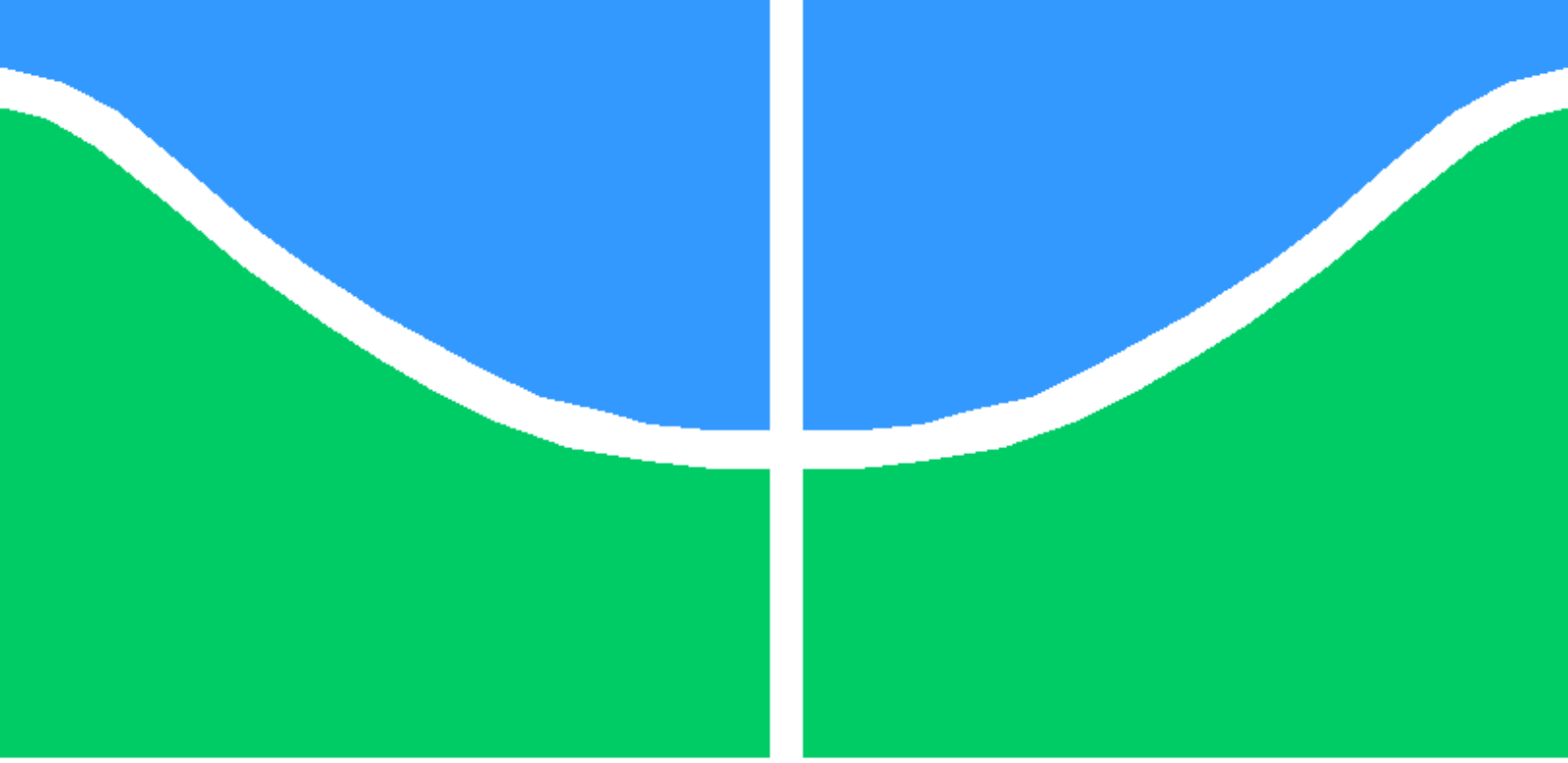

DESENVOLVIMENTO DE SISTEMA DE COMUNICAÇÃO MULTIPLATAFORMA PARA VEÍCULOS AÉREOS DE ASA FIXA

HENRIQUE CARNEIRO PITA

TRABALHO DE CONCLUSÃO DE CURSO EM ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

DESENVOLVIMENTO DE SISTEMA DE COMUNICAÇÃO MULTIPLATAFORMA PARA VEÍCULOS AÉREOS DE ASA FIXA

HENRIQUE CARNEIRO PITA

Orientador: PROF. DR. HENRIQUE CEZAR FERREIRA, ENE/UNB

TRABALHO DE CONCLUSÃO DE CURSO EM ENGENHARIA ELÉTRICA

BRASÍLIA-DF, 07 DE DEZEMBRO DE 2018.

Resumo

Neste trabalho foi incorporado um computador embarcado a um sistema de veículos autônomos não tripulados (VANTs) e implementado um sistema para a comunicação entre três VANTs e uma estação terra. Iniciamos as atividades trabalhando com o computador embarcado, instalamos um sistema operacional baseado em linux específico para este dispositivo e testamos algumas funções básicas que poderiam ser utilizadas posteriormente. Prosseguimos com um estudo sobre o funcionamento do modem, que seria utilizado para realizar a comunicação entre os sistemas, e projetamos um *hardware* para realizar a conexão entre o computador embarcado e o modem. Após a montagem de um protótipo do *hardware* prosseguimos com a implementação de um *software* para a comunicação dos dispositivos utilizando o protocolo MAVLink. Por último o trabalho é finalizado com a realização de um teste para medir o atraso da comunicação entre os modens.

Abstract

In this work an embedded system was incorporated into an unmanned autonomous vehicle (UAV) and a system was implemented for the communication between three UAVs and a ground station. We started the activities working with the embedded computer, installed a linux-based operating system specific to this device and tested some basic functions that could be used later. We proceeded with a study on the operation of the modem, which would be used to perform the communication between the systems, and we designed a hardware to make the connection between the embedded computer and the modem. After assembly of a hardware prototype, we proceed with the implementation of a software to communicate the devices using the MAVLink protocol. Finally the work is finished with the accomplishment of a test to measure the delay of the communication between the modens.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE CÓDIGOS FONTE

LISTA DE TERMOS E SIGLAS

CI	Circuito Integrado
COM	Computer-On-Module
DMA	Direct Memory Access
DSR	Data Set Ready
DTC	Data Carrier Detect
DTR	Data Terminal Ready
FD	File Descriptor
GPIO	General Purpose Input/Output
MAVLink	Micro Air Vehicle Link
PCB	Printed circuit board
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read-Only Memory
RTS/CTS	Request to Send / Clear to Send
RX	Receptor
RXD	Receiver Data
SCM	System Control Module
SDK	Software Development Kit
STX	Start Of Text
TRM	Technical Reference Manual
TX	Transmissor

TXD	Transmiter Data
UART	Universal Asynchronous Receiver/Transmitter
VANTs	Veículos Aéreos Não Tripulados

Capítulo 1

Introdução

O desenvolvimento de sistemas artificiais autônomos faz parte dos desejos da sociedade a anos. Apesar de ainda ser tratada como ficção científica na maioria dos casos, já está presente em diversas aplicações do cotidiano e continua sendo alvo de investimentos tecnológicos. Podemos destacar como exemplo de sistemas artificiais autônomos os veículos autônomos ou, mais especificamente, Veículos Aéreos Não Tripulados (VANTs) ou *drones*.

Estes robôs aéreos foram inicialmente desenvolvidos para fins militares com o objetivo de serem utilizados em missões de alto risco para humanos, como, por exemplo, missões que envolviam reconhecimento de território e ataques a territórios inimigos. Com o passar do tempo ocorreu a inevitável difusão da tecnologia para o meio civil, assim diversas outras aplicações surgiram como o auxílio em resgates, coleta de informações para a agricultura, lazer, competições e, a aplicação mais comum, obtenção de imagens aéreas. Além disso, as pesquisas continuam apontando para novos usos da tecnologia. A empresa de tecnologia *Amazon.com, Inc*, por exemplo, estuda e testa a possibilidade de utilizar VANTs para realizar entregas e encomendas.[1]

1.1 Objetivo

Este trabalho é uma continuação das atividades do Laboratório de Robótica Aérea da Universidade de Brasília, em especial as atividades desenvolvidas em [1] e [2] que posteriormente serão mais discutidas. Esse trabalho faz parte de um projeto do CNPq que pretende possibilitar que três aeronaves de asa fixa presentes no laboratório voem, autonomamente, em formação de esquadrilha. Atualmente as aeronaves já estão equipadas com um sistema de piloto automático e são capazes de realizar voo individual de forma autônoma. Apesar de algumas funcionalidades ainda não estarem funcionando em condições ideais, neste trabalho, foi adicionado um computador embarcado ao sistema de modo que possamos processar as informações obtidas das outras aeronaves e de uma estação terra e determinar as ações da

[1]Fonte: `https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011`

Figura 1.1: Imagem com uma aeronave semelhante às aeronaves presentes no laboratório de robótica aérea.

aeronave.

A figura 1.1 contém um avião semelhante a um dos aviões presentes no laboratório de robótica aérea. O trabalho realizado aqui será aplicado também a essa aeronave.

O objetivo deste trabalho é possibilitar a comunicação entre os diversos dispositivos que compõem a aeronave, entre os aviões e a estação terra. A abordagem para a solução deste problema foi o desenvolvimento de um módulo que envolve um *hardware* e um *software* que juntos permitem a comunicação entre todas essas plataformas.

Assim restará a trabalhos futuros a implementação de tarefas específicas após o recebimento das mensagens e programar a lei de controle no microprocessador para que ele ordene o piloto automático a manter-se na posição em que pertença à formação em esquadrilha.

Existem trabalhos internacionais que como o trabalho presente em [3] realizaram tarefas similares as quais estamos fazendo neste projeto. No trabalho feito em [3] é modelado, identificado, simulado e testado uma lei de controle para que duas aeronaves voem em formação de esquadrilha na configuração de líder virtual. No trabalho citado as aeronaves utilizadas foram totalmente desenvolvidas durante o projeto na *West Virginia University* nos Estados Unidos.

Ainda comentando sobre o trabalho citado, a abordagem utilizada para elaboração das leis de controle foi um projeto de controle de realimentação interna e externa. Nessa configuração de projeto a malha interna controla os aspectos da dinâmica da aeronave enquanto a malha externa preocupa-se com sua posição com relação ao líder virtual. A figura 1.2[2] ilustra o esquema de realimentação descrito.

Com as aeronaves que estão sendo montadas neste projeto pretende-se validar e aplicar

[2]Imagem original obtida em [2]

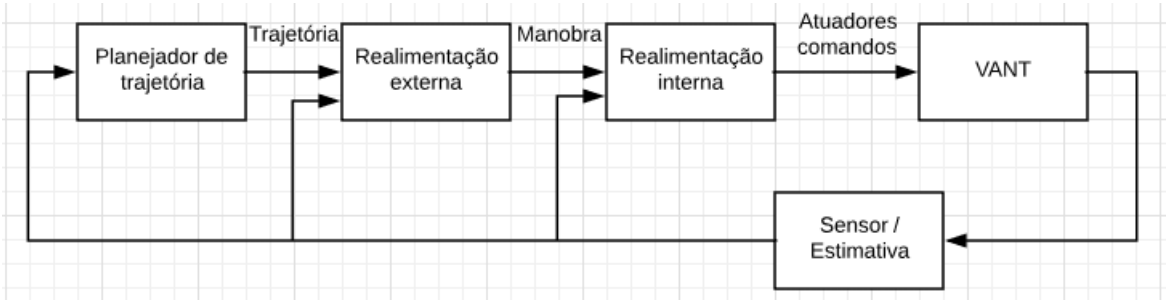

Figura 1.2: Diagrama de blocos de sistemas de controle em cascata.

o recente desenvolvimento da tese de doutorado do Thiago Cordeiro ([2]). No trabalho citado foram desenvolvidos controladores para que as aeronaves possam acompanhar uma outra aeronave líder evitando colisão enquanto a formação a qual estão contidas se altera, ou não, no tempo. É importante destacar que nos trabalhos desenvolvidos em [2] as informações relativas aos dados do líder são conhecidas por todos os outros aviões porém os outros aviões não precisam de comunicação bidirecional com todas as outras aeronaves.

Por fim é evidente que não é a primeira vez que um sistema de comunicação para veículos aéreos é desenvolvido. Em [4] foi realizado o desenvolvimento e teste de um sistema de comunicação utilizando dois VANTs. Nesse trabalho o objetivo principal era que o alcance da comunicação da estação terra fosse ampliado. No trabalho citado caso um dos VANTs não recebe a mensagem enviada pela estação terra o VANT que recebeu a mensagem irá retransmiti-la ao primeiro. No trabalho citado também foram desenvolvidos um módulo composto de *hardware* e *software* para realizar a comunicação das aeronaves. Observe que apesar do objetivo ao final do trabalho citado ser bem distante do objetivo final deste trabalho o foco de ambos será o desenvolvimento de um sistema de comunicações multiplataforma para veículos aéreos.

1.2 Organização do trabalho

Este trabalho foi dividido em três partes.

A primeira composta pelos capítulos 2 e 3 tratam sobre o computador embarcado que será utilizado neste trabalho. Mais especificamente estes capítulos falam sobre a instalação do sistema operacional, conhecimentos essenciais para manipular o sistema, como implementar a comunicação serial e utilizar o GPIO do computador embarcado.

A segunda etapa composta pelos capítulos 4 e 5 são referentes aos componentes externos ao computador embarcado que também serão incorporados à aeronave. Nestes capítulos é apresentada a operação dos modens, indicado alguns registradores que foram utilizados na configuração dos modens, são discutidos aspectos das interfaces de comunicação entre computador embarcado e modem, é projetado um módulo para conectar o controlador, o computador embarcado e o modem e, por último, é testado um protótipo para o prossegui-

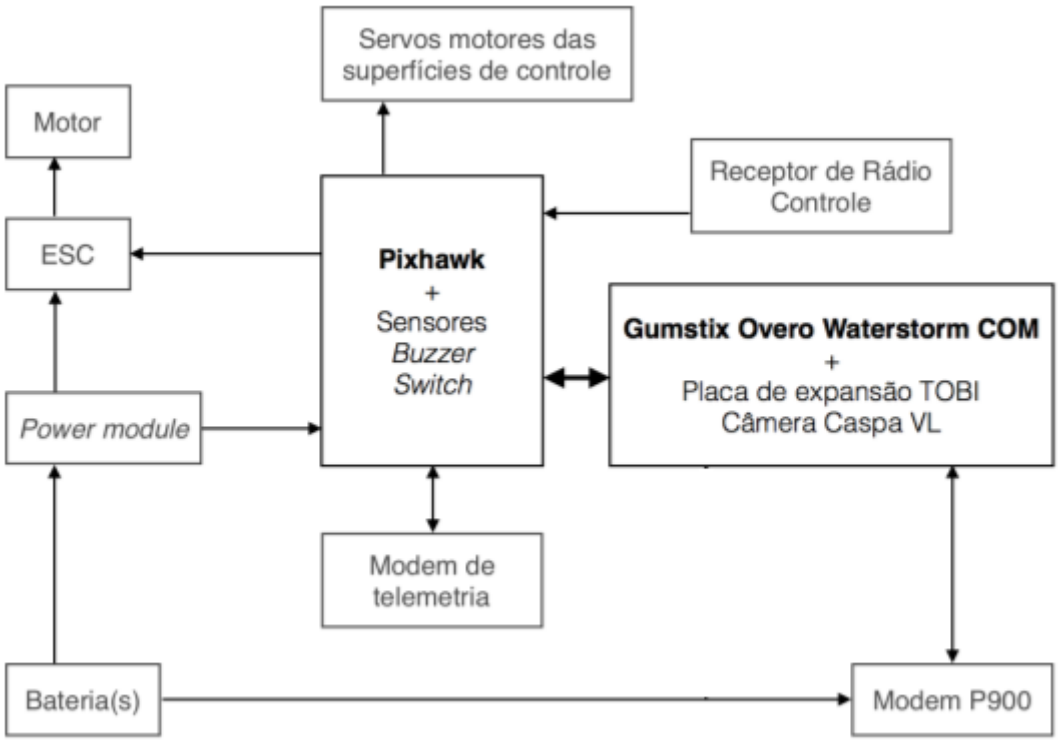

Figura 1.3: Diagrama de blocos dos principais componentes dos aviões.

mento das atividades.

A última etapa composta pelo capítulo 6 trata da implementação do *software* que irá operar nas aeronaves. Nesse capítulo é discutido o protocolo de comunicação MAVLink, é implementado a parte essencial de um código para utilizar este protocolo e é finalizado com um teste da comunicação que, como resultado, nos fornece dados relativos ao atraso da comunicação entre os modens.

1.2.1 Ponto de partida

As atividades de montagem e preparação das aeronaves foram iniciadas no trabalho de graduação de Eduardo Rocha ([1]). No trabalho citado foram identificados e detalhados os principais componentes utilizados nas aeronaves, tais como a estrutura, os sensores, o piloto automático (Pixhawk), o computador embarcado e o modem, além disso ocorreu a efetiva montagem da aeronave e foram realizados voos individuais autônomos como teste, porém não foram inclusos ao sistema da aeronave no trabalho citado o computador embarcado nem o modem para comunicação entre as aeronaves. Por fim a referência indica a utilização da plataforma "RT-MaG"e Xenomai que haviam sido instaladas no computador embarcado durante as atividades de [1], porém não foram utilizados neste trabalho, esse tema será mais discutido no capítulo 2.

A figura 1.3[3] ilustra a conexão dos principais componentes dos aviões. É importante

[3]Imagem original disponível em [1]

mencionar que a montagem da Pixhawk e dos outros dispositivos já foi feita em [1] e, portanto, neste trabalho nos dedicaremos ao computador embarcado, ao modem e ao *hardware* que irão compor o sistema de comunicação da aeronave.

Capítulo 2

Primeiros Passos

No capítulo 1 foi explicitado que este trabalho é a continuação das atividades realizadas em [1]. O autor do trabalho citado afirma que o próximo passo para o procedimento das atividades é dedicar-se ao computador embarcado uma vez que ele é o único dispositivo que ainda não foi implementado no sistema das aeronaves.

O computador embarcado será adicionado à aeronave de modo a comunicar-se com a Pixhawk (sistema do piloto automático) e um modem que transmitirá informações de outros computadores embarcados que estarão em situações semelhantes em outras aeronaves. O microprocessador deverá processar as informações obtidas através do modem e repassar instruções ao piloto automático Pixhawk. Neste capítulo será explicitado os primeiros passos e procedimentos que tiveram de ser realizados para operar o microprocessador.

2.1 Computadores embarcados

Computadores embarcados são processadores digitais acoplados a diversos periféricos com o objetivo de tratar uma atividade pré determinada. A pré determinação da sua atividade permite que os computadores embarcados possuam menor tamanho, peso, preço e capacidade de processamento do que computadores comuns que desempenham funções similares.

Contudo existem desvantagens para sua aplicação causadas pelo mesmo motivo que os tornam atrativos. Por serem bem específicos geralmente estes dispositivos não podem ter sua função inicial alterada sem mudar boa parte de seu *software* ou estrutura do *hardware*, em alguns casos estes dispositivos chegam a não apresentar interface com o usuário.

Em nosso caso mais especificamente o computador embarcado escolhido foi o Overo® *WaterStorm Computer-On-Module*(COM), esse sistema embarcado apresenta um processador DM3730 com arquitetura ARM Cortex-A8 e clock de base do processador de até 1 GHz. Além disso, esse computador está acoplado a uma placa de expansão Tobi que acrescenta ao computador embarcado conexões do tipo *display DVI*, *Ethernet*, USB *Host*, USB *OTG*, USB

console, áudio *Stereo* e um segmento com 40 *pin-headers* que podem ser utilizados para a mais diversas funções, como modulação PWM, GPIO, alimentação, conversão analógico digital e comunicação serial. Mais detalhes e especificações podem ser encontrados na loja online do próprio site da *Gumstix*[1].

O computador embarcado (nome genérico) pode ser, portanto, chamado de Overo (nome específico) ou Gumstix (nome da marca).

2.2 Sistema Operacional

Qualquer computador digital com alguma complexidade que exija administração de seus recursos e funções primárias necessita de um sistema operacional. O núcleo ou *kernel* é a parte mais importante e de nível mais baixo de um sistema operacional, ele tem a função de administrar o uso da memória do dispositivo, seus processamentos, seus drivers e sua comunicação. Por exemplo, a simples existência de comunicação USB em um dispositivo já exije que este seja capaz de receber e processar dados enviados por este canal.

Inicialmente foi decidido em trabalhos anteriores que seria utilizada a ferramenta *RT-Mag* em nosso sistema. Essa ferramenta nos ofereceria uma facilidade muito grande no projeto de controladores uma vez que permitiria a interação com ferramentas como Matlab® e Simulink®, além de um núcleo específico para o computador embarcado que vamos utilizar.

A ferramenta *RT-Mag* toma para si muitas das operações necessárias para a operação do nosso sistema o que impossibilita utiliza-lo da maneira que ele foi idealizado, em consequência disto a demasiada simplificação da etapa poderia prejudicar aplicações futuras. Com essa ferramenta seria impossível utilizar o protocolo de comunicação "MAVLink"do piloto automático para comunicação entre os dispositivos ou aeronaves, por exemplo.

Destaca-se ainda a documentação desatualizada, que dificultou a instalação dos componentes da ferramenta como a *toolbox* do Matlab®, que nunca chegou a funcionar, e o sistema operacional do computador embarcado. A complexidade na utilização do sistema aumentava a cada etapa enquanto mesmo as etapas iniciais mais simples ainda não funcionavam adequadamente.

Como dito chegamos a instalar a ferramenta no sistema embarcado entretanto, devido a complicações posteriores à instalação do sistema operacional, optou-se por não mais utilizar essa ferramenta.

Em seguida decidimos utilizar o núcleo oferecido pelo Projeto Yocto® por ser específico para o modelo de computador embarcado utilizado por nós e recomendado pela fabricante. O projeto Yocto é um projeto de colaboração open source que fornece modelos, ferramentas e métodos para ajudar seus usuários a criar sistemas baseados em Linux para sistemas embarcados, independentemente da arquitetura do sistema. Mais detalhes do projeto Yocto

[1] `https://store.gumstix.com/products`

podem ser encontrados em seu endereço eletrônico[2].

Outra opção viável apresentada durante os trabalhos do laboratório foi a utilização do sistema operacional Ubuntu. A vantagem de se utilizar o sistema Ubuntu é que esse é um sistema operacional a partir do núcleo Linux muito difundido que já contém diversos softwares que podem ser úteis para algumas aplicações futuras, ele contém, por exemplo, um compilador o que facilita a criação e execução de códigos simples para testes rápidos. A desvantagem de se utilizar este sistema operacional é que podem ser executadas muitas tarefas desnecessárias que diminuem a especificidade e o desempenho do computador embarcado. Realizamos a instalação deste sistema em um dos computadores embarcados com o intuito de analisar as diferenças entre as duas principais opções de sistemas operacionais. O sistema Ubuntu instalado foi o Ubuntu 15.04 por ser uma versão estável e adaptada para o sistema em questão.

A instalação do sistema operacional não foi algo trivial, além disso existe uma escassez de documentação detalhada e completa que explique como instalar o sistema operacional no computador embarcado, logo será descrito os principais procedimentos necessários para a instalação de um sistema operacional. Na fase atual dos trabalhos instalamos ambos os sistemas, mais tarde podemos decidir qual dos dois sistemas será melhor para nossa aplicação. Nas seções seguintes irei traduzir, comentar e realizar pequenas modificações em tutoriais que podem ser encontrados no próprio site da gumstix[3] e nos repositórios do GitHub do projeto Yocto[4] e Ubuntu para a Gumstix[5].

2.2.1 Obtenção de imagens do sistema operacional

Essencialmente o dispositivo precisa apenas executar um pequeno programa, geralmente localizado em uma mémoria não volátil do tipo *Read-Only Memory* (ROM), para acessar a outro dispositivo de memória não volátil que armazene o sistema operacional, e carregar o sistema operacional na memória volátil de rápido acesso ou *Random Access Memory* (RAM) onde ele poderá ser executado. Em sistemas mais robustos ocorre, na verdade, um encadeamento desses pequenos programas, chamados de *bootloaders*, onde um primeiro estágio executa um segundo estágio que carrega programas mais complexos e, por sua vez, executa um terceiro estágio e assim por diante até que o sistema operacional seja completamente carregado e esteja pronto para ser executado por si só.

Logo, para realizar o processo de iniciação do sistema operacional no computador embarcado, em nosso caso, precisamos de três arquivos específicos, são eles o primeiro estágio do sistema de iniciação, o arquivo *MLO*, o segundo estágio do sistema de iniciação, o arquivo *u-boot.img*, e a imagem do sistema, que em nosso caso será o Yocto 1.8.2 com *kernel* Linux

[2] `https://www.yoctoproject.org`
[3] `https://www.gumstix.com/`
[4] `https://github.com/gumstix/yocto-manifest`
[5] `https://github.com/gumstix/live-build`

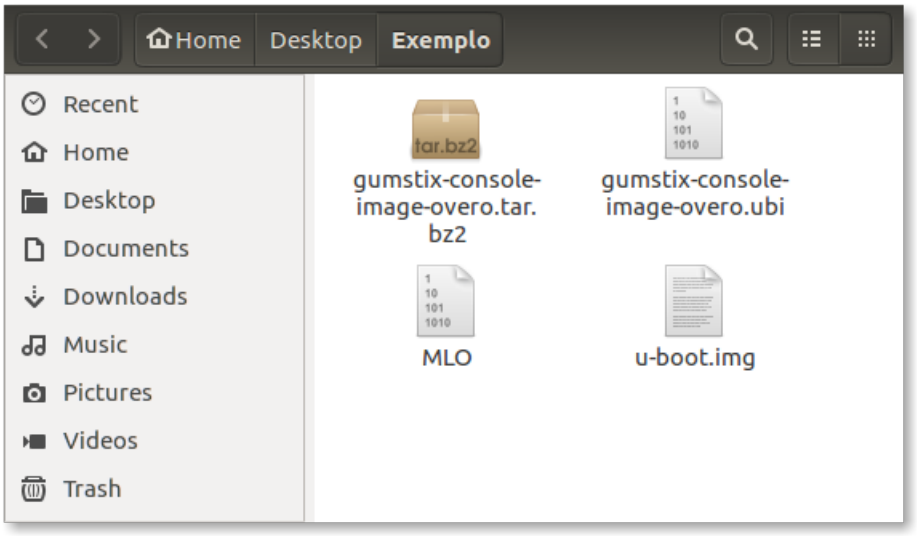

Figura 2.1: Exemplos de arquivos necessários para instalação da imagem.

