UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

ROBÔ AUTÔNOMO DE EIXO ÚNICO

Autor: _____
André Leite Futuro

Orientador: _____
M.Sc. Carlos José Ribas D'Avila

Examinador: _____
M.Sc. José Paulo Brafman

Examinador: _____
M.Sc. Jomar Gozzi

DEL
Dezembro de 2006

a meus Pais,

pelo amor

e dedicação

Agradecimento

Primeiramente gostaria de agradecer a toda minha família, por nossa união e por estarem sempre presente em todos os momentos de minha vida. Em especial a meus pais, Maura e Fernando Luiz, pelo apoio e incentivo durante toda faculdade, a meus irmãos, Leonardo, Mariana e Lais, pela amizade e alegria que nos mantém juntos, a minhas avós, Isis e Miracy, pelo carinho e amor que têm por mim. A meus tios Maria Helena e Marcelo, por terem me ajudado com esse projeto, no momento em que eu não sabia como prosseguir. À Letícia Esperança, pela sua presença nos momentos bons e ruins de minha vida universitária, me apoiando em todas minhas decisões.

Ao meu orientador Carlos José D'Avila – Casé – por ter acreditado em mim, me dando seu voto de confiança mesmo quando eu já estava sem esperanças de conseguir terminar. Por todo apoio material que me disponibilizou, sem o qual seria impossível completar o projeto. Ao Pablo Salino, por ter me acompanhado durante boa parte da caminhada desse trabalho. Ao Paulo Gentil, pela sua ajuda constante, principalmente na reta final.

Um especial agradecimento a todos os amigos que fiz durante essa longa jornada na Universidade, pela união que temos e desafios que enfrentamos juntos.

Agradeço principalmente a Deus, pois sem ele nada disso seria possível.

Resumo

O robô de equilíbrio tem como característica principal seu corpo rígido e seu eixo único com duas rodas independentes. Ele é baseado no conceito do pêndulo invertido. A idéia é bem simples: giram-se as rodas na direção em que a parte superior do robô está caindo. Conseguindo fazer com que as rodas permaneçam em baixo do centro de gravidade do robô, mantê-lo-á equilibrado.

Diversos e diversificados itens são necessários para fazer esse projeto. Ele envolve mecânica, eletrônica, controle e computação. Uma estrutura de alumínio, contendo mancais, eixos, rolamentos, suporte para motores e pilares para prateleira de acessórios é necessária. Dois motores, com suficiente torque para mover a massa da estrutura e manter as rodas em baixo do centro, devem ser utilizados.

Para se controlar o sistema instável, é necessário que um estado seja conhecido para fazer a realimentação. Esse estado é o ângulo de inclinação com a vertical. Obtemos ele utilizando dois sensores de aceleração, postos perpendicularmente entre si e perpendicular ao eixo das rodas, e um giroscópio, também posicionado perpendicular ao eixo das rodas. Assim, os acelerômetros conseguem medir a aceleração da gravidade e o giroscópio a velocidade angular do robô. Um Filtro de Kalman é implementado em um micro-controlador de baixo custo – PIC – para juntar os valores dos sensores e fornecer o resultado. Esse sensor de inclinação foi feito de forma modular, podendo ser utilizado em outros projetos.

Com apenas um controle PID já é necessário controlar o sistema instável do robô de balanço. Há um tratamento especial na componente integral, para evitar o efeito *Windup* do controlador e melhorar a resposta do sistema real às perturbações.

O resultado é um robô de 50 cm com apenas duas rodas que consegue permanecer equilibrado na vertical, mesmo sofrendo perturbações externas.

Palavras-chave

- Robô de Equilíbrio
- Pêndulo Invertido
- Controle PID
- Sensor de Inclinação
- Acelerômetro
- Giroscópio
- Filtro de Kalman

Índice do Texto

Índice de Figuras

Índice de Tabelas

1. APRESENTAÇÃO

Este trabalho apresenta um robô de equilíbrio – *ballancing robot*. Ele é constituído de um corpo rígido e apenas um eixo com duas rodas. Por ser um sistema instável, técnicas de controle devem ser aplicadas para que este se mantenha equilibrado na vertical. Percebe-se durante a leitura desse trabalho que a diversidade de assuntos foi um dos maiores desafios encontrados. Estaremos falando sobre teorias de controle, estrutura física, motores, programação, e eletrônica.

Esta monografia está estruturada em nove capítulos. No capítulo 2 é feita uma introdução ao projeto, apresentando a motivação, os objetivos e metas do projeto. Nesse capítulo também é feita a contextualização do problema.

No capítulo 3, a fundamentação teórica é feita com os conceitos básicos de controle, a metodologia aplicada, a modelagem do sistema, o desenho do compensador e o controle PID. Algumas simulações são utilizadas durante o processo.

No capítulo 4, a parte mecânica e a estrutura do robô são vistas. Apresentamos a modelagem da estrutura física, assim como os cálculos de potência e torque para a escolha dos motores, engrenagens e baterias.

No capítulo 5, a implementação do sistema é vista. Nesse capítulo, apresentamos os sensores tanto o de inclinação como o de posição. Mostramos, também, o circuito de potência e a geração do sinal PWM para a alimentação do motor. Finalizando este capítulo, mostramos a implementação do controle PID no PIC e a alimentação dos circuitos.

No capítulo 6, os resultados obtidos no projeto são apresentados. No capítulo 7, temos com as conclusões e falamos dos trabalhos seguintes também. Por fim, o capítulo 8 contém a bibliografia do trabalho. Sete anexos podem ser encontrados nas últimas páginas da monografia, incluindo os arquivos de simulação do Matlab, os cálculos do acelerômetro, os esquemáticos dos circuitos, os códigos fonte, e uma especificação do sistema.

2. INTRODUÇÃO

Atualmente, o pêndulo invertido é um dos modelos de controle mais clássicos encontrados. Ele já foi explorado por diversos pesquisadores e implementado de diversos modos, em forma acadêmica ou em algum projeto no campo.

A idéia do projeto descrito nessa dissertação visa utilizar os conhecimentos de controle de um pêndulo invertido e aplicá-los a um robô que se equilibre em apenas um eixo com duas rodas. Este robô, além de ser capaz de permanecer parado no local, sem desequilibrar, também deve ser capaz (em uma evolução desse projeto) de andar para frente e para trás, assim como fazer curvas. Suas decisões são tomadas automaticamente sem ser necessário a interface com um operador. O robô foi construído desde o princípio, incluindo todos os circuitos eletrônicos necessários, a estrutura física e a programação de um controle adequado.

Este projeto tem um valor acadêmico considerável, vendo que as técnicas de controle clássicas de um pêndulo invertido estão sendo aplicadas a um modelo real. Olhando pelo lado da eletrônica aplicada, distintos campos são abordados, como sensores, filtros e eletrônica de potência. Os conhecimentos de micro-compudadores também são postos à prova, nos dois micro-controladores usados no projeto.

Apesar de em países estrangeiros existirem diversos trabalhos semelhantes, não foi encontrada nenhuma referência nacional de um trabalho semelhante construído ou ao menos projetado. O desafio em relação às outras implementações encontradas é a utilização de um micro-controlador de baixo custo e não, como nos outros casos, um micro-processador.

2.1. História

Em nossa sociedade, o conceito do pêndulo invertido é antigo. Ele é utilizado para diferentes fins.

Sua primeira utilização foi pelo sismógrafo escocês James Forbes para fins geológicos. Após uma pesquisa realizada em uma série de pequenos terremotos próximos a Perthshire, Escócia, Forbes criou o *Pêndulo Invertido Seismometer* 8.

O conceito do pêndulo invertido também é aplicado para os foguetes. O mesmo comportamento visto no pêndulo invertido pode ser verificado ao modelar um foguete. Então,

o estudo desse modelo passou a ser prioridade em pesquisas espaciais para que se pudesse ter controle dos foguetes lançados ao espaço no século XX.

Na década de noventa, um novo conceito de transporte foi introduzido ao mercado pelo engenheiro americano Dean Kamen: o Segway. Analisando como as cidades de hoje são preparadas para pessoas que se equilibram (nos equilibramos em duas pernas), Kamen desenvolveu um sistema para equilibrar humanos em duas rodas. Dois produtos fizeram muito sucesso: o Segway HT 8, para transporte de pedestres e as cadeiras de rodas iBOT 8.

2.2. Estado da Arte

Como o pêndulo invertido é o modelo de controle mais estudado no mundo, o conceito de "balancing robot", ou robô de equilíbrio, já está relativamente difundido. Existem umas dezenas de referências de implementações espalhadas pela Europa, EUA e Japão. Nenhum trabalho nacional foi encontrado.

Não foi encontrado no mercado uma implementação do controle desse sistema instável utilizando recursos de baixo custo.

2.3. Conhecimento do Problema

Diversos desafios estão envolvidos em um projeto como esse. Eis aqui alguns desafios enfrentados pelos autores:

- Modelagem e construção de uma estrutura
- Aquisição de materiais, como motores, engrenagens e eixos
- Sensores e eletrônica associada
- Micro-controladores
- Eletrônica de potência para alimentação de motores
- Aplicação do controle de um pêndulo invertido em um micro-controlador

2.4. Motivação e Justificativa

Este projeto tem um valor acadêmico considerável, vendo que estamos aplicando a um modelo real as técnicas de controle clássicas de um pêndulo invertido. Os problemas apresentados no item anterior mostram que este é um trabalho multidisciplinar, pois além de incluir quase todas as especializações do curso de engenharia eletrônica, também inclui itens de mecânica. Consequentemente é um grande desafio a ser considerado.

Vemos esse projeto como solução para diferentes problemas:

- para robôs (com quatro rodas) que precisam ter uma base muito pequena e ao mesmo tempo devem ser relativamente compridos que necessitem andar por um plano inclinado mantendo uma postura vertical e/ou que necessitem ter uma mobilidade maior;
- criar um robô que se assemelhe mais ao comportamento humano, podendo se equilibrar em apenas dois de seus membros. Essa solução pode ser útil para robôs que devem interagir com humanos para que sua recepção seja mais aceitável.
- demonstrar uma aplicação já conhecida comercialmente, o *Segway*, um veículo para pedestres baseado nos mesmos conceitos do nosso projeto;

Esta é também uma oportunidade de termos também no Brasil um projeto de tal âmbito, já que não existe nenhuma referência nacional que tenha sido encontrada.

Os conhecimentos adquiridos nesse projeto, assim como partes do projeto podem ser utilizados para criar produtos que poderiam ser comercializados. Um exemplo seria o módulo constituído pelo sensor de inclinação totalmente modularizável e consequentemente utilizável em diferentes aplicações.

2.5. Objetivos e Metas

Como principais objetivos deste projeto final temos:

- Modelar o sistema em questão e modelar o controle necessário para manter o sistema de um robô baseado no pêndulo invertido estável. Com isso, colocaremos os conhecimentos de controles obtidos durante o curso em prática.

- Otimizar e desenvolver um software com o algoritmo de controle para que possa ser utilizado em um micro-controlador de baixo custo e que não possui cálculos com ponto-flutuante. Esse software deve conter uma alta eficiência em termos de quantidade de instruções, tempo de processamento e resposta obtida.
- Conhecer e utilizar de um micro-controlador, incluindo todas suas principais funções.
- Criar sensores eletrônicos para monitorar o sistema e, assim, auxiliar no controle. Esses sensores devem ser feitos como módulos independentes, para que possam ser reutilizados em outros projetos e produtos.
- Construir a estrutura do robô para que o estudo possa ser colocado em prática.

3. FUNDAMENTAÇÃO TEÓRICA

3.1. Conceitos Básicos de Controle

À nossa volta vemos todos os dias situações em que, caso não haja alguma interferência externa, essa situação irá para o caminho não desejado. Essa ação é sempre feita tomando em consideração algum resultado anterior, que se possa basear. E dessa simples idéia de ação, resultado e retroação, que se fundamentam os conceitos de controle. Esses fundamentos podem ser aplicados nas mais variados itens de nosso cotidiano, como quando fritamos um ovo (controle visual), dirigimos um carro, quando aquecemos a água do chuveiro ou mesmo quando pequenos brincávamos de equilibrar um cabo de vassouras na ponta dos dedos.

Um sistema de controle pode ser considerado uma interconexão de itens formando uma configuração de sistema que produzirá uma resposta desejada ao sistema. Temos como base para a análise de um sistema os fundamentos fornecidos pela teoria dos sistemas lineares, que supõe que para os componentes do sistema há uma relação de causa e efeito. A representação de um processo que desejamos controlar é normalmente representado pela Figura 3 .1.

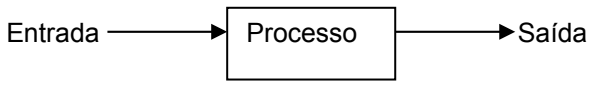


Figura 3.1 – Processo a ser controlado.

A relação de causa e efeito é representada pela relação de entrada-saída. Como há uma influencia da saída sobre a entrada, dizemos que esse é um sistema em malha aberta. Assim como a relação de entrada-saída, podemos criar uma relação de saída-entrada.

Diferente de um sistema de controle a malha aberta, um sistema de controle a malha fechada utiliza uma medida adicional da saída real com a resposta desejada. A medida da saída é chamada de sinal de realimentação como mostrado na Figura 3 .2.

