UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

TIAGO LAZAROTTO

DESENVOLVIMENTO DE DRIVER E INTERFACE EM SISTEMA OPERACIONAL LINUX PARA CONTROLE DE SENSORES E ATUADORES EM AUTOMÓVEIS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2018

TIAGO LAZAROTTO

DESENVOLVIMENTO DE DRIVER E INTERFACE EM SISTEMA OPERACIONAL LINUX PARA CONTROLE DE SENSORES E ATUADORES EM AUTOMÓVEIS

Trabalho de Conclusão de Curso de graduação, do Bacharelado em Engenharia de Computação do Departamento Acadêmico de Informática – DAINF – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. Gustavo Weber Denardin

PATO BRANCO

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 9 horas e 30 minutos do dia 13 de dezembro de 2018, na sala V105, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Gustavo Weber Denardin (orientador), Diogo Ribeiro Vargas e Marco Antonio de Castro Barbosa para avaliar o trabalho de conclusão de curso com o título **Desenvolvimento de driver e interface em sistema operacional Linux para controle de sensores e atuadores em automóveis**, do aluno **Tiago Lazarotto** matrícula 00833975, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguida pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Prof. Gustavo Weber Denardin
Orientador (UTFPR)

_____
Prof. Diogo Ribeiro Vargas
(UTFPR)

_____
Prof. Fábio Favarim
(UTFPR)

_____
Profa. Beatriz Terezinha Borsoi
Coordenador de TCC

_____
Prof. Pablo Gauterio Cavalcanti
Coordenador do Curso de
Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

LAZAROTTO, Tiago. Desenvolvimento de Driver e Interface em Sistema Operacional Linux para Controle de Sensores e Atuadores em Automóveis. 2018. 57 f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2018.

A mais nova geração de veículos automotivos, incluindo os veículos autônomos, está cada vez mais conectada aos seus componentes e até mesmo com outros veículos por meio da Internet das Coisas, o que vai muito além dos aplicativos de entretenimento oferecidos pelas empresas do mercado hoje. Tendo isso em vista, este trabalho contempla o desenvolvimento de um *driver* de dispositivo e uma API de comunicação na distribuição do sistema operacional Linux chamada Debian, *driver* o qual será desenvolvido em linguagem C por meio de um depurador remoto que se utiliza das bibliotecas desse sistema operacional, o qual é adotado pela plataforma BeagleBone Black, que será utilizada como o dispositivo central da rede. O *driver* tem por objetivo controlar uma rede sem fio de sensores e atuadores veiculares de forma modular, obedecendo os identificadores desses dispositivos, utilizados na rede CAN de automóveis sendo que, para a comunicação entre os dispositivos, será utilizado o transceptor MRF24J40MA que segue o padrão IEEE 802.15.4. Para que possa haver uma comunicação por parte de programas de alto nível com o *driver*, foi criada uma API que é invocada por chamada de sistema, de forma que a mesma vai se comunicar com o *driver* principal por meio de *sockets* locais e *threads* para que se tenha um canal multicliente para atender os chamados da API, nos quais poderão ser solicitadas informações de sensores específicos ou acionamento de determinados atuadores. Para o desenvolvimento do *driver* e da API será configurado um depurador remoto entre um computador de propósito geral (PC) e a plataforma do Debian, de forma a utilizar as bibliotecas nela contidas e facilitar o desenvolvimento em uma interface gráfica, nesse caso o NetBeans. O *Driver* Principal será formado dos *drivers* de SPI e Transceptor, juntamente com o Controlador de Módulos e dos *sub-drivers* dos sensores e atuadores. Com o dispositivo central pronto, se faz necessário utilizar outro microcontrolador para efetuar a comunicação sem fio e emular um sensor ou atuador da rede, e para isso foi escolhida a plataforma FRDM KL25Z da NXP. Os resultados obtidos foram a depuração remota realizada com sucesso, dessa forma possibilitando também a demonstração da importação bem-sucedida dos módulos e da configuração dos *drivers* da SPI e do Transceptor. O resultado da chamada de sistema para a API pode também ser visualizado e comprovado, e a partir disso pode-se concluir que o trabalho traz uma contribuição para as plataformas automobilísticas que optarem por utilizar sensores e atuadores sem fio, de forma a baratear a fabricação desses veículos. O trabalho também traz contribuições para depuração remota e modularização de bibliotecas na linguagem C. Como contribuição a toda comunidade de desenvolvimento, o código resultante está disponibilizado como código aberto no GitHub, conforme descrito na conclusão do trabalho.

Palavras-chave: Vehicle. Driver. Wireless. Sensors. Linux.

ABSTRACT

LAZAROTTO, Tiago. Desenvolvimento de Driver e Interface em Sistema Operacional Linux para Controle de Sensores e Atuadores em Automóveis. 2018. 57 f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2018.

The newest generation of automotive vehicles, including autonomous vehicles, is evermore connected to its components and even to other vehicles as well through the Internet of Things, which expands well beyond the current scenario of entertainment applications offered by companies in the market today. Having that in mind, this paper contemplates the development of a device driver and a communication API in the Debian distribution of the Linux operational system, driver which will be developed using C language through a remote debugger that uses the libraries already present in that operational system, which is adopted by the BeagleBone Black platform, used as the main device of the network. The driver's objective is to control a wireless network of vehicular sensors and actuators in a modular way, following the CAN network device identifiers used in automobiles, using the transceiver MRF24J40MA to communicate between these devices, as this transceiver adopts the IEEE 802.15.4 standard. In order to allow communication between high level programs and the driver, an API was created to be invoked by system call in a way that it will communicate with the main driver through local sockets and threads, allowing it to have a multiclient channel to attend multiple system calls, through which information can be solicitated regarding specific sensors or action commands can be delivered to actuators. For the development of both the driver and API, a remote debugger will be configured between a general-purpose computer (PC) and the Debian platform in order to utilize the libraries contained in it and facilitate the development when using a graphical interface, in this case, NetBeans. The main driver will contain both SPI and Transceiver drivers, as well as the Module Controller together with the sensors and actuators sub-drivers. With the main device completed, it a necessity to have another microcontroller to establish the wireless communication and emulate a sensor or actuator in the network, and for that purpose the FRDM KL25Z platform from NXP was chosen. One of the acquired results was the successful remote debugging of the driver, which allowed for a successful demonstration of the module imports and the SPI and Transceiver drivers configurations. The result of the API system call can also be visualized and confirmed, and through these facts it can be concluded that this paper brings a contribution to the automotive platforms that opt for using wireless sensors and actuators, with the idea of making these vehicles more affordable. This paper also brings contribution to remote debugging and modularization of C language libraries. As a contribution to all the development community, the resulting code of this paper is available as an open source project in GitHub, as described in the conclusion of the paper.

Keywords: Vehicle. Driver. Wireless. Sensors. Linux.

LISTA DE FIGURAS

Figura 1 – Arquitetura do Processador ARM 17
Figura 2 – *Automotive Grade Linux* 19
Figura 3 – Topologias em estrela, árvore e malha. 22
Figura 4 – Diagrama do Sistema. 26
Figura 5 – Conexão bem-sucedida do depurador remoto. 50
Figura 6 – *Breakpoint* de inicialização do transceptor. 50
Figura 7 – Transceptor inicializado com sucesso. 51
Figura 8 – Demonstração de que a API está pronta e esperando chamadas. ... 52
Figura 9 – Controlador de módulos carregando um *sub-driver* com sucesso. . 53

LISTAGENS DE CÓDIGO

Listagem 1 – IP estático em Linux. 28
Listagem 2 – Privilégios e compartilhamento utilizando o Samba *Share*. 29
Listagem 3 – Configuração do *driver* PinCtrl para SPI. 31
Listagem 4 – Compilação e criação do arquivo do dispositivo físico. 32
Listagem 5 – Configuração do *driver* PinCtrl para evento de interrupção. 33
Listagem 6 – Configuração do *driver* PinCtrl para evento de interrupção. 33
Listagem 7 – Aquisição do descritor do dispositivo SPI. 34
Listagem 8 – Método para escrita na SPI. 35
Listagem 9 – Métodos para processamento de interrupção. 36
Listagem 10 – Métodos utilizáveis do *driver* SPI. 37
Listagem 11 – Objetos da biblioteca do transceptor. 38
Listagem 12 – Estruturas do controlador de módulos. 39
Listagem 13 – Estruturas do controlador de módulos. 41
Listagem 14 – Métodos do controlador de módulos. 41
Listagem 15 – Biblioteca do comunicador (communicator.h). 43
Listagem 16 – Exemplo de biblioteca de um *sub-driver* denominada "autoDemo" (autoDemo.h). 43
Listagem 17 – Cabeçalho do *Driver* Principal. 44
Listagem 18 – Estrutura do *Driver* Principal (awd.c). 48
Listagem 19 - Métodos de comunicação com o transceptor na FRDM KL25Z. 49

LISTA DE SIGLAS

ABS	*Anti-lock Breaking System*
ADC	*Analogic to Digital Conversion*
AES	*Advanced Encryption Standard*
AGL	*Automotive Grade Linux*
APB	*Advanced Peripheral Bus*
API	*Application Program Interface*
CAN	*Controller Area Network*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DSSS	*Direct Sequence Spread Spectrum*
DTB	*Device Tree Blob*
DTBO	*Device Tree Blob Object*
DTO	*Device Tree Overlay*
DTS	*Device Tree Source*
eMMC	*Embedded Multimedia Card*
FIFO	*First In First Out*
GCC	GNU *Compiler Collection*
I2C	*Inter-Integrated Circuit*
ID	*Identification*
IEEE	*Institute of Electrical and Electronics Engineers*
IOCTL	*Input/Output Control*
IP	*Internet Protocol*
Kbps	*Kilobits per Second*
MAN	*Metropolitan Area Network*
MUX	*Multiplexer*
OMAP	*Open Multimedia Applications Platform*
OSI	*Open Systems Interconnection*
PAN	*Personal Area Network*
PCB	*Printed Circuit Board*
PID	*Process Identification*
PinCtrl	*Pin Controller*

PWM	*Pulse Width Modulation*
RSSF	Redes de Sensores Sem Fio
RTOS	*Real-Time Operational System*
SD	*Secure Digital Memory*
SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
TCP	*Transmission Control Protocol*
UART	*Universal Asynchronous Receiver-Transmitter*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*

SUMÁRIO

1. INTRODUÇÃO 11
1.1 CONSIDERAÇÕES INICIAIS 11
1.2 OBJETIVOS 12
1.2.1 Objetivo Geral 12
1.2.2 Objetivos Específicos 12
1.3 JUSTIFICATIVA 13
2. REFERENCIAL TEÓRICO 14
2.1 SISTEMAS OPERACIONAIS 14
2.1.1 Linux Embarcado 15
2.1.2 *Driver* 15
2.1.3 *Drivers* de Plataforma em Linux 16
2.1.4 *Driver* de Dispositivo 18
2.1.5 *Automotive Grade Linux* 18
2.2 SENSORES E ATUADORES AUTOMOTIVOS 19
2.3 ATUADORES 20
2.4 REDE CAN 20
2.5 TOPOLOGIA DE COMUNICAÇÃO PARA DISPOSITIVOS SEM FIO 21
2.6 *SOCKET* 23
2.7 *THREAD* 24
3. MATERIAIS E MÉTODOS 25
4. DESENVOLVIMENTO 27
4.1 Depurador Remoto 27
4.2 Comunicação SPI e Interrupções em Linux 30
4.3 *Driver* SPI 33
4.4 *Driver* do Transceptor 37
4.5 Controlador de Módulos 38
4.6 *Driver* Principal 41
4.7 Sensor Emulado na FRDM KL25Z 48
4.8 Resultados Obtidos 49
5. CONCLUSÃO 54
6. REFERÊNCIAS 55

1. INTRODUÇÃO

Este capítulo apresenta as considerações iniciais, os objetivos e a justificativa do trabalho.

1.1 CONSIDERAÇÕES INICIAIS

A mais nova geração de veículos automotivos, incluindo os veículos autônomos, está cada vez mais conectada aos seus componentes e até mesmo com outros veículos por meio da Internet das Coisas, o que vai muito além dos aplicativos de entretenimento oferecidos pelas empresas do mercado, como Microsoft, Google e Apple. Essa nova geração, também chamada de “Era dos Automóveis Conectados”, está se utilizando de serviços em nuvem que aumentam a segurança e a interatividade com o condutor, com uma vasta área de serviços adicionais (JAYARAMAN, 2014).

Hoje existem sistemas operacionais automotivos como o Ford Sync, desenvolvido pela Microsoft, o Android Auto, desenvolvido pela Google, e o Apple CarPlay, todos sistemas proprietários. Porém, também existem soluções em código aberto para realizar essa mesma função, e podem ser utilizados e adaptados por qualquer montadora de veículos, se tornando uma plataforma automobilística mais barata e podendo se usufruir de contribuições externas (PETRUCELLI, 2018).

Muitos dos veículos de hoje possuem de 60 a 100 sensores que controlam desde o clima interno até os *AirBags*. Os fabricantes da área estimam que esse número dobre conforme os automóveis vão ficando mais inteligentes (GRANDE, 2018). Esses automóveis conectados já estão em produção, e dentre algumas fabricantes a Fundação Linux está presente com o *Automotive Grade Linux*, ou, em português, Linux Automotivo, que foi desenvolvido especialmente para aplicações embarcadas e de tempo real, além de possuir sua base em código aberto.

Para se obter essa quantidade de sensores na geração atual de automóveis, as montadoras utilizam um sistema de cabeamento de custo e peso elevados que interliga todos os sensores com o computador de bordo. A nova geração de veículos pretende utilizar Redes de Sensores Sem Fio (RSSF), baseadas no padrão IEEE 802.15.4 (GASCÓN, 2008), que dá ênfase na baixa potência de operação, na baixa taxa de transmissão de dados e no baixo custo de implantação.