3.18. As sigla *MLO* e *u-boot*, vem de *Minimal Loader* e *Universal Bootloader* respectivamente.

A figura 2.1 mostra um exemplo dos arquivos descritos no parágrafo anterior, observe que, neste caso, há também uma pasta compactada que contém os arquivos raiz do sistema operacional. O modo mais simples encontrado para se obter esses arquivos e a imagem do sistema operacional é seguindo os passos do arquivo "*Readme*" do repositório do projeto Yocto, o endereço já foi apresentado na seção 2.2. A vantagem de se utilizar esse método ao invés de simplesmente obter a imagem pronta do sistema operacional é que caso seja necessário poderemos modifica-la.

Esse tutorial explica como obter alguns *scripts* que podem ser úteis na montagem da imagem e realiza todos os procedimentos através de linhas de comando do terminal do Linux. Portanto para executar essa etapa é altamente recomendado a utilização de uma das versões do Linux indicadas pelo projeto Yocto. Essas versões do Linux podem ser encontradas, junto de mais informações úteis no manual de referência do projeto Yocto[6], mais especificamente no item *1.3.1 Supported Linux Distributions*. A versão do sistema operacional utilizada nas atividades foi Ubuntu 14.04.

Além disso, recomenda-se atualizar todos os comandos do Linux que serão utilizados nessa seção. Recomendamos, também, que qualquer pessoa que venha a reproduzir esta tarefa executar as linhas de comando presentes na listagem 2.1 uma a uma para que seja mais fácil corrigir eventuais problemas. Além disso, a última etapa do processo baixa alguns *gigabytes* de códigos fonte e os compila, assim é recomendado que haja, pelo menos, 25GB de memória livres no dispositivo que vier a executar os comandos apresentados na listagem 2.1 ou no manual de referência do projeto Yocto.

Listagem 2.1: Linhas de comando Linux para obtenção e montagem da imagem

[6] `https://www.yoctoproject.org/docs/1.7/ref-manual/ref-manual.html`

```
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo >
    repo #Baixa os scripts do repositorio

$ chmod a+x repo #Torna-os arquivos executaveis

$ sudo mv repo /usr/local/bin/ #Move-os para o caminho do sistema

$ repo --help #Se tudo ocorrer bem devera aparecer uma mensagem de
    utilizacao, esse comando nao e obrigatorio

$ mkdir yocto #Cria um diretorio para os arquivos

$ cd yocto #Altera o diretorio de execucao para o novo diretorio

$ repo init -u git://github.com/gumstix/yocto-manifest.git -b fido #
    Seleciona o ramo mais estavel do repositorio

$ repo sync #Baixa os arquivos do repositorio

$ sync #Forca todos os arquivos temporarios a serem escritos em
    dispositivos persistentes

$ export TEMPLATECONF=meta-gumstix-extras/conf

$ source ./poky/oe-init-build-env #Cria ambiente the variaveis para o
    sistema de montagem da imagem

$ bitbake gumstix-console-image #Baixa os codigos fonte e compila as
    imagens do sistema
```

Após a finalização da execução de todos os comandos recomenda-se verificar a pasta **/yocto/build/tmp/deploy/images/overo**, essa pasta deve conter arquivos binários de *kernel* e *bootloaders* e arquivos de diretório raiz no formato **.tar**. Possíveis causas de falha já foram explicadas anteriormente, e, provavelmente, são softwares faltosos ou desatualizados ou sistema operacional não compatível. A figura 2.2 apresenta um exemplo do conteúdo da pasta descrita que deve ser semelhante ao obtido após a execução dos procedimentos anteriores.

Na figura podemos encontrar tanto os *bootloaders* necessários descritos anteriormente como o binário (**.ubi**) e arquivos do diretório raiz de algumas versões do projeto Yocto. A versão utilizada foi a mais recente à época, "gumstix-console-image-overo-20180509042558.rootfs.tar.bz2", entretanto tudo oque foi implementado foi testado também, na versão recomendada, "gumstix-console-image-overo.tar.bz2", portanto as duas imagens podem ser utilizadas. Os *bootloaders* utilizados foram "MLO-overo" e "u-boot-overo.img".

A obtenção das imagens Ubuntu ocorre de maneira semelhante à realizadas para a obtenção da imagem do projeto Yocto com pequenas modificações. Os procedimentos exatos podem ser encontrados no link do GitHub passado no início da seção.

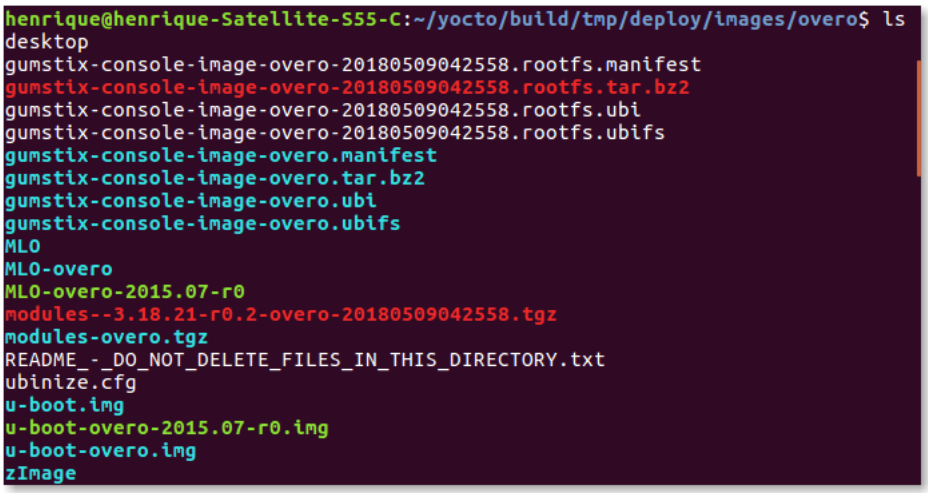

Figura 2.2: Imagens obtidas através dos procedimentos descritos.

Outra opção, como já comentada, é a obtenção da imagem já pronta o que simplifica muito esse procedimento que se resume a baixar apenas três arquivos. Entretanto, nesse caso, não será possível realizar alterações no núcleo o que mais para a frente pode vir a ser um empecilho.

2.3 Instalando o Sistema Operacional

2.3.1 Preparando o Cartão de Memória

Uma vez obtida a imagem do sistema operacional podemos transferir os arquivos para o computador embarcado para, enfim, ligá-lo. Essa tarefa será realizada por meio de um cartão SD que funcionará como o disco rígido do computador embarcado.

Logo, o cartão SD irá conter tanto os programas necessários para boot, que serão utilizados apenas na inicialização do computador, quanto os outros programas, que podem ser utilizados a qualquer momento e realizarão modificações constantes no cartão SD. Portanto a melhor maneira de lidar com essa divisão é particionar o cartão SD em duas partições que serão denominadas **boot** e **rootfs**.

Esse é um procedimento muito comum e existem inúmeras maneiras de fazê-lo, a maneira recomendada pelo fabricante do computador embarcado é utilizar um script que pode ser obtido em seu repositório GitHub[7]. Outro ponto negativo deste script é o excesso de memória alocada para a partição de boot, no caso são reservados 528 MB à partição de boot e utilizam-se menos de 100 MB. Sendo assim, caso futuramente venha a faltar espaço para armazenamento de dados será possível ampliar a partição roots refazendo esta divisão. Essa divisão pode ser modificada alterando-se os valores logo abaixo de "sfdisk" no script.

[7] https://github.com/gumstix/meta-gumstix-extras/blob/dizzy/scripts/mk2partsd

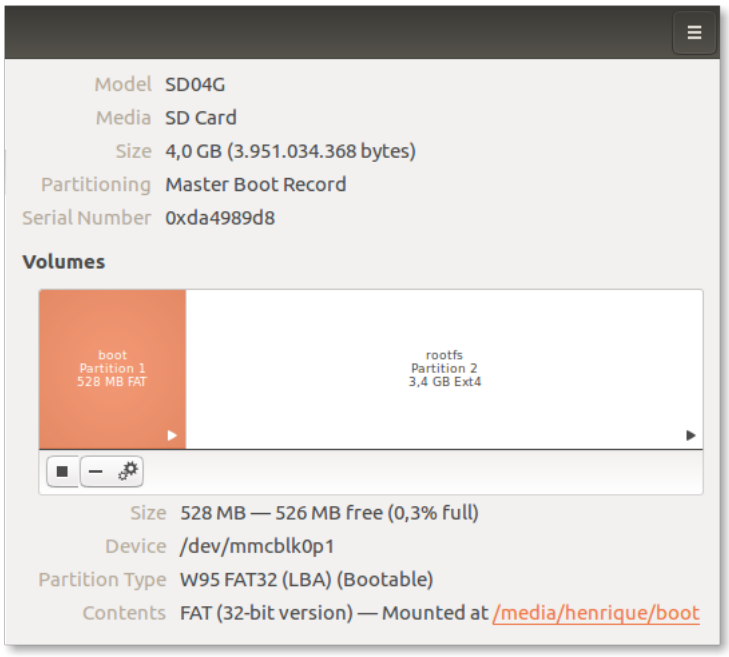

Figura 2.3: Exemplo de divisão de cartão SD.

Os sistemas de gestão de arquivos da partição boot é "**VFAT**" e da partição rootfs é "**ext4**".

O sistema de gestão de arquivos define o método que o sistema operacional irá utilizar para armazenar nos espaços de memória os arquivos e suas informações, ou metadados dos arquivos, como nome, espaço de memória ocupado, datas de alterações e últimos acessos. Existe uma grande variedade de sistemas de gestão de arquivos com as mais diversas complexidades. Mas o que podemos precisar nesse trabalho e em trabalhos futuros é que o sistema "FAT" é um sistema antigo geralmente utilizado em mídias e, normalmente, universal. Já o "ext" é um sistema elaborado especificamente para o Linux e não é possível acessá-lo por um outro sistema operacional sem um programa para essa finalidade.

A figura 2.3 apresenta um exemplo de cartão de memória com as partições já definidas, montadas e contendo o sistema operacional do computador embarcado. No exemplo o cartão SD possui um total de 4 GB, porém, para o projeto Yocto, um cartão de memória de 2 GB deve ser suficiente.

Após dividido o cartão SD podemos prosseguir com a instalação do sistema montando suas partições e copiando os arquivos obtidos na seção anterior, os dois arquivos *bootloaders*, para a pasta em que a partição de boot foi montada e extraindo os diretórios do sistema para a pasta em que a partição rootfs foi montada. Depois disso, lembre-se de desmontar as partições antes de remover o cartão SD.

O procedimento de montagem de uma partição de memória consiste em uma atividade do sistema operacional para garantir que a transferência de informação será feita da maneira correta, basicamente o dispositivo conectado é lido por inteiro para identificar os arquivos nele armazenados e aonde podem ser escritas novas informações sem que haja sobreposição

de dados. Porém mais importante que a montagem da partição é desmontar a partição antes de desconectar o periférico, pois garante que nenhuma atividade de escrita na partição esteja ocorrendo no momento que o dispositivo for removido. Esse procedimento garante, também, que todas as alterações solicitadas tenham sido feitas no periférico e não estejam salvas em arquivos temporários ou *buffers* do sistema. O procedimento descrito pode ser resumido em uma sequência de linhas de comando para o terminal Linux apresentadas na listagem 2.2, novamente é recomendado que cada linha seja executada de cada vez para facilitar a identificação e correção de erros. É recomendado também que seja criado um diretório com todos os arquivos necessários já descritos anteriormente.

Listagem 2.2: Linhas de comando para preparação do cartão SD

```
1  $ sudo ./mk2partsd </dev/mmcblk0>  #Cria cartao sd bootavel
2
3  $ sudo mkdir /media/{boot,rootfs} #Gera diretorios para montagem
4
5  $ sudo mount -t vfat /dev/<mmcblk0p>1 /media/boot
6  $ sudo mount -t ext4 /dev/<mmcblk0p>2 /media/rootfs #Monta as particoes
7
8  $ sudo cp MLO /media/boot/MLO #Cria uma copia do MLO
9
10 $ sudo cp u-boot.img /media/boot/u-boot.img
11
12 $ sudo tar -xjvf <rootfs.tar.bz2> -C /media/rootfs #Extrai arquivos
13 $ sync
14
15 $ sudo umount /media/boot
16 $ sudo umount /media/rootfs #Desmonta particoes
```

Tendo sido todos os comandos da listagem 2.2 executados de maneira correta o cartão SD deve estar pronto para ser utilizado. Agora podemos colocar nosso cartão SD no computador embarcado e prosseguir com a inicialização do sistema.

2.3.2 Conectando-se à Gumstix

O computador embarcado pode ser conectado a um computador regular de diversas maneiras. Nesse trabalho vamos optar por ligá-lo a um computador Linux por simplicidade, porém isso poderia ser realizado por meio de um computador com Windows ou simplesmente um monitor com DVI e teclado USB, por exemplo.

Para ligar o computador embarcado ao computador conecte um cabo USB ao computador e ao USB console da placa de expansão tobi. Feito isso, uma luz verde deve se acender indicando a conexão correta. Em seguida verifique em qual porta de comunicação serial a gumstix foi conectada, no Windows isso pode ser verificado acessando o "Gerenciador de Dispositivos" e em seguida "Portas(COM e LPT)", no Linux basta executar o comando "**dmesg**", ou "**dmesg | grep tty**", a placa Gumstix deve ser a ultima entrada a aparecer.

O comando "dmesg" é um comando que imprime as mensagens núcleo que, na maioria das vezes, são mensagens dos drivers do dispositivo. Quando acrescentamos "grep tty" estamos realizando uma busca nas saídas da função "dmesg" pelo termo "tty" e restringindo a sua saída aquelas mensagens que contém este termo.

Em seguida será necessário executar um programa para emular o terminal, recomenda-se o programa screen, caso ainda não o tenha instalado basta executar a linha de comando "**sudo apt-get install screen**", ou no caso de utilizar o sistema operacional Windows recomenda-se o PuTTY.

Os programas que emulam terminais executam apenas a tarefa de imprimir os caracteres recebidos pela porta serial, ou USB no caso, e enviar por essa mesma porta os caracteres digitados.

Para iniciar o terminal de comunicação com a Gumstix basta executar, por exemplo, a seguinte linha de comando: "**sudo screen </dev/ttyUSB0> 115200**". No caso da linha de comando de exemplo apresentada o termo </dev/ttyUSB0> foi a porta encontrada ao utilizar o comando "dmesg" e "115200" é a velocidade de comunicação em *baud*. Nesse momento a comunicação entre a gumstix e o computador deve ser estabelecida e assim que a gumstix for ligada os caracteres devem começar a ser impressos na tela do computador.

Feito isso a gumstix estará pronta para ser ligada à tomada, porém antes de ligá-la é importante comentar que o fabricante recomenda a limpeza de variáveis da memória flash sempre que iniciar uma nova versão do sistema operacional no computador embarcado pela primeira vez. Para fazê-lo basta interromper o processo de boot antes de seu início no momento em que aparece uma contagem regressiva na tela. Uma vez interrompido o boot do sistema basta executar o seguinte comando "**nand erase 240000 20000**" para limpar as variáveis salvas e "**reset**" para reiniciar o processo de boot.

A figura 2.4 ilustra este procedimento. Os caracteres são impressos rapidamente e a contagem de tempo é de apenas 1 segundo para os núcleos do projeto Yocto, portanto é necessário ficar atento para interromper o processo.

Feito isso o processo de boot deve iniciar e diversas mensagens irão aparecer na tela. É importante verificar, na primeira vez que se inicia o sistema operacional, se nenhuma mensagem de erro aparece e, se tudo ocorrer bem, ao final do processo será exigido uma senha, se o computador embarcado chegou a esse ponto provavelmente tudo esta em ordem. A senha de acesso ao sistema Yocto é **root** e para o sistema Ubuntu **gumstix**, caso necessário a senha é igual ao usuário.

2.3.3 Salvando a imagem do sistema operacional na memória flash

O computador embarcado Overo® WaterSTORM COM da Gumstix® conta com uma memória interna não volátil de 1 GB do tipo Flash, memória suficiente para armazenarmos o sistema operacional. Apesar de o mais recomendado ser continuar usando o cartão SD,

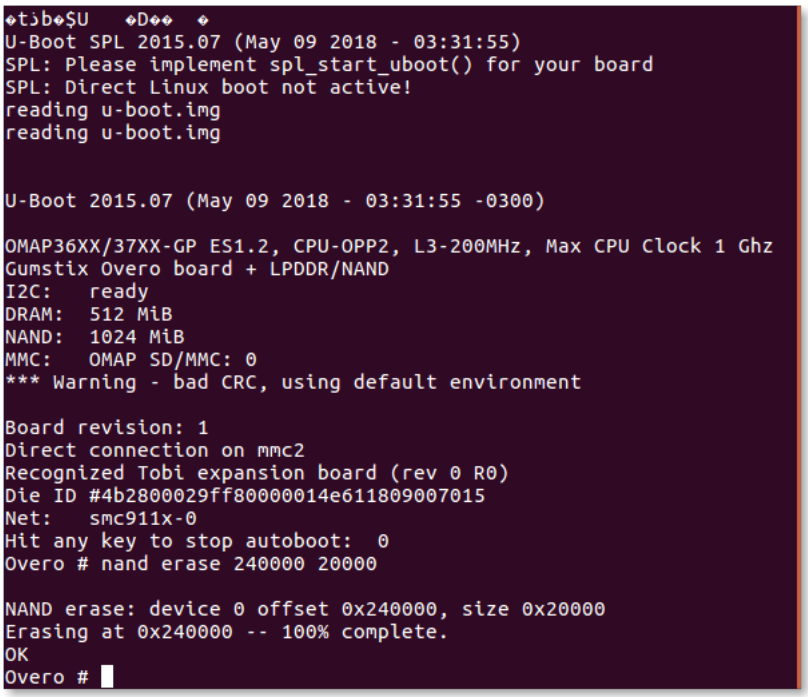

Figura 2.4: Exemplo do procedimento de limpeza de variáveis da memória *flash*.

por possuir mais memória e ser transferido entre dispositivos com mais facilidade, ter o sistema operacional salvo na memória flash do computador embarcado pode ser útil. O site do fabricante descreve quatro maneiras distintas de se realizar este procedimento a maneira que apresentou o melhor resultado foi a última das opções explicadas e é resumida a instalar na memória flash tudo o que foi instalado no cartão de memória e somado ao binário do núcleo através de um script fornecido em seu endereço eletrônico[8]. O script desejado é o último script da seção "**Flashing with U-Boot**".

Uma vez obtido o script é necessário torna-lo executável e adiciona-lo à partição de boot do cartão SD bootável, para isso basta executar e seguinte linha de comando em que "flash-all" é o nome do script "**mkimage -A arm -O Linux -T script -C none -a 0 -e 0 -n "flash-all" -d flash-all.cmd /media/boot/flash-all.scr**". O comando "mkimage" é um comando utilizado para fazer imagens para serem utilizadas pelo "u-boot", as opções do comando e suas explicações são facilmente obtidas digitando "**man mkimage**" no terminal do Linux. Lembre-se de editar os nomes dos arquivos no script para coincidirem com os nomes dos arquivos que serão adicionados a seguir.

Após adicionar o executável do script dentro da **partição** de "boot" do cartão SD devemos, também, armazenar dentro da **pasta** "boot" da **partição** "rootfs" o novo MLO, u-boot.img e o binário do núcleo. Observe que esses *bootloaders* que serão adicionados à **pasta** "boot" não são os mesmos que estão na **partição** "boot" pois estes novos *bootloaders* devem ser específicos para operar da memória flash. Esses novos arquivos podem ser obtidos, como já explicado, do projeto Yocto ou do próprio site do fabricante.

[8] `https://www.gumstix.com/support/faq/write-images-flash/`

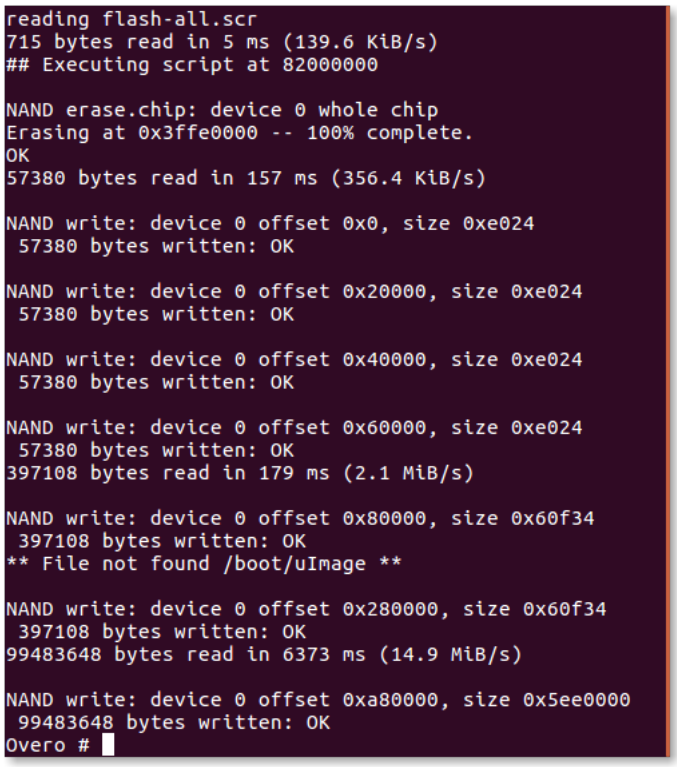

Figura 2.5: Saída obtida após execução dos comandos para escrita na memória *flash* do computador embarcado.

Uma vez que todos os arquivos estiverem salvos no local correto podemos desmontar o cartão SD e colocá-lo no computador embarcado e interromper o processo de boot no momento da contagem regressiva, assim como foi feito na primeira vez em que iniciamos o sistema. Interrompido o processo de *boot* podemos executar a linha de comando "**mmc rescan 0; load mmc 0 $loadaddr flash-all.scr; source $loadaddr**". Essa linha de comando irá executar o script passando os *bootloaders*, o binário do núcleo e os arquivos raiz do sistema operacional para a memória flash do sistema embarcado e as mensagens apresentadas na figura 2.5 devem ser impressas.

Se tudo ocorrer bem, assim que o processo for finalizado e o sistema reiniciado, o computador embarcado deverá iniciar normalmente mesmo que sem o cartão SD inserido.

Capítulo 3

Utilizando o computador embarcado

Neste capítulo serão discutidos conhecimentos básicos necessários para a manipulação do sistema operacional linux e do sistema operacional instalado específico instalado no computador embarcado e, posteriormente, são realizados testes do GPIO e da comunicação serial do computador embarcado. As informações descritas nessas etapas são exaustivamente utilizadas ao longo do trabalho e serão utilizadas por aqueles que vierem a continuá-lo.

3.1 Ambientando-se no Linux

Realizados os procedimentos apresentados na seção anterior de forma correta o computador embarcado estará operando com um sistema operacional Linux muito semelhante ao que estamos habituados em computadores regulares. Como demonstração podemos realizar alguns procedimentos simples para que possamos explorar um pouco o ambiente ao qual vamos trabalhar. O que será demostrado nessa etapa são procedimentos, comandos e informações padrão dos sistemas Linux. Caso o leitor já esteja habituado ao ambiente de trabalho Linux recomenda-se saltar a seção 3.1.

Para obter mais detalhes sobre quaisquer comandos listados aqui basta executar o comando seguido de --help, por exemplo "uname --help", para imprimir uma breve descrição dos comandos seguidos de instruções de uso. Para reduzir a quantidade de conteúdo impresso pode se usar less", por exemplo "ls --help | less".

Primeiro podemos executar o comando "**cat /etc/issue**" no terminal do overo. O comando "cat" lê o conteúdo presente no arquivo indicado e o imprime na tela. Utilizando esse comando deve ser impresso na tela o ramo e a versão do sistema operacional utilizado.

Outros comandos populares que podem vir a ser úteis durante a utilização do sistema são os comandos "**echo**", "**mkdir**", "**rm**", "**cp**" e "**mv**". O comando "echo" escreve algo desejado em um arquivo ou mesmo na tela de comando. O comando "mkdir" cria um novo diretório. O comando "rm" exclui um arquivo ou diretório. o comando "cp" cria uma cópia do arquivo especificado. Por último o comando "mv" move um arquivo para o diretório especificado.

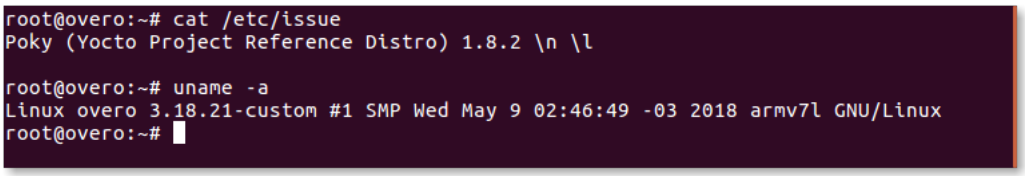

Figura 3.1: Versão do Sistema Operacional e do núcleo.

Um programa que, nessa primeira parte, merece destaque é o programa "**uname**". O "uname" é na verdade um programa que imprime certas informações do sistema, no caso a opção -a" solicita a impressão de todas as informações disponíveis pelo programa, o resultado da execução do comando "*uname -a*" está presente na figura 3.1. Para apagar os caracteres impressos na tela podemos executar o comando "**clear**".

Como podemos ver na figura 3.1, estamos utilizando o ramo Poky do projeto Yocto versão 1.8.2, a versão do núcleo Linux é a 3.18.21 editada para o overo.

Outros dois comandos interessantes são os comandos "**cd**", de *change directory*, e "**ls**", que lista os arquivos e diretórios presentes no diretório destino. Uma opção muito utilizada do comando "ls" é a opção "**ls -la**" que além de listar todos os arquivos e pastas no diretório atual também imprime algumas informações úteis sobre cada um deles.

Na figura 3.2 temos um exemplo de saída do comando "ls -la", nele podemos ver que para cada arquivo é impresso uma linha com várias colunas de informação. Explicar o que cada coluna significa se faz desnecessário, entretanto é importante saber o que as primeiras letras significam, pois muitas vezes essa é a causa de alguns problemas.

