir de testes com os *drivers* de dispositivo e notou-se que uma das principais características de *driver* de dispositivo é isolar as particularidades de um *hardware* no *kernel*, disponibilizando uma interface com maior abstração.

Palavras-chave: *drivers* de dispositivo. Linux. Sistemas embarcados.

Abstract

This work seeks to study and develop device drivers, developed for hardware modules in the context of embedded systems, that uses Linux operating system. A bibliography analysis was done, to discover what is a device driver, when to use them and the difficulties of developing device drivers. In this work, device drivers were developed, one to use displays that are compatible with the LCD Hitachi HD 44780 controller, and other with the RFID RC522 board, the device drivers were tested and correctly interfaced the hardware modules during the tests, one of the main characteristics of device drivers is to isolate the hardware implementation in the kernel, providing a more abstract form to interact with the hardware in the user space.

Key-words: device drivers. Linux. Embedded systems.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ARM	Advanced RISC Machine (Máquina RISC avançada)
CI	Circuito Integrado
FIFO	First in, First out (Primeira a entrar, primeiro a sair)
GPIO	General Purpose Input/Output (Entrada/Saída de uso geral)
I^2C	Inter-Integrated Circuit (Circuito Inter-Integrado)
IDE	Integrated Development Environment (Ambiente de desenvolvimento integrado)
IO	Input/Output (Entrada/Saída)
IoT	Internet of things (Internet das Coisas)
LCD	Liquid Crystal Display (*Display* de cristal líquido)
MISO	Master-Input Slave-Output (Entrada no Mestre, saída no escravo)
MOSI	Master-Output Slave-Input (Saída no Mestre, entrada no escravo)
OS	Operating System (Sistema Operacional)
RFID	Radio-frequency identification (Identificação por rádio-frequência)
RISC	Reduced Instruction Set Computer (Computador de conjunto reduzido de instruções)
SOC	System-On-a-Chip (Sistema em um chip),
SPI	Serial Peripheral Interface (Interface Periférica Serial)
SSH	Secure Shell (Shell seguro)
TCC	Trabalho de conclusão de curso
TV	Televisão
UART	Universal Asynchronous Receiver / Transmitter (Receptor/Transmissor Assíncrono Universal)
UID	Unique Identifier (Identificador Único)
USB	Universal Serial Bus (Barramento Serial Universal)

Sumário

1 Introdução

Esse documento trata do estudo e da criação de interfaces para dispositivos de *hardware* conhecidos como *drivers* de dispositivos, com foco no desenvolvimento para sistemas operacionais Linux. Assim, a partir da análise da bibliografia existente, busca-se delinear os casos de uso para os *drivers* de dispositivos, e o desenvolvimento de *drivers*. São elaborados dois *drivers* de dispositivos de módulos de *hardware* comumente utilizados, buscando entender dificuldades de desenvolvimento, e para que os *drivers* fiquem disponíveis para uso. Esse capítulo busca introduzir conceitos básicos de *drivers* de dispositivos e sua relação com sistemas operacionais Linux e seu *kernel*.

1.1 Contextualização

Sistemas embarcados, ou sistemas computadorizados dedicados a finalidades específicas, encontram amplo uso ao longo de diversas indústrias, desde saúde e transportes até educação. Tem se tornado comum a utilização de microcomputadores com sistemas operacionais para sistemas embarcados, e entre as opções de sistemas operacionais o mais utilizado é o sistema operacional Linux (SIMMONDS; PURDIE, 2015, p. xiii)(MADIEU, 2022, p. 3). Para realizar a integração entre o *hardware* e o sistema operacional, é necessário a criação de *drivers* de dispositivos, que no *kernel* Linux geralmente são implementados como módulos carregáveis do *kernel*, que são módulos que podem ser carregados e descarregados em tempo de execução (MADIEU, 2022, p. 26). Esses *drivers* podem expor as funcionalidades do *hardware* como uma abstração para o espaço do usuário, assim, ocultando detalhes do funcionamento do *hardware* e facilitando o desenvolvimento no espaço de usuário (CORBET et al., 2005, p. 3), no sistema operacional Linux, esses *drivers* são disponibilizados como arquivos, e a interação com esse arquivo abstrai as interações com o *hardware* em si, isolando os detalhes da interação com o *hardware* para dentro do *kernel*. Porém, módulos do *kernel* são significativamente mais complexos de se desenvolver, há restrições nas bibliotecas que podem ser utilizadas, na linguagem C, as bibliotecas padrão da linguagem não ficam disponíveis, sendo utilizáveis apenas funções disponibilizadas pelo *kernel*.

1.2 Proposta do tema

Como indicado por Li et al. (2019), sistemas operacionais usualmente possuem *drivers* de dispositivos para integrar periféricos e módulos de *hardware*. Isso também é verdade para Linux embarcado, visto que muitos fornecedores de *hardware* distribuem

drivers de dispositivos para seus produtos, e alguns até podem vir integrados ao *kernel* Linux (SIMMONDS; PURDIE, 2015). Assim, o desenvolvimento de *drivers* de dispositivos é de suma importância para sistemas operacionais embarcados, visto que fornecem a interface entre o *hardware* utilizado e o espaço do usuário. Baseado nesse fato, esse trabalho busca entender as dificuldades, vantagens e necessidades do desenvolvimento de *drivers* de dispostivos.

1.3 Objetivos

1.3.1 Objetivos gerais

Esse trabalho tem como objetivo o desenvolvimento e a análise do desenvolvimento de módulos do *kernel* Linux que implementam *drivers* de dispositivos no contexto de sistemas embarcados.

1.3.2 Objetivos específicos

Levando em consideração esse objetivo, foram traçados os seguintes objetivos específicos:

- Entender o papel dos *drivers* de dispositivos em sistemas embarcados.
- Desenvolver um *driver* de dispositivo para o *display* LCD.
- Desenvolver um *driver* de dispositivo para o módulo RFID RC522.

1.4 Estrutura do documento

O Capítulo 2 apresenta uma revisão bibliográfica do tema, visando estabelecer conceitos utilizados durante o trabalho, e oferecendo embasamento para o desenvolvimento dos *drivers* de dispositivos com referências de trabalhos anteriores. O Capítulo 3 detalha o *hardware* que foi utilizado durante o projeto, o ambiente de desenvolvimento utilizado para o desenvolvimento, e trata sobre os *drivers* de dispositivos desenvolvidos, detalhando seu funcionamento. Em sequência, o Capítulo 4 traz os resultados obtidos durante o desenvolvimento, e o Capítulo 5 conclui o texto.

2 Revisão bibliográfica

2.1 Linux Embarcado

Um sistema embarcado pode ser definido como um sistema computadorizado criado para uma finalidade específica (WHITE, 2012, p. 1), de geladeiras à carros autônomos, sistemas embarcados se tornam cada vez mais presente no dia-a-dia das pessoas. À medida que a complexidade dessas aplicações aumenta, também aumenta a necessidade de poder computacional destes sistemas, a ponto de algum momento se tornar útil ou até necessário a utilização de sistemas operacionais robustos em processadores de uso geral (SIMMONDS; PURDIE, 2015, p. xiii). Uma escolha muito comum são os sistemas operacionais com *kernel* Linux, devido ao fato dele ser um *software* livre e bastante flexível, visto que o acesso ao código-fonte permite edita-lo, além de haver uma comunidade ativa em torno de melhorar e manter o *kernel*. Além disso, o Linux suporta diversas arquiteturas de processadores, inclusive muitas usadas por processadores comumente utilizados para sistemas embarcados, como a arquitetura ARM (SIMMONDS; PURDIE, 2015, p. 2).

Um sistema operacional é dotado de diversas ferramentas diferentes, entre elas está o *kernel*. O *kernel* é responsável por criar a interface entre o *hardware* de um computador e o *software*, além de gerenciar memória, sistemas de arquivos, processos, *drivers* e conexões de rede (REDHAT, 2019). Uma das funcionalidades de sistemas operacionais Linux é a possibilidade de estender o *kernel* em tempo de execução, os chamados módulos carregáveis, ou apenas módulos (CORBET et al., 2005, p. 4-5).

2.2 *Drivers* de dispositivos

Módulos podem implementar diversas funções, desde a geração de números pseudo-aleatórios até *drivers* de dispositivos, cuja principal função é realizar a interface entre um *hardware* e os processos do sistema operacional. Assim, no espaço do usuário se utiliza os *drivers* de dispositivos de forma transparente, sem conhecer os detalhes de funcionamento do *hardware* e com uma interface mais acessível. Como o sistema operacional Linux é baseado em UNIX, os *drivers* de dispositivos no espaço do usuário são utilizados como arquivos visto a similaridade de ambos (SILBERSCHATZ; GAGNE; GALVIN, 2018, p. 72).

Módulos do *kernel* que implementam dispositivos podem ser caracterizados em 3 tipos: de caractere, de bloco ou de rede. Os dispositivos de caractere são definidos por serem acessados *byte* a *byte*, funcionando como arquivos de *stream* de *bytes*. Esses dispositivos implementam periféricos no geral e portas seriais. Os dispositivos de bloco, como o

próprio nome indica, transferem dados em bloco, tendendo a serem mais rápidos que os de caractere, e estão associados a *hardware* de armazenamento de arquivos. Dispositivos de rede criam interfaces para receber e enviar pacotes de dados entre *hosts* diferentes. Esse documento trata principalmente de dispositivos de caractere, visto que para dispositivos de *hardware* comumente utilizados junto com microcontroladores, como sensores, *displays* e módulos de comunicação, esse tipo de dispositivo é o mais apropriado (MADIEU, 2022, p. 141-142)(CORBET et al., 2005, p. 6-7).

Existem diversas vantagens em se utilizar um módulo do *kernel* ao invés de um programa no espaço de usuário, tais como um maior acesso a interrupções, um acesso mais veloz às portas *IO*, e um tempo de resposta menor para módulos do *kernel* (CORBET et al., 2005, p. 38-39), além disso, por questão de eficiência e segurança, programas do espaço do usuário não podem controlar diretamente dispositivos *IO* (SILBERSCHATZ; GAGNE; GALVIN, 2018, p. 56). Dentre as desvantagens, podemos citar a limitação no uso de bibliotecas da linguagem C padrão, a impossibilidade de usar a maioria das outras linguagens de programação, há suporte para Rust, porém a maior parte do desenvolvimento e documentação disponível é relacionado as linguagens C e Assembly, uma maior dificuldade de acesso a documentações e exemplos e um *debug* significativamente mais complexo (CORBET et al., 2005, p. 37-38). Alterações errôneas no *kernel* facilmente causam problemas que interrompem o sistema operacional, em comparação, acessos indevidos de memória no espaço do usuário tendem a causar um erro de "*Segmention Fault*", porém, o desenvolvimento segue normalmente, acessos indevidos de memória no *kernel* facilmente interrompem o sistema operacional, sendo necessário reiniciar o dispositivo. Além disso, há menos ferramentas de *debug* e teste para um módulo do kernel em comparação com um programa no espaço de usuário.

Sendo assim, é importante ponderar a criação de módulos do *kernel*. Em muitos casos é mais vantajoso o uso de um *driver* de dispositivo, como por exemplo para *drivers* para protocolo SPI e I^2C (MADIEU, 2022, p.271, 303), mas existem casos em que um programa no espaço de usuário pode ter maior eficiência do que um módulo do *kernel*, Shukla et al. (2004) indica que ao se comparar servidores *web* no espaço do usuário e no *kernel*, em situações que é retornado conteúdo estático, a aplicação implementada no *kernel* parecia ser mais rápida, já quando é necessário o retorno de conteúdo dinâmico, resultado de um processamento do servidor, a aplicação no espaço do usuário parecia ser mais eficiente.

2.3 Árvore de dispositivos (*device-tree*)

Uma árvore de dispositivos é uma estrutura de dados/linguagem utilizada para descrever *hardware*, de forma a abstrair informações específicas do *hardware*, em um pa-

drão legível por software. Apesar de ser utilizado no *kernel Linux*, a especificação da árvore de dispositivos é um projeto a parte, que também é utilizado em outros projetos (DEVICETREE. . . , s.d.).

Nas versões mais recentes do *kernel Linux*, a representação de dispositivos é feita por um árvores de dispositivos. Isso permite um maior desacoplamento da configuração do *hardware* e o *driver* de dispositivo no *kernel Linux*, permitindo especificar diferenças no *hardware* por meio da árvore de dispositivos, e utilizar uma abstração no *driver* de dispositivo. O *kernel Linux* utiliza a árvore de dispositivos, para identificar a plataforma, durante o processo de inicialização, para configurações em tempo de execução, visto que a árvore de dispositivos é uma das únicas pontes entre o *firmware* e o *kernel*, e para a população de dispositivos, permitindo ao invés de se utilizar uma lista de dispositivos estáticas no código é possível se obter as informações em tempo de execução a partir da árvore de dispositivos (LINUX. . . , s.d.).

2.3.1 Estrutura de uma árvore de dispositivos

A estrutura de uma árvore de dispositivos pode ser vista no exemplo indicado no apêndice A.1, cada dispositivo de *hardware*, incluindo a própria placa, é representando por um nó na árvore, dispositivos filhos na árvore indicam dependência em relação ao nó pai, como por exemplo, um dispositivo I^2C *slave* será dependente do dispositivo I^2C da própria placa. Em cada nó, é possível listar propriedades específicas daquele *hardware*, como por exemplo a velocidade do *clock*, além dos dispositivos filhos.

Além da própria árvore de dispositivos, é possível criar *overlays*, que são fragmentos de árvores que são utilizados para alterar uma árvore em tempo de execução. O apêndice A.2 mostra um *overlay* que poderia ser inserido na árvore de exemplo, a estrutura é similar a de uma árvore de dispositivos, porém são listados fragmentos, onde cada fragmento tem um nó alvo, e o *overlay* que será aplicado. É possível alterar tanto propriedades como nós filhos do nó selecionado, o exemplo mostra a inserção de um dispositivo I^2C *slave* no dispositivo I^2C da placa (MADIEU, 2022, p. 186-187).

2.4 Desenvolvimento de *drivers* de dispositivo do Linux

2.4.1 Pré-requisitos

Para desenvolver módulos carregáveis para Linux, a primeira necessidade é obter o código-fonte do *kernel* (CORBET et al., 2005, p. 15-16), disponível no *site* oficial do *kernel* Linux (THE. . . , s.d.). Dependendo da distribuição do sistema operacional Linux sendo utilizado, podem haver variações (KERNEL. . . , s.d.).

2.4.2 Estrutura mínima para um *driver* de dispositivo

Módulos do *kernel* possuem macros que indicam funcionamentos específicos. Existem duas principais, uma executada na instalação do módulo no *kernel*, e outra na remoção do módulo, respectivamente, *module_init()* e *module_exit()* (DRIVER..., s.d.). Além das macros, ao se instalar o módulo do *kernel* este recebe acesso a funcionalidades de outros módulos que foram expostas como públicas, sendo possível inclusive criar um módulo para expor funcionalidades e reutilizá-las em outros módulos.

Apesar dos módulos serem desenvolvidos em linguagem C, as bibliotecas-padrão da linguagem C não são utilizadas pelo *kernel*, visto que este roda independentemente de tais bibliotecas, porém, funcionalidades comuns das bibliotecas da linguagem C podem ter outra funcionalidade similar no *kernel*. Por exemplo, a função *printk()* é similar a função *printf()* da biblioteca-padrão de entrada e saída de dados na linguagem C (CORBET et al., 2005, p. 16-17).

Criar um módulo carregável não é o suficiente para criar um dispositivo, envolvendo outras etapas importantes. A primeira é relacionada à identificação do dispositivo através de dois números, um número *major* e um *minor*, que podem ser feitos por registro e por alocação. No primeiro caso, é necessário saber previamente o número *major* que aquele dispositivo ocupará, e por esse motivo é muito mais comum se utilizar a alocação, onde o *kernel* registra um número *major* para o dispositivo, evitando conflitos.

Após o número de identificação ter sido criado, é necessário criar o dispositivo, reservando uma estrutura que o representa (uma *struct* em C), informar as operações de arquivo que vão ser realizadas pelo dispositivo, e por último adicionar o dispositivo ao *kernel* (CORBET et al., 2005, p. 42-56). A partir desse momento, o dispositivo já foi criado e está exposto ao *kernel*.

As funcionalidades do seu dispositivo são realizadas por suas operações de arquivo. Cada dispositivo criado tem um arquivo referente a ele no *kernel*, podendo ser aberto, fechado, lido, escrito etc. Além das operações tradicionais de arquivo também é possível utilizar a opção de *Input/Output Control (ioctl)*, que permite estender a lista de comandos disponíveis em um arquivo (MADIEU, 2022, p. 169), quando há alguma dessas interações com o arquivo, o seu dispositivo executará o código referente a aquela operação específica que foi indicada na inicialização do dispositivo. As operações *ioctl* são diferenciadas a partir de um número específico, a biblioteca *linux/types.h* guarda os macros utilizados para criar esses números, esses macros são, *_IO* para uma operação que não há troca de dados, *_IOW* para operação onde o espaço de usuário escreve dados para o kernel, *_IOR* onde o espaço do usuário recebe dados do *kernel* e *_IOWR* para dados em ambas as direções.

A última etapa é fazer com que o dispositivo possa ser acessado no espaço do usuá-

rio. Para isso, é primeiro necessário criar uma classe que representa o tipo de dispositivo sendo registrado (DEVICE..., s.d.a), e criar o dispositivo no espaço do usuário (DEVICE..., s.d.b)(MADIEU, 2022, p. 147-152). Durante a criação do dispositivo, diversas estruturas são alocadas e registradas no *kernel*, e também são realizadas outras operações, como o registro de uso de GPIOs (*General-Purpose Input and Output*, ou entradas e saídas de uso geral). Assim, é necessário desalocar e remover os registros realizados quando o dispositivo não for mais utilizado. O diagrama de fluxo da Figura 1 mostra esse processo de criação de um *driver* de dispositivo.

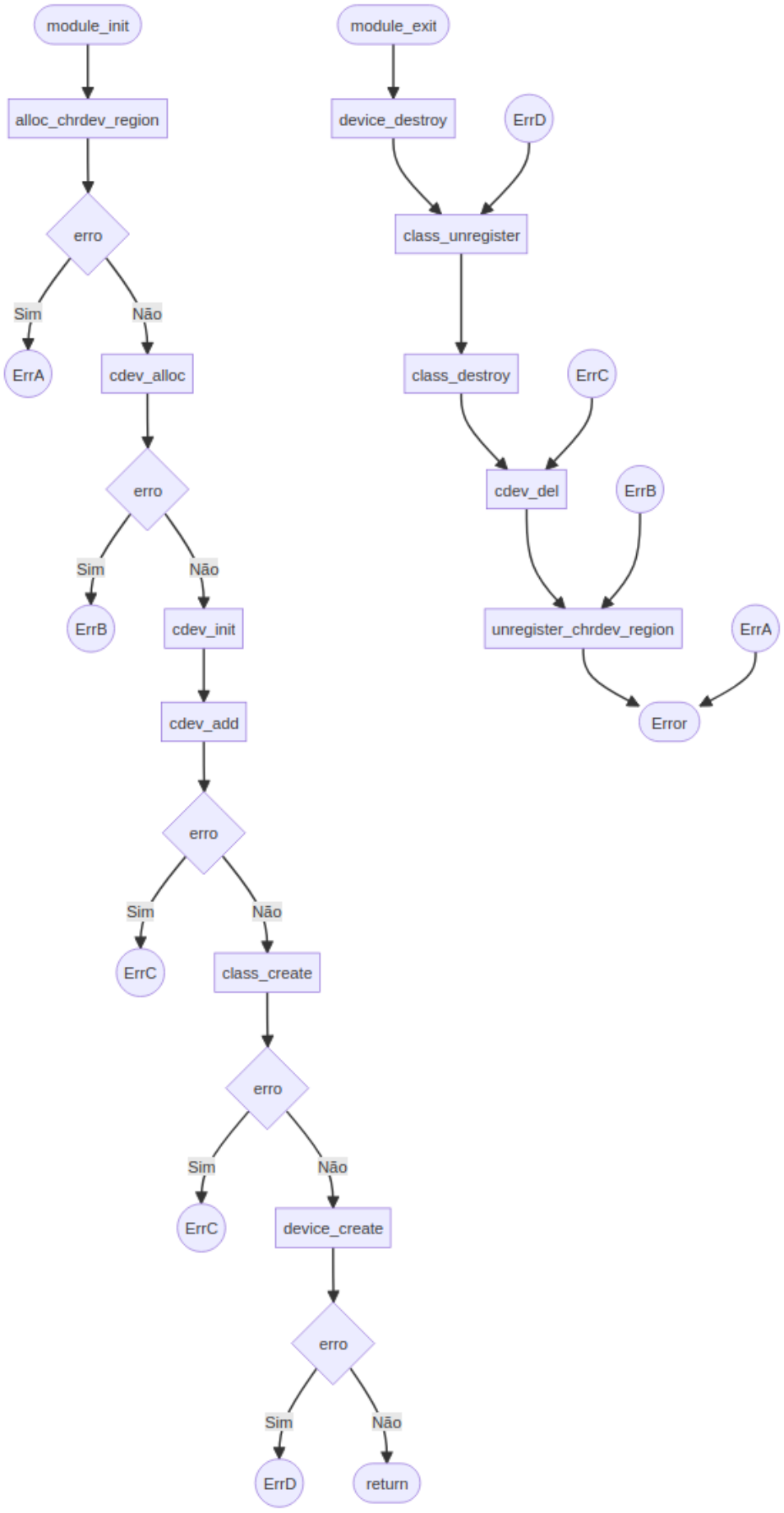

Figura 1 – Diagrama de fluxo de inicialização de um módulo do *kernel* Linux.