A partir dessas premissas torna-se necessário o desenvolvimento de um

software que faça a integração entre os sensores e atuadores com um sistema operacional, em que um sistema de código aberto Linux tem se destacado. Esse tipo de *software* de baixo nível é conhecido como *driver* e é utilizado no AGL para possibilitar que um aplicativo de interface gráfica de alto nível possa utilizar os dados desses sensores para mostrar ao usuário o que está ocorrendo no veículo, bem como para realizar ações em atuadores dentro do veículo.

Para a comunicação são utilizados transceptores (dispositivos que enviam e recebem mensagens em forma de *bytes* por meio de radiofrequência), que permitem a troca de mensagens com o computador de bordo. Este, a partir de um *driver,* pode ter acesso às leituras dos sensores e realizar ações em atuadores.

O resultado esperado do trabalho é gerar um meio confiável de comunicação entre o computador de bordo de automóveis e os dispositivos de sensoriamento e atuação de um veículo. Esse meio pode vir a padronizar a topologia de troca de informação entre esses componentes, evitando assim que empresas venham a criar meios proprietários que dificultam a criação de novas formas de interagir com o veículo.

1.2 OBJETIVOS

A seguir estão o objetivo geral e os objetivos específicos.

1.2.1 Objetivo Geral

Desenvolver um *driver* e uma API de acesso para o sistema operacional Linux, especificamente para a distribuição Debian, visando efetuar o monitoramento e o controle de sensores e atuadores como módulos do *driver*, sendo que estes dispositivos estarão dispostos em uma rede compatível com o padrão IEEE 802.15.4.

1.2.2 Objetivos Específicos

- Estabelecer um depurador remoto que consiga se utilizar das bibliotecas do Debian contido na plataforma BeagleBone Black;
- Desenvolver um *driver* de baixo nível para sistemas operacionais Linux que permita a comunicação com um transceptor MRF24J40MA;

- Desenvolver uma API para o acesso às mensagens trocadas pela rede automotiva de sensores e atuadores;
- Desenvolver um controlador de *sub-drivers*, o qual vai importar as bibliotecas desses *sub-drivers*, permitindo assim que o *driver* possa ser modular.

1.3 JUSTIFICATIVA

O uso de redes de sensores sem fio baseadas no padrão IEEE 802.15.4 foi escolhida para o desenvolvimento desse trabalho devido a sua grande utilização em diversos setores que utilizam sensoriamento (industrial, médico, militar, entre outros), especialmente o setor automotivo, no qual a mesma vem sendo utilizada recentemente com a crescente necessidade de se reduzir o cabeamento em veículos por meio de redes *wireless*.

A distribuição Debian foi escolhida para o sistema operacional em que esse trabalho será desenvolvido por ser a distribuição padrão da plataforma BeableBone Black da Texas Instruments, além de ser uma distribuição estável do Linux.

Os *sub-drivers* serão desenvolvidos como módulos para facilitar a criação de produtos de código aberto como um incentivo para que cada vez menos *drivers* proprietários sejam criados, reduzindo assim o custo para o cliente final se essa plataforma for utilizada em veículos.

A configuração e utilização de um depurador remoto se faz essencial para facilitar o desenvolvimento do *driver* e de seus *sub-drivers* como um todo, pois permite que a programação seja realizada em um equipamento com desempenho muito superior ao da plataforma BeagleBone Black.

2. REFERENCIAL TEÓRICO

Nesta seção são apresentados os conceitos utilizados para realizar os objetivos propostos do trabalho.

2.1 SISTEMAS OPERACIONAIS

Um sistema operacional é o *software* mais importante que executa em um computador. Ele gerencia a memória, os processos, e todos os outros *softwares* e, também, *hardwares* conectados à máquina (GOODWILL..., 2014).

Esses sistemas são utilizados tanto em computadores de propósito geral (ex.: computador pessoal) quanto em dispositivos embarcados (ex.: computador de bordo). O projeto de um dispositivo embarcado envolve *software* e *hardware*, em que se concebe um computador de propósito específico, com funções e comportamentos bem determinados para uma determinada aplicação. Tal abordagem provê desempenho e tempo de resposta mais adequado aos objetivos da aplicação, bem como permite um menor encapsulamento da estrutura física do mesmo. Em veículos automotivos, sistemas embarcados são utilizados para controlar componentes como *Anti-lock Breaking System* (ABS), direção elétrica, acelerador eletrônico, controle de emissão de gases, informações mostradas no painel, entre outros. Para tanto, requerem um sistema operacional mais leve de forma a não utilizar um *hardware* de alta capacidade de processamento e, consequentemente, de alto custo e consumo energético (AGARWAL, 2016).

Entre os vários sistemas operacionais embarcados existentes, o Linux se destaca por ser um sistema operacional de código aberto que possui muitas bibliotecas e sistemas de vários desenvolvedores em todo mundo. Esses *softwares*, quando juntos, acabam por formar um sistema com operações complexas de alto nível e muitas vezes de fácil utilização, úteis para usuários leigos (COMPUTER..., 2014). O Linux, porém, é apenas o núcleo (*kernel*), existindo muitas distribuições diferentes do mesmo, sendo que cada distribuição é formada por um conjunto de aplicações que proporcionam várias ferramentas e interfaces gráficas variadas (LINUX T..., 2018).

Uma dessas distribuições é o *Automotive Grade Linux* (Linux de Qualidade Automotiva), ou AGL, que tem por objetivo disponibilizar uma plataforma aberta de desenvolvimento para que qualquer usuário possa construir suas próprias aplicações

automotivas (LINUX T..., 2018).

2.1.1 Linux Embarcado

Um sistema embarcado é direcionado para aplicações que necessitam de um baixo consumo de energia e espaço físico, e essa demanda é contemplada por um *hardware* mais compacto e leve. Dessa forma, as plataformas que se utilizam de sistemas embarcados possuem um *hardware* mais limitado em termos de poder de processamento e variedade de periféricos, mas mesmo assim conseguem realizar computações complexas que necessitem de um poder de processamento em torno de 1 GHz (EMBEDDED ..., 2018).

O Linux embarcado é construído de forma que apenas os componentes essenciais do *Kernel* e as bibliotecas necessárias para o seu funcionamento estejam presentes, geralmente não contendo uma interface gráfica de usuário. Por meio desse pacote compacto, o sistema operacional não consome muitos recursos do *hardware*, possibilitando que outros programas possam usufruir desses recursos de forma mais eficiente (EMBEDDED ..., 2018).

Para a construção e compilação de programas em sistemas Linux, tanto embarcado quanto o normal, é necessário que se utilize a *Tool Chain* (ou Cadeia de Ferramentas) específica do processador e sistema operacional utilizados. Essa *Tool Chain* possui as ferramentas necessárias para compilar, depurar e executar os programas de toda uma plataforma, e pode ser utilizada remotamente, o que é chamado de *Cross - Tool Chain*. Em sistemas embarcados geralmente são utilizadas *Cross – Tool Chains* pelo fato de que é mais cômodo aos programadores utilizar uma interface gráfica para a programação e depuração do programa designado a funcionar dentro do sistema embarcado, realizando as operações de depuração e compilação de forma remota por meio de, por exemplo, uma rede cabeada (EMBEDDED ..., 2018).

2.1.2 *Driver*

O sistema operacional gerencia todos os *tipos de hardware* do sistema utilizando uma parte de seu *software* denominada *kernel*, o qual por sua vez controla todas as interações entre os processos e os periféricos, gerenciando assim qual processo será executado (pela troca de contexto por prioridade, por exemplo), quanto

tempo permanecerá executando (preempção), e quais recursos o mesmo tem permissão de acessar. O *kernel* realiza todas essas funções por meio do acesso direto ao processador e a memória, a partir dos quais adquiri acesso a todos os outros periféricos de *hardware* da máquina, e faz isso com ajuda de interrupções geradas pela interação desses dispositivos com o processador, o qual sempre recorre ao *kernel* para gerenciá-las (LINUX I..., 2005).

Quando um dispositivo deseja se comunicar com o sistema, o *kernel* procura em uma lista por um *software* que seja capaz de interpretar as informações vindas daquele tipo de dispositivo, esse *software* é chamado de *driver*. O nome em inglês *driver* significa motorista, isso porque esse *software* é responsável por dirigir a execução das instruções necessárias para que processos possam se comunicar com um periférico e vice-versa. Em geral, esse programa é feito de tal modo que ponteiros de funções são atribuídos a lista de *drivers* do *kernel*, em que essas funções devem fazer todas as operações necessárias para inicializar o *driver* e o dispositivo, ler e escrever dados, e por final realizar a remoção do dispositivo, sendo que no sistema Linux a interface com um dispositivo é feita utilizando um arquivo (CALBET, 2006).

Um conceito muito importante quando se inicia na programação de *drivers* para Linux é saber a diferença entre *drivers* de plataforma e *drivers* de dispositivo (CALBET, 2006).

2.1.3 *Drivers* de Plataforma em Linux

Estes *drivers* são responsáveis por controlar dispositivos que aparecem como entidades autônomas no sistema e contêm informações de *hardware*, estando associados a controladores de barramento como SPI (*Serial Peripheral Interface*, ou Interface Serial de Periféricos), I2C (*Inter-Integrated Circuit*, ou Circuito Inter Integrado) e USB (*Universal Serial Bus*, ou Porta Serial Universal) (EMBARCADOS, 2014).

Então, em termos práticos, *driver* de plataforma é uma abstração para representar um controlador ou adaptador interno de um SoC (*System on Chip*, ou Sistema no Chip), e também outros periféricos mais complexos como controlador de Interrupções e controlador de DMA (*Direct Memory Access*, ou Acesso Direto à Memória). Ele também pode ser visto como um pseudo-barramento que conecta a CPU (*Central Processing Unit*, ou Unidade de Processamento Central) a um

controlador interno a ele, isto é, no mesmo chip. Na Figura 1 são apresentados vários controladores conectados a uma CPU ARM por meio do barramento APB (*Advanced Peripheral Bus*, ou Barramento Avançado de Periféricos) (EMBARCADOS, 2014).

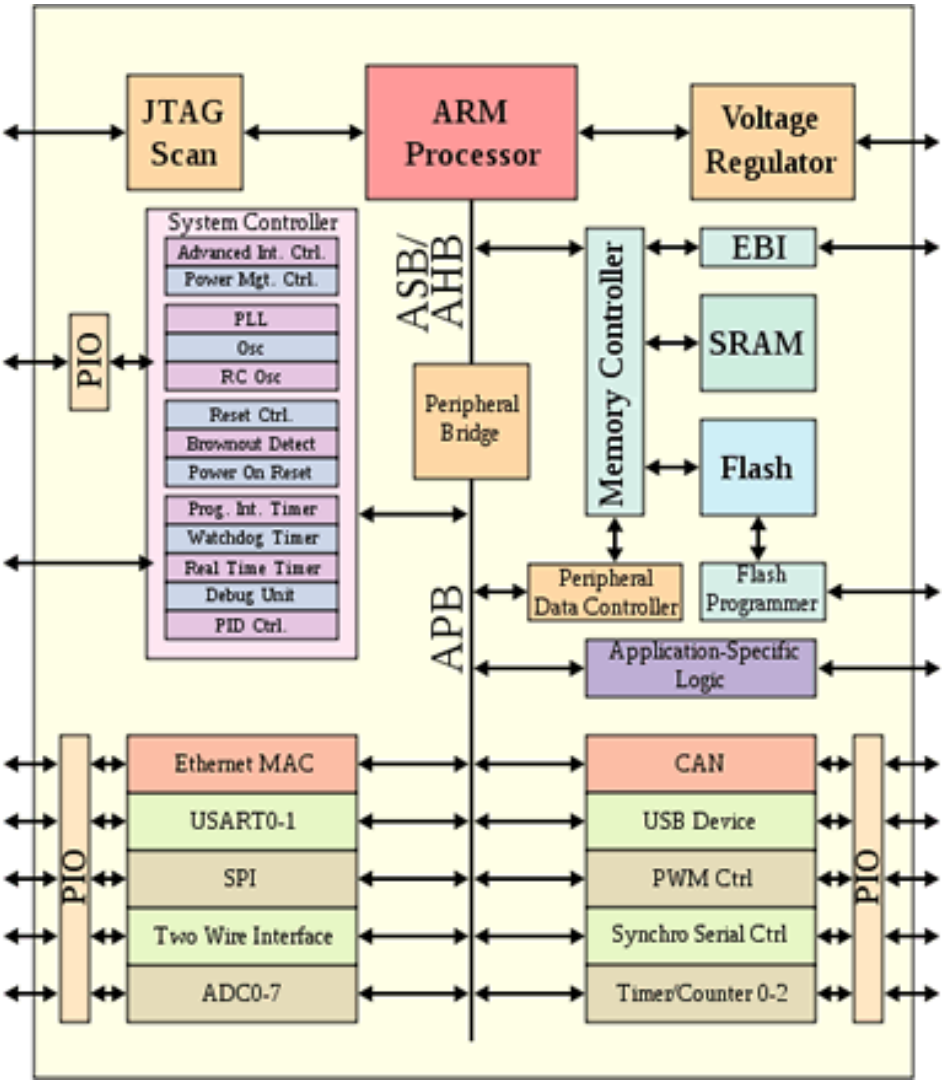


Figura 1 – Arquitetura do Processador ARM
Fonte: Embarcados (EMBARCADOS, 2014).

Para as placas embarcadas existe um *driver* de plataforma que acompanha a distribuição padrão do Linux, chamado PinCtrl (*Pin Controller*, ou Controlador de Pinos), esse *driver* utiliza o mapeamento dos pinos do microcontrolador e das funções de cada um para realizar as operações inscritas em arquivos de configuração, de forma que as operações de escrita e leitura em pinos se dão por meio de *Pipes* (Canos), que nada mais são do que arquivos que armazenam as informações a serem transmitidas e recebidas. Esse mapeamento de pinos é feito utilizando o MUX (*Multiplexer*, ou Multiplexador) do microcontrolador, onde cada porta encontrada no

esquemático do mesmo é mapeada em um barramento específico com um código específico (EMBARCADOS, 2014).