Essas 10 primeiras colunas que são compostas por **"-"** e letras variadas indicam as o tipo do arquivo e as permissões dos usuários quanto aqueles arquivos.

Na figura 3.2 a primeira coluna, que é sempre indicada pela letra **"d"**, nós mostra que o arquivo é um diretório, se o arquivo fosse um programa ou um arquivo de texto regular este seria indicado por um **"-"**. As noves letras seguintes podem ser separadas em grupos de 3 indicando as permissões do dono, grupo e outros, respectivamente. As letras **"r"**, **"w"** e **"x"** indicam leitura, escrita e execução, respectivamente. Se analisarmos, portanto, os dados da pasta "usr" veremos que o dono da pasta possui permissão para ler, escrever e executar, porém seu grupo e outros usuários terão permissão apenas para ler e executar.

As permissões podem ser alteradas pelo comando **"chmod"** ou pode se adquirir permissão total utilizando o comando **"sudo"** que habilita permissões totais para o próximo comando.

Passadas essas informações e estes comandos básicos já somos capazes de explorar os arquivos do sistema. Portanto permita-nos migrar para o primeiro diretório do sistema executando "**cd ..**" duas vezes. E em seguida executar o comando "ls -la" para que possamos visualizar as pastas do sistema. Se tudo for executado como explicado devemos obter algo como mostrado na figura 3.2.

```
root@overo:~# cd ..
root@overo:/home# cd ..
root@overo:/# ls
bin  boot  dev  etc  home  lib  media  mnt  proc  run  sbin  sys  tmp  usr  var
root@overo:/# ls -la
total 48
drwxr-xr-x 17 root root 4096 May  9 05:41 .
drwxr-xr-x 17 root root 4096 May  9 05:41 ..
drwxr-xr-x  2 root root 4096 May  9  2018 bin
drwxr-xr-x  2 root root 4096 May  9  2018 boot
drwxr-xr-x 11 root root 3620 May  9 04:57 dev
drwxr-xr-x 37 root root 4096 May  9 04:57 etc
drwxr-xr-x  4 root root 4096 Jun  1  2018 home
drwxr-xr-x  9 root root 4096 May  9  2018 lib
drwxr-xr-x  2 root root 4096 May  9 04:45 media
drwxr-xr-x  3 root root 4096 May  9  2018 mnt
dr-xr-xr-x 68 root root    0 Jan  1  1970 proc
drwxr-xr-x 12 root root  360 May  9 04:57 run
drwxr-xr-x  2 root root 4096 May  9  2018 sbin
dr-xr-xr-x 12 root root    0 Jan  1  2000 sys
drwxrwxrwt  8 root root  160 Jan  1  2000 tmp
drwxr-xr-x  9 root root 4096 May  9  2018 usr
drwxr-xr-x  8 root root 4096 May  9 04:57 var
root@overo:/#
```

Figura 3.2: Principais diretórios do sistema.

Dos vários diretórios presentes na figura 3.2 destacam-se os diretórios "/bin", "/boot", "/dev", "/lib" e "/sys".

O diretório "/bin" é aonde ficam armazenados os binários dos comandos essenciais do Linux, como os comandos apresentados anteriormente, logo caso se faça necessário acrescentar ao microprocessador mais algum *software* que se faça necessário ele deve ser adicionado a esta pasta para que possa ser encontrado pelo sistema operacional quando requisitado.

O diretório "/boot" já foi utilizado neste trabalho e é o local aonde devem ser armazenados os *bootloaders* e outros programas que fazem parte da inicialização do sistema.

O diretório "/dev" é o diretório onde ficam armazenados os arquivos de dispositivos do sistema. Arquivo de dispositivo é uma maneira que o sistema Linux utiliza para gerar uma interface de comunicação com *drivers* de dispositivos. Ele será muito utilizado mais para a frente durante a comunicação serial, por exemplo.

O diretório "/lib" é o diretório que contém as bibliotecas essenciais para os binários contidos no diretório "/bin", assim caso seja necessário instalação de um novo *software* provavelmente também precisaremos adicionar alguma biblioteca a este diretório.

Por último, o diretório "/sys" é o diretório que contém informações de dispositivos e *drivers*. Está pasta será muito utilizados caso seja necessário utilizar funções como *general purpose input/output* (GPIO), I2C e *direct memory access* (DMA).

3.2 Compilação Cruzada

A compilação cruzada ocorre quando um dispositivo compila um código fonte para uma plataforma diferente daquela que compilou o código, por exemplo, em nosso caso um computador com Linux Ubuntu irá compilar um código para o computador embarcado rodando

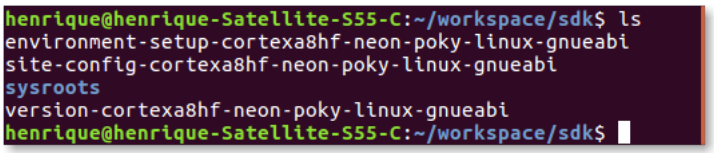

Figura 3.3: Pasta com os arquivos do SDK.

um sistema Linux adaptado.

Utilizar a compilação cruzada para trabalhar com sistemas embarcados é muito comum, principalmente quando estes não possuem capacidade de processamento para suportar um compilador. Para nós seria possível compilar no próprio sistema embarcado, porém, apesar de os recursos não serem tão limitados assim, não queremos desperdiça-los. Além disso programar em um computador regular com a disposição de diversos tipos de IDE diferentes é muito melhor do que programar em um computador embarcado com recursos de interface limitadas.

Para realizarmos esse processo de compilação cruzada precisamos primeiro obter um *software development kit* (SDK). O SDK nada mais é que um conjunto de ferramentas para a compilação dos *softwares*. O projeto Yocto oferece um tutorial para se obter o SDK para seu sistema em sua página do Github[1].

Para obter o SDK para a imagem do sistema operacional que estamos utilizando basta executar o comando "**bitbake <gumstix-console-image> -c populate_sdk**", em que «gumstix-console-image>" é o nome do binário da imagem utilizada no computador embarcado, e será gerado um arquivo "**sdk.sh**". Quando esse arquivo for executado ele irá gerar uma pasta com o conteúdo apresentado na figura 3.3.

O diretório "**sysroot**" contém os arquivos raiz dos dois sistemas, tanto do sistema que irá compilar o código quanto do sistema que irá executar o programa, e o primeiro arquivo da lista importa os endereços e variáveis importantes para a compilação do código.

Para podermos prosseguir com a compilação do código é necessária a execução da seguinte linha de comando "**source sdk/environment-setup-cortexa8hf-neon-poky-Linux-gnueabi**". Uma vez realizado este procedimento basta escrever um código em um editor de texto qualquer, salva-lo como "nome_do_código.c" e executar o comando "**make nome_do_código**". Essa última linha de comando irá criar um arquivo binário executável do seu código.

O comando "make" é na verdade a simplificação de uma extensa linha de comando que chama um compilador "arm-poky-Linux-gnueabi-gcc" e da a ele os parâmetros contidos na pasta SDK. Tudo isso graças ao comando "source" utilizado anteriormente.

Uma vez obtido o executável do código basta copiá-lo para uma das pastas do cartão de memória transferi-lo para o Overo e executá-lo, lembre-se que o diretório principal é o

[1] https://github.com/gumstix/yocto-manifest/wiki/Cross-Compile-with-Yocto-SDK

```
henrique@henrique-Satellite-S55-C:~/workspace/gpio/do0$ make HelloWorld
arm-poky-linux-gnueabi-gcc  -march=armv7-a -mfpu=neon -mfloat-abi=hard -mcpu=corte
x-a8 --sysroot=/home/henrique/workspace/sdk/sysroots/cortexa8hf-neon-poky-linux-gn
ueabi  -O2 -pipe -g -feliminate-unused-debug-types   -Wl,-O1 -Wl,--hash-style=gnu
-Wl,--as-needed  HelloWorld.c   -o HelloWorld
henrique@henrique-Satellite-S55-C:~/workspace/gpio/do0$
```

Figura 3.4: Exemplo de compilação cruzada.

diretório "/home/root/" então se o arquivo for colocado dentro deste diretório será bem fácil encontra-lo.

Na figura 3.4 temos um exemplo da compilação de um programa denominado "HelloWorld".

Depois de inserido o cartão de memória no Overo, podemos inicia-lo normalmente. Quando iniciado, vamos até o diretório em que o programa foi salvo e o executamos com o comando "./nome_do_programa". Se tudo ocorrer bem, o programa deverá ser executado.

3.3 Registradores

Seguindo as etapas das seções anteriores, somos capazes de iniciar o sistema e gerar programas a serem executados pelo sistema operacional. O próximo passo é, portanto, controlar os sinais que podem ser enviados a outros dispositivos pelo computador embarcado para estabelecer a comunicação entre os dispositivos.

A comunicação entre dispositivos é feita pela alteração dos níveis de tensão dos pinos do computador embarcado. Esses pinos estão, de uma maneira resumida, conectados a espaços de memória do sistema e quando alteramos o bit armazenado neste espaço de memória alteramos também o nível de tensão do pino, permitindo a codificação de uma mensagem e sua transmissão a outro dispositivo.

Posteriormente a comunicação entre dispositivos será melhor discutida, mas neste momento o que mais nos importa são os "espaços de memória" citados no parágrafo anterior. Esses espaços de memória são na verdade circuitos digitais voláteis que são capazes de armazenar níveis de tensão, o acesso ao conteúdo desses espaços de memória é extremamente rápido e a estes espaços de memória é dado o nome de registrador.

Sendo assim para que possamos implementar a comunicação entre dois dispositivos, um modem e o computador embarcado, por exemplo, precisamos, primeiro, executar uma tarefa mais simples de alterar os níveis de tensão de um pino. Esse processo de alterar os níveis de tensão de um pino possui diversas aplicações que vão desde o simples controle de *ON/OFF* de um LED até comunicação serial entre dispositivos. Aos pinos com esse propósito é dado o nome de *General Purpose Input/Output* ou GPIO.[2]

Como comentado no capítulo 2 estamos utilizando o computador embarcado Overo junto

[2] `https://en.wikipedia.org/wiki/General-purpose_input/output`

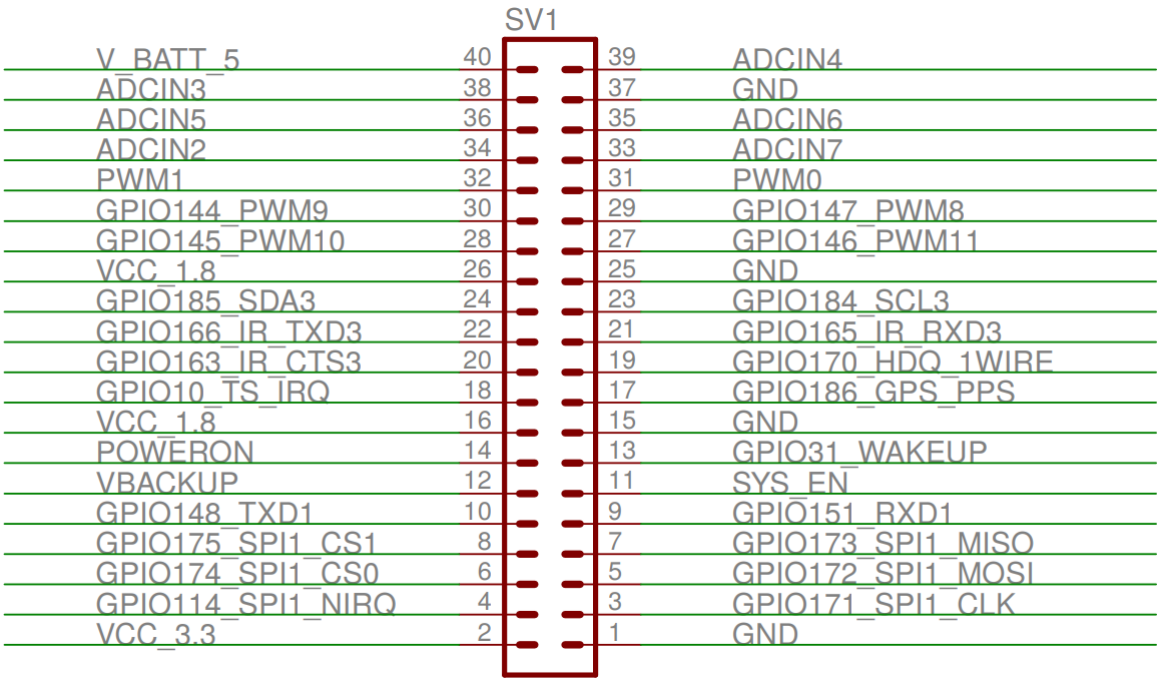

Figura 3.5: Diagrama dos pinos da placa de expansão tobi.

a uma placa de expansão tobi. Uma das funções desta placa é fornecer acesso ao usuário aos pinos do computador embarcado, portanto os pinos do computador embarcado que podemos acessar fisicamente são os pinos da placa de expansão tobi. Na figura 3.5 podemos visualizar um diagrama que contém, de maneira resumida, quais funções ou pinos do computador embarcado estão conectadas a cada pino da placa de expansão tobi. Observe que alguns desses pinos possuem mais de uma função.

3.3.1 Controle do GPIO via terminal

A maneira mais simples, porém menos eficiente, de se controlar o GPIO está descrita no próprio site da fabricante. Lá eles indicam controlar o GPIO pelo próprio terminal do sistema Linux através de um sistema "sysfs". O sistema "sysfs" é um sistema de pseudo arquivos oferecidos pelo núcleo do Linux para o controle e comunicação com dispositivos e *drivers* através do terminal do Linux.

Se, por exemplo, desejarmos controlar a saída do GPIO10 através deste método para piscar um led precisaremos exportar o GPIO10 para o espaço do usuário escrevendo "10" no arquivo "**/sys/class/gpio/export**" isso irá gerar um diretório com outros arquivos para a manipulação do GPIO10, definir sua direção como direção de saída escrevendo "out" em "**/sys/class/gpio/gpio10/direction**" e definir seu valor como alto ou baixo escrevendo 1 ou 0 em "**/sys/class/gpio/gpio10/value**". A função de configuração de interrupção também é acessível pelo terminal.

Isso pode ser feito tanto pelo terminal do usuário com o comando "echo", por exemplo "echo 10 > /sys/class/gpio/export" e também podemos fazer um programa que abra esse ar-

```
root@overo# echo 146 > /sys/class/gpio/export
root@overo:/sys/class/gpio# cat gpio146/direction
in
root@overo# echo out > /sys/class/gpio/gpio146/direction
root@overo:/sys/class/gpio# cat gpio146/direction
out
root@overo# cat /sys/class/gpio/gpio146/value
0
root@overo# echo 1 > /sys/class/gpio/gpio146/value
root@overo# cat /sys/class/gpio/gpio146/value
1
```

Figura 3.6: Exemplo de controle do GPIO146 presente no site da Gumstix.

quivo e escreve nela por nós. Porém como já comentado esse método é bem lento e não pode ser utilizado para comunicação entre dispositivos. Entretanto para atividades com períodos superiores a 100 milissegundos este método pode ser utilizado tranquilamente.

Outra abordagem, utilizando o mesmo método, é utilizar um código semelhante ao código 3.1 que escreve diretamente nos arquivos da figura 3.6. Essa abordagem foi testada e melhorou consideravelmente o tempo de resposta do GPIO de uma maneira bem mais simples do que a maneira apresentada na seção 3.3.2. A seguir será apresentado um código que alterna o nível de tensão do pino o mais rápido o possível.

Listagem 3.1: Teste de velocidade de inversão dos pinos pelo método da escrita em arquivo.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <termios.h>
7
8  int main(){
9          int arq = open("/sys/class/gpio/export",O_WRONLY);
10         write(arq,"10",2);
11         close(arq);
12
13         arq = open("/sys/class/gpio/gpio10/direction",O_WRONLY);
14         write(arq,"out",3);
15         close(arq);
16
17         arq = open("/sys/class/gpio/gpio10/value",O_RDWR);
18         while(1){
19                 write(arq,"1",1);
```

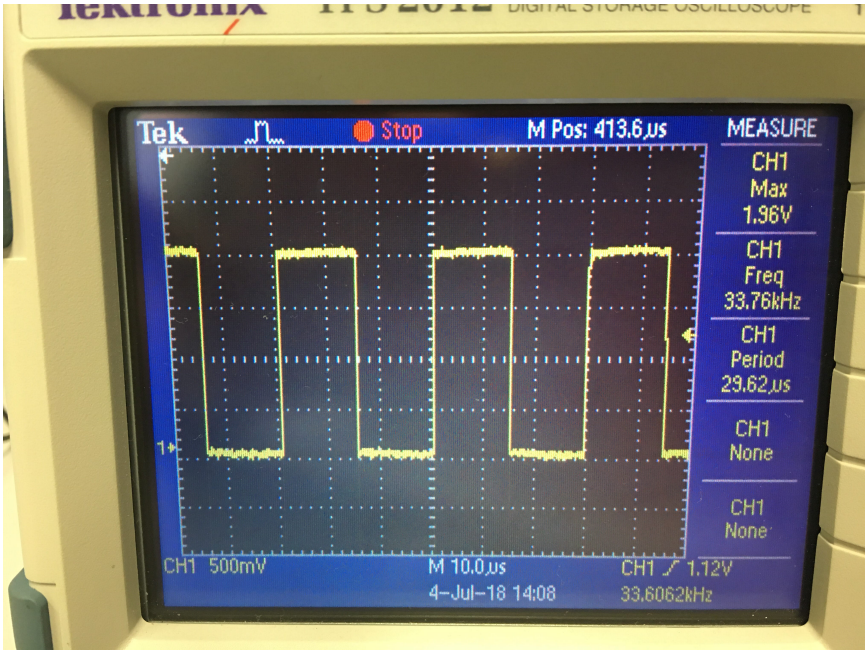

Figura 3.7: Resultado do teste.

```
20                      //usleep(500000);
21                      write(arq,"0",1);
22                      //usleep(500000);
23              }
24              close(arq);
25              return 0;
26      }
```

Para testar o código da listagem 3.1 foi conectado o pino 18 (pino do gpio 10) a um osciloscópio com o objetivo de medir o período da forma de onda. O resultado dessa medida pode ser visto na figura 3.7, nela podemos ver a amplitude da forma de onda de 1,96 V, frequência de 33,76 kHz e período de 29,62 microssegundos. Para a maioria das aplicações podemos utilizar esse método.

3.3.2 Controle do GPIO via registradores

Outra maneira de se controlar o GPIO é escrevendo diretamente nos registradores do sistema. Apesar de o procedimento ser um pouco mais complexo essa, na verdade, é a maneira mais comum e mais recomendada de se realizar esse procedimento oferecendo resultados muito mais rápidos.

Para utilizar este método precisamos, primeiro, definir em quais registradores devemos escrever e o que devemos escrever neles. Essa informação só pode ser encontrada no *Technical Reference Manual* (TRM) do processador DM3730. O TRM [5] pode ser obtida no próprio site da Texas Instruments®.

Como é explicado na seção 25 em [5], a partir da página 3477 a interface de controle

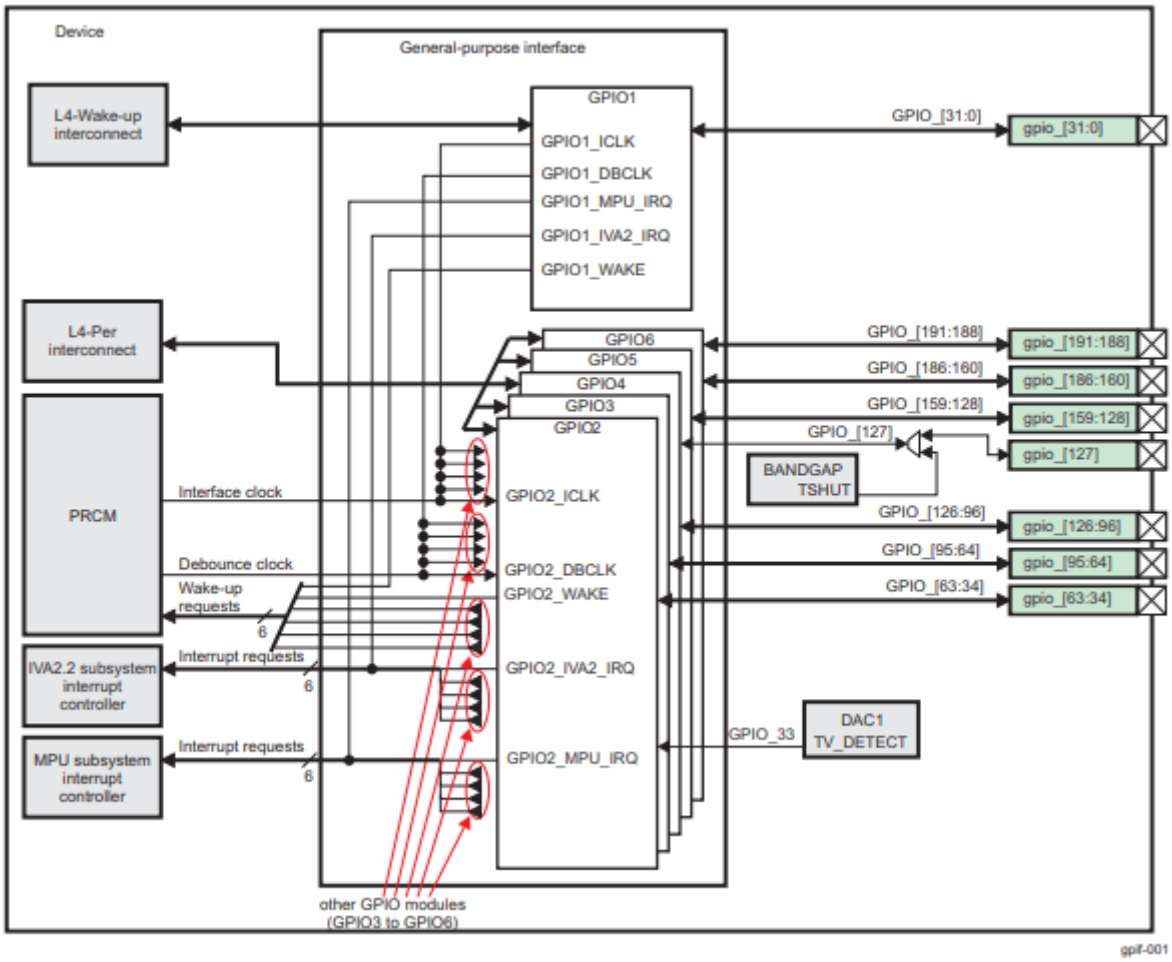

Figura 3.8: Diagrama da interface de GPIO.

combina seis bancos de GPIO. Cada modulo de GPIO providencia 32 pinos totalizando 192 pinos que podem ser utilizados como *input* e/ou *output*. Em nosso caso apenas alguns desses 192 pinos estão fisicamente acessíveis como pode ser visto na figura 3.5.

A figura 3.8 foi retirada de [5] e mostra um pouco mais detalhadamente como esses pinos estão distribuídos entre os módulos dos GPIO.

Cada banco de GPIO possui 26 registradores distribuídos a partir de um endereço de base, cada um desses registradores possui um comprimento de 32 bits ou 4 bytes. Explicação detalhada de cada um desses registradores pode ser encontrada em [5], neste trabalho apenas dois dos registradores serão comentados de forma a ilustrar o funcionamento desses registradores.

O registrador "**GPIO_OE**" é o registrador que define a direção do pino que esta sendo configurado. A abreviação "OE" vem de "*output enable*". Esse registrador possui um *offset* de endereço igual a "0x034", ou seja, seu endereço será o endereço de base do modulo do GPIO mais 34 em hexadecimal. Esse registrador possui 32 bits do tipo "RW", sendo assim se o valor 0 estiver armazenado no pino correspondente à porta GPIO essa porta GPIO estará configurada para operar como *output*, caso neste pino esteja o valor 1 a porta estará configurada como *input*.

O registrador "**GPIO_SETDATAOUT**" é o registrador que tem a função de colocar o bit correspondente no registrador "**GPIO_DATAOUT**" em 1, se tudo estiver configurado corretamente, surgirá no pino físico correspondente o valor de tensão correspondente ao valor 1. Esse registrador possui endereço de *offset* igual a "0x094". Assim como o registrador comentado anteriormente este registrador é constituído por 32 bits do tipo "RW". A leitura de qualquer um dos bits deste registrador retorna o valor do bit correspondente em "GPIO_DATAOUT".

Além dos registradores apresentados na seção 25 de [5] também é necessário configurar um registrador do *system control module* (SCM). Informações sobre o SCM podem ser encontradas na seção 13 do TRM.

O SCM é um módulo que permite o controle através de *software* de várias funções do dispositivo. Para nossa aplicação o SCM é o ponto primário de controle da função de GPIO é nele onde vamos realizar a multiplexação, que determina se o pino irá operar na função de GPIO ou em sua função específica, e definiremos se o GPIO será do tipo *pullup* ou *pulldown*, por exemplo.

Os registradores do SCM são divididos em cinco classes. Entretanto, para nossa aplicação, iremos utilizar apenas uma o bloco de registradores de configuração e multiplexação. Esse bloco é um conjunto de registradores de 32 bits, que configura 2 pinos e define, além dos dois parâmetros mencionados anteriormente, a função de *wakeup*. Aos registradores pertencentes a esse bloco é dado o nome de *Configuration Register Functionality*.

Para encontrarmos qual o endereço de cada registrador deste tipo podemos procurar na tabela 13-4 do TRM. Nessa tabela será dado o endereço físico exato de cada registrador (base+offset). No caso o endereço base é o próprio endereço dos registradores "PADCONFS" da interface do SCM, encontrado na seção 13.6.1 do TRM e o endereço *offset* de cada registrador deste bloco pode ser encontrado na tabela 13-73 do mesmo documento.

Após a identificação dos registradores podemos iniciar a elaboração de um código para modifica-los. Assim nos deparamos com mais um desafio, sistemas operacionais trabalham com dois conceitos de memória, memória física e memória virtual.

Memória física é a memória do hardware, aquela qual sabemos o endereço e pois verificamos no TRM. Entretanto se criarmos um ponteiro que aponta para a memória "0x4800000", por exemplo, ele não irá apontar para a memória física que possui este endereço pois o sistema operacional mapeia um espaço da memória física diferente para cada programa com os principais objetivos de aumentar a segurança e evitar conflitos de dados entre programas.

Entretanto para ter acesso à memória física do sistema precisamos solicitar ao sistema operacional que mapeie esse espaço de memória para a aplicação. Uma maneira de realizar esse procedimento é através da função "**mmap()**". Detalhes do funcionamento dessa função e seus parâmetro podem ser facilmente encontrados na literatura.

Vamos supor que queremos mapear o espaço de memória físico de "0x45000000"

até "0x45001000" e para isso decidimos usar a função "mmap()". Portanto chamamos a função da seguinte maneira, por exemplo, "mmap(NULL,0x1000,PROT_WRITE || PROT_READ,MAP_SHARED,fd,0x45000000)", executando isso a função irá retornar um ponteiro que aponta para um endereço de memória virtual endereçado no endereço de memória física "0x45000000". Em que, para ter acesso à memória física do dispositivo, "fd" é o *file descriptor* direcionado para "/dev/mem".

Com essas informações, temos tudo o que é necessário para implementar testes acerca deste modo de operação. A seguir temos um código que aplica o método descrito nesta seção para alternar o nível de tensão do pino "186". Esse código foi implementado para se realizar o mesmo teste da seção 3.3.1 e o resultado pode ser visto na figura 3.9.

O código da listagem 3.2 foi obtido no fórum de discussões da Gumstix[3] e foram realizadas pequenas alterações para evitar o excesso de informação e facilitar sua compreensão.

Listagem 3.2: Código para realização do teste de velocidade