2.4.3 Estruturação da lógica do *driver* de dispositivo

O processo de criação de um dispositivo descrito anteriormente deve ser executado na instalação do módulo, para que o dispositivo se torne acessível para o usuário. Da mesma forma, durante a remoção do módulo é necessário desalocar e remover os registros realizados no *kernel*. Durante a criação do módulo, são atribuídas operações de arquivos, e a maior parte da lógica de funcionamento dos *drivers* é feita nessas operações, permitindo ao espaço de usuário interagir com o módulo. Desta forma, toda a lógica da interação do usuário com o *hardware* é interfaceada pelas operações de arquivo (MADIEU, 2022, p. 143)(CORBET et al., 2005, p. 49). Por exemplo, um *driver* hipotético para um *array* de 8 LEDs poderia utilizar a operação de leitura do arquivo para se obter o estado atual das portas de saída que controlam os LEDs, e a operação de escrita para alterar o estado das mesmas portas. Neste exemplo, a única parte da lógica que não estaria relacionada às operações de arquivo é a lógica do registros dos GPIOs das portas, que deveria ser feito junto com a criação do dispositivo durante a instalação do módulo.

2.4.4 Interação com a árvore de dispositivos

Quando se está desenvolvendo um *driver* de dispositivo para um dispositivo da árvore de dispositivos, é necessário também, inicializar esse dispositivo, o método de inicialização pode variar de acordo com o tipo de dispositivo, no caso desse trabalho será tratado a inicialização de um dispositivo *SPI*, porém, a implementação para dispositivos I^2C é similar. Na árvore de dispositivos é necessário que o dispositivo que você está implementando seja um nó filho de um dispositivo *SPI* da placa.

Para registrar um dispositivo *SPI* deve-se, a partir do método *module_init()* do módulo do *kernel*, chamar a função *spi_register_driver*, para essa função é necessário fornecer as informações do próprio *driver*, entre essas informações, deve-se mandar uma tabela de compatibilidade que será utilizada para identificar a presença de um dispositivo compatível na árvore de dispositivos, além disso, também é preciso enviar duas funções, a primeira é a *probe*, essa função é chamada quando um dispositivo compatível é inserido na árvore de dispositivos, ou quando já há um dispositivo compatível e o módulo é inserido, a outra função é a função *remove*, que é chamada quando o dispositivo é removido da árvore ou quando o módulo do*kernel* vai ser removido. (MADIEU, 2022, p. 295-300)

Assim, é possível registrar o módulo do *kernel* previamente, porém apenas ativa-lo na presença de um dispositivo compatível, ainda, também permite reaproveitar o mesmo módulo do *kernel* para diferentes dispositivos, caso seja possível. Então nos casos de dispositivo da árvore de dispositivos é interessante realizar a inicialização do *driver* de carácter na função *probe* ao invés da função *module_init()*, visto que se fosse mantido no *module_init()*, as funções de interação com o módulo ficaram abertas ao espaço de usuário mesmo na ausência do dispositivo físico.

2.5 Trabalhos similares

A bibliografia mostra diversos exemplos de uso de *drivers* de dispositivos para interface de dispositivos de *hardware*. Em Nguyen (2014), o autor desenvolve um *driver* de dispositivo para as GPIOs de uma placa Raspberry Pi 2B. De formar similar, Patoliya et al. (2021) implementa um sistema de IoT (*Internet of things*, ou Internet das Coisas), onde um sensor de presença tem como interface um *driver* de dispositivo. Já em Ansari e Kaur (2017), é desenvolvido um *driver* de dispositivo para um emissor e receptor de luz infravermelha. Outro estudo importante é o de Sang-Pil Moon et al. (2003), que desenvolve um sistema operacional para uma TV, utilizando um *driver* de dispositivo para a comunicação utilizando o protocolo I^2C.

Estes exemplos são interessantes para a continuação do presente trabalho de conclusão de curso, visto que lidam com interrupções, utilizam placas da família Raspberry Pi e sistemas operacionais parecidos, e reaproveitam módulos de comunicação como os do protocolo I^2C.

3 Materiais e métodos

Neste capítulo, são apresentadas as principais decisões de projeto do trabalho presente. Durante o processo de escolha dos materiais utilizados, o principal critério empregado foi o de praticidade, visto que durante o trabalho não são feitas avaliações em relação à otimização de desempenho dos projetos desenvolvidos.

3.1 *Hardware*

3.1.1 Raspberry Pi 3B

A Raspberry Pi é um microcomputador de baixo custo, e neste projeto foi utilizado a plataforma em seu modelo 3B revisão 2.0. Essa versão da placa Raspberry Pi utiliza um SOC BCM2837, que inclui uma CPU ARM e uma GPU. A Raspberry Pi possui diversas formas de conexão, como USB, WiFi, Ethernet e HDMI, dentre outros. Além disso, a placa possui 26 pinos de GPIO (PI, s.d.). A Figura 2 mostra esses pinos, entre os quais alguns são adaptados para melhor utilização de certos protocolos de comunicação como I^2C e UART. A placa é alimentada por meio de uma entrada micro USB.

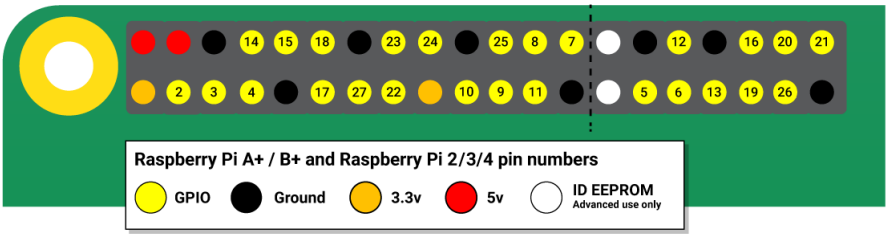

Figura 2 – GPIOs Raspberry PI 3B, obtido em (PI, s.d.)

3.1.2 *Display* LCD Hitachi HD 44780 16x2

Inicialmente, esse *display* foi escolhido devido à existência de um *driver* de dispositivo para o *display*, porém com funcionalidade limitada, sendo assim o aprimoramento desse *driver* de dispositivo seria uma boa introdução para o desenvolvimento de *drivers* de dispositivos. Além disso, o uso de *displays* LCDs em protótipos é bastante comum, visto que é um meio simples de mostrar dados a usuários, porém mais completo que um *display* de 7 segmentos. Os *displays* LCDs mais comuns para aplicações embarcadas são os que utilizam controlador Hitachi HD 44780, ou controlador compatível, o que enriquece

a disponibilização do *driver* de dispositivo, visto que o *driver* deve funcionar sem muitas mudanças independentemente do fabricante, desde que seja compatível com o controlador Hitachi HD 44780.

A pinagem do *display* pode ser vista na Tabela 1. A alimentação do *display* é feita pelas portas V_{dd} e V_{ss}, com tensão nominal de alimentação de 5V. O pino V_0 é um pino de controle do contraste, sendo comum utilizar um potenciômetro de $10K\Omega$ com a mesma alimentação de 5V para ajustar a tensão no mesmo. O *display* tem modos de leitura e de escrita, definidos pela porta RW como nível lógico 0 para escrita e 1 para leitura. Porém, o modo de leitura não foi utilizado no *driver*, logo esse pino foi fixado com 0. Já o pino RS indica se os bits enviados são dados (1) ou uma configuração do *display* (0) (EONE, 1997; ILETT, 2005).

Tabela 1 – Pinagem do *display* LCD Hitachi HD 44780

Pino	**Sigla**	**Nome**
1	V_{ss}	**Aterramento**
2	V_{dd}	**Alimentação**
3	V_0	**Alimentação de contraste**
4	RS	**Seleção de registrador**
5	RW	**Leitura/Escrita**
6	E	***Enable***
7	D_0	**Bit de dados 0**
8	D_1	**Bit de dados 1**
9	D_2	**Bit de dados 2**
10	D_3	**Bit de dados 3**
11	D_4	**Bit de dados 4**
12	D_5	**Bit de dados 5**
13	D_6	**Bit de dados 6**
14	D_7	**Bit de dados 7**
15	A	**Alimentação da luz de fundo**
16	K	**Aterramento da luz de fundo**

3.1.3 RFID-RC522

O módulo RFID-RC522 utiliza o CI MFRC522 para comunicação sem contato compatível com o padrão ISO14443. Ele pode ser configurado para comunicação por SPI, UART ou I^2C (NXP, 2016). A escolha desse módulo foi feita devido ao fato da utilização do protocolo SPI, que como visto anteriormente, quando se utiliza esse protocolo é recomendado o desenvolvimento de um *device driver*.

3.1.3.1 Pinagem

A Tabela 2 indica as entradas da placa RC522 e suas funcionalidades utilizando SPI como protocolo de comunicação.

Tabela 2 – Pinagem da placa RC522

Pino	**SPI**	**Nome**
SDA	NSS	Utilizado na seleção de *slave* na comunicação SPI
SCK	SCK	*Clock* serial
MOSI	MOSI	Saída do dispositivo *master*, entrada no dispositivo *slave*
MISO	MISO	Entrada do dispositivo *master*, saída no dispositivo *slave*
IRQ	-	Pino que gera interrupções
GND	-	Pino de aterramento
RST	-	Pino de reset, é ativado com nível lógico 0
3.3V	-	Pino de alimentação com tensão nominal de 3.3V

3.1.3.2 Comunicação entre o dispositivo *host* e a placa

Como citado anteriormente, há mais de uma opção de protocolo de comunicação entre a placa e o dispositivo *host*, neste trabalho foi utilizado o protocolo *SPI*. A comunicação com a placa tem duas principais funções, a leitura e a escrita nos registradores das placa, a partir dessas operações todas outras são realizadas, em toda comunicação entre o *host* e a placa um *byte* é escrito na linha *MISO* e outro é lido na linha *MOSI* pela placa.

Cada registrador da placa tem um endereço associado a ele, a lista completa é acessível no *datasheet* em NXP (2016, p. 36-37), para interagir com um registrador específico, na hora da comunicação, deve-se enviar um *byte* de endereçamento. Esse *byte* tem um formato padrão, a comunicação é feita enviando primeiro o *MSB*, o *MSB* é o bit utilizado para indicar se é uma escrita, valor 0, ou leitura, valor 1, os próximos 6 bits devem ser o endereço do registrador de interesse, e o último bit deve ser sempre 0.

Para realizar a leitura, deve-se enviar o *byte* de endereço pela comunicação *SPI*, o *byte* recebido nessa primeira troca pode ser descartado, após enviar o *byte* de endereço, na próxima troca deve-se enviar um *byte* zerado e então o *byte* recebido na linha *MISO* é o dado contido no registrador especificado, é possível enviar outro *byte* de endereço, ao invés de um *byte* zerado, em sequência para realizar múltiplas leituras, após todas as leituras serem realizadas envia-se o *byte* zero. No caso da escrita, todos os *byte*s da linha *MISO* podem ser descartados, primeiro envia-se o *byte* de endereçamento, então, envia-se o *byte* a ser escrito no registrador. É possível escrever múltiplas vezes no mesmo registrador, para isso basta continuar enviando *bytes* de dados em sequência (NXP, 2016, p. 10-11).

3.1.3.3 Executando um comando

A placa possui um *buffer* interno de de 64 *bytes*, associado a ele está um registrador, ao escrever nesse registrador, um *byte* é inserido no *buffer*, e ao ler no registrador, um *byte* é recuperado do *buffer*. A maioria dos comandos realizados na placa consistem em escrever uma série de dados no *buffer* e ativar o comando específico a ser executado, enviando o

byte relacionado ao comando ao registrador de comandos, os principais comandos são relacionados a se comunicar com as *tags RFID* (NXP, 2016, p. 70).

As *tags RFID* utilizam o padrão ISO14443, e tem um mecanismo de concorrência para o caso em que há múltiplas *tags* se comunicando com a placa, além disso, cada *tag* tem uma memória interna que armazenam os dados. Quando uma *tag* é energizada, ela entra em um estado de espera, a placa então pode enviar um comando de requisição de *tags*, o que faz as *tags* próximas entrarem em estado de prontidão, quando a *tag* está no estado de prontidão, a placa pode realizar o protocolo anti-colisão para uma *tag* específica, processo que resulta na recepção de um *UID* específico referente a *tag*, e com esse *UID* a placa pode selecionar essa *tag* (ISO/IEC. . . , 2018). Com a *tag* selecionado, a placa pode fazer escritas e leituras na memória da *tag*, a memória de uma *tag* pode variar de acordo com o fabricante e modelo, os modelos mais comuns possuem 16 setores, cada setor com 4 blocos, de 16 bits, o último bloco do setor é onde contém as informações de autenticação do setor, é possível apenas escrever nesses blocos, após autenticação é possível ler e escrever nos outros blocos do setor.

3.2 Ambiente de desenvolvimento

No desenvolvimento para embarcados é comum a necessidade de um computador externo ao sistema embarcado para configuração do mesmo, geralmente se referindo ao sistema embarcado como dispositivo alvo, e o computador como dispositivo *host*.

Como indicado anteriormente, esse trabalho utiliza o micro-computador Raspberry PI 3B, devido a isso, foi utilizado o sistema operacional Raspberry Pi OS, que é um sistema Linux baseado na distribuição Debian. Apesar de ser o mais, comum é possível utilizar outras distribuições Linux no Raspberry Pi, porém o Raspberry Pi OS é otimizado para o *hardware* das placas Raspberry Pi, também tendo uma instalação muito facilitada visto que a empresa responsável disponibiliza ferramentas e documentações para configuração da placa e do sistema operacional. Porém, a única restrição para replicar este trabalho é utilizar um sistema operacional Linux, logo, é possível utilizar outras distribuições de Linux.

Para o dispositivo *host*, as únicas restrições de *hardware* são as necessárias para o uso das ferramentas de desenvolvimento. Não há restrições de utilização de sistema operacional, visto que os *drivers* de dispositivos são executados apenas no dispositivo alvo.

Para estabelecer a comunicação entre o dispositivo *host* e dispositivo alvo, há diferentes formas, como conexão via Ethernet, WiFi, UART, e até transferindo dados por meio de um *pendrive*. Neste projeto, foi utilizado a comunicação por WiFi através do protocolo SSH (PI, 2019). Os tempos de compilação dos códigos não foram longos o

suficiente para atrasar o desenvolvimento, de forma que o acesso remoto foi usado para a compilação local dos módulos.

A IDE Visual Studio Code (MICROSOFT, 2016) foi utilizada como editor de texto para os *drivers* de dispositivo, principalmente devido a seus *plugins* que facilitam o desenvolvimento remoto por SSH. Assim, o processamento da IDE, que pode ser um fator limitante para o dispositivo alvo, é feito no dispositivo *host*, sendo transferidas apenas alterações nos arquivos e comandos de terminal para um servidor VScode que é instalado no dispositivo alvo (AJANI, 2019), como mostrado na Figura 3.

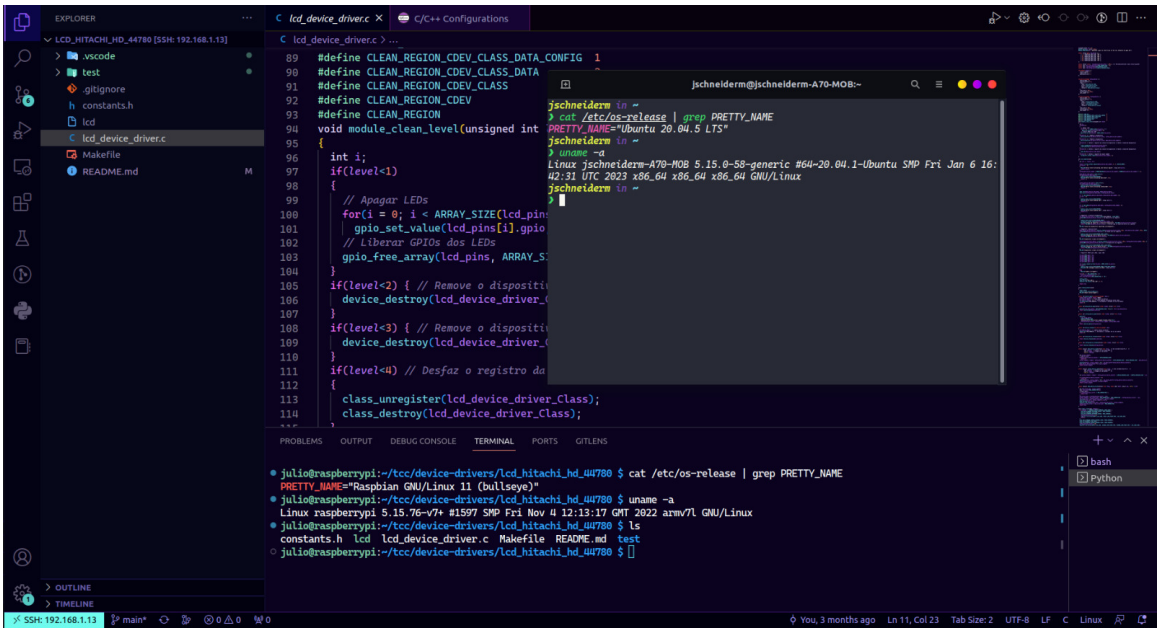

Figura 3 – Ambiente de desenvolvimento

3.3 *Driver* de dispositivo para *display* LCD Hitachi

A partir do *driver* de dispositivo desenvolvido por (GARCIA, 2021), foram implementadas melhorias, visto que as funcionalidades do mesmo eram limitadas. No *driver* já existente, a partir da escrita no arquivo */dev/lcd_device_driver*, o *driver* de dispositivo só permitia a escrita na primeira linha do *display* e antes da escrita o *display* era sempre limpo, e as opções de configurações não eram acessíveis no espaço de usuário. Buscou-se estender a possibilidade de escrita, podendo-se escrever em ambas as linhas do *display*, além de disponibilizar várias opções de configuração, como controle do cursor do *display*, limpeza das linhas do *display* desacoplada da operação de escrita e configuração do modo uma ou duas linhas.

A inicialização do *driver* de dispositivo é similar a indicada na Subseção 2.4.2, porém, como há mais de um *driver* de dispositivo deve-se seguir o mesmo processo de inicialização para ambos os dispositivos, exceto pela etapa de criação da classe dos dispo-

sitivos, que caso seja útil, pode ser a mesma para ambos os dispositivos, como foi o caso neste trabalho. Além disso, nesse módulo era necessário o registro de pinos GPIO após o processo de criação do dispositivo mostrado na Subseção previamente citada, mas ainda dentro da execução da função *module_init()*. Também foram adicionados parâmetros opcionais para a instalação do módulo: os parâmetros *EN, RS, D4, D5, D6 e D7* podem receber números de portas GPIO para alterar quais portas seriam utilizadas pelo *display*. A Figura 4 apresenta o diagrama de fluxo de inicialização do módulo do *display* LCD Hitachi HD 44780.