2.1.4 *Driver* de Dispositivo

Pode ser definido como uma estrutura estaticamente alocada, ou de forma mais simples, é a camada de *software* usada para comunicação com um dispositivo externo ao processador e que está associada a algum barramento como, por exemplo, barramento I2C, SPI ou USB. Esse tipo de *driver*, diferente do *driver* de plataforma, funciona em qualquer processador, mesmo que sejam de arquiteturas diferentes. Ou seja, seu código é independente de *hardware*, enquanto que o *driver* de plataforma depende do *hardware* (EMBARCADOS, 2014).

Para realizar as operações necessárias no *hardware*, o *driver* de dispositivo se utiliza dos *drivers* de plataforma por meio de *Pipes* com arquivos de escrita e leitura de instruções. Geralmente esse tipo de *driver* possui uma camada de *software* que gerencia os pacotes de informação enviados e recebidos, atendendo a chamadas de *softwares* externos por meio de chamadas de sistema (EMBARCADOS, 2014).

2.1.5 *Automotive Grade Linux*

O ramo da tecnologia automotiva está crescendo cada vez mais, com empresas como Google, Apple e Microsoft competindo por dominância sobre esse mercado. Com uma visão mais pública e de fácil desenvolvimento, empresas se uniram e criaram o projeto *Automotive Grade Linux* (MARTIN, 2014). A Figura 2 apresenta a interface gráfica desse sistema.

Figura 2 – *Automotive Grade Linux*
Fonte: FutureCar (WALZ, 2018).

Esse novo tipo de sistema operacional baseado em Linux foi lançado em uma conferência sobre Linux em Tóquio, realizada em maio de 2013. Sua idealização surgiu de uma colaboração entre 44 empresas, dentre elas a Linux Foundation, Jaguar Land Rover, Nissan, Toyota, Hyundai, Intel, Nvidia, Texas Instruments, Samsung e Panasonic, com o intuito de disponibilizar uma plataforma de código aberto em que qualquer desenvolvedor possa ter acesso ao código fonte do sistema, possibilitando que o mesmo crie seus próprios aplicativos. O projeto é disponibilizado via repositórios em Gerrit e *Git*, tendo o C como sua linguagem de baixo nível, e para o alto nível as linguagens HTML5 e JavaScript (THE LINUX..., 2018).

2.2 SENSORES E ATUADORES AUTOMOTIVOS

Sensores são dispositivos que detectam alguma grandeza física e convertem essa medida em outra grandeza (geralmente elétrica) seguindo um padrão de medida, o que permite que algum tipo de controlador ou monitor de eventos possa utilizar ou representar essas medidas (ROUSE, 2014).

Com esses sensores monitorando os estados dos componentes do veículo, há a necessidade de se controlar alguma variável física dos mesmos, e isso pode ser

realizado por meio de atuadores. Atuadores são dispositivos que realizam alguma ação sobre um determinado componente controlável do veículo, e podem ser ativados de forma elétrica ou mecânica. Para que possam atuar conforme as medidas dos sensores, a grande maioria desses atuadores precisam ser elétricos, como a injeção eletrônica, em que os sensores dos cilindros do motor monitoram a quantidade de massa dentro da câmara de combustão e, em conjunto com a medida de deslocamento do pedal do acelerador, obtém a informação necessária para que o computador de bordo possa calcular o tempo que os injetores (que são um tipo de atuador) devem ficar abertos para que mais combustível entre na câmara, acelerando, assim, o veículo (HUBPAGES, 2011).

Explorando os tipos de sensores existentes em um veículo moderno tem-se os sensores Hall, ultrassônico, indutivo, de temperatura, de pressão, de velocidade, de oxigênio, etc (HUBPAGES, 2011).

2.3 ATUADORES

Os sensores, como descrito na Seção 2.2, monitoraram constantemente os estados de certos componentes do veículo, dessa forma possibilitando que o computador de bordo possa tomar decisões utilizando as medidas recebidas, e então realizar ações. Para realizar essas ações o veículo possui dispositivos denominados atuadores, os quais consistem basicamente de motores e solenoides, que recebem sinais digitais ou analógicos e então atuam de forma elétrica, mecânica, hidráulica ou pneumática sobre um componente, controlando desde o fluxo de fluídos e gases, até sistemas de ABS e *AirBag* (THE GREEN BOOKS, 2013).

Os atuadores que possuem motores de passo são normalmente controlados por um sinal elétrico de largura proporcional ao quanto ele deve girar, como na abertura de uma válvula de passagem de ar, por exemplo. Já os solenoides podem ser controlados por um sinal modulado por largura de pulso chamado PWM (*Pulse Width Modulation*) que informa ao atuador o tempo de permanência nos estados ativo e inativo de seu mecanismo, e é utilizado, por exemplo, na abertura de uma válvula de injeção (THE GREEN BOOKS, 2013).

2.4 REDE CAN

É uma rede de comunicação entre as diversas unidades de controle dentro de

um veículo, as quais operam obedecendo um protocolo de comunicação serial bidirecional denominada CAN (*Controller Area Network*, ou Rede de Controle em Área).

O protocolo CAN utiliza uma topologia onde pode haver várias unidades de controle, que são responsáveis por iniciar uma comunicação, e vários escalonadores, que se comportam como roteadores de informação. Essa rede opera com todas as unidades de controle tendo o mesmo direito de acesso ao barramento. Esse protocolo tem como vantagem continuar funcionando caso uma das unidades apresente problemas.

Ao iniciar o processo de transmissão de informação, a unidade de controle envia um pacote de dados na rede. Cada pacote é composto por um código identificador e logo em seguida as informações.

O código identificador nomeia qual será o pacote a ser transmitido, sendo que em um mesmo veículo não existem pacotes de dados com o mesmo código identificador. Esse código também é usado para definir a prioridade entre os pacotes, de forma que o menor código identificador tem mais prioridade e o maior menos, ou seja, se dois módulos estiverem transmitindo informações ao mesmo tempo, a prioridade será dada ao número identificador menor, tendo em vista que duas informações não podem passar ao mesmo tempo pelo barramento.

2.5 TOPOLOGIA DE COMUNICAÇÃO PARA DISPOSITIVOS SEM FIO

Uma rede de sensores sem fio necessita de uma topologia e de um padrão de comunicação para ter uma utilização mais fácil e poder ser ampliada mais rapidamente. Pensando nisso, o Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE, ou *Institute of Electrical and Electronics Engineers*) reuniu um grupo de pesquisa, incorporado ao comitê de padronização de redes locais e metropolitanas, que por sua vez desenvolveu o padrão de comunicação 802.15 para redes PAN (*Personal Area Network*) e MAN (*Metropolitan Area Network*), dentro do qual existe a seção 802.15.4 que visa a implementação de redes sem fio de baixo custo e baixo consumo de energia (GASCÓN, 2008).

Para se obter essa economia de energia o padrão dita a utilização de um método de modulação conhecido como Espectro de Propagação em Sequência Direta (*Direct Sequence Spread Spectrum*, ou DSSS), que consiste em dividir a informação

a ser transmitida em quatro sinais diferentes. Isso faz com que cada sinal utilize uma densidade espectral menor, dessa forma reduzindo o consumo de energia mesmo que a largura de banda ocupada seja maior (GASCÓN, 2008).

Esse padrão sugere três topologias de rede que podem ser utilizadas conforme o tipo de aplicação e do objetivo que se deseja atingir, sendo estas as topologias em estrela, em árvore e em malha (*mesh*) (JENNIC, 2007). A Figura 3 demonstra essas topologias.

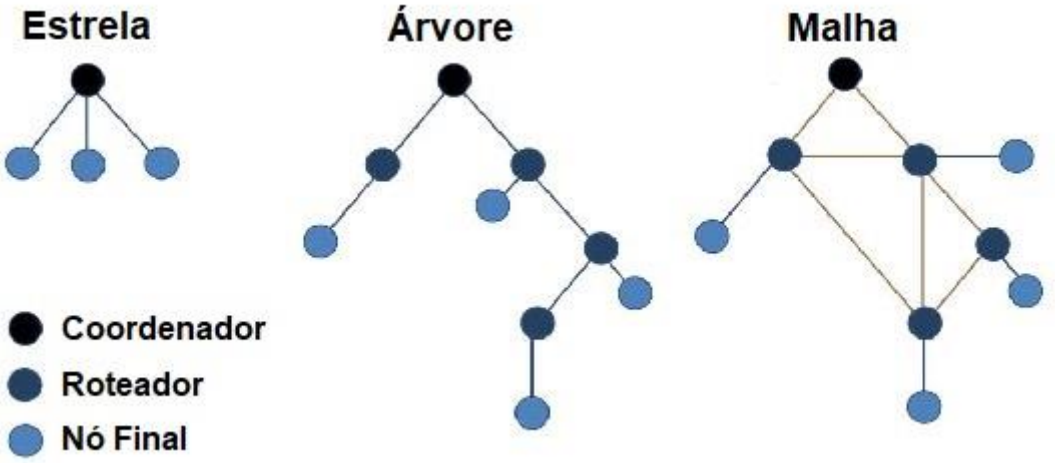


Figura 3 – Topologias em estrela, árvore e malha.
Fonte: Adaptado de National Instruments (NATIONAL..., 2009).

a) Topologia em estrela

Há um nó central (computador de bordo, por exemplo) que está conectado diretamente com todos os outros nós (sensores e atuadores), e todas as mensagens de comunicação tem de passar pelo nó central (JENNIC, 2007).

b) Topologia em árvore

Os nós estão conectados ao nó central direta ou indiretamente, assim um nó pode receber mensagens de outro nó diretamente sem a necessidade de intervenção do nó central. Porém nessa topologia os nós seguem uma hierarquia (árvore) que não pode apresentar laços em suas conexões (JENNIC, 2007).

c) Topologia em malha

Funciona quase da mesma forma que a topologia em árvore, porém é permitido que haja laços de conexões, como em um grafo completo (JENNIC, 2007).

2.6 *SOCKET*

Diversas aplicações de equipamentos computacionais utilizadas no dia-a-dia precisam se comunicar entre si, na maioria dos casos isso ocorre utilizando-se a rede de Intranet ou Internet, podendo também ocorrer de forma local onde um processo se comunica com outro, estando ambos no mesmo dispositivo. Para realizar essa comunicação geralmente são utilizados protocolos TCP (*Transmission Control Protocol*) ou UDP (*User Datagram Protocol*), e pra isso existe uma API (*Application Program Interface*) que foi criada para gerir toda a parte física dessa comunicação, denominada *Socket* (PANTUZA, 2017).

As aplicações de *sockets* são incontáveis, sendo utilizados em navegadores de Internet para requisitar páginas e arquivos, comunicação entre sistemas e seus bancos de dados, e até em chamadas de telecomunicações. Para identificar cada aplicação e seu respectivo *socket*, o próprio *socket* em conjunto com o sistema operacional utilizam e gerenciam portas, as quais são geralmente numeradas de 0 a 65535. Dessa forma, dois processos distintos não podem estar registrados no mesmo número de porta (STEVE, 2018).

O modelo OSI (*Open Systems Interconnection*) especifica um padrão de redes para criação de protocolos. Esse modelo divide uma pilha de redes em 7 camadas, cada uma com suas responsabilidades. A Internet utiliza como base para todas as suas comunicações a pilha TCP que possui apenas 4 camadas. Baseado no modelo OSI, o TCP é, atualmente, o padrão de comunicação em redes (PANTUZA, 2017).

Considerando a Internet e o TCP, os *sockets* estão entre a camada de transporte e a de aplicações. Estando nesse ponto de intercessão, os mesmos conseguem fazer uma interface entre aplicação e rede de maneira bem transparente, assim, aplicações são implementadas utilizando uma comunicação lógica em que, para tais programas, os mesmos estão se comunicando diretamente um com o outro, mas na prática, os dados da comunicação desses programas estão passando pela

rede para trocar mensagens (PANTUZA, 2017).

Quando se programa utilizando *sockets*, uma abordagem muito comum para esses programas é utilizar um servidor e um ou mais clientes. Para essa abordagem é implementado um programa cliente e um programa servidor. Ambos fazem uso da mesma API de *sockets* (PANTUZA, 2017).

Os *sockets* do tipo TCP são orientados a conexão e tem um canal exclusivo de comunicação entre cliente e servidor. Eles garantem a ordem dos pacotes, são considerados confiáveis e sem perda. No entanto, quando se trata de se recuperar de falhas e perda de pacotes, ele é mais lento (PANTUZA, 2017).

Já os *sockets* do tipo UDP desconsideram os quesitos de ordem de pacotes e recuperação de falhas. No entanto, por ser simples, ele é mais rápido que o TCP para alguns tipos de aplicações (PANTUZA, 2017).

2.7 *THREAD*

Dentro de um dispositivo com um sistema operacional existem a todo o momento vários processos sendo executados e processados, parcialmente de forma a atender todas as execuções em paralelo. Um processo pode ter vários subprocessos que executam parte do seu código em paralelo com o processo em si e seus demais subprocessos. Esses subprocessos criados a partir do processo principal são denominados *Threads* (PILLAI, 2018).

O termo *thread*, quando traduzido, significa fio, e foi utilizado para a representação de subprocessos pelo fato de que esses subprocessos são uma extensão de um processo específico. O núcleo do sistema operacional mantém uma lista dos processos sendo executados e de seus respectivos subprocessos ou *threads*, onde cada um recebe um identificador chamado PID (*Process Identification*), o qual é único para cada processo e para cada *thread* em execução (PILLAI, 2018).

Esse tipo de divisão de processos é muito utilizado em aplicações que envolvem s*ockets*, onde a cada nova conexão, o servidor cria uma *thread* para tratar da solicitação do cliente e continua a esperar chamadas de outros clientes (PILLAI, 2018).

