Monografia:

Projeto de uma Arquitetura de Hardware e Software para um Veículo Aéreo Não-Tripulado para Supervisão de Instalações de Petróleo

Djalma Teixeira Maranhão Neto

Natal, Agosto de 2009

UFRN - CT - NUPEG - Campus Universitário - CEP: 59070-970 - Natal-RN - Brasil
Fone-Fax: (84) 215-3773 - www.nupeg.ufrn.br - prhanp14@nupeg.ufrn.br

Dedico esse trabalho a meus amados pais, não apenas em reconhecimento por terem me dado vida. Mas, por sempre incutir em mim a necessidade de lutar pelos meus sonhos. E por estarem sempre ao meu lado nos momentos alegres e especialmente nos momentos difícies. O seu amor e sua dedicação são sem dúvida o combustível das minhas realizações.

Agradecimentos

Ao meu orientador e ao meu co-orientador, professores Pablo e Adelardo, sou grato pela orientação e amizade.

Aos colegas João Paulo e Gutemberg pelo apoio ao longo desse trabalho.

Aos demais colegas do Laboratório de Robótica pelas sugestões e críticas.

A minha família pela compreensão e apoio ao longo dessa jornada.

A ANP, pelo apoio financeiro.

Resumo

Um robô é um sistema heterogêneo composto de diversos elementos de hardware e software. Para que esse sistema funcione adequadamente é fundamental estabelecer uma arquitetura que modele suas diversas inter-relações. Assim, o objetivo do presente trabalho é projetar uma arquitetura de hardware e software que será utilizada num veículo aéreo não-tripulado (VANT).

Esse tipo de robô possui requisitos bem específicos que são determinantes no projeto de sua arquitetura. Por exemplo, o VANT proposto no projeto AEROPETRO utilizará fortemente técnicas de visão computacional, o que praticamente inviabiliza a utilização de sistemas operacionais de tempo real, pois, para atender as garantias de tempo os drivers das câmeras, precisam ser projetados para esse tipo de sistema, o que não acontece na prática. Por outro lado, controlar um sistema desse tipo sem obedecer certas restrições temporais ou sob risco de falhas de comunicação poderia levar a situações catastróficas. Por esses motivos é necessário propor uma arquitetura que se adeque as características dessa aplicação.

A arquitetura proposta nesse trabalho segue o modelo mestre-escravo e utiliza o protocolo USB como interface de comunicação. Todo o sistema se comunica através de um *backbone* USB que trabalha sobre um modelo de interrupção. Esse modo de comunicação provê ao sistema algumas características bem interessantes como: garantia na entrega dos pacotes de dados e especialmente que esses dados serão entregues dentro de uma janela de tempo previamente estabelecida. Assim, conforme comprovado pelos resultados obtidos é possível construir um robô utilizando um sistema operacional Linux como base e mesmo assim, ter certas garantias de tempo real (*soft real-time*).

Palavras-chave: Robô, arquitetura de hardware e software, mestre-escravo, USB.

Abstract

A robot is a heterogeneous system composed by different elements of hardware and software. To work properly, it is extremely important to establish an architecture that models the different inter-relations between the systems. Hence, the aid of this work is proposing a hardware and software architecture that works in an UAV (Unmanned Aerial Vehicle).

This kind of robot needs to attend very specific requests, that are really important in choosing an architecture. For example, the UAV in development in the project AEROPETRO, will use strongly computer vision techniques, making impractical the use of a real-time operational system. So, to attend the real-time requests the cameras' drivers need to be projected for this kind of application, what does not happen in practice. However, control this kind of system without any time restrictions or with the danger of communications loss could lead to a chaotic situation. For all this, it is important to propose an architecture that meet all the system demands.

The architecture proposed in this work follows the master-slave model and uses the USB as the communication interface. The whole system communicates through a USB backbone working under an interruption model. This kind of communication provides some interesting characteristics to the system, like: guaranties data package deliverance and the packages will be delivered in a fixed rate. Therefore, as can be proved by the results, it's possible to build a robot using a standard Linux as operational system, and even though, attend to certain deadlines.

Keywords: Robot, hardware and software architecture, master-slave, USB

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A construção de um sistema robótico é uma atividade desafiadora e fascinante. Diferente de outras ciências que estão restritas apenas a uma área do conhecimento a robótica é uma ciência ampla e complexa que engloba diversos campos. Pode-se até considerar a robótica como a arte da integração. Num robô sistemas elétricos, mecânicos e computacionais interagem com o fim de executar tarefas cada vez mais complexas.

Por esse motivo um aspecto fundamentalmente importante no projeto de qualquer sistema robótico é a definição de sua arquitetura de hardware e software. Essa arquitetura define como serão feitas as inter-conexões físicas e lógicas do robô.

A arquitetura de um sistema robótico deve ser projetada de modo a possibilitar ao robô a realização de suas tarefas, de acordo com certos requisitos. Assim uma arquitetura que poderia funcionar precisamente para um dado robô pode ser completamente inadequada para outro. Desse modo, antes de descrevermos a arquitetura proposta nesse trabalho é necessário compreender de modo um pouco mais claro o que é uma *arquitetura* de um sistema robótico.

1.1 Arquitetura de um Sistema Robótico

A arquitetura de um sistema robótico refere-se a maneira como as estruturas heterogêneas de hardware e software interagem para controlar o robô. Por exemplo, um sistema robótico com diversas placas embarcadas e diferentes unidades de software necessita de uma interface de comunicação e uma série de protocolos para que possa executar suas tarefas. Assim, quando o projetista começa a programar as interfaces para cada módulo e define como será feita a comunicação entre eles, na realidade ele está dando os primeiros passos na definição de uma arquitetura.

Vale salientar que o projeto de uma arquitetura não é uma tarefa trivial. O projetista deve muitas vezes equilibrar requisitos conflitantes. Note que o robô é um sistema complexo que integra diversos sensores e atuadores, que pode ter muitos graus de liberdade e que deve conciliar sistemas que são *hard real-time* com sistemas que não precisam atender aos *deadlines* de tempo real. Isso implica em diferentes necessidades de comunicação. Enquanto por um lado se lida com sistemas síncronos que demandam restrições temporais, por outro lado o robô também possui sistemas assíncronos, ou orientados a eventos, que não possuem tal restrição.

Desse modo, a tarefa do projetista da arquitetura pode ser resumida em algumas atividades básicas:

1. Analisar os requisitos do sistema.
2. Identificar os elementos síncronos e assíncronos.
3. Especificar um modelo de comunicação para o sistema. Ex: mestre-escravo.
4. Especificar a interface de comunicação do sistema. Ex: rede, barramento ou ponto a ponto.
5. Definir um protocolo de comunicação.
6. Programar módulos de software que permitam a administração dos recursos físicos do sistema.

A seguir apresentaremos a arquitetura proposta nesse trabalho e analisaremos alguns aspectos teóricos relevantes na sua proposição.

Capítulo 2

Arquitetura do Sistema

Como destacado na seção anterior o primeiro passo na definição de uma arquitetura de hardware e software é especificar os requisitos do sistema. A arquitetura proposta nesse trabalho tem por objetivo principal ser implantada no projeto AEROPETRO, em desenvolvimento pelo Laboratório de Robótica da UFRN. O projeto visa construir um VANT (veículo aéreo não-tripulado) que será utilizado em tarefas de inspeção e supervisão de instalações da indústria de petróleo e gás.

Os VANTs podem ser construídos sobre qualquer plataforma aérea como um avião, dirigível ou helicóptero. Atualmente, existem muitos projetos ao redor do mundo que utilizam helicópteros como plataforma base, graças a sua mobilidade e a possibilidade de vôo estacionário. No entanto, helicópteros comuns constituem modelos dinâmicos difíceis de serem controlados e com um alto número de variáveis de estado. Por esse motivo, costuma-se utilizar um tipo de helicóptero mais simples e relativamente mais fácil de controlar: o *Quadrotor*.

Um Quadrotor é um tipo de helicóptero, como ilustrado na Figura 2.1, que utiliza quatro acionamentos fixos, colocados em cada vértice de uma estrutura na forma de um quadrado com os rotores adjacentes girando em sentidos opostos, para equilibrar os momentos e produzir os movimentos desejados. Desse modo, o controle de vôo é possível ajustando-se a velocidade de cada um dos quatro motores.

Figura 2.1: Exemplo de um quadrotor comercial

Para cumprir seus objetivos o robô será equipado com quatro motores *brushless* e com um conjunto de sensores que incluem: câmera, sonar, IMU (unidade de medida inercial),

bússola digital e GPS. O robô contará ainda com um computador embarcado, que será a unidade responsável pelo processamento de imagens e também controle do sistema.

Com base na descrição acima se consegue extrair algumas características importantes. A primeira envolve as dimensões do robô. Como esperado, esse robô deve ser pequeno para que possa navegar em diferentes tipos de ambiente. Além disso, por utilizar motores elétricos no seu acionamento seu peso deve ser reduzido. De fato, a especificação feita pela equipe do projeto definiu que o robô deverá ocupar no máximo 1 m^2 de área e ter um peso total de 5 kg. Tendo em vista que boa parte da carga do helicóptero é devida a estrutura mecânica e as baterias, é primordial um projeto criterioso da eletrônica embarcada para que o sistema não ultrapasse os limites da especificação.

No que se refere à aplicação, podem-se destacar duas características marcantes. A primeira é que o sistema deve ser robusto e funcionar sobre requisitos de tempo real. Atrasos de comunicação e perdas de pacotes podem ser potencialmente perigosos nesse tipo de sistema. Outra característica é o uso massivo de visão computacional e processamento de imagens.

Essas duas características geram uma situação difícil de harmonizar na prática. Embora por um lado o processamento de imagens sugira a utilização de um computador embarcado com um sistema operacional convencional, a própria natureza do sistema sugere uma implementação que tenha comportamento de um sistema de tempo real, o que poderia ser alcançado sem necessariamente fazer uso de um computador embarcado.

A solução ótima seria utilizar um computador embarcado com um sistema operacional de tempo real. No entanto, embora essa solução seja muito interessante do ponto de vista teórico, na prática ela é extremamente difícil de ser aplicada. O motivo básico é que quando se utilizam módulos de tempo real os drivers presentes no sistema operacional convencional deixam de funcionar [Karim Yaghmour & Gerum 2008], pois, para garantir que o sistema funcione em tempo real, o sistema operacional deve ter certeza de que os drivers também funcionam em tempo real, o que não acontece na prática.