```
1  #include <stdio.h>  // for lprint instruction
2  #include <fcntl.h>      //ok for mmap param
3  #include <sys/mman.h>   //ok for mmap
4  #include <unistd.h>
5
6  #define SCM_INTERFACE_BASE 0x48002000
7  #define SCM_PADCONFS_BASE 0x48002030
8  #define CONTROL_PADCONF_SYS_NIRQ (*(volatile unsigned long *)0x480021E0)
9  #define CONTROL_PADCONF_SYS_NIRQ_OFFSET 0x1B0
10
11 #define GPIO6_BASE 0x49058000
12 #define GPIO6_SYSCONFIG_OFFSET 0x10
13 #define GPIO6_CLEARDATAOUT_OFFSET 0x90
14 #define GPIO6_SETDATAOUT_OFFSET 0x94
15 % * <henriquepita9@gmail.com> 2018-07-03T21:22:24.877Z:
16 %
17 % ^.
18 #define GPIO6_OE_OFFSET 0x34
19 #define GPIO6_CTRL_OFFSET 0x30
20
21 #define MAP_SIZE (volatile unsigned long) 4*1024
22 #define MAP_MASK (volatile unsigned long)( MAP_SIZE - 1 )
23
24 #define u32 volatile unsigned long
25 u32 *A;
26 u32 *B;
27
28 int main(void){
29 unsigned long i;
30 int fd;
```

[3] http://gumstix.8.x6.nabble.com/Direct-register-access-control-of-GPIO-ARM-interface-o
html

```
31
32 fd = open("/dev/mem", O_RDWR | O_SYNC);
33
34 A = (u32 *) mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
       SCM_INTERFACE_BASE & ~MAP_MASK);
35
36  *(u32 *)((u32)A+0x30+CONTROL_PADCONF_SYS_NIRQ_OFFSET)&=(0xfff8ffff);
37  *(u32 *)((u32)A+0x30+CONTROL_PADCONF_SYS_NIRQ_OFFSET)|=(0x00040000); //
       imposta mode 4 sul registro configurazione pad 186; abilita uso pin
       digitale
38  close(fd);
39 /*********************************************/
40
41 fd = open("/dev/mem", O_RDWR | O_SYNC);
42 B = (volatile unsigned long*) mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE
       , MAP_SHARED, fd, GPIO6_BASE & ~MAP_MASK); // COM1 0x4806A000
43
44 // gpio_186 handling
45  *(u32 *)((u32)B+GPIO6_SYSCONFIG_OFFSET)&= 0xffffffe0;
46  *(u32 *)((u32)B+GPIO6_SYSCONFIG_OFFSET)|= 0x00000014;// bit2=1 enable/
       wake up, free running  clock
47
48    *(u32 *)((u32)B+GPIO6_CTRL_OFFSET)&= 0xfffffff8;  // bit0=0,bit1=0,
         bit2=0 module enabled, clock not gated , clock=interface clock not
         divided
49
50  *(u32 *)((u32)B+GPIO6_OE_OFFSET)&= 0xfbffffff;  // bit26=0, gpio_186
       output
51
52 // generate a pulse stream on gpio_186 pin output
53 for (i=0;i<100000;i++){
54          *( u32 *)((u32)B+(GPIO6_SETDATAOUT_OFFSET))  |= 0x04000000;
55          //printf("Saida = 1\n");
56          //usleep(500000);
57          *( u32 *)((u32)B+(GPIO6_CLEARDATAOUT_OFFSET))|= 0x04000000;
58          //printf("Saida = 0\n");
59          //usleep(500000);
60  }
61 close(fd);
62 return(0);
63 }
```

O código da listagem 3.2 foi testado da mesma maneira que o código da listagem 3.1 apresentado na seção anterior, na figura 3.9 é possível ver o resultado deste teste. Observe que dessa vez o tempo obtido foi 720,3 nanossegundos, ou seja, aproximadamente 42 vezes mais rápido que o resultado apresentado na figura 3.7. Além disso, podemos observar que a forma de onda não é mais um sinal retangular exato, a presença de um efeito capacitivo retardando o processo é evidente, portanto é possível que essa seja a velocidade máxima em

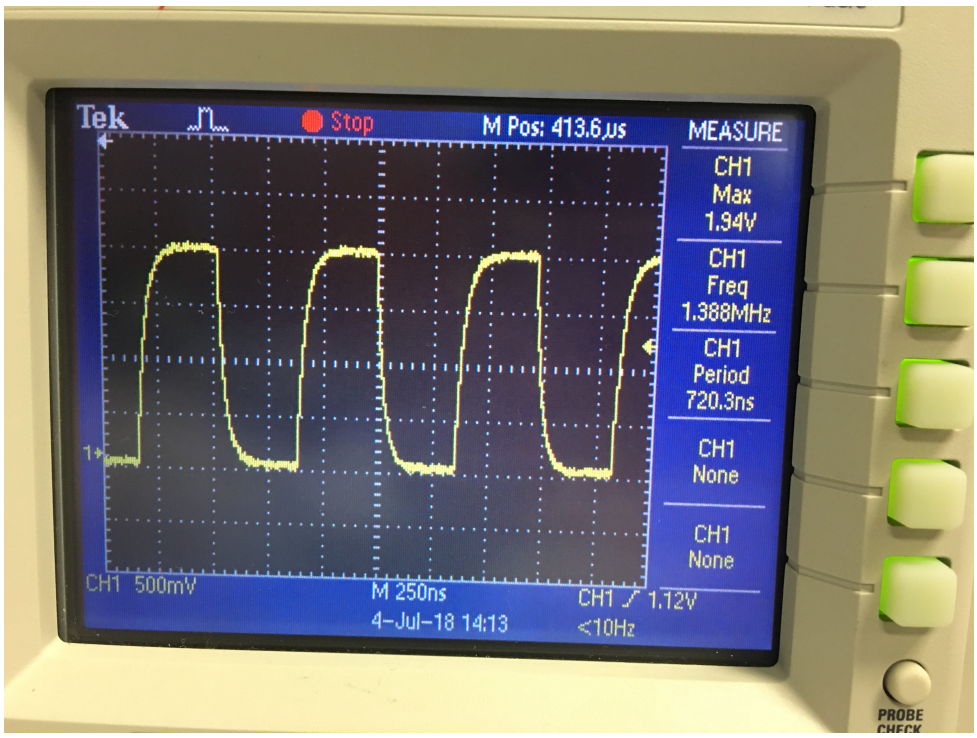

Figura 3.9: Teste de inversão de nível de tensão através dos registradores.

```
root@overo:~# devmem2 0x49058030 w 0x8
/dev/mem opened.
Memory mapped at address 0xb6f3d000.
Read at address  0x49058030 (0xb6f3d030): 0x00000000
Write at address 0x49058030 (0xb6f3d030): 0x00000008, readback 0x00000008
root@overo:~# devmem2 0x49058030
/dev/mem opened.
Memory mapped at address 0xb6fe2000.
Read at address  0x49058030 (0xb6fe2030): 0x00000000
root@overo:~#
```

Figura 3.10: Ilustração do problema em aberto encontrado durante os testes.

que o sinal de um pino pode ser alterado.

Muito dificilmente alguma aplicação envolvendo GPIO não será satisfeita por algum dos métodos aqui apresentados, para finalizar este último tópico é necessário destacar um último problema que ainda não foi resolvido envolvendo escrita em registradores.

O problema atual ocorre sempre que alteramos qualquer valor de qualquer registrador, o que ocorre é que instantes após a alteração do valor do registrador o valor do registrador retorna ao valor que possuía antes de ser alterado. Como o teste desta seção apresentou frequência muito alta ele não foi interrompido por este efeito, porém o fenômeno ocorre inclusive quando alteramos valores dos registradores por comandos do terminal, como o "**devmem2**". Esse problema está exemplificado na figura 3.10.

Na figura executamos o comando "devmem2" para modificar o registrador "0x49058030" que é o registrador que controla o *clock* de todo o bloco do "GPIO6", a modificação deve-

ria realizar uma redução na velocidade do *clock* dividindo-o por 4, logo após a execução do comando é realizado um procedimento de leitura que garante que tudo foi escrito no registrador como o esperado. No entanto o mesmo comando é executado instantes depois no modo de leitura e retorna um valor nulo no registrador. No caso o valor existente no registrador antes da modificação era nulo, porém o registrador sempre retorna ao valor anteriormente armazenado.

Esse problema foi descoberto a pouco tempo e, portanto, se trata de um problema ainda não resolvido. Não se sabe ao certo porque esse comportamento está ocorrendo, porém acredita-se que exista algum processo do sistema operacional que retorne os registradores para os valores iniciais. Esse problema não ocorre, no entanto, para o método da seção 3.3.1, este método opera até que receba uma ordem de parada do usuário. Corrigir este problema demandaria um pouco mais de tempo.

3.4 Comunicação Serial

Nessa seção será discutido mais especificamente o sistema de comunicação serial. No caso do Overo, temos três sistemas de comunicação serial assíncrona universal ou *Universal Asynchronous Receiver/Transmitter* (UART) implementado por hardware à nossa disposição; o que faz desnecessário qualquer implementação manual através de software utilizando GPIO.

Primeiro precisamos entender como funciona o protocolo de comunicação UART. Essa comunicação funciona através da conexão do transmissor (TX) de um dispositivo com o receptor (RX) de outro dispositivo, no caso, apenas o TX realiza alterações no nível de tensão da linha, sendo assim, a comunicação é, para cada conexão, uma via de mão única. Logo, para realizar uma comunicação de mão dupla iremos utilizar duas conexões, uma será a ligação de RX do dispositivo 1 com o TX do dispositivo 2 e a outra ligação será o oposto, RX do dispositivo 2 com TX do dispositivo 1.

Podemos analisar a situação da comunicação a nível de bit. Por exemplo, para uma comunicação UART do tipo "8N1" (8 bits de dados, 0 bits de paridade e 1 bit de parada) teremos o canal em estado *idle*, que significa "não operando", é representado pelo nível de tensão estático em alto, quando pretende-se iniciar a comunicação é enviado um pulso de nível baixo, logo em seguida são enviados os oito bits de dados que será acompanhado de um bits de parada em estado alto.

É importante comentar que a função do bit de parada é realizar uma pausa na transmissão para algum processamento interno dos dispositivos, não é necessário, portanto, nenhum tempo adicional entre os dados transmitidos.

Como essa é uma comunicação assíncrona, é essencial que a velocidade da comunicação seja pré-determinada. Essa velocidade é geralmente dada em *baudrate* e, em nosso caso,

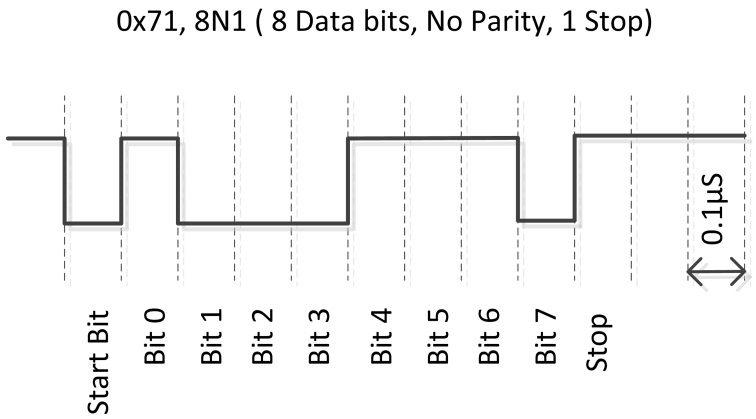

Figura 3.11: Ilustração de funcionamento da comunicação UART.

será essencialmente o tempo de duração de cada bit.

A figura 3.11[4] é um esquemático que ilustra o que foi explicado nos parágrafos anteriores, na ilustração a velocidade de comunicação é de 10.000.000 baud/s.

3.4.1 Particularidades da Gumstix Overo

O computador embarcado Overo conectado a uma placa de expansão tobi possui dois desses UARTs disponíveis, por padrão, para nosso uso nos pinos apresentados na figura 3.5. Como pode ser visto a UART1 está conectada aos pinos 10 e 9 e a UART3 está conectada aos pinos 22 e 21. Ainda é importante dizer que a porta serial UART3 é o mesmo pino utilizado pelo "USB console", ou seja, é o mesmo pino que usamos para controlar a gumstix pelo computador, ou seja, em configuração padrão as mensagens do sistema e as mensagens para o sistema são enviadas por esta porta.

Caso as duas portas seriais já mencionadas não sejam suficientes para a aplicação existe, também, a porta serial UART2 que não está, por padrão, disponível em um dos pinos para nossa utilização. Na verdade ela foi reservada para comunicar-se com o *bluetooth*, contudo apenas versões posteriores ao nosso computador embarcado possuem *bluetooth*, portanto podemos, caso necessário, exportar a UART2 para os pinos e utiliza-los.

Para utilizar esta porta serial é necessário modificar o *u-boot* de modo a multiplexar sua função ao GPIO 146/147 que, como mostrado na figura 3.5, estão ligados aos pinos 29 e 27. Portanto para faze-lo é necessário modificar o arquivo "overo.h" removendo as linhas de comando referentes ao modo de GPIO dos pinos 146 e 147, remover as linhas que desabilitam a UART2 e adicionar as linhas que habilitam a comunicação serial pela UART2.

Para entender detalhadamente o que precisa ser feito e quais registradores serão alterados deve-se consultar a seção de comunicação serial de [5]. Contudo existe um tópico no fórum de discussões da gumstix[5] que indica diretamente quais alterações devem ser feitos no "u-

[4]Retirada de `https://ece353.engr.wisc.edu/serial-interfaces/uart-basics/`
[5]`http://gumstix.8.x6.nabble.com/Using-UART-2-on-an-Overo-td660403.html`

boot" para que se possa utilizar a UART2, apesar disso a solução apresentada nesse fórum não foi testada durante este trabalho.

3.4.2 Configuração da UART

Como já comentado na seção 3.4 o computador embarcado possui um hardware específico para comunicação UART, ou seja, não é necessário realizar uma implementação manual, para utilizar a comunicação UART basta escrever em alguns registradores para enviar a mensagem.

Em nosso caso é, na verdade, ainda mais simples, pois no sistema operacional instalado já vieram configurados *drivers* para a aplicação da comunicação serial. Portanto, não é necessário acessar a memória física do dispositivo. Precisamos apenas escrever no *driver* o que deve ser transmitido.

Os *drivers* de comunicação serial são arquivos do tipo caractere com nome "**ttyOx**", em que "x" é um número exclusivo para cada uma das UARTs. Esses *drivers* estão localizados em "**/dev**" e funcionam como comunicação em terminal.

Por exemplo, o *driver* "ttyO2" é o *driver* de comunicação serial da porta "USB Console" a mesma que conectamos ao computador. Ou seja, ao escrever ou ler dessa porta estaremos escrevendo para o computador conectado à gumstix, escrever nesse *driver* terá o mesmo resultado final de chamar a função "printf()" quando um computador estiver conectado a essa porta com o terminal aberto.

A configuração das portas seriais pode ser feita de duas maneiras, por linhas de comando no terminal Linux ou por um código que altere as configurações do hardware. A mais simples e, novamente, mais limitada ou menos eficiente é a configuração por meio de linhas de comando, a configuração por esse modo costuma ser usada apenas quando feita por um usuário humano em tempo real.

Para realizar a configuração por meio do terminal Linux devemos utilizar o comando "**stty**" esse comando possui uma enorme quantidade de parâmetros que permite estabelecer a comunicação serial da forma desejada, para visualizar todos os parâmetros basta executar "stty –help". Se, por exemplo, for executada a linha de comando "stty -F /dev/ttyO0 -a" serão impressas todas as configurações da comunicação serial UART1 do dispositivo. Para imprimir apenas as principais configurações, deve-se suprimir a última opção do comando. Caso a alteração da velocidade seja desejável, ela pode ser alterada simplesmente acrescentando a velocidade desejada ao final da linha de comando.

A figura 3.12 apresenta um exemplo de configuração da UART1 por meio do terminal de comandos Linux.

A outra maneira de configurar a comunicação serial feita por esses *drivers* sem alterar manualmente o conteúdo do endereço físico da memória, como feito na seção 3.3.2, é com

Figura 3.12: Exemplo de configuração de UART por meio do terminal.

o auxílio da biblioteca "**termios.h**". Essa biblioteca possui uma ampla variedade de funções que configuram a comunicação serial com base nos parâmetros de uma estrutura "termios" também definida nesta biblioteca.

São dois os parâmetros da comunicação UART, além dos mencionados anteriormente, que se destacam, o número mínimo de bits que se espera ler em cada tentativa de leitura e o tempo máximo de espera por um novo caractere após a transmissão do último caractere após o número mínimo de caracteres ser atingido.

O número mínimo de bits que se espera ser lido e o tempo máximo de espera pelo próximo bit em décimos de segundo podem ser configurados com os seguintes comandos "**termios.c_cc[VMIN] =**" e "**termios.c_cc[VTIME] =**", em que "termios" é o nome de sua estrutura. Para a configuração de velocidade recomenda-se usar a função "**cfsetspeed()**". A função "**cfmakeraw()**" configura, além de outros parâmetros, o funcionamento sem bit de paridade e com 8 bits de dados.

Após realizados os ajustes na estrutura é necessário executar a função "cfsetattr()" para que as alterações sejam feitas na UART.

O código da listagem 3.3 é um exemplo de função, que foi usado nos testes do computador embarcado. No teste dois computadores embarcados idênticos que futuramente serão colocados nos aviões são conectados e executam o código da listagem 3.4. O objetivo deste teste é realizar a comunicação serial entre os dois dispositivos e verificar alguma possível falha.

Listagem 3.3: Exemplo de função para configuração de comunicação serial

```
1 #include <termios.h>
2
3 int configUART1(){
4         struct termios cUART1;
5         int UART1 = open("/dev/ttyO0",O_RDWR);
6         if(tcgetattr(UART1,&cUART1))printf("Erro tcgetattr");
7         cfmakeraw(&cUART1);
8         cfsetspeed(&cUART1,B115200);
9         cUART1.c_cflag &= ~CSTOPB;
```

```
10          cUART1.c_cc[VMIN] = 1;
11          cUART1.c_cc[VTIME] = 1;
12          if(tcsetattr(UART1,TCSANOW,&cUART1))printf("Erro tcsetattr");
13          return UART1;
14  }
```

Observe que nessa função de configuração não foi utilizada a flag "O_NONBLOCK" na função "open()" e foi definido como 1 o número mínimo de caracteres a serem retornados após uma tentativa de leitura, portanto caso o código seja executado e nenhuma informação seja enviada para este canal o processador aguardará eternamente por esse caractere. A contagem de tempo, definida como 0,1 segundo, só inicia após o número mínimo de caracteres ser atingido.

Uma vez feita a função de configuração foi implementado também o código da listagem 3.4, onde um dispositivo envia uma mensagem para o outro dispositivo que responde com uma mensagem semelhante para o primeiro dispositivo, em seguida ambos os dispositivos imprimem a mensagem recebida.

Listagem 3.4: Código para teste de comunicação serial