A principal diferença dos dois dispositivos são as operações de arquivos utilizadas, que vão prover a diferença lógica dos dispositivos. Em geral, ao se escrever no arquivo */dev/lcd_driver/config*, o *bit RS* era setado como 0 e a partir do que era escrito no arquivo eram enviadas operações de configuração para o *display*. Essas funções de configuração estão indicadas na Tabela 3, para realizar cada função bastava escrever no arquivo o caractere indicado na Tabela. Além disso, foi criado o arquivo */dev/lcd_device_driver/data* para se enviar dados para serem escritos na tela, e ao se escrever nesse arquivo o *bit RS* era setado como 1, e o que foi escrito no arquivo era enviado ao *display*. A escrita no arquivo é feita de forma a respeitar o padrão do controlador Hitachi HD 44780, assim, mesmo que ao utilizar um *display* 16x2, a escrita é feita contemplando duas linhas de 40 caracteres. Ainda, caso a escrita passe o caractere do índice 79, ela é realizada até esse caractere, o restante da *string* é descartada e o cursor retorna para a primeira posição. Logo, foram implementados dois *drivers* de dispositivo diferentes para o mesmo *hardware* em um único módulo do *kernel*, porém com funções diferentes.

Tabela 3 – Tabela de comandos do *driver* de dispositivo de configuração do *display* LCD Hitachi HD 44780

Comando	**Caractere**	**Descrição**
Modo uma linha	1	Modo em que apenas a linha de cima do *display* é utilizada
Modo duas linhas	2	Modo em que ambas as linhas do *display* são utilizadas
Cursor visível	3	Torna a posição atual do cursor visível, piscando no *display*
Cursor não visível	4	Desabilita o cursor visível
Limpar *display*	5	Remove qualquer carácter escrito no *display*
Retornar cursor	6	Retorna o cursor para a primeira posição do *display*
Ir para segunda linha	7	Move o cursor para a primeira posição da segunda linha do *display*
Limpar primeira linha	8	Limpa a primeira linha (0 - 15) e posiciona o cursor no começo da primeira linha
Limpar segunda linha	9	Limpa a segunda linha (40 - 55) e posiciona o cursor no começo da segunda linha

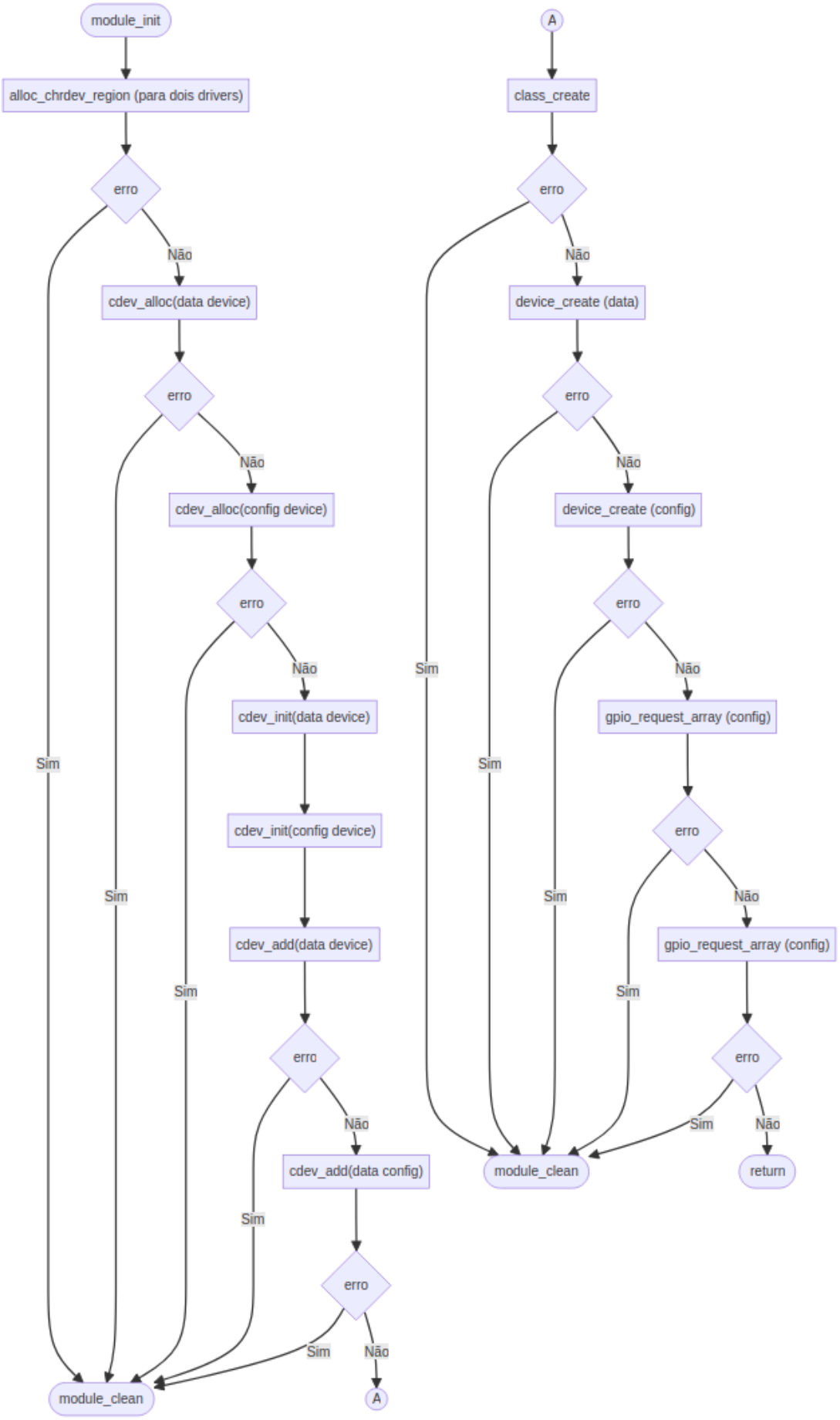

Figura 4 – Diagrama de fluxo de inicialização do módulo do *display* LCD Hitachi HD 44780.

3.4 *Driver* de dispositivo para o dispositivo RC522

Foi criado um novo *driver* de dispositivo desenvolvido para módulo RC522, esse *driver* era acessível no espaço de usuário a partir da interação com o arquivo */dev/rfid_rc522_driver*, todas as operações eram executadas a partir desse arquivo, usando operações *ioctl*.

A inicialização é relativamente similar realizada no *display LCD Hitashi*, porém, como comentando na subseção 2.4.4, na função *module_init* foi implementado apenas a inicialização do dispositivo SPI, cabendo a função *probe* inicializar o *driver* de carácter, além disso, inicializando também um pino GPIO como saída para controlar o *reset* da placa, o fluxo de inicialização pode ser visto na Figura 5. Diferentemente do *driver* de carácter anterior, as operações no módulo do kernel não estavam relacionadas as operações de arquivo de escrita e leitura, assim, sendo necessário se utilizar a operação *ioctl* (GENERIC. . . , s.d.). Foram disponibilizadas para o espaço do usuário operações de mais alto nível, os comandos discutidos na subseção 3.1.3.3, e também as operações de mais baixo nível, a leitura e escrita nos registradores da placa, assim, fornecendo uma interface para comunicações básicas com a placa e mantendo a possibilidade de extensão a partir da escrita direta nos registradores. A Tabela 4 mostra as operações e a definição de seus macros.

Tabela 4 – Tabela de comandos do *driver* de dispositivo para a placa RC522

Comando	**Macro utilizado**
Ler registrador	_IOWR('a', 0, uint8_t *)
Escrita em registrador	_IOW('a', 1, struct rfid_rc522_write_register_dto *)
Requisitar um *tag*	_IOR('a', 2, struct rfid_rc522_req_a_picc_dto *)
Realizar protocolo anticolisão	_IOWR('a', 3, struct rfid_rc522_anticollision_dto *)
Selecionar *tag*	_IOWR('a', 4, struct rfid_rc522_select_tag_dto *)
Autenticar em um setor da memória	_IOWR('a', 5, struct rfid_rc522_authenticate_dto *)
Ler um bloco da memória	_IOWR('a', 6, struct rfid_rc522_read_picc_block_dto *)
Escrever em um bloco da memória	_IOWR('a', 7, struct rfid_rc522_write_picc_block_dto *)
Desselecionar a *tag*	_IO('a', 8)

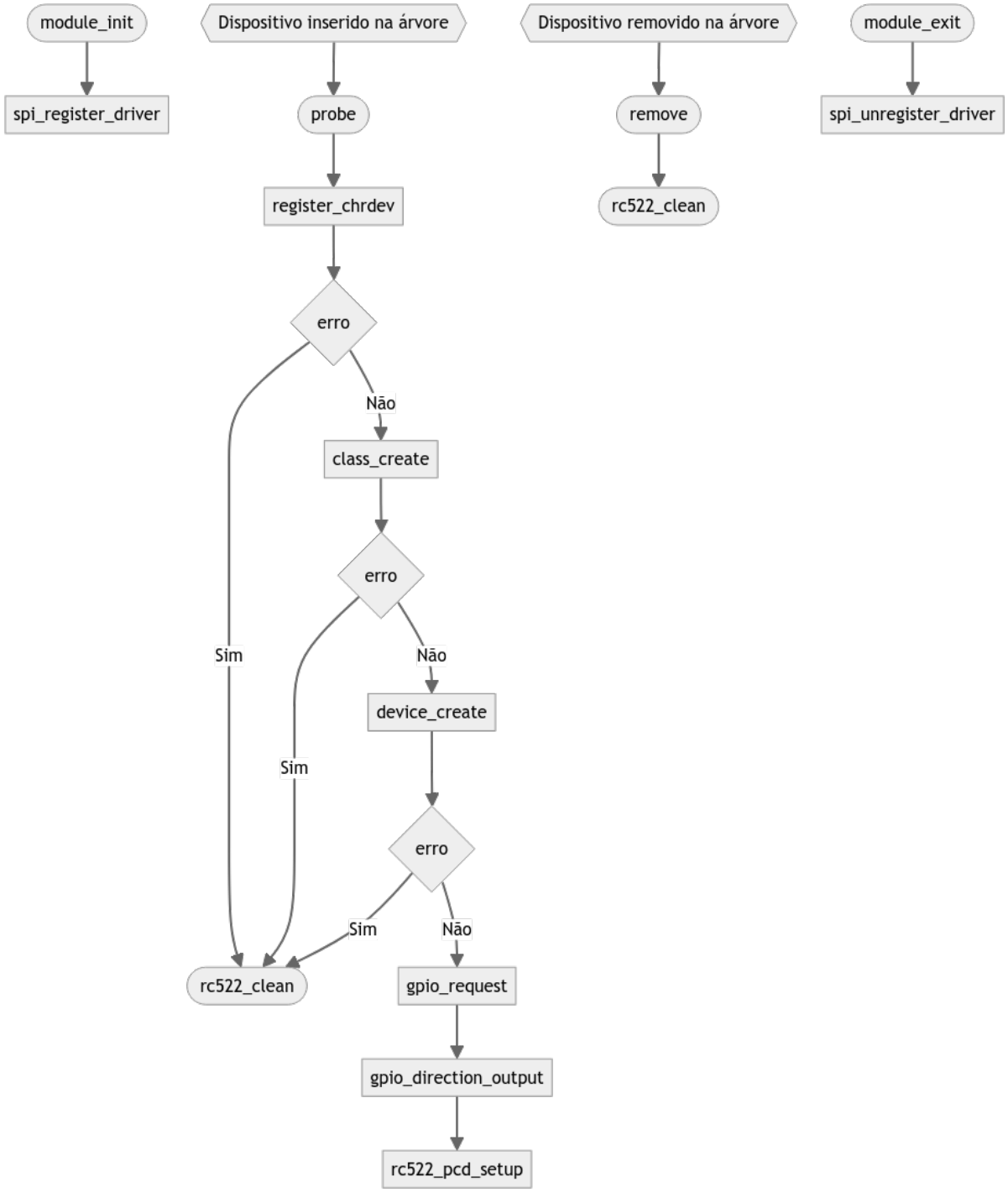

Figura 5 – Diagrama de fluxo de inicialização do módulo para a placa RC522.

4 Resultados e discussões

Neste capítulo são apresentados os resultados no desenvolvimento dos módulos do *kernel*;

4.1 *Display* LCD Hitachi HD 44780 16x2

Visando testar o *driver* de dispositivo para o *display LCD*, foi elaborado um *script* na linguagem Python, que ao ser executado escreve nos arquivos do *driver* de dispositivo e indica no terminal qual é o funcionamento esperado para aquela alteração. A execução do teste é mostrada nas Figuras 6 a 13, e o código desenvolvido pode ser visualizado nos Apêndices B.5 e B.6.

Figura 6 – Resultado do primeiro teste, que escreve a palavra “Teste” na primeira linha do *display*, e nada na segunda linha.

Figura 7 – Resultado do segundo teste, que limpa todo o *display*.

Figura 8 – Resultado do terceiro teste, que escreve a palavra “Teste” na primeira linha, e “Teste2” na segunda.

Figura 9 – Resultado do quarto teste, que limpa somente a segunda linha.

Figura 10 – Resultado do quinto teste, que reescreve “Teste2” na segunda linha.

Figura 11 – Resultado do sexto teste, que limpa somente a primeira linha.

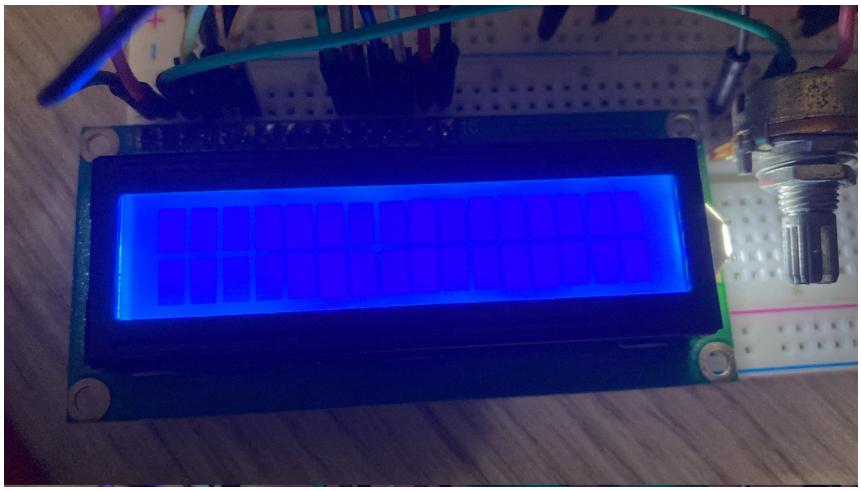

Figura 12 – Resultado do sétimo teste, que oculta o cursor.

Figura 13 – Resultado do oitavo teste, que torna o cursor visível.

Como pode ser visto nos testes, o *driver* de dispositivo isola completamente os detalhes do funcionamento do *hardware*. No *script* de teste, a única operação realizada é a de escrita nos arquivos específicos, e toda a interação com o *hardware* foi executada pelo *kernel*. Ainda, nos testes foi possível verificar o correto funcionamento das operações executadas, como limpar o *display*, mover o cursor, escrever texto, limpar linhas específicas e tornar ou não o cursor visível.

4.2 RFID-RC522

Visando utilizar o *driver* de dispositivo para a placa RFID-RC522, foram elaborados dois códigos-fonte na linguagem 'C'. Como foi utilizado o protocol de comunicação SPI, foi necessário registrar o dispositivo na árvore de dispositivos, para isso foi utilizado o overlay que pode ser visto no Apêndice C.1, ao inserir o *overlay* a árvore do próprio dispositivo é alterada, a árvore atual do sistema foi obtida utilizando o comando 'dtc -I fs /sys/firmware/devicetree/base'. É possível notar comparando os fragmentos relacionados a SPI da árvore de dispositivo indicados nos anexos C.6 e C.7 que após a inserção do *overlay* ambos os dispositivos relacionado com a interface *spidev* foram desabilitadas e o dispositivo relacionado a placa RC522 foi inserido.

O primeiro código visou testar a função de leitura e escrita nos registradores da placa, para isso, primeiro é obtido o valor presente no registrador da versão da placa, então é performada uma sequência de leituras e escritas no registrador do *buffer* da placa, e entre essas leituras e escritas é recuperado o valor do registrador que indica a quantidade de dados presente no *buffer*, o código C.8 pode ser visto no apêndice. O resultado da execução do código pode ser visto no arquivo 4.1, é possível perceber que o valor do registrador de versão foi recuperado com sucesso, e que os valores escritos no *buffer* foram recuperados ordenados, que é o comportamento de um *buffer* FIFO, além disso, foi possível ver que

o valor do registrador que indica a quantidade de *bytes* no *buffer(FIFOLevelReg)* indica corretamente a quantidade de *bytes* armazenados.

```
VersionReg Value: 0x92
FIFOLevelReg Value(inicial): 0
FIFOLevelReg Value(apos inserir um byte): 1
FIFOLevelReg Value(apos mais 4 bytes): 5
FIFO_DATA_REG_ADDR(primeiro valor inserido) Value: 11
FIFO_DATA_REG_ADDR(segundo valor inserido) Value: 12
FIFO_DATA_REG_ADDR(terceiro valor inserido) Value: 22
FIFO_DATA_REG_ADDR(quarto valor inserido) Value: 32
FIFO_DATA_REG_ADDR(quinto valor inserido) Value: 42
FIFOLevelReg Value(apos recuperar todos os bytes que haviam sido
   inseridos): 0
```

Arquivo 4.1 – Arquivo de texto resultado do teste de leitura e escrita em registradores do *driver* de dispositivo para o dispositivo RFID-RC522

O segundo código foi utilizado para utilizar as funções pré-implementadas de comunicação com as *tags* RFID, primeiro a placa é acionada para enviar uma requisição das *tags* próximas, então é realizado o protocolo anti-colisão para obter o UID de uma *tag* específica, a partir do UID a *tag* é selecionada, então é feita a leitura de todos os blocos de memória da *tag*, realizando previamente a autenticação em cada setor, então realizando uma escrita em blocos diferentes e por último lendo todos os blocos novamente, esse teste foi realizado com duas *tags* diferentes o código C.9 pode ser visto no Apêndice. O resultado do teste pode ser visto nos arquivos 4.2 e 4.3, é possível notar a execução das diferentes etapas da conexão entre a placa e a *tag*, além disso, são lidos todos os blocos da *tag*, em seguida escrita em todos os blocos, novamente leitura de todos os blocos e por último, escrita em blocos individuais, de forma a indicar o correto funcionamento do *driver* de dispositivo, também nota-se que *tags* diferentes possuem UIDs diferente, e o valor do primeiro bloco diferente, visto que é um bloco com informações de fábrica.

```
req_a_picc status 0
req_a_picc[0] = 4
req_a_picc[1] = 0
anticollision status 0
uid bytes: ca 24 ec 80 82
dto_select_tag status 0
bloco 0: |202362361281308409899100101102103104105|
bloco 1: |                 |
bloco 2: |                 |
bloco 4: |                 |
bloco 5: |                 |
bloco 6: |                 |
bloco 8: |                 |
bloco 9: |                 |
```

```
bloco 10: |                |
bloco 12: |                |
bloco 13: |                |
bloco 14: |                |
bloco 16: |                |
bloco 17: |                |
bloco 18: |                |
bloco 20: |                |
bloco 21: |                |
bloco 22: |                |
bloco 24: |                |
bloco 25: |                |
bloco 26: |                |
bloco 28: |                |
bloco 29: |                |
bloco 30: |                |
bloco 32: |                |
bloco 33: |                |
bloco 34: |                |
bloco 36: |                |
bloco 37: |                |
bloco 38: |                |
bloco 40: |                |
bloco 41: |                |
bloco 42: |                |
bloco 44: |                |
bloco 45: |                |
bloco 46: |                |
bloco 48: |                |
bloco 49: |                |
bloco 50: |                |
bloco 52: |                |
bloco 53: |                |
bloco 54: |                |
bloco 56: |                |
bloco 57: |                |
bloco 58: |                |
bloco 60: |                |
bloco 61: |                |
bloco 62: |                |
read all done
write all done
bloco 0: |2023623612813084098991001011021031041 05|
bloco 1: |                |
bloco 2: |                |
bloco 4: |bloco: 4        |
bloco 5: |bloco: 5        |
```

```
bloco 6: |bloco: 6        |
bloco 8: |bloco: 8        |
bloco 9: |bloco: 9        |
bloco 10: |bloco: 10       |
bloco 12: |bloco: 12       |
bloco 13: |bloco: 13       |
bloco 14: |bloco: 14       |
bloco 16: |bloco: 16       |
bloco 17: |bloco: 17       |
bloco 18: |bloco: 18       |
bloco 20: |bloco: 20       |
bloco 21: |bloco: 21       |
bloco 22: |bloco: 22       |
bloco 24: |bloco: 24       |
bloco 25: |bloco: 25       |
bloco 26: |bloco: 26       |
bloco 28: |bloco: 28       |
bloco 29: |bloco: 29       |
bloco 30: |bloco: 30       |
bloco 32: |bloco: 32       |
bloco 33: |bloco: 33       |
bloco 34: |bloco: 34       |
bloco 36: |bloco: 36       |
bloco 37: |bloco: 37       |
bloco 38: |bloco: 38       |
bloco 40: |bloco: 40       |
bloco 41: |bloco: 41       |
bloco 42: |bloco: 42       |
bloco 44: |bloco: 44       |
bloco 45: |bloco: 45       |
bloco 46: |bloco: 46       |
bloco 48: |bloco: 48       |
bloco 49: |bloco: 49       |
bloco 50: |bloco: 50       |
bloco 52: |bloco: 52       |
bloco 53: |bloco: 53       |
bloco 54: |bloco: 54       |
bloco 56: |bloco: 56       |
bloco 57: |bloco: 57       |
bloco 58: |bloco: 58       |
bloco 60: |bloco: 60       |
bloco 61: |bloco: 61       |
bloco 62: |bloco: 62       |
read all done
auth again 0
write done
auth again 0
```

```
write done
auth again 0
write done
bloco 0: |2023623612813084098991001011102103104105|
bloco 1: |                |
bloco 2: |                |
bloco 4: |bloco: 4        |
bloco 5: |bloco: 5        |
bloco 6: |bloco: 6        |
bloco 8: |Ola mundo, 8    |
bloco 9: |bloco: 9        |
bloco 10: |bloco: 10       |
bloco 12: |bloco: 12       |
bloco 13: |bloco: 13       |
bloco 14: |bloco: 14       |
bloco 16: |Ola mundo, 16   |
bloco 17: |bloco: 17       |
bloco 18: |bloco: 18       |
bloco 20: |bloco: 20       |
bloco 21: |bloco: 21       |
bloco 22: |bloco: 22       |
bloco 24: |bloco: 24       |
bloco 25: |bloco: 25       |
bloco 26: |bloco: 26       |
bloco 28: |bloco: 28       |
bloco 29: |bloco: 29       |
bloco 30: |bloco: 30       |
bloco 32: |Ola mundo, 32   |
bloco 33: |bloco: 33       |
bloco 34: |bloco: 34       |
bloco 36: |bloco: 36       |
bloco 37: |bloco: 37       |
bloco 38: |bloco: 38       |
bloco 40: |bloco: 40       |
bloco 41: |bloco: 41       |
bloco 42: |bloco: 42       |
bloco 44: |bloco: 44       |
bloco 45: |bloco: 45       |
bloco 46: |bloco: 46       |
bloco 48: |bloco: 48       |
bloco 49: |bloco: 49       |
bloco 50: |bloco: 50       |
bloco 52: |bloco: 52       |
bloco 53: |bloco: 53       |
bloco 54: |bloco: 54       |
bloco 56: |bloco: 56       |
bloco 57: |bloco: 57       |
```

```
bloco 58: |bloco: 58       |
bloco 60: |bloco: 60       |
bloco 61: |bloco: 61       |
bloco 62: |bloco: 62       |
read all done
```