3. MATERIAIS E MÉTODOS

O desempenho exigido para o processamento de sinais de redes sem fio e de renderização de interface gráfica é geralmente alto para um dispositivo central, já que este tem de atender a todos os nós da rede dentro de um tempo aceitável para que nenhum dos sistemas por ele responsáveis deixe de atuar como desejado, ou venha a perder informações importantes, bem como responder aos programas utilizados pelo usuário na interface gráfica. Outro ponto importante são as bibliotecas já existentes em código aberto que tornam a utilização de dispositivos como telas sensíveis ao toque e comunicação serial mais fáceis de se implementar e utilizar. Em vista dessas exigências a plataforma BeagleBone Black da Texas Instruments foi escolhida para ser o computador de bordo do sistema por ter um processador ARM Cortex-A8 de 1 GHz, além de possuir 4 GB de armazenamento e mais dois microcontroladores de 200 MHz para tarefas paralelas, além de já disponibilizar uma interface de comunicação com SPI, UART (*Universal Asynchronous Receiver-Transmitter*), I2C e USB (BEAGLEBOARD, 2014).

Para a implementação da rede de sensores sem fio e atuadores dentro do padrão IEEE 802.15.4 foi escolhido o rádio transceptor MRF24J40MA da Microchip, o qual opera na faixa de frequência de 2,4 GHz por meio de uma antena integrada (PCB, ou *Printed Circuit Board*) com uma taxa de transferência de 250 kbps (*Kilobits per Second*). Além de conter todos os parâmetros definidos no padrão, como a encriptação de 128-bit (AES, ou *Advanced Encryption Standard*), o módulo também possui retransmissão automática de pacotes perdidos e é acoplável a quase qualquer superfície (MICROCHIP, 2014). Para receber os dados a serem transmitidos, esse transceptor se utiliza de uma comunicação SPI a uma frequência máxima de 10 MHz.

Os sensores e atuadores são conectados a um microcontrolador FRDM KL25Z de menor desempenho da NXP, o qual tem a tarefa de emular um sensor e então transmitir os dados fictícios de uma leitura por meio de um transceptor, junto de um parâmetro de identificação único de cada sensor, e para os atuadores, o acionamento de um dispositivo utilizando as informações recebidas do transceptor de como realizar a ação desejada, obedecendo os mesmos códigos de identificação adotados pelos sensores e atuadores da rede cabeada CAN existente hoje nos veículos. A ferramenta Eclipse 4.9 foi utilizada para a programação, juntamente com o depurador da J-Link.

A Figura 4 representa o diagrama de blocos que resume o funcionamento do sistema.

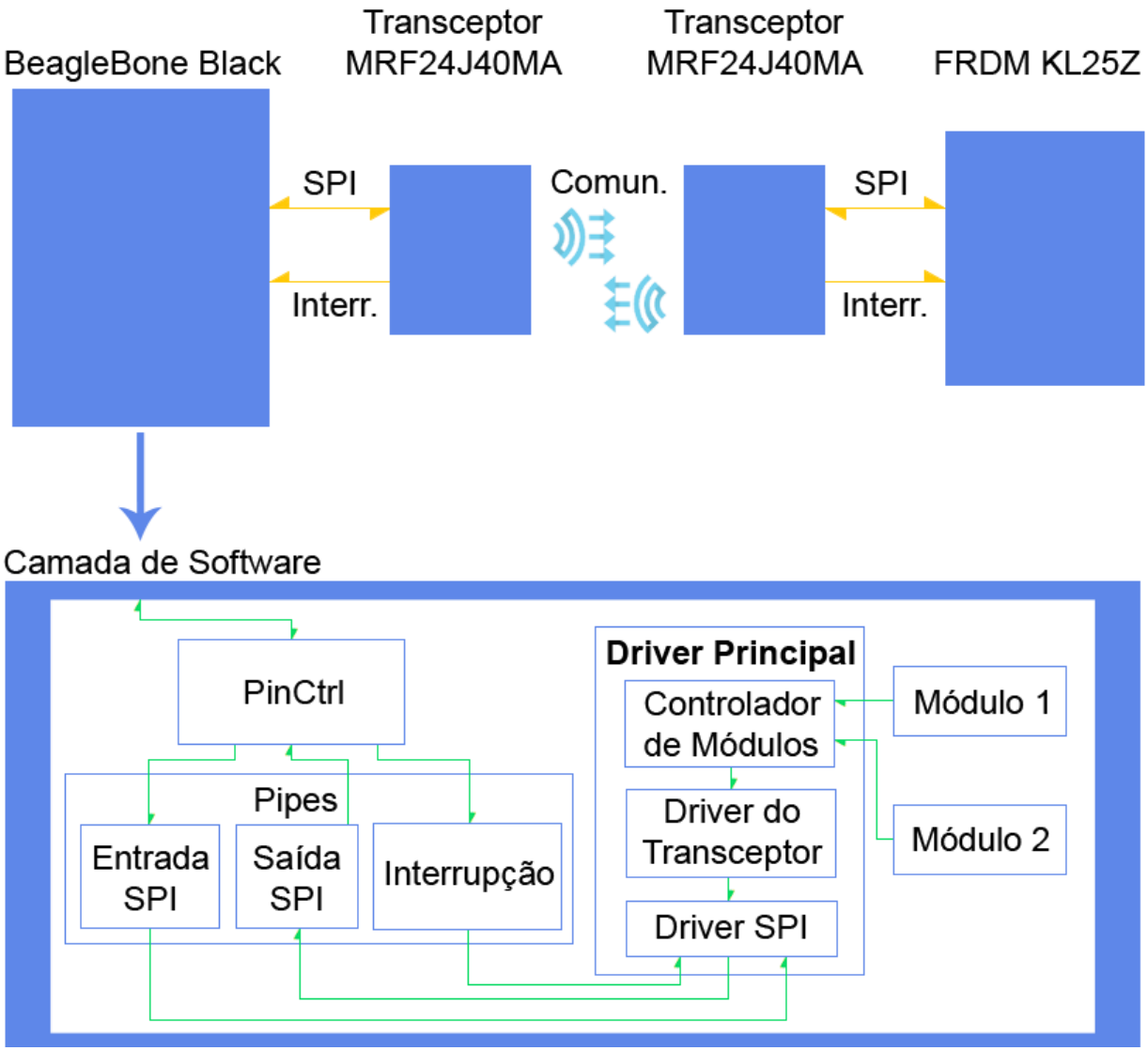


Figura 4 – Diagrama do Sistema.

Fonte: Própria.

O *driver* por sua vez foi implementado utilizando a distribuição padrão de sistema operacional Linux da BeagleBone chamado Debian, a partir da qual foi estabelecido o depurador remoto, e então desenvolvida a comunicação SPI e suas interrupções entre a BeableBone e o transceptor MRF24J40MA por meio do *driver* de plataforma PinCtrl.

Após essa etapa, a lógica da comunicação foi implementada em linguagem C utilizando a *Tool Chain* do Debian remotamente por meio do NetBeans 8. Nessa etapa serão desenvolvidos os *drivers* de SPI, do transceptor e o controlador de módulos.

4. DESENVOLVIMENTO

O dispositivo embarcado utilizado, BEAGLEBONE BLACK, já vem com o sistema operacional Debian, o qual foi utilizado por já possuir a *Tool Chain* correta instalada e pelo fato de que a documentação desse dispositivo também é baseada nesse OS. A partir dessa plataforma, foi desenvolvido um *driver* de dispositivo para sistemas Linux que possibilite receber e enviar pacotes a partir de um transceptor MRF24J40MA. Esse *driver* se utilizou também do *driver* de plataforma PinCtrl para realizar as operações de leitura e escritas nos pinos do microcontrolador.

Nesse capítulo são apresentadas as etapas de desenvolvimento do *driver*, desde a preparação do ambiente de desenvolvimento até os *sub-drivers* desenvolvidos e os resultados obtidos.

4.1 Depurador Remoto

Em um ambiente de desenvolvimento de programas, é normal ocorrerem erros no código (conhecido como *bug*), e muitas vezes esses erros são difíceis de serem detectados, ainda mais quando o código é muito modular. Para tornar o processo de identificação de erros mais rápido, as ferramentas de desenvolvimento possuem uma funcionalidade chamada depurador, o qual, em conjunto com o compilador, pode percorrer o código linha a linha, provendo valores de variáveis em tempo de execução. No entanto, esse método é pouco utilizado em aplicações embarcadas que possuem um sistema operacional, pois desenvolver diretamente no dispositivo é muito demorado, e até as vezes impraticável, devido as limitações de desempenho do mesmo. Levando em consideração essas necessidades e limitações, algumas ferramentas desenvolveram uma forma de realizar a depuração utilizando o compilador do dispositivo embarcado, mas sendo programado e executado de um computador conectado por rede, conhecida como depurador remoto.

Nesse projeto, a ferramenta escolhida para o desenvolvimento em linguagem C foi o NetBeans 8.0.2, mas antes do depurador remoto ser configurado, foi necessário realizar algumas adaptações na BeagleBone para permitir a criação e modificação de arquivos em uma pasta designada ao projeto.

A primeira mudança necessária foi a atualização do sistema operacional da BeagleBone para a versão 7.8 do Debian de 2015, disponível no site da fabricante, o

que consistiu em gravar a imagem em um cartão SD (*Secure Digital Memory*) e inseri-lo na BeagleBone, sendo necessário editar o arquivo de inicialização da imagem para habilitar a regravação do *firmware* (configuração eMMC *Flasher*, ou *Embedded Multimedia Card Flasher*), e então reiniciar o dispositivo para que a gravação ocorresse.

Ao se tentar utilizar o depurador remoto pela rede virtual da USB, não se conseguiu estabelecer uma conexão com o host (BeableBone), então foi necessário configurar um endereço IP (*Internet Protocol*) estático editando o arquivo “interfaces”, localizado no caminho “/etc/network/”, de acordo com a Listagem 1, e conectá-la na rede local.

```
$ nano /etc/network/interfaces

auto eth0
  iface eth0 inet static
    address 192.168.0.82
    broadcast 192.168.0.255
    netmask 255.255.255.0
    network 192.168.0.1
    gateway 192.168.0.1
    dns-nameservers 192.168.0.1 8.8.8.8

$ sudo /etc/init.d/networking restart
```

Listagem 1 – IP estático em Linux.
Fonte: Própria.

Com o *firmware* atualizado e a rede configurada, é possível agora criar uma pasta compartilhada onde o projeto possa ser criado, sendo necessário atribuir privilégios de criação e modificação de arquivos para a pasta compartilhada. Em um ambiente Linux, uma das mais conhecidas ferramentas de compartilhamento de pastas e arquivos é o Samba, o qual pode ser configurado em um único arquivo, conforme a Listagem 2. Para o compartilhamento funcionar de Linux pra Windows, é necessário que algumas configurações sejam alteradas, como também uma outra arquitetura de processadores ARM, chamada Armel, precisa ser instalada já que o Samba tem suporte apenas a essa arquitetura antiga.

```
$ mkdir /app
$ sudo chmod 777 /app
$ nano /etc/samba/smb.conf

...
wins support = yes
...
security = user
...
[beagle]
   comment = Beagleboard
   path = /
   guest ok = yes
   browsable = yes
   read only = no


$ smbpasswd -a nobody
$ sudo /etc/init.d/samba restart

$ dpkg --add-architecture armel
$ apt-get update
$ apt-get install libc6:armel
```

Listagem 2 – Privilégios e compartilhamento utilizando o Samba *Share*.
Fonte: Própria.

Como se pode observar na Listagem 2, uma pasta chamada 'app' foi criada para ser a pasta do projeto e, tanto a pasta quanto o compartilhamento foram configurados para serem acessados e alterados sem a necessidade de nenhum privilégio, o que pode ser um risco a segurança do projeto, porém nesse caso nenhuma segurança a nível de arquivos de projeto foi necessária já que não se trata de um produto a ser patenteado. O nome da pasta que aparece na rede foi denominada 'beagle', já que é um compartilhamento do sistema inteiro ('path = /').

Para que o NetBeans possa acessar o projeto, é necessário que a pasta compartilhada esteja mapeada em uma unidade de rede, então o compartilhamento (com caminho de rede '\\BEAGLEBOARD\beagle') foi mapeado como unidade 'Z'.

Com essas etapas concluídas, agora o NetBeans consegue encontrar a pasta do projeto, e também executar o depurador remotamente, bastando apenas configurá-

lo para utilizar o compilador GCC (GNU *Compiler Collection*) da BeagleBone, e abrir o projeto a partir da pasta mapeada.

4.2 Comunicação SPI e Interrupções em Linux

Uma comunicação SPI em Linux pode ser implementada em nível de *kernel* para *drivers*, ou a nível de usuário por programas que necessitam utilizá-las. A nível de *kernel*, um *driver* pode se utilizar de um outro *driver*, próprio do Linux, chamado OMAP (*Open Multimedia Applications Platform*, ou Plataforma Aberta de Aplicações de Multimídias), o qual possui várias versões e é dependente da arquitetura do processador sendo utilizado. No entanto, programar diretamente a nível de *kernel* não é aconselhável, pois a depuração do código é mais complexa do que a nível de usuário, tornando novas implementações em um *driver* mais demoradas. O ideal seria um sistema que tenha o seu *driver*, mas que também possua uma estrutura a nível de usuário para o desenvolvimento de novos módulos, podendo estes então serem facilmente depurados, e então transformados em bibliotecas para serem utilizadas pelo *driver*.

Com esse conceito em mente, primeiramente foram implementados métodos a nível de usuário para gerenciar a comunicação SPI e os transceptores. Porém, para que a SPI funcione nesse nível é necessário que algum *driver* de plataforma já esteja funcionando com as configurações dos pinos do processador e dos tipos de entradas e saídas da comunicação serial, e a versão do sistema operacional utilizada já possui o *driver* de plataforma PinCtrl genérico que torna possível configurar essas portas.