Portanto, o grande desafio deste trabalho é propor uma solução o mais próxima possível da solução ótima e que permita integrar o sistema de visão aos demais sensores e atuadores. Assim, podemos resumir os requisitos do sistema em:

- Peso total de todos os componentes de hardware restrito a 1,5 kg.
- Utilizar um computador embarcado.
- Possibilitar que a lei de controle seja executada no computador embarcado.

2.1 Arquitetura Proposta

Toda arquitetura é baseada num modelo abstrato que define como se dará o fluxo de comunicação no interior do sistema. A partir da descrição do sistema chegamos a conclusão que o *modelo mestre-escravo* é a solução ideal para a aplicação. No modelo mestre-escravo uma única unidade de maior poder computacional (o mestre) é responsável pelo processamento e administração do sistema. As demais unidades (os escravos) realizam apenas tarefas específicas requisitadas pelo mestre. No que tange a comunicação ela é feita apenas entre o mestre e um escravo, nunca de escravo para escravo.

Embora esse modelo pareça demasiadamente restritivo, ele é suficiente para o sistema proposto. No nosso caso, contamos com apenas uma unidade mestre (o computador embarcado), que é responsável pelo processamento de imagem e controle, já os escravos são unidades microcontroladas responsáveis pelo acionamento dos motores, leitura de sensores, etc.

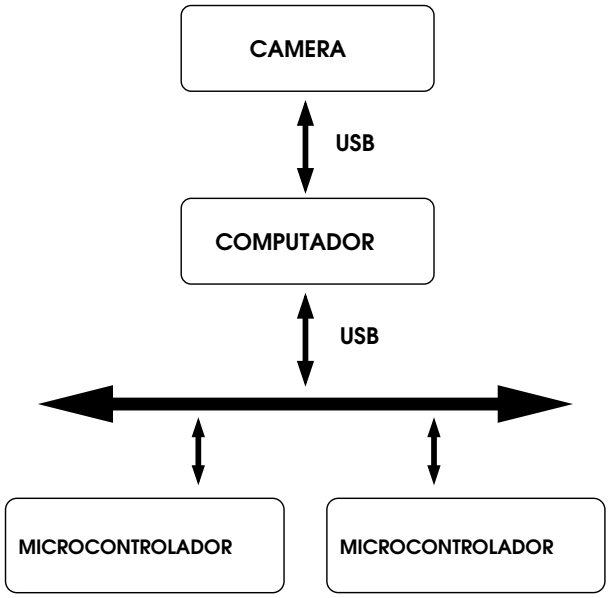

Figura 2.2: Arquitetura proposta para o robô AEROPETRO

Após definido o modelo de comunicação e que unidades de processamento utilizar, o próximo passo na definição da arquitetura é selecionar uma interface de comunicação. Atualmente existe uma infinidade de interfaces de comunicação disponíveis, que vão desde as interfaces de entrada/saída ponto-à-ponto até as redes.

No caso das interfaces ponto-à-ponto, a maioria dos computadores modernos trazem consigo um conjunto de portas USB. A USB foi desenvolvida com o objetivo de servir como uma interface padrão para entrada e saída de dados num computador, substituindo outras interfaces mais lentas como as portas paralelas e seriais. Dentre as diversas características da USB destacam-se:

- Protocolo de comunicação inerentemente mestre-escravo.
- Identificação automática de dispositivos. (*plug and play*)
- Drivers de classes de aplicação disponíveis na maioria dos sistemas operacionais.

Uma outra possibilidade seria utilizar uma rede de comunicação como interface de enlace, nesse sentido a classe de redes que mais se adequa a nossa aplicação são as redes industriais, especialmente a rede CAN. A CAN (Controller Area Network) é uma rede especialmente projetada para aplicações que necessitam de comunicação em tempo real, ela foi criada com o objetivo de servir como um barramento de comunicação para dispositivos eletrônicos dentro de ambientes ruidosos como o interior de um automóvel. Como principais características da CAN destacam-se:

- Rígido controle de erros.
- Garantia do recebimento de mensagens.
- Interface de comunicação simples. (Par trançado)

Note que ambas as tecnologias, embora muito diferentes uma da outra, atendem aos requisitos de comunicação da arquitetura proposta. No entanto, por tratar-se duma rede a CAN é uma solução mais indicada quando se trabalha com sistemas distribuídos ou multi-mestres. Além disso, a CAN não é uma interface nativa na maioria dos computadores, por isso, é necessário adquirir placas de aquisição ou de conversão para que o computador possa conectar-se a rede. A USB, por outro lado, é uma solução inerentemente mestre-escravo e prontamente disponível nos computadores convencionais. Além disso, a USB apresenta características interessantes como: diferentes modelos de comunicação e a possibilidade de organizar as funções do dispositivo num conjunto de interfaces independentes. Por esses motivos, optamos por utilizar a USB como interface de comunicação em nossa arquitetura.

De fato, o grande diferencial deste trabalho é a utilização da USB como *backbone* (ou interface de comunicação) do sistema. Com a utilização da USB em modo de interrupção foi possível atingir resultados interessantes. Como ficará claro no decorrer do trabalho, o robô, mesmo sendo implementado com um sistema operacional Linux comum, obteve resultados que permitem classificá-lo como um sistema *soft real-time*, desde que operando sobre certas condições.

Portanto, o foco desse trabalho é como conectar os microcontroladores com USB integrada (PIC 18F2550), com um computador rodando um sistema operacional Linux. Para atingir esse objetivo desenvolvemos uma biblioteca (USBRobot) que utiliza os drivers da classe HID, presentes no Linux, para criar uma interface de comunicação em espaço de usuário entre os programas implementados no mestre e os dispositivos escravos. Ou seja, o resultado deste trabalho é a criação de um *framework* que possibilita ao projetista de sistemas robóticos um ambiente integrado para o desenvolvimento de robôs com arquitetura mestre-escravo e *backbone* USB, sem a necessidade de utilizar um sistema operacional de tempo real.

Capítulo 3

Comunicação

Como discutido anteriormente, a comunicação desempenha um papel chave em qualquer arquitetura de hardware e software. Boa parte das características de uma arquitetura são dependentes de sua interface de comunicação. Nessa seção analisaremos o protocolo USB e suas características básicas.

3.1 Visão Geral do Padrão USB

A USB (*Universal Serial Bus*) é um meio de conexão entre um computador *host* e um certo número de periféricos. Ela foi inicialmente criada para substituir um conjunto de interfaces de comunicações mais lentas como — portas paralelas, seriais e de jogos — por uma única interface capaz de suportar todos esses dispositivos. Com o passar dos tempos, a USB deixou de ser usada apenas para conectar dispositivos de baixa velocidade, como os acima citados, e passou a ser a interface de comunicação com praticamente qualquer dispositivo que possa ser conectado a um computador. A última revisão da especificação adicionou conexões *high speed* com um limite de velocidade de 480 Mbps.

No entanto, diferente do que normalmente é expresso a USB não é um barramento. Na realidade, topologicamente a USB parece muito mais com uma árvore com diversas conexões ponto a ponto. Já do ponto de vista físico, a conexão se dá através de quatro fios (ground, power e dois de sinal) que conectam um *device* ou um *hub* ao computador de modo semelhante a um par trançado Ethernet.

Adicionalmente, pode-se considerar a controladora USB presente no *host* como sendo o coração da USB. A controladora tem o dever de periodicamente verificar se o *device* tem algum dado a enviar. Graças a sua topologia nenhum dispositivo USB pode enviar dados, a menos que seja requisitado pelo *host*. Essa configuração gera um sistema *plug-and-play* extremamente flexível, onde os dispositivos são automaticamente configurados pelo computador.

Do ponto de vista tecnológico, a USB é extremamente simples, não passando de uma aplicação mestre-escravo onde o *host* verifica os *devices* periodicamente em busca de dados. Apesar dessa aparente limitação, o barramento tem algumas características interessantes, como a possibilidade de um dispositivo requisitar certa largura de banda, por exemplo, para aplicações de transmissão de áudio. Outra característica importante é que a USB funciona apenas como um meio de comunicação, não determinando o formato dos

dados que trafegam sobre ela.

A especificação do protocolo USB define uma série de padrões que todo dispositivo de um determinado tipo pode seguir. Se o dispositivo seguir essas definições ele poderá utilizar um *driver* padrão prontamente disponível no sistema operacional. Esses diferentes padrões são denominados *classes* e são definições de dispositivos comuns que têm as mesmas necessidades de comunicação como teclados, mouses, joysticks, dispositivos de rede e modems. Já outros dispositivos que não se adequam a esses padrões necessitam de drivers específicos produzidos pelo fabricante.

Todas essas características tornam a USB um meio de comunicação prático e de baixo custo para a conexão de diversos periféricos, sem a necessidade de reiniciar o sistema, instalar placas e ficar perdido em meio a fios e conectores.

3.2 Protocolo de Comunicação

Conforme destacado na seção anterior a USB é uma interface de comunicação extremamente flexível. No entanto, toda essa flexibilidade no nível de usuário tem um custo — complexidade no nível de implementação. Felizmente, boa parte dessa complexidade é ocultada tanto pela controladora USB quanto pelo próprio sistema operacional.

Em outras palavras, com a USB acontece algo semelhante ao conceito de pilha de protocolos utilizado nas redes de computadores. Por exemplo, numa implementação em camadas como o modelo OSI ou TCP/IP as camadas superiores requisitam serviços das camadas inferiores e fornecem uma interface para as camadas superiores. De tal modo, que os detalhes de implementação estão ocultos em cada uma das camadas. Com a USB a idéia é a mesma. A controladora USB trata das questões referentes às camadas física e de enlace, enquanto fornece ao sistema operacional uma interface de acesso. O sistema operacional, por sua vez, recebe esses dados e disponibiliza uma interface para os *drivers* do dispositivo se comunicarem com o mesmo.

Portanto, para utilizar o protocolo adequadamente precisamos compreender algumas características gerais de seu funcionamento e em especial saber como se dá a comunicação com as camadas superiores.