Arquivo 4.2 – Arquivo de texto resultado do teste com *tags* em contato com o *driver* de dispositivo para o dispositivo RFID-RC522 para a primeira *tag*

```
req_a_picc status 0
req_a_picc[0] = 4
req_a_picc[1] = 0
anticollision status 0
uid bytes: 2a a9 99 16 c
dto_select_tag status 0
bloco 0: |42169153221284098991001011021031041 05|
bloco 1: |                |
bloco 2: |                |
bloco 4: |                |
bloco 5: |                |
bloco 6: |                |
bloco 8: |                |
bloco 9: |                |
bloco 10: |                |
bloco 12: |                |
bloco 13: |                |
bloco 14: |                |
bloco 16: |                |
bloco 17: |                |
bloco 18: |                |
bloco 20: |                |
bloco 21: |                |
bloco 22: |                |
bloco 24: |                |
bloco 25: |                |
bloco 26: |                |
bloco 28: |                |
bloco 29: |                |
bloco 30: |                |
bloco 32: |                |
bloco 33: |                |
bloco 34: |                |
bloco 36: |                |
bloco 37: |                |
bloco 38: |                |
bloco 40: |                |
bloco 41: |                |
bloco 42: |                |
```

```
bloco 44: |                |
bloco 45: |                |
bloco 46: |                |
bloco 48: |                |
bloco 49: |                |
bloco 50: |                |
bloco 52: |                |
bloco 53: |                |
bloco 54: |                |
bloco 56: |                |
bloco 57: |                |
bloco 58: |                |
bloco 60: |                |
bloco 61: |                |
bloco 62: |                |
read all done
write all done
bloco 0: |42169153221284098991001011021031041 05|
bloco 1: |                |
bloco 2: |                |
bloco 4: |bloco: 4        |
bloco 5: |bloco: 5        |
bloco 6: |bloco: 6        |
bloco 8: |bloco: 8        |
bloco 9: |bloco: 9        |
bloco 10: |bloco: 10       |
bloco 12: |bloco: 12       |
bloco 13: |bloco: 13       |
bloco 14: |bloco: 14       |
bloco 16: |bloco: 16       |
bloco 17: |bloco: 17       |
bloco 18: |bloco: 18       |
bloco 20: |bloco: 20       |
bloco 21: |bloco: 21       |
bloco 22: |bloco: 22       |
bloco 24: |bloco: 24       |
bloco 25: |bloco: 25       |
bloco 26: |bloco: 26       |
bloco 28: |bloco: 28       |
bloco 29: |bloco: 29       |
bloco 30: |bloco: 30       |
bloco 32: |bloco: 32       |
bloco 33: |bloco: 33       |
bloco 34: |bloco: 34       |
bloco 36: |bloco: 36       |
bloco 37: |bloco: 37       |
bloco 38: |bloco: 38       |
```

```
bloco 40: |bloco: 40       |
bloco 41: |bloco: 41       |
bloco 42: |bloco: 42       |
bloco 44: |bloco: 44       |
bloco 45: |bloco: 45       |
bloco 46: |bloco: 46       |
bloco 48: |bloco: 48       |
bloco 49: |bloco: 49       |
bloco 50: |bloco: 50       |
bloco 52: |bloco: 52       |
bloco 53: |bloco: 53       |
bloco 54: |bloco: 54       |
bloco 56: |bloco: 56       |
bloco 57: |bloco: 57       |
bloco 58: |bloco: 58       |
bloco 60: |bloco: 60       |
bloco 61: |bloco: 61       |
bloco 62: |bloco: 62       |
read all done
auth again 0
write done
auth again 0
write done
auth again 0
write done
bloco 0: |42169153221284098991001011021031041 05|
bloco 1: |                |
bloco 2: |                |
bloco 4: |bloco: 4        |
bloco 5: |bloco: 5        |
bloco 6: |bloco: 6        |
bloco 8: |Ola mundo, 8    |
bloco 9: |bloco: 9        |
bloco 10: |bloco: 10       |
bloco 12: |bloco: 12       |
bloco 13: |bloco: 13       |
bloco 14: |bloco: 14       |
bloco 16: |Ola mundo, 16   |
bloco 17: |bloco: 17       |
bloco 18: |bloco: 18       |
bloco 20: |bloco: 20       |
bloco 21: |bloco: 21       |
bloco 22: |bloco: 22       |
bloco 24: |bloco: 24       |
bloco 25: |bloco: 25       |
bloco 26: |bloco: 26       |
bloco 28: |bloco: 28       |
```

```
134 bloco 29: |bloco: 29       |
135 bloco 30: |bloco: 30       |
136 bloco 32: |Ola mundo, 32   |
137 bloco 33: |bloco: 33       |
138 bloco 34: |bloco: 34       |
139 bloco 36: |bloco: 36       |
140 bloco 37: |bloco: 37       |
141 bloco 38: |bloco: 38       |
142 bloco 40: |bloco: 40       |
143 bloco 41: |bloco: 41       |
144 bloco 42: |bloco: 42       |
145 bloco 44: |bloco: 44       |
146 bloco 45: |bloco: 45       |
147 bloco 46: |bloco: 46       |
148 bloco 48: |bloco: 48       |
149 bloco 49: |bloco: 49       |
150 bloco 50: |bloco: 50       |
151 bloco 52: |bloco: 52       |
152 bloco 53: |bloco: 53       |
153 bloco 54: |bloco: 54       |
154 bloco 56: |bloco: 56       |
155 bloco 57: |bloco: 57       |
156 bloco 58: |bloco: 58       |
157 bloco 60: |bloco: 60       |
158 bloco 61: |bloco: 61       |
159 bloco 62: |bloco: 62       |
160 read all done
```

Arquivo 4.3 – Arquivo de texto resultado do teste com *tags* em contato com o *driver* de dispositivo para o dispositivo RFID-RC522 para a segunda *tag*

5 Conclusões

Durante esse trabalho foram abordados conceitos sobre sistemas embarcados, o sistema operacional baseado em *kernel* Linux e o desenvolvimento de *drivers* de dispositivos para interfacear dispositivos de *hardware*. O estudo destacou a importância dos *drivers* de dispositivos como componentes essenciais para a comunicação eficiente entre o *hardware* e o sistema operacional, facilitando a interação dos usuários com os dispositivos de maneira abstraída e segura. Também foi possível abordar as dificuldades adicionados no desenvolvimento, como uma menor quantidade de materiais disponíveis e uma maior curva de aprendizagem, em comparação com o desenvolvimento em espaço de usuário.

Esse trabalho também buscou desenvolver *drivers* de dispositivos de módulos de *hardware* comumente utilizados. O primeiro foi a sofisticação de um *driver* de dispositivo já existente para *displays* LCD Hitachi HD 44780, visando entender os benefícios do desenvolvimento de *drivers* de dispositivos, e gerando como resultado esse *driver* de dispositivo aprimorado, disponível nos apêndices B.3 e B.4. Também, foi desenvolvido um novo *driver* de dispositivo para módulos RFID-RC522, disponível nos apêndices C.3, C.4 e C.5, disponibilizando uma forma de interagir com esse *hardware* a partir do *kernel* Linux. Os resultados obtidos fornecem uma base sólida para futuros desenvolvimentos e melhorias, visto que o código será disponibilizado de forma aberta.

Referências

AJANI, S. *Remote SSH with Visual Studio Code*. 2019. Disponível em: <https://code.visualstudio.com/blogs/2019/07/25/remote-ssh>. Citado na página 27.

ANSARI, B. R.; KAUR, M. Design and implementation of character device driver for customized kernel of arm based platform. In: *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*. [S.l.: s.n.], 2017. p. 721–724. Citado na página 22.

CORBET, J. et al. *Linux device drivers*. 3rd ed. ed. [S.l.]: O'Reilly, 2005. ISBN 978-0-596-00590-0. Citado 6 vezes nas páginas 13, 15, 16, 17, 18 e 21.

DEVICE Classes. Linux Kernel Organization, s.d. Disponível em: <https://www.kernel.org/doc/html/v5.4/driver-api/driver-model/class.html>. Citado na página 19.

DEVICE drivers infrastructure. Linux Kernel Organization, s.d. Disponível em: <https://www.kernel.org/doc/html/v5.4/driver-api/infrastructure.html?highlight=device_create#c.device_create>. Citado na página 19.

DEVICETREE Specification. Linaro, s.d. Disponível em: <https://www.devicetree.org/specifications/>. Citado na página 17.

DRIVER basics. Linux Kernel Organization, s.d. Disponível em: <https://www.kernel.org/doc/html/v5.4/driver-api/basics.html#driver-entry-and-exit-points>. Citado na página 18.

EONE, S. *HOW TO USE INTELLIGENT L.C.D.S*. 1997. Disponível em: <http://www.emcu.it/LCD/lcd1.pdf>. Citado na página 24.

GARCIA, D. C. *lcd_device_driver*. 2021. Disponível em: <https://github.com/DiogoCaetanoGarcia/Sistemas_Embarcados/blob/c04a3e19722d61e9c35284f77ed8be101d53e990/5_T%C3%B3picos_avan%C3%A7ados/5.6_Aplica%C3%A7%C3%B5es/2_LCD_device_driver/lcd_device_driver.c>. Citado na página 27.

GENERIC I/O Control operations. GNU, s.d. Disponível em: <https://www.gnu.org/software/libc/manual/html_node/IOCTLs.html>. Citado na página 31.

ILETT, J. *Specification for LCD Module 1602A-1*. 2005. Disponível em: <https://www.openhacks.com/uploadsproductos/eone-1602a1.pdf>. Citado na página 24.

ISO/IEC 14443-3:2018. ISO, 2018. Disponível em: <https://www.iso.org/standard/73598.html>. Citado na página 26.

KERNEL source tree for raspberry pi. Raspberrypi, s.d. Disponível em: <https://github.com/raspberrypi/linux>. Citado na página 17.

LI, Z. et al. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. Canterbury CA United Kingdom: ACM, 2019. p. 1–10. ISBN

978-1-4503-7164-3. Disponível em: <https://dl.acm.org/doi/10.1145/3339252.3340506>. Citado na página 13.

LINUX and the Devicetree. Linux Kernel Organization, s.d. Disponível em: <https://www.kernel.org/doc/html/next/devicetree/usage-model.html>. Citado na página 17.

MADIEU, J. *Linux device driver development: everything you need to start with device driver development for Linux kernel and embedded Linux*. Second edition. [S.l.]: Packt Publishing Ltd., 2022. Citado 6 vezes nas páginas 13, 16, 17, 18, 19 e 21.

MICROSOFT. *Visual Studio Code*. 2016. Disponível em: <https://code.visualstudio.com/>. Citado na página 27.

NGUYEN, V. Implementation of linux gpio device driver on raspberry pi platform. Helsinki Metropolia University of Applied Sciences, 2014. Citado na página 22.

NXP. *MFRC522 Standard performance MIFARE and NTAG frontend*. 2016. Disponível em: <https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf>. Citado 3 vezes nas páginas 24, 25 e 26.

PATOLIYA, J. J. et al. Embedded Linux Based Smart Secure IoT Intruder Alarm System Implemented on BeagleBone Black. In: PATEL, K. K. et al. (Ed.). *Soft Computing and its Engineering Applications*. Singapore: Springer Singapore, 2021. v. 1374, p. 343–355. ISBN 9789811607073 9789811607080. Series Title: Communications in Computer and Information Science. Disponível em: <https://link.springer.com/10.1007/978-981-16-0708-0_28>. Citado na página 22.

PI, R. *Remote access*. 2019. Disponível em: <https://www.raspberrypi.com/documentation/computers/remote-access.html>. Citado na página 26.

PI, R. *Raspberry Pi hardware*. s.d. Disponível em: <https://www.raspberrypi.com/documentation/computers/raspberry-pi.html>. Citado 2 vezes nas páginas 7 e 23.

REDHAT. *Kernel linux: O que É O kernel linux?* Red Hat, 2019. Disponível em: <https://www.redhat.com/pt-br/topics/linux/what-is-the-linux-kernel>. Citado na página 15.

Sang-Pil Moon et al. Embedded linux implementation on a commercial digital TV system. *IEEE Transactions on Consumer Electronics*, v. 49, n. 4, p. 1402–1407, nov. 2003. ISSN 0098-3063. Disponível em: <http://ieeexplore.ieee.org/document/1261247/>. Citado na página 22.

SHUKLA, A. et al. Evaluating the performance of user-space and kernel-space web servers. In: *CASCON*. [S.l.: s.n.], 2004. v. 4, p. 189–201. Citado na página 16.

SILBERSCHATZ, A.; GAGNE, G.; GALVIN, P. B. *Operating System Concepts*. 10th edition. ed. [S.l.]: Wiley Global Education US, 2018. ISBN 9781119320913. Citado 2 vezes nas páginas 15 e 16.

SIMMONDS, C.; PURDIE, R. *Mastering embedded Linux programming: harness the power of Linux to create versatile and robust embedded solutions*. [S.l.]: Packt Publishing, 2015. ISBN 978-1-78439-902-3. Citado 3 vezes nas páginas 13, 14 e 15.

THE Linux Kernel Archives. Linux Kernel Organization, s.d. Disponível em: <https://www.kernel.org/>. Citado na página 17.

WHITE, E. *Making embedded systems: design patterns for great software*. 1. ed. ed. [S.l.]: O'Reilly, 2012. ISBN 978-1-4493-0214-6. Citado na página 15.

Apêndices

APÊNDICE A – Exemplos de árvores de dispositivos

A.1 Exemplo da estrutura de um árvore de dispositivos

```
/dts-v1/;
/ {
    compatible = "organizao,modelo";
    serial-number = "numero serial";
    model = "Nome do modelo";

    soc: soc {
        gpu: gpu {
            compatible = "organizao-gpu,modelo-gpu";
            status = "okay";
        };

        i2c: i2c {
            clocks = <0x08 0x14>;
            compatible = "organizao-i2c,modelo-i2c";
            status = "okay";
        };
    };
};
```

A.2 Exemplo de overlay para árvore de dispositivos

```
/dts-v1/;
/plugin/;
/{
    fragment@0 {
        target = <&i2c>;
        __overlay__ {
            i2c-slave: i2c-slave@0 {
                compatible = "company,i2c-dev-model";
                status = "okay";
                reg = <0x0>;
            };
         };
    };

    fragment@1 {
```

```
16          target = <&gpu>;
17          __overlay__ {
18              firmware = <0x06>;
19          };
20      };
21
22
23  };
```

APÊNDICE B – Códigos utilizados no *driver* de dispositivo do *display* Hitashi hd44780

B.1 Repositório dos códigos

Os códigos desenvolvidos estão disponíveis em <https://github.com/jschneiderm98/device-drivers/tree/6c6494f8e966837c5cbff90c6aa55e44e2531da1/lcd_hitachi_hd_44780>

B.2 Arquivo *Makefile* para compilação do *driver* de dispositivo para o *display* lcd

```
obj-m := lcd_device_driver.o
all:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
msg:
  dmesg | tail -20
```

B.3 Arquivo de código do *header* do *driver* de dispositivo para o *display* lcd

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

// Constantes gerais do projeto

#define DEVICE_NAME "lcd_device_driver"
#define CLASS_NAME  "lcd_device_driver_class"
#define LCD_LINE_LEN 16
#define MAX_CURSOR_POS 80
#define FIRST_LINE_FIRST_POS 0
#define SECOND_LINE_FIRST_POS 40
#define EMPTY_LCD_LINE "                                "
#define CONFIG_MESSAGE_SIZE 48
#define DATA_MESSAGE_SIZE 84

// Constantes para comando do display lcd

```

```
17 #define MODO_COMANDO 0
18 #define COMANDO_NIBBLE_INICIAL_MODO_4_BITS 0x2
19 #define COMANDO_1_LINE 0x20
20 #define COMANDO_1_LINE_CHAR '1'
21 #define COMANDO_2_LINE 0x28
22 #define COMANDO_2_LINE_CHAR '2'
23 #define COMANDO_CURSOR_VISIVEL 0x0F
24 #define COMANDO_CURSOR_VISIVEL_CHAR '3'
25 #define COMANDO_CURSOR_NAO_VISIVEL 0x0C
26 #define COMANDO_CURSOR_NAO_VISIVEL_CHAR '4'
27 #define COMANDO_LIMPAR_DISPLAY 0x01
28 #define COMANDO_LIMPAR_DISPLAY_CHAR '5'
29 #define COMANDO_RETORNAR_CURSOR 0x02
30 #define COMANDO_RETORNAR_CURSOR_CHAR '6'
31 #define COMANDO_CURSOR_SEGUNDO_LINHA 0xC0
32 #define COMANDO_CURSOR_SEGUNDO_LINHA_CHAR '7'
33 #define COMANDO_CURSOR_SEGUNDO_LINHA 0xC0
34 #define COMANDO_LIMPAR_PRIMEIRA_LINHA_CHAR '8'
35 #define COMANDO_LIMPAR_SEGUNDO_LINHA_CHAR '9'
36 #define COMANDO_INVALIDO 0x0
37 #define COMANDO_INICIAL 0x3
38 #define MODO_DADO 1
39
40 // Constantes da posição de cada bit no array de gpio's
41
42 #define EN 0
43 #define RS 1
44 #define D4 2
45 #define D5 3
46 #define D6 4
47 #define D7 5
48
49 typedef struct lcd_device_manager
50 {
51     int Device_Open;    // Device aberto? Usado para prevenir acesso
    multiplo ao device
52     int Device_Counter; // Posicao do arquivo para leitura e escrita
53     struct device* driver_device;
54     struct cdev* cdev_data;
55     struct file_operations fops;
56     dev_t dev_number;
57 } lcd_device_manager;
58
59 typedef struct lcd_config
60 {
61     int posicao_cursor;  //indica posição autal do cursor 0-39 primeira
    linha, 40-79 segunda linha, max -> 80
```