O PinCtrl utiliza arquivos de configuração chamados *Device Tree Overlays* (ou Árvores de Sobreposição de Dispositivos), com uma extensão DTS (*Device Tree Source*) para arquivos fonte e DTB (*Device Tree Blob*) ou DTBO (*Device Tree Blob Object*) para arquivos já compilados. Para se obter acesso a um determinado pino, é necessário conhecer o número da porta em que o mesmo está multiplexado no processador (conhecido por *MUX ID*), sendo que o MUX da BeagleBone Black pode ser encontrado na página da fabricante. Para habilitar o barramento SPI foi criado um arquivo chamado BBB_SPI0.dts, dentro da pasta "/lib/firmware" da BeagleBone, o qual contém os pinos utilizados, junto ao tipo de entrada, o nível de saída e o MUX interno de cada um, conforme mostra a Listagem 3.

```
$ cat > BBB_SPI0.dts
/dts-v1/;
/plugin/;
/{
  compatible = "ti,beaglebone-black";
  part-number = "BBB_SPI0";
  version = "00A0";

  // Porta P9, pinos 17, 18, 21 e 22, utilizando canal SPI_0
  exclusive-use = "P9.22", "P9.21", "P9.18", "P9.17", "spi0";

  fragment@0 {
    target = <&am33xx_pinmux>;
    __overlay__ {
      pinctrl_spi0: pinctrl_spi0_pins {
        pinctrl-single, pins = <
          0x150  0x20   //spi0_sclk (clock)          P9_22 pulldown
          0x154  0x20   //spi0_do   (saida)          P9_21 pulldown
          0x158  0x00   //spi0_di    (entrada)       P9_18 pulldown
          0x15C  0x10   //spi0_cso  (chip-select)   P9_17 pullup
        >;
  }; }; };
  fragment@1 {
    target = <&spi0>;
    __overlay__ {
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&pinctrl_spi0>;
      cs-gpios = <&gpio1 5 0>;

      spi0_0 {
        compatible = "spidev";
        reg = <0>;
        spi-max-frequency = <5000000>; // 5MHz de frequencia SPI
}; }; }; };
```

Listagem 3 – Configuração do *driver* PinCtrl para SPI.

Fonte: Própria.

Após criar o arquivo de configuração (BBB_SPI0.dts) do *driver* SPI, o mesmo foi compilado em um arquivo DTBO, e este por sua vez foi carregado como um

dispositivo físico da porta P9, o qual é criado automaticamente pelo PinCtrl e se encontrada no caminho "/dev/spidev1.0", conforme ilustra a Listagem 4.

```
$ dtc -O dtb -o BBB_SPI0-00A0.dtbo -b 0 -@ BBB_SPI0.dts
$ echo BBB_SPI0 > /sys/devices/bone_capemgr.9/slots
```

Listagem 4 – Compilação e criação do arquivo do dispositivo físico.
Fonte: Própria.

Para monitorar a interrupção da SPI, foi criado um *driver* de eventos do tipo interrupção para o pino 12 da porta P9, demonstrado na Listagem 5. Esse sistema de interrupção se utiliza do PinCtrl apenas para realizar o MUX do pino 12, mas não para controlar a interrupção. Para isso é utilizada a biblioteca da linguagem C chamada Glib, mais especificamente os seus métodos GIO, os quais monitoram o valor dentro de um arquivo (nesse caso, o arquivo descritor do pino 12 mencionado anteriormente), sendo que, se o valor trocar de 0 pra 1 é uma borda de subida da interrupção e, do contrário, é uma borda de descida.

```
$ cat > BBB_GPIO_INTERRUPT.dts
/dts-v1/;
/plugin/;
/ {
   compatible = "ti,beaglebone", "ti,beaglebone-black";
   part-number = "BBB_GPIO_INTERRUPT";
   version = "00A0";
   fragment@0 {
     target = <&am33xx_pinmux>;
     __overlay__ {
       interr_pins: pinmux_interr_pins {
         pinctrl-single, pins = < 0x078  0x37   // entrada GPIO pullup no pino P12 >;
   }; }; };
   fragment@1 {
     target = <&ocp>;
     __overlay__ {
       gpio-keys {
          compatible = "gpio-keys";
          pinctrl-names = "default";
          pinctrl-0 = <&end_stop_pins>;
```

```
            switch_x1 {
                label = "End-stop-X1";
                linux,code = <1>;
                interrupt-controller;
                gpios = <&gpio1 28 0x5>; // modo evento no pino P12 (mapeado 28)
                                         // como entrada (0x4) em pullup (+ 0x1)
                gpio-key,wakeup;
                interrupt-parent = <&gpio1>;
                interrupts = <97 0x10>;
             };
 }; }; }; }; };
```

Listagem 5 – Configuração do *driver* PinCtrl para evento de interrupção.
Fonte: Própria.

Após criar o arquivo de configuração (BBB_SPI0.dts) do *driver* SPI, o mesmo foi compilado em um arquivo DTBO, e este por sua vez foi carregado como uma porta física que recebe sinais de 0 ou 1 na porta P9, conforme ilustra a Listagem 6.

```
$ dtc -O dtb -o BBB_GPIO_INTERRUPT-00A0.dtbo -b 0 -@ BBB_GPIO_INTERRUPT.dts
$ echo BBB_GPIO_INTERRUPT > /sys/devices/bone_capemgr.9/slots
```

Listagem 6 – Configuração do *driver* PinCtrl para evento de interrupção.
Fonte: Própria.

Com os canais de comunicação e interrupção abertos, é possível se utilizar de funções a nível de usuário para ler e escrever no transceptor, como também receber interrupções do mesmo. Assim, com o depurador funcionando e os canais disponíveis, foi criado um projeto em C para serem desenvolvidos os *drivers* da SPI e do transceptor.

4.3 *Driver* SPI

Como abordado anteriormente, para se obter acesso a um dispositivo em Linux é necessário ter o mesmo mapeado em um arquivo dentro do diretório de dispositivos (caminho "/dev/"), isso porquê esses arquivos contém informações, como aquelas configuradas no PinCtrl, utilizadas por *drivers* do sistema que, pela descrição do dispositivo, identificam a forma de envio e recebimento de mensagens, atuando de

acordo. Um desses *drivers*, o *Input/Output Control* (Controle de Entrada/Saída), ou IOCTL, é responsável pela escrita e leitura de dispositivos, implementado para suportar diversos protocolos de comunicação (SPI, I2C, etc.). Com base nesse funcionamento, a inicialização do programa, descrita na Listagem 7, consistiu em obter o descritor do dispositivo "spidev1.0" para a utilização do mesmo pelas funções do *driver*.

```
1 - ch = open("/dev/spidev1.0", O_RDWR);
```

Listagem 7 – Aquisição do descritor do dispositivo SPI.

Fonte: Própria.

Agora funções podem ser criadas para se comunicar pela SPI. A Listagem 8 demonstra a função de leitura da SPI.

```
1 - /* Método que escreve o conteúdo de "buff", com uma quantidade de bytes "buffSize",
2 -  * e espera uma resposta de tamanho "retSize", que será colocada no buffer "ret".
3 -  */
4 - int readSPI(unsigned char * buff, int buffSize, unsigned char * ret, int retSize) {
5 -     struct spi_ioc_transfer xfer[2];          // Vetor de configuração
6 -     unsigned char empty[1] = {0x00};      // Vetor vazio
7 -     int res;                                  // Resposta de sucesso ou falha de envio
8 -
9 -     // Posição zero é a de transferência (pode ser de recepção full-duplex), e é utilizada
10 -    // da mesma forma para a função de escrita "writeSPI"
11 -    xfer[0].tx_buf = (unsigned long) buff; // Buffer de dados de transferência
12 -    xfer[0].rx_buf = (unsigned long) empty;
13 -    xfer[0].len = buffSize;                   // Tamanho do buffer de transferência em bytes
14 -    xfer[0].speed_hz = SPI_SPEED;          // Frequência da SPI (5 MHz)
15 -    xfer[0].cs_change = SPI_CS;            // Estado de ativação do chip-select (0)
16 -    xfer[0].delay_usecs = 0;                  // Atraso entre pacotes
17 -    xfer[0].bits_per_word = SPI_NBITS;    // Tamanho da palavra transmitida (8 bits)
18 -
19 -    // Posição um, como não há nada para transferir, irá receber
20 -    xfer[1].tx_buf = (unsigned long) empty;
21 -    xfer[1].rx_buf = (unsigned long) ret;   // Buffer de dados de recepção
22 -    xfer[1].len = retSize;                    // Tamanho do buffer de resposta em bytes
23 -    xfer[1].speed_hz = SPI_SPEED;
24 -    xfer[1].cs_change = SPI_CS;
```

```
25 -     xfer[1].delay_usecs = 0;
26 -     xfer[1].bits_per_word = SPI_NBITS;
27 -
28 -     /* Chama o IOCTL para comunicação SPI (SPI_IOC_MESSAGE), com um vetor de
29 -      * tamanho 2 (xref[2]), utilizando o descritor do dispositivo (ch). Retorna 0 se
30 -      * for sucesso, e -1 se ocorrer um erro.
31 -      */
32 -     res = ioctl(ch, SPI_IOC_MESSAGE(2), &xfer);
33 -     return res;
34 - }
```

Listagem 8 – Método para escrita na SPI.

Fonte: Própria.

A Listagem 9 mostra as funções utilizadas para ativar a leitura do descritor de interrupções e atribuição de um método genérico para realizar alguma ação assim que a interrupção ocorra.

```
 1 - /* Método utilizado pela thread de interrupção para identificar quando uma IRQ
 2 -  * ocorre. Quando identificada, chama o handler responsável por tratá-la.
 3 -  */
 4 - static void * _interrupt(void * pars) {
 5 -     unsigned char buffer[64];
 6 -
 7 -     while(1) {
 8 -         fread(buffer, 1, sizeof(buffer), inch); // Lê o arquivo de eventos de interrupção
 9 -         (*i_handler)(); // Executa o método passado ao "getInterrupt" pra tratar a IRQ
10 -     }
11 - }
12 -
13 - /* Método utilizado para ativar o recebimento de interrupções SPI, utilizando o
14 -  * método (handler) passado como parâmetro para tratar a interrupção.
15 -  */
16 - void getInterrupt(void (* handler)(void)) {
17 -     pthread_t poll;
18 -     long thaddr;
19 -     int thresp;
20 -
21 -     // Abre o descritor do arquivo de eventos de interrupção (inch)
22 -     inch = fopen("/dev/input/event1", "r");
23 -     if (inch < 0)
```

```
24 -        printf("Erro: abertura da interrupção falhou!!\n");
25 -    else {
26 -        // O método passado é guardado em uma variável global, para poder ser
27 -        // chamado de dentro da thread
28 -        i_handler = handler;
29 -        // Thread para ler continuamente o descritor, assim monitorando a IRQ
30 -        thresp = pthread_create(&poll, NULL, (void *) _interrupt, (void *) thaddr);
31 -    }
32 - }
```

Listagem 9 – Métodos para processamento de interrupção.

Fonte: Própria.

O *driver* SPI se utiliza basicamente desses dois métodos e do método de escrita, que é muito similar ao de leitura. A Listagem 10 mostra todos os métodos contidos na biblioteca, dos quais a maioria se utiliza dos métodos mencionados acima para realizar suas ações, sendo utilizados para possibilitar uma quantidade maior de formas de chamar as funções de escrita e leitura.

```
1 - // Inicializa o descritor da SPI
2 - void initSPI(void);
3 -
4 - // Habilita a interrupção, executando o "handler" quando ocorrer
5 - void getInterrupt(void (* handler)(void));
6 -
7 - // Escreve o conteúdo de "buff", com um tamanho "arraySize" em bytes, na SPI
8 - int writeSPI(unsigned char * buff, int arraySize);
9 -
10 - // Transforma os valores de "values" em bytes, com uma divisão de "valueNBits", e
11 - // escreve na SPI utilizando "writeSPI"
12 - int writeSPIBuffer(unsigned int * values, int arraySize, int valueNBits);
13 -
14 - // Escreve o endereço "addr", dividindo-o em bytes com divisão de "addrNBits", e
15 - // depois escreve o valor "value", dividindo-o em bytes com divisão de "valueNBits"
16 - int writeSPIAddrSingle(unsigned int addr, int addrNBits, unsigned int value,
17 -     int valueNBits);
18 -
19 - // Escreve o endereço "addr", dividindo-o em bytes com divisão de "addrNBits", e
20 - // depois escreve o vetor "values", com tamanho de "arraySize" e dividindo-o em
21 - // bytes com divisão de "valueNBits"
```

```
22 - int writeSPIAddrBuffer(unsigned int addr, int addrNBits, unsigned int * values,
23 -     int arraySize, int valueNBits);
24 -
25 - // Escreve o vetor de bytes "buff", e lê a resposta pro endereço "ret"
26 - int readSPI(unsigned char * buff, int buffSize, unsigned char * ret, int retSize);
27 -
28 - // Escreve o endereço "addr", dividindo-o em bytes com divisão de "addrNBits", e
29 - // depois lê a resposta do tamanho de um byte, e a retorna
30 - unsigned char readSPIAddrSingle(unsigned int addr, int addrNBits);
31 -
32 - // Escreve o endereço "addr", dividindo-o em bytes com divisão de "addrNBits", e
33 - // depois lê a resposta pra "values", com tamanho de "arraySize"
34 - int readSPIAddrBuffer(unsigned int addr, int addrNBits, unsigned char * values,
35 -     int arraySize);
36 -
37 - // Pausa a execução do programa atual em um tempo de "nano" nano segundos
38 - int pauseNanoSec(long nano);
```

Listagem 10 – Métodos utilizáveis do *driver* SPI.

Fonte: Própria.

Com a SPI implementada, agora é possível se utilizar dos métodos descritos anteriormente para qualquer tipo de programa que queira utilizar a SPI em Linux para se comunicar com um dispositivo.