3.2.1 Endpoints

A forma mais básica de comunicação sobre a USB é através de algo conhecido como *endpoint*. Um endpoint funciona como uma via de dados unidirecional, do computador para o dispositivo (OUT endpoint) ou do dispositivo para o computador (IN endpoint). Em nível de periférico pode-se encarar o endpoint como a ponte entre o hardware do dispositivo e o seu firmware.

Já do ponto de vista do computador os dados, são trocados através de *pipes*. Um *pipe* é um canal lógico que interconecta um *host* a um endpoint. Cada *pipe* possui ainda uma série de parâmetros que precisam ser devidamente ajustado, como largura de banda, direção do fluxo de dados, tamanho máximo do buffer/dados e tipo de transmissão utilizada. Por exemplo, o default pipe é composto pelo endpoint zero IN e o endpoint zero OUT com transferência de dados do tipo *Control*.

A USB descreve basicamente dois tipos de pipes:

Stream Pipes: esse tipo de pipe não possui um formato de dados definido, funciona apenas como uma via. Os dados são enviados seqüencialmente numa direção predefinida IN ou OUT e podem ser controlados tanto pelo *host* como pelo *device*.

Message Pipes: esse tipo de pipe tem um formato de dados bem definido. São controlados pelo host. E os dados são transferidos no sentido informado pela requisição. Portanto, são bi-direcionais.

Resumindo, do ponto de vista do computador a comunicação é feita através de pipes, enquanto a comunicação entre a controladora e o dispositivo utiliza o conceito de endpoints.

Tipos de Endpoints

Um endpoint pode ser de quatro tipos diferentes. Cada um desses tipos descreve como os dados são transmitidos:

CONTROL: O control endpoint é utilizado como meio de acesso para diferentes propósitos dentro do protocolo. Eles são comumente utilizados para configurar um dispositivo, retornar informações de status, enviar comandos para o dispositivo ou enviar informações sobre o mesmo. Esses endpoints são normalmente pequenos, alguns poucos bytes, no máximo 64 bytes, e todo dispositivo USB deve ter o endpoint zero IN e OUT, utilizados para tarefas administrativas. Esse tipo de transferência tem banda garantida pelo protocolo em qualquer situação.

INTERRUPT: Esse endpoint transfere pequenos pacotes de dados a intervalos bem determinados sempre que o *host* USB requisita. É normalmente utilizado por dispositivos de interface como teclados, mouses e joysticks. Além dessas, aplicações o método de interrupção também é utilizado para controlar dispositivos, mas sem transferir grandes quantidades de dados. Esse tipo de transferência tem uma banda reservada diretamente pelo protocolo. Na USB 2.0 *full-speed* esse endpoint tem uma banda reservada de 64 Kbps.

BULK: Os dispositivos que usam esse tipo de transferência precisam transferir uma grande quantidade de dados sem perda de informações. Esse tipo de transmissão não tem banda reservada pelo protocolo. Quando uma requisição de transmissão é feita a controladora utiliza a largura de banda disponível para a transferência. Desse modo, sua largura de banda é dependente da carga do sistema. A transferência BULK é normalmente utilizada por *flash drivers* e por impressoras.

ISOCHRONOUS: Esse tipo de endpoint transmite grande quantidade de dados com latência constante, no entanto sem garantia de integridade. Ele é utilizado por dispositivos que podem lidar com perda de dados, mas que necessitam garantir um fluxo contínuo das informações. Essa característica de transmissão é util especialmente na transferência de áudio e vídeo.

Na aplicação proposta nesse trabalho a USB é utilizada como meio de comunicação entre os microcontroladores e o computador. Note que esse tipo de comunicação

tem características perfeitamente compatíveis com as dos endpoint do tipo INTERRUPT. Precisam-se transferir dados que demandam pouca largura de banda como leitura de sensores e referências para atuadores. Adicionalmente como o robô é um sistema de tempo real deseja-se que esses dados cheguem num intervalo de tempo bem determinado. Portanto, a utilização do modo de transferência baseado em interrupções dá à arquitetura a possibilidade de responder em tempo real graças às características do próprio protocolo.

3.2.2 Interfaces

Na arquitetura de comunicação da USB os endpoints não são diretamente acessíveis ao *host*, eles estão contidos em estruturas lógicas chamadas *interfaces*. Uma interface USB é capaz de manipular apenas um tipo de conexão lógica, por exemplo, teclado, mouse ou uma impressora. Um dispositivo USB pode ainda ter múltiplas interfaces, por exemplo, uma PABX tem uma interface para controle dos botões do tipo INTERUPT, uma interface para impressão com endpoint do tipo BULK, e uma interface de áudio com endpoint do tipo ISOCHRONOUS para o telefone. Como as interfaces USB representam funcionalidades básicas, cada driver USB controla uma interface; desse modo, para o exemplo do PABX o computador utilizará três drivers diferentes.

Essa funcionalidade só é possível porque cada periférico é registrado pela controladora USB com um endereço de dispositivo e com um conjunto de endereços de interfaces. Essa característica permite localizar a interface dentro da árvore de dispositivos. A figura a seguir ilustra esse fato.

Um detalhe que talvez tenha passado despercebido é como a controladora registra o dispositivo dentro da árvore. Para fazer isso o dispositivo precisa informar à controladora que está conectado e em seguida informar suas características. Esse processo é chamado de *enumeração* e é uma parte importante do protocolo. Durante a enumeração, o dispositivo é devidamente registrado e, a partir de então, a controladora tem condições de verificar se pode atender as necessidades de comunicação requisitadas pelo dispositivo. Essas informações enviadas pelos dispositivos são chamadas de *descritores*.

3.2.3 Configuration

Num nível hierárquico acima da interface encontram-se as *configurações*. Essas estruturas descrevem algumas características gerais do dispositivo como: potência consumida, se o dispositivo é auto-alimentado ou alimentado pelo barramento e o número de interfaces presentes naquela configuração. Quando o dispositivo entra no processo de enumeração, ele envia o descritor do dispositivo e a partir dessa informação a controladora decide que configuração usar. Vale salientar que um dispositivo pode ter mais de uma configuração, no entanto isso é pouco usado na prática. Na maioria das vezes um dispositivo USB tem: um descritor de dispositivo, um descritor de configuração, um conjunto de descritores de interface, e por fim cada interface tem um ou mais descritores de endpoint.

A figura a seguir dá uma visão geral do conjunto de descritores enviados durante o processo de enumeração.

Para maiores informações sobre a USB queira considerar [usb.org 2000]

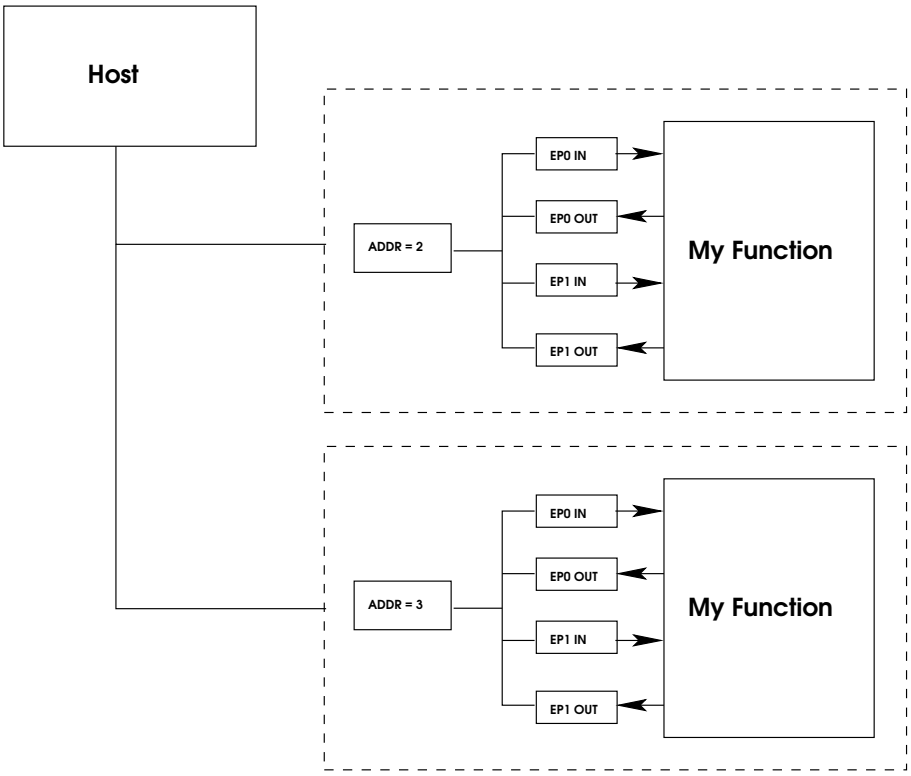

Figura 3.1: Arquitetura de comunicação USB.

3.3 HID - Human Interface Devices

Até o momento foram destacados alguns aspectos gerais do funcionamento da USB como: a arquitetura do protocolo e o modelo de comunicação entre o *host* e o *device*. No entanto, para que os dados possam ser acessados em nível de aplicação é necessário definir uma camada de interface entre os dados provenientes da controladora e o sistema operacional. Essa interface é de responsabilidade do *driver*.

O driver é um programa de baixo nível que conhece as excentricidades do dispositivo e que é capaz de comunicar-se com o mesmo. Ele também é responsável por implementar uma série de funções (interface) que possibilitem ao sistema operacional acessar aquele dispositivo. Assim todo dispositivo conectado a USB, após passar pelo processo de enumeração, deve ser associado a um driver para enfim ser acessível ao sistema.

Desse modo, quando projetamos um dispositivo USB adicionalmente precisamos providenciar um driver, para que o SO possa se comunicar com o dispositivo. No entanto, implementar um driver não é uma tarefa fácil. Pois, o programador precisa estar familiarizado com as estruturas, funções e excentricidades do kernel do sistema operacional. Felizmente, a USB trabalha com um conceito extremamente útil, o conceito de classes de dispositivos.