```
1  #include <termios.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(){
6           int UART1 = configUART1();
7
8           char dis[2], out[100],string[100];
9           printf("Que dispositivo eu sou?");
10          scanf("%c",&dis[0]);
11          dis[1]=0;
12      string[0]=0;
13      strcat(string,"Ola! Essa e uma mensagem do dispositivo ");
14      strcat(string,dis);
15
16          //testa UART
17          write(UART1,string,strlen(string));
18      sleep(1);
19      read(UART1,out,100);
20          printf("Mensagem lida pelo dispositivo %s: %s\n",dis,out);
21          close(UART1);
22          return 0;
23  }
```

Como os dois dispositivos são idênticos e, em ambos os casos, foi configurada a UART1 será necessário, conectar o pino 10 de um dispositivo com o pino 9 do outro dispositivo e vice versa. Utilizando esse código como base é possível enviar qualquer mensagem de um dispositivo ao outro.

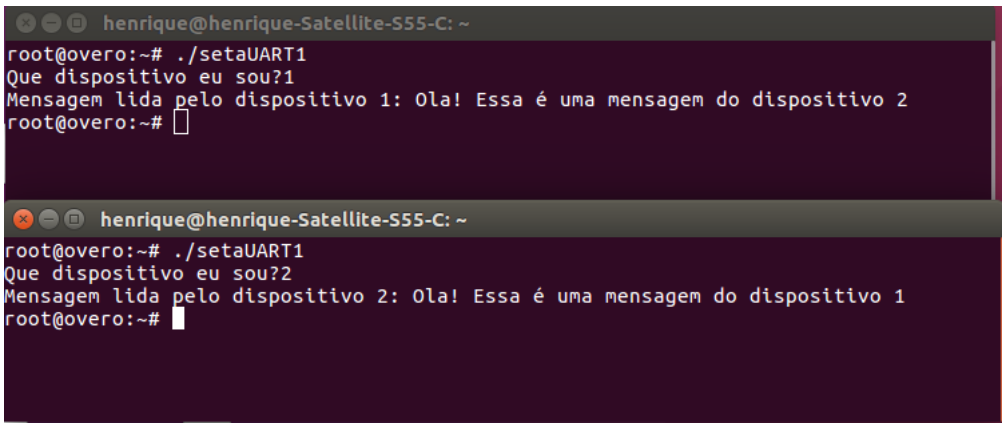

Figura 3.13: Execução do código apresentado.

A figura 3.13 apresenta o resultado do teste dos códigos apresentados. Nessa figura podemos ver dois terminais do Linux, cada um vinculado a um computador embarcado, e ambos chamam a mesma função, logo em seguida vemos a mensagem lida por cada um dos dispositivos.

Capítulo 4

Comunicação via modem

Como explicado no capítulo 1 o computador embarcado será conectado a um modem para comunicar-se com as outras aeronaves e a estação terra. Neste capítulo será analisado mais detalhadamente a comunicação computador embarcado-modem e a comunicação entre os modens.

O modem que será utilizado é o modem P900 da microhard® cujo manual de referência [6] pode ser obtido no próprio site da fabricante[1]. A figura 4.1[2] contém uma foto do modem estudado neste capítulo e utilizado durante o trabalho.

4.1 Configuração do Modem

Para que seja feita a configuração dos modens é necessário que este seja conectado a um computador regular. A comunicação de um computador regular com o modem é feita, da mesma maneira que a comunicação de um computador regular com o computador embarcado, ou seja, por uma comunicação serial. Por padrão a comunicação serial do modem é feita por uma comunicação UART de 9600 baud/s, 8 bits de dados, sem bit de paridade e 1 bit de parada, além dessas configurações da comunicação UART é adotado o padrão físico de conexão serial RS-232.

Logo para conectar um computador regular ao modem é necessário, primeiro, um cabo conversor de USB para RS-232. Assim como na comunicação com o computador embarcado recomenda-se a utilização de um computador linux com o programa "*screen*"para a configuração dos modens.

A configuração do modem só pode ser feita utilizando as configurações padrões do modem, ou seja, sempre que o modo configuração for ativado o modem retorna, temporariamente, para as configurações descritas no parágrafo anterior, após sair do modo de con-

[1] `http://www.microhardcorp.com/`

[2] Imagem original disponível em `https://loja.smartcore.com.br/to60vab07-modem-900mhz-mesh-1w-p900`

Figura 4.1: Modem microhard P900.

figuração o modem volta a operar de acordo com as configurações determinada em seus registradores.

O modo de configuração pode ser iniciado de duas maneiras. Em ambos os casos é necessário, primeiro, conectar o computador regular ao modem, iniciar o terminal do computador regular e executar o programa "*screen*". A primeira maneira de acessar o modo de configuração é pressionar e segurar o botão "CONFIG", energizar o modem e só depois soltar o botão, a outra opção consiste em ligar o modem normalmente e, depois que este estiver no modo de dados (modo de operação regular) enviar a sequência de caracteres "+++" e esperar 1 segundo. Realizado qualquer um dos dois métodos a mensagem "NO CARRIER OK" deve aparecer no monitor indicando a entrada no modo de configuração.

4.1.1 Comandos básicos

Uma vez acessado o modo de configuração, podemos começar a dar comandos ao modem. Todos os comandos do modem possuem um prefixo de dois caracteres "AT", logo depois desses dois caracteres deve ser inserido o comando propriamente dito. Por exemplo, o comando que retorna para o modo de transmissão de dados é o comando "A", portanto para retornar ao modo de dados deve-se digitar "ATA".

O comando "I" retorna informações sobre o modem P900. O comando "login" protege o modo de configuração por uma senha a ser definida pelo usuário. O comando "M" ativa um menu do modo de operação rede, que será explicado mais a frente, útil para identificação de erros e obtenção de *logs*. O comando "&Fn" configura o modem com alguma das configurações padrão de fábrica. O comando "&V" imprime os registradores do modem e seus valores. O comando "Sn=<value>" altera o valor do registrador "n". Por último o comando "&W" escreve as alterações feitas nos registradores do modem, se esse último comando não for executado antes de sair do modo de configuração todas as alterações serão descartadas.

4.1.2 Registradores

O modem apresenta muitos registradores, não irei descrever todos aqui, entretanto quase todas as características do modem podem ser alteradas, porém, para faze-las, recomenda-se consultar o manual [6] para especificações detalhadas.

O caractere para entrada no modo de configuração pode ser desabilitado/alterado no registrador "2". A função do modem dentro do modo de operação, a velocidade da comunicação serial com o dispositivo conectado ao modem e a velocidade da comunicação com os outros modens via propagação de ondas eletromagnéticas podem ser modificadas nos registradores "101", "102" e "103", respectivamente. O formato da comunicação serial com o dispositivo e o número mínimo de bytes a serem transmitidos podem ser alterados nos registrador "110" e "111", respectivamente. O tipo de modo de operação é determinado pelo registrador "133". O modo de comunicação serial é determinado pelo registrador "142". O modo de acesso ao canal de comunicação pode ser alterado no registrador "244".

4.1.3 Modos de operação

O modem possui três modos de operação: modo de rede, modo ponto-a-ponto e modo ponto-a-multiponto.

O modo de rede é um modo de operação onde todos os dispositivos conectados à rede comunicam-se entre si, a mensagem enviada por um modem é recebida simultaneamente por todos os outros modens com a mesma configuração, que estejam dentro da área de cobertura.

O modo de configuração ponto-a-ponto é um modo de operação em que a comunicação é apenas entre um modem "mestre" e um modem "escravo". Podem haver repetidores de sinal entre eles, porém a mensagem enviada por um é recebida apenas pelo seu correspondente.

E, por último, existe o modo ponto-a-multiponto onde um modem mestre se comunica com vários modens escravos, toda a informação originada dos escravos é direcionada ao mestre e, quando necessário ou desejável, cabe ao mestre repassar essa informação ao destino final, nessa topologia toda a informação passa pelo mestre, que controla a rede.

Evidentemente, para a nossa aplicação, a topologia mais interessante é a topologia de rede. Nessa topologia todos os modens receberão a informação simultaneamente, sendo mais rápida que outras topologias, além disso a maior parte da informação gerada em nosso caso tem mesmo o objetivo de ser transmitida a todos os outros dispositivos.

4.1.4 Modos de acesso ao canal

Existem, também, três modos de acesso ao canal, "Aloha", "RTS/CTS" e "TDMA".

O modo "Aloha" é um protocolo de acesso ao meio no qual sempre que um dispositivos

possui dados a serem enviados esse dispositivo aguarda um período aleatório e tenta enviar esse dado. Caso, nessa tentativa, seja recebido pelo dispositivo algum outro sinal é assumido que houve colisão de dados e portanto a transmissão de dados é abortada, aguardam-se, novamente, um período de tempo aleatório até que a mensagem seja novamente enviada. O processo se repete até que o dado tenha sido inteiramente enviado sem que haja colisão.

O modo "RTS/CTS" do inglês *Request to Send / Clear to Send* é um modo que tem o objetivo de diminuir a colisão de transferência de dados, inclusive devido ao problema do terminal escondido. Nesse modo cada modem escravo, quando possui dados para enviar, solicita permissão de envio para o modem mestre por um canal alternativo, o modem mestre verifica se o canal principal está ocupado e responde à solicitação permitindo ou não a transferência de dados. As mensagens de solicitação e liberação são endereçadas para garantir que dois modens distintos não entendam que estão liberados para enviar informações.

Por último o modo "TDMA" do inglês *Time Domain Multiple Access*, nesse modo a cada modem é definido um intervalo de tempo ao qual o modem pode transmitir dados. Após o fim do intervalo de tempo de um modem se inicia o intervalo de tempo do modem seguinte e assim por diante, quando o intervalo de tempo do último modem acabar o processo se reinicia. Uma desvantagem desse modo é a necessidade de esperar um intervalo de tempo de um dispositivo mesmo que ele não possua dados para transmitir.

Dos modos apresentados o modo RTS/CTS é o modo que, aparentemente, vai apresentar melhor resultado pois não é necessário esperar por dispositivos que não tem dados a enviar e apresenta pequenas chances de colisão de dados.

4.2 Modem e Computador Embarcado

A conexão do computador embarcado com o modem é uma configuração de fundamental importância. O computador embarcado possui um pino para transmissão e outro para recepção via UART enquanto o modem apresenta conexões por meio de uma interface elétrica do tipo "RS232" ou "RS485".

Imagine, por exemplo, que o nosso computador embarcado fosse ligado diretamente a outro computador embarcado a 1 km de distância, nesse caso é razoável imaginar que a comunicação não iria funcionar, pois a tensão de 1,8 V e o excesso de ruído na linha de transmissão iriam maquiar os bits enviados. Para resolver problemas como isso surgiram as interfaces de comunicação, que definem apenas as características elétricas do processo de comunicação e não são o protocolos de comunicação.

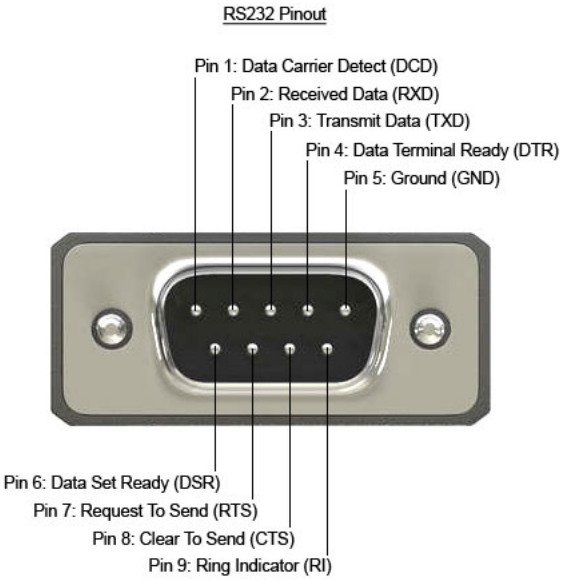

Figura 4.2: Exemplo cabo conector RS-232.

4.2.1 Interface RS-232

A interface de comunicação RS-232 é a configuração padrão do modem é ela é um pouco mais complexa. Essa interface não tem o objetivo de transmitir o sinal a grandes distâncias, seu objetivo é, na verdade, além de diminuir as perdas causadas pelo ruído do sistema, organizar a informação enviada pelos dispositivos de modo que estes sinais possam ser trabalhados individualmente. Essa interface utiliza uma conexão com 9 pinos (ou fios) nomeados como "DCD", "RXD", "TXD", "DTR", "GND", "DSR", "RTS" e "CTS ". O ultimo pino da conexão não tem aplicação na interface RS-232.

Na figura 4.2[3] temos um exemplo de conector de um cabo para o formato RS-232, observe que nessa imagem o pino 9 possui uma finalidade, entretanto, em nosso caso, isso não se aplica.

O "*data carrier detect*" (DTC) é simplesmente um sinal de alto e baixo enviado pelo modem ao microprocessador, esse sinal indica com o sinal "baixo" quando o modem estabeleceu a conexão com outro modem ou uma rede. Esse sinal pode ser configurado para funcionar de uma maneira mais específica dependendo do tipo de conexão realizado, para mais detalhes pode-se consultar [6], página 99, comando "&C".

A conexão "RXD" (*receiver data*) é o próprio canal de comunicação em si ele é a saída do modem que transmite ao dispositivo a mensagem recebida da rede. Em analogia a conexão "TXD" (*Transmiter Data*) é o canal de comunicação de entrada do modem, de onde o modem recebe a mensagem do computador embarcado.

O "DTR" (*data terminal ready*) é um sinal que informa ao modem que o dispositivo conectado está pronto para receber dados, esse sinal é ativado no estado "baixo". Analoga-

[3] A imagem foi obtida em `http://labdegaragem.com`.

mente o "DSR" (*Data Set Ready*) é o sinal que informa ao computador embarcado quando o modem esta pronto para receber dados.

Os canais de "RTS" e "CTS" são canais para implementação de *handshaking*, mais detalhes podem ser encontrados em [6].

O nível de tensão determinado por essa interface é de -15 V a 15 V, onde o nível de tensão de 3 V a 15 V significa o nível lógico "baixo" e o nível de tensão de -3 V a -15 V representa o nível lógico "alto".

Para realizar a conversão da comunicação UART para RS-232 podemos usar um CI "MAX232" da Texas Instruments, por exemplo, esse dispositivo realiza toda a conversão de níveis de tensão e a única modificação que deveria ser feita é um ajuste de tensão "alto" do computador embarcado de 1,8 V para 3,3 V ou 5 V, esse ajuste pode ser feito por um divisor de tensão ou *buffer*. Mais detalhes sobre o MAX232 podem ser obtidos em seu manual disponível no site da fabricante[4].

4.2.2 Interface RS-485

A interface RS-485 é mais simples que a interface RS-232, o objetivo principal dessa interface é a transmissão de dados a longa distância e redução do impacto dos ruídos. Essa interface opera apenas com 5 pinos para a comunicação *full duplex* e 3 pinos para a comunicação *half duplex*. Como a comunicação em apenas um sentido (*half duplex*) não faz sentido em nossa aplicação ela será deixada de lado. Os 5 pinos do RS-485 são "RX+", "RX-", "TX+", "TX-" e "GND".

Como podemos ver essa interface é muito mais simples pois não divide a informação necessária para a comunicação em diversos canais. Nessa interface o alcance dos valores de tensão são de -7 V a 12 V, apesar de valores negativos não serem comuns. E os sinais lógicos são avaliados como a diferença entre o pino positivo e o pino negativo, se essa diferença for positiva, temos o nível lógico "alto" e se ela for negativa temos o nível lógico "baixo".

Por exemplo, se durante um instante da transmissão de um sinal o canal "TX+" apresenta tensão de 6 V e o canal "TX-" apresenta o nível de tensão de 4 V, temos uma diferença de tensão de 2 V, logo o nível lógico será "alto". Invertendo-se os níveis de tensão do exemplo teremos uma diferença de tensão de - 2 V e, consequentemente, o nível lógico será "baixo".

A figura 4.3[5] ilustra o funcionamento da comunicação através da interface RS-485.

A conversão da comunicação UART para a interface RS-485 pode ser realizada por meio de um CI "MAX485" da Maxim Integrated, o *datasheet* deste dispositivo pode ser obtido no site da fabricante[6]. Esse dispositivo converte a comunicação UART para o padrão RS-485, novamente a única alteração adicional que deve ser feita durante a implementação deste

[4] `http://www.ti.com/lit/ds/symlink/max232.pdf`
[5] A imagem foi obtida em `https://www.wikipedia.org/`.
[6] `https://datasheets.maximintegrated.com/en/ds/MAX1487-MAX491.pdf`

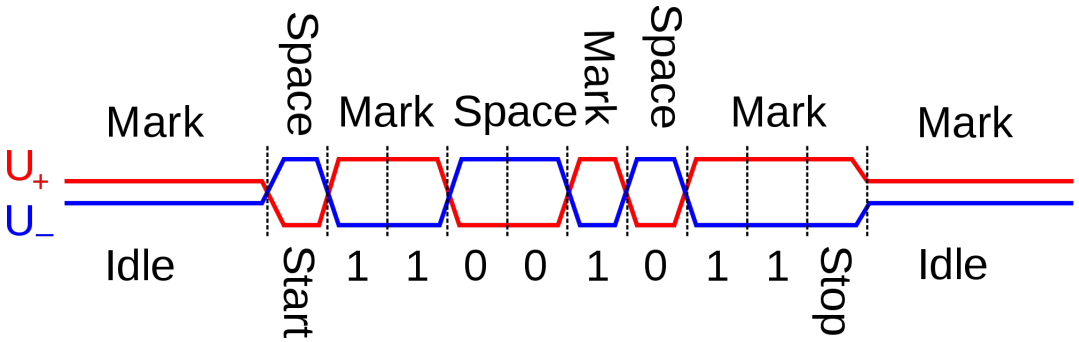

Figura 4.3: Exemplo de funcionamento da interface RS-485.

hardware é o ajuste do nível de tensão originado e destinado ao computador embarcado.

Capítulo 5

Desenvolvimento do Hardware do Módulo Central do VANT

Neste ponto do projeto temos o computador embarcado funcionando bem e o modem corretamente configurado. Portanto para progredirmos precisamos montar um hardware ou uma "estação de trabalho" que alimente e faça a conexão entre todos os dispositivos.

Essa etapa resolve um problema que costuma estar presente quando trabalhamos dispositivos variados e independentes, como vamos fazer a conexão entre a Pixhawk, que opera em 3,3V, com o computador embarcado, que opera em 1,8V, e com o modem, que utiliza uma interface RS-232?

Parte da solução já foi apresentada no final do capítulo 4 com a sugestão de se utilizar os CIs MAX232 ou MAX485. Entretanto existem outros aspectos que também devem ser avaliados como a alimentação destes dispositivos, o posicionamento e a firmeza dos dispositivos no hardware.

Outro ponto importante é que para o funcionamento do Hardware, como inicialmente projetado, é essencial que o protocolo de comunicação das aeronaves, ou seja, entre os modens, seja o mesmo protocolo de comunicação usado pela Pixhawk que é o protocolo Mavlink. O protocolo Mavlink já foi introduzido em [1] e será melhor detalhado no capítulo 6.

5.1 Projeto do Módulo

Para projetarmos o hardware de nosso sistema precisamos, primeiro, determinar quais tarefas o hardware deve ser capaz de realizar e esses serão seus requisitos. Duas dessas tarefas, a alimentação e a conexão dos dispositivos são básicas, já terceira, por sua vez, merece um pouco mais de atenção.

A terceira função a ser realizada pelo hardware seria a de sistema de segurança. Para

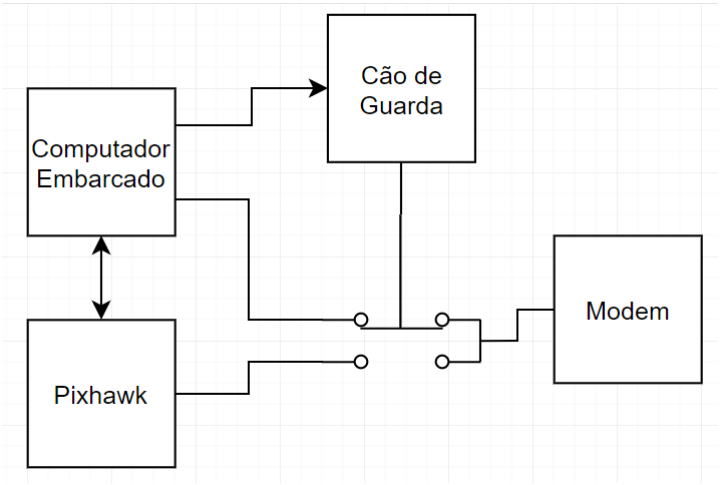

Figura 5.1: Diagrama de blocos da função de cão de guarda do módulo.

evitar falhas ou desconexões do computador embarcado seria interessante que o hardware funcionasse como um cão de guarda que verificasse um pulso periódico vindo do computador embarcado e, caso ocorresse a interrupção deste pulso, o sistema tomasse alguma medida de segurança como conectar diretamente a Pixhawk ao modem de modo que esta possa receber comandos de outro sistema em caso de algum problema com o computador embarcado.

O diagrama de blocos da figura 5.1 ilustra as conexões fundamentais do módulo. Neste diagrama de blocos o computador embarcado e a Pixhawk são conectados ao modem por meio de uma chave e essa chave é controlada pelo cão de guarda que recebe um sinal do computador embarcado. Caso o computador embarcado pare de enviar o sinal periódico ao cão de guarda este altera o estado da chave realizando uma conexão direta entre modem e Pixhawk. Outro ponto importante é a existência de conexão direta entre Pixhawk e computador embarcado que permite a comunicação entre estes dois dispositivos. Por último esta plataforma deve fornecer firmeza ao sistema, é importante que nenhum dos componentes se solte durante o voo e, portanto, todos eles devem ser soldados ou parafusados ao módulo, observe no entanto que essa última característica não está explicita na figura 5.1.

É importante destacar que esta parte nunca chegou a ser implementada devido a problemas com a obtenção dos componentes, contudo, além de ter sido projetado, foi reservado espaço no protótipo do hardware para implementação deste sistema em trabalhos futuros. Ao final deste trabalho foi realizado o projeto de uma PCB que inclui essa terceira função do *hardware*.

Para nosso sistema seriam necessários, portanto, conversores de nível lógico de 1,8V para 5V e de 3,3V para 5V, um CI MAX232 ou equivalente, um multiplexador e um monoestável redisparável. Esses dois últimos seriam os componentes necessários para o cão de guarda.

A bateria que teremos a disposição no VANT é uma bateria LiPo de 14,8V ou seja a bateria não pode alimentar diretamente o computador embarcado, portanto optamos por alimentar todo o hardware pela Pixhawk que possui uma saída de 5V. O computador embarcado pode ser alimentada por 5V no pino 40, como mostrado na figura 3.5. Por último a alimenta-

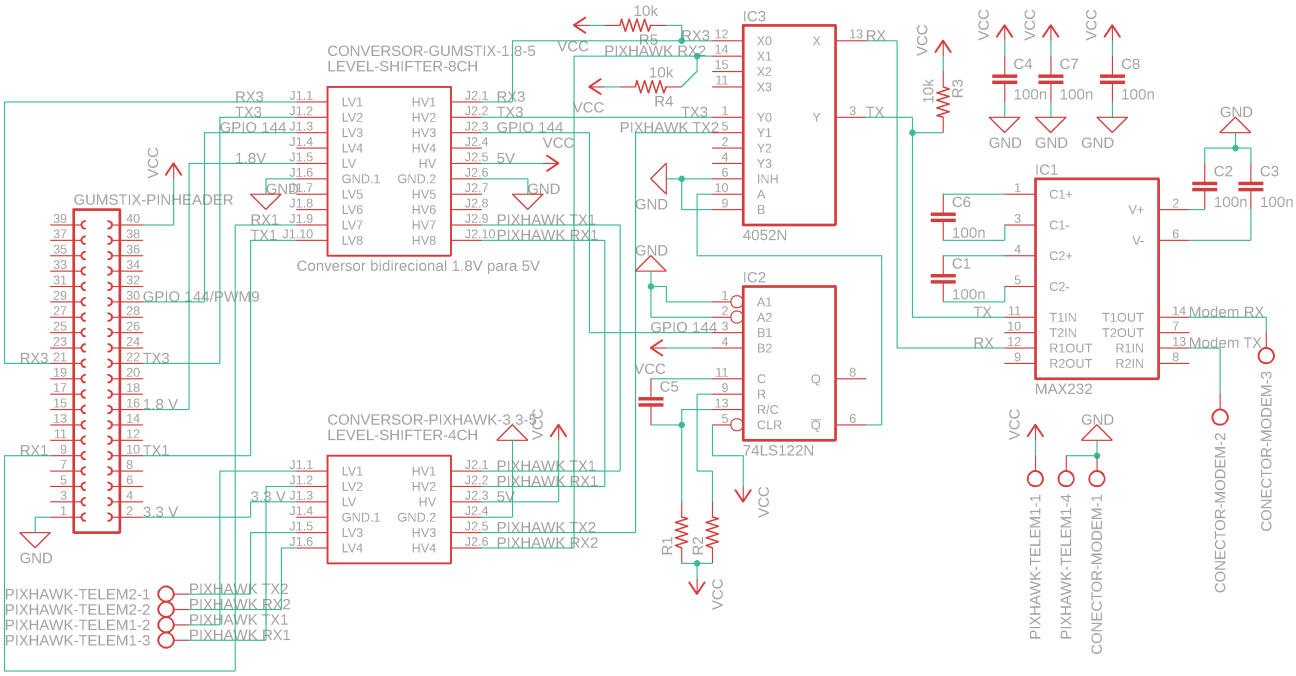

Figura 5.2: Esquemático completo do hardware a ser implementado.

ção do modem pode ser feita de forma direta com a bateria já que a sua tensão de alimentação é de 9V a 30V.

A figura 5.2 contém o esquemático de como seriam feitas as conexões do hardware. Esse esquemático aprofunda o que foi apresentado na figura 5.1. Observe que os valores dos resistores "R1" e "R2" e do capacitor "C5" do esquemático, dependeriam do intervalo de tempo desejado para o monoestável. Já os outros capacitores ou auxiliam o MAX232 a atingir tensões de saída superiores à sua alimentação ou são capacitores para desacoplamento DC que realizam filtragem e estabilização da alimentação, para ambos os casos foram utilizados capacitores de 100 nF. Por último os resistores "R3", "R4" e "R5" são resistores de "pull-up" que tem a função de evitar oscilações nos pinos de RX quando houver comutação nos circuitos do multiplexador.

Por fim temos a figura 5.3 que contém o projeto da placa PCB. O lado inferior, ilustrado em azul, contém um barramento GND e foi coberto com um plano terra para redução da interferência na placa, o lado superior, ilustrado em vermelho, possui o barramento de alimentação e as trilhas de sinais. Além disso foram colocados oito furos na placa, quatro para que o computador embarcado possa ser parafusada a ela e quatro para prender a PCB ao veículo aéreo. A ideia é que o computador embarcado seja posicionada sobre a placa e paralela a esta de modo que se apoie sobre os conectores, de maneira semelhante ao mostrado na figura 5.8, portanto a PCB é ligeiramente maior que a placa de expansão tobi. As informações de dimensões e posições dos dispositivos do computador embarcado foram obtidos no endereço eletrônico da loja disponível no capítulo 2.

Por fim é importante destacar que os dois CIs ao centro da placa na figura 5.3, o 4052N e o 74LS122N, são responsáveis pelo cão de guarda e são o multiplexador e o monoestável

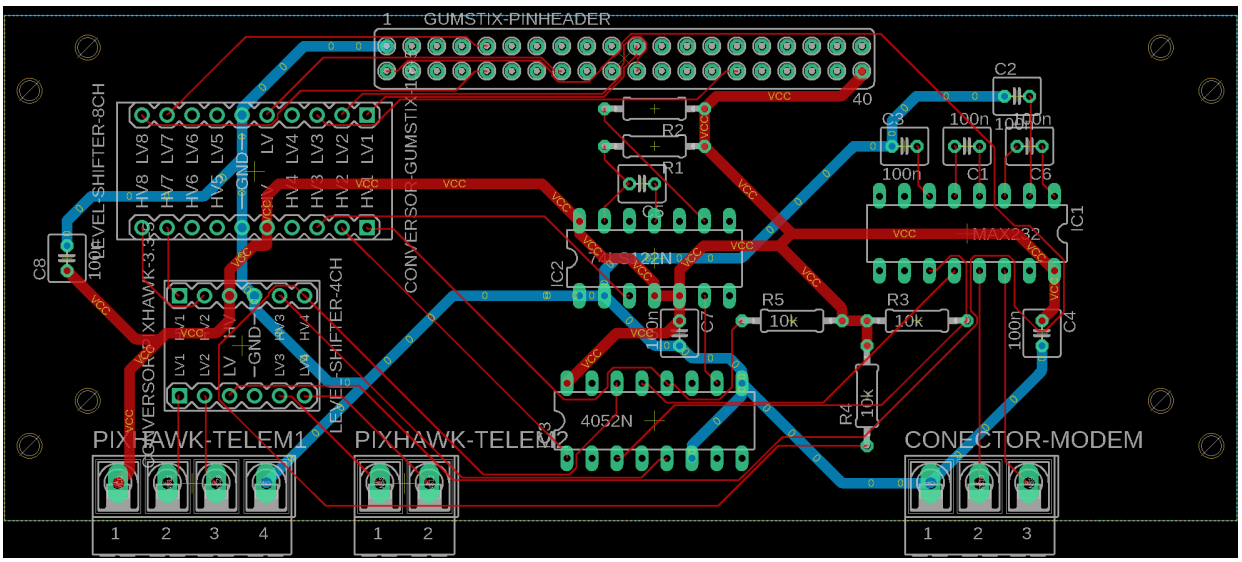

Figura 5.3: Projeto da PCB do hardware.

redisparável, respectivamente. No canto mais a direita próximo à conexão do modem temos o CI MAX232 responsável pela conversão da comunicação UART para RS232. Por último, no canto mais a esquerda, temos dois conversores de níveis lógicos.

5.1.1 Cão de Guarda

O cão de guarda ou *Watchdog* é um temporizador que realiza alguma ação de proteção emergencial, como por exemplo reiniciar um sistema, quando esse tempo se esgota. Assim podemos enviar um pulso que reinicia a contagem do cão de guarda em um período inferior ao do temporizador e, deste modo, a contagem do cão de guarda só será encerrada quando houver algum problema com o dispositivo que envia o pulso.

Em nosso caso idealizamos o pulso como um sinal de PWM ou GPIO enviado pelo computador embarcado periodicamente. Caso esse sinal não seja enviado por algum motivo, como algum problema na alimentação do computador embarcado ou travamento no software o cão de guarda transfere a comunicação com o modem para a Pixhawk.

Como podemos ver na figura 5.2 o cão de guarda seria implementado por um CI 74LS122N[1] e um multiplexador duplo 4052N. Na entrada do 74LS122N teríamos um pulso PWM do computador embarcado e a saída do CI estaria ligada à entrada de controle menos significativa do multiplexador que determina qual dispositivo estaria conectado ao RS-232.

Para entender melhor o funcionamento de um monoestável redispáravel, função realizada pelo 74LS122N, podemos observar a figura 5.4 de um monoestável redisparável implementado com flip-flop R-S. Inicialmente todos os níveis de tensão, com exceção da entrada, são 0, quando surge um pulso de valor 0 na entrada a saída assume valor alto e o circuito RC começa a carregar o capacitor. Quando a tensão no capacitor atinge o valor mínimo para

[1]Datasheet obtido em `https://www.uni-kl.de/elektronik-lager/417682`

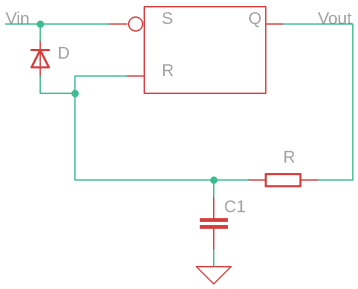

Figura 5.4: Esquemático de monoestável redisparável implementado com flip-flop RS.

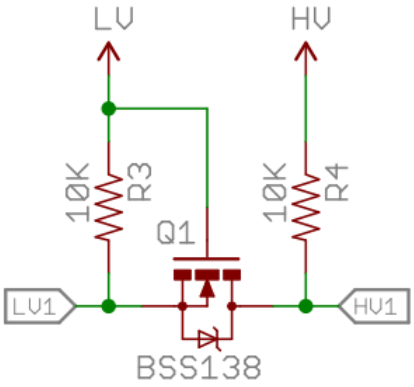

Figura 5.5: Esquemático de um conversor lógico.

ativar o *reset* a saída Q é zerada. Caso antes do capacitor ser carregado o suficiente para que o *reset* seja acionado ocorra outro pulso na entrada o diodo entra na região ativa e o capacitor irá descarregar, zerando a contagem de tempo.

5.1.2 Conversor lógico

Os conversores lógicos são circuitos simples e, ao mesmo tempo, essenciais para o funcionamento de nosso hardware. Uma imagem desse circuito pode ser visto na figura 5.5[2].

Esses circuitos são feitos por um MOSFET, um diodo e dois resistores de pullup para níveis de tensão diferentes. Teremos, portanto, três níveis de tensão em operação, alto, por exemplo 5V, baixo, por exemplo 3,3V, e nulo, aproximadamente 0V. Observamos que o lado de baixa tensão do dispositivo está ligado à fonte do MOSFET, o lado de alta tensão ao dreno e à comporta é ligado constantemente no nível de baixa tensão. Para explicar o funcionamento desse circuito podemos dividi-lo em três situações.

A primeira situação ocorre quando nenhum dispositivo anula o nível de tensão nas entradas/saídas. Nesse caso a diferença de tensão entre a fonte e a comporta é nula e portanto o MOSFET esta em região de corte e não conduz, consequentemente cada um dos lados terá o nível de tensão fornecido pelo resistor de pullup.

[2]Figura obtida em `https://static.sparkfun.com/datasheets/BreakoutBoards/Logic_Level_Bidirectional.pdf`

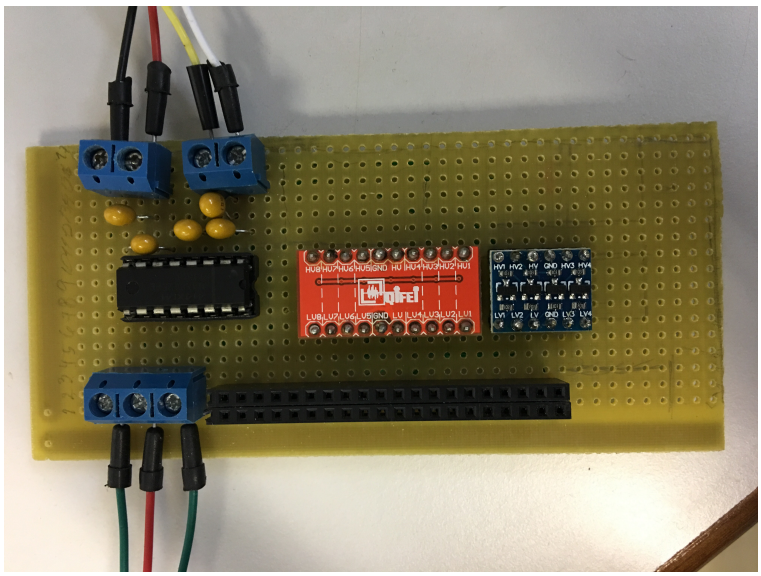

Figura 5.6: Foto do Hardware montado no laboratório.

A segunda situação ocorre quando o lado de baixa anula o seu nível de tensão, nesse caso existirá uma diferença de tensão entre a comporta e a fonte e o MOSFET irá conduzir corrente anulando o nível de tensão no lado de alta.

A terceira situação ocorre quando o lado de alta anula seu nível de tensão, nesse caso o diodo passa a conduzir, o que reduz o nível de tensão no lado de baixa que, por sua vez, faz com que o MOSFET também conduza corrente e, consequentemente, ambos os lados estarão com nível nulo de tensão.

Portanto para que o circuito opere corretamente são necessários duas características: Primeiro, que o lado de baixa tensão tenha nível lógico de tensão alto inferior ao nível lógico alto do lado de alta tensão; Segundo o nível lógico alto do lado de baixa tensão deve ser suficiente para que o MOSFET conduza corrente, ou seja, ele deve ser maior que a tensão de *threshold* do MOSFET.

5.2 Protótipo do módulo

O Hardware que foi feito no laboratório foi uma versão simplificada do hardware projetado na seção 5.1, na realidade não conseguimos obter um CI como o 74LS122N e precisávamos completar essa etapa com urgência para progredir com o projeto e, portanto, optamos por não implementar o cão de guarda, contudo o espaço e a posição dos dispositivos na placa foi planejado de modo que houvesse espaço para uma implementação futura do cão de guarda.

A versão simplificada do hardware envolve a ligação direta dos pinos UART3 do computador embarcado ao MAX232 logo após a conversão do nível lógico de tensão e, além disso, não são necessárias duas comunicações seriais da Pixhawk, apenas uma para comunicar-se

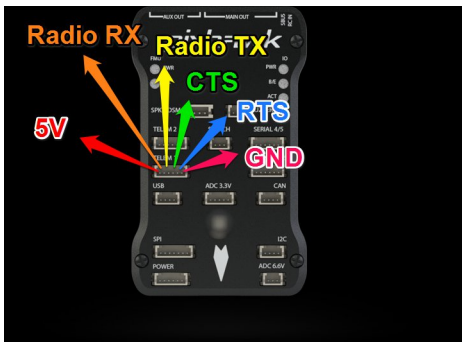

Figura 5.7: Ilustração dos pinos da saída de telemetria da Pixhawk.

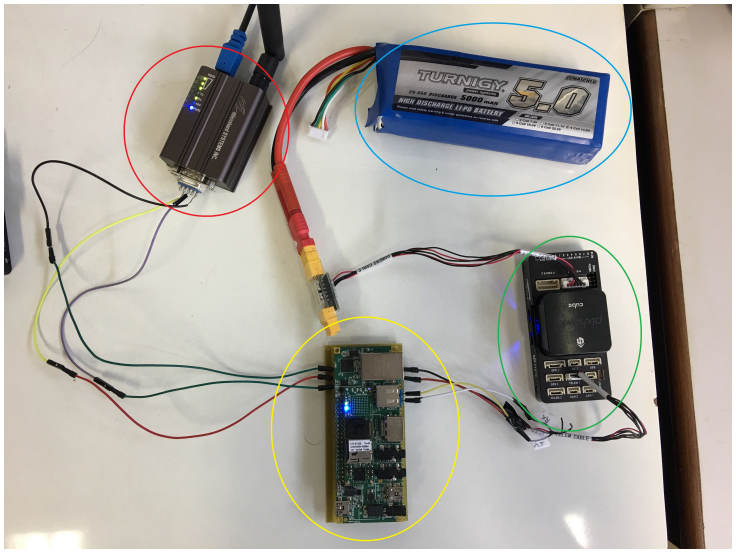

Figura 5.8: Foto do hardware montado e em operação.

com o computador embarcado é suficiente.

A figura 5.6 contém uma foto do Hardware pronto, sem o computador embarcado, observe que as entradas da placa são parafusadas para evitar que a Pixhawk ou o modem se soltem, além disso o computador embarcado, quando colocada, deita-se sobre a placa e também é presa a ela por parafusos assim evita-se que ela se solte devido a vibrações em voo, na imagem o computador embarcado não foi parafusada por se tratar de um protótipo que não iria voar. Todas as conexões foram soldadas no verso da placa.

Para conectar a Pixhawk ao Hardware basta utilizar os pinos de telemetria da Pixhawk, os pinos de telemetria estão mostrados na figura 5.7[3], observe que em nosso caso apenas as conexões de TX, RX, VCC e GND são necessárias, os pinos de RTS e CTS podem ser isolados. No laboratório estamos utilizando a Pixhawk 2, enquanto na figura temos a Pixhawk 1, entretanto a imagem ainda é válida porque não houve alteração da pinagem de telemetria de uma versão para a outra.

Por fim a figura 5.8 mostra o sistema completo em operação. Na imagem o componente

[3]Imagen obtida em https://discuss.ardupilot.org/t/what-is-the-pin-layout-for-pixhawk-2-1-telemetry-port/23128/4

circulado em vermelho é o modem, em azul é a bateria, em verde a Pixhawk e em amarelo o computador embarcado junto a sua placa de expansão tobi preso sobre o protótipo do módulo. A conexão da Pixhawk e do modem foram parafusadas ao módulo. Foram feitas 2 placas iguais a esta para possibilitar a realização de testes simulando dois aviões.

Capítulo 6

Desenvolvimento do Software do Módulo Central do VANT

Uma vez montado o protótipo do hardware do sistema podemos progredir para o desenvolvimento do software que irá operar na base do sistema da aeronave. Observe que o objetivo dessa etapa é desenvolver um software que irá realizar a tarefa de comunicação entre os dispositivos e o processamento de algumas mensagens simples, entretanto ele não estará voltado à implementação da lei de controle.

O Software terá que cuidar de todos os aspectos da comunicação entre o computador embarcado e o modem (entre as outras aeronaves e a estação terra) e entre o computador embarcado e a Pixhawk, como explicado no capítulo 5 o protocolo de comunicação das aeronaves terá de ser o mesmo protocolo de comunicação utilizado pela Pixhawk e, portanto, será utilizado o protocolo MAVLink (Micro Air Vehicle Link).

A etapa de desenvolvimento do software é uma etapa extensa, portanto não foi possível finaliza-la apenas neste trabalho. Em reuniões no laboratório foi decidido que essa etapa seria feita aos poucos conforme a necessidade de implementação de comandos para solução de futuros problemas. Ao fim do desenvolvimento do capítulo foi realizado também um teste para medir o atraso do sistema.

6.1 Protocolo MAVLink

O protocolo de comunicação MAVLink foi um protocolo desenvolvido para comunicação com pequenos veículos não tripulados, é um protocolo muito leve e amplamente difundido. Informações completas e detalhadas sobre o protocolo MAVLink podem ser encontradas em seu endereço eletrônico[1].

Existem, na verdade, duas versões do protocolo, MAVLink 1.0 e MAVLink 2.0. Existem

[1] `https://mavlink.io/en/`

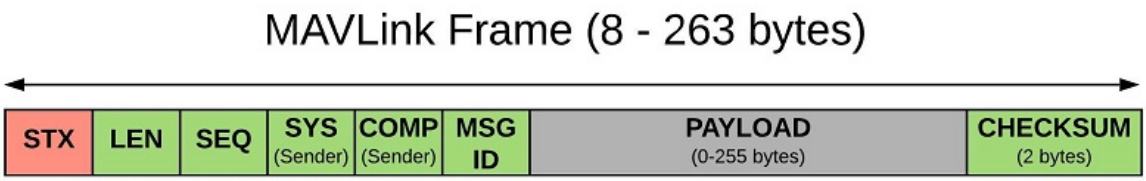

Figura 6.1: Ilustração do *frame* do MAVLink versão 1.

poucas diferenças entre as duas versões e seu funcionamento é basicamente o mesmo. Em ambos os casos o quadro da mensagem ou *frame* possui tamanho variável e pode ter um destino específico dependendo de qual tipo de mensagem está sendo enviada. As próximas subseções explicaram melhor o funcionamento destes protocolos.

6.1.1 Cabeçalho MAVLink

A mensagem ou *frame* da primeira versão do protocolo MAVLink possui tamanho variável de 8 a 263 bytes e é dividida, como mostra a figura 6.1[2], em 8 segmentos. Com exceção do "*PAYLOAD*" e do "*CHECKSUM*" todos os seguimentos estão restritos a 1 byte.

O primeiro seguimento "STX" do inglês *start of text* serve apenas para indicar o início da mensagem MAVLink, assim um dispositivo pode separar dados quaisquer de uma mensagem MAVLink em um canal compartilhado. Essa mensagem, portanto, deve ser fixa e comum ao protocolo seu valor é "FE" em hexadecimal (0xFE).

O segundo seguimento "len" é uma abreviação de "*length*" e contém o tamanho em bytes do seguimento "PAYLOAD", portanto pode assumir valores entre 0 e 255 e, para essa versão do protocolo MAVLink, está amarrado ao tipo de mensagem que esta sendo enviado, ou seja cada mensagem tem um tamanho de "PAYLOAD" específico e este deve ser indicado no segmento "len". Nessa versão do protocolo essa é, então, uma mensagem redundante uma vez que já existe um segmento com o identificador da mensagem.

O terceiro segmento "seq" é utilizado como um contador para detectar a perda de pacotes. Assim a cada mensagem enviada por um dispositivo incrementa-se o valor deste segmento e caso o receptor perceba que houve um salto ele saberá que uma mensagem foi perdida.

O quarto segmento "sys" é referente ao identificador do sistema (*System ID*) do dispositivo que enviou a mensagem. Esse é um identificador que está associado a um determinado veículo ou estação terra, portanto todos os dispositivos em cada aeronave terão o mesmo *system ID*. Os sistemas podem assumir qualquer valor entre 1 e 255, observer que geralmente adota-se 255 para a estação terra e 0 para mensagens broadcast, portanto nenhum sistema deve assumir este *system ID*.

O quinto segmento "comp" é referente ao identificador do componente, ou dispositivo, do sistema (*Component ID*) que enviou a mensagem. Esse identificador serve para diferenciar

[2]Imagem obtida em `https://mavlink.io/en/guide/serialization.html`

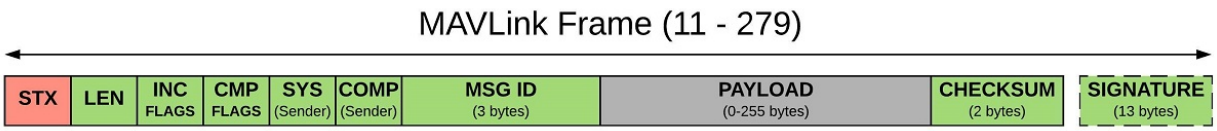

Figura 6.2: Ilustração do *frame* do MAVLink versão 2.

componentes dentro de um mesmo sistema. Esse segmento pode assumir valores entre 0 e 255. É importante comentar que existem padrões para esses identificadores, portanto é recomendado verificar a lista[3] antes de determinar o identificador de um componente do sistema, no entanto lembrar que "0" é reservado para broadcast, "1" para o piloto automático (Pixhawk) e não há nenhuma outra reserva antes do identificador "100" é suficiente para a maioria das aplicações. Por último apesar do *component ID* "0" ser utilizado para broadcast não há problemas em utilizar esse identificador em um dispositivo quando este for o único dispositivo em um sistema, como geralmente acontece no caso de uma estação terra.

O sexto segmento "MSG ID" é o número identificador da mensagem. Cada mensagem possui um número identificador próprio para que o dispositivo de destino seja capaz de identificar a mensagem, interpretar a mensagem contida no "PAYLOAD" e executar a tarefa que foi requisitada.

O sétimo seguimento "PAYLOAD" é a informação da mensagem, seu conteúdo. O tamanho desse seguimento pode variar de 0 a 255 bytes e irá conter outras subdivisões, como *system ID* e *component ID* de destino e valores de parâmetros.

Por último temos o seguimento "*checksum*" utilizado para garantir a integridade da mensagem ao longo do canal. Esse *checksum* é o mesmo utilizado no padrão "ITU X.25", sendo que o processo de obtenção de seu valor inclui toda a mensagem com exceção do primeiro seguimento.

O *frame* da segunda versão do protocolo MAVLink, ilustrado na figura 6.2[4], possui a mesma essência da primeira versão, portanto comentarei apenas os aspectos nos quais os dois se diferenciam. Como podemos ver, desta vez, teremos um total de 10 seguimentos com um tamanho total variável de 11 a 279 bytes. É importante destacar, também, que a segunda versão do MAVLink é compatível com a primeira, portanto um dispositivo que utiliza a segunda versão do protocolo MAVLink consegue se comunicar com outro dispositivo utilizando uma versão anterior, é necessário, no entanto, um *handshaking* no início da comunicação para determinar qual deve ser o protocolo utilizado.

A única diferença no primeiro segmento dessa versão é que o valor do STX é "FD" em hexadecimal (0xFD). Essa é uma possível maneira para o dispositivo identificar qual a versão do MAVLink está sendo utilizada.

O segundo seguimento contém exatamente a mesma informação da versão anterior, en-

[3]Lista de padrões de identificador de componente podem ser encontrados em `https://mavlink.io/en/messages/common.html`

[4]Imagem obtida em `https://mavlink.io/en/guide/serialization.html`

tretanto este seguimento deixou de ser redundante. A diferença ocorre por que os parâmetros finais do "PAYLOAD" não foram utilizados e, consequentemente, possuem valor nulo deixaram de ser transmitidos. Na versão anterior esses bytes eram preenchidos com zeros.

O terceiro seguimento "INC" são *flags* de incompatibilidade. Esse segmento inclui flags que são essenciais para a compreensão da mensagem e, portanto, caso o receptor não compreenda alguma dessas *flags* deve descartar a mensagem. Atualmente a única *flag* incorporada é a *flags* de assinatura do pacote.

O quarto seguimento "CMP" são as *flags* de compatibilidade. Essas *flags* não impedem que a mensagem seja processada com sucesso caso elas não sejam compreendidas e, portanto, não justificam o descarte da mensagem. Um exemplo de *flag* de compatibilidade seria uma *flag* de alta prioridade da mensagem, mesmo que um dispositivo não saiba o que a *flag* significa, e portanto não a priorize, espera-se que o dispositivo processe a mensagem.

O sétimo seguimento "MSG ID" é equivalente ao seu respectivo na primeira versão MAVLink, no entanto são reservados até 3 bytes para a identificação da mensagem, ao invés de apenas 1 como na versão anterior. Atualmente todas as mensagens restritas a segunda versão do protocolo MAVLink possuem identificadores de mensagem superiores a 255.

A única diferença quanto ao seguimento de "PAYLOAD" nessa versão é que seu tamanho agora é variável para um mesmo identificador de mensagem, pois removem-se os zeros que seriam enviados ao final do *payload* da mensagem, como explicado anteriormente.

Por último foi acrescentado um seguimento de assinatura, esse seguimento não é incorporado ao *checksum*, assim como o primeiro seguimento, e consiste em um hash, seu objetivo é conceder autenticidade ao protocolo. Esse seguimento é o grande diferencial entre as duas versões do protocolo e sua existência implica na necessidade de um *handshake* para troca de chaves uma vez que a hash está sendo utilizada para autenticidade. É importante mencionar que este segmento é opcional e implica na *flag* "0x01" nas *flags* de incompatibilidade.

6.1.2 Menssagens MAVLink

O protocolo MAVLink é um protocolo livre que abre espaço para a implementação de novas mensagens, portanto é natural que existam variações do protocolo, com diferentes mensagens, para cada sistema que o implementa. A essas variações da biblioteca de mensagem é dado o nome de "dialetos". Existe, no entanto, um dialeto que serve como base para a maioria dos outros dialetos e, além disso, está implementado na maioria dos sistemas disponíveis no mercado esse é o dialeto "commom".

A quantidade de mensagens MAVLink é grande e, consequentemente, é inviável comentar todas as mensagens nesse trabalho, portanto explicarei apenas algumas que foram usadas nas seções seguintes ou provavelmente serão usadas em trabalhos futuros. Todas essas mensagens aqui comentadas serão do dialeto comum e da primeira versão do protocolo

MAVLink.[5]

- **HEARTBEAT**: Essa mensagem é utilizada para que um dispositivo saiba que está conectado a outro dispositivo. É uma mensagem de *broadcast*, ou seja, não possui endereço de destino. A frequência a qual esse sinal deve ser enviado varia de sistema para sistema e normalmente quando não se recebem 5 sinais de *heartbeat* seguidos considera-se que o dispositivo foi desconectado;

- **SYS_STATUS**: Mensagem que tem o objetivo de transmitir informações sobre as condições em que o sistema está operando, envia informações como nível, tensão e corrente da bateria, erros, taxa de perda de pacotes de comunicação, situação dos sensores e modo de operação. Essa mensagem também não é endereçada, ou seja, é enviada para todos os dispositivos que estejam escutando determinado canal;

- **PING**: Mensagem para medição de atraso no sistema de comunicação. Quando não endereçada, essa mensagem, funciona como requisição e está solicitando que todos os dispositivos que a receberem retornem-a endereçada ao dispositivo que a enviou. A mensagem contém informação do horário de envio, portanto o dispositivo que a receber pode estimar o atraso do canal;

- **CHANGE_OPERATOR_CONTROL**: Essa mensagem costuma ser enviada pela estação terra para que ela possa enviar comandos ao sistema. Em nosso caso ela deverá ser usada para que o computador embarcado possa enviar comandos à Pixhawk. Essa mensagem exige uma mensagem de *acknowledge* como resposta;

- **PARAM_REQUEST_READ**: Mensagem de requisição de envio de informação de algum parâmetro do dispositivo. Após o envio dessa mensagem o dispositivo ao qual a mensagem era endereçada responde a essa mensagem com informações a respeito do parâmetro solicitado;

- **PARAM_REQUEST_LIST**: Semelhante a mensagem anterior, entretanto, neste caso, é enviada uma resposta para cada parâmetro que existir no dispositivo de destino da mensagem de requisição. Em outras palavras, o dispositivo de destino responde com as informações de todos os seus parâmetros;

- **PARAM_SET**: Mensagem para alteração do valor de algum parâmetro do dispositivo;

- **MANUAL_SETPOINT**: Essa mensagem é utilizada para definir, manualmente, aspectos como potência dos motores, *roll*, *pitch* e *yall* da aeronave. É por meio deste comando que o computador embarcado irá controlar a aeronave. Observe que antes de enviar este comando a mensagem "CHANGE_OPERATOR_CONTROL" deve ter sido aceita pelo receptor;

[5]Uma lista completa e detalhada das mensagens MAVLink podem ser encontradas em `https://mavlink.io/en/messages/common.html`

- **ENCAPSULATED_DATA**: Por último essa é uma mensagem para transmissão de dados, geralmente utilizada para troca de dados de fotos pela câmera. Observe que antes do envio desta mensagem são necessárias outras mensagens de *handshake*.

É importante comentar que, como já deve ter sido percebido, cada mensagem possui uma espécie de sub-protocolo diferente. Algumas exigem respostas, outras apenas uma mensagem de *acknowledge* e outras não exigem resposta alguma. Portanto antes de implementar uma nova mensagem deve-se sempre observar essas peculiaridades, pois um dos problemas enfrentados durante esse trabalho ocorreu devido a inobservância da mensagem de confirmação da mensagem de "change_operator_control", por exemplo.

6.2 Biblioteca MAVLink

Existe uma biblioteca oficial do protocolo MAVLink muito útil que define estruturas, funções e constantes para a implementação deste protocolo. No entanto é uma biblioteca apenas para codificação e decodificação dos cabeçalhos e das mensagens, nada mais. Ou seja as tarefas realizadas pelo dispositivo e as respostas a serem enviadas após cada mensagem recebida devem ser implementadas. Consequentemente o volume de trabalho para que se tenha um sistema completo com a maioria das mensagens implementadas é extremamente grande. Decidiu-se, então, que a maioria das funções seria implementada conforme a necessidade. Nesse capítulo irei introduzir a biblioteca e instruir o fundamental de como está funciona para que, em trabalhos futuros, seja possível continuar essa tarefa.

6.2.1 Arquivos

A biblioteca da primeira versão do protocolo MAVLink é composta por 1 arquivo ".h" para cada mensagem do protocolo e mais 5 arquivos ".h" para manipulações em geral, não relacionados a uma mensagem em específica. São eles "checksum.h", que contém funções para manipulação da verificação de integridade da mensagem, "mavlink_conversion.h", uma biblioteca para conversão de unidades utilizadas no protocolo MAVLink para unidades universais, "mavlink_helpers.h", que contém funções para processamento de mensagem recebida, "mavlink_types.h", que define várias estruturas utilizadas pela biblioteca e que serão utilizadas pelo usuário, e "protocol.h", que contém algumas funções utilizadas para manipulação de dados por outras bibliotecas.

A a biblioteca da segunda versão do protocolo MAVLink possui os 5 arquivos da biblioteca da primeira versão, com pequenas modificações para adapta-los ao novo protocolo, e mais dois arquivos "mavlink_get_info.h", para obtenção das informações da mensagem a partir do nome ou do ID da mensagem, e "mavlink_sha256.h", com funções para a utilizar a *hash* na assinatura do protocolo.

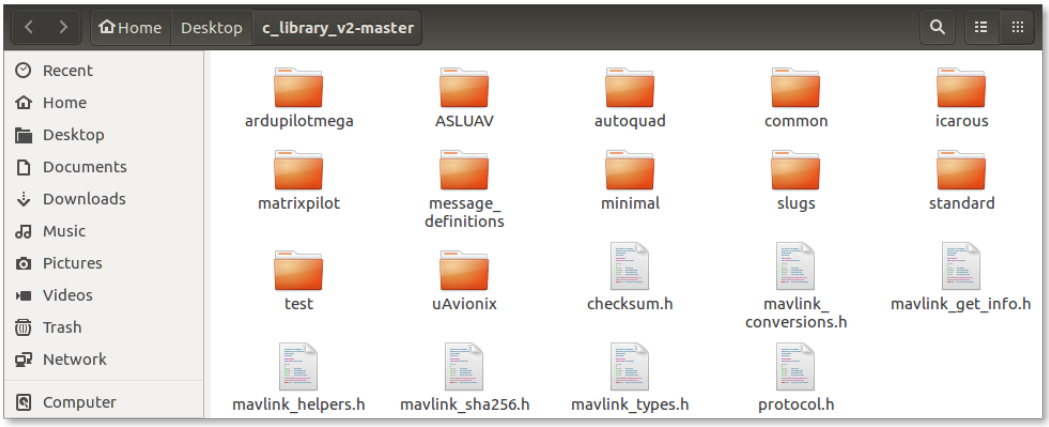

Figura 6.3: Biblioteca do MAVLink versão 2.

Os outros arquivos são específicos de cada dialeto. Para o dialeto comum temos, além dos arquivos específicos de cada função, temos dois arquivos "commom.h", que define várias constantes e variáveis utilizadas pelas funções e inclui todos os arquivos ".h" necessários para utilizar a biblioteca, e "mavlink.h", que apenas define alguns parâmetros e inclui o "commom.h".

A figura 6.3 mostra uma pasta contendo a biblioteca da segunda versão do protocolo MAVLink. Observe que os diretórios são referentes aos dialetos do protocolo, dentro de cada um deles encontram-se os arquivos ".h" necessários para a utilização do respectivo dialeto.

É importante destacar que manteve-se um bom nível de compatibilidade com as duas bibliotecas e, a princípio, caso um código funcione com a primeira versão do protocolo MAVLink, são necessários poucos ou nenhum ajuste no código para a troca de e, consequentemente, de uma versão pela outra.

6.2.2 Principais Objetos

Para que seja possível utilizar a biblioteca é essencial entender o seu funcionamento, portanto irei comentar as principais funções e variáveis da biblioteca nesta seção. A partir deste ponto até o final do trabalho sempre que não especificado estarei falando do protocolo MAVLink versão 1 pois este foi o protocolo utilizado durante o trabalho, portanto é importante destacar que no caso de se usar a segunda versão do protocolo pequenos ajustes podem ser necessários.

6.2.2.1 Objetos Específicos de cada Mensagem

Os objetos específicos são os objetos declarados dentro dos arquivos ".h" específicos de cada mensagem, portanto sua aplicação se restringe a uma mensagem em específico.

Como explicado na seção 6.1.1 cada mensagem possui um *payload* diferente, logo é

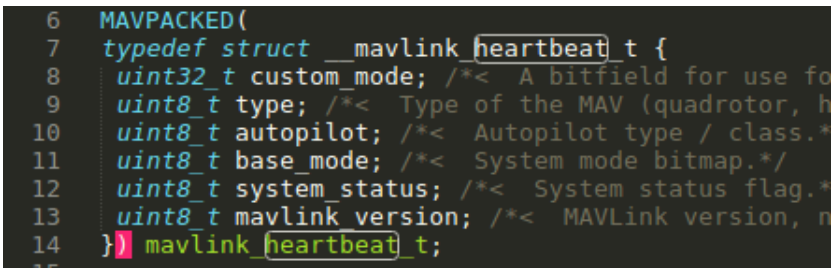

Figura 6.4: Definição da estrutura "mavlink_heartbeat_t".

necessário que para cada tipo de mensagem exista uma estrutura que separe em variáveis os dados do *payload* da mensagem. Essas estruturas são nomeadas "mavlink_*_t" , em que "*" representa o nome da mensagem. A figura 6.4 contém a definição de uma dessas estruturas, a mensagem a qual ela se refere é o *heartbeat*.

É interessante destacar que o endereço de destino da mensagem quando existe é determinado pelo *payload* e, consequentemente, deve ser escrito nessa estrutura. O nosso exemplo da figura 6.4 é de uma mensagem que sempre será *broadcast* e, portanto, não possui uma variável para endereço de destino.

Pelo mesmo motivo são necessários também duas funções específicas. Uma para extrair as informações de uma mensagem e preencher as variáveis da estrutura e outra para preencher uma mensagem com as variáveis contidas em uma estrutura.

Essas funções são nomeadas "mavlink_msg_*_decode" e "mavlink_msg_*_encode" respectivamente. A primeira recebe dois argumentos, um ponteiro para uma estrutura de mensagem, que será comentada na seção 6.2.2.2, e um ponteiro para a estrutura específica da mensagem, essa função apenas transcreve o conteúdo do *payload* da estrutura da mensagem nas variáveis da estrutura específica. A segunda recebe quatro parâmetros o *system ID* e *component ID* do dispositivo que está enviando a mensagem, um ponteiro para a estrutura de mensagem e um ponteiro para a estrutura específica, essa função preenche as variáveis da estrutura de mensagem com o conteúdo da estrutura específica.

Dois exemplos de funções específicas são as funções "mavlink_msg_follow_target_encode()" e "mavlink_msg_follow_target_decode()".

6.2.2.2 Objetos não Específicos de cada Mensagem

Os objetos não específicos de cada mensagem são os objetos que são utilizados por toda a biblioteca MAVLink e, portanto, sua aplicação não está restrita a um tipo específico de mensagem. Esses objetos são utilizados por todos os dialetos.

A estrutura definida na biblioteca mais utilizada pelo usuário provavelmente é a "**mavlink_message_t**". Essa é uma estrutura que possui como variáveis os campos da mensagem MAVLink apresentados na figura 6.1. Todas as mensagens, antes de serem enviadas ou após serem recebidas, devem ser transcritas para uma estrutura desse tipo para que possam