```
62     int cursor_ligado;  // indica se o cursor esta ligado no display, 0
       - false, 1 - true
63     int modo_linha; // indica quantas linhas estão atividas no display 1
       - 1 linha, 2 - 2 linhas
64 } lcd_config;
65
66 int init_module(void);
67 void cleanup_module(void);
68 static int data_device_open(struct inode *, struct file *);
69 static int data_device_release(struct inode *, struct file *);
70 static ssize_t data_device_read(struct file *, char *, size_t, loff_t *)
       ;
71 static ssize_t data_device_write(struct file *, const char *, size_t,
       loff_t *);
72 static int config_device_open(struct inode *, struct file *);
73 static int config_device_release(struct inode *, struct file *);
74 static ssize_t config_device_read(struct file *, char *, size_t, loff_t
       *);
75 static ssize_t config_device_write(struct file *, const char *, size_t,
       loff_t *);
76 char Send_Nibble(char nibble, char nibble_type);
77 char Send_Byte(char byte, char byte_type);
78 void Clear_LCD(void);
79 void Config_LCD(void);
80 void Send_String(char *str);
81 void jiffies_delay(unsigned int n);
82 void register_lcd_values(char *str, int start, int length);
83 void clear_lcd_values(void);
```

B.4 Arquivo de código do *driver* de dispositivo para o *display* lcd

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/fs.h>
4 #include <asm/uaccess.h>
5 #include <linux/init.h>
6 #include <linux/cdev.h>
7 #include <linux/device.h>
8 #include <linux/gpio.h>
9 #include "constants.h"
10
11 MODULE_LICENSE("GPL v2");
12 MODULE_AUTHOR("Diogo Caetano Garcia <diogogarcia@unb.br>, Júlio César
       Schneider Martins <jschneiderm98@gmail.com>");
13 MODULE_DESCRIPTION("Device driver to interface with lcd displays that
       use hitashi_hd_44780");
14
```

```
15 #define MSG_OK(s) printk(KERN_INFO "%s: %s\n", DEVICE_NAME, s)
16 #define MSG_BAD(s, err_val) printk(KERN_ERR "%s: %s %ld\n", DEVICE_NAME,
       s, err_val)
17
18 unsigned short en = 4;
19 module_param(en, short, 0644);
20 MODULE_PARM_DESC(en, "Optional, gpio for the EN pin of the lcd. Defaults
       to gpio 4");
21 unsigned short rs = 17;
22 module_param(rs, short, 0644);
23 MODULE_PARM_DESC(rs, "Optional, gpio for the RS pin of the lcd. Defaults
       to gpio 17");
24 unsigned short d4 = 22;
25 module_param(d4, short, 0644);
26 MODULE_PARM_DESC(d4, "Optional, gpio for the D4 pin of the lcd. Defaults
       to gpio 22");
27 unsigned short d5 = 23;
28 module_param(d5, short, 0644);
29 MODULE_PARM_DESC(d5, "Optional, gpio for the D5 pin of the lcd. Defaults
       to gpio 23");
30 unsigned short d6 = 24;
31 module_param(d6, short, 0644);
32 MODULE_PARM_DESC(d6, "Optional, gpio for the D6 pin of the lcd. Defaults
       to gpio 24");
33 unsigned short d7 = 25;
34 module_param(d7, short, 0644);
35 MODULE_PARM_DESC(d7, "Optional, gpio for the D7 pin of the lcd. Defaults
       to gpio 25");
36
37 static struct gpio lcd_pins[] = {
38     {  4, GPIOF_OUT_INIT_LOW, "EN" },
39     { 17, GPIOF_OUT_INIT_LOW, "RS" },
40     { 22, GPIOF_OUT_INIT_LOW, "D4" },
41     { 23, GPIOF_OUT_INIT_LOW, "D5" },
42     { 24, GPIOF_OUT_INIT_LOW, "D6" },
43     { 25, GPIOF_OUT_INIT_LOW, "D7" },
44 };
45
46 static struct class*  lcd_device_driver_Class  = NULL; ///< The device-
      driver class struct pointer
47 static char full_lcd_characters[MAX_CURSOR_POS + 1];
48 static char lcd_data_buffer[DATA_MESSAGE_SIZE];
49 static char lcd_config_buffer[CONFIG_MESSAGE_SIZE];
50
51 lcd_config config = {
52   .cursor_ligado = 1,
53   .modo_linha = 1,
```

```
54   .posicao_cursor = 7
55 };
56
57
58 lcd_device_manager data_device = {
59   .Device_Open = 0,
60   .Device_Counter = 0,
61   .fops = {
62     .read = data_device_read,
63     .write = data_device_write,
64     .open = data_device_open,
65     .release = data_device_release
66   },
67   .cdev_data = NULL,
68   .driver_device = NULL,
69   .dev_number = 0
70 };
71
72 lcd_device_manager config_device = {
73   .Device_Open = 0,
74   .Device_Counter = 0,
75   .fops = {
76     .read = config_device_read,
77     .write = config_device_write,
78     .open = config_device_open,
79     .release = config_device_release
80   },
81   .cdev_data = NULL,
82   .driver_device = NULL,
83   .dev_number = 0
84 };
85
86 #define CLEAN_ALL                              0
87 #define CLEAN_DEVICE_CONFIG                    1
88 #define CLEAN_DEVICE_DATA                      2
89 #define CLEAN_CLASS                            3
90 #define CLEAN_CDEV_CONFIG                      4
91 #define CLEAN_CDEV_DATA                        5
92 #define CLEAN_REGION                           6
93 void module_clean_level(unsigned int level)
94 {
95   int i;
96   if(level<1)
97   {
98     // Apagar LEDs
99     for(i = 0; i < ARRAY_SIZE(lcd_pins); i++)
100      gpio_set_value(lcd_pins[i].gpio, 0);
```

```
101     // Liberar GPIOs dos LEDs
102     gpio_free_array(lcd_pins, ARRAY_SIZE(lcd_pins));
103   }
104   if(level<2) { // Remove o dispositivo de configuracao
105     device_destroy(lcd_device_driver_Class, config_device.dev_number);
106   }
107   if(level<3) { // Remove o dispositivo de dados
108     device_destroy(lcd_device_driver_Class, data_device.dev_number);
109   }
110   if(level<4) // Desfaz o registro da classe de dispositivo e remove a
       classe de dispositivo
111   {
112     class_unregister(lcd_device_driver_Class);
113     class_destroy(lcd_device_driver_Class);
114   }
115   if(level<5) // Desaloca a struct cdev para o dispositivo de
       configuracao
116   {
117     cdev_del(config_device.cdev_data);
118   }
119   if(level<6) // Desaloca a struct cdev para o dispositivo de dados
120   {
121     cdev_del(data_device.cdev_data);
122   }
123   if(level<7) // Desfaz o registro do major number e da regiao alocada
124     unregister_chrdev_region(data_device.dev_number, 1);
125 }
126
127 int init_module(void)
128 {
129   int ret, i, result, err;
130
131   result = alloc_chrdev_region(&data_device.dev_number, 0, 2,
       DEVICE_NAME);
132   if(result < 0)
133   {
134     MSG_BAD("Error while allocating char device region", (long int)
       result);
135     return result;
136   }
137   config_device.dev_number = MKDEV(MAJOR(data_device.dev_number), MINOR(
       data_device.dev_number) + 1);
138
139   data_device.cdev_data = cdev_alloc();
140   if(!data_device.cdev_data) {
141     module_clean_level(CLEAN_REGION);
142     MSG_BAD("Error while allocating data-cdev", 0L);
```

```
143     return -1;
144   }
145
146   config_device.cdev_data = cdev_alloc();
147   if(!config_device.cdev_data) {
148     module_clean_level(CLEAN_CDEV_DATA);
149     MSG_BAD("Error while allocating config-cdev", 0L);
150     return -1;
151   }
152
153   cdev_init(data_device.cdev_data, &data_device.fops);
154   cdev_init(config_device.cdev_data, &config_device.fops);
155
156   err = cdev_add(data_device.cdev_data, data_device.dev_number, 1);
157   if (err < 0)
158   {
159     module_clean_level(CLEAN_CDEV_CONFIG);
160     MSG_BAD("Error while adding cdev", (long int)err);
161     return err;
162   }
163
164   err = cdev_add(config_device.cdev_data, config_device.dev_number, 1);
165   if (err < 0)
166   {
167     module_clean_level(CLEAN_CDEV_CONFIG);
168     MSG_BAD("Error while adding cdev", (long int)err);
169     return err;
170   }
171
172   // Registrar a classe do dispositivo
173   lcd_device_driver_Class = class_create(THIS_MODULE, CLASS_NAME);
174   if(IS_ERR(lcd_device_driver_Class)) // Se houve erro no registro
175   {
176     module_clean_level(CLEAN_CDEV_CONFIG);
177     unregister_chrdev_region(data_device.dev_number, 1);
178     MSG_BAD("falhou em registrar a classe do dispositivo", PTR_ERR(
      lcd_device_driver_Class));
179     return PTR_ERR(lcd_device_driver_Class); // Correct way to return an
      error on a pointer
180   }
181   MSG_OK("classe do dispositivo registrada corretamente");
182
183   // Registrar o device driver
184   data_device.driver_device = device_create(lcd_device_driver_Class,
      NULL, data_device.dev_number, NULL, DEVICE_NAME "!data");
185   if(IS_ERR(data_device.driver_device)) // Se houve erro no registro
186   {
```

```
187     module_clean_level(CLEAN_CLASS);
188     MSG_BAD("falhou em criar o device driver de dados", PTR_ERR(
        data_device.driver_device));
189     return PTR_ERR(data_device.driver_device);
190   }
191   MSG_OK("dispositivo de dados criado corretamente");
192
193   config_device.driver_device = device_create(lcd_device_driver_Class,
       NULL, config_device.dev_number, NULL, DEVICE_NAME "!config");
194   if(IS_ERR(config_device.driver_device)) // Se houve erro no registro
195   {
196     module_clean_level(CLEAN_DEVICE_DATA);
197     MSG_BAD("falhou em criar o device driver de configuracao", PTR_ERR(
       config_device.driver_device));
198     return PTR_ERR(config_device.driver_device);
199   }
200   MSG_OK("dispositivo de configuracao criado corretamente");
201
202   // Registrar GPIOs para LEDs, ligar LEDs
203
204   lcd_pins[EN].gpio = en;
205   lcd_pins[RS].gpio = rs;
206   lcd_pins[D4].gpio = d4;
207   lcd_pins[D5].gpio = d5;
208   lcd_pins[D6].gpio = d6;
209   lcd_pins[D7].gpio = d7;
210
211   ret = gpio_request_array(lcd_pins, ARRAY_SIZE(lcd_pins));
212   if(ret) {
213     module_clean_level(CLEAN_DEVICE_CONFIG);
214     MSG_BAD("não conseguiu acesso ao GPIO", (long int)ret);
215   }
216   else
217     MSG_OK("modulo carregado");
218
219   for(i=0; i < MAX_CURSOR_POS; i++)
220     full_lcd_characters[i] = ' ';
221   full_lcd_characters[MAX_CURSOR_POS] = '\0';
222
223   Config_LCD();
224   Send_String("Ola LCD");
225   register_lcd_values("Ola LCD", 0, 7);
226
227   return ret;
228 }
229
230 void cleanup_module(void)
```

```
231 {
232   Clear_LCD();
233   module_clean_level(CLEAN_ALL);
234   MSG_OK("modulo descarregado");
235 }
236
237 static int device_open(lcd_device_manager *dev) {
238   if(dev->Device_Open) return -EBUSY;
239   dev->Device_Open++; // Trava acesso ao device
240   dev->Device_Counter = 0; // Conta quantos caracteres foram lidos
241   try_module_get(THIS_MODULE); // Incrementa o contador de uso do modulo
242   return 0;
243 }
244
245 static int data_device_open(struct inode *inode, struct file *file)
246 {
247   snprintf(lcd_data_buffer, DATA_MESSAGE_SIZE, "|%s|\n",
       full_lcd_characters);
248   return device_open(&data_device);
249 }
250
251 static int config_device_open(struct inode *inode, struct file *file)
252 {
253   snprintf(
254     lcd_config_buffer,
255     CONFIG_MESSAGE_SIZE,
256     "posicao_cursor:%02d;cursor_ligado:%d;modo_linha:%d\n",
257     config.posicao_cursor, config.cursor_ligado, config.modo_linha
258   );
259   return device_open(&config_device);
260 }
261
262 static int device_release(lcd_device_manager *dev)
263 {
264   dev->Device_Open--; // Libera acesso ao device
265   module_put(THIS_MODULE); // Decrementa o contador de uso do modulo
266   return 0;
267 }
268
269 static int data_device_release(struct inode *inode, struct file *file)
270 {
271   return device_release(&data_device);
272 }
273
274 static int config_device_release(struct inode *inode, struct file *file)
275 {
276   return device_release(&config_device);
```

```
277 }
278
279 static ssize_t data_device_read(struct file *filp,  /* see include/linux
    /fs.h   */
280         char *buffer,  /* buffer to fill with data */
281         size_t length, /* length of the buffer     */
282         loff_t * offset)
283 {
284   int actual_length;
285   // End of file
286   if(data_device.Device_Counter >= DATA_MESSAGE_SIZE)
287     return 0;
288   actual_length = length + data_device.Device_Counter >
     DATA_MESSAGE_SIZE ? (DATA_MESSAGE_SIZE - data_device.Device_Counter)
     : length;
289
290   snprintf(buffer, actual_length, "%s", lcd_data_buffer+data_device.
     Device_Counter);
291   data_device.Device_Counter += actual_length;
292   return actual_length;
293 }
294
295 static ssize_t config_device_read(struct file *filp,  /* see include/
    linux/fs.h   */
296         char *buffer,  /* buffer to fill with data */
297         size_t length, /* length of the buffer     */
298         loff_t * offset)
299 {
300   int actual_length = length + config_device.Device_Counter >
     CONFIG_MESSAGE_SIZE ? (CONFIG_MESSAGE_SIZE - config_device.
     Device_Counter) : length;
301
302   if (config_device.Device_Counter > 0)
303     return 0;
304   snprintf(buffer, actual_length, "%s", lcd_config_buffer+config_device.
     Device_Counter);
305   config_device.Device_Counter = actual_length;
306   return actual_length;
307 }
308
309 static ssize_t data_device_write(struct file *filp, const char *buff,
    size_t len, loff_t * off)
310 {
311   int new_cursor_pos, actual_length;
312   char local_buff[MAX_CURSOR_POS+1];
313   //Clear_LCD();
314   if (config.posicao_cursor >= MAX_CURSOR_POS) {
```

```
315     return len;
316   }
317   new_cursor_pos = config.posicao_cursor + len;
318   actual_length = new_cursor_pos > MAX_CURSOR_POS ? (MAX_CURSOR_POS -
       config.posicao_cursor) : len;
319   new_cursor_pos = config.posicao_cursor + actual_length;
320   snprintf(local_buff, actual_length+1, buff);
321   Send_String(local_buff);
322   register_lcd_values(local_buff, config.posicao_cursor, actual_length);
323   config.posicao_cursor = new_cursor_pos % MAX_CURSOR_POS;
324   return len;
325 }
326
327 void limpar_linha(char comando) {
328   if (comando == COMANDO_LIMPAR_PRIMEIRA_LINHA_CHAR) {
329     Send_Byte(COMANDO_RETORNAR_CURSOR, MODO_COMANDO);
330     Send_String(EMPTY_LCD_LINE);
331     Send_Byte(COMANDO_RETORNAR_CURSOR, MODO_COMANDO);
332     config.posicao_cursor = 0;
333     register_lcd_values(EMPTY_LCD_LINE, FIRST_LINE_FIRST_POS,
      LCD_LINE_LEN);
334     return;
335   }
336   Send_Byte(COMANDO_CURSOR_SEGUNDO_LINHA, MODO_COMANDO);
337   Send_String(EMPTY_LCD_LINE);
338   Send_Byte(COMANDO_CURSOR_SEGUNDO_LINHA, MODO_COMANDO);
339   config.posicao_cursor = 40;
340   register_lcd_values(EMPTY_LCD_LINE, SECOND_LINE_FIRST_POS,
      SECOND_LINE_FIRST_POS + LCD_LINE_LEN);
341 }
342
343 char select_configuration(char indicador) {
344   switch(indicador) {
345     case COMANDO_1_LINE_CHAR:
346       Send_Byte(COMANDO_LIMPAR_DISPLAY, MODO_COMANDO);
347       clear_lcd_values();
348       config.modo_linha = 1;
349       return COMANDO_1_LINE;
350     case COMANDO_2_LINE_CHAR:
351       Send_Byte(COMANDO_LIMPAR_DISPLAY, MODO_COMANDO);
352       clear_lcd_values();
353       config.modo_linha = 2;
354       return COMANDO_2_LINE;
355     case COMANDO_CURSOR_VISIVEL_CHAR:
356       config.cursor_ligado = 1;
357       return COMANDO_CURSOR_VISIVEL;
358     case COMANDO_CURSOR_NAO_VISIVEL_CHAR:
```

```
359       config.cursor_ligado = 0;
360       return COMANDO_CURSOR_NAO_VISIVEL;
361     case COMANDO_LIMPAR_DISPLAY_CHAR:
362       clear_lcd_values();
363       return COMANDO_LIMPAR_DISPLAY;
364     case COMANDO_RETORNAR_CURSOR_CHAR:
365       config.posicao_cursor = 0;
366       return COMANDO_RETORNAR_CURSOR;
367     case COMANDO_CURSOR_SEGUNDO_LINHA_CHAR:
368       config.posicao_cursor = 40;
369       return COMANDO_CURSOR_SEGUNDO_LINHA;
370     case COMANDO_LIMPAR_PRIMEIRA_LINHA_CHAR:
371     case COMANDO_LIMPAR_SEGUNDO_LINHA_CHAR:
372       limpar_linha(indicador);
373       return COMANDO_INVALIDO;
374     default:
375       return COMANDO_INVALIDO;
376   }
377 }
378
379 static ssize_t config_device_write(struct file *filp, const char *buff,
      size_t len, loff_t * off)
380 {
381   char comando = select_configuration(buff[0]);
382   if(comando != COMANDO_INVALIDO)
383     Send_Byte(comando, MODO_COMANDO);
384   return 1;
385 }
386
387 char Send_Nibble(char nibble, char nibble_type)
388 {
389   if((nibble_type!=MODO_DADO)&&(nibble_type!=MODO_COMANDO))
390     return -1;
391   gpio_set_value(lcd_pins[EN].gpio, 1);
392   gpio_set_value(lcd_pins[RS].gpio, nibble_type);
393   gpio_set_value(lcd_pins[D4].gpio, nibble&1);
394   gpio_set_value(lcd_pins[D5].gpio, (nibble>>1)&1);
395   gpio_set_value(lcd_pins[D6].gpio, (nibble>>2)&1);
396   gpio_set_value(lcd_pins[D7].gpio, (nibble>>3)&1);
397   gpio_set_value(lcd_pins[EN].gpio, 0);
398   jiffies_delay(1);
399   return 0;
400 }
401
402 char Send_Byte(char byte, char byte_type)
403 {
404   if(Send_Nibble(byte>>4, byte_type)==-1)
```

```
405     return -1;
406   Send_Nibble(byte & 0xF, byte_type);
407   return 0;
408 }
409
410 void Clear_LCD(void)
411 {
412   Send_Byte(COMANDO_LIMPAR_DISPLAY, MODO_COMANDO);
413   jiffies_delay(2);
414   Send_Byte(COMANDO_RETORNAR_CURSOR, MODO_COMANDO);
415   jiffies_delay(2);
416 }
417
418 void force_4bit_mode(void)
419 {
420   // Manda 3 nibbles 0011, garantindo que o lcd esteja em um modo 8 bits
421   // independente de estar no modo 4 bits ou 8 bits anteriormente
422   Send_Nibble(COMANDO_INICIAL, MODO_COMANDO);
423   Send_Nibble(COMANDO_INICIAL, MODO_COMANDO);
424   Send_Nibble(COMANDO_INICIAL, MODO_COMANDO);
425   // Manda 1 nibble setando o modo 4 bits
426   Send_Nibble(COMANDO_NIBBLE_INICIAL_MODO_4_BITS, MODO_COMANDO);
427 }
428
429 void Config_LCD(void)
430 {
431   jiffies_delay(1);
432   force_4bit_mode();
433   Send_Byte(COMANDO_CURSOR_VISIVEL, MODO_COMANDO);
434   Send_Byte(COMANDO_1_LINE, MODO_COMANDO);
435   Clear_LCD();
436 }
437
438 void Send_String(char *str)
439 {
440   int i = 0;
441   while(str[i]!='\0')
442   {
443     Send_Byte(str[i], MODO_DADO);
444     i++;
445   }
446 }
447
448 void jiffies_delay(unsigned int n)
449 {
450   unsigned long t_end = jiffies + n*HZ/100;
451   while(jiffies<t_end);
```

```
452 }
453
454 void register_lcd_values(char *str, int start, int length) {
455   int i, new_cursor_pos, actual_length;
456   if (start >= MAX_CURSOR_POS) {
457     return;
458   }
459
460   new_cursor_pos = start + length;
461   actual_length = new_cursor_pos > MAX_CURSOR_POS ? (MAX_CURSOR_POS -
       start) : length;
462   new_cursor_pos = start + actual_length;
463
464   for(i = 0; i < actual_length && str[i] != '\0' && str[i] != '\x00'; i
       ++) {
465     full_lcd_characters[i + start] = str[i];
466   }
467 }
468
469 void clear_lcd_values(void) {
470   int i;
471   for(i=0; i < MAX_CURSOR_POS; i++)
472     full_lcd_characters[i] = ' ';
473   full_lcd_characters[MAX_CURSOR_POS] = '\0';
474   config.posicao_cursor = 0;
475 }
```