Como destacado anteriormente, existem dispositivos que possuem características e necessidades de comunicação semelhantes. Esses dispositivos podem ser agrupados em classes e ao invés de implementar um driver para cada novo dispositivo basta ter um

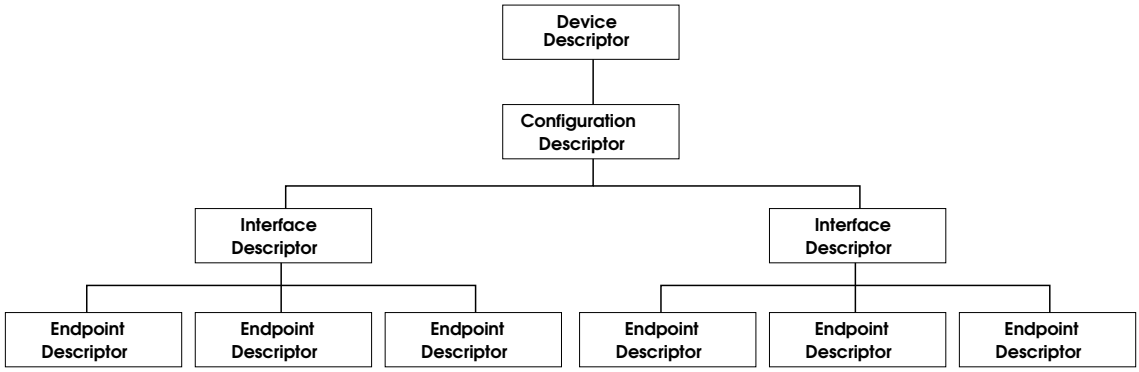

Figura 3.2: Árvore de descritores.

driver da classe. Existem três grandes classes de dispositivos comumente associados a USB: dispositivos de interface humana (HID), dispositivos de armazenamento em massa (MSD) e dispositivos de comunicação (CDC). Cada uma dessas classes já possui um driver implementado na grande maioria dos sistema operacionais.

Portanto, se adequarmos o firmware de nosso dispositivo para ser compatível com uma dessas classes não haverá necessidade de implementar um driver.

3.3.1 Visão Geral

Após analisar os requisitos de comunicação demandados por nossa aplicação chegamos a conclusão que podíamos adequar nossos microcontroladores a classe HID. Essa classe é normalmente utilizada para conectar dispositivos que interagem diretamente com o ser humano, através de entrada e saída de dados como: mouses, teclados e joysticks. Embora o nome dê a impressão que apenas dispositivos que interagem com o homem façam parte dessa classe. Outros dispositivos com necessidades de comunicação semelhante também fazem parte de sua especificação.

No que se refere a comunicação, os dispositivos HID possuem pelo menos dois endpoints: um endpoint de controle (default) e um interrupt endpoint de entrada. Opcionalmente um interrupt endpoint de saída também pode ser adicionado à mesma interface. O endpoint default é utilizado para enviar as mensagens administrativas típicas da USB e as requisições HID como: pedido ou notificação de envio. Já os interrupt endpoints são utilizados para o envio de dados.

Para compreendermos como se dá esse modelo de comunicação considere o exemplo de um teclado USB. A cada *n* milissegundos (valor estipulado no descritor do interrupt endpoint) o *host* envia para o *device* uma mensagem perguntando se o mesmo possui dados a enviar. Se o *device* possuir dados, no caso do teclado uma ou mais teclas que foram pressionadas, ele deverá confirmar a disponibilidade de dados e enviá-los através do interrupt endpoint de entrada. Os dados são então recebidos pelo driver HID que repassará os dados para interface do sistema operacional.

3.3.2 Descritor HID

Um ponto importante que precisa ser compreendido é que o driver HID é um driver padrão que deve funcionar para uma infinidade de dispositivos. Para alcançar esse objetivo o HID usa uma idéia interessante: o dispositivo deve descrever o formato de seus dados e informar para que o dado serve. Essas informações são repassadas para o driver através de um tipo especial de descritor, chamado de *HID descriptor*.

Diferente dos descritores estudados anteriormente o descritor HID não é um conjunto de dados guardados na ROM do microcontrolador. Na realidade o descritor HID é um conjunto de mensagens que são enviados para o driver após a fase de enumeração. Essas mensagens são interpretadas pelo driver através de um *parser*, que ao final do processo, sabe o formato e o significado das mensagens enviadas pelo dispositivo.

Vale salientar que os descritores HID permitem descrever um número infinito de mensagens, o que provê grande flexibilidade a essa classe. Por exemplo, atualmente simuladores de avião, equipamentos de realidade virtual, instrumentos hospitalares e uma grande quantidade de outros dispositivos comunicam-se através da classe HID.

Mas, toda essa flexibilidade tem um custo. Além da complexidade da implementação do *parser*, a própria escrita do descritor pode constituir-se num processo exaustivo. Para maiores informações a respeito da classe HID queira considerar [Forum 2001].

Capítulo 4

Desenvolvimento

Como destacado anteriormente, uma arquitetura de hardware e software tem por objetivo descrever como se dará a comunicação entre as diversas partes de um sistema robótico. A arquitetura proposta nesse trabalho utiliza duas unidades básicas: os microcontroladores e o computador.

Contudo, a comunicação entre as diferentes unidades computacionais não ocorre naturalmente. Antes, é preciso estabelecer um protocolo entre eles. Ou seja, é necessário programar os microcontroladores e o computador de tal forma que a comunicação entre eles seja possível.

4.1 Microcontrolador

Considerando o modelo mestre-escravo, utilizado nesse projeto, os microcontroladores desempenham o papel de escravos. Eles são responsáveis por todas as tarefas de baixo nível associadas ao robô como: interfacear sensores, gerar sinais para acionamento de atuadores etc.

Na arquitetura proposta nesse trabalho utilizou-se o microcontrolador PIC 18F2550 [Microchip 2007], fabricado pela Microchip Technology. Esse microcontrolador tem implementado um hardware capaz de comunicar-se diretamente com a USB, sem a utilização de transciever externo. Além disso, ele possui uma série de outros recursos que o torna altamente flexível e ideal para aplicações dedicadas. Dentre eles podemos destacar:

- Diversos pinos de E/S digital (24 pinos).
- Conversores AD (10 canais).
- Quatro Timers.
- Interrupções com prioridade ajustável (19 tipos).
- Dois módulos CCP (Capture/Compare/PWM).
- Modo de operação economizador de energia.
- Comunicação serial: USART, SPI e I2C.

4.1.1 Framework

Para propiciar que seus microcontroladores se comunicassem através da USB a Microchip desenvolveu um framework. Esse conta com uma API que implementa a pilha de proto-

colos utilizada pela USB, bem como uma API para cada uma das principais classes de dispositivos. Esse framework ainda disponibiliza uma série de códigos fontes e exemplos compilados de diferentes aplicações utilizando as funcionalidades USB.

Embora o framework da Microchip forneça uma fonte inestimável de ajuda no desenvolvimento de aplicações embarcadas. Vale salientar que essa ferramenta ainda está em desenvolvimento e que para utilizá-la efetivamente em projetos reais o programador deve adaptar seus códigos a sua realidade.

Por exemplo, no caso do PIC 18F2550 nenhum dos códigos fontes presentes no framework fora projetado para esse tipo de microcontrolador. Assim, antes de implementar qualquer firmware de teste, um bom tempo foi despendido para adaptar os códigos do framework para o microcontrolador utilizado no projeto.

Embora não tenhamos o objetivo de explicar detalhadamente o funcionamento do framework, compreender algumas de suas características é de fundamental importância. Um primeiro aspecto é que todas as funções da API podem ser divididas em dois grandes grupos:

1. Funções da pilha de protocolo
   - Funções Administrativas
   - Funções de Callback
2. Funções da Classe de Dispositivo

As funções administrativas, como o próprio nome indica, são responsáveis por cuidar dos detalhes do protocolo. Essas funções são responsáveis pelas trocas de mensagens durante o processo de enumeração, pela definição do estado de operação, administração dos endpoints etc.

Todas essas funções são praticamente ocultas ao desenvolvedor do software embarcado. No entanto, uma dessas funções é de importância vital para o bom funcionamento do sistema: a função *USBDeviceTasks*. Essa função é a principal do ponto de vista do dispositivo. Ela é responsável por implementar a máquina de estados da USB, e por isso, deve ser chamada periodicamente para receber os pacotes através da pilha de protocolos.

Um detalhe interessante é que essa função deve ser chamada a cada 100us durante o processo de enumeração. Note que esse tempo é extremamente pequeno para a maioria das aplicações práticas. No entanto, essa restrição é relaxada após a enumeração. Na realidade, após a enumeração essa função deve ser chamada tão rapidamente quanto os envio de dados para o computador. Vale salientar que essa função pode ser chamada explicitamente no main, funcionando em modo de polling ou pode-se utilizar o modo de interrupção. A vantagem de utilizar o modo de interrupção é que o programador não precisará se preocupar com essa função, pois sempre que houver a necessidade de atualizar a máquina de estados da USB uma interrupção será gerada e a *USBDeviceTasks* será automaticamente chamada. E para efeitos práticos essa função leva em torno de 50 ciclos de máquina para ser executada.

Ainda no que se refere às funções da pilha de protocolo, existem as funções de callback. Essas funções servem para o tratamento de situações específicas dentro do protocolo. Por exemplo, a função *USBCBSuspend* é utilizada para executar tarefas impostas pelo programador quando o dispositivo entrar em modo suspenso pela USB (o que indica

uma inatividade durante 3ms). Vale salientar que essas funções em sua maioria estão apenas declaradas no código, sendo de responsabilidade do programador sua implementação.

Já as funções de classe de dispositivo são funções de nível superior dependentes das funções da pilha USB. Essas funções são responsáveis pela comunicação do firmware com o driver da classe. No caso da classe HID utilizada nesse projeto a interface com o driver se dá através de cinco funções básicas:

USBCheckHIDRequest: utilizada pela função de callback USBCBCheckOtherReq para informar que no endpoint default trafegam, além de informações típicas da pilha, funções pertinentes a classe de dispositivo.

HIDTxPacket: utilizado para enviar dados através de um interrupt endpoint especificado.

HIDRxPacket: utilizado para receber dados através de um interrupt endpoint especificado.

HIDTxHandleBusy: retorna se o microcontrolador está com o controle do endpoint. No caso da transmissão, isso indica que não existe nenhuma transmissão pendente.

HIDRxHandleBusy: retorna se o microcontrolador está com o controle do endpoint. No caso da recepção, isso significa que há dados disponíveis no buffer de leitura.