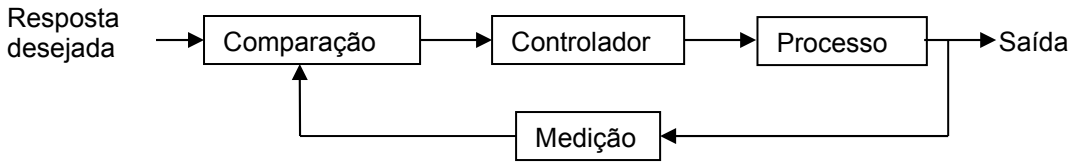


Figura 3.2 – Sistema de controle com retroação a malha fechada.

Um sistema de controle com realimentação é um sistema que tende a manter uma relação preestabelecida entre uma variável de sistema e outra, comparando funções destas variáveis e usando a diferença como meio de controle. Nesse tipo de sistema utiliza-se freqüentemente uma função e uma relação preestabelecida entre a saída e a entrada para controlar o processo, sendo que quase sempre a diferença entre a saída do sistema e a referência de entrada é amplificada e usada para controlar o processo de modo que a diferença seja continuamente reduzida. O conceito de realimentação tem sido o alicerce para a análise e o projeto dos sistemas de controle.

Por causa do aumento de complexidade dos sistemas de controle e do interesse em obter um desempenho ótimo, a importância da engenharia de controle cresceu nas últimas décadas. Além disso, como os sistemas se tornaram mais complexos, o inter-relacionamento de muitas variáveis controladas deve ser considerado na estrutura de controle. Um diagrama de blocos retratando um sistema de controle multivariável está ilustrado na Figura 3 .3.

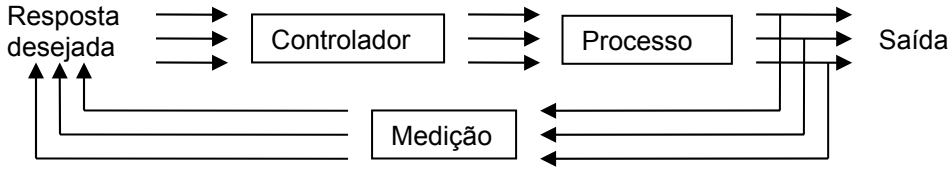


Figura 3.3 – Sistema de controle multivariável.

Para exemplificar um sistema comum de controle a malha aberta podemos considerar a torradeira elétrica de cozinha. Um exemplo de um sistema de controle de malha fechada é uma pessoa dirigindo um automóvel (supondo que os olhos desta pessoa estejam abertos) a partir da observação da posição do automóvel e fazendo ajustes.

3.2. Modelagem do Sistema

3.2.1. Demonstração do Problema, Requerimentos de Desenho

O robô de equilíbrio, cujo esquema simplificado é mostrado na Figura 3 .4, é um sistema instável. Teremos que identificar as equações dinâmicas que descrevem o sistema e, considerando que o robô quase não se inclina, linearizá-las para um ângulo θ tendendo a zero. Em seguida teremos que encontrar um controle que satisfaça nossos requerimentos de equilíbrio.

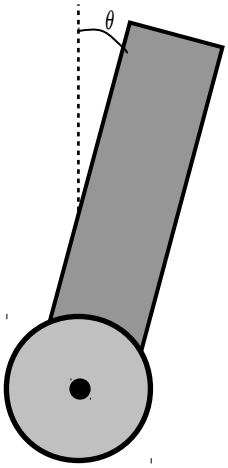


Figura 3.4 – Representação do robô de equilíbrio: corpo e eixo de rodas.

Para nossos cálculos, vamos assumir:

- M → massa da roda
- m → massa da parte superior, que chamaremos de pêndulo
- b → coeficiente de fricção da roda
- l → distância do eixo ao centro de massa
- I → momento de inércia do pêndulo
- F → força que deve ser aplicada às rodas por um motor
- x → posição da roda
- θ → ângulo do pêndulo com a vertical

Os requerimentos para esse sistema são:

- tempo para estabilizar de no máximo 1 segundo
- a variação do ângulo não deve exceder 30º, sendo 15º para cada lado.

3.2.2. Análise de forças e sistema de equações

Podemos considerar o sistema do robô de equilíbrio como um sistema de dois corpos. Veja na Figura 3 .5 os diagramas dos dois corpos que regem o sistema separadamente.

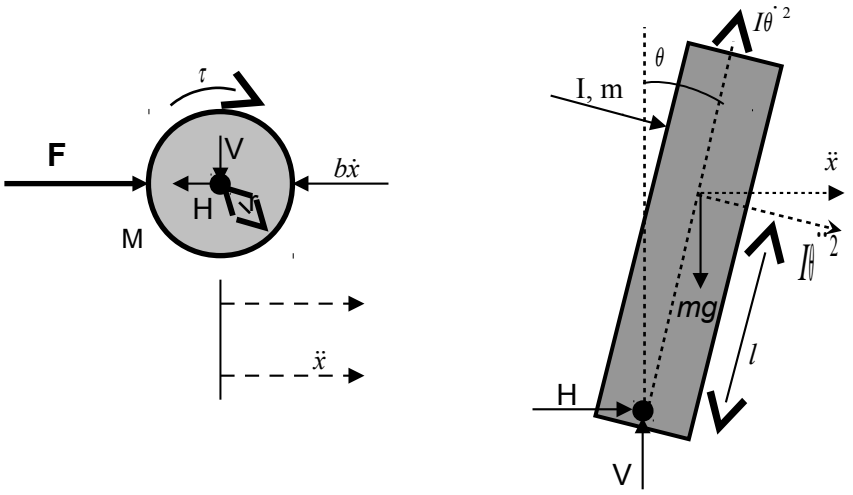


Figura 3.5 – Representação em dois sistemas separados.

Assumindo as forças na horizontal no diagrama da roda, encontramos a seguinte equação:

$$M\ddot{x} + b\dot{x} + H = F \qquad (3.1)$$

Note que poderíamos, também, ter somado as forças na vertical, mas não obteríamos informações úteis para nosso desenvolvimento, pois não há trabalho nesse eixo.

Lembramos também que no nosso caso, não estamos apenas atrás da força que iremos aplicar no sistema, mas principalmente atrás do torque que devemos aplicar para exercer essa força. Então temos que:

$$\tau = r \times F = r \cdot F \cdot \operatorname{sen}\alpha \qquad \textit{onde r é o raio da roda} \qquad (3.2)$$

Em nosso caso temos que α será sempre 90º, portanto, seu seno é unitário. Temos então,

$$\tau = r \cdot F \qquad (3.3)$$

$$\Rightarrow F = \frac{\tau}{r}$$

Vamos continuar trabalhando com F, pois ainda não decidimos o torque do motor que estaremos utilizando, nem o raio da roda.

Somando as forças na horizontal no diagrama do pêndulo acima, encontra-se a seguinte equação:

$$H = m\ddot{x} - ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2 sen\theta \tag{3.4}$$

Substituindo a equação (3 .4) na equação (3 .1), obtemos o seguinte resultado:

$$\boxed{(M+m)\ddot{x} + b\dot{x} - ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2 sen\theta = F} \tag{3.5}$$

Para obter a segunda equação de movimento, somam-se as forças perpendiculares ao pêndulo. Resolvendo o sistema por esse eixo, obtemos a equação (3 .6).

$$V sen\theta + H\cos\theta - mgsen\theta = -ml\ddot{\theta} + m\ddot{x}\cos\theta \tag{3.6}$$

Para se livrar dos termos em V e H na equação (3 .6), somamos os momentos ao redor do centróide do pêndulo.

$$-Vlsen\theta - Hl\cos\theta = -I\ddot{\theta} \tag{3.7}$$

Lembramos que o momento de inércia I de um pêndulo invertido ideal pode ser calculado com a fórmula apresentada em (3 .8).

$$I = \frac{1}{3}ml^2 \tag{3.8}$$

Combinando as equações (3 .6) e (3 .7), conseguimos a segunda equação dinâmica do pêndulo.

$$\boxed{(I+ml^2)\ddot{\theta} - mgl sen\theta = ml\ddot{x}\cos\theta} \tag{3.9}$$

Com isso, obtivemos nossas duas equações dinâmicas que descrevem o sistema. Elas são reescritas abaixo para efeitos visuais.

$$\begin{cases} (M+m)\ddot{x} + b\dot{x} - ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2 sen\theta = F \\ (I+ml^2)\ddot{\theta} - mgl sen\theta = ml\ddot{x}\cos\theta \end{cases} \tag{3.10}$$

Podemos notar que alguns parâmetros aparecem repetidamente nas expressões acima. Vamos, então, fazer algumas substituições para simplificar nossas expressões e facilitar nossa álgebra.

$$\begin{cases} J = I + ml^2 = \frac{1}{3}ml^2 + ml^2 = \frac{4}{3}ml^2 \\ K = mgl \\ L = ml \\ P = M + m \end{cases} \tag{3.11}$$

Nosso objetivo, agora, é fazer com que cada equação contenha apenas uma derivada de segunda ordem. Após algum trabalho algébrico, podemos obter o resultado descrito nas equações (3 .12) e (3 .13).

$$\ddot{x} = \frac{-Jb\dot{x} + KL\cos\theta\,sen\theta + LJ\dot{\theta}^2 sen\theta + JF}{PJ - L^2\cos^2\theta} \tag{3.12}$$

$$\ddot{\theta} = \frac{-Lb\dot{x}\cos\theta + KPsen\theta + L^2\dot{\theta}^2\cos\theta\,sen\theta + LF\cos\theta}{PJ - L^2\cos^2\theta} \tag{3.13}$$

3.2.3. Lineariazação do Sistema

Para podermos trabalhar com esse sistema de equações e construirmos um modelo de controle, temos que linearizá-las.

θ = 0º

Como pretendemos equilibrar o pêndulo na vertical, o ponto de linearização mais propício a ser escolhido é com θ bem próximo de 0º. Assumimos, então, $\theta = 0° + \phi$ (onde ϕ representa um ângulo muito pequeno). Assim temos que:

$$\begin{cases} \cos(\theta) = 1 \\ sen(\theta) = \theta \\ \dot{\theta}^2 = 0 \end{cases} \tag{3.14}$$

Depois de efetuarmos a linearização no conjunto de equações (3 .12) e (3 .13), as equações de movimento se tornam:

$$\begin{cases} \ddot{x} = \dfrac{-Jb\dot{x} + KL\theta + JF}{PJ - L^2} & (3.15) \\ \ddot{\theta} = \dfrac{-Lb\dot{x} + KP\theta + LF}{PJ - L^2} & (3.16) \end{cases}$$

Equação de Espaço de Estado

Escrevemos, então, na forma matricial:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \dfrac{-Jb}{PJ - L^2} & \dfrac{KL}{PJ - L^2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \dfrac{-Lb}{PJ - L^2} & \dfrac{+KP}{PJ - L^2} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \dfrac{J}{PJ - L^2} \\ 0 \\ \dfrac{L}{PJ - L^2} \end{bmatrix} F(t) \qquad (3.17)$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} F(t)$$

A matriz C da equação (3 .17) é 2 por 4, pois tanto a posição do robô quanto a inclinação fazem parte da saída. Para o problema do desenho de estado-espaço, estaremos controlando um sistema multi-saídas, com a posição do robô na primeira linha, e a inclinação na segunda linha.

Função de Transferência

Para obter a função de transferência de nossas equações linearizadas, vamos utilizar a transformada de Laplace sob as equações.

$$\begin{cases} X(s)s^2 = \dfrac{-JbX(s)s + KL\theta(s) + JF(s)}{PJ - L^2} & (3.18) \\ \theta(s)s^2 = \dfrac{-LbX(s)s + KP\theta(s) + LF(s)}{PJ - L^2} & (3.19) \end{cases}$$

Isolando X(s) na equação (3 .18), obtemos:

$$X(s) = \frac{KL\Theta(s) + JF(s)}{(PJ - L^2)s^2 + Jbs} \tag{3.20}$$

Isolando X(s) na equação(3 .19), permanecemos com:

$$X(s) = \frac{-(PJ - L^2)\Theta(s)s^2 + KP\Theta(s) + LF(s)}{Lbs} \tag{3.21}$$

Igualando as últimas duas equações, teremos:

$$\frac{\Theta(s)}{F(s)} = \frac{L(PJ - L^2)s^2}{(PJ - L^2)^2 s^4 + Jb(PJ - L^2)s^3 - KP(PJ - L^2)s^2 - KPJbs + KL^2bs} \tag{3.22}$$

Simplificando e substituindo os valores de (3 .11), obtemos:

$$\boxed{\frac{\Theta(s)}{F(s)} = \frac{\frac{ml}{q}s}{s^3 + \frac{(I + ml^2)b}{q}s^2 - \frac{(M + m)mgl}{q}s - \frac{m\lg b}{q}}} \tag{3.23}$$

onde $q = (M + m)(I + ml^2) - m^2l^2$ (3.24)

Se desconsiderarmos o coeficiente de fricção, isto é, considerando b=0, podemos simplificar nossa função de transferência para:

$$\boxed{\frac{\Theta(s)}{F(s)} = \frac{\frac{ml}{q}}{s^2 - \frac{(M + m)mgl}{q}}} \tag{3.25}$$

Fazendo uma análise desta função de transferência, percebemos que ela não possui nenhum zero, mas possui dois pólos. Estes serão:

$$s = +\sqrt{\frac{(M + m)mgl}{q}}$$
$$s = -\sqrt{\frac{(M + m)mgl}{q}} \tag{3.26}$$

Como temos um dos pólos no lado direito do eixo s, o sistema é claramente instável.