```
114  MAVPACKED(
115  typedef struct __mavlink_message {
116      uint16_t checksum; ///< sent at end of packet
117      uint8_t magic;   ///< protocol magic marker
118      uint8_t len;     ///< Length of payload
119      uint8_t seq;     ///< Sequence of packet
120      uint8_t sysid;   ///< ID of message sender system/aircraft
121      uint8_t compid;  ///< ID of the message sender component
122      uint8_t msgid;   ///< ID of message in payload
123      uint64_t payload64[(MAVLINK_MAX_PAYLOAD_LEN+MAVLINK_NUM_CHECKSUM_BYTES+7)/8];
124  }) mavlink_message_t;
```

Figura 6.5: Definição da estrutura "mavlink_message_t".

ser compreendidas pelas outras funções. A figura 6.5 mostra definição dessa estrutura.

Uma vez definida a estrutura de mensagem devemos destacar mais duas funções que são usadas para decifrar uma mensagem MAVLink e escreve-la em uma estrutura de mensagem e transcrever uma estrutura de mensagem para uma mensagem MAVLink. Essas mensagens são, respectivamente, "mavlink_parse_char" e "mavlink_msg_to_send_buffer".

A primeira função avalia cada byte recebido individualmente até que se tenha recebido uma mensagem, ao final ele transcreve a mensagem na estrutura de mensagem. Essa função recebe quatro parâmetros o canal, um caractere, um ponteiro para a estrutura de mensagem e um ponteiro para uma estrutura de *status*. Essa função retorna 1 quando uma nova mensagem foi transcrita para a estrutura da mensagem caso contrário retorna 0.

- O canal é utilizado para que se possa estabelecer comunicação com mais de um dispositivo MAVLink simultaneamente, ele é utilizado para que, em *software*, seja possível diferenciar as mensagens originadas de um canal das mensagens originadas de outro canal.
- O caractere é o último caractere da mensagem que foi lido pela porta serial.
- A o ponteiro para a estrutura da mensagem é aonde a mensagem será escrita quando todos os caracteres da mensagem tiverem sido obtidos e não houver erro de integridade detectado. Até a obtenção da mensagem ser um sucesso ela fica registrada em uma estrutura estática da função.
- Por último a estrutura de *status* é uma estrutura utilizada para registrar a situação atual do processamento da mensagem MAVLink. Como essa função decodifica caractere por caractere é necessário que a situação do processo de decifra da mensagem total seja salvo entre as chamadas da função, portanto usam-se variáveis estáticas. A estrutura de *status* será uma cópia da estrutura estática da função.

A segunda função é bem mais simples, ela recebe dois parâmetros um vetor de caracteres (*string*) e um ponteiro para a estrutura da mensagem. A função apenas escreve a mensagem da estrutura ordenadamente na *string* e retorna o tamanho da *string*.

6.3 Implementação

Uma vez compreendido o funcionamento da biblioteca podemos começar a implementar funções e tentar ler e escrever mensagens. Nessa seção iremos aplicar o conhecimento explicado neste capítulo e nos capítulos anteriores para realizar uma tarefa básica de envio e leitura de uma mensagem utilizando o protocolo MAVLink.

Uma ideia para a utilização do código seria implementar duas *threads*, sendo uma para leitura e outra para escrita, enquanto o processo principal ficaria encarregado de processar as mensagens.

Inicialmente como o sistema ainda não está pronto a única informação enviada pela *thread* de escrita são sinais de *heartbeat*, posteriormente ela pode servir para envio de sinais como posição e velocidade atual. É importante destacar que uma *thread* responsável pelo envio do *heartbeat* ou do sinal do cão de guarda não deve ficar totalmente alheia à operação do sistema, estas possíveis *threads* devem sempre observar o estado atual do sistema.

Essa abordagem foi baseada no exemplo de implementação do protocolo MAVLink.[6] Entretanto nos deparamos com algumas complicações do exemplo e optamos por desenvolver um sistema parecido.

Abrindo um parênteses em nossa implementação irei, brevemente, explicar e comentar o funcionamento das *threads*.

Threads são sequências de um programa principal que podem ser executadas concorrentemente com este. A execução das *threads* é administrada pelo escalonador do processador que alterna entre a execução dessas tarefas rapidamente de modo que, para o usuário, as *threads* e o programa principal aparentem operar simultaneamente.

Em consequência da utilização das *threads* também é necessário a utilização de um "*mutex*" para evitar que duas *threads* acessem simultaneamente a mesma porta serial (seção crítica). Para essa fase da implementação não foi determinada prioridade para as *threads* nem para o *mutex*, porém ao se implementar o sistema de controle é recomendável alterar as configurações padrão das funções de inicialização destes objetos.

6.3.1 Código

6.3.1.1 Porta Serial

O primeiro segmento de código necessário para a implementação do *software* é um código para iniciação e configuração da porta serial. Como não sabíamos ao certo como a porta serial deveria ser configurada optamos por aproveitar um código de exemplo com pequenas modificações, como explicado na seção 6.3.

[6] Disponível em `https://mavlink.io/en/examples/c_uart.html`

A parte do código referente configuração da porta serial, como utilizado, é composto por três funções. A função "uart_init", que apenas chama as outras duas, "setup_port" e "open_port".

A função **"uart_init"** recebe como argumentos uma *string* contendo o nome do arquivo "tty" correspondente à porta serial e um inteiro equivalente à velocidade da porta serial em *baudrate*. Essa função retorna um inteiro "0" em caso de falha ou, em caso de sucesso, retorna o *file descriptor* (FD) da porta serial. Essencialmente essa função apenas chama as funções "setup_port" e "open_port".

A função **"setup_port"** recebe como argumentos o FD da porta serial e a velocidade em baud. Em caso de falha essa função retorna "false" e em caso de sucesso retorna "true". Essa função configura a comunicação serial basicamente zerando diversas flags referentes a entrada, saída, processamento de linhas e processamento de caracteres, ou seja, transforma a comunicação serial na sua essência, além disso define o nosso padrão dom 8 bits de dado e sem bit de parada. Por último a função ajusta a velocidade e descarta o que foi escrito na porta serial antes de sua iniciação.

A função **"open_port"** recebe como argumento apenas um ponteiro para a *string* que contém o nome do arquivo "tty". Essa função retorna a saída da função "open" que é o FD em caso de sucesso ou -1 em caso de erro. Essencialmente essa função apenas executa a função "open" e verifica seu resultado.

Para organizar as informações relativas ao canal de comunicação ou à porta serial foi definida uma estrutura que associa o FD da porta serial, o canal de comunicação, a existência de mensagem lida pela *thread* e ainda não processada, a última mensagem lida pela *thread* e o *mutex* responsável pelo bloqueio do canal. A definição dessa estrutura está no código 6.1.

Listagem 6.1: Definição de estrutura com dados do canal.