Novamente, vale salientar que o framework USB da Microchip ainda está em desenvolvimento. Desse modo, algumas funcionalidades ainda poderão ser alteradas ou adicionadas. Mas, de modo geral, ele já oferece atualmente um conjunto de funções suficientes para o desenvolvimento de uma infinidade de aplicações utilizando a USB. Para maiores detalhes sobre o framework queira analisar: *www.microchip.com* e informações pertinentes no documento USB Device Library Help.

4.2 Firmware

Ao se deparar pela primeira vez com o emaranhado de funções e arquivos utilizados pelo framework da Microchip o programador pode facilmente sentir-se intimidado. No entanto, com um pouco de paciência é possível compreender a estrutura lógica do mesmo e fazer algumas adaptações que facilitam o desenvolvimento de aplicações futuras.

Existem alguns arquivos do framework que são utilizados com freqüência e que devem ser devidamente editados pelo programador. Entre eles pode-se destacar:

- **HardwareProfile.h:** esse arquivo contém um conjunto de definições específicas do hardware do usuário que serão utilizados no main. Por exemplo, mnemônicos de periféricos como LEDs, MOTORES e etc.
- **usb_config.h:** contém uma série de definições de configuração do protocolo. É nesse arquivo que o programador define se utilizará o USBDeviceTasks em modo de polling ou interrupção; define como será o sistema de bufferização usado pelo hardware; as configurações de velocidade; configurações de descritores etc.
- **usb_desciptors.c:** esse arquivo contém a árvore de descritores do dispositivo. Descritores de dispositivo, configuração, interface, endpoint e de classe.

- **main.c:** arquivo principal.

O programador deve atentar que as configurações do microcontrolador sejam feitas adequadamente. Para isso existe uma função chamada InitializeSystem() que é responsável pela configuração bruta da USB. Qualquer outro tipo de inicialização deve ser feita através da função UserInit(), implementada pelo programador.

Outro aspecto importante é como o programador deve organizar seu código para tirar o máximo de proveito do framework. Por exemplo, uma análise criteriosa dos diversos arquivos main presentes no framework ilustra uma sub-divisão desse código em pelo menos quatro seções distintas que devem ser devidamente programadas:

- Tratamento de interrupção: Os microcontroladores PIC permitem que o programador utilize funções para o tratamento de interrupções. Especificamente o PIC 18F2550 permite classificar as interrupções por classe: interrupções de alta e de baixa prioridade.
- ProcessIO: Essa função permite ao programador tratar diretamente as mensagens recebidas através da USB. Isso possibilita separar nitidamente o código responsável pela comunicação do restante do programa.
- Funções de Callback: Como destacado anteriormente existem uma série de situações dentro do protocolo USB que podem ser importantes para aplicação. Essas situações podem ser devidamente tratadas utilizando as funções de callback.
- Demais funções: são funções auxiliares definidas pelo usuário e dependentes da aplicação.

4.2.1 Protocolo de Aplicação

Até o momento foi compreendido que toda tarefa de comunicação entre o dispositivo e o computador, do ponto de vista de firmware, é devidamente tratada pelas funções da API da Microchip. A tarefa do programador no que tange a comunicação resume-se a prover a descrição dos dados que trafegarão pela USB.

Essa descrição é feita através do conjunto de descritores citados no capítulo três. Assim, antes de implementar qualquer lógica referente a aplicação o projetista deve especificar esses descritores criteriosamente. Após estudarmos diversas configurações possíveis desses descritores selecionamos um modelo básico que pode ser eficientemente usado em aplicações de robótica.

Esse modelo utiliza uma única interface HID, e procura criar duas vias de dados independentes uma para entrada e outra para saída. Uma rápida leitura de seu descritor HID deixa claro que o sistema trabalhará com dois endpoints, um de entrada e outro de saída, ambos de 64 bytes. Cada endpoint é codificado para trabalhar com unidades de 8-bits, sem codificação de sinal (valores de 0-255) e indexados de 0-63.

```
//Descritor Específico da Classe HID
ROM struct{BYTE report[HID_RPT01_SIZE];}hid_rpt01={
{
    0x06, 0x00, 0xFF, // Usage Page = 0xFFFF (Vendor Defined)
```

```
    0x09, 0x01,       // Usage
    0xA1, 0x01,       // Collection (Application)
    0x19, 0x01,       //      Usage Minimum (Vendor Usage = 0)
    0x29, 0x40,       //      Usage Maximum (Vendor Usage = 64)
    0x15, 0x00,       //      Logical Minimum (Vendor Usage = 0)
    0x26, 0xFF, 0x00, //      Logical Maximum (Vendor Usage = 255)
    0x75, 0x08,       //      Report Size 8 bits (one full byte)
    0x95, 0x40,       //      Report Count 64 bytes in a full report.
    0x81, 0x02,       //      Input (Data, Var, Abs)
    0x19, 0x01,       //      Usage Minimum (Vendor Usage = 0)
    0x29, 0x40,       //      Usage Maximum (Vendor Usage = 64)
    0x91, 0x02,       //      Output (Data, Var, Ads)
    0xC0}
};                    // End Collection
```

Note que a descrição dos dados é extremamente genérica, ou seja, o protocolo não especifica em detalhes a forma dos dados que trafegarão sobre a USB. Essa característica pode então ser utilizada para gerar um protocolo de aplicação baseado em troca de mensagens. Onde o desenvolvedor do software embarcado e o programador de alto nível especificam um conjunto de mensagens que serão trocadas entre o microcontrolador e o computador. Embora essa abordagem seja extremamente simples, na prática ela apresenta bons resultados. Tendo como principal vantagem a criação de um protocolo facilmente extensível e de fácil manutenção.

Por outro lado, o modelo de troca de mensagem tem como principal desvantagem não disponibilizar ao computador meios de descobrir como se comunicar com o dispositivo. Em outras palavras, o programador de alto-nível precisa conhecer o protocolo de comunicação antes de se comunicar com dispositivo. Assim, pode-se considerar esse modelo como sendo orientado pelo dispositivo. Por exemplo, se o programador de alto-nível desejar ler os dados de um determinado sensor ele precisará saber previamente com que microcontrolador se comunicar e enviar um conjunto de mensagens apropriadas, para enfim receber os dados do sensor.

4.2.2 Sistema de Testes

O firmware de testes implementado tenta se aproximar o máximo possível de sua aplicação prática num VANT. Onde um grande conjunto de sensores e atuadores estará conectado ao mesmo microcontrolador. No sistema de testes, o microcontrolador é responsável pela interface de quatro motores brushless, uma bússola digital e um sonar com o computador.

Para possibilitar essa comunicação foi especificado um pequeno conjunto de mensagens, conforme o apêndice A. Além disso, procuramos explorar o conceito de interfaces disponibilizado no padrão USB. Para isso, fizemos algumas adaptações no framework Microchip para que fosse possível trabalhar com várias interfaces HID diferentes.

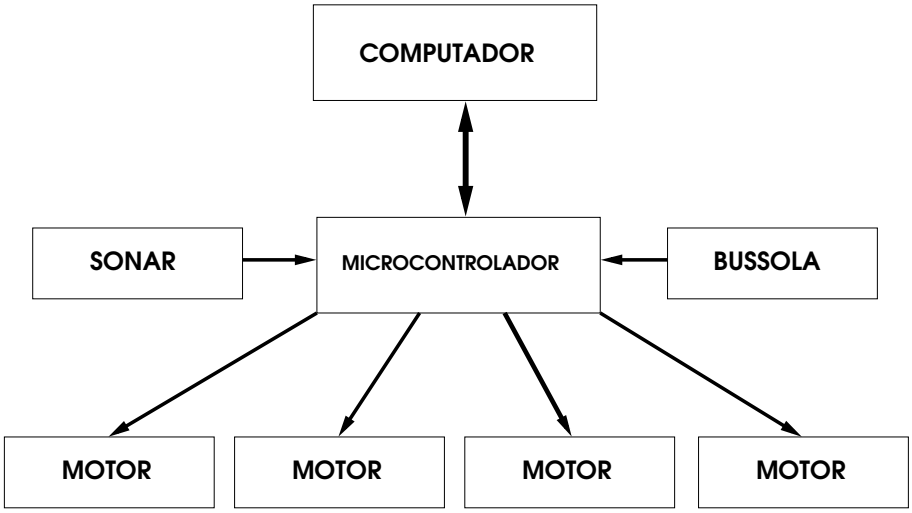

Figura 4.1: Sistema de Testes

4.3 Software de Alto-nível

Nessa seção será descrito o conjunto de softwares de alto-nível responsável pelo enlace entre o dispositivo USB e o computador. O objetivo dessa camada de software é tornar os recursos disponibilizados no microcontrolador visíveis ao programador de alto-nível. Essa abstração é feita através de um conjunto de classes que permitem ao usuário interagir diretamente com o microcontrolador, sem conhecer os detalhes de implementação do protocolo de enlace.

4.3.1 Sistema Operacional

Para acessar qualquer recurso de hardware um programa em nível de usuário deve fazer requisições ou chamadas ao sistema operacional. O sistema operacional então se comunicará com o dispositivo, fazendo uma ponte entre o hardware e o programa. Portanto, para compreender como se dará a comunicação através da USB é fundamental compreender como o sistema operacional processa esses dados.

É evidente que cada sistema operacional tratará as operações de I/O de forma diferenciada. Assim, nos concentraremos em compreender como essas operações são tratadas no Linux, sistema operacional utilizado nesse desenvolvimento. Vale salientar ainda, que a biblioteca desenvolvida nesse trabalho funciona apenas no Linux. Mais especificamente nas versões do kernel a partir da 2.6.24

4.3.2 Subsistema USB

Todo sistema operacional é divido em diversos subsistemas responsáveis por tarefas específicas. Entre esses subsistemas, o de I/O (*input/output*) é especialmente importante

quando se trabalha com interfaceamento de hardware. O objetivo desse subsistema é tornar visível aos usuários os elementos de hardware conectados ao computador.

Essa tarefa é bastante desafiadora do ponto de vista computacional. De um lado, o subsistema de I/O deve lidar com hardwares com características de comunicação muito diferentes, por exemplo, um modem e uma impressora. Por outro, precisa fornecer uma série de interfaces padronizadas para os usuários do sistema.