3.2.4. Parâmetros do Sistema

Nos próximos passos, estaremos fazendo diversas simulações e cálculos de controladores. Por isso iremos necessitar dos dados reais de nossa estrutura mecânica. Vamos apresentar aqui os valores utilizados para os cálculos. Esses valores serão mostrados com detalhes no capítulo 4.

Item	Valor Real
M	0,3 kg
m	4,3 kg
l	0,153 m
g	9,8 N
I	0,00047 kgm^2
b	0,1

Tabela 3.1 : Valores reais.

Agora, podemos calcular o valor de q:

$$q = (M + m)(I + ml^2) - m^2l^2 = 0{,}0325 \tag{3.27}$$

Vamos apresentar também nossa função de transferência representada na equação (3 . 23) já com os valores substituídos:

$$\boxed{\frac{\Theta(s)}{F(s)} = \frac{20{,}2964s}{s^3 + 0{,}3126s^2 - 914{,}9624s - 19{,}8905}} \tag{3.28}$$

Ou então, desconsiderando o coeficiente de atrito (b=0), temos a equação (3 .25) reescrita da seguinte forma:

$$\boxed{\frac{\Theta(s)}{F(s)} = \frac{20{,}2964}{s^2 - 914{,}9624}} \tag{3.29}$$

3.2.5. Análise do Sistema sem Compensação

Vamos comprovar utilizando a simulação de nosso sistema sem compensação daquilo que, utilizando nosso bom senso, podemos concluir: o sistema é instável.

Considerando coeficiente de atrito

Na Figura 3 .6, apresentamos nosso sistema completo levando em consideração a equação (3 .23), respondendo a um impulso como entrada.

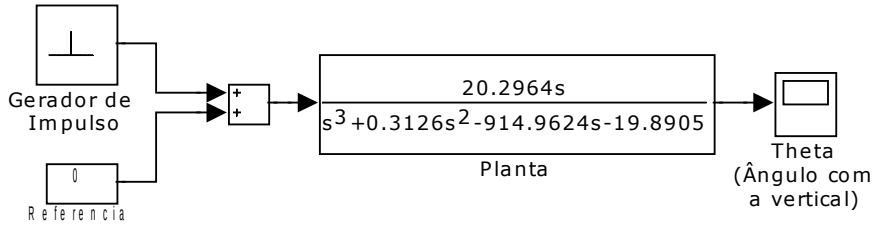


Figura 3.6 – Função de transferência em circuito aberto.

Como já comentamos antes, Figura 3 .7 mostra que o sistema sem compensação é instável, por possuir pólos no lado direito do eixo *s*.

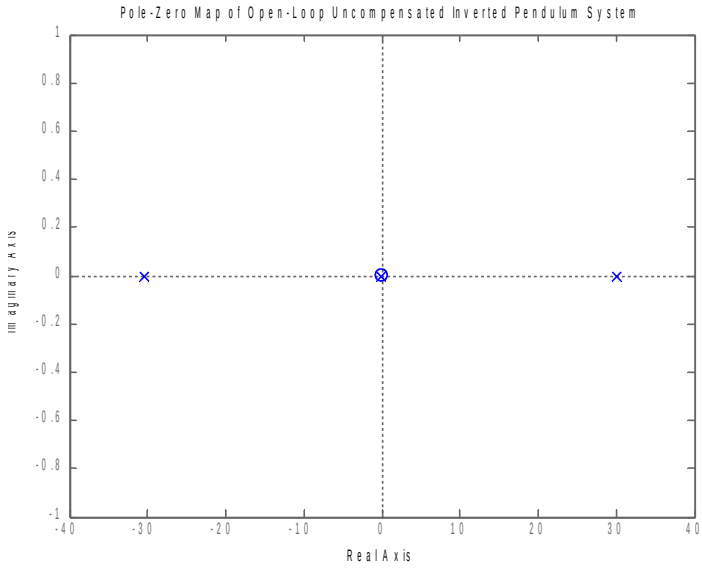


Figura 3.7 – Mapa Pólo-Zero do sistema sem compensação.

Também podemos ver claramente que o sistema é altamente instável através do gráfico da resposta do sistema a um impulso (Figura 3 .8) e a um degrau (Figura 3 .9). O ângulo θ diverge rapidamente.

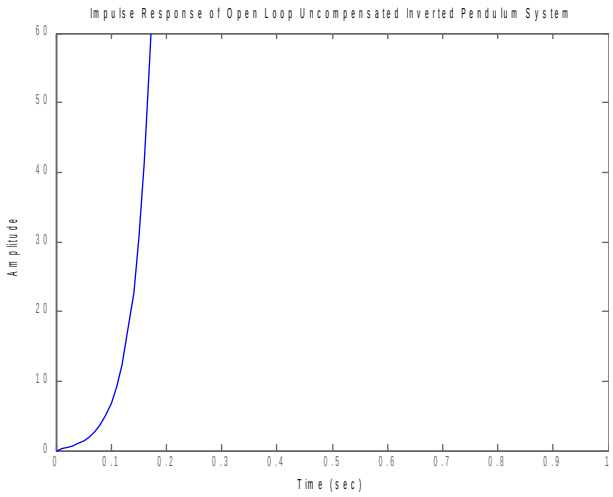


Figura 3.8 – Resposta a um impulso com a função de transferência em circuito aberto.

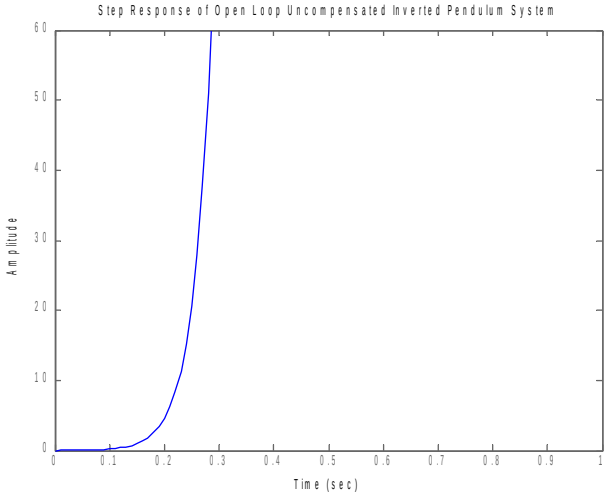


Figura 3.9 – Resposta a um degrau com a função de transferência em circuito aberto.

Existem casos em que um sistema instável em circuito aberto passa a ser estável. Isso ocorre apenas quando o circuito está fechado. O sistema em circuito fechado sem compensação pode ser estudado, olhando o gráfico do *Root Locus* aplicado ao sistema. Veja esse gráfico na figura abaixo.

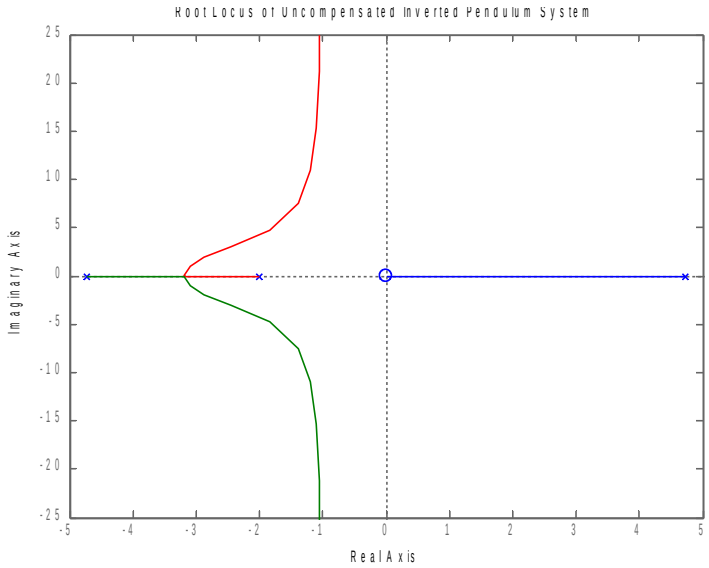


Figura 3.10 – Root Locus do sistema sem compensação.

O gráfico revela que o sistema não pode ser controlado simplesmente utilizando de um circuito fechado unitário. Não importando o valor do ganho aplicado ao circuito fechado, uma parte permanece na região instável do gráfico. Isso torna o sistema impossível de ser controlado utilizando um circuito fechado unitário.

Desconsiderando coeficiente de atrito

Na Figura 3 .11, apresentamos nosso sistema completo levando em consideração a equação(3 .25), respondendo a um impulso como entrada.

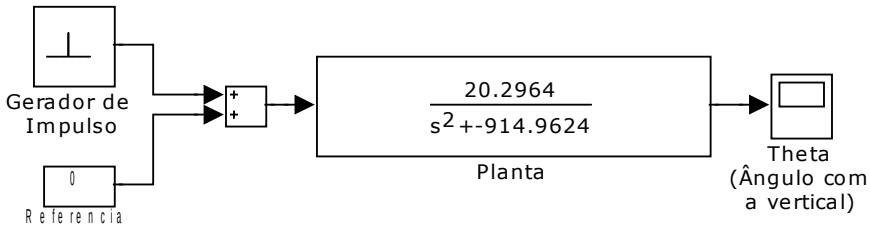


Figura 3.11 – Função de transferência em circuito aberto.

Como já comentamos antes, Figura 3 .12 mostra que o sistema sem compensação é instável, por possuir pólos no lado direito do eixo *s*.

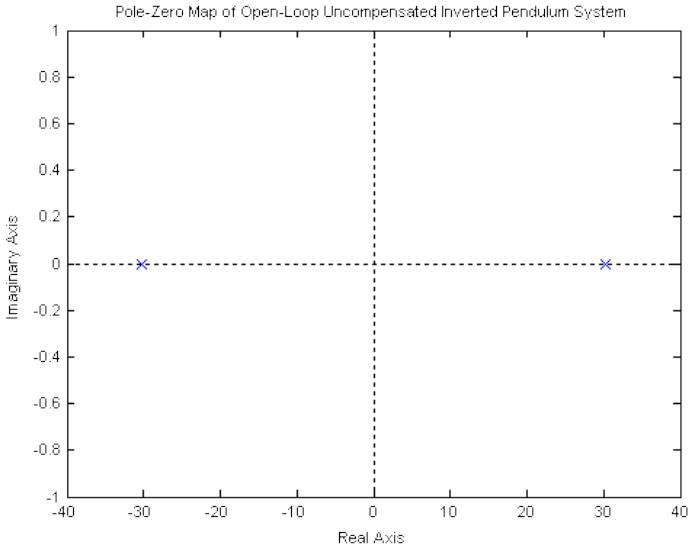


Figura 3.12 – Mapa Pólo-Zero do sistema sem compensação.

Também podemos ver claramente que o sistema é altamente instável através do gráfico da resposta do sistema a um impulso (Figura 3 .13)e a um degrau (Figura 3 .14). O ângulo θ diverge rapidamente.

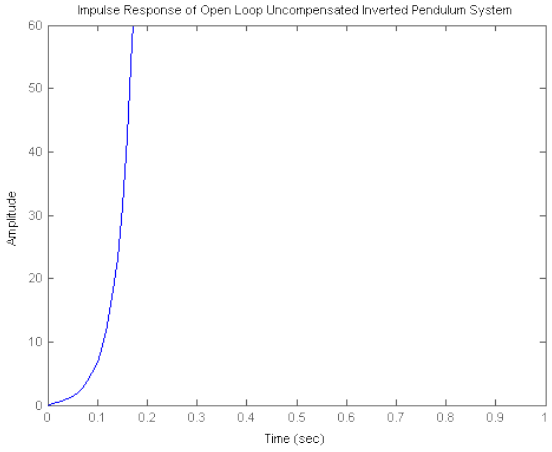


Figura 3.13 – Resposta a um impulso com a função de transferência em circuito aberto.

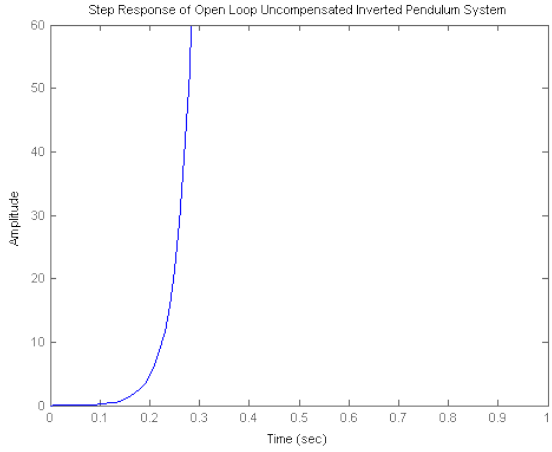


Figura 3.14 – Resposta a um degrau com a função de transferência em circuito aberto.

Existem casos em que um sistema instável em circuito aberto passa a ser estável. Isso ocorre apenas quando o circuito está fechado. O sistema em circuito fechado sem compensação pode ser estudado, olhando o gráfico do *Root Locus* aplicado ao sistema. Veja esse gráfico na figura abaixo.