```
1 typedef struct __serial_port {
2         int fd;
3         int chan;
4         int new_msg;
5         pthread_mutex_t lock_chan;
6         mavlink_message_t last_msg;
7 } serial_port_t;
```

6.3.1.2 Leitura

Como explicado anteriormente a operação de leitura principal ocorrerá por meio de uma *thread*, portanto é necessário que tenhamos duas funções uma função padrão para leitura do protocolo MAVLink e outra função infinita (*thread*) que recursivamente lê as mensagens e escreve na estrutura do canal para processamento futuro.

Observe que optou-se por não adicionar um *buffer* de mesnagens a serem processadas pois acredita-se que não se faz necessário. Porém caso testes futuros apontem a necessi-

dade de um *buffer* para essas mensagens basta modificar a estrutura da porta serial para que "new_msg" e "last_message" sejam vetores de inteiros e vetor de estrutura de mensagem, respectivamente, e operem como filas.

A primeira função **"read_message"** recebe como parâmetros o FD da porta serial, o endereço do *mutex* da porta serial, um ponteiro para a mensagem em que a saída será escrita e um inteiro para o canal. Observe que todos os parâmetros recebidos por essa função estão presentes na estrutura "serial_port", porém optou-se por mante-los separados pois muitos testes a usavam desta maneira. Como essa foi uma tarefa realizada em um dos testes essa redundância pode ser removida posteriormente ou pode ser feita uma função que entre os parâmetros da estrutura na função "read_message". Essa função retorna -1 caso o *buffer* da porta serial esteja vazio, 0 caso tenha processado algum byte da porta serial e 1 caso haja mensagem nova. Essencialmente essa função administra o *mutex* e realiza um *switch-case* para alteração do canal. O código 6.2 é o código fonte dessa função.

Listagem 6.2: Função para leitura de mensagens MAVLink.

```
1  int read_message(int fd, pthread_mutex_t* lock_chan, mavlink_message_t*
       message, int chan)
2  {
3      uint8_t           cp;
4      mavlink_status_t status;
5      uint8_t           msgReceived = false;
6
7      int auxiliar = 0;
8
9          pthread_mutex_lock(lock_chan);
10
11         auxiliar = read(fd, &cp, 1);
12
13         pthread_mutex_unlock(lock_chan);
14
15         if(auxiliar <= 0)return -1;//Retorna -1 caso buffer vazio
16
17         switch (chan)
18         {
19             default:
20             return -2;
21
22             case pixhawk_chan:
23                 msgReceived = mavlink_parse_char(MAVLINK_COMM_0, cp,
                       message, &status);
24             break;
25
26             case modem_chan:
27                 msgReceived = mavlink_parse_char(MAVLINK_COMM_1, cp,
                       message, &status);
28             break;
29
```

```
30              case 2:
31                  msgReceived = mavlink_parse_char(MAVLINK_COMM_2, cp,
                        message, &status);
32              break;
33
34              case 3:
35                  msgReceived = mavlink_parse_char(MAVLINK_COMM_3, cp,
                        message, &status);
36              break;
37          }
38
39      return msgReceived;//Retorna 0 enquanto nao terminar leitura e 1
            quando houver mensagem nova
40  }
```

Em seguida temos a *thread* de leitura que recebe como parâmetros apenas um ponteiro para a estrutura da porta serial. Como normalmente ocorre a *thread* não retorna nada e, nesse caso, é um loop infinito, portanto para interrompe-la deve-se utilizar a função "pthread_cancel". A *thread* apenas chama a função "read_message" com os parâmetros da porta serial e atualiza a estrutura da porta serial quando necessário. O código 6.3 é o código da *thread* de leitura.

Listagem 6.3: *Thread* de leitura de mensagens MAVLink.

```
1  void* read_thread(void* _serial_port)
2  {
3      serial_port_t* serial_port = (serial_port_t*) _serial_port;
4      int msgReceived;
5
6      while(1)
7      {
8          do
9          {
10             msgReceived = read_message(serial_port->fd, &serial_port->
                   lock_chan,&serial_port->last_msg, serial_port->chan);
11         }while(msgReceived == 0);
12
13         if(msgReceived == 1)serial_port->new_msg = 1;
14
15         usleep(500); //tempo para outras funcoes
16     }
17 }
```

6.3.1.3 Escrita

Semelhantemente às funções de leitura foram feitas uma função de escrita que utiliza o protocolo MAVLink e uma *thread* para envio do *heartbeat* e, futuramente, dados de posição e velocidade.

A função **"write_message"** recebe o FD da porta serial, o endereço do *mutex* do canal, um endereço para uma mensagem MAVLink e um inteiro correspondente ao canal. Pelo mesmo motivo apresentado na seção 6.3.1.2 optou-se por dividir os parâmetros da estrutura da porta serial na entrada dessa função. Essa função retorna o mesmo que a função "write", ou seja, em caso de erro retorna -1 e em caso de sucesso o número de bytes escritos. O código 6.4 contém o código da função "write_message".

Listagem 6.4: Função para escrita de mensagens MAVLink.

```
1  int write_message(int fd, pthread_mutex_t* lock_chan, mavlink_message_t
       message)
2  {
3      char buf[300];
4
5      unsigned len = mavlink_msg_to_send_buffer((uint8_t*)buf, &message);
6
7      pthread_mutex_lock(lock_chan);
8
9      const int bytesWritten = write(fd, buf,len);
10
11     pthread_mutex_unlock(lock_chan);
12
13     return bytesWritten;
14 }
```

A *thread* de escrita está apresentada no código 6.5. A *thread* de leitura recebe como argumentos um ponteiro para uma estrutura da porta serial e utiliza-se também de uma estrutura global de *heartbeat*, por meio dessa estrutura global é possível manter a *thread* de escrita atualizada quanto a situação do resto do sistema. Uma opção é transformar o *heartbeat* global em uma entrada para a *thread*. A *thread* de escrita não retorna nada.

Listagem 6.5: *Thread* de escrita de mensagens MAVLink.

```
1  void* write_thread(void* _serial_port)
2  {
3      serial_port_t* serial_port = (serial_port_t*) _serial_port;
4      mavlink_message_t msg;
5      mavlink_heartbeat_t hb;
6
7      while(1)
8      {
9          hb = global_hb_message;
10         mavlink_msg_heartbeat_encode(my.sysid,my.compid,&msg,&hb);
11         write_message(serial_port->fd, &serial_port->lock_chan, msg);
12
13         usleep(1000000); //f = 1 hz
14     }
15 }
```

6.3.1.4 Inicialização e Finalização

O primeiro ponto que devemos destacar nessa seção é que a inicialização da estação terra será ligeiramente diferente da inicialização das aeronaves e, portanto, são necessárias duas funções, uma para a estação terra e outra para os sistemas aéreos, contudo não foram feitas sub-rotinas para o processo de inicialização e, consequentemente, muitas tarefas das duas funções ficaram repetidas.

Foram criadas, portanto, duas funções, uma de inicialização e outra de finalização, para os sistemas aéreos e outras duas para a estação terra. Para evitar as repetições já explicadas irei tratar nesta seção apenas das funções do sistema aéreo e a partir destas informações é fácil compreender as funções para a estação terra.

A função **"sys_init"** recebe como argumentos dois ponteiros para estruturas de portas seriais e duas estruturas *mutex*. Essa função sempre retorna 1. Essa função define os parâmetros globais do sistema que são o ID do sistema, ID do componente e o *heartbeat*; chama as funções de inicialização da porta serial, do *mutex* e das *threads*; define os parâmetros da estrutura da porta serial com os dados obtidos; e define o ID do sistema da Pixhawk para ser igual ao ID do sistema atual. Futuramente esta função também deverá iniciar o pulso PWM para o cão de guarda. O código da função é o código 6.6 deste documento.

Por último é valido destacar que não foi criada uma *thread* de escrita para a comunicação com a Pixhawk, entretanto não haveriam problemas de se criar uma bastaria acrescentar uma linha ao código 6.6.

Listagem 6.6: Inicialização do sistema aéreo.

```
1  int sys_init(serial_port_t* sp_px, serial_port_t* sp_modem,
       pthread_mutex_t lock, pthread_mutex_t lockmod)
2  {
3      mavlink_heartbeat_t hb;
4
5      my.sysid = Plane1_system_id;
6      my.compid = 2;
7
8      sp_px->fd = uart_init(uart0_name,uart0_baud);
9      sp_px->chan = pixhawk_chan;
10     sp_px->lock_chan = lock;
11
12     pthread_mutex_init(&sp_px->lock_chan,NULL);
13
14     pthread_create(&thread_leitura[sp_px->chan], NULL, read_thread, sp_px
           );
15
16     espera_hb(sp_px, &hb,0);   //espera por um sinal da px
17
18     if(Pixhawk.sysid != my.sysid)   //Define ID da px
19     {
```

```
20          set_pixhawk_id(sp_px, my.sysid);
21          espera_hb(sp_px, &hb,0);
22      }
23
24      global_hb_message = hb;
25
26      sp_modem->fd = uart_init(uart2_name, uart2_baud);
27      sp_modem->chan = modem_chan;
28      sp_modem->lock_chan = lockmod;
29
30      pthread_mutex_init(&sp_modem->lock_chan,NULL);
31
32      pthread_create(&thread_leitura[sp_modem->chan], NULL, read_thread,
            sp_modem);
33      pthread_create(&thread_escrita[sp_modem->chan], NULL, write_thread,
            sp_modem);
34
35      //pwm_init();//Aqui entra a iniciacao do pwm do watchdog
36
37      return 1;
38  }
```

A função **"sys_stop"** recebe como argumentos as duas estruturas de portas seriais e sempre retorna 1. Essencialmente essa função primeiro modifica o sinal de *heartbeat* global para sinalizar que está se desligando e o envia e, em seguida, finaliza as *threads*, os *mutex* e encerra os FDs. Essa função está apresentada no código 6.7

Listagem 6.7: Finalização do sistema aéreo.

```
1   int sys_stop(serial_port_t* sp_px, serial_port_t* sp_modem)
2   {
3       mavlink_message_t msg;
4
5       global_hb_message.system_status = MAV_STATE_POWEROFF;
6       mavlink_msg_heartbeat_encode(my.sysid,my.compid,&msg,&
            global_hb_message);
7       write_message(sp_modem->fd, &sp_modem->lock_chan, msg);
8
9       //pwm_stop();//Aqui entra a finalizacao do pwm
10
11      pthread_cancel(thread_leitura[sp_px->chan]);
12      pthread_cancel(thread_leitura[sp_modem->chan]);
13      pthread_cancel(thread_escrita[sp_modem->chan]);
14
15      pthread_mutex_destroy(&sp_modem->lock_chan);
16      pthread_mutex_destroy(&sp_px->lock_chan);
17
18      close(sp_modem->fd);
19      close(sp_px->fd);
20
```

```
21      return 1;
22  }
```

6.3.1.5 Processamento das mensagens

Como já foi possível observar processamento das mensagens não ocorre nas *threads* de leitura, nesse primeiro momento ela está ocorrendo na *main*, entretanto o ideal seria que ela operasse em uma *threads* e na função principal teríamos a operação do sistema de controle. Como estamos na fase de testes e implementações o programa opera desta maneira, entretanto espera-se que no futuro a função de processamento de mensagens se tone uma *thread*.

O processamento das mensagens ocorre com um *switch-case* que determina qual a mensagem recebida e, em seguida, chama uma função para o tratamento dessa mensagem. A essa função foi dado o nome de **"sys"**. Essa função é aonde devem se concentrar os próximos trabalhos pois atualmente apenas 1 mensagem MAVLink está completamente implementada e para que se possam iniciar as atividades de vôo é necessário que as principais mensagens estejam operando.

A função **"sys"** está apresentada no código 6.8, para a realização dos testes determinou-se que a função iria operar até que a estação base fosse desligada, ou seja, até que uma mensagem de *heartbeat* indicando o inicio do processo de desligamento fosse recebida.

Listagem 6.8: Função para processamento das mensagens.

```
1  int sys()
2  {
3      serial_port_t sp_modem, sp_px;
4      pthread_mutex_t lock_modem, lock_px;
5      mavlink_message_t msg_recebida;
6      mavlink_heartbeat_t hb;
7      mavlink_ping_t ping;
8
9      sys_init(&sp_px, &sp_modem, lock_px, lock_modem);
10
11     while(hb.system_status != MAV_STATE_POWEROFF) //Ate o GCS desligar
12     //(utilizado apenas para testes)
13     {
14         while(sp_modem.new_msg == 0); //espera nova mensagem
15         msg_recebida = sp_modem.last_msg;
16         sp_modem.new_msg = 0;
17
18         switch(msg_recebida.msgid)
19         {
20             case MAVLINK_MSG_ID_HEARTBEAT://atualiza condicao de saida
21                 mavlink_msg_heartbeat_decode(&msg_recebida,&hb);
22             break;
```

```
23
24              case MAVLINK_MSG_ID_PING: //completamente implementada
25                  mavlink_msg_ping_decode(&msg_recebida, &ping);
26                  ping_handler(&sp_modem, &ping, &msg_recebida);
27              break;
28
29              case MAVLINK_MSG_ID_PARAM_SET: //parcialmente implementada
30                  param_set_handler(&sp_modem, &msg_recebida);
31              break;
32
33              default: //Descarta mensagem nao implementada
34              break;
35          }
36      }
37
38      sys_stop(&sp_px, &sp_modem);
39
40      return 1;
41  }
```

As funções **"*_handler"** são funções para realizar as tarefas requisitadas pelas mensagens. Observe que apenas a mensagem de resposta ao teste de latência está implementada. Portanto ela será usada como exemplo para a implementação de funções deste tipo no futuro.

A função **"ping_handler"** recebe como parâmetros um ponteiro para a estrutura da porta serial do modem, um ponteiro para a estrutura que contém a mensagem MAVLink e um ponteiro para a estrutura específica *ping*. Essa função retorna 1 caso tenha enviado uma resposta à requisição de *ping* e 0 caso a mensagem recebida não fosse uma requisição de *ping*, mas uma resposta a alguma requisição, ou seja estava endereçada. Destaca-se que essa função retorna a mensagem com sem modificar a lacuna do tempo, isso ocorre porque o relógio do computador embarcado não é confiável uma vez que ele nunca foi ajustado e não temos garantia de que ele continuará correto depois de ajustado.

Listagem 6.9: Função de processamento da mensagem de *ping*.