Para lidar com esses desafios o Linux organiza seu subsistema de I/O em duas partes nítidas. Os drivers, que são programas básicos responsáveis pelo interfaceamento com o hardware. Esses programas conhecem as necessidades e excentricidades da comunicação de cada dispositivo e são essenciais para a comunicação entre o PC e o dispositivo. Sem o driver adequado, fornecido pelo fabricante, é impossível comunicar-se com o dispositivo.

Outra parte importante do subsistema de I/O são as camadas lógicas de interface. Essas camadas intermediárias estão localizadas logicamente acima dos drivers e são responsáveis por proverem serviços ao sistema operacional através do drivers. Essas interfaces incluem dispositivos com características de comunicação parecida. Por exemplo, um leitor de DVD, tem características de comunicação idênticas a um HD e portanto ambos são tratados como dispositivos de bloco pelo Linux. Enquanto um teclado e um mouse são tratados como dispositivos de caracteres.

A USB, como era de se esperar, deve ser tratada pelo sistema operacional através do subsistema de I/O. No entanto, graças a suas características particulares, a USB precisa de um tratamento especial no subsistema de I/O. A figura 4.2 ilustra a arquitetura do subsistema USB do Linux. Para maiores informações sobre o subsistema USB e como são implementados os *device* drivers considere [Jonathan Corbet & Kroah-Hartman 2005].

Essa figura deixa claro que a controladora USB é diretamente responsável pelo enlace entre o device e o host. Em seguida uma camada de software intermediária, a USB-Core, é responsável por prover uma interface entre a controladora e o driver do dispositivo. Esse driver pode ser tanto um driver de classe (HID, CDC ou MSD), como um driver proprietário. Em seguida o driver deve comunicar-se com as camadas de interface, exatamente como descrito anteriormente.

4.3.3 Drivers

Como destacado até o momento os drivers desempenham um papel vital no interfaceamento entre o dispositivo e o sistema operacional. Quando um dispositivo USB é conectado ao computador automaticamente o Linux associa uma série de drivers aquele dispositivo. Se o dispositivo implementar as funções da classe HID, dois drivers específicos serão associados ao dispositivo: o HIDDEV e o HIDRAW. Esses drivers geram automaticamente um *file descriptor* responsável por representar o dispositivo dentro do sistema operacional.

Esse *file descriptor* é um tipo de descritor de arquivo especial, pois representa um dispositivo não um ponteiro para um conjunto de dados armazenado na unidade física. No entanto, assim como um arquivo convencional, é possível executar uma série de operações sobre esse descritor. Isso significa que a interface do sistema operacional responsável pelas transações com um *file descriptor* que representa um dispositivo e um arquivo con-

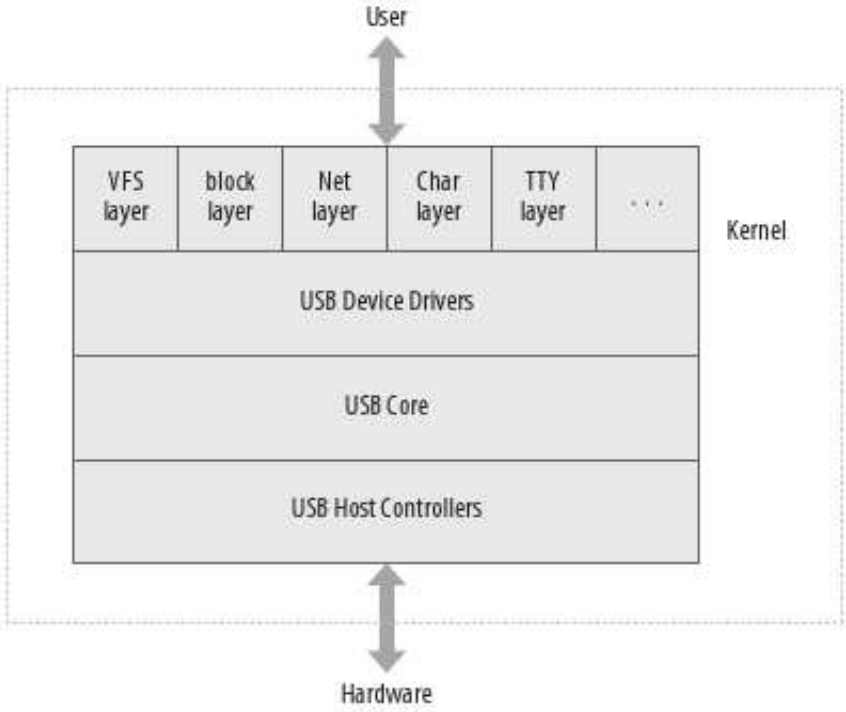

Figura 4.2: Subsistema USB no Linux

vencional são muito parecidas.

Por exemplo, todo driver presente no sistema operacional deve fornecer à camada de aplicação um conjunto de interfaces que possibilitarão a comunicação com o dispositivo. Embora essa interface possibilite uma infinidade de funções, algumas são prioritárias. Essas funções incluem: Open, Close, Read, Write e opcionalmente uma função genérica chamada Ioctl, responsável por configurar o dispositivo dentre outras funcionalidades. Com essas funções devidamente implementadas um programa a nível de usuário tem totais condições de se comunicar com um dispositivo de hardware. Note ainda que com exceção da Ioctl as demais funções possuem a mesma semântica de suas correspondentes em arquivos convencionais.

Um aspecto importante é que ambos os drivers (HIDDEV e HIDRAW) implementam, como era de esperar, as mesmas funcionalidades. E ambos funcionam comunicando-se com um driver de baixo nível chamado hid-core. Esse último é responsável pela troca de mensagens com o dispositivo e implementa o parser capaz de traduzir os descritores HID. Assim, tanto o HIDDEV como o HIDRAW funcionam na verdade como uma interface entre o programa a nível de aplicação e o hid-core.

Uma pergunta importante é: por que o sistema operacional associa esses dois drivers ao hid-core? O principal motivo é que o desenvolvimento de soluções HID ainda está em andamento no Linux. Esses drivers foram criados em diferentes épocas e com diferentes objetivos. O HIDDEV, por exemplo, foi criado com o objetivo de criar uma interface HID

padrão. No entanto, esse driver utiliza amplamente conceitos e nomeclaturas baseados no padrão HID, que vale salientar, é algo extremamente complexo. Isso acabou tornando o driver excessivamente complexo e com poucas aplicações práticas. Junte a esse fato a pouca documentação disponível sobre o mesmo, e você encontrará o motivo porque é tão complexo o desenvolvimento de soluções HID para o Linux.

O HIDRAW, por outro lado, surgiu como uma "solução" para os problemas HIDDEV, e serve como um modo unificado de se comunicar com *raw devices* através da USB ou Bluetooth. Sua filosofia é lançar toda a complexidade da interpretação das mensagens HID para o nível de aplicação. Assim o programador de alto-nível é que deve interpretar a semântica das mensagens HID e não o driver. Uma observação importante é que o HIDRAW nasceu para substituir o HIDDEV, no entanto, como esse, ainda tem alguns problemas de implementação, em especial na interface de escrita. Ainda levará certo tempo até tornar-se o padrão do Linux.

4.3.4 Biblioteca

Na seção anterior foram destacas as interfaces do Linux utilizadas para comunicação com dispositivos HID. É fácil perceber que essas soluções individualmente não permitiriam a construção de uma biblioteca de comunicação eficiente. Desse modo, a idéia básica que utilizamos para desenvolver a biblioteca USBRobot foi aproveitar o que cada uma dessas interfaces possui de melhor.

Inicialmente, foi observado que o driver HIDDEV possibilita um meio eficiente de escrita, ou seja, transmissão de dados para o dispositivo. Após um estudo exaustivo de seu código fonte foi possível estabelecer a comunicação com o dispositivo. Essa comunicação é feita através de um conjunto de estruturas características do driver e um conjunto de chamadas IOCTL. Uma observação interessante é que nas versões anteriores do driver era necessário realizar uma chamada de sistema operacional para cada byte a ser enviado, o que é extremamente custoso. Nas versões mais recentes foi possível enviar dados utilizando apenas duas chamadas de sistema operacional: uma para montagem do pacote e outra para enviar o pacote. O HIDDEV também conta com uma interface eficiente para comunicação não bloqueante, onde suas estruturas internas são atualizadas automaticamente em nível de kernel.

No entanto, percebeu-se que o HIDDEV era extremamente ineficiente na leitura bloqueante. Por algum motivo, sua implementação fazia que muitos dados fossem perdidos, e portanto, não visíveis a nível de aplicação. Por outro lado, a interface HIDRAW, como foi projetada em especial para leitura de dispositivos, é extremamente eficiente nesse sentido. Por esse motivo, as funções de leitura bloqueante foram implementadas utilizando o HIDRAW como base. A seguir encontra-se uma lista das principais funções implementadas pela USBRobot.

USBrobot Construtor da classe.
˜ USBrobot Destrutor da classe
openDevice Abrir o dispositivo.
closeDevice Fechar o dispositivo
writeDevice Escrita não bloqueante no dispositivo.

readDevice Leitura não bloqueante do dispositivo.
readBlocking Leitura bloqueante do dispositivo.
setDeviceId Definir id de produto a ser verificada pela classe.
getDeviceId Retornar id do produto.
setVendorId Definir id do vendedor a ser verificada pela classe.
getVendorId Retornar id do vendedor.
getInSize Tamanho do endpoint de entrada.
getOutSize Tamanho do endpoint de saída.

Além de possibilitar um acesso mais simples e intuitivo aos dispositivos USB, a classe USBRobot conta ainda com um esquema de tratamento e recuperação de erros. O objetivo do modelo de tratamento de exceção implementado na classe é possibilitar ao programador de alto nível total controle sobre o que está acontecendo na comunicação, no entanto, sem sobrecarregá-lo com muitos detalhes de implementação. Para isso, foi projetado um conjunto de classes que permitem analisar os erros com diferentes níveis de detalhamento. Por exemplo, o usuário pode utilizar a classe USBException para capturar todas as exceções que ocorram na comunicação. Ou poderia preferir tratar um tipo de erro específico como erros de leitura ou escrita. Adicionalmente, ainda existe a possibilidade de saber exatamente que tipo de erro ocorreu, inclusive em alguns casos é possível ter acesso ao *errno* disponibilizado pelo sistema operacional. A figura 4.3 mostra o modelo das classes responsáveis pela identificação dos diferentes tipos de erros.