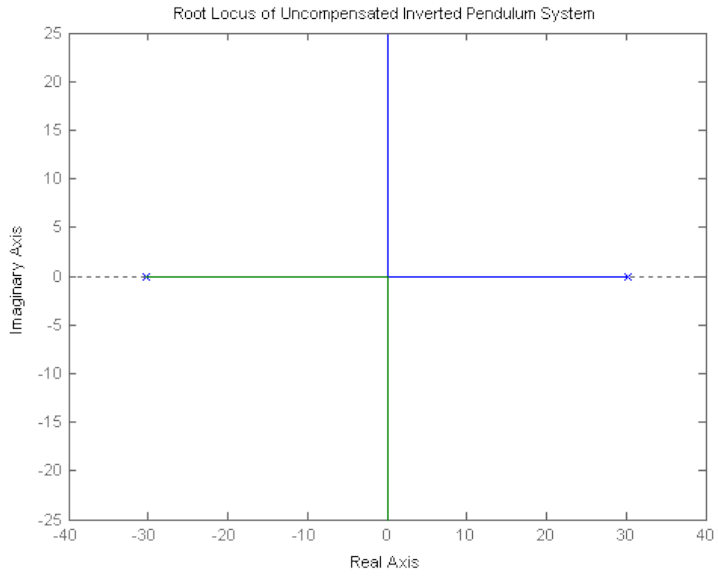


Figura 3.15 – Root Locus do sistema sem compensação.

O gráfico revela que o sistema não pode ser controlado simplesmente utilizando de um circuito fechado unitário. Não importando o valor do ganho aplicado ao circuito fechado, uma parte permanece na região instável do gráfico. Isso torna o sistema impossível de ser controlado utilizando um circuito fechado unitário.

3.3. Desenho de um Compensador

Um compensador é necessário para estabilizar nosso sistema, pois vimos na sessão anterior que, mesmo com um circuito fechado, o sistema permanece instável. Teremos que ajustar nosso sistema para que o *Root Locus* tenha suas raízes no lado esquerdo do plano *s*, colocando-os assim na região estável.

Vamos rever aqui nossos objetivos para o sistema:

- Assentamento da parte transiente: até 1 segundo
- *Overshoot* deve ser menor que 10%
- A razão de *damping* ζ deve ser maior que 0,5
- O erro (ângulo com a normal) deve ser zero quando estiver estabilizado.

Calculamos nosso damping e freqüência natural considerando as formulas abaixo:

$$\text{Percentual de overshoot} = PO = 100e^{-\zeta\pi/\sqrt{1-\zeta^2}} \tag{3.30}$$

$$\text{Tempo de assentamento} = T_s = \frac{4}{\zeta w_n} \tag{3.31}$$

Com isso obtivemos:

$$\begin{aligned} \zeta &= 1{,}3 \\ w_n &= 3{,}1 \end{aligned} \tag{3.32}$$

3.3.1. Compensador Root Locus

Para a manipulação do sistema, introduzindo pólos e zeros, vamos utilizar a ferramenta *SISO* do Matlab 8. Essa é especialmente desenhada para a compensação de sistemas de uma entrada e uma saída. Na figura abaixo temos o modelo de compensador que estaremos utilizando:

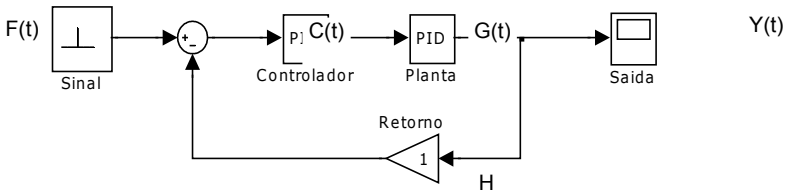


Figura 3.16 – Sistema compensado.

Compensador considerando coeficiente de atrito

Primeiramente, introduzimos um pólo na origem para cancelar o zero. Com isso, temos a seguinte configuração no Root Locus:

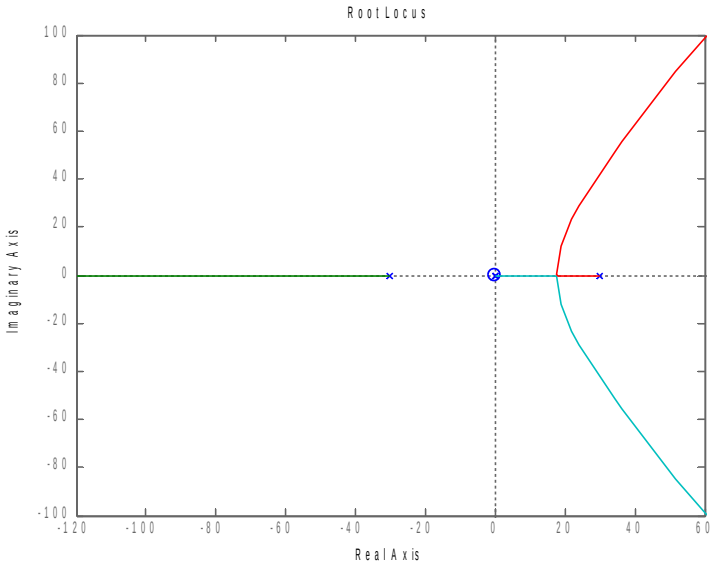


Figura 3.17 – Root Locus do sistema adicionando um pólo em zero.

Temos que adicionar dois zeros para puxar as curvas que estão tendendo para o infinito no lado instável. Colocando dois zeros em -30, temos o seguinte resultado:

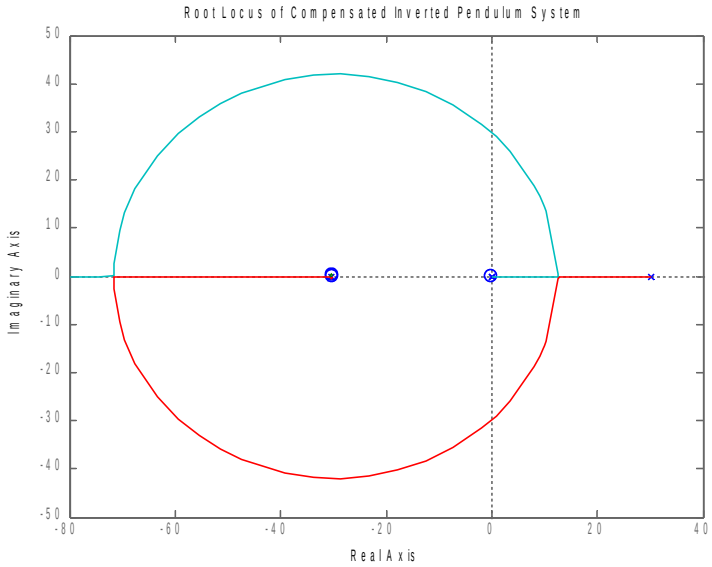


Figura 3.18 – Root Locus do sistema adicionando mais dois zeros em -30.

A equação do compensador encontrado foi:

$$C(s) = 35 \cdot \frac{s^2 + 60s + 900}{s} \tag{3.33}$$

3.4. Controle PID

3.4.1. Introdução ao controle PID

Um controlador PID é constituído por três componentes: um termo proporcional, um integral e um derivativo. Podemos dizer então que sua função de transferência teria a seguinte apresentação:

$$K_P + \frac{K_I}{s} + K_D s = \frac{K_D s^2 + K_P s + K_I}{s} \tag{3.34}$$

Nesse caso, Kp é nossa constante proporcional, Ki é a integral e Kd é a derivativa.

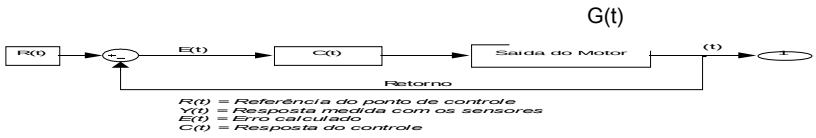


Figura 3.19 – Descrição de um sistema de controle.

Primeiramente, vamos analisar os efeitos de um controle PID em um sistema de circuito fechado, como mostra a Figura 3 .19. A variável E(t) rastreia o erro, mais precisamente, a diferença entre o valor desejado R(t) e a saída real Y(t). Do erro, calculamos sua derivada e integral. Em fim, enviamos para a planta G(t) a soma do erro com seu ganho proporcional Kp, mais a integral do erro com seu ganho integral Ki, mais a derivada do erro com seu ganho Kd.

Para analisarmos os efeitos de cada uma das três constantes sobre o sistema, vamos olhar a função de transferência canônica de segunda ordem de um circuito aberto mostrada abaixo:

$$\frac{1}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{3.35}$$

A constante Kp reduz o tempo de subida e também reduz o erro com a referência, porém sem nunca eliminá-lo. A constante integral Ki tem a capacidade de eliminar o erro com a referência, porém, a resposta transiente será afetada. Se for necessário utilizar uma componente integral no controle, devemos sempre tentar um valor baixo. O controle derivativo irá aumentar a estabilidade do sistema, reduzindo o *overshoot* e melhorando a resposta transiente. Podemos ver um resumo dos efeitos causados pela introdução de um controle PID na tabela abaixo.

Componente	Tempo de subida	*Overshoot*	Tempo de assentamento	Erro c/ R(t)
K_P	Diminui	Aumenta	Não afeta	Diminui
K_I	Diminui	Aumenta	Aumenta	Elimina
K_D	Não afeta	Diminui	Diminui	Não afeta

Tabela 3.2 : Resposta de um sistema a um controle PID.

É claro que essas informações não são exatas, pois como as constantes são relacionadas, um termo afeta o outro. Essa informação é útil caso se queira determinar os valores de Kp, Ki e Kd na tentativa e erro.

3.4.2. Análise do Controlador

Quando comparamos o resultado que encontramos na estabilização do sistema e a equação geral do controle PID, podemos tirar algumas conclusões. Vamos então analisar as equações (3 .33) e (3 .34). Repetimo-las abaixo para efeitos visuais:

$$C(s) = 35 \cdot \frac{s^2 + 60s + 900}{s}$$

$$K_P + \frac{K_I}{s} + K_D s = \frac{K_D s^2 + K_P s + K_I}{s}$$

Com isso, podemos concluir que:

$$C(s) = K \cdot \frac{K_D s^2 + K_P s + K_I}{s} = 35 \cdot \frac{s^2 + 60s + 900}{s}$$

$$\text{com} \begin{cases} K = 35 \\ K_p = 60 \\ K_D = 1 \\ K_I = 900 \end{cases} \tag{3.36}$$

3.4.3. Aplicando o Controlador

É importante termos em mente que o sistema será implementado com diversas limitações, incluindo um microprocessador de baixo custo. Esse é o principal motivo pelo qual devemos utilizar um controlador de simples implementação. A Figura 3 .19 representa o sistema de controle que estaremos utilizando.

Com a referencia R(t) desejada, sendo com θ = 0º, temos o sistema equilibrado. Qualquer valor diferente de 0º representa o erro Y(t) = E(t).

Implementando o sistema de controle PID, temos três termos baseados na medida do erro:

- **Termo Proporcional :** $K_I \cdot E(t)$ – onde K_P é a constante proporcional.
- **Termo Integral :** $K_I \cdot \int_0^t E(t)dt$ – onde K_I é a constante integral.
- **Termo Derivativo :** $K_D \cdot dE(t)/dt$ – onde K_D é a constante derivativa.

Juntando os termos, temos a seguinte equação:

$$C(t) = K_P \cdot E(t) + K_I \cdot \int_0^t E(t)dt + K_D \cdot {E(t)}/{dt} \tag{3.37}$$

Nesse sistema, o sinal de saída do controlador irá definir a direção de rotação do motor. A magnitude de C(t) corresponde diretamente à magnitude do sinal enviado ao motor, definindo a velocidade que o motor irá rodar.

Outra representação para essa equação pode ser encontrolada abaixo:

$$C(t) = K \cdot \left[E(t) + \frac{1}{T_i} \int E(t)dt + T_d \frac{dE(t)}{dt} \right]$$

$$com \begin{cases} K_p = K \\ K_I = \dfrac{K}{T_i} \\ K_D = K \cdot T_d \end{cases} \tag{3.38}$$

3.4.4. Integral *Windup*

Ao representarmos um controle PID é comum basear-lo no conceito de linearidade, mas temos que levar em consideração os componentes não lineares do sistema real, como o motor. A tensão enviada para o motor tem limites e isso significa que temos que considerar a saturação do sinal enviado ao motor 8. Se nenhuma ação é tomada no controle em relação aos limites do atuador, este chegará ao limite, independente da saída do processo. Na componente integral de nosso controle, o sinal continuará a ser integrado, ficando excessivamente grande, ou *wind up*. O erro deve mudar de sinal durante um tempo prolongado para o sinal voltar ao normal. Isso ocasiona um transiente muito grande na saturação, podendo até instabilizar o sistema.

Para evitar o efeito *Windup*, adicionaremos mais uma realimentação no controlador utilizando como erro e_s a diferença do sinal de saída do atuador e a saída do sinal do controle. O sinal e_s alimenta o integrador já existente através do sinal $1/T_t$. enquanto não há saturação no sinal do controlador, o erro e_s é zero, mas quando o sinal satura, o erro e_s cresce. A realimentação faz com que esse erro volte a ser zero.

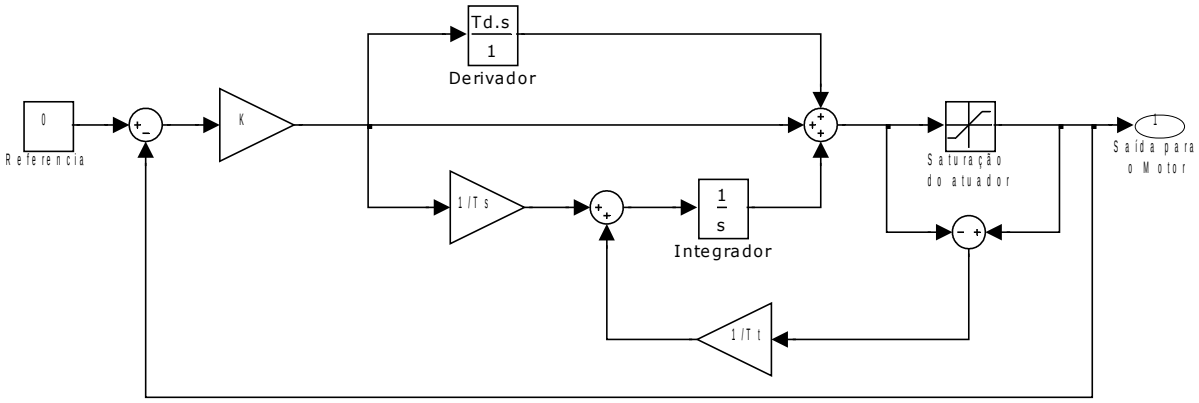


Figura 3.20 – Controle PID com componente integral *Windup*.