```
1  int ping_handler(serial_port_t* serial_port, mavlink_ping_t* ping,
       mavlink_message_t* msg_recebida)
2  {
3      mavlink_message_t msg;
4
5      if((ping->target_system == 0) && (ping->target_component == 0))
6      {
7          ping->target_component = msg_recebida->compid;
8          ping->target_system = msg_recebida->sysid;
9
10         mavlink_msg_ping_encode(my.sysid,my.compid,&msg,ping);
11
12         write_message(serial_port->fd, &serial_port->lock_chan, msg);
13         return 1; //ping respondido
```

```
14      }
15
16      return 0;//nada foi feito
17  }
```

6.4 Testando o Atraso

Os testes das outras implementações foram suprimidos deste trabalho, pois o teste realizado nesta seção exige, naturalmente, que todas as outras etapas estejam funcionando adequadamente.

Uma das maiores preocupações quando se fala de sistemas de controle que dependem de sistemas de comunicação é o atraso gerado por esta comunicação. Com isso em mente é que passamos para essa seção do trabalho que consiste na determinação do atraso pela comunicação entre sistemas.

A estratégia para a estimação deste atraso será por meio da utilização da mensagem de *ping* do protocolo MAVLink. No entanto sabemos que o relógio do computador embarcado não é confiável uma vez que não podemos sincroniza-lo adequadamente, portanto iremos repetir a mensagem de *ping* recebida pelo computador embarcado à estação terra e a diferença de tempo entre a mensagem recebida e a mensagem enviada será dada pela soma dos tempos de processamento dos dispositivos somado aos atrasos da comunicação. Ou seja:

$$\Delta t = t_{rec} - t_{env} = t_{gum} + t_{gcs} + t_{trans1} + t_{trans2} \tag{6.1}$$

Em que t_{rec}, t_{env}, t_{gum}, t_{gcs}, t_{trans1} e t_{trans2} são, respectivamente, o tempo no instante em que a mensagem de resposta foi identificada pela estação terra, o tempo no instante em que a estação terra iniciou a codificação da mensagem, o tempo para o computador embarcado processar e responder a mensagem, o tempo para a estação terra processar a mensagem e a resposta da mensagem, o tempo de transmissão da GCS ao computador embarcado e o tempo de transmissão do computador embarcado à estação terra.

É intuitivo e foi verificado durante os testes que os tempos de processamento são extremamente inferiores aos tempos de transmissão, os tempos de processamento da estação terra, no pior dos casos, não ultrapassou 30 μs. Portanto:

$$t_{gum} \approx t_{gcs} << t_{trans1} \approx t_{trans2} \tag{6.2}$$

$$\Delta t = 2 \cdot t_{trans} \tag{6.3}$$

Sabemos ainda que o tempo de transmissão t_{trans} pode ser dividido em diversas etapas como, por exemplo, a sincronização, requisição do canal para transmissão, retransmissão

de pacotes, comunicação serial e a taxa da da transmissão sem fio. Por mais que não seja possível determinar todos esses tempos, os tempos da comunicação serial e a taxa da comunicação sem fio são determinados nós. Atualmente a velocidade da comunicação serial com os modens é de 115.200 baud/s e a velocidade da comunicação sem fio entre os modens é de 172.800 bps, observe que para a comunicação serial UART 1 baud/s é equivalente a 1 bps. Portanto temos que, para a mensagem completa de *ping* de 14 bytes de *payload* e 8 bytes de *header*, a mensagem tem um total de 176 bits, para a comunicação serial, no entanto, devemos acrescentar 1 bit de início e 1 bit de parada para cada byte transmitido. Consequentemente:

$$t_{trans} > 2 \cdot \frac{8 \cdot 8 + 64 + 32 + 16 + 2 \cdot 22}{115.200} + \frac{8 \cdot 8 + 64 + 32 + 16 + 2 \cdot 22}{172.800} \tag{6.4}$$

$$t_{trans} > 5,093\, ms \tag{6.5}$$

O tempo de 5,093 ms obtido é apenas uma ilustração de quão rápido o sistema poderia ser, contudo sabemos que atingir esse valor é impossível já que as etapas mais lentas do processo são, na verdade, o controle de tráfego e sincronismo realizado pelo modem.

6.4.1 Método

Para conseguirmos determinar o atraso do sistemas nas condições atuais utilizamos o código apresentado anteriormente para um computador embarcado conectada a um dos modens e um código um pouco modificado para a estação terra. Esse código escreve em um arquivo os tempos Δt em um formato similar à declaração de vetores do MATLAB, seguidos pelo número de mensagens de *ping* enviadas e em um outro arquivo escreve quais sequências de das mensagens foram recebidas, para que seja possível determinar quais foram perdidas.

O tempo é obtido utilizando a função **"get_time_usec"** que estava presente no exemplo obtido no endereço eletrônico do protocolo MAVLink. O essa função está no código 6.10. Essa função foi testada de diversas maneiras antes de ser utilizada no código.

Listagem 6.10: Função para obtenção do tempo em μs.

```
1 double get_time_usec()
2 {
3     static struct timeval _time_stamp;
4     gettimeofday(&_time_stamp, NULL);
5     return _time_stamp.tv_sec*1000000 + _time_stamp.tv_usec;
6 }
```

Portanto ao executar o código é registrado o tempo t_{env} imediatamente antes da codificação da mensagem MAVLink e o tempo t_{rec} é registrado imediatamente depois do programa

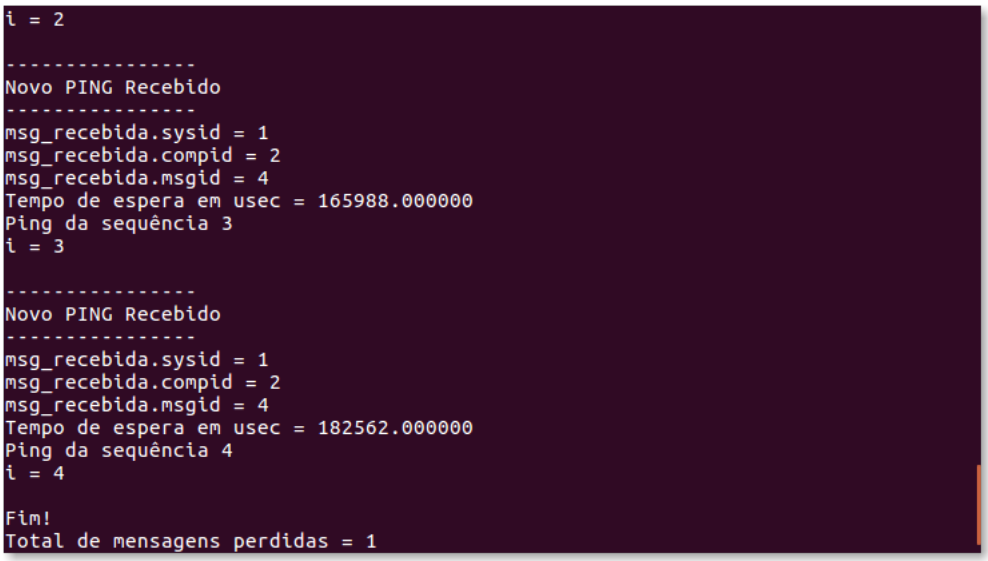

Figura 6.6: Mensagens impressas durante a validação do teste de latência.

identificar o retorno da mensagem de *ping*.

Depois disso as informações apresentadas na figura 6.6 são impressas na tela do computador e o teste se repete. O teste foi efetuado para 12.000 pontos o que totalizou, aproximadamente, 30 minutos de experimento. Todas as mensagens que demoraram mais de 5 segundos para retornarem foram consideradas como perdidas.

O teste foi validado de uma maneira simples e muito semelhante à determinação dos atrasos de processamento. A maior parte do código operava normalmente enquanto a *thread* de leitura não lia uma porta serial, mas um arquivo de texto que continha as mensagens MAVLink. Por fim foram adicionados tempos de espera com a função "usleep" para permitir a verificação da contagem de tempo e quando não foram detectados mais erros na execução do programa prosseguiu-se com os testes. Uma vez que o teste havia sido validado foi verificado que todas as etapas anteriores também funcionavam em boas condições, por este motivo foram suprimidos testes anteriores.

6.4.2 Resultados

Os resultados obtidos foram escritos em um vetor e passados ao MATLAB, é importante destacar um comportamento percebido durante os testes, as primeiras mensagens, quando se iniciava o sistema, eram sempre mais suscetíveis a erros que as demais. Do total de 12.000 testes apenas 31 mensagens foram perdidas, porém em validações de 5 a 10 mensagens frequentemente se perdiam até 3. Inclusive no teste de 12.000 mensagens a primeira mensagem da sequência foi perdida. Como o programa rodado no computador embarcado não estava se encerrando durante as validações e os testes acredita-se que durante os intervalos de execução dos testes o ruído deve ter causado a leitura de algum caractere o que complicava a leitura da mensagem byte a byte realizada pelo código e, consequentemente, perdia(m)-se

a(s) primeira(s) mensagem(ns).

Listagem 6.11: Script para processamento dos dados do teste de latência.

```
clc
a=0;
mensagens_perdidas=[];
while(a<i)
    if(sum(seq(seq==a))==0)
        mensagens_perdidas=[mensagens_perdidas a];
    end
    a=a+1;
end
dadosperdidos = (i-length(t))/i
BERminima = dadosperdidos/(2*176)
media = mean(t/1000)
desvio = std(t/1000)
minimo = min(t/1000)
maximo = max(t/1000)
figure(1)
histogram(t/1000);
grid;
title('Histograma dos atrasos no teste')
xlabel('Tempo (ms)')
ylabel('Repeticoes')
axis([0 450 0 1250])
```

Os dados obtidos foram então processados pelo MATLAB, obtendo-se a média, o desvio padrão e montando um histograma para a análise de comportamento dos dados. O *script* utilizado no MATLAB para o processamento dos dados encontra-se no código 6.11.

Com a execução do *script* 6.11 os resultados obtidos foram 0,26% das mensagens foram perdidas, BER mínima de $7{,}339 \cdot 10^{-6}$, atraso médio de 179,763 ms, desvio padrão de 56,632 ms, o menor atraso obtido 43,279 ms e o maior atraso obtido 1.099 ms. Portanto, para apenas um trecho da comunicação devemos ter:

- 0,26% das mensagens foram perdidas;
- BER mínima de $7{,}339 \cdot 10^{-6}$;
- Atraso médio de 89,882 ms;
- Desvio padrão de 28,316 ms;
- Menor atraso obtido 21,640 ms;
- Maior atraso obtido 549,382 ms.

É importante destacar que para o cálculo da BER não foram considerados retransmissões e foi assumido apenas 1 bit errado por mensagem errada no destino final.

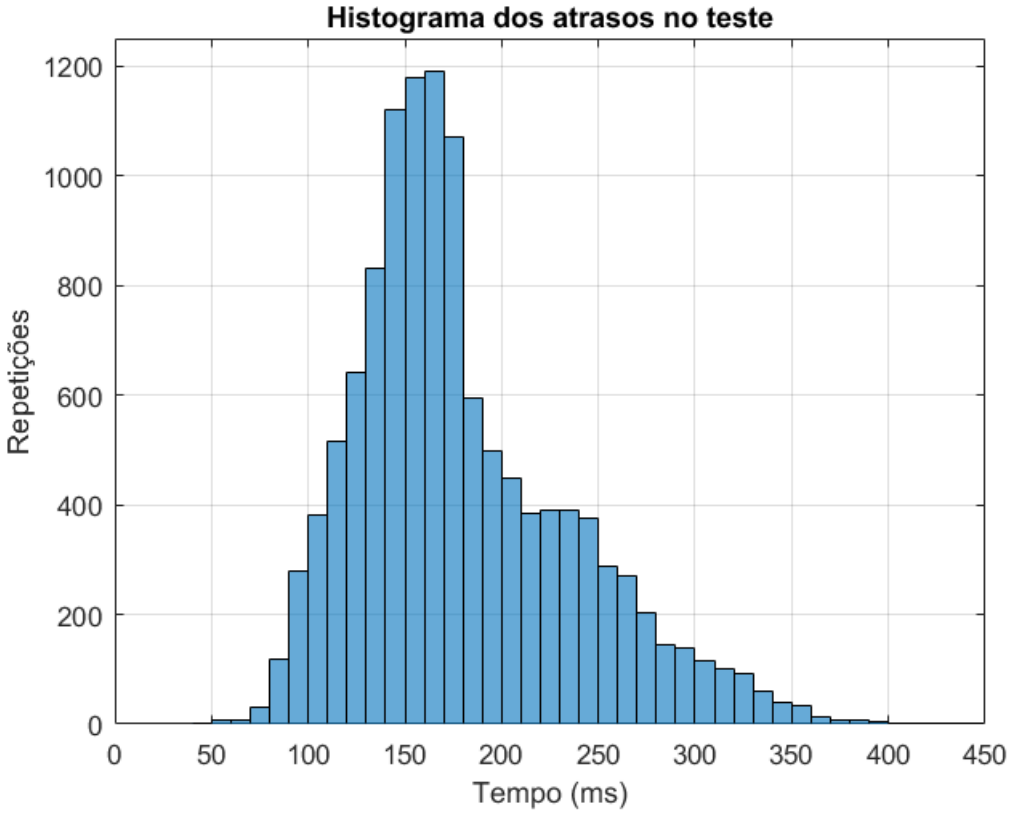

Figura 6.7: Primeiro histograma obtido ao executar o código 6.11.

A figura 6.7 contém o histograma obtido ao executar o código 6.11, observe que este histograma foi obtido com relação ao atraso total, para se determinar o histograma de Δt todos os valores do eixo das abscissas devem ser divididos por 2.

Os resultados obtidos apresentam resultados bem desagradáveis, entretanto provavelmente estão corretos pois a lentidão na transmissão via modem já havia sido observada previamente durante os testes e durante as operações com o computador embarcado. Além disso a distância entre os modens era de aproximadamente 1 metro enquanto o alcance desses modens é de quilômetros e, por último, haviam apenas dois dispositivos se comunicando enquanto, na prática, teremos vários.

Por último é importante destacar que existe sim possibilidade de melhora do sistema, como mostrado na seção 6.4 o atraso ideal mínimo, se as velocidades de transmissão não forem alteradas, é de 5,093 ms. Podemos tentar reduzir a diferença entre o resultado obtido e o resultado ideal alterando algumas das configurações do modem, por exemplo, a quantidade de retransmissões e reduzindo a margem de aleatoriedade do canal de acesso. É evidente que alterar os valores destes parâmetros podem prejudicar outros aspectos da transmissão, entretanto estes dispositivos foram projetados para estarem inseridos em sistemas muito maiores do que os quais estamos lidando, então é razoável que eles não estejam configurados da melhor maneira possível.

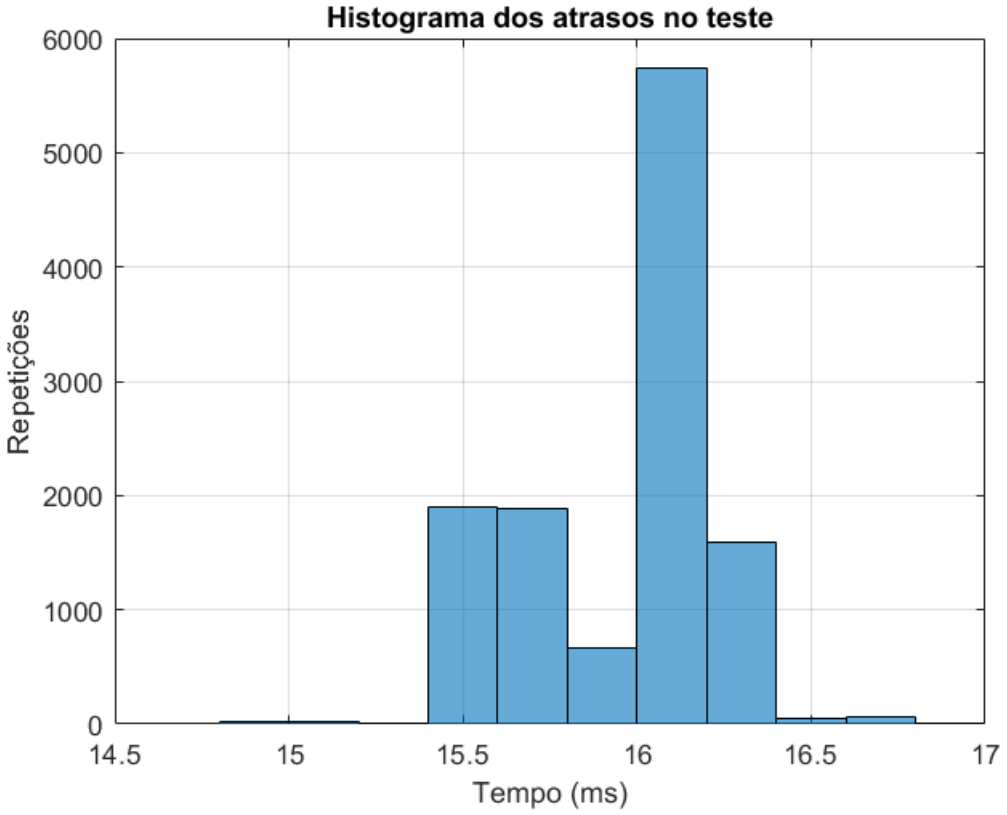

Figura 6.8: Segundo histograma obtido ao executar o código 6.11.

6.4.3 Atraso na comunicação por fio

Durante a validação do teste da seção anterior não foi utilizado o modem para a comunicação dos dois computadores embarcados mas uma conexão direta com fios nos terminais. Comentar os resultados dessa validação é interessante. O procedimento dessa seção foi exatamente o mesmo procedimento descrito na seção anterior. A única diferença é que, desta vez, a comunicação entre estação terra e computador embarcado não era feita por um modem mas por um fio padrão. Com exceção desse fator nada no teste foi alterado.

Dessa vez os resultados obtidos com a execução do *script* 6.11 foram nenhuma mensagem perdida, atraso médio de 16,039 ms, desvio padrão de 1,353 ms, menor atraso de 14,944 ms e atraso máximo de 64,410 ms. Consequentemente, para Δt os resultados foram:

- nenhuma mensagem perdida;
- Atraso médio de 8,020 ms;
- Desvio padrão de 676,4 μs;
- Menor atraso obtido 7,472 ms;
- Maior atraso obtido 32,205 ms.

A figura 6.8 contém o histograma obtido após a execução do código. Para terminar a análise é importante destacar que até mesmo a comunicação serial por fio distanciou-se do resultado mínimo apresentado no início desta seção. Destaca-se ainda que aquele resultado mínimo é superior ao resultado mínimo que seria calculado para a comunicação por fio uma vez que está envolve apenas uma comunicação serial direta entre estação terra e computador embarcado.

6.4.4 Atraso na comunicação a uma distância de 200 m

Por fim, para finalizar o trabalho foi realizado mais um teste semelhante aos testes das seções anteriores contudo desta vez foram utilizados os modens e a distância entre a estação terra e o computador embarcado era de aproximadamente 200 m.

Nesta execução do teste das primeiras 10 requisições de ping apenas as duas últimas obtiveram sucesso, como estes foram pontos que se distanciaram exageradamente dos outros 11990 optamos por descartar as 10 primeiras amostras deste teste. Consequentemente, para Δt os resultados foram:

- 78 mensagens perdidas (0,65%);
- Atraso médio de 90,6684 ms;
- Desvio padrão de 31,3744 μs;
- Menor atraso obtido 23,0955 ms;
- Maior atraso obtido 313,2125 ms.

A figura 6.9 contém o histograma obtido após a execução do código. Comparando os resultados deste teste com os testes anteriores percebemos que a forma do histograma só se alterou significativamente na comunicação por fio e o aumento da distância na comunicação com modem não influenciou significativamente nos tempos dos atrasos, porém piorou drasticamente o número de mensagens perdidas.

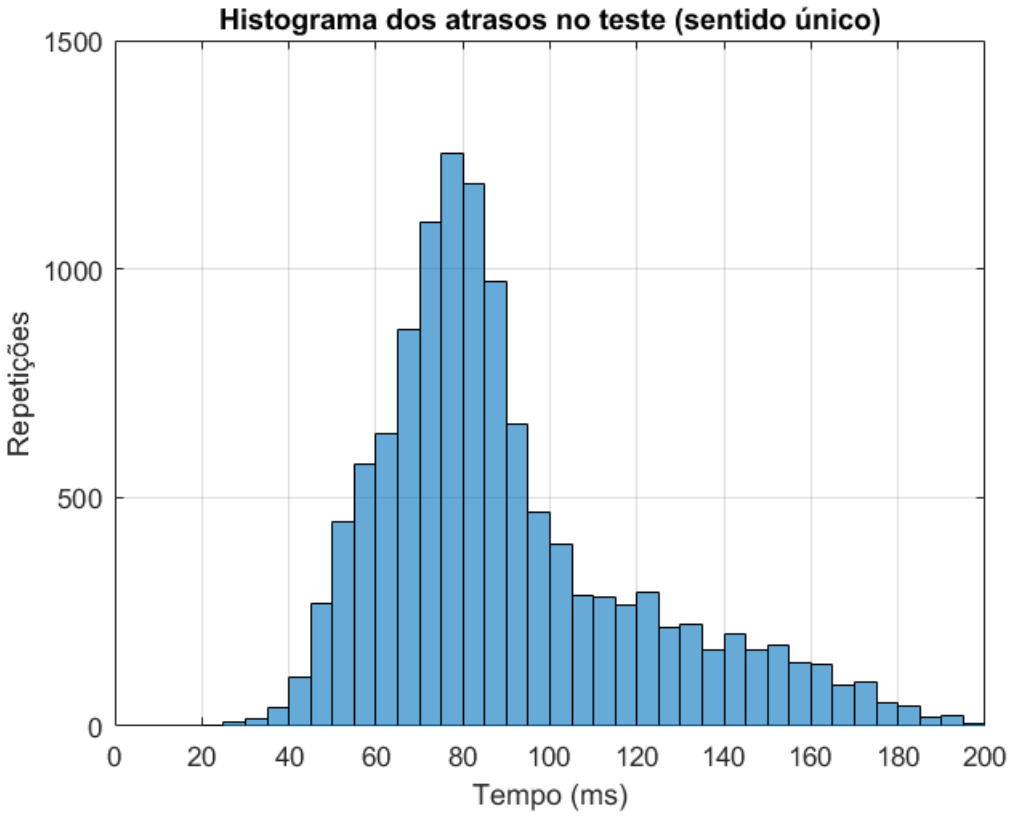

Figura 6.9: Terceiro histograma obtido ao executar o código 6.11.

Capítulo 7

Conclusão

Neste trabalho foi possível entender a níveis mais profundos o funcionamento de computadores, o processo de montagem da imagem de um sistema operacional, do sistema operacional linux, de microprocessadores, de modens, de sistemas de comunicação e interfaces de comunicação.

Na primeira parte do trabalho as dificuldades encontradas estavam em dois aspectos principais, definir qual seria a melhor opção de sistema operacional a ser utilizado e encontrar documentação relativa aos procedimentos de instalação deste sistema operacional, veja que a documentação para esta etapa nunca foi encontrada, o que ocorreu, na verdade, foi um compilado de documentações, acerca do mesmo assunto, que juntas funcionavam. Uma vez obtidas essas documentações, foi possível montar, modificar e instalar o sistema operacional em nosso computador embarcado.

Na segunda etapa do trabalho, o real desafio estava na utilização de um novo sistema operacional, encontrar métodos para realização dos objetivos e por último a implementação do código para se obter o objetivo final desejado nas condições desejadas. Apesar de o objetivo final da seção de implementar a comunicação serial entre computadores embarcados do computador embarcado ter sido atingido não foi possível obter o melhor resultado possível nas condições ideais para o objetivo secundário de controle da interface GPIO.

A terceira etapa do trabalho foi uma etapa de testes, estudo e planejamento, aprendeu-se muito sobre um dos dispositivos que será utilizado no projeto. Além disso o dispositivo foi testado e opera com condições normais.

A quarta etapa do projeto envolveu o projeto de um *hardware* e a montagem de um protótipo para a continuação das atividades enquanto não tínhamos os dispositivos definitivos. Essa etapa foi extremamente prática, envolveu inúmeras falhas e os mais variados problemas que demandavam muito tempo para serem encontrados, muitos deles fáceis de se solucionar porém quando não encontrados impediam o progresso dos trabalhos.

A quinta etapa envolve o desenvolvimento do *software* está, sem dúvidas, é uma etapa que irá demandar muito tempo na continuação deste trabalho. Apesar de não serem ne-

cessárias a implementação de todas as mensagens MAVLink no código muitas mensagens ainda precisam ser implementadas para que se tenha o mínimo necessário para a operação do sistema, além disso a implementação de cada mensagem por si só já demanda bastante tempo.

Ao final foi realizado um teste do sistema que nos apresentou uma falha no tempo de entrega das mensagens pelo modem, aparentemente podemos configurar o modem de outra maneira para tentar reduzir esse atraso. Entretanto não sabemos ao certo o quanto podemos reduzi-lo e, portanto, fica um desafio para as próximas etapas do trabalho. É válido mencionar que esse atraso pode complicar significativamente o sistema de controle que será implementado nas aeronaves.

O objetivo inicial deste era desenvolver um sistema de comunicação multiplataforma para os VANTs e a estação terra, ou seja, incluir o microcontrolador e os modens ao sistema e implementar o sistema de comunicação, tanto computador embarcado comunicando-se com a Pixhawk quanto a comunicação computador embarcado-modem. Na situação atual dos trabalhos foi desenvolvido um módulo para a realização desta tarefa. A partir dos testes com o protótipo percebemos que a comunicação do computador embarcado com o modem e a Pixhawk dispositivos funciona, desde que sejam utilizadas alguma das mensagens já implementadas. Como os dispositivos de diversas plataformas se comunicam e o protótipo opera em boas condições é evidente que os objetivos do trabalho foram atingidos.

Distanciando-se um pouco dos resultados físicos do projeto é necessário destacar que muito conhecimento e experiência no desenvolvimento de projetos foram adquiridos durante a realização deste trabalho, sobretudo na área de desenvolvimento de *software* e funcionamento de microcontroladores.

Para trabalhos futuros recomenda-se continuar com a implementação do *software* produzindo mais algumas funções para o processamento das mensagens, criar algumas sub-rotinas para reorganizar as funções de inicialização do sistema, implementação do pulso PWM para a operação do cão de guarda, tentar diminuir o atraso de comunicação causado pelos modens, realização de testes semelhantes aos realizados no 6 com mais de um computador embarcado simultaneamente para observar o efeito da colisão e o desenvolvimento da lei de controle.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ROCHA, E. M. C. Desenvolvimento de um sistema com veículos aéreos não-tripulados autônomos. *Faculdade de Tecnologia, Universidade de Brasília*, 2017.

[2] CORDEIRO, T. F. K. Desenvolvimento de um sistema com veículos aéreos não-tripulados autônomos. *Faculdade de Tecnologia, Universidade de Brasília*, 2018.

[3] CAMPA, G. et al. Design and flight-testing of non-linear formation control laws. *Control Engineering Practice 15*, p. 1077–1092, 2007.

[4] LI, B. et al. Development and Testing of a Two-UAV Communication Relay System. *Multidisciplinary Digital Publishing Institute*, 2016. Disponível em: <https://www.mdpi.com/>.

[5] TEXAS INSTRUMENTS. *AM/DM37x Multimedia Device Technical Reference Manual*. 12500 T I Blvd, Dallas, TX 75243, EUA, 2012. Version R. Disponível em: <http://www.ti.com/>.

[6] MICROHARD SYSTEMS INC. *Pico Series P900 Operating Manual*. 150 Country Hills Landing NW, Calgary, AB T3K 5P3, Canadá, 2016. v1.8.7. Disponível em: <http://www.microhardcorp.com/>.