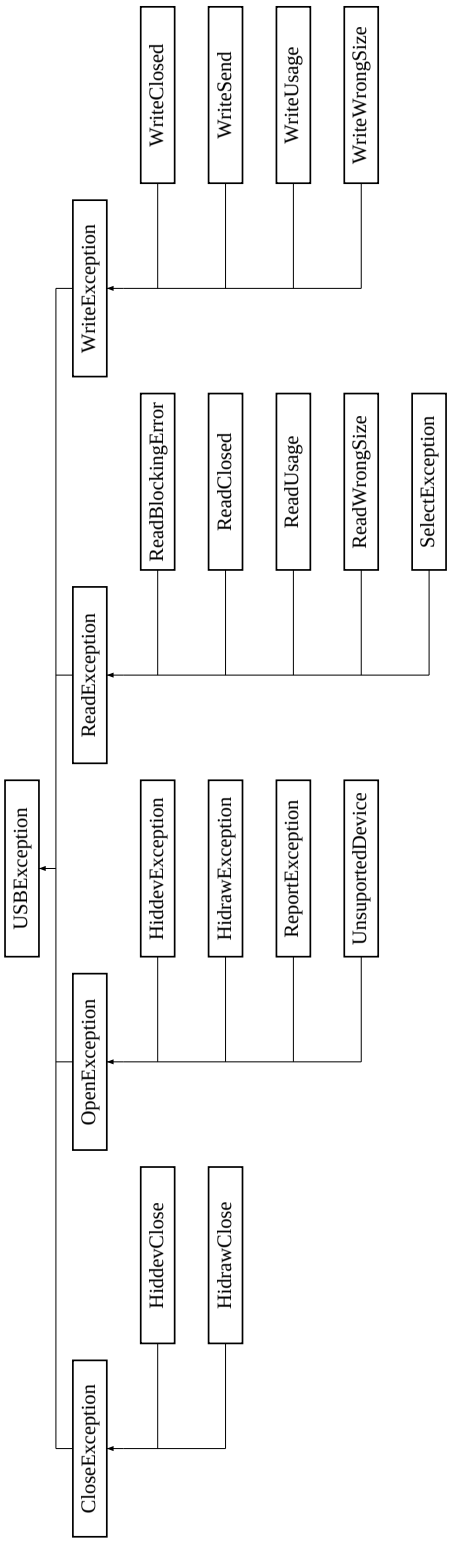

Figura 4.3: Sistema de Testes

Capítulo 5

Resultados e Conclusões

Uma etapa importante no desenvolvimento de qualquer arquitetura de hardware/software é a análise de desempenho do sistema. Para validar a arquitetura proposta nesse trabalho desenvolvemos, como destacado anteriormente, uma pequena aplicação que engloba dois sensores (uma bússola e um sonar) e um conjunto de atuadores (quatro motores brushless). Como os motores brushless funcionam em malha aberta eles não fornecem uma resposta mensurável em termos de comunicação. Por esse motivo, os testes executados consistem basicamente na análise dos tempos de respostas da USB e dos sensores.

O programa de testes de alto-nível é composto por duas threads: uma de escrita e outra de leitura. A thread de escrita envia um dado para o microcontrolador através da USB, e fica bloqueada esperando que a thread de leitura consuma o dado. A thread de leitura, por sua vez, fica em modo bloqueado esperando a chegada dos dados enviados pelo microcontrolador. Durante os testes foram avaliados três fatores: a integridade dos dados, o tempo que a thread de leitura ficou bloqueada e o tempo decorrido entre a escrita e a leitura.

Foram feitos três experimentos diferentes. O primeiro teste consistiu em enviar uma requisição através da USB. Neste teste, o microcontrolador recebe uma requisição, e deve incrementar uma variável interna e enviá-la de volta ao computador. Esse teste permite medir os tempos de envio através da USB, sem se preocupar com os atrasos decorrentes dos sensores.

O segundo teste utilizou uma bússola digital como base. Nesse teste foram enviadas 1.000 requisições de bússola e recebidas suas respectivas respostas. Através da análise desses tempos é possível entender como a arquitetura se comporta ao responder a um sensor com tempo de resposta rápido. O terceiro teste, por fim, teve como objetivo analisar o comportamento da arquitetura ao lidar com um sensor que possui atraso no tempo resposta. Para esse teste foram feitas 500 requisições ao sonar a intervalos fixos de 50ms. A partir do atraso decorrente do envio dos dados até sua chegada é possível avaliar como os diferentes tempos de polling da USB influenciam o comportamento do sistema.

Durante os testes percebemos alguns resultados interessantes, como destacados a seguir. O primeiro deles é que em nenhum dos experimentos houve perda de pacotes. Isso ocorre por que os endpoints utilizados pela classe HID são do tipo Interrupt, esse tipo de transferência garante a integridade dos dados e procura enviá-los em intervalos de tempo bem definidos. Assim, se houver alguma falha no envio durante um ciclo de polling, o dado será enviado novamente no próximo ciclo. Outro aspecto interessante é que por uti-

lizar um sistema operacional convencional não existe determinismo nos dados. Portanto, os dados colhidos nos testes precisam ser analisados estatisticamente. A tabela 5.1 mostra os resultados obtidos ao analisar-se o tempo decorrido desde o envio dos dados para o microcontrolador até o recebimento dos dados no computador. A tabela informa a média e o desvio padrão do tempo de espera para cada um dos experimentos citados acima.

Polling USB	USB		BÚSSOLA		SONAR	
	Média	Desvio	Média	Desvio	Média	Desvio
1ms	1.988	0.151	2.072	0.308	15.908	0.071
2ms	3.961	0.729	3.965	0.821	15.907	0.107
3ms	3.962	0.927	3.961	0.756	15.913	0.075
4ms	7.959	0.223	7.959	0.259	17.917	0.090
5ms	7.960	0.295	7.960	0.340	17.927	0.092
6ms	7.961	0.199	7.958	0.156	17.927	0.075
7ms	7.959	0.332	7.962	0.283	17.918	0.182
8ms	15.972	0.637	15.959	0.085	21.898	0.235
9ms	15.956	0.210	15.955	0.193	21.903	0.167
10ms	15.962	0.074	15.957	0.112	21.909	0.033

Tabela 5.1: Médias e desvios padrão do tempo de espera em milissegundos.

Note que em todos os casos o tempo de espera real é sempre superior ao tempo de polling da USB. Isso ocorre porque tanto o endpoint de entrada, como o de saída estão sujeitos ao mesmo tempo de polling. Por exemplo, se o polling é de 1ms o endpoint In e o endpoint Out são verificados a 1ms cada, o que implica no pior caso um atraso de 2ms. Outro aspecto importante que influencia os resultados é a atuação do escalonador do sistema operacional. Quando o processo é escalonado ele perde o domínio sobre a CPU e pode ficar certo período de tempo bloqueado, no caso do Linux, esse tempo é de no mínimo 10ms. Por esse motivo podem existir alguns picos de amplitude média de 10ms em alguns gráficos do tempo de espera, como na Figura 5.1.

A partir da análise dos dados acima e do estudo dos gráficos do tempo de resposta percebemos que as melhores respostas do sistema foram com os tempos de polling de 1ms, 6ms e 10ms. O principal parâmetro dessa avaliação é a oscilação do tempo de resposta, ou seja, o melhor tempo de resposta é aquele que no decorrer da execução menos se afastou do valor médio. Para ilustrar alguns desses tempos de resposta as figuras 5.2, 5.3 e 5.4 mostram o comportamento do sistema para cada um dos experimentos descritos acima, utilizando o tempo de polling de 1ms.

Assim, ao se analisar o comportamento da arquitetura de hardware/software proposta nesse trabalho chega-se a algumas conclusões interessantes. A primeira é que a utilização da USB como *backbone* de comunicação entre um computador e um conjunto de microcontroladores permite a construção de um sistema de comunicação extremamente rápido e seguro. Um segundo ponto é que graças as funcionalidades disponibilizadas pela USB é possível construir praticamente qualquer sistema robótico com arquitetura mestre-escravo utilizando o modelo proposto nesse trabalho.

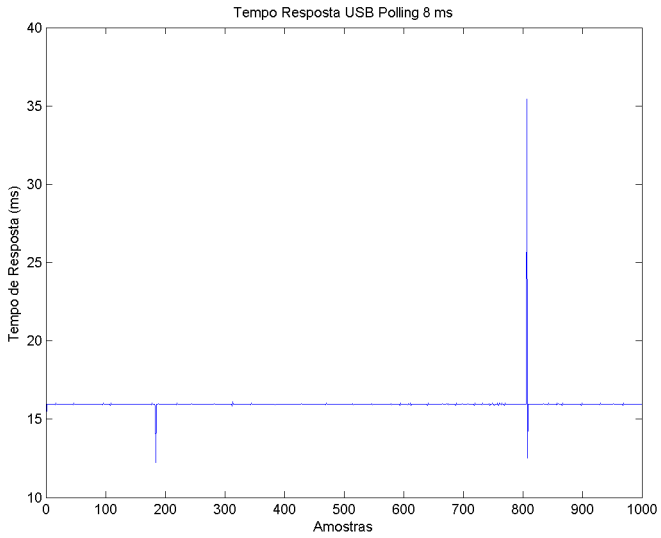

Figura 5.1: Tempo de resposta USB com polling de 8ms.

Um outro resultado interessante foi a possibilidade de usar diversas interfaces HID diferentes no mesmo microcontrolador. Como destacado em seções anteriores, o sistema operacional associa um driver a cada interface e não a cada dispositivo. Assim, utilizando várias interfaces é possível associar cada sensor conectado ao microcontrolador a uma interface específica. Desse modo, a nível de sistema operacional, no lugar de um único dispositivo (o microcontrolador), o sistema visualiza três “dispositivos” (interfaces) diferentes: os motores, a bússola e o sonar.