4.4 *Driver* do Transceptor

Para que a API de comunicação possa enviar e receber mensagens de sensores e atuadores, é necessário ter uma biblioteca com métodos que tratem da leitura, escrita e configurações para o transceptor. A Listagem 11 contém alguns dos métodos da biblioteca.

```
1 - // Escreve um valor de dois bytes em um endereço do transceptor.
2 - void writeLong(unsigned short addr, unsigned char value);
3 -
4 - // Escreve um valor de um byte em um endereço do transceptor.
5 - void writeShort(unsigned char addr, unsigned char value);
6 -
7 - // Lê um valor de dois bytes em um endereço do transceptor.
```

```
8 - unsigned char readLong(unsigned short addr);
9 -
10 - // Lê um valor de um byte em um endereço do transceptor.
11 - unsigned char readShort(unsigned char addr);
12 -
13 - // Inicializa as portas de leitura e escrita e configura o transceptor.
14 - int transceiverInit(unsigned char * mac64Address);
15 -
16 - // Realiza um ping em outro transceptor da rede para verificar se está respondendo.
17 - void transceiverPing(unsigned short macAddress);
18 -
19 - // Envia os dados "payload" de tamanho "payload_size" para outro transceptor com
20 - // endereço específico "macAddress", mapeado como nó "node_id".
21 - void transceiverSend(unsigned short macAddress, unsigned char payload_size,
22 - unsigned char *payload, unsigned char node_id);
23 -
24 - // É chamado assim que há uma interrupção na SPI do transceptor, indicando que há
25 - // algo para ler ou que o dado foi enviado com sucesso.
26 - static gboolean interrupt(GIOChannel *channel, GIOCondition condition, gpointer
27 - user_data);
```

Listagem 11 – Objetos da biblioteca do transceptor.

Fonte: Própria.

Os métodos disponíveis na Listagem 11 se utilizam do *driver* de SPI (Seção 4.3) para ler e escrever para o transceptor MRF24J40MA e encapsulam os padrões de leitura e escrita do mesmo.

Agora com os *drivers* de comunicação implementados é possível construir uma interface que possibilite o desenvolvimento de *drivers* específicos para cada sensor e atuador disponível na rede, de forma que estes se comportem como módulos do *driver* principal sendo gerenciados pelo Controlador de Módulos.

4.5 Controlador de Módulos

Com o intuito de facilitar o desenvolvimento e também possibilitar *drivers* proprietários, os *drivers* de cada sensor e atuador serão compilados como um projeto a parte. Por esse motivo, o controlador precisa carregar esses *drivers* já compilados e mantê-los em um vetor de bibliotecas.

Para alcançar essa modularidade foi criada uma estrutura que contém as

instâncias de cada *driver*, conforme Listagem 12.

```
1 - // Objeto que armazena o identificador (nome do arquivo da biblioteca) "moduleid" do
2 - // driver junto a instância da biblioteca do driver "initHandler" e do ponteiro para o
3 - // próximo objeto da lista.
4 - struct Tmod_node {
5 -     unsigned short moduleid;
6 -     void (* initHandler)(void *);
7 -     struct Tmod_node * next;
8 - };
9 -
10 - // Estrutura de inicialização da lista de módulos.
11 - typedef struct {
12 -     int size;
13 -     struct Tmod_node * first;
14 - } Tmodules;
15 -
16 - // Lista de módulos.
17 - Tmodules allModules;
```

Listagem 12 – Estruturas do controlador de módulos.

Fonte: Própria.

A partir destas estruturas é possível construir o método responsável por importar as bibliotecas dos *drivers* compilados. Esse método lê todas as pastas contidas dentro da pasta de módulos, carregando todos os objetos com extensão ".so" e os carrega para o vetor de *drivers* utilizando a biblioteca "*dlfcn*", fazendo uso do método "*dlopen*" para abrir o objeto do módulo, e do método "*dlsym*" para procurar os métodos que são padrões do *driver* principal e que precisam ser implementados em cada *driver* de cada sensor e atuador, constituindo assim uma interface de comunicação entre controlador e módulos.

Após a importação ser feita é criada uma *thread* (um subprocesso independente do processo principal do controlador) para cada módulo, de modo a permitir que cada *driver* se auto gerencie. Para que cada *driver* possa realizar operações de leitura e escrita para seu respectivo sensor ou atuador, são entregues como parâmetros, à função de inicialização, dois métodos do *driver* principal, um para leitura de pacotes já recebidos e outro para armazenamento de pacotes na fila de escritas, a qual se dá por uma lista FIFO (*First In First Out*, ou primeiro a chegar

primeiro a sair) que envia os pacotes em ordem de chegada utilizando o método de envio criado para o transponder, citado na Listagem 9.

A Listagem 13 contém as estruturas da biblioteca dos métodos responsáveis por enviar e ler pacotes da rede de sensores e atuadores para cada *driver* específico que tenha realizado uma chamada de leitura ou escrita na rede.

```
1 - // Estrutura dos pacotes de dados, contendo a id do nó "nodeid", o endereço
2 - // "macaddress" e o dado em sí "data", bem como os ponteiros do anterior e próximo
3 - // da lista.
4 - struct package {
5 -     unsigned short nodeid;
6 -     unsigned short address;
7 -     unsigned char * data;
8 -
9 -     struct package * next;
10 -    struct package * prev;
11 - };
12 -
13 - // Estrutura da lista de pacotes de dados de um módulo específico "moduleid".
14 - struct packageList {
15 -    unsigned short moduleid;
16 -    int packCount;
17 -
18 -    struct package * first;
19 -    struct package * last;
20 -    struct packageList * next;
21 - };
22 -
23 - // Estrutura da lista onde estão todas as listas de todos os módulos (packageList).
24 - struct allPacks {
25 -    struct packageList * first;
26 - };
27 -
28 - // Declaração da lista de todos os pacotes de todos os módulos.
29 - struct allPacks allPackages;
30 -
31 - // O máximo de pacotes que serão armazenados antes que o controlador se obrigue a
32 - // realizar um shift na lista, perdendo-se assim o primeiro pacote da lista que está cheia.
33 - extern const int maxPackages;
34 -
```

```
35 - // Lista de módulos.
36 - Tmodules allModules;
```

Listagem 13 – Estruturas do controlador de módulos.

Fonte: Própria.

A Listagem 14 contém os métodos dessa mesma biblioteca.

```
1 - // Inicializa as listas de pacotes.
2 - void initPackages();
3 -
4 - // Insere um novo pacote na lista de envio "packageList".
5 - void insertPackage(unsigned short module, unsigned short macaddress, unsigned short
6 -                     nodeid, unsigned char * data);
7 -
8 - // Verifica se há um ou mais pacotes aguardando a leitura de um módulo específico.
9 - unsigned char * readData(unsigned short module);
10 -
11 - // Insere um novo módulo na lista para futuras leituras.
12 - void insertModule(unsigned short module);
```

Listagem 14 – Métodos do controlador de módulos.

Fonte: Própria.

Todas essas bibliotecas citadas nesse item compõem o controlador de módulos, o qual controla a escrita e leitura de pacotes para os *drivers* específicos, assim como instancia os mesmos em suas respectivas *threads*.

4.6 *Driver* Principal

Para que o *driver* proposto neste trabalho, ou *Driver* Principal, funcione é necessário que todos os *drivers* auxiliares e o controlador de módulos, citados nos itens anteriores, trabalhem juntos para que seja possível ter um meio de comunicação dinâmico para ser utilizado pelos *drivers* específicos de cada sensor e atuador da rede, garantindo assim uma certa independência de processamento e controle para cada um dos mesmos, como também criando um padrão de comunicação na rede.

Porém é necessário que haja uma forma de interagir com o usuário para que seja possível apresentar informações úteis extraídas da rede, como por exemplo a temperatura ou a rotação por minuto, medidas essas obtidas de determinados

sensores. Para isso foi criado uma interface de comunicação que responde a chamadas de sistema, como por exemplo um terminal do Linux.

A Listagem 15 contém a biblioteca do comunicador, responsável por atender as chamadas externas, direcionando essas chamadas a seus *sub-drivers* específicos.

```
1 - void * communication_handler(void * pars) {
2 -    unsigned short modId;
3 -    char request[MAX_PACK_SIZE], *token, *mArgs[100], *response = "";
4 -    int mLen = 0, reserve = 0, socket = (int) pars;
5 -
6 -    // Lê a mensagem do cliente, e se certifica de que é um texto.
7 -    int len = read(socket, request, MAX_PACK_SIZE);
8 -    request[len] = 0;
9 -
10 -   // Procura pelo ID do modulo (-m) e constrói um vetor de parâmetros para o modulo.
11 -   while ((token = strsep(&request, "-;-"))) {
12 -      if (strcmp(token, "-m") == 0 || reserve == 1) {
13 -         if (reserve == 0) {
14 -            reserve = 1;
15 -         } else {
16 -            modId = token;
17 -            reserve = 0;
18 -         }
19 -      } else {
20 -         mArgs[mLen] = token;
21 -         mLen++;
22 -      }
23 -   }
24 -
25 -   // Se encontrou o ID, então procura o modulo na lista e chama a função de resposta.
26 -   if (modId != NULL) {
27 -      struct Tmod_node * module = allModules.first;
28 -
29 -      while (module) {
30 -         if (module->moduleid == modId) {
31 -            response = module->respHandler(mArgs);
32 -            break;
33 -         }
34 -         module = module->next;
35 -      }
```

```
36 -     }
37 -     write(socket, response, sizeof(response));
38 -     close(socket);
39 - }
```

Listagem 15 – Biblioteca do comunicador (communicator.h).

Fonte: Própria.

Na Listagem 16 é apresentado um exemplo de uma estrutura padrão de um *sub-driver*, em que se faz obrigatória a implementação dos métodos "*Init*", "*Responder*" e "*GetId*", precedidos pelo nome da própria biblioteca que são utilizados pelo *driver* principal. Essa biblioteca pode conter, além dos métodos exigidos pela plataforma, quaisquer outros métodos que se façam necessários ao módulo, e que geralmente serão utilizados pelo método "*Init*", já que o mesmo é mantido como uma *thread* dentro do *Driver* Principal.

```
1 - // Identificador correspondente ao código do sensor ou atuador da nomenclatura da
2 - // rede CAN.
3 - unsigned short modID = 0xb0c;
4 -
5 - // Objeto que vai conter os ponteiros para as funções de leitura e escrita na rede,
6 - // sendo essas as funções "sendData" e "readData" respectivamente.
7 - ModArgs * handlers;
8 -
9 - // Método principal do sub-driver, responsável por manter o mesmo em funcionamento
10 - // dentro de uma thread e também de inicializar este sub-driver.
11 - void * autoDemoInit(void * modArgs);
12 -
13 - // Método responsável por atender a chamadas da API para este sub-driver específico,
14 - // que será chamado pelo "communicator.h".
15 - void * autoDemoResponder(char *args[100]);
16 -
17 - // Método responsável por retornar o código identificador do sensor ou atuador dentro
18 - // da nomenclatura da rede CAN.
19 - void * autoDemoGetId(unsigned short *idHere);
```

Listagem 16 – Exemplo de biblioteca de um *sub-driver* denominada "autoDemo" (autoDemo.h).

Fonte: Própria.

A Listagem 17 mostra o cabeçalho do *Driver* Principal, com todas as suas

bibliotecas e o endereço do transmissor principal, além das funções de inicialização do servidor de *socket* TCP e, caso já tenha sido iniciado, a inicialização de cada *thread* de clientes da API que venham a solicitar alguma informação.

```
20 - #include <stdio.h>
21 - #include <unistd.h>
22 - #include <string.h>
23 - #include <stdlib.h>
24 - #include "nodes.h" // Nós que armazenam o endereço e os pacotes a serem enviados
25 - #include "transceivercom.h" // Driver do transmissor
26 - #include "communicator.h" // Comunicador que atende aos sockets dos clientes
27 - #include "packages.h" // Biblioteca de pacotes a serem enviados pela rede de sensores
28 - #include "modloader.h" // Carregador de módulos
29 - #include "packetHandler.h" // Controlador dos pacotes da biblioteca de pacotes acima
30 - #include <sys/types.h>
31 - #include <sys/socket.h>
32 - #include <arpa/inet.h>
33 - #include <netinet/in.h>
34 - #include <netinet/tcp.h>
35 - #include <netdb.h>
36 -
37 - #define EUI_7 0xCA
38 - #define EUI_6 0xBA
39 - #define EUI_5 0x60
40 - #define EUI_4 0x89
41 - #define EUI_3 0x50
42 - #define EUI_2 0x16
43 - #define EUI_1 0x77
44 - #define EUI_0 0x84
45 -
46 - // Porta do socket em que o Driver Principal atende a chamadas externas.
47 - #define SERV_PORT 30003
48 - // Endereços do transmissor principal, do ping na rede e o de difusão, respectivamente.
49 - unsigned char mac64Address[8]={EUI_0,EUI_1,EUI_2,EUI_3,EUI_4,EUI_5,EUI_6,EUI_7};
50 - unsigned short mac64AddressPing = 0x0001;
51 - unsigned short mac64AddressBroadcast = 0xFFFF;
52 - unsigned char macAddr = 0x00;
```

Listagem 17 – Cabeçalho do *Driver* Principal.

Fonte: Própria.