Em termos práticos, os limites de nosso atuador estão no sinal para o motor. Possuímos uma velocidade máxima para cada lado, diretamente relacionada à tensão. Em nosso caso, controlamos a tensão utilizando um sinal modulado por pulsos e um filtro passa baixas (próprio motor). Por isso, os limites de nosso atuador são o menor pulso e o mais largo pulso que pudermos emitir.

Com isso temos uma nova formula para o controlador mostrado abaixo:

$$C(t) = K \cdot \left[E(t) + \left(\frac{1}{T_i} + \frac{1}{T_t} e_s(t) \right) \int E(t)dt + T_d \frac{dE(t)}{dt} \right] \tag{3.39}$$

Para facilitar o *tunning* das constantes, é aconselhado que o valor de T_t seja igual a T_i 8.

3.4.5. Discretização para um Sistema Digital

Programaremos o sistema de controle em um micro controlador, e algumas aproximações serão necessárias devido à implementação dos termos integral e derivativo.

Termo Proporcional

Não há necessidade de uma aproximação no termo proporcional. O resultado é:

$$P = K \cdot E(t) \tag{3.40}$$

Termo Integral

A seguinte equação pode ser usada para o termo integral:

$$I(t) = K\left(\frac{1}{T_i} - \frac{1}{T_t}e_s\right)\cdot\int_0^t E(t)dt \approx K\left(\frac{1}{T_i} - \frac{1}{T_t}e_s\right)T_S\sum_0^N E(n) \tag{3.41}$$

fazendo $T_t = T_i$

$$I(k) = \frac{K}{T_i}\cdot(1 - e_s)\cdot T_S\sum_0^N E(n) \tag{3.42}$$

Aqui T_S é o período de amostragem.

Outro método para se fazer a discretização do sinal é derivando os dois lados da equação (3 .41). A seguinte equação pode ser usada para o termo integral:

$$I(t) = K\left(\frac{1}{T_i} - \frac{1}{T_t}e_s\right)\cdot\int_0^t E(t)dt = \frac{K}{T_i}\cdot(1 - e_s)\cdot\int_0^t E(t)dt \tag{3.43}$$

$$\frac{dI(t)}{dt} = \frac{K}{T_i}\cdot(1 - e_s)\cdot E(t) \tag{3.44}$$

Após discretizarmos:

$$\frac{I_{k+1} - I_k}{T_S} = \frac{K}{T_i}\cdot(1 - e_{s_k})\cdot E_k \tag{3.45}$$

$$I_{k+1} = I_k + \frac{K}{T_i\cdot F_S}\cdot(1 - e_{s_k})\cdot E_k \tag{3.46}$$

Termo Derivativo

Para o termo derivativo, podemos utilizar a seguinte equação:

$$D(t) = K\cdot T_D\cdot {dE(t)}/{dt} \tag{3.47}$$

$$D(n) = K\cdot T_D\cdot\frac{[E(n) - E(n-1)]}{T_S} \tag{3.48}$$

Temos E(n) como o erro atual, e E(n-1) como o erro anterior.

Considerando as aproximações nas últimas duas equações, podemos reescrever C(t) como mostra a equação (3 .49).

$$\boxed{C(t) = K_P\cdot E(t) + K_I\cdot(1 - e_s)\cdot T_S\sum_0^N E(n) + K_D\cdot[E(n) - E(n-1)]/T_S} \tag{3.49}$$

4. MECÂNICA E ESTRUTURA

Neste capítulo, veremos como foi elaborada a estrutura do robô, assim como toda a parte mecânica necessária para a implementação. Isso inclui também a escolha dos motores e o cálculo das engrenagens.

4.1. Estrutura do Robô

Para o desenvolvimento da estrutura do robô, foi necessário que projetássemos peça por peça, pois não tínhamos um produto semelhante no mercado que pudéssemos reaproveitar. Para isso, foi utilizado o programa *SolidWorks*. Este software permite a fácil modelagem e fornece a estimativa de medidas úteis para nosso algoritmo de controle, como o momento de inércia, centro de gravidade e massa entre outras.

A estrutura é constituída de duas partes: a base inferior, na qual ficam os motores e onde é fixado o eixo de rotação das rodas, e a parte superior, na qual prateleiras são colocadas para acomodar as placas de circuito impresso e as baterias.

A base inferior é constituída por três mancais, de alumínio, presos à placa principal inferior. Os mancais são utilizados para fixar quatro rolamentos, dois para cada eixo (são dois eixos independentes, um para cada roda). Também presa a placa principal, temos duas cantoneiras utilizadas para fixar os dois motores. Veja na Figura 4 .21 a base inferior vista por baixo.

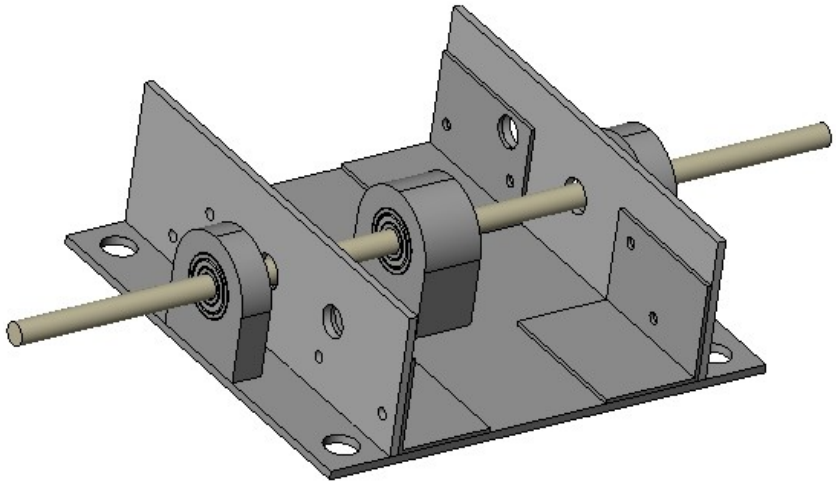


Figura 4.21 – Base inferior vista por baixo.

Para fixarmos as rodas ao eixo de rotação, tivemos que criar buchas que o encaixe fosse perfeito. Após conectarmos os motores e as rodas de 15 cm de diâmetro, a base inferior ficou como mostra a figura abaixo.

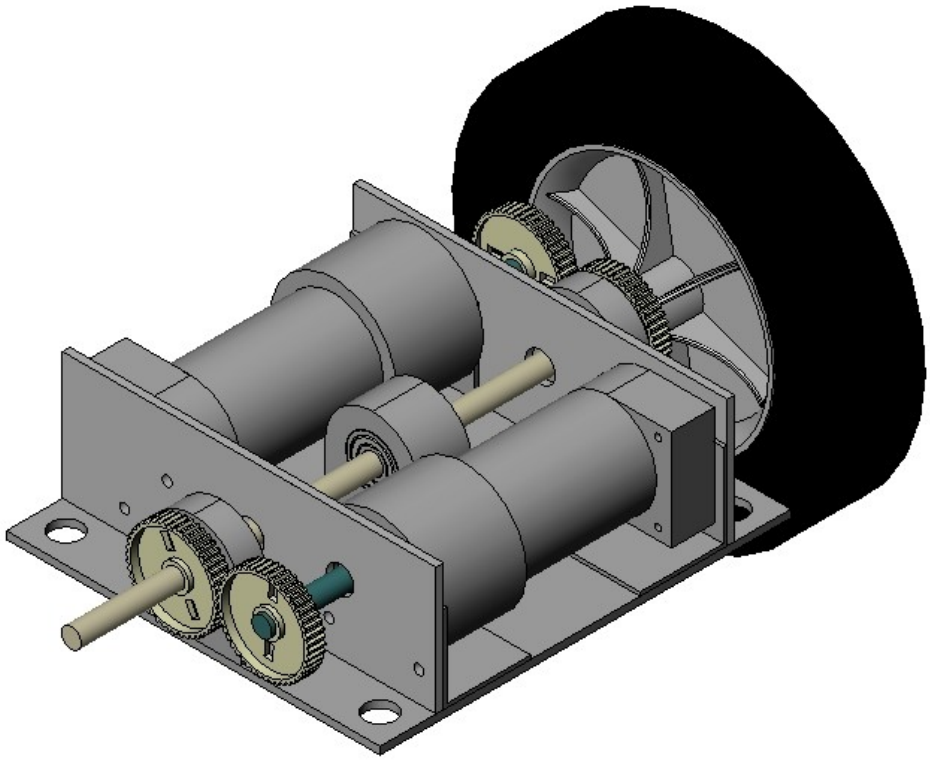


Figura 4.22 – Base inferior com os motores.

A parte superior da estrutura é constituída de quatro pilares que sustentam quatro prateleiras. A prateleira inferior é utilizada para acomodar o sensor de inclinação. Por estar no ponto mais baixo, esse será menos suscetível aos deslocamentos que ocorrem no topo do robô. A segunda prateleira é utilizada para acomodar a placa de circuito impresso principal. A terceira e quarta prateleiras são utilizadas para acomodar as baterias. Colocando-as no ponto mais alto, elevamos o centro de gravidade da estrutura, acarretando uma maior sensibilidade ao controle. Utilizamos o acrílico como material para fazer as prateleiras.

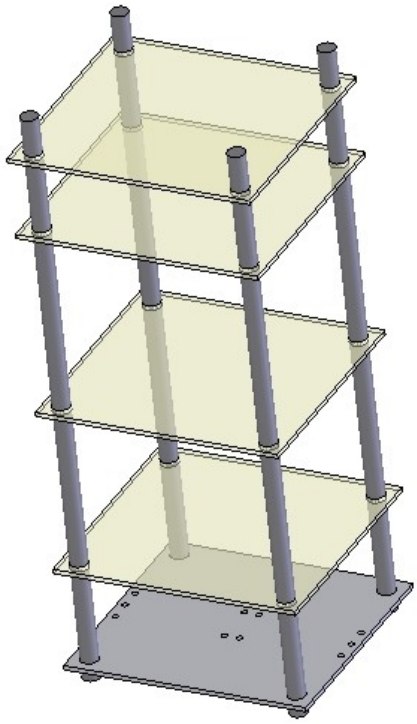


Figura 4.23 – Parte superior da estrutura.

Ao juntarmos todas as peças, a estrutura fica com o aspecto mostrado na Figura 4 .24

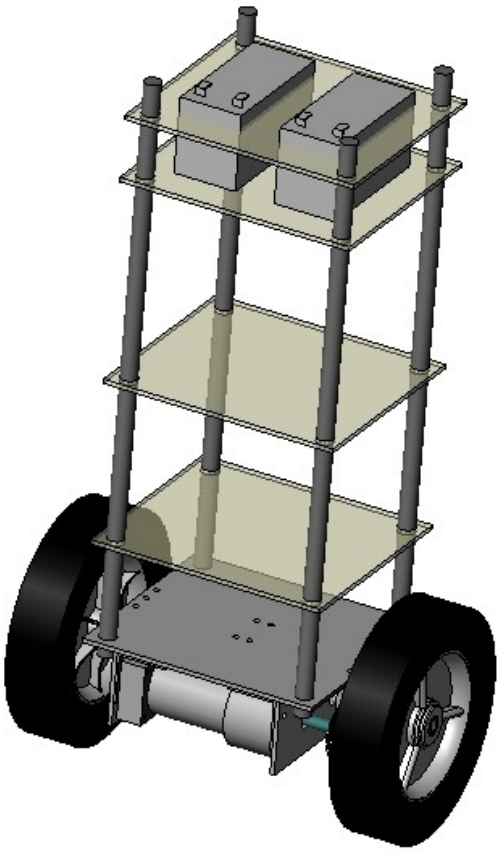


Figura 4.24 – Estrutura completa do robô de equilíbrio.

Configurando corretamente as densidades dos materiais utilizados na construção da estrutura, o programa *SolidWorks* é capaz de gerar um relatório completo da estrutura criada. Veja nas tabelas seguintes esse relatório.

Massa Total	4594,92 g
Massa de cada Roda	293g
Volume	2095,07 cm^3
Densidade	2,19 g/cm^3
Área de superfície	7380,45 cm^2

Tabela 4.3 – Medições básicas da estrutura.

O ponto de origem da estrutura foi configurado como sendo exatamente na linha do eixo, entre as duas rodas. Assim, os cálculos gerados no relatório estarão sempre relacionados ao eixo do robô.

Eixo de Referência	**Valor**
X	0,00 cm
Y	15,33 cm
Z	0,00 cm

Tabela 4.4 : Centro de Massa da estrutura.

Esses são os principais eixos de inércia e principais momentos de inércia, retirados do centro de massa:

Eixos de Inércia	
I_x	(0.00, 1.00, 0.00)
I_y	(-0.01, -0.00, 1.00)
I_z	(1.00, -0.00, 0.01)

Momentos de Inércia	
P_x	281196,83 $g.cm^2$
P_y	1559462,93 $g.cm^2$
P_z	1699485.51 $g.cm^2$

Tabela 4.5 : Eixos de inércia e Momentos de inércia da estrutura.