A utilização do modelo de interfaces possibilita, ao programador de alto-nível, um modelo extremamente flexível de acesso aos recursos do robô. Por exemplo, o programador poderia definir uma classe específica para acessar e administrar cada um dos sensores, tudo isso abstraindo a presença do microcontrolador. Uma observação interessante é que a utilização de interfaces não diminui a taxa de transmissão do sistema, como cada uma dessas interfaces utiliza apenas endpoints Interrupt, a controladora USB reserva banda suficiente para cada um desses endpoints. Vale lembrar que cada um desses endpoints tem taxa de transmissão de 64 Kbits/s, enquanto o host pode disponibilizar até 480 Mbits/s. Portanto, é possível utilizar um número razoável de interfaces, limitado apenas pelo tamanho do campo de endereçamento, que suporta 128 endereços diferentes.

Por fim, o resultado mais animador do trabalho foi a possibilidade de mesmo trabalhando com um SO Linux convencional, poder atender a certas condições de tempo real. Como observado nos experimentos a arquitetura tem a tendência de sempre responder num tempo médio aceitável e com um desvio padrão pequeno. É verdade que graças a atuação do escalador ou da carga de processos presentes no sistema, em alguns momentos o tempo de resposta será maior do que o esperado. No entanto, se a aplicação projetada para utilizar essa arquitetura tiver meios de se recuperar de atrasos esporádicos oriundos do SO, a arquitetura proposta nesse trabalho poderá ser empregada.

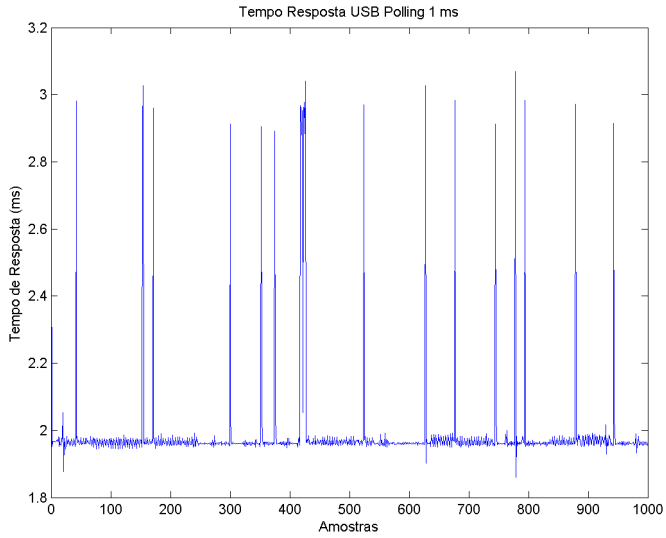

Figura 5.2: Tempo de resposta USB com polling de 1ms.

Vale salientar que no projeto AEROPETRO, onde essa arquitetura será primeiramente empregada, as restrições de tempo real não são rígidas. Primeiro porque a utilização de um sistema operacional de tempo real seria impraticável devido à necessidade de processar imagens. Segundo, porque o controle que será embarcado no sistema tem um tempo de resposta compatível com os resultados da arquitetura. Os atuadores que serão empregados no robô possuem um tempo de resposta de 20ms, portanto, superior aos tempos de resposta médios obtidos nos testes. Desse modo, a arquitetura de hardware e software desenvolvida nesse trabalho é perfeitamente adaptável para utilização em veículos aéreos não-tripulados.

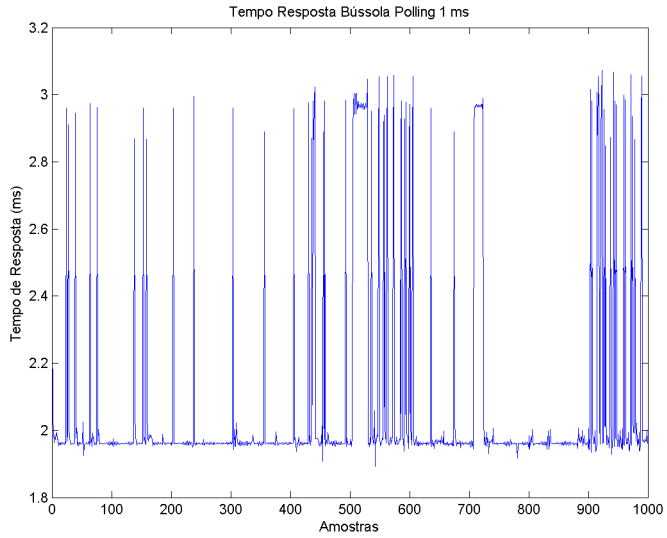

Figura 5.3: Tempo de resposta da Bússola com polling de 1ms.

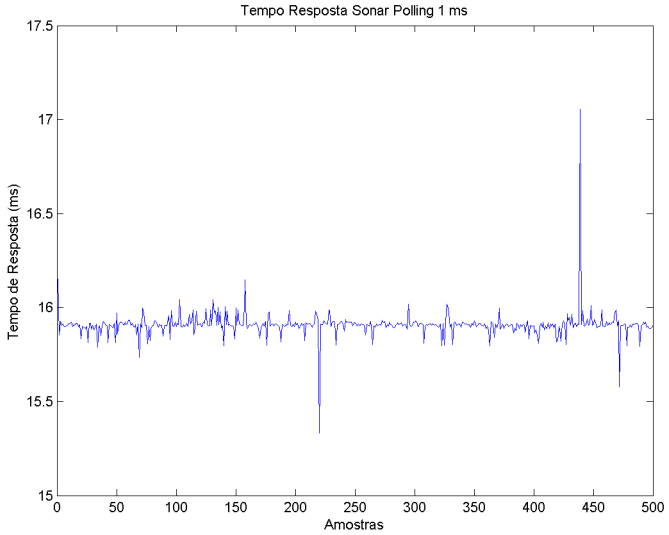

Figura 5.4: Tempo de resposta Sonar com polling de 1ms.

Referências Bibliográficas

Forum, USB Implementers' (2001), *Device Class Definition for Human Interface Devices (HID)*, USB Implementers' Forum.

Jonathan Corbet, Alessandro Rubini & Greg Kroah-Hartman (2005), *Linux Device Drivers, Third Edition*, O'Reilly Media.

Karim Yaghmour, Jon Masters, Gilad Ben-Yossef & Philippe Gerum (2008), *Building Embedded Linux Systems, Second Edition*, O'Reilly Media.

Microchip (2007), *PIC 18F2455/2550/4455/4550 Data Sheet*, Microchip Technologies.

usb.org (2000), *Universal Serial Bus Specification Revision 2.0*, usb.org.

Apêndice A

Modelo de Mensagens

Para que o computador e o microcontrolador possam comunicar-se adequadamente é preciso estabelecer um protocolo entre eles. Com esse objetivo foi criado um pequeno protocolo de comunicação baseado em troca de mensagens para o firmware de testes. As mensagens a seguir são divididas em quatro grupos básicos:

1. Mensagens de Motores
2. Mensagens da Bússola
3. Mensagens do Sonar
4. Mensagens de Modo de Operação

Para facilitar a compreensão das mensagens abaixo é importante compreender alguns detalhes da notação. Primeiro, cada bloco destacado no modelo representa um byte individual. E todas as mensagens seguem o modelo abaixo:

TIPO	OPÇÕES	PAYLOAD

A.1 Motores

SET_MOTORES: Cada motor é acionado por um valor entre 0-1000.

MOTORES	SET_MOTORES	MT1_H	MT1_L	...	MT4_H	MT4_L

GET_MOTORES: Requisição

MOTORES	GET_MOTORES

GET_MOTORES: Retorna um valor entre 0-1000 para cada motor

MOTORES	GET_MOTORES	MT1_H	MT1_L	...	MT4_H	MT4_L

A.2 Bússola

REQ_BUSSOLA: Requisição da Bússola

BUSSOLA	REQ_BUSSOLA

REPLY_BUSSOLA: Resposta da requisição

BUSSOLA	REPLY_BUSSOLA	BUSSOLA_H	BUSSOLA_L

- Retorna o valor em graus 0.00 até 360.00.
- O valor está armazenado em dois inteiros.

STATUS_BUSSOLA: Requisição

BUSSOLA	STATUS_BUSSOLA

STATUS_BUSSOLA: Resposta

BUSSOLA	STATUS_BUSSOLA	MODO_DE_OPERAÇÃO	POLLING

- MODO_DE_OPERAÇÃO: 0 (Requisição) e 1 (Automático)
- POLLING: se estiver em automático, determina de quanto em quanto tempo envia dados.
- Caso deseje mudar do modo automático para o manual basta enviar uma REQ_BUSSOLA.

ERROR_BUSSOLA: Inidica que ocorreu um erro no bússola

BUSSOLA	ERROR_BUSSOLA

A.3 Sonar

REQ_SOANR: Requisição de Sonar

SONAR	REQ_SONAR

REPLY_SONAR: Resposta da requisição

SONAR	REPLY_SONAR	SONAR_H	SONAR_L

- Retorna a distância em cm.
- O valor está armazenado em dois inteiros.

STATUS_SONAR: Requisição

SONAR	STATUS_SONAR

STATUS_SONAR: Resposta

SONAR	STATUS_SONAR	MODO_DE_OPERAÇÃO	POLLING

- MODO_DE_OPERAÇÃO: 0 (Requisição) e 1 (Automático)
- POLLING: se estiver em automático, determina de quanto em quanto tempo envia dados.
- Caso deseje mudar do modo automático para o manual basta enviar uma REQ_SONAR.

ERROR_SONAR: Inidica que ocorreu um erro no sonar

SONAR	ERROR_SONAR

A.4 Modo Automático

Nesse modo o sensor recebe uma requisição avisando que ele deve enviar os dados continuamente ao mestre. Esses dados são enviados a uma taxa fixa especificada na requisição.

POLLING: Configura o modo automático, e define a taxa de envio dos dados.

AUTO	POLLING	MS_H	MS_L

- O intervalo de envio é definido em milissegundos.
- O valor máximo de 1.000, ou seja, 1 segundo.

SENSOR: Define que sensor entrará em modo de envio automático

AUTO	SENSOR	TIPO

- O TIPO pode ser SONAR ou BUSSOLA.

Cada uma das TAGs descritas nas mensagens acima estão definidas no arquivo *messageflags.h*, disponibilizado no pacote da biblioteca USBRobot.