B.5 Arquivo de código de definição do *teste* do *driver* de dispositivo para o *display* lcd

```
1  from enum import Enum
2
3  LCD_DEVICE_DRIVER_CONFIG_PATH = '/dev/lcd_device_driver/config'
4  LCD_DEVICE_DRIVER_DATA_PATH = '/dev/lcd_device_driver/data'
5
6  class ConfigOperation(Enum):
7    MODO_UMA_LINHA = '1'
8    MODO_DUAS_LINHAS = '2'
9    CURSOR_VISIVEL = '3'
10   CURSOR_NAO_VISIVEL = '4'
11   LIMPAR_DISPLAY = '5'
12   MOVER_CURSOR_PRIMEIRA_LINHA = '6'
13   MOVER_CURSOR_SEGUNDA_LINHA = '7'
14   LIMPAR_PRIMEIRA_LINHA = '8'
15   LIMPAR_SEGUNDA_LINHA = '9'
16
```

```
def lcd_config(operation: ConfigOperation):
  with open(LCD_DEVICE_DRIVER_CONFIG_PATH, 'w') as config_device:
    config_device.write(operation.value)

def lcd_data(data: str) -> None:
  with open(LCD_DEVICE_DRIVER_DATA_PATH, 'w') as data_device:
    data_device.write(data)
```

B.6 Arquivo de código do *teste* do *driver* de dispositivo para o *display* lcd

```
import os
from lcd_driver import lcd_config, lcd_data, ConfigOperation

def waitForResult(expected_result: str) -> None:
  os.system('clear')
  print(f'Resultado esperado: {expected_result}')
  print('Digite para continuar')
  input()

print('Iniciando testes')
lcd_config(ConfigOperation.LIMPAR_DISPLAY)
lcd_config(ConfigOperation.MODO_UMA_LINHA)
lcd_data('Teste')
lcd_config(ConfigOperation.MOVER_CURSOR_SEGUNDA_LINHA)
lcd_data('N deve aparecer')
waitForResult('a palavra "Teste" na primeira linha do display e nada na
    segunda')

lcd_config(ConfigOperation.LIMPAR_DISPLAY)
waitForResult('a linha deve ter sido limpa')

lcd_config(ConfigOperation.LIMPAR_DISPLAY)
lcd_config(ConfigOperation.MODO_DUAS_LINHAS)
lcd_data("Teste")
lcd_config(ConfigOperation.MOVER_CURSOR_SEGUNDA_LINHA)
lcd_data("Teste2")
waitForResult('a palavra "Teste" na primeira linha do display e "Teste2"
    na segunda')

lcd_config(ConfigOperation.LIMPAR_SEGUNDA_LINHA)
waitForResult('a segunda linha deve ter sido limpa')

lcd_config(ConfigOperation.MOVER_CURSOR_SEGUNDA_LINHA)
lcd_data("Teste2")
waitForResult('a segunda linha deve ter sido escrita novamente')
```

```
34
35 lcd_config(ConfigOperation.LIMPAR_PRIMEIRA_LINHA)
36 waitForResult('a primeira linha deve ter sido limpa')
37
38 lcd_config(ConfigOperation.LIMPAR_DISPLAY)
39 lcd_config(ConfigOperation.CURSOR_NAO_VISIVEL)
40 waitForResult('O cursor não deve ser visível')
41
42 lcd_config(ConfigOperation.LIMPAR_DISPLAY)
43 lcd_config(ConfigOperation.CURSOR_VISIVEL)
44 waitForResult('O cursor deve ser visível')
45
46 lcd_config(ConfigOperation.LIMPAR_DISPLAY)
```

APÊNDICE C – Códigos utilizados no *driver* de dispositivo do módulo RFID-RC522

C.1 Overlay para árvore de dispositivos utilizado para inserir o dispositivo RFID-RC522 e desativar os dispositivos spidev

```
/dts-v1/;
/plugin/;
/{
    fragment@0 {
        target = <&spidev0>;
        __overlay__ {
            status = "disabled";
        };
    };

    fragment@1 {
        target = <&spidev1>;
        __overlay__ {
            status = "disabled";
        };
    };

    fragment@2 {
        target = <&spi0>;
        __overlay__ {
            status = "okay";

            rc522: rc522@0 {
                compatible = "mytest,rc522";
                status = "okay";
                reg = <0x0>;
                spi-max-frequency = <4000000>;
            };
         };
    };
};
```

C.2 Arquivo *Makefile* para compilação do *driver* de dispositivo para o dispositivo RFID-RC522

```
obj-m := rfid_rc522_device_driver.o
all:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
msg:
  dmesg | tail -20
```

C.3 Arquivo de código do *header* do *driver* de dispositivo para o dispositivo RFID-RC522

```
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/types.h>
#include "rfid_rc522_drive_ioctl.h"

#define DEVICE_NAME "rfid_rc522_driver"
#define CLASS_NAME  "rfid_rc522_driver_class"

#define SPI_CHAN 0
#define RC522_RST_PIN 537

#define MSG_OK(s) printk(KERN_INFO "%s: %s\n", DEVICE_NAME, s)
#define MSG_BAD(s, err_val) printk(KERN_ERR "%s: %s %ld\n", DEVICE_NAME,
    s, err_val)

#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte)  \
  ((byte) & 0x80 ? '1' : '0'), \
  ((byte) & 0x40 ? '1' : '0'), \
  ((byte) & 0x20 ? '1' : '0'), \
  ((byte) & 0x10 ? '1' : '0'), \
  ((byte) & 0x08 ? '1' : '0'), \
  ((byte) & 0x04 ? '1' : '0'), \
  ((byte) & 0x02 ? '1' : '0'), \
  ((byte) & 0x01 ? '1' : '0')

typedef enum mfrc522_registers {
    // Command and status registers
    Reserved00 = 0x00,
    CommandReg = 0x01,
    ComlEnReg = 0x02,
    DivlEnReg = 0x03,
    ComIrqReg = 0x04,
    DivIrqReg = 0x05,
    ErrorReg = 0x06,
```

```
35      Status1Reg = 0x07,
36      Status2Reg = 0x08,
37      FIFODataReg = 0x09,
38      FIFOLevelReg = 0x0A,
39      WaterLevelReg = 0x0B,
40      ControlReg = 0x0C,
41      BitFramingReg = 0x0D,
42      CollReg = 0x0E,
43      Reserved0F = 0x0F,
44      // Commands registers
45      Reserved10 = 0x10,
46      ModeReg = 0x11,
47      TxModeReg = 0x12,
48      RxModeReg = 0x13,
49      TxControlReg = 0x14,
50      TxASKReg = 0x15,
51      TxSelReg = 0x16,
52      RxSelReg = 0x17,
53      RxThresholdReg = 0x18,
54      DemodReg = 0x19,
55      Reserved1A = 0x1A,
56      Reserved1B = 0x1B,
57      MfTxReg = 0x1C,
58      MfRxReg = 0x1D,
59      Reserved1E = 0x1E,
60      SerialSpeedReg = 0x1F,
61      // Configuration registers
62      CRCResultRegMSB = 0x21,
63      CRCResultRegLSB = 0x22,
64      Reserved23 = 0x23,
65      ModWidthReg = 0x24,
66      Reserved25 = 0x25,
67      RFCfgReg = 0x26,
68      GsNReg = 0x27,
69      CWGsPReg = 0x28,
70      ModGsPReg = 0x29,
71      TModeReg = 0x2A,
72      TPrescalerReg = 0x2B,
73      TReloadRegMSB = 0x2C,
74      TReloadRegLSB = 0x2D,
75      TCounterValRegMSB = 0x2E,
76      TCounterValRegLSB = 0x2F,
77      // Test registers
78      Reserved30 = 0x30,
79      TestSel1Reg = 0x31,
80      TestSel2Reg = 0x32,
81      TestPinEnReg = 0x33,
```

```
82      TestPinValueReg = 0x34,
83      TestBusReg = 0x35,
84      AutoTestReg = 0x36,
85      VersionReg = 0x37,
86      AnalogTestReg = 0x38,
87      TestDAC1Reg = 0x39,
88      TestDAC2Reg = 0x3A,
89      Reserved3C = 0x3C,
90      Reserved3D = 0x3D,
91      Reserved3E = 0x3E,
92      Reserved3F = 0x3F,
93  } mfrc522_registers;
94
95  typedef enum adress_bit_mode {
96      address_read_mode = 1,
97      address_write_mode = 0,
98  } adress_bit_mode;
99
100 typedef enum rc522_commands {
101     Idle = 0b0000,
102     Mem = 0b0001,
103     Generate_RandomID = 0b0010,
104     CalcCRC = 0b0011,
105     Transmit = 0b0100,
106     NoCmdChange = 0b0111,
107     Receive = 0b1000,
108     Transceive = 0b1100,
109     Reserved = 0b1101,
110     MFAuthent = 0b1110,
111     SoftReset = 0b1111,
112 } rc522_commands;
113
114 typedef enum picc_transceive_commands {
115     PICC_REQA = 0x26,
116     PICC_WUPA = 0x52,
117     PICC_ANTICOLL = 0x93,
118     PICC_SElECTTAG = 0x93,
119     PICC_READ = 0x30,
120     PICC_WRITE = 0xA0,
121     PICC_DECREMENT = 0xC0,
122     PICC_INCREMENT = 0xC1,
123     PICC_RESTORE = 0xC2,
124     PICC_TRANSFER = 0xB0,
125     PICC_HALT = 0x50,
126     PICC_AUTHENT1A = 0x60,
127     PICC_AUTHENT1B = 0x61,
128 } picc_transceive_commands;
```

```
129
130 typedef enum rc522_cleanup_level {
131     RC522_CLEAN_MAJOR,
132     RC522_CLEAN_CLASS,
133     RC522_CLEAN_DEVICE,
134     RC522_CLEAN_ALL,
135 } rc522_cleanup_level;
136
137 typedef struct rc522_device_manager
138 {
139     int Device_Open;    // Device aberto? Usado para prevenir acesso
        multiplo ao device
140     struct device* driver_device;
141     struct file_operations fops;
142     dev_t dev_number;
143 } rc522_device_manager;
144
145 void rc522_clean(rc522_cleanup_level level);
146
147 int rc522_char_device_open(struct inode *inode, struct file *file);
148 int rc522_char_device_release(struct inode *inode, struct file *file);
149 long int rc522_char_device_ioctl(struct file *file, unsigned cmd,
        unsigned long arg);
150
151 void rc522_reset(void);
152 void antenna_on(void);
153 uint8_t rc522_pcd_setup(void);
154 void rc522_self_test(void);
155
156 uint8_t format_address_to_byte(mfrc522_registers reg, adress_bit_mode
        mode);
157 void write_to_register(mfrc522_registers reg, uint8_t data);
158 void write_to_register_multiple(mfrc522_registers reg, uint8_t *data,
        size_t data_size);
159 uint8_t read_from_register(mfrc522_registers reg);
160 void read_from_register_multiple(mfrc522_registers reg, uint8_t *res,
        size_t num_reads);
161 void clear_bits_in_reg(mfrc522_registers reg, uint8_t bits_to_set);
162 void set_bits_in_reg(mfrc522_registers reg, uint8_t bits_to_set);
163
164 int isCrcOperationDone(void);
165 uint8_t is_crc_from_picc_valid(uint8_t *data, uint8_t data_size);
166 rc522_status send_command(uint8_t command, uint8_t *data, size_t
        data_size, uint8_t *response, uint8_t *response_size, uint8_t *
        response_size_bits, uint8_t should_check_crc, uint8_t bit_framing);
167 rc522_status calculate_crc(uint8_t *data, size_t data_size, uint8_t *
        result);
```

```
168 rc522_status req_a_picc(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits);
169 rc522_status anticollision(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits);
170 rc522_status select_tag(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, uint8_t *uid);
171 rc522_status authenticate(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, picc_transceive_commands auth_mode, uint8_t
    block_address, uint8_t *sector_key, uint8_t sector_key_size, uint8_t
    *uid);
172 void stop_authentication(void);
173 rc522_status read_block(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, uint8_t blockAddr);
174 rc522_status write_block(uint8_t *data, uint8_t data_size, uint8_t *res,
     uint8_t *res_size, uint8_t *res_size_bits, uint8_t blockAddr);
```