Os momentos de inércia, retirados no centro de massa e alinhados com a saída do sistema de coordenadas, podem ser vistos na Tabela 4 .6.

L_{xx}	1699470,39 $g.cm^2$	L_{xy}	2805,68 $g.cm^2$	L_{xz}	-1156,77 $g.cm^2$
L_{yx}	2805,68 $g.cm^2$	L_{yy}	281202,46 $g.cm^2$	L_{yz}	328,30 $g.cm^2$
L_{zx}	-1156,77 $g.cm^2$	L_{zy}	328,30 $g.cm^2$	L_{zz}	1559472,42 $g.cm^2$

Tabela 4.6 : Momentos de inércia no centro de massa.

Os momentos de inércia, retirados na saída do sistema de coordenadas, podem ser vistos na Tabela 4 .7.

I_{xx}	2779588,42 g.cm^2	I_{xy}	4728,54 g.cm^2	I_{xz}	-1156,51 g.cm^2
I_{yx}	4728,54 g.cm^2	I_{yy}	281205,91 g.cm^2	I_{yz}	473,80 g.cm^2
I_{zx}	-1156,51 g.cm^2	I_{zy}	473,80 g.cm^2	I_{zz}	2639593,85 g.cm^2

Tabela 4.7 : Momentos de inércia na saída do sistema de coordenadas.

4.2. Motores e Engrenagens

Com a escolha do material e de posse do peso dos componentes, o programa *SolidWorks* fornece as medidas necessárias para o cálculo do torque. Com isso, podemos calcular o fator de redução das engrenagens.

A Tabela 4 .8 mostra alguns dados do motor retirados de sua especificação 8.

torque	36,5x10^{-3} Nm/A
corrente máxima	8 A
caixa de redução	19,1:1

Tabela 4.8 : Dados do motor.

O torque máximo ocorre quando o robô está com a inclinação máxima. Como visto no capítulo III, a inclinação máxima que estaremos considerando foi estipulada em ±15º, visto que, para linearizar o modelo, foi tomado como base ângulos pequenos com referência a vertical. Assim temos:

$$\tau_{\max} = M \cdot g \cdot sen(15) \cdot l \tag{4.50}$$

M é a massa da estrutura, *g* é a aceleração da gravidade e *l* é a distância entre o centro de massa e o eixo. Substituindo os valores, temos:

$$\tau_{\max} = 4{,}595 \cdot 9{,}8 \cdot sen(15) \cdot 0{,}1533 \tag{4.51}$$

$$\boxed{\tau_{\max} = 1{,}79\,Nm} \tag{4.52}$$

Devido à limitação do componente eletrônico que fornece potência aos motores, estipulamos uma corrente máxima de 1,5 A para cada motor. Assim temos:

$$\tau_{motor} = 1{,}5 \cdot 36{,}5 \times 10^{-3}\,Nm \tag{4.53}$$

Como temos 2 motores, totalizamos:

$$2 \times \tau_{motor} = 109{,}5 \times 10^{-3}\,Nm \tag{4.54}$$

Como o torque aplicado às rodas precisa ser maior que o torque máximo (τ_{max}) calculado anteriormente, temos:

$$2\times \tau_{motor} \cdot k \geq \tau_{\max} \tag{4.55}$$

$$2\times \tau_{motor} \cdot k \geq 1{,}79Nm \tag{4.56}$$

$$k \geq \frac{1{,}79}{109{,}5\times 10^{-3}} \tag{4.57}$$

$$\boxed{k \geq 16{,}3} \tag{4.58}$$

Aqui k é o fator de redução das engrenagens. Como o motor vem acoplado a uma caixa de redução de 19,1:1, não será necessário fazer a redução nas engrenagens que transferem o torque do motor para a roda. Com isso, estaremos utilizado engrenagens com a razão entre elas de 1:1.

5. IMPLEMENTAÇÃO DO SISTEMA

Após termos modelado nosso sistema do um robô de equilíbrio e montado nossa estrutura mecânica, precisamos focar nossas atenções à implementação do sistema. Neste capítulo veremos todos os componentes utilizados para o funcionamento do sistema.

A Figura 5 .25 mostra o diagrama de blocos da solução.

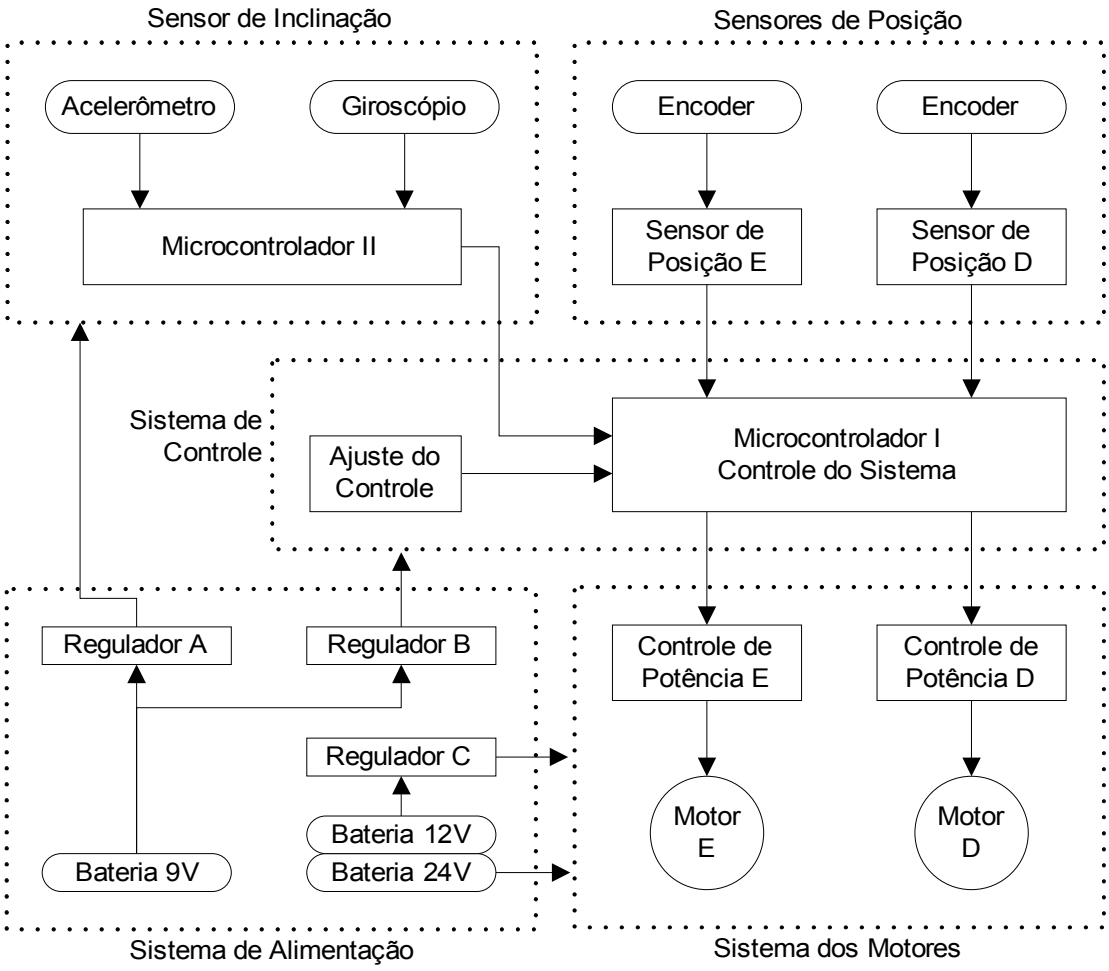


Figura 5.25 – Diagrama de blocos.

Como podemos ver, temos quatro blocos principais em nossa solução: sensor de inclinação, sensores de posição, sistema dos motores e – o bloco principal – sistema de controle. Vamos falar primeiro dos sensores que alimentam o sistema de controle, depois do sistema dos motores, em seguida o sistema de controle e por último o sistema de alimentação.

5.1. Sensor de Inclinação

Um dos estados que estaremos monitorando é o ângulo θ que o robô faz com a vertical (denominado ângulo de *pitch*), assim como sua velocidade angular. Para adquirirmos essa entrada, temos que utilizar um sensor que capture essa informação em tempo real, pois ela será utilizada para realimentar o controlador. Os requisitos para esse sensor são:

- Precisão de 0,5º;
- Resposta rápida, acima de 50 leituras por segundo;
- Sem sujeição a erro com o tempo;
- Sem sujeição a erro com movimentos.

Como não foi possível encontrar um único sensor que satisfizesse os requisitos necessários para o projeto, decidimos que mais de um sensor deveriam ser utilizados em conjunto para adquirir o ângulo de inclinação do robô. Para essa tarefa, dois sensores foram escolhidos: um acelerômetro e um giroscópio. Ambos estarão sendo descritos separadamente nessa seção.

Os sinais gerados por esses sensores serão capturados separadamente por um micro-controlador. Este será responsável por tratá-los para que possam ser utilizados em conjunto. O método utilizado para juntar as saídas dos dois sensores e gerar a inclinação em tempo real será através de um filtro de Kalman.

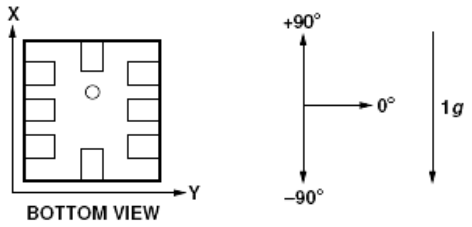
5.1.1. Acelerômetro

Um acelerômetro, sendo posicionado para medir a aceleração na vertical, é capaz de captar a aceleração da gravidade. Girando-o em torno de um eixo na horizontal, este será capaz de captar apenas uma fração da aceleração da gravidade, conseguindo distinguir o quanto o sensor está inclinado.

Medir a aceleração da gravidade é um meio eficaz de se obter um ângulo *pitch*. Colocamos dois acelerômetros defasados em 90º em um plano transversal ao eixo de rotação que desejamos medir e conseguimos obter, também, a direção para a qual está girando o sensor.

Uma das principais aplicações do acelerômetro ADXL202E é justamente a medição de inclinação. Esse componente já possui dois acelerômetros defasados em 90º, o que facilita significativamente o processo. O sensor está em sua posição mais sensível em relação à inclinação quando posicionado perpendicular à força da gravidade. Isto é, quando está

paralelo à superfície da terra. Quando o sensor está posicionado no eixo da gravidade, isto é, fazendo uma leitura de +1g ou -1g, a diferença na aceleração captada por graus não é significante 8. Veja na figura abaixo as mudanças nos eixo X e Y do acelerômetro em relação à sua posição com o eixo de gravidade.



X Axis Orientation to Horizon (°)	X Output		Y Output (g)	
	X Output (g)	Δ per Degree of Tilt (mg)	Y Output (g)	Δ per Degree of Tilt (mg)
–90	–1.000	–0.2	0.000	17.5
–75	–0.966	4.4	0.259	16.9
–60	–0.866	8.6	0.500	15.2
–45	–0.707	12.2	0.707	12.4
–30	–0.500	15.0	0.866	8.9
–15	–0.259	16.8	0.966	4.7
0	0.000	17.5	1.000	0.2
15	0.259	16.9	0.966	–4.4
30	0.500	15.2	0.866	–8.6
45	0.707	12.4	0.707	–12.2
60	0.866	8.9	0.500	–15.0
75	0.966	4.7	0.259	–16.8
90	1.000	0.2	0.000	–17.5

Figura 5.26 – Resposta dos eixos X e Y a mudanças na inclinação 8.

O acelerômetro ADXL202E possui uma relação entre o grau de precisão de suas medidas com a voltagem utilizada, a largura de banda analógica e a freqüência de aquisição de dados. Veja na tabela abaixo os parâmetros utilizados para a nossa aplicação.

Voltagem de alimentação	5,0	V
Largura de banda analógica	25	Hz
Freqüência de aquisição de dados	54	Leituras por segundo
Resolução (em g's)	0,013	G
Resolução (graus de inclinação)	0,74	° de inclinação
T2	3,12963	mS
Temperatuda máxima (T_{max})	40	ºC
Temperatuda mínima (T_{min})	15	ºC
Alteração em zero g com T_{max}	0,03	G
Alteração em zero g com T_{min}	0,02	G

Tabela 5.9 – Parâmetros para o acelerômetro.

Detalhes sobre a aquisição desses dados, assim como o método para calcular o valor dos componentes eletrônicos utilizados para configurar o acelerômetro, estão descritos no Anexo 2.

O componente ADXL202E possui duas saídas, uma para cada eixo de aceleração medido. O sinal é codificado em largura de pulsos. Portanto, podemos fazer medidas regulares com um intervalo de tempo preciso. Para uma largura de pulso igual à metade do período de oscilação (T/2), temos uma gravidade equivalente a zero. Para uma largura de pulso igual a T, temos a aceleração igual a duas vezes aceleração da gravidade (2*g). Para a largura de pulso igual a zero, temos a aceleração igual a menos duas vezes a aceleração da gravidade (-2*g). Por isso, para capturarmos o valor dos acelerômetros, utilizamos o módulo de captura de largura de pulso do micro-controlador. Veja na figura abaixo um exemplo do sinal emitido pelo sensor.