A Listagem 18 mostra a estrutura do *Driver* Principal, que inicializa todos os

seus componentes e aguarda por chamadas externas por informações.

```
1 - int main(int argc, char **argv) {
2 -   char * semName = "driversema";
3 -
4 -   // O Driver Principal já está instanciado? Verifica por meio de um semáfaro único.
5 -   errno = 0; // Flag (#define) de erro da própria biblioteca de semáfaros.
6 -   if (sem_open(semName, O_CREAT | O_EXCL, "0777", 3) == SEM_FAILED && errno ==
    EEXIST) {
7 -     // Sim, então se comporta como um cliente para o processo do Driver Principal
8 -
9 -     int client_sockfd;
10 -    struct sockaddr_in serv_addr;
11 -    char *serv_host = "localhost";
12 -    struct hostent *host_ptr;
13 -
14 -    if ((host_ptr = gethostbyname(serv_host)) == NULL) {
15 -      perror("Erro: Processo nao encontrado."); exit(1);
16 -    }
17 -
18 -    if (host_ptr->h_addrtype != AF_INET) {
19 -      perror("Erro: Enderecador invalido."); exit(1);
20 -    }
21 -
22 -    // Configurações básicas do socket
23 -    bzero((char *) &serv_addr, sizeof(serv_addr));
24 -    serv_addr.sin_family = AF_INET;
25 -    serv_addr.sin_addr.s_addr = ((struct in_addr *) host_ptr->h_addr_list[0])->s_addr;
26 -    serv_addr.sin_port = htobe16(SERV_PORT);
27 -
28 -    client_sockfd = socket(AF_INET, SOCK_STREAM, 0);
29 -
30 -    if (client_sockfd < 0) {
31 -      perror("Erro: Nao foi possivel abrir o socket."); exit(1);
32 -    }
33 -
34 -    if (connect(client_sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
35 -      perror("Erro: O driver nao aceitou a conexao."); exit(1);
36 -    }
37 -    // Constrói um texto com os parâmetros para o Driver Principal com separadores
```

```
38 -     char * request = "";
39 -     int i;
40 -     for (i = i; i < argc; i++) {
41 -        strcat(request, argv[i]);
42 -        strcat(request, "-;-"); // Esse separador foi escolhido por ser não usual
43 -     }
44 -
45 -     // Escreve o comando para o driver principal
46 -     write(client_sockfd, request, sizeof(request));
47 -
48 -     // Lê e retorna a resposta
49 -     // MAX_PACK_SIZE é definido na biblioteca do communicator.h como 100 KBytes
50 -     char response[MAX_PACK_SIZE];
51 -     int len = read(client_sockfd, response, MAX_PACK_SIZE);
52 -     response[len] = 0;
53 -
54 -     close(client_sockfd);
55 -     return response;
56 -
57 -  } else { // O Driver Principal ainda não foi instanciado
58 -     unsigned char buf, send[2] = {0x12, 0x71};
59 -     int i, ret;
60 -
61 -     // Inicializa a SPI
62 -     initSPI();
63 -
64 -     // Inicializa o Rádio Transmissor
65 -     ret = RADIO_ERROR;
66 -     while(ret == RADIO_ERROR) {
67 -        ret = transceiverInit(mac64Address);
68 -
69 -        if (ret == RADIO_ERROR) {
70 -           printf("Falha de Comunicacao\n");
71 -           sleep(2);
72 -        }
73 -     }
74 -
75 -     // Inicializa a lista de transmissão para sensores/atuadores
76 -     initNodes();
77 -
78 -     // Inicializa a lista de pacotes recebidos
```

```
79 -    initPackages();
80 -
81 -    // Inicializa o importador de módulos
82 -    initHandles();
83 -
84 -    // Aqui começa a parte de inicialização do comunicador
85 -    int server_sockfd, thread_sock, server_sock, clilen, opt = TRUE;
86 -    struct sockaddr_in cli_addr, serv_addr;
87 -
88 -    // Cria um socket e o configura para ser reutilizado a cada nova conexão
89 -    clilen = sizeof(cli_addr);
90 -    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
91 -    setsockopt(server_sockfd,SOL_SOCKET,SO_REUSEADDR,(char *)&opt, sizeof(opt));
92 -
93 -    if (server_sockfd < 0) {
94 -       perror("Erro: Nao foi possivel abrir o socket."); exit(1);
95 -    }
96 -
97 -    // Realiza as configurações básicas do socket, como endereço, tipo e porta
98 -    bzero((char *) &serv_addr, sizeof(serv_addr));
99 -    serv_addr.sin_family = AF_INET;
100 -   serv_addr.sin_addr.s_addr = htobe32(INADDR_ANY);
101 -   serv_addr.sin_port = htobe16(SERV_PORT);
102 -
103 -   // Abre o socket e o fixa a seu endereço de memória
104 -   if (bind(server_sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
105 -      perror("Erro: Nao foi possivel abrir o socket na porta correta."); exit(1);
106 -   }
107 -
108 -   sleep(2);
109 -   listen(server_sockfd, 5); // Inicia o aguardo a novas conexões no socket
110 -
111 -   // Mantém o Driver Principal ativo, esperando por chamadas externas
112 -   while(1) {
113 -      server_sock = accept(server_sockfd, (struct sockaddr *) &cli_addr, &clilen);
114 -
115 -      if (server_sock < 0) {
116 -         perror("Erro: A conexao do cliente falhou.");
117 -      } else {
118 -         thread_sock = malloc(4);
119 -         *thread_sock = server_sock;
```

```
120 -
121 -             // Se comunica com o cliente que abriu a chamada por meio de uma thread
122 -             pthread_create(NULL, NULL, communication_handler, &thread_sock);
123 -         }
124 -     }
125 -   }
126 -
127 -   return 0;
128 - }
```

Listagem 18 – Estrutura do *Driver* Principal (awd.c).
Fonte: Própria.

Com essa última estrutura o *driver* proposto nesse trabalho se dá por completo, com todos os seus *sub-drivers* e bibliotecas trabalhando em conjunto pra formar um controlador modular e útil a qualquer outro programa que deseje utilizá-lo por chamadas de sistema.

4.7 Sensor Emulado na FRDM KL25Z

Essa plataforma foi obtida já com um sistema RTOS (*Real-Time Operational System*, ou Sistema Operacional de Tempo Real) provido pelo orientador desse trabalho, o que facilitou o escalonamento de tarefas, sendo apenas necessário construir a lógica do transceptor e das respostas ao *driver* construído na BeagleBone Black, de forma a emular um sensor na rede. A Listagem 19 apresenta os métodos de comunicação com o transceptor.

```
1 - // Seta o canal de comunicação utilizado entre os transceptores
2 - void mrf24j40_set_channel(uint8_t ch);
3 -
4 - // Seta o identificador do transceptor
5 - void mrf24j40_set_panid(uint8_t *id);
6 -
7 - // Seta um endereço do transceptor
8 - void mrf24j40_set_short_mac_addr(uint8_t *addr);
9 -
10 - // Seta um endereço longo do transceptor
11 - void mrf24j40_set_extended_mac_addr(uint8_t *addr);
```

```
12 -
13 - // Obtém o valor de um endereço do transceptor
14 - void mrf24j40_get_short_mac_addr(uint16_t * addr);
15 -
16 - // Obtém o valor de um endereço longo do transceptor
17 - void mrf24j40_get_extended_mac_addr(uint64_t * addr);
18 -
19 - // Seta a potência de transmissão
20 - void mrf24j40_set_tx_power(uint8_t pwr);
21 -
22 - // Retorna o estado  do transceptor
23 - uint8_t mrf24j40_get_status(void);
24 -
25 - // Envia um pacote para um outro transceptor na rede
26 - int32_t mrf24j40_set_txfifo(const uint8_t * buf, uint8_t buf_len);
27 -
28 - // Recebe um pacote de outro transceptor na rede
29 - int32_t mrf24j40_get_rxfifo(uint8_t * buf, uint16_t buf_len);
```

Listagem 19 - Métodos de comunicação com o transceptor na FRDM KL25Z.
Fonte: Própria.

No método de envio de pacotes foi deixado fixo o endereço do transceptor da BeagleBone Black, de forma a simplificar a emulação, e a partir disso foram feitos testes com o depurador da J-Link no Eclipse 4.9 para monitorar os pacotes.

4.8 Resultados Obtidos

Para que os resultados possam ser verificados, se faz essencial o funcionamento do depurador remoto, o qual foi configurado para se conectar com o usuário “root” no IP “192.168.7.2”, que é o endereço criado pelo *driver* USB já existente na BeagleBone Black. Como essa emulação por USB de uma rede é mais lenta do que a rede normal, foi preciso aumentar o tempo de espera para 60 segundos na configuração do depurador remoto do NetBeans para que o mesmo conseguisse se conectar com sucesso.

A Figura 5 mostra a conexão bem-sucedida entre o GCC da BeagleBone e o NetBeans.

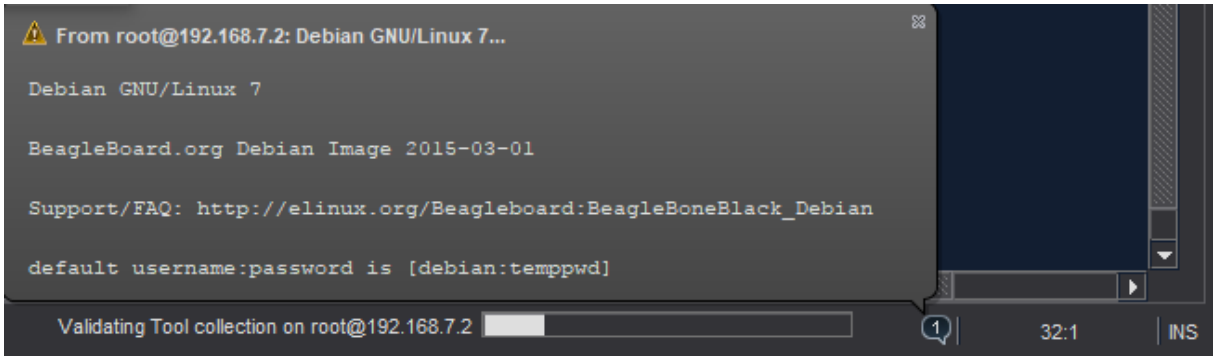


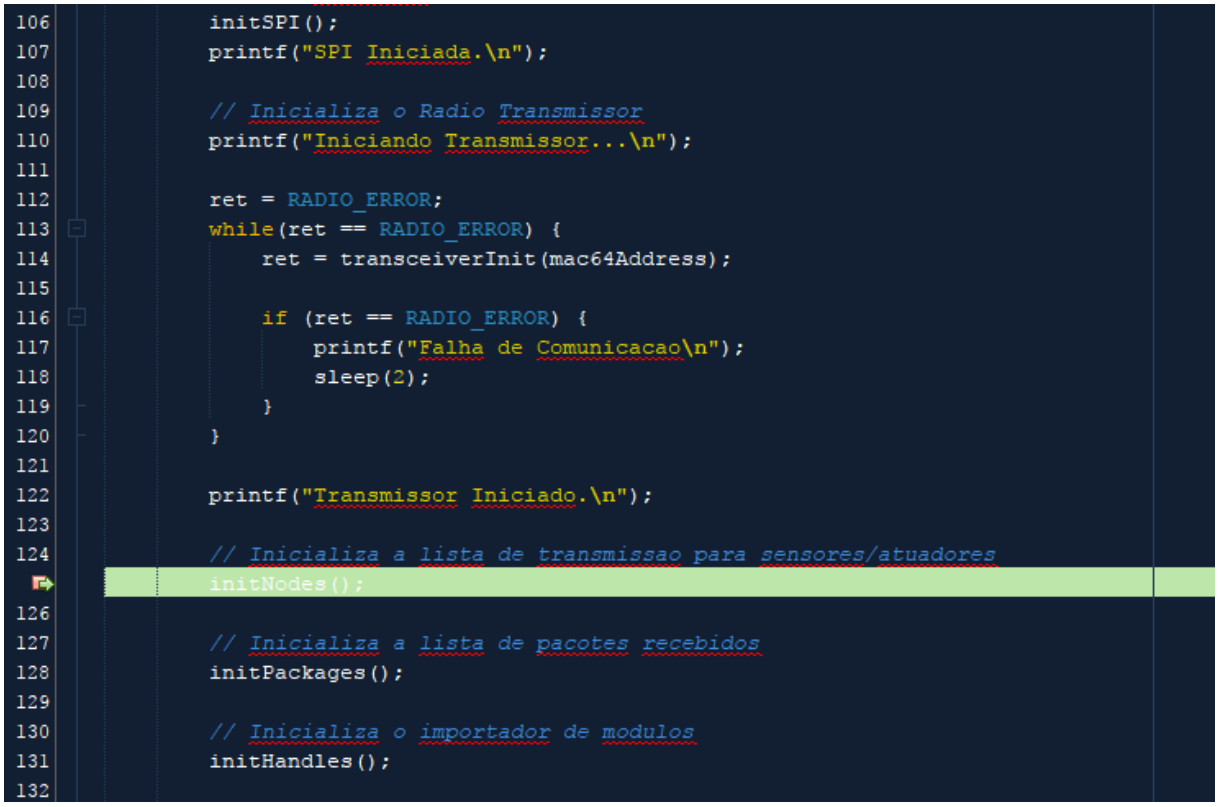
Figura 5 – Conexão bem-sucedida do depurador remoto.
Fonte: Própria.

Com esse resultado se torna possível verificar se a API está realmente aguardando resposta, de forma que a Figura 6 ilustra o *breakpoint* de depuração colocado logo após o transceptor MRF24J40MA ser configurado. Se o transceptor se encontrar com problemas ou a configuração for feita de forma equivocada, o código não sairá do loop (linha 113) mostrado também na Figura 6.

```
106         initSPI();
107         printf("SPI Iniciada.\n");
108
109         // Inicializa o Radio Transmissor
110         printf("Iniciando Transmissor...\n");
111
112         ret = RADIO_ERROR;
113         while(ret == RADIO_ERROR) {
114             ret = transceiverInit(mac64Address);
115
116             if (ret == RADIO_ERROR) {
117                 printf("Falha de Comunicacao\n");
118                 sleep(2);
119             }
120         }
121
122         printf("Transmissor Iniciado.\n");
123
124         // Inicializa a lista de transmissao para sensores/atuadores
            initNodes();
126
127         // Inicializa a lista de pacotes recebidos
128         initPackages();
129
130         // Inicializa o importador de modulos
131         initHandles();
```

Figura 6 – *Breakpoint* de inicialização do transceptor.
Fonte: Própria.

A Figura 7 demonstra que o transceptor foi configurado corretamente e passou do teste, possibilitando assim as comunicações entre o *Driver* Principal e seus sensores e atuadores.



```
106         initSPI();
107         printf("SPI Iniciada.\n");
108
109         // Inicializa o Radio Transmissor
110         printf("Iniciando Transmissor...\n");
111
112         ret = RADIO_ERROR;
113         while(ret == RADIO_ERROR) {
114             ret = transceiverInit(mac64Address);
115
116             if (ret == RADIO_ERROR) {
117                 printf("Falha de Comunicacao\n");
118                 sleep(2);
119             }
120         }
121
122         printf("Transmissor Iniciado.\n");
123
124         // Inicializa a lista de transmissao para sensores/atuadores
            initNodes();
126
127         // Inicializa a lista de pacotes recebidos
128         initPackages();
129
130         // Inicializa o importador de modulos
131         initHandles();
132
```

Figura 7 – Transceptor inicializado com sucesso.
Fonte: Própria.