C.4 Arquivo de código do *driver* de dispositivo para o dispositivo RFID-RC522

```
1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/spi/spi.h>
5  #include <linux/of.h>
6  #include <linux/gpio.h>
7  #include <linux/slab.h>
8  #include <linux/delay.h>
9
10 #include "rfid_rc522_device_driver.h"
11
12 static struct class*  rc522_char_dev_class  = NULL;
13 static struct spi_device *rc522_spi_dev;
14
15 static struct gpio rc522_rst_pin = {  RC522_RST_PIN, GPIOF_OUT_INIT_HIGH
    , "EN" };
16
17 rc522_device_manager rc522_char_device = {
18   .Device_Open = 0,
19   .fops = {
20         .unlocked_ioctl = rc522_char_device_ioctl,
21     .open = rc522_char_device_open,
22     .release = rc522_char_device_release
23   },
24   .driver_device = NULL,
25   .dev_number = 0,
26 };
```

```
27
28 uint8_t format_address_to_byte(mfrc522_registers reg, adress_bit_mode
       mode) {
29     if(mode == address_write_mode)
30         return (uint8_t) (reg << 1) & 0x7E;
31     return (uint8_t) ((reg << 1) & 0x7E) | 0x80;
32 }
33
34 void write_to_register(mfrc522_registers reg, uint8_t data) {
35     uint8_t *message = kmalloc(2, GFP_KERNEL);
36     message[0] = format_address_to_byte(reg, address_write_mode);
37     message[1] = data;
38
39     struct spi_transfer t = {
40         .tx_buf = message,
41         .len = 2,
42     };
43
44     struct spi_message spi_m;
45     spi_message_init(&spi_m);
46     spi_message_add_tail(&t, &spi_m);
47
48     int ret = spi_sync(rc522_spi_dev, &spi_m);
49
50     if (ret < 0) {
51         printk(KERN_ERR "Erro ao enviar dados via SPI\n");
52     }
53     kfree(message);
54 }
55
56 void write_to_register_multiple(mfrc522_registers reg, uint8_t *data,
       size_t data_size) {
57     uint8_t *buffer = kmalloc(data_size+1, GFP_KERNEL);// byte0 -
       address, bytes 1-n - data
58     memcpy(buffer+1, data, data_size);
59     buffer[0] = format_address_to_byte(reg, address_write_mode);
60
61     struct spi_transfer t = {
62         .tx_buf = buffer,
63         .len = data_size + 1,
64     };
65
66     struct spi_message spi_m;
67     spi_message_init(&spi_m);
68     spi_message_add_tail(&t, &spi_m);
69
70     int ret = spi_sync(rc522_spi_dev, &spi_m);
```

```
71
72      if (ret < 0) {
73          printk(KERN_ERR "Erro ao enviar dados via SPI\n");
74      }
75
76      kfree(buffer);
77  }
78
79  uint8_t read_from_register(mfrc522_registers reg) {
80      uint8_t *message_res = kmalloc(2, GFP_KERNEL);
81      uint8_t *message = kmalloc(2, GFP_KERNEL);
82      message[0] = format_address_to_byte(reg, address_read_mode);
83      message[1] = 0x00;
84
85      struct spi_transfer t = {
86          .tx_buf = message,
87          .rx_buf = message_res,
88          .len = 2,
89      };
90
91      struct spi_message spi_m;
92      spi_message_init(&spi_m);
93      spi_message_add_tail(&t, &spi_m);
94
95      int ret = spi_sync(rc522_spi_dev, &spi_m);
96
97      if (ret < 0) {
98          printk(KERN_ERR "Erro ao ler dados via SPI\n");
99          return ret;
100     }
101     uint8_t res = message_res[1];
102     kfree(message_res);
103     kfree(message);
104     return res;
105 }
106
107 void read_from_register_multiple(mfrc522_registers reg, uint8_t *res,
        size_t num_reads) {
108     uint8_t *req_buffer = kmalloc(num_reads+1, GFP_KERNEL);
109     uint8_t *res_buffer = kmalloc(num_reads+1, GFP_KERNEL);
110     uint8_t reg_formatted = format_address_to_byte(reg,
        address_read_mode);
111
112     for (size_t i = 0; i < num_reads; i++)
113     {
114         req_buffer[i] = reg_formatted;
115     }
```

```
116     req_buffer[num_reads] = 0x00;
117 
118     struct spi_transfer t = {
119         .tx_buf = req_buffer,
120         .rx_buf = res_buffer,
121         .len = num_reads+1,
122     };
123 
124     struct spi_message spi_m;
125     spi_message_init(&spi_m);
126     spi_message_add_tail(&t, &spi_m);
127 
128     int ret = spi_sync(rc522_spi_dev, &spi_m);
129 
130     if (ret < 0) {
131         printk(KERN_ERR "Erro ao ler dados via SPI\n");
132     }
133 
134     memcpy(res, res_buffer + 1, num_reads);
135 
136     kfree(req_buffer);
137     kfree(res_buffer);
138 }
139 
140 void set_bits_in_reg(mfrc522_registers reg, uint8_t bits_to_set) {
141     uint8_t reg_current_byte = read_from_register(reg);
142     write_to_register(reg, reg_current_byte | bits_to_set);
143 }
144 
145 void clear_bits_in_reg(mfrc522_registers reg, uint8_t bits_to_set) {
146     uint8_t reg_current_byte = read_from_register(reg);
147     write_to_register(reg, reg_current_byte & (~bits_to_set));
148 }
149 
150 int isCrcOperationDone(void) {
151     uint8_t divIrqRegByte = read_from_register(DivIrqReg);
152     return divIrqRegByte & 0x04;
153 }
154 
155 rc522_status calculate_crc(uint8_t *data, size_t data_size, uint8_t *
    result) {
156     uint16_t i;
157     write_to_register(CommandReg, Idle);
158     clear_bits_in_reg(DivIrqReg, 0x04);
159     set_bits_in_reg(FIFOLevelReg, 0x80);
160 
161     //write the data to FIFO Buffer
```

```
162     write_to_register_multiple(FIFODataReg, data, data_size);
163 
164     write_to_register(CommandReg, CalcCRC);
165 
166     i = 5000;
167     while (i>0) {
168         if(isCrcOperationDone()) {
169             write_to_register(CommandReg, Idle);
170             result[0] = read_from_register(CRCResultRegLSB);
171             result[1] = read_from_register(CRCResultRegMSB);
172             return RC522_OK;
173         }
174         i--;
175     }
176     write_to_register(CommandReg, Idle);
177     return RC522_TIMEOUT;
178 }
179 
180 uint8_t is_crc_from_picc_valid(uint8_t *data, uint8_t data_size) {
181     uint8_t result[2];
182     uint8_t data_size_without_crc = data_size - 2;
183     uint8_t crc_position = data_size - 2;
184 
185     calculate_crc(data, data_size_without_crc, result);
186 
187     if(data[crc_position] != result[0] || data[crc_position+1] != result
    [1]) return 0;
188     return 1;
189 }
190 
191 uint8_t is_uid_valid_picc(uint8_t *uid) {
192     uint8_t check = 0;
193     for (size_t i = 0; i < 4; i++)
194     {
195         check = check ^ uid[i];
196     }
197     return check == uid[4];
198 }
199 
200 void rc522_reset(void) {
201     write_to_register(CommandReg, SoftReset);
202     uint8_t countTries = 0;
203     do {
204         msleep(50);
205     } while((read_from_register(CommandReg) & (1 << 4)) && ((++
    countTries) < 3));
206 }
```

```
207
208 void rc522_antenna_on(void)
209 {
210   uint8_t txControlRegByte = read_from_register(TxControlReg);
211   if ((txControlRegByte & 0b00000011) != 0b00000011) {
212     set_bits_in_reg(TxControlReg, 0b00000011);
213   }
214 }
215
216 uint8_t rc522_pcd_setup(void) {
217     rc522_reset();
218
219     write_to_register(TxModeReg, 0x00);
220     write_to_register(RxModeReg, 0x00);
221     write_to_register(ModWidthReg, 0x26);
222     //configure timeout for communication with PICC
223     write_to_register(TModeReg, 0x80);
224     write_to_register(TPrescalerReg, 0xA9);
225     write_to_register(TReloadRegMSB, 0x03);
226     write_to_register(TReloadRegLSB, 0xE8);
227
228     write_to_register(TxASKReg, 0x40); // 100% ASK modulation
229     write_to_register(ModeReg, 0x3D);
230     rc522_antenna_on();
231
232     usleep_range(4500, 5500);
233
234     return read_from_register(VersionReg);
235 }
236
237 void rc522_self_test(void) {
238     rc522_reset();
239
240     uint8_t *buffer = kmalloc(64, GFP_KERNEL);
241     for (uint8_t i = 0; i < 25; i++)
242     {
243         buffer[i] = 0x00;
244     }
245
246     //write the data to FIFO Buffer
247     write_to_register_multiple(FIFODataReg, buffer, 25);
248
249     set_bits_in_reg(FIFOLevelReg, 0x80);
250
251     write_to_register(AutoTestReg, 0x09);
252     write_to_register(FIFODataReg, 0x00);
253     write_to_register(CommandReg, CalcCRC);
```

```
254     usleep_range(15000, 25000);
255     read_from_register_multiple(FIFODataReg, buffer, 64);
256     for (size_t i = 0; i < 8; i++)
257     {
258         printk(
259             KERN_INFO "0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x
    0x%.2x",
260             buffer[i*8],
261             buffer[(i*8)+1],
262             buffer[(i*8)+2],
263             buffer[(i*8)+3],
264             buffer[(i*8)+4],
265             buffer[(i*8)+5],
266             buffer[(i*8)+6],
267             buffer[(i*8)+7]
268         );
269     }
270     rc522_reset();
271     kfree(buffer);
272 }
273
274 rc522_status send_command(uint8_t command, uint8_t *data, size_t
    data_size, uint8_t *response, uint8_t *response_size, uint8_t *
    valid_bits, uint8_t should_check_crc, uint8_t bit_framing) {
275     uint8_t irqEn = 0x00;
276     uint8_t waitIrq = 0x00;
277     uint8_t numFIFOBytes = 0;
278
279     //uint8_t tx_last_bits = valid_bits ? valid_bits : 0;
280     //uint8_t bit_framing = tx_last_bits;
281
282     if(command == MFAuthent) {
283         irqEn = 0x12;
284         waitIrq = 0x10;
285     }
286
287     if(command == Transceive) {
288         irqEn = 0x77;
289         waitIrq = 0x30;
290     }
291     //write_to_register(ComlEnReg, irqEn | 0x80);
292     clear_bits_in_reg(ComIrqReg, 0x80);
293     set_bits_in_reg(FIFOLevelReg, 0x80);
294     write_to_register(BitFramingReg, bit_framing);
295     write_to_register(CommandReg, Idle);
296     //write_to_register(BitFramingReg, 0x07);
297
```

```
298     write_to_register_multiple(FIFODataReg, data, data_size);
299     usleep_range(3000, 3001);
300 
301     for (size_t i = 0; i < data_size; i++)
302     {
303         printk(KERN_CONT "0x%x ", data[i]);
304     }
305 
306 
307     write_to_register(CommandReg, command);
308     if(command == Transceive) {
309         set_bits_in_reg(BitFramingReg, 0b10000000);
310     }
311 
312     int i = 2000;
313 
314     while (1) {
315         uint8_t comIrqRegByte = read_from_register(ComIrqReg);
316         if (i-- < 1 || comIrqRegByte & waitIrq) break;
317     }
318 
319     clear_bits_in_reg(BitFramingReg, 0x80);
320     if(command == Transceive) {
321         write_to_register(CommandReg, Idle);
322     }
323 
324     //if(i < 1) return RC522_TIMEOUT;
325 
326 
327     uint8_t errorRegValue = read_from_register(ErrorReg); // ErrorReg
    [7..0] bits are: WrErr TempErr reserved BufferOvfl CollErr CRCErr
    ParityErr ProtocolErr
328     if (errorRegValue & 0x13)
329     {  // BufferOvfl ParityErr ProtocolEr
330         printk(KERN_ERR"Error detected, errorRegValue: %x\n",
    errorRegValue);
331         return RC522_ERR;
332     }
333     if(read_from_register(ComIrqReg) & irqEn & 0x01) return
    RC522_NOTAGERR;
334 
335     numFIFOBytes = read_from_register(FIFOLevelReg);
336 
337     uint8_t last_bits = read_from_register(ControlReg) & 0b00000111;
338     if(numFIFOBytes > 0) {
339         uint8_t *buffer = kmalloc(numFIFOBytes, GFP_KERNEL);
340         read_from_register_multiple(FIFODataReg, buffer, numFIFOBytes);
```

```
341         if (numFIFOBytes > *response_size) {
342             kfree(buffer);
343             printk(KERN_WARNING "RC522_BUFFER_TOO_SMALL");
344             return RC522_BUFFER_TOO_SMALL;
345         }
346         memcpy(response, buffer, numFIFOBytes);
347         *response_size = numFIFOBytes;
348         *valid_bits = last_bits ? ((*response_size) - 1) * 8 + last_bits
    : (*response_size)*8;
349         kfree(buffer);
350     }
351 
352     if(!should_check_crc) return RC522_OK;
353 
354     if (*response_size < 3 || !is_crc_from_picc_valid(response, *
    response_size)) return RC522_FAILED_CRC_CHECK;
355 
356     return RC522_OK;
357 }
358 
359 rc522_status req_a_picc(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits) {
360     uint8_t req_mode = PICC_REQA;
361     clear_bits_in_reg(CollReg, 0x80);
362 
363     rc522_status status = send_command(Transceive, &req_mode, 1, res,
    res_size, res_size_bits, 0, 0x07);
364 
365     if(status != RC522_OK) return status;
366     if(*res_size_bits != 0x10) return RC522_ERR;
367     return RC522_OK;
368 }
369 
370 rc522_status anticollision(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits) {
371     uint8_t data[2] = { PICC_ANTICOLL, 0x20 };
372 
373     rc522_status status = send_command(Transceive, data, 2, res,
    res_size, res_size_bits, 0, 0);
374 
375     if(status != RC522_OK) return status;
376     if(*res_size != 5) return RC522_ERR;
377     if(!is_uid_valid_picc(res)) return RC522_FAILED_UID_CHECK;
378 
379     return RC522_OK;
380 }
381 
```

```
382 rc522_status select_tag(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, uint8_t *uid) {
383     uint8_t *buffer = kmalloc(9, GFP_KERNEL);
384     buffer[0] = PICC_SElECTTAG;
385     buffer[1] = 0x70;
386     memcpy(buffer+2, uid, 5);
387 
388     calculate_crc(buffer, 7, buffer+7);
389 
390     rc522_status status = send_command(Transceive, buffer, 9, res,
    res_size, res_size_bits, 0, 0);
391     kfree(buffer);
392 
393     if(status != RC522_OK) return status;
394     if(*res_size != 3) return RC522_ERR;
395     return RC522_OK;
396 }
397 
398 rc522_status authenticate(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, picc_transceive_commands auth_mode, uint8_t
    block_address, uint8_t *sector_key, uint8_t sector_key_size, uint8_t
    *uid) {
399     // 1 for the auth_mode and 1 for address, and 4 for the first 4
    bytes of uid
400     uint8_t buffer_size = 2 + sector_key_size + 4;
401     uint8_t *buffer = kmalloc(buffer_size, GFP_KERNEL);
402     buffer[0] = auth_mode;
403     buffer[1] = block_address;
404     memcpy(buffer+2, sector_key, sector_key_size);
405     memcpy(buffer+2+sector_key_size, uid, 4);
406 
407     rc522_status status = send_command(MFAuthent, buffer, buffer_size,
    res, res_size, res_size_bits, 0, 0);
408     kfree(buffer);
409 
410     if(status != RC522_OK) return status;
411     if(!(read_from_register(Status2Reg) & 0x08)) return RC522_ERR;
412     return RC522_OK;
413 }
414 
415 void stop_authentication(void) {
416     clear_bits_in_reg(Status2Reg, 0x08);
417 }
418 
419 rc522_status read_block(uint8_t *res, uint8_t *res_size, uint8_t *
    res_size_bits, uint8_t blockAddr) {
420     uint8_t *buffer = kmalloc(4, GFP_KERNEL);
```

```
421     rc522_status status;
422     buffer[0] = PICC_READ;
423     buffer[1] = blockAddr;
424     status = calculate_crc(buffer, 2, buffer+2);
425     if(status != RC522_OK) return status;
426 
427     status = send_command(Transceive, buffer, 4, res, res_size,
    res_size_bits, 1, 0);
428     kfree(buffer);
429     if(status != RC522_OK) return status;
430 
431     return RC522_OK;
432 }
433 
434 rc522_status write_block(uint8_t *data, uint8_t data_size, uint8_t *res,
     uint8_t *res_size, uint8_t *res_size_bits, uint8_t blockAddr) {
435     uint8_t *buffer = kmalloc(4, GFP_KERNEL), buffer_res_size = 1,
    buffer_res_size_bits;
436     uint8_t *buffer_res  = kmalloc(buffer_res_size, GFP_KERNEL);
437     uint8_t *data_buffer;
438     rc522_status status;
439     buffer[0] = PICC_WRITE;
440     buffer[1] = blockAddr;
441     status = calculate_crc(buffer, 2, &buffer[2]);
442     if(status != RC522_OK) {
443         kfree(buffer);
444         kfree(buffer_res);
445         return status;
446     }
447 
448     status = send_command(Transceive, buffer, 4, buffer_res, &
    buffer_res_size, &buffer_res_size_bits, 0, 0);
449     kfree(buffer);
450     kfree(buffer_res);
451 
452     if(status != RC522_OK) {
453         return status;
454     }
455     data_buffer = kmalloc(18, GFP_KERNEL);
456     if(data_size < 16) {
457         memcpy(data_buffer, data, data_size);
458         for (uint8_t i = data_size; i < 16; i++)
459         {
460             data_buffer[i] = ' ';
461         }
462     }
463     else {
```

```
464         memcpy(data_buffer, data, 16);
465     }
466     status = calculate_crc(data_buffer, 16, data_buffer+16);
467     if(status != RC522_OK) {
468         kfree(data_buffer);
469         return status;
470     }
471     status = send_command(Transceive, data_buffer, 18, res, res_size,
    res_size_bits, 0, 0);
472     kfree(data_buffer);
473     //if(*res_size != 16) return RC522_ERR;
474     return status;
475 }
476
477
478 static int rc522_spi_probe(struct spi_device *spi)
479 {
480     int ret, result;
481     printk(KERN_INFO "Probe Chamado\n");
482     rc522_spi_dev = spi;
483
484     result = register_chrdev(0, DEVICE_NAME, &rc522_char_device.fops);
485   if(result < 0)
486   {
487     MSG_BAD("Erro ao registrar driver de caracter", (long int)result);
488     return result;
489   }
490     rc522_char_device.dev_number = MKDEV(result, 0);
491
492     rc522_char_dev_class = class_create(CLASS_NAME);
493   if(IS_ERR(rc522_char_dev_class))
494   {
495     rc522_clean(RC522_CLEAN_MAJOR);
496     MSG_BAD("Falhou em registrar a classe do dispositivo", PTR_ERR(
    rc522_char_dev_class));
497     return PTR_ERR(rc522_char_dev_class);
498   }
499   MSG_OK("classe do dispositivo registrada corretamente");
500
501     rc522_char_device.driver_device = device_create(rc522_char_dev_class
    , NULL, rc522_char_device.dev_number, NULL, DEVICE_NAME);
502   if(IS_ERR(rc522_char_device.driver_device)) // Se houve erro no
    registro
503   {
504     rc522_clean(RC522_CLEAN_CLASS);
505     MSG_BAD("falhou em criar o device driver", PTR_ERR(rc522_char_device
    .driver_device));
```

```
506     return PTR_ERR(rc522_char_device.driver_device);
507   }
508   MSG_OK("dispositivo criado corretamente");
509 
510     printk(KERN_INFO "Iniciando GPIO Reset\n");
511     ret = gpio_request(rc522_rst_pin.gpio, rc522_rst_pin.label);
512     if(ret) {
513     rc522_clean(RC522_CLEAN_DEVICE);
514     MSG_BAD("Não foi possível obter acesso ao GPIO.", (long int)ret);
515         return ret;
516     }
517     printk(KERN_INFO "GPIO Reset iniciado, setando como output com valor
        1\n");
518     gpio_direction_output(rc522_rst_pin.gpio, 1);
519     printk(KERN_INFO "GPIO Reset configurado com sucesso\n");
520 
521     uint8_t version = rc522_pcd_setup();
522     printk(KERN_INFO "Version byte 0x%x\n", version);
523 
524     return 0;
525 }
526 
527 static void rc522_spi_remove(struct spi_device *spi)
528 {
529     rc522_clean(RC522_CLEAN_ALL);
530 }
531 
532 static const struct spi_device_id rc522_spi_idtable[] = {
533     {"rc522", 2},
534     {},
535 };
536 
537 MODULE_DEVICE_TABLE(spi, rc522_spi_idtable);
538 
539 static const struct of_device_id rc522_spi_of_match[] = {
540     {
541         .compatible = "mytest,rc522",
542     },
543     {},
544 };
545 
546 MODULE_DEVICE_TABLE(of, rc522_spi_of_match);
547 
548 static struct spi_driver rc522_spi_driver = {
549     .driver = {
550         .name = "rc522_spi_device",
551         .owner = THIS_MODULE,
```

```
552         .of_match_table = of_match_ptr(rc522_spi_of_match),
553     },
554     .probe = rc522_spi_probe,
555     .remove = rc522_spi_remove,
556     .id_table = rc522_spi_idtable,
557 };
558 
559 int rc522_char_device_open(struct inode *inode, struct file *file)
560 {
561     if(rc522_char_device.Device_Open) return -EBUSY;
562   rc522_char_device.Device_Open++;
563   try_module_get(THIS_MODULE);
564   return 0;
565 }
566 
567 int rc522_char_device_release(struct inode *inode, struct file *file)
568 {
569   rc522_char_device.Device_Open--; // Libera acesso ao device
570   module_put(THIS_MODULE); // Decrementa o contador de uso do modulo
571   return 0;
572 }
573 
574 
575 void ioctl_read_from_register(unsigned cmd, unsigned long arg) {
576     uint8_t addr, res;
577     if(copy_from_user(&addr, (int32_t *) arg, sizeof(addr))) {
578         printk(KERN_ERR "IOCTL_RC522_READ_REGISTER: Error copying data
    from user space\n");
579     }
580     res = read_from_register(addr);
581     if(copy_to_user((int32_t *) arg, &res, sizeof(res)))
582         printk(KERN_ERR "IOCTL_RC522_READ_REGISTER: failed to write res
    to user space\n");
583 }
584 
585 void ioctl_write_to_register(unsigned cmd, unsigned long arg) {
586     struct rfid_rc522_write_register_dto dto;
587     if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto)))
588         printk(KERN_ERR "IOCTL_RC522_WRITE_REGISTER: Error copying data
    from user space\n");
589     write_to_register(dto.addr, dto.data);
590 }
591 
592 void ioctl_write_to_register_multiple(unsigned cmd, unsigned long arg) {
593     struct rfid_rc522_write_multiple_register_dto dto;
594     if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto)))
595         printk(KERN_ERR "IOCTL_RC522_WRITE_REGISTER: Error copying data
```

```
    from user space\n");
596     write_to_register_multiple(dto.addr, dto.data, dto.data_size);
597 }
598
599 void ioctl_req_a_picc(unsigned cmd, unsigned long arg) {
600     uint8_t res_size = 2, res_size_bits = 0, *res = kmalloc(res_size,
    GFP_KERNEL);
601     struct rfid_rc522_req_a_picc_dto dto;
602
603     dto.status = req_a_picc(res, &res_size, &res_size_bits);
604
605     memcpy(dto.res, res, res_size);
606     dto.res_size = res_size;
607     kfree(res);
608
609     if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
610         printk(KERN_INFO "IOCTL_RC522_REQ_A_PICC: failed to write
    response to user space\n");
611 }
612
613 void ioctl_anticollision(unsigned cmd, unsigned long arg) {
614     uint8_t res_size = 5, res_size_bits = 0, *res = kmalloc(res_size,
    GFP_KERNEL);
615     struct rfid_rc522_anticollision_dto dto;
616     dto.status = anticollision(res, &res_size, &res_size_bits);
617
618     memcpy(dto.uid, res, res_size);
619     dto.res_size = res_size;
620     kfree(res);
621
622     if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
623         printk(KERN_INFO "IOCTL_RC522_ANTICOLLISION: failed to write
    response to user space\n");
624 }
625
626 void ioctl_select_tag(unsigned cmd, unsigned long arg) {
627     uint8_t res_size = 3, res_size_bits = 0, *res = kmalloc(res_size,
    GFP_KERNEL);
628     struct rfid_rc522_select_tag_dto dto;
629     if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto))) {
630         printk(KERN_ERR "IOCTL_RC522_SELECT_TAG: Error copying data from
     user space\n");
631         return;
632     }
633
634     dto.status = select_tag(res, &res_size, &res_size_bits, dto.uid);
635
```

```
636     memcpy(dto.res, res, res_size);
637     dto.res_size = res_size;
638     kfree(res);
639 
640     if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
641         printk(KERN_ERR "IOCTL_RC522_SELECT_TAG: failed to write
    response to user space\n");
642 }
643 
644 void ioctl_authenticate(unsigned cmd, unsigned long arg) {
645     uint8_t res_size = 1, res_size_bits = 0, *res = kmalloc(res_size,
    GFP_KERNEL);
646     struct rfid_rc522_authenticate_dto dto;
647     if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto))) {
648         printk(KERN_ERR "IOCTL_RC522_AUTHENTICATE: Error copying data
    from user space\n");
649         return;
650     }
651 
652     dto.status = authenticate(res, &res_size, &res_size_bits,
    PICC_AUTHENT1A, dto.block_address, dto.sector_key, 6, dto.uid);
653     kfree(res);
654 
655     if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
656         printk(KERN_ERR "IOCTL_RC522_AUTHENTICATE: failed to write
    response to user space\n");
657 }
658 
659 void ioctl_read_picc_block(unsigned cmd, unsigned long arg) {
660     uint8_t res_size = 18, res_size_bits = 0, *res = kmalloc(res_size,
    GFP_KERNEL);
661     struct rfid_rc522_read_picc_block_dto dto;
662     if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto))) {
663         printk(KERN_ERR "IOCTL_RC522_READ_PICC_BLOCK: Error copying data
     from user space\n");
664         return;
665     }
666 
667     dto.status = read_block(res, &res_size, &res_size_bits, dto.
    block_address);
668 
669     memcpy(dto.res, res, res_size > 16 ? 16 : res_size);
670     dto.res_size = res_size > 16 ? 16 : res_size;
671 
672     kfree(res);
673     if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
674         printk(KERN_ERR "IOCTL_RC522_READ_PICC_BLOCK: failed to write
```

```
    response to user space\n");
}

void ioctl_write_picc_block(unsigned cmd, unsigned long arg) {
    uint8_t res_size = 1, res_size_bits = 0, res;
    struct rfid_rc522_write_picc_block_dto dto;
    if(copy_from_user(&dto, (int32_t *) arg, sizeof(dto))) {
        printk(KERN_ERR "IOCTL_RC522_WRITE_PICC_BLOCK: Error copying
    data from user space\n");
        return;
    }

    dto.status = write_block(dto.input, 16, &res, &res_size, &
    res_size_bits, dto.block_address);

    if(copy_to_user((int32_t *) arg, &dto, sizeof(dto)))
        printk(KERN_ERR "IOCTL_RC522_WRITE_PICC_BLOCK: failed to write
    response to user space\n");
}

void ioctl_stop_auth(unsigned cmd, unsigned long arg) {
    stop_authentication();
}

long int rc522_char_device_ioctl(struct file *file, unsigned cmd,
    unsigned long arg){
  switch(cmd){
    case IOCTL_RC522_READ_REGISTER:
            ioctl_read_from_register(cmd, arg);
      break;
    case IOCTL_RC522_WRITE_REGISTER:
      ioctl_write_to_register(cmd, arg);
      break;
        case IOCTL_RC522_WRITE_REGISTER_MULTIPLE:
      ioctl_write_to_register_multiple(cmd, arg);
      break;
        case IOCTL_RC522_REQ_A_PICC:
      ioctl_req_a_picc(cmd, arg);
      break;
        case IOCTL_RC522_ANTICOLLISION:
      ioctl_anticollision(cmd, arg);
      break;
        case IOCTL_RC522_SELECT_TAG:
      ioctl_select_tag(cmd, arg);
      break;
        case IOCTL_RC522_AUTHENTICATE:
      ioctl_authenticate(cmd, arg);
```

```
717         break;
718           case IOCTL_RC522_READ_PICC_BLOCK:
719         ioctl_read_picc_block(cmd, arg);
720         break;
721           case IOCTL_RC522_WRITE_PICC_BLOCK:
722         ioctl_write_picc_block(cmd, arg);
723         break;
724           case IOCTL_RC522_STOP_AUTH:
725         ioctl_stop_auth(cmd, arg);
726         break;
727   }
728   return 0;
729 }
730
731 void rc522_clean(rc522_cleanup_level level)
732 {
733     if(level >= RC522_CLEAN_ALL)
734   {
735     gpio_set_value(rc522_rst_pin.gpio, 0);
736         gpio_free(rc522_rst_pin.gpio);
737   }
738     if(level >= RC522_CLEAN_DEVICE)
739   {
740         device_destroy(rc522_char_dev_class, rc522_char_device.
    dev_number);
741   }
742     if(level >= RC522_CLEAN_CLASS)
743   {
744         class_unregister(rc522_char_dev_class);
745     class_destroy(rc522_char_dev_class);
746   }
747     if(level >= RC522_CLEAN_MAJOR)
748   {
749         unregister_chrdev(MAJOR(rc522_char_device.dev_number),
    DEVICE_NAME);
750   }
751 }
752
753 module_spi_driver(rc522_spi_driver);
754
755 MODULE_LICENSE("GPL v2");
756 MODULE_AUTHOR("Julio Cesar Schneider Martins <jschneiderm98@gmail.com>")
    ;
757 MODULE_DESCRIPTION("Device driver to interface with RFID RC522");
```