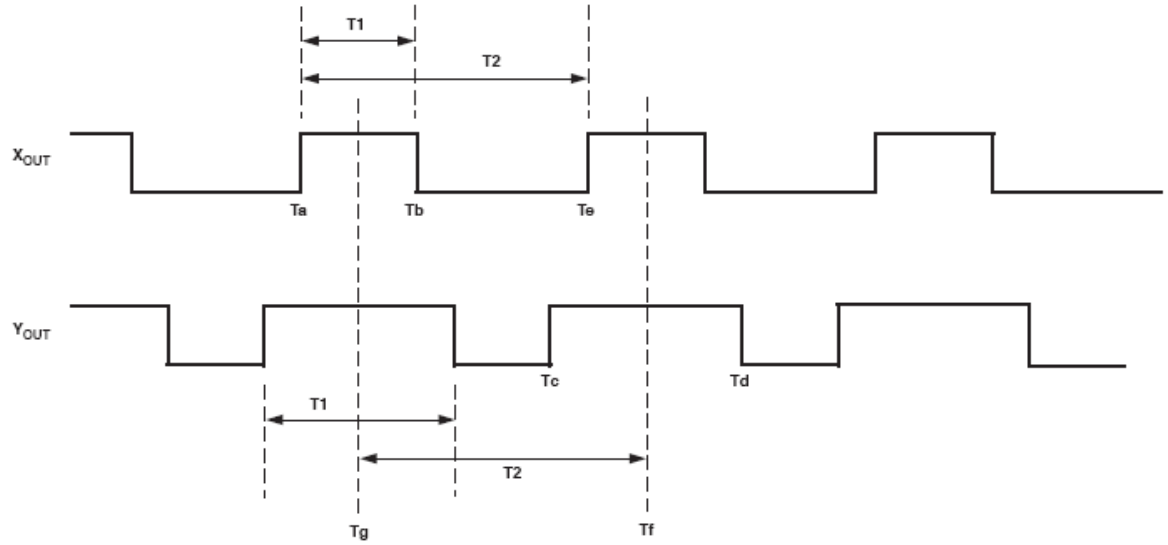


Figura 5.27 – Exemplo de sinal emitido pelo acelerômetro 8.

O método para calcular o ângulo de inclinação com a vertical, utilizando os sinais dos acelerômetros transversais, é bem simples. Podemos considerar que os sensores estão posicionados em um círculo trigonométrico. Cada um dos sensores é o seno ou co-seno do ângulo desejado. Então, para descobrimos um ângulo θ, quando temos um seno e um co-seno, basta calculamos o arco, cuja tangente é o seno sobre o co-seno. Com isso chegamos à equação (5 .59).

$$\theta_m = arctg\left(\frac{sen\theta}{\cos\theta}\right) = arctg\left(\frac{\text{aceleração eixo } X}{\text{aceleração eixo } Z}\right) \quad (5.59)$$

Temos, ainda, a vantagem de não precisar escalonar o valor medido da aceleração, pois teremos um sinal dividido pelo outro e o fator de escalonamento seria cancelado. O resultado obtido já é escalonado em radianos.

Infelizmente, para nossa aplicação, esse tipo de sensor também tem suas desvantagens. Um acelerômetro é sensível à aceleração estática, mas também é sensível à aceleração dinâmica. Isso significa que o sinal gerado por esse sensor é impreciso, pois além da aceleração da gravidade, é medida também a aceleração do veículo.

Para resolver esse problema, estaremos utilizando esse sensor junto com um outro, o giroscópio, que é detalhado na sessão 5.1.2.

5.1.2. Giroscópio

Giroscópios são utilizados para medir a velocidade angular, ou seja, o quão rápido um objeto gira. A rotação é normalmente medida em relação a um dos eixos de referência: *yaw*, *pitch* ou *roll*. No nosso caso, estaremos referenciando o *pitch* 8.

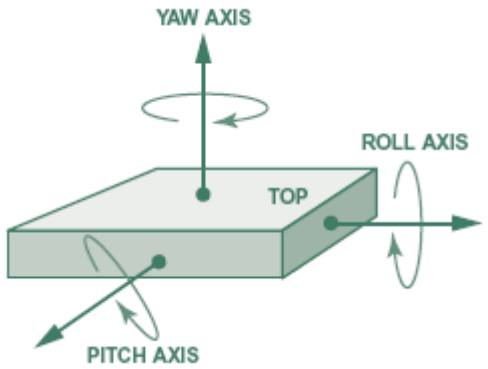


Figura 5.28 – Eixos de rotação 8.

O giroscópio selecionado para esse projeto foi o *ADXRS150*, um dispositivo de baixo custo e que já está integrado com a eletrônica necessária para seu funcionamento. O sinal de saída, *RATEOUT*, é uma tensão proporcional à velocidade angular sob o eixo normal à superfície do dispositivo 8. Esta deverá ser capturada pelo microcontrolador utilizando um módulo de conversão analógico-digital (A/D).

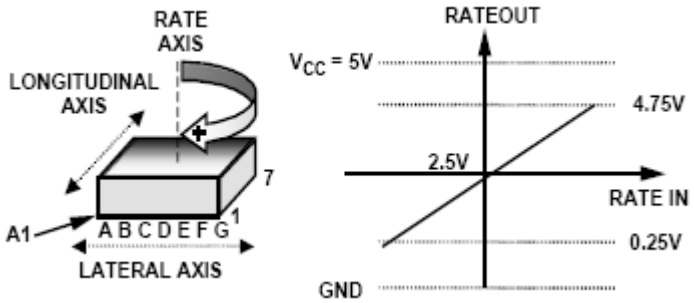


Figura 5.29 – Sinal *RATEOUT* aumenta com a rotação sentido horário 8.

Veja na figura abaixo o sinal de saída do giroscópio em três configurações diferentes: parado, movimentando-se para um lado e movimentando-se para o lado oposto.

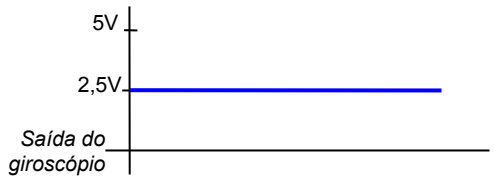


Figura 5.30 – Sinal *RATEOUT* com o giroscópio parado.

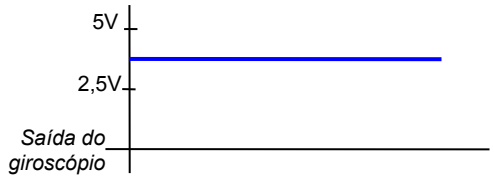


Figura 5.31 – Sinal *RATEOUT* com o giroscópio rodando sentido anti-horário.

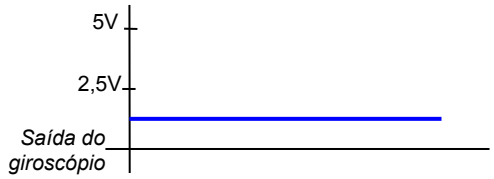


Figura 5.32 – Sinal *RATEOUT* com o giroscópio rodando sentido horário.

Para acharmos o ângulo de uma velocidade angular é necessária a integração do sinal no tempo. Isso introduz um erro no cálculo do ângulo, que assim aumentará exponencialmente. Para evitar um efeito "avalanche" com o valor obtido, estaremos integrando essa medida junto com o resultado do acelerômetro em um filtro de Kalman. Este será detalhado na seção seguinte.

5.1.3. Filtro de Kalman

O Filtro de Kalman é um dos mais conhecidos filtros e uma freqüentemente usada ferramenta para a estimação estocástica de medidas de sensores ruidosos. Esse é essencialmente um conjunto de equações matemáticas que implementam um estimador do tipo preditor-corretor, que pode ser considerado *ótimo* por minimizar a covariância do erro estimado.

Como nossas medidas só estão disponíveis em momentos discretos de tempo, vamos utilizar o Filtro de Kalman Discreto.

Os principais motivos de termos selecionado esse filtro para nosso problema, foi sua já extensiva utilização em diversos sensores, sua fácil implementação em um sistema real e a

robustez que ele apresenta. Nosso objetivo aqui não é descrever toda a matemática envolvida no filtro, pois essa pode ser vista nas referências desse trabalho 8. Vamos nos limitar somente à parte essencial.

Algorítmo do Filtro

O filtro de Kalman estima um processo utilizando um tipo de controle realimentado: o filtro estima o estado em um determinado momento e obtêm um retorno na forma de medidas, calculando-se assim um erro. Com isso, podemos resumir o filtro em dois processos: atualização no tempo e atualização por medidas. O processo de atualização no tempo é responsável por fazer uma projeção no tempo – estimação – do estado atual e da covariância do erro, obtendo a estimação *a priori* para a próxima chamada. O processo de atualização por medidas é responsável pelo retorno – realimentação – incorporando uma nova medida à estimação a *priori*, obtendo uma mais precisa estimativa chamada de *a posteriori*.

Podemos então considerar o processo de atualização no tempo como as equações de um *preditor*, enquanto que o processo de atualização por medidas pode ser considerado como as equações de um *corretor*. O algoritmo *preditor-corretor* pode ser exposto da seguinte forma:

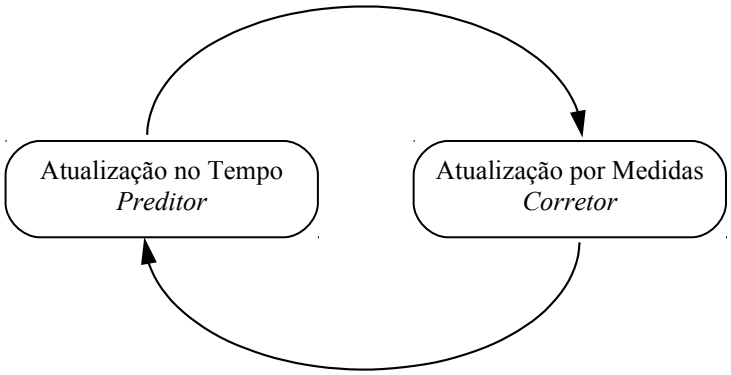


Figura 5.33 – Ciclo do filtro de Kalman.

As equações de cada um desses processos pode ser encontrada abaixo:

$$preditor \begin{cases} \hat{x}_k^- = A\hat{x}_{k-1} + Bu_k & (5.60) \\ P_k^- = AP_{k-1}A^T + Q & (5.61) \end{cases}$$

$$corretor \begin{cases} K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} & (5.62) \\ \hat{x}_k = A\hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) & (5.63) \\ P_k = (I - K_k H) P_k^- & (5.64) \end{cases}$$

Aqui $\hat{x}_k^-$ é nosso estado estimado *a priori* no passo *k*; e $\hat{x}_k$ é nosso estado estimado *a posteriori*. A matriz *A* relaciona o estado entre o momento anterior e o atual. A matriz *B* relaciona o controle opcional *u* ao estado *x*. A matriz *H* relaciona o estado à medida z_k. A covariância do ruído no processo é representada por *Q*, e a covariância do ruído na media é representada por *R*. Temos também P_k^- representando a covariância da estimativa do erro *a priori*, e P_k representando a covariância da estimativa do erro *a posteriori*. Por último, temos a matriz K sendo o ganho que minimiza a covariância do erro *a posteriori*.

Após cada passo de atualização de tempo e medida, o processo se repete novamente, fazendo uma estimação dos valores *a priori* seguintes, utilizando o último *a posteriori* encontrado. Essa recursividade no algoritmo faz com que sua implementação seja viável; o filtro de Kalman condiciona recursivamente a estimação atual sobre todas as medidas anteriores.

Aplicando ao nosso projeto

Utilizar o filtro de Kalman para calcular um ângulo de inclinação, juntando um sensor de aceleração de dois eixos e um giroscópio de eixo único que meça a velocidade angular é um caminho conciso. Os dois sensores são fundidos utilizando o filtro de Kalman com dois estados: um sendo o ângulo e o outro o erro sistemático do giroscópio. No final veremos que o erro depositado pelo giroscópio é automaticamente cancelado pelo filtro.

Primeiramente, vamos definir nossas variáveis que são utilizadas pelo filtro. Como comentado, temos dois estados a serem monitorados: o ângulo, θ, e o erro na medida do giroscópio, *e*. Nosso vetor de estado fica do seguinte modo:

$$X_k = \begin{bmatrix} \theta \\ e \end{bmatrix}, \text{ com } X_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (5.65)$$

Temos nossa matriz de covariância da estimativa do erro *P*, que será quadrada e de 2ª ordem por estarmos monitorando dois estados. Essa será atualizada a cada passo no tempo,

para determinar a acuracidade do rastreamento dos estados. Deve ser iniciada com a matriz Identidade.

$$P_0 = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.66}$$

A matriz 2x2 Q representa a covariância do ruído do processo, indicando, em nosso caso, quanto confiamos no sinal dos acelerômetros em relação ao giroscópio. A matriz 1x1 R representa a covariância do ruído da medida, indicando a vibração esperada na medida de nossos acelerômetros. Os valores escolhidos encontram-se abaixo (lembramos que os ângulos são em radianos).

$$Q = \begin{bmatrix} 0{,}001 & 0 \\ 0 & 0{,}003 \end{bmatrix} \tag{5.67}$$

$$R = [0{,}3] \tag{5.68}$$

Teremos dois processos – duas funções – sendo executadas para a implementação do filtro de Kalman: o *state update*, nosso *preditor*, e o *kalman update*, nosso *corretor*. Vamos falar da implementação dos dois processos separadamente abaixo.