Depois que o transceptor foi inicializado com sucesso e está pronto para comunicações, se faz necessário que a API de comunicação funcione apropriadamente, aguardando conexões por *socket* em uma determinada porta para atender as chamadas de sistema que podem ser feitas por outros programas.

A Figura 8 mostra um *breakpoint* inserido na linha onde a API fica aguardando conexões, de forma que, se o depurador chegou até essa linha, então o *socket* foi aberto com sucesso na porta escolhida (linha 153 da Figura 8), nesse caso, a porta 30003.

```
147         // Configuracoes do socket
148         bzero((char *) &serv_addr, sizeof(serv_addr));
149         serv_addr.sin_family = AF_INET;
150         serv_addr.sin_addr.s_addr = htobe32(INADDR_ANY);
151         serv_addr.sin_port = htobe16(SERV_PORT);
152
153         if (bind(server_sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
154             perror("Erro: Nao foi possivel abrir o socket na porta correta.");
155             exit(1);
156         }
157
158         sleep(2);
159         listen(server_sockfd, 5);
160
161         while(1) {
162             server_sock = accept(server_sockfd, (struct sockaddr *) &cli_addr, &clilen);
163
164             if (server_sock < 0) {
165                 perror("Erro: A conexao do cliente falhou.");
166             } else {
167                 thread_sock = malloc(4);
168                 thread_sock = server_sock;
169
170                 pthread_create(NULL, NULL, communication_handler, &thread_sock);
171             }
172         }
173     }
174
175     return 0;
176 }
177
```

Figura 8 – Demonstração de que a API está pronta e esperando chamadas.
Fonte: Própria.

Antes da inicialização de API há uma etapa apenas para carregar os *sub-drivers*, ou módulos, de forma a utilizar bibliotecas dinâmicas (.so) para que possam ser lidas e seus métodos importados no *driver*. Essa etapa é realizada pelo Controlador de Módulos.

A Figura 9 mostra os valores das variáveis carregadas na memória e seus valores, de forma que pode ser observado os métodos “autoDemoInit”, “autoDemoResponder” e “autoDemoGetId”, os quais são os métodos padrões e obrigatórios de cada *sub-driver*. Os ponteiros para esses métodos são então inseridos em um novo objeto de estrutura de *sub-drivers* (estrutura “Tmod_node”, linhas 77 a 80 da Figura 9).

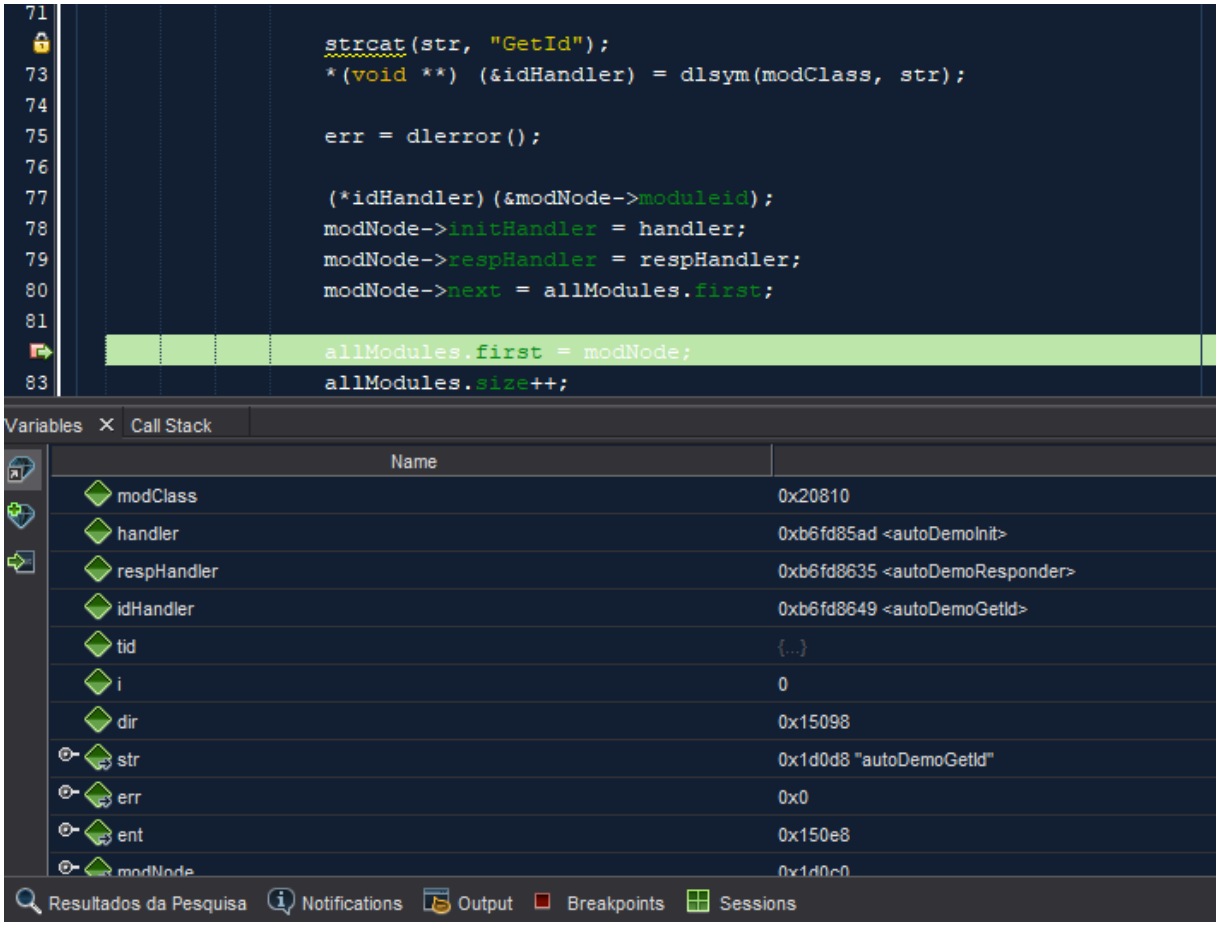


Figura 9 – Controlador de módulos carregando um *sub-driver* com sucesso.

Fonte: Própria.

Todas essas demonstrações mostram que os objetivos propostos nesse trabalho foram atingidos.

5. CONCLUSÃO

O objetivo desse trabalho foi atingindo, com o resultado sendo um *driver* genérico e modular para dispositivos com comunicação *wireless* que operem com a mesma tecnologia de transceptor utilizada nesse trabalho e que tenham, em seu modelo de construção, um controlador principal, o qual controla todos os dispositivos ao seu alcance que estejam de acordo com a interface de *sub-drivers* criada nesse projeto.

A ideia original deste trabalho era fazer, além do *driver*, um porte do mesmo para o sistema operacional AGL porém, até a data de publicação deste trabalho, não há uma distribuição desse sistema que funcione na plataforma BeagleBone Black, e por esse motivo não foi possível realizar esse porte.

O resultado obtido com esse projeto permite a automação de processos que hoje são feitos de forma mecânica (como abrir portas, controlar atuadores de diversos tipos, por exemplo) ou também substituir automações cabeadas por uma rede sem fio. Além disso, por ser desenvolvido em um sistema operacional Linux, as manutenções e melhorias que possam vir a serem feitas no *driver* ou em seus *sub-drivers* terão mais recursos de bibliotecas prontas do próprio sistema ou de terceiros, e podem ser depuradas em tempo real.

O código mencionado neste trabalho é de código aberto e, como uma forma de contribuir para toda a comunidade de desenvolvimento, o mesmo será disponibilizado no GitHub, na pasta remota "ChaosEvil/autoWirelessDriver.git".

6. REFERÊNCIAS

AGARWAL, Tarun. **Embedded Systems Role in Automobiles with Applications**, 2016. Disponível em <https://www.edgefx.in/importance-of-embedded-systems-in-automobiles-with-applications>. Acesso em 02 dez. 2018.

AUTOMOTIVE GRADE LINUX. **Automotive Grade Linux Delivers Open Automotive Software Stack for the Connected Car**, 2014. Disponível em:<https://automotive.linuxfoundation.org/news/2014-06-30/automotive-grade-linux-delivers-open-automotive-software-stack-for-the-connected-car>.Acesso em: 08 out. 2014.

BEAGLEBOARD. **BeagleBone Black**, 2014. Disponível em: <http://beagleboard.org/BLACK>. Acesso em: 26 de nov. 2014.

BMW. **In-car Technology - Automotive Sensors**, 2008. Disponível em: <http://www.bmweducation.co.uk/academy/downloads/L1_Auto_Sensors_Accessible.pdf>. Acesso em: 12 out. 2014.

CALBET, Xavier. **Writing device drivers in Linux: A brief tutorial**, 2006. Disponível em: <http://www.freesoftwaremagazine.com/articles/drivers_linux>. Acesso em: 08 de dez. 2014.

COMPUTER HOPE. **Embedded Linux**, 2014. Disponível em: <http://www.computerhope.com/jargon/e/embedded-linux.htm>. Acesso em: 08 out. 2014.

ELETRONICS TOTURIALS. **Temperature Sensor Types**, 2014. Disponível em: <http://www.electronics-tutorials.ws/io/io_3.html>. Acesso em: 22 de out. 2014.

EMBARCADOS. **Linux Device Drivers - Diferenças entre Drivers de Plataforma e de Dispositivo**, 2014. Disponível em: <https://www.embarcados.com.br/linux-device-drivers/>. Acesso em: 13 dez. 2018.

EMBEDDED LINUX WIKI. **Toolchains**, 2018. Disponível em: <https://elinux.org/Toolchains>. Acesso em: 02 dez. 2018.

GASCÓN, David. **IEEE 802.15.4**, 2008. Disponível em: <http://sensor-networks.org/index.php?page=0823123150>. Acesso em: 17 de nov. 2014.

GOODWILL COMMUNITY FOUNDATION. **Computer Basics: Understanding Operating Systems**, 2014. Disponível em: <http://www.gcflearnfree.org/computerbasics/2>. Acesso em: 08 out. 2014.

GRANDE, Paulo Campo. **Novas tecnologias: Carros atuais têm até 100 sensores a bordo**, 2018. Disponível em: <https://quatrorodas.abril.com.br/noticias/novas-

tecnologias-carros-atuais-tem-ate-100-sensores-a-bordo/>. Acesso em: 02 dez. 2018.

HUBPAGES. **Understanding Car Computers, Sensors and Actuators**, 2011. Disponível em: <http://arksys.hubpages.com/hub/Understanding-Car-Sensors-and-On-board-Computers>. Acesso em: 11 out. 2014.

JAYARAMAN, Krishna. **Connected Cars in a Connected Era**, 2014. Disponível em: <https://www.automotiveworld.com/articles/connected-cars-connected-era>. Acesso em: 02 dez. 2018.

JENNIC. **ZigBee Topologies, Topology Overview**, 2007. Disponível em: <http://www.jennic.com/elearning/zigbee/files/html/module2/module2-2.htm>. Acesso em: 13 out. 2014.

LINUX INFORMATION PROJET. **Kernel Definition**, 2005. Disponível em: <https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Kernel_(computer_science).html>. Acesso em: 08 dez. 2014.

LINUX TRAINING ACADEMY. **Linux Distribution Introduction and Overview**, 2018. Disponível em: <https://www.linuxtrainingacademy.com/linux-distribution-intro>. Acesso em: 02 dez. 2018.

MARTIN, Jim. **Linux for cars: Automotive Grade Linux rivals CarPlay and Android Auto**, 2014. Disponível em: <http://www.pcadvisor.co.uk/news/linux/3530657/linux-for-cars-automotive-grade-linux/>. Acesso em: 08 out. 2014.

MICROCHIP. **MRF24J40MA Datasheet**, 2014. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/70329b.pdf>. Acesso em: 27 de nov. 2014.

PANTUZA, Gustavo. **O que são e como Funcionam os Sockets**, 2017. Disponível em: <https://blog.pantuza.com/artigos/o-que-sao-e-como-funcionam-os-sockets>. Acesso em: 14 dez. 2018.

PETRUCELLI, John. **Ford, Apple, Google and the Race to Phone and Car Connectivity**, 2018. Disponível em: <https://www.motus.com/ford-apple-google-and-the-race-to-phone-and-car-connectivity>. Acesso em: 02 dez. 2018.

PILLAI, Sarath. **Difference Between Process And Thread in Linux**, 2018. Disponível em: <https://www.slashroot.in/difference-between-process-and-thread-linux>. Acesso em: 14 dez. 2018.

ROUSE, Margaret. **Sensor**, 2014. Disponível em: <http://searchdatacenter.techtarget.com/definition/sensor-network>. Acesso em: 11 out. 2014.

STEVE. **TCP/IP Ports and Sockets Explained**, 2018. Disponível em:

<http://www.steves-internet-guide.com/tcpip-ports-sockets>. Acesso em: 14 dez. 2018.

THE GREEN BOOK. **Actuators**, 2013. Disponível em: <http://www.thegreenbook.com/actuators.htm>. Acesso em: 13 de nov. 2014.

THE LINUX FUNDATION. **Automotive Grade Linux Enables Telematics and Instrument Cluster Applications with Latest UCB 6.0 Release**, 2018. Disponível em: <https://www.linuxfoundation.org/press-release/2018/10/automotive-grade-linux-enables-telematics-and-instrument-cluster-applications-with-latest-ucb-6-0-release/>. Acesso em: 13 dez. 2018.

WALZ, Eric. **Toyota Adopts Open Source Automotive Grade Linux for the 2018 Camry**. Disponível em: <http://www.futurecar.com/1047/Toyota-Adopts-Open-Source-Automotive-Grade-Linux-for-the-2018-Camry>. Acesso em: 13 dez. 2018.