C.5 Arquivo de código do *header* para operações IOCTL associadas ao *driver* de dispositivo para o dispositivo RFID-RC522

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

#ifndef RC522_IOCTL_H
#define RC522_IOCTL_H

    typedef enum rc522_status {
        RC522_UNDEFINED = -1,
        RC522_OK = 0,
        RC522_NOTAGERR = 1,
        RC522_ERR = 2,
        RC522_TIMEOUT = 3,
        RC522_FAILED_UID_CHECK = 4,
        RC522_FAILED_CRC_CHECK = 5,
        RC522_COLLISION = 6,
        RC522_BUFFER_TOO_SMALL = 7,
        RC522_NACK = 0Xff,
    } rc522_status;

    struct rfid_rc522_write_register_dto
    {
        uint8_t addr;
        uint8_t data;
    };

    struct rfid_rc522_write_multiple_register_dto
    {
        uint8_t addr;
        uint8_t data[64];
        uint8_t data_size;
    };

    struct rfid_rc522_req_a_picc_dto
    {
        uint8_t res_size;
        uint8_t res[2];
        rc522_status status;
    };

    struct rfid_rc522_anticollision_dto
    {
        uint8_t uid[5];
        uint8_t res_size;
        rc522_status status;
    };
```

```
45
46      struct rfid_rc522_select_tag_dto
47      {
48          uint8_t uid[5];
49          uint8_t res_size;
50          uint8_t res[3];
51          rc522_status status;
52      };
53
54      struct rfid_rc522_authenticate_dto
55      {
56          uint8_t block_address;
57          uint8_t sector_key[6];
58          uint8_t uid[5];
59          rc522_status status;
60      };
61
62      struct rfid_rc522_read_picc_block_dto
63      {
64          uint8_t block_address;
65          uint8_t res_size;
66          uint8_t res[16];
67          rc522_status status;
68      };
69
70      struct rfid_rc522_write_picc_block_dto
71      {
72          uint8_t block_address;
73          uint8_t input[16];
74          uint8_t res[1];
75          rc522_status status;
76      };
77
78      #define IOCTL_RC522_READ_REGISTER _IOWR('k', 0xc0, uint8_t *)
79      #define IOCTL_RC522_WRITE_REGISTER _IOW('k', 0xc1, struct
    rfid_rc522_write_register_dto *)
80      #define IOCTL_RC522_WRITE_REGISTER_MULTIPLE _IOW('k', 0xc2, struct
    rfid_rc522_write_multiple_register_dto *)
81      #define IOCTL_RC522_REQ_A_PICC _IOR('k', 0xc3, struct
    rfid_rc522_req_a_picc_dto *)
82      #define IOCTL_RC522_ANTICOLLISION _IOWR('k', 0xc4, struct
    rfid_rc522_anticollision_dto *)
83      #define IOCTL_RC522_SELECT_TAG _IOWR('k', 0xc5, struct
    rfid_rc522_select_tag_dto *)
84      #define IOCTL_RC522_AUTHENTICATE _IOWR('k', 0xc6, struct
    rfid_rc522_authenticate_dto *)
85      #define IOCTL_RC522_READ_PICC_BLOCK _IOWR('k', 0xc7, struct
```

```
    rfid_rc522_read_picc_block_dto *)
86   #define IOCTL_RC522_WRITE_PICC_BLOCK _IOWR('k', 0xc8, struct
    rfid_rc522_write_picc_block_dto *)
87   #define IOCTL_RC522_STOP_AUTH _IO('k', 0xc9)
88 #endif
```

C.6 Fragmento da árvore de dispostivos antes da inserção do *overlay* para o dispositivo RC522

```
1  spi@7e204000 {
2      compatible = "brcm,bcm2835-spi";
3      clocks = <0x08 0x14>;
4      status = "okay";
5      #address-cells = <0x01>;
6      interrupts = <0x02 0x16>;
7      cs-gpios = <0x07 0x08 0x01 0x07 0x07 0x01>;
8      #size-cells = <0x00>;
9      dma-names = "tx\0rx";
10     phandle = <0x29>;
11     reg = <0x7e204000 0x200>;
12     pinctrl-0 = <0x0f 0x10>;
13     dmas = <0x0c 0x06 0x0c 0x07>;
14     pinctrl-names = "default";
15
16     spidev@1 {
17         compatible = "spidev";
18         #address-cells = <0x01>;
19         #size-cells = <0x00>;
20         phandle = <0x7a>;
21         reg = <0x01>;
22         spi-max-frequency = <0x7735940>;
23     };
24
25     spidev@0 {
26         compatible = "spidev";
27         #address-cells = <0x01>;
28         #size-cells = <0x00>;
29         phandle = <0x79>;
30         reg = <0x00>;
31         spi-max-frequency = <0x7735940>;
32     };
33 };
```

C.7 Fragmento da árvore de dispostivos depois da inserção do *overlay* para o dispositivo RC522

```
spi@7e204000 {
    compatible = "brcm,bcm2835-spi";
    clocks = <0x08 0x14>;
    status = "okay";
    #address-cells = <0x01>;
    interrupts = <0x02 0x16>;
    cs-gpios = <0x07 0x08 0x01 0x07 0x07 0x01>;
    #size-cells = <0x00>;
    dma-names = "tx\0rx";
    phandle = <0x29>;
    reg = <0x7e204000 0x200>;
    pinctrl-0 = <0x0f 0x10>;
    dmas = <0x0c 0x06 0x0c 0x07>;
    pinctrl-names = "default";

    spidev@1 {
        compatible = "spidev";
        status = "disabled";
        #address-cells = <0x01>;
        #size-cells = <0x00>;
        phandle = <0x7a>;
        reg = <0x01>;
        spi-max-frequency = <0x7735940>;
    };

    rc522@0 {
        compatible = "mytest,rc522";
        status = "okay";
        phandle = <0x9c>;
        reg = <0x00>;
        spi-max-frequency = <0x989680>;
    };

    spidev@0 {
        compatible = "spidev";
        status = "disabled";
        #address-cells = <0x01>;
        #size-cells = <0x00>;
        phandle = <0x79>;
        reg = <0x00>;
        spi-max-frequency = <0x7735940>;
    };
};
```

C.8 Arquivo de código para teste de leitura e escrita em registradores do *driver* de dispositivo para o dispositivo RFID-RC522

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <string.h>
#include <signal.h>

#include "rfid_rc522_drive_ioctl.h"

int dev;

void intHandler(int dummy) {
  close(dev);
  exit(0);
}

int main() {
  const uint8_t VERSION_REG_ADDR = 0x37;
  const uint8_t FIFO_DATA_REG_ADDR = 0x09;
  const uint8_t FIFO_LEVEL_REG_ADDR = 0x0A;
  uint8_t read_ioctl_data;
  struct rfid_rc522_write_register_dto write_dto;
  struct rfid_rc522_write_multiple_register_dto write_multiple_dto;

  dev = open("/dev/rfid_rc522_driver", O_NONBLOCK);
  if(dev == -1) {
    printf("Não foi possível abrir arquivo\n");
    return -1;
  }
  signal(SIGINT, intHandler);
  read_ioctl_data = VERSION_REG_ADDR;
  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
  printf("VersionReg Value: 0x%x\n", read_ioctl_data);

  read_ioctl_data = FIFO_LEVEL_REG_ADDR;
  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
  printf("FIFOLevelReg Value(inicial): %u\n", read_ioctl_data);

  write_dto.addr = FIFO_DATA_REG_ADDR;
  write_dto.data = 11;
  ioctl(dev, IOCTL_RC522_WRITE_REGISTER, &write_dto);

```

```
45  read_ioctl_data = FIFO_LEVEL_REG_ADDR;
46  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
47  printf("FIFOLevelReg Value(apos inserir um byte): %u\n",
     read_ioctl_data);
48
49  write_multiple_dto.addr = FIFO_DATA_REG_ADDR;
50  write_multiple_dto.data[0] = 12;
51  write_multiple_dto.data[1] = 22;
52  write_multiple_dto.data[2] = 32;
53  write_multiple_dto.data[3] = 42;
54  write_multiple_dto.data_size = 4;
55  ioctl(dev, IOCTL_RC522_WRITE_REGISTER_MULTIPLE, &write_multiple_dto);
56
57  read_ioctl_data = FIFO_LEVEL_REG_ADDR;
58  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
59  printf("FIFOLevelReg Value(apos mais 4 bytes): %u\n", read_ioctl_data)
     ;
60
61  read_ioctl_data = FIFO_DATA_REG_ADDR;
62  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
63  printf("FIFO_DATA_REG_ADDR(primeiro valor inserido) Value: %u\n",
     read_ioctl_data);
64
65  read_ioctl_data = FIFO_DATA_REG_ADDR;
66  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
67  printf("FIFO_DATA_REG_ADDR(segundo valor inserido) Value: %u\n",
     read_ioctl_data);
68
69  read_ioctl_data = FIFO_DATA_REG_ADDR;
70  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
71  printf("FIFO_DATA_REG_ADDR(terceiro valor inserido) Value: %u\n",
     read_ioctl_data);
72
73  read_ioctl_data = FIFO_DATA_REG_ADDR;
74  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
75  printf("FIFO_DATA_REG_ADDR(quarto valor inserido) Value: %u\n",
     read_ioctl_data);
76
77  read_ioctl_data = FIFO_DATA_REG_ADDR;
78  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
79  printf("FIFO_DATA_REG_ADDR(quinto valor inserido) Value: %u\n",
     read_ioctl_data);
80
81  read_ioctl_data = FIFO_LEVEL_REG_ADDR;
82  ioctl(dev, IOCTL_RC522_READ_REGISTER, &read_ioctl_data);
83  printf("FIFOLevelReg Value(apos recuperar todos os bytes que haviam
     sido inseridos): %u\n", read_ioctl_data);
```

```
84
85    close(dev);
86    return 0;
87 }
```

C.9 Arquivo de código para teste de comunicação com tags do *driver* de dispositivo para o dispositivo RFID-RC522

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <sys/ioctl.h>
7  #include <string.h>
8  #include <signal.h>
9
10 #include "rfid_rc522_drive_ioctl.h"
11
12 int dev;
13
14 void intHandler(int dummy) {
15   close(dev);
16   exit(0);
17 }
18
19 rc522_status request_picc(int file_id) {
20   struct rfid_rc522_req_a_picc_dto dto;
21
22   ioctl(file_id, IOCTL_RC522_REQ_A_PICC, &dto);
23   printf("req_a_picc status %u\n", dto.status);
24   if(dto.status != 0) return dto.status;
25   for (size_t i = 0; i < dto.res_size; i++)
26   {
27     printf("req_a_picc[%d] = %x\n", i, dto.res[i]);
28   }
29   return dto.status;
30 }
31
32 rc522_status anticollision(int file_id, uint8_t uid[5]) {
33   struct rfid_rc522_anticollision_dto dto_anticollision;
34
35   ioctl(file_id, IOCTL_RC522_ANTICOLLISION, &dto_anticollision);
36   printf("anticollision status %u\n", dto_anticollision.status);
37   if(dto_anticollision.status != 0) return dto_anticollision.status;
38     printf("uid bytes: ");
```

```
39   for (size_t i = 0; i < dto_anticollision.res_size; i++)
40   {
41     printf("%x ", dto_anticollision.uid[i]);
42   }
43   printf("\n");
44   memcpy(uid, dto_anticollision.uid, 5);
45   return dto_anticollision.status;
46 }
47
48 rc522_status select_tag(int file_id, uint8_t uid[5]) {
49   struct rfid_rc522_select_tag_dto dto_select_tag;
50
51   memcpy(dto_select_tag.uid, uid, 5);
52   ioctl(file_id, IOCTL_RC522_SELECT_TAG, &dto_select_tag);
53   printf("dto_select_tag status %u\n", dto_select_tag.status);
54   if(dto_select_tag.status != 0) return dto_select_tag.status;
55   return dto_select_tag.status;
56 }
57
58 rc522_status auth_in_sector(int file_id, uint8_t sector, uint8_t uid[5],
       uint8_t sector_key[6]) {
59   struct rfid_rc522_authenticate_dto dto_auth;
60
61   memcpy(dto_auth.uid, uid, 5);
62   memcpy(dto_auth.sector_key, sector_key, 6);
63   dto_auth.block_address = sector*4 + 3;
64   ioctl(file_id, IOCTL_RC522_AUTHENTICATE, &dto_auth);
65   return dto_auth.status;
66 }
67
68 rc522_status read_block(int file_id, uint8_t block_address, int silent)
     {
69   struct rfid_rc522_read_picc_block_dto dto_read;
70
71   dto_read.block_address = block_address;
72   ioctl(file_id, IOCTL_RC522_READ_PICC_BLOCK, &dto_read);
73
74   if(dto_read.status != 0) return dto_read.status;
75   if(silent) return dto_read.status;
76   printf("bloco %u: |", block_address);
77   for (size_t i = 0; i < dto_read.res_size; i++)
78   {
79     if(block_address != 0) {
80       printf("%c", dto_read.res[i]);
81       continue;
82     }
83     printf("%u", dto_read.res[i]);
```

```
84    }
85    printf("|\n");
86    return dto_read.status;
87  }
88
89  rc522_status write_block(int file_id, uint8_t block_address, char *data,
        int silent) {
90    struct rfid_rc522_write_picc_block_dto dto_write;
91
92    memcpy(dto_write.input, data, 16);
93    dto_write.block_address = block_address;
94    ioctl(file_id, IOCTL_RC522_WRITE_PICC_BLOCK, &dto_write);
95    if(!silent) printf("write_block status %u\n", dto_write.status);
96    return dto_write.status;
97  }
98
99  void write_all_blocks(int file_id, uint8_t *uid, uint8_t *sector_key) {
100   rc522_status status;
101   for (size_t sector = 1; sector < 16; sector++)
102   {
103     uint16_t cont_auth = 0;
104     do
105     {
106       status = auth_in_sector(file_id, sector, uid, sector_key);
107       cont_auth++;
108     } while (status != RC522_OK && cont_auth < 256);
109     if(cont_auth >= 256) {
110       printf("Não foi possível autenticar em setor %u, rc522_status: %u\
      n", sector, status);
111       continue;
112     }
113     for (size_t block_in_sector = 0; block_in_sector < 3;
      block_in_sector++)
114     {
115       size_t current_block = sector*4 + block_in_sector;
116       uint16_t cont_write = 0;
117       do
118       {
119         char data[30];
120         sprintf(data, "bloco: %u          ", current_block);
121         status = write_block(file_id, current_block, data, 1);
122         cont_write++;
123         usleep(10000);
124       } while (status != RC522_OK && cont_write < 256);
125       if(cont_write >= 256) {
126         printf("Não foi possível ler o bloco %u, rc522_status: %u\n",
      current_block, status);
```

```
127       }
128     }
129   }
130 }
131
132 void read_all_blocks(int file_id, uint8_t *uid, uint8_t *sector_key) {
133   rc522_status status;
134   for (size_t sector = 0; sector < 16; sector++)
135   {
136     uint16_t cont_auth = 0;
137     do
138     {
139       status = auth_in_sector(file_id, sector, uid, sector_key);
140       cont_auth++;
141     } while (status != RC522_OK && cont_auth < 256);
142     if(cont_auth >= 256) {
143       printf("Não foi possível autenticar em setor %u, rc522_status: %u\
    n", sector, status);
144       continue;
145       }
146     for (size_t block_in_sector = 0; block_in_sector < 3;
    block_in_sector++)
147     {
148       size_t current_block = sector*4 + block_in_sector;
149       uint16_t cont_read = 0;
150       do
151       {
152         status = read_block(file_id, current_block, 0);
153         cont_read++;
154       } while (status != RC522_OK && cont_read < 256);
155       if(cont_read >= 256) {
156         printf("Não foi possível ler o bloco %u, rc522_status: %u\n",
    current_block, status);
157       }
158     }
159   }
160 }
161
162 int main() {
163   rc522_status status;
164   uint8_t key[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
165   uint8_t uid[5];
166   uint16_t cont_read = 0;
167   uint16_t cont_auth = 0;
168
169   dev = open("/dev/rfid_rc522_driver", O_NONBLOCK);
170   if(dev == -1) {
```

```
171     printf("Não foi possível abrir arquivo\n");
172     return -1;
173   }
174   signal(SIGINT, intHandler);
175
176   ioctl(dev, IOCTL_RC522_STOP_AUTH);
177
178   status = request_picc(dev);
179   if(status != RC522_OK) {
180     printf("Não foi possível requisitar uma tag, rc522_status: %u\n",
      status);
181     return -1;
182   }
183
184   status = anticollision(dev, uid);
185   if(status != RC522_OK) {
186     printf("Não foi possível realizar protocolo anticolisão,
     rc522_status: %u\n", status);
187     return -1;
188   }
189
190   status = select_tag(dev, uid);
191   if(status != RC522_OK) {
192     printf("Não foi possível realizar selecionar tag, rc522_status: %u\n
     ", status);
193     return -1;
194   }
195
196   read_all_blocks(dev, uid, key);
197   printf("read all done\n");
198   sleep(1);
199
200   write_all_blocks(dev, uid, key);
201   printf("write all done\n");
202
203   sleep(1);
204   read_all_blocks(dev, uid, key);
205   printf("read all done\n");
206
207   status = auth_in_sector(dev, 2, uid, key);
208   printf("auth again %u\n", status);
209   write_block(dev, 8, "Ola mundo, 8        ", 1);
210   printf("write done\n");
211
212   status = auth_in_sector(dev, 4, uid, key);
213   printf("auth again %u\n", status);
214   write_block(dev, 16, "Ola mundo, 16        ", 1);
```

```
215   printf("write done\n");
216
217   status = auth_in_sector(dev, 8, uid, key);
218   printf("auth again %u\n", status);
219   write_block(dev, 32, "Ola mundo, 32        ", 1);
220   printf("write done\n");
221
222   read_all_blocks(dev, uid, key);
223   printf("read all done\n");
224   write_all_blocks(dev, uid, key);
225
226   ioctl(dev, IOCTL_RC522_STOP_AUTH);
227
228   close(dev);
229   return 0;
230 }
```