State Update

Este processo representa nosso *preditor* e será chamado em um intervalo de tempo preciso, pois a entrada será a saída do giroscópio ainda com o erro de sua medida. O ângulo atual e a velocidade estimada serão atualizados. Lembramos que nossa entrada já deve estar na escala correta, isso é, em radianos por segundo. Porém não é necessário filtrar nada do erro na medida já que isso será feito pelo filtro.

Sendo nosso estado, mostrado em (5 .65), iremos avançar nossa estimativa de próximo estado – próximo ângulo, da equação (5 .60),Para isso, utilizaremos a derivada de nosso estado atual e o intervalo de tempo *dt,* que representa o intervalo de tempo no qual o processo é executado.

$$\dot{X}_k = \begin{bmatrix} d(\theta) \\ d(e) \end{bmatrix} = \begin{bmatrix} giroscópio - erro \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ 0 \end{bmatrix} \tag{5.69}$$

$$\begin{aligned} x_k^- &= x_{k-1} + \dot{x}_k^- \cdot dt \\ \theta_k^- &= \theta_{k-1} + \dot{\theta}_k \cdot dt \end{aligned} \tag{5.70}$$

Para a segunda parte do processo de *preditor*, temos a matriz de covariância sendo atualizada, mostrado na equação (5 .61). A matriz A, 2x2, é nossa Matriz Jacobiana de X^- em relação aos estados.

$$A = \begin{bmatrix} \frac{\partial(\dot{\theta})}{\partial\theta} & \frac{\partial(\dot{\theta})}{\partial e} \\ \frac{\partial(\dot{e})}{\partial\theta} & \frac{\partial(\dot{e})}{\partial e} \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} \tag{5.71}$$

Para computar a derivada da matriz de covariância P, mostrada em (5 .66), para o próximo estado, vamos necessitar otimizar os cálculos previamente. Temos então nossa matriz de covariância para o próximo estado que será:

$$\dot{P}_k^- = \begin{bmatrix} Q(0,0) - P_{k-1}(0,1) - P_{k-1}(1,0) & -P_{k-1}(1,1) \\ -P_{k-1}(1,1) & Q(1,1) \end{bmatrix} \tag{5.72}$$

$$P_k^- = \begin{bmatrix} P_{k-1}(0,0) + \dot{P}_k^-(0,0)\cdot dt & P_{k-1}(0,1) + \dot{P}_k^-(0,1)\cdot dt \\ P_{k-1}(1,0) + \dot{P}_k^-(1,0)\cdot dt & P_{k-1}(1,1) + \dot{P}_k^-(1,1)\cdot dt \end{bmatrix} \tag{5.73}$$

Mais uma vez lembramos que esse algoritmo deve ser implementado em um microprocessador de baixo custo, portanto o código deve ser devidamente simplificado, evitando algumas contas que podemos calcular o resultado previamente. O código pode ser visto no Anexo 5.

Kalman Update

Este processo representa nosso corretor e só será chamado quando uma leitura dos acelerômetros estiver disponível. A entrada do processo será o ângulo calculado pelo arco tangente das acelerações perpendiculares, como mostra a equação (5 .59).

Um dos fatores utilizado será a matriz *H*, 1*x*2 (medidas *x* estados). Ela representa a matriz Jacobiana das medidas em relação aos estados.

$$H = \begin{bmatrix} \frac{\partial\theta_m}{\partial\theta} & \frac{\partial\theta_m}{\partial e} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{5.74}$$

Para a primeira equação (5 .62), vamos separar em dois passos, o cálculo de *E* e em seguida o cálculo de *K*. *E* representa a matriz 1*x*1 de estimativa do erro, mostrada na equação a seguir:

$$E = HP_k^- H^T + R \tag{5.75}$$

Para diminuir o número de cálculos no microprocessador, simplificamos previamente os itens que sabemos que serão iguais a zero. O resultado ficará assim:

$$E = [P(0,0) + R] \tag{5.76}$$

No calculo de K precisaremos utilzar a inversa de E, porém, para nossa vantagem, essa é uma matriz 1*x*1, sendo sua inversa $E^{-1} = 1/E$.

$$\begin{aligned} K_k &= P_k^- \cdot H^T \cdot E^{-1} \\ K &= \begin{bmatrix} \dfrac{P(0,0)}{E} \\ \dfrac{P(1,0)}{E} \end{bmatrix} \end{aligned} \tag{5.77}$$

A segunda equação do processo, vista em (5 .63), atualiza nossa estimativa dos estados. Para nosso caso, temos ela reescrita do seguinte modo:

$$X_k = X_k^- + K \cdot (\theta_m - \theta_k^-) \tag{5.78}$$

$$X_k = \begin{bmatrix} \theta_k^- + K(0) \cdot (\theta_m - \theta_k^-) \\ e_k^- + K(1) \cdot (\theta_m - \theta_k^-) \end{bmatrix} \tag{5.79}$$

Assim, $\theta_m - \theta_k^-$ é o erro da estimativa, calculado pela diferença entre a medida do ângulo feita pelos acelerômetros e a estimativa do ângulo feita no processo *preditor*.

Por último, calculamos a equação (5 .64). Novamente, será necessária uma simplificação para poupar instruções em nosso microprocessador. O resultado pode ser visto na equação abaixo.

$$P_k = \begin{bmatrix} P_k^-(0,0) - K(0) \cdot P_k^-(0,0) & P_k^-(0,1) - K(0) \cdot P_k^-(0,1) \\ P_k^-(1,0) - K(1) \cdot P_k^-(0,0) & P_k^-(1,1) - K(1) \cdot P_k^-(0,1) \end{bmatrix} \tag{5.80}$$

5.1.4. Microprocessador

Para fazer leitura dos sensores e processar o Filtro de Kalman, temos alguns requisitos mínimos para o microcontrolador utilizado. São eles:

- possuir módulo de captura com pelo menos duas entradas;
- possuir módulo de conversor A/D com pelo menos uma entrada;
- possuir módulo de transmissão de dados em alta velocidade e pouco suscetível a erros;
- possuir módulo de ponto flutuante, ou aceitar um *clock* de freqüência alta o suficiente para fazer os cálculos necessários;
- ser de baixo custo.

Para atender a essas exigências mínimas, escolhemos o microprocessador da Microchip PICF876A. Este possui duas entradas para captura de sinais PWM, e cinco entradas para conversor A/D e protocolo de transmissão de dados USART. Apesar de não possuir um módulo de cálculo com ponto flutuante, o PIC pode ser utilizado com uma freqüência de operação de até 20MHz, que é rápido o suficiente para os cálculos que desejamos realizar. E a maior vantagem oferecida por esse dispositivo é sua fácil programação e custo baixo (na escala de unidades de Real).

Aquisição do sinal do Acelerômetro

Como já mencionamos antes, os sinais PWM dos acelerômetros deverão ser capturados pelos módulos de captura do PIC. O sinal recebido pelo microcontrolador pode ser visto na Figura 5 .27. A largura dos pulsos nos eixos X (T1x) e Y (T1y), e a largura do período são calculados utilizando os pontos de referência Ta, Tb, Tc, Td e Te 8. O Timer 1, um relógio de dois bytes, é utilizado como um contador, pois necessitamos dele para definir os T's mencionados. Como a ordem dos pontos de referência pode ser alterada, dependendo da inclinação do sensor, o contador deverá correr livremente, sendo somente necessário prestar atenção quando ele estoura. Veja nas figuras seguintes os três diagramas de estados para a aquisição do acelerômetro: a inicialização do módulo, a captura de T1x e T2, e a captura de T1y respectivamente.

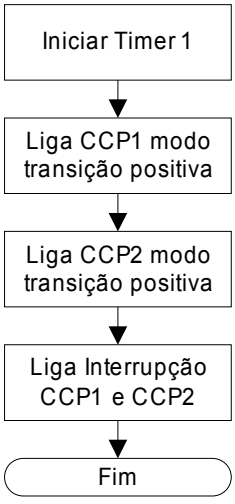


Figura 5.34 – Diagrama de estados para a inicialização da captura dos sinais do acelerômetro.

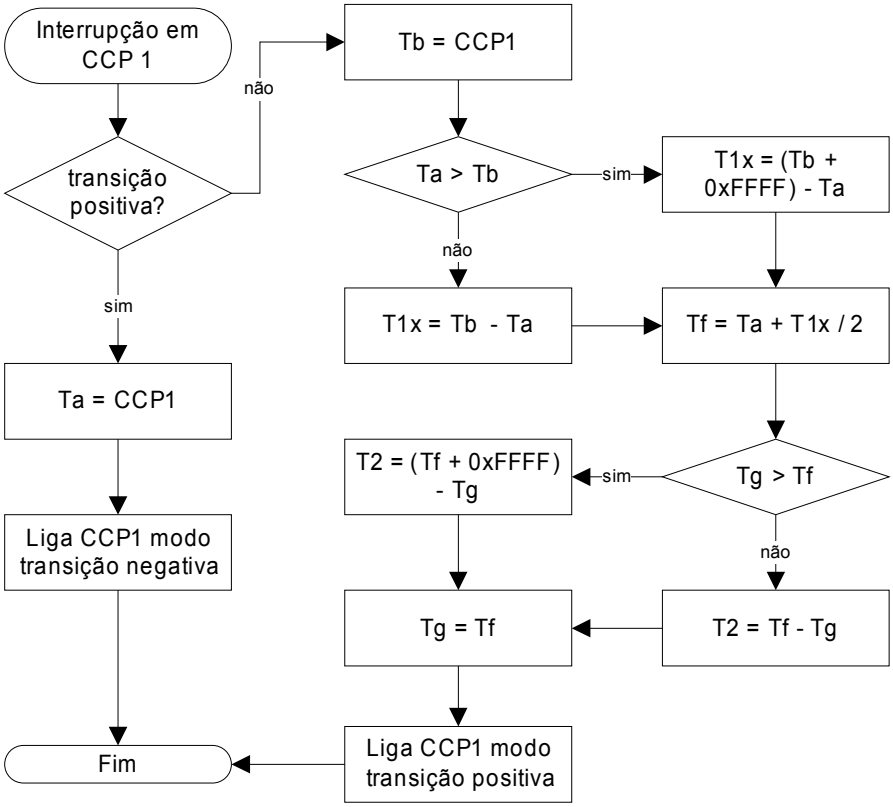


Figura 5.35 – Diagrama de fluxo para a captura de T1x e T2.

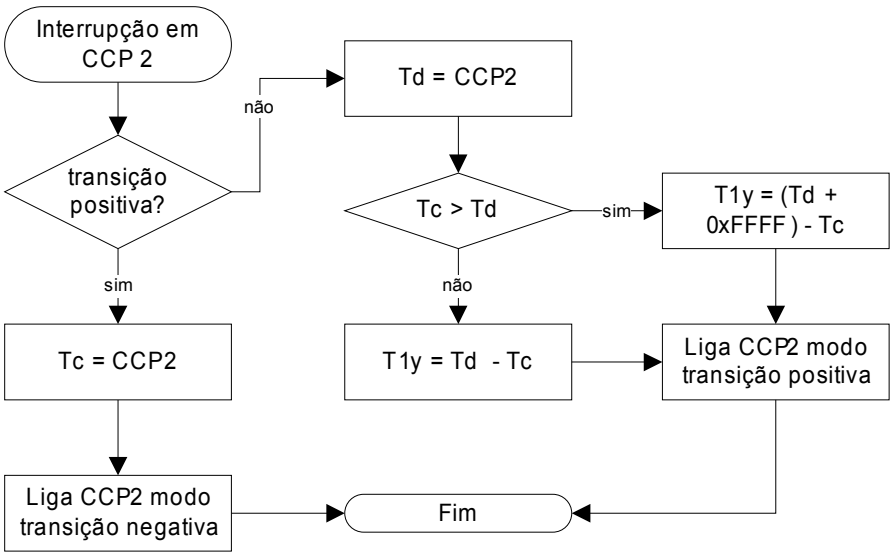


Figura 5.36 – Diagrama de fluxo para a captura de T1y.

Para programar o cálculo do ângulo θ seguindo a equação (5 .59), precisamos calcular ainda a aceleração em X e em Y como descrito em 8. Essa pode ser vista nas equações abaixo:

$$aceleração = \frac{\text{largura de pulso - largura de pulso em 0g}}{\text{largura de pulso por g}} = \frac{\left(T1/T2\right) - 50\%}{12{,}5\%} \qquad (5.81)$$

$$accelX = \frac{\left(T1x/T2\right) - \frac{1}{2}}{\frac{1}{8}} = 4\left(2 \cdot T1x/T2 - 1\right)$$
$$accelY = 4\left(2 \cdot T1y/T2 - 1\right) \tag{5.82}$$

$$\theta_m = arctg\left(\frac{accelX}{accelY}\right) = arctg\left(\frac{\left(2 \cdot T1x/T2\right) - 1}{\left(2 \cdot T1y/T2\right) - 1}\right) = arctg\left(\frac{2 \cdot T1x - T2}{2 \cdot T1y - T2}\right) \tag{5.83}$$

O código fonte para a captura do acelerômetro pode ser encontrado no Anexo 5.

Calibrando o Sinal do Acelerômetro

O processo de calibragem do acelerômetro consiste em fazer 1024 amostras do sinal dos acelerômetros por um período de 30 segundos. Durante esse tempo, o sensor deve ser girado 360º no eixo perpendicular aos sensores. O maior e menos valor de cada sensor – representando a aceleração –g e +g – é utilizado para se calcular o valor intermediário e achar a aceleração que corresponde a *0 m/s*.

Aquisição do sinal do Giroscópio

O sinal gerado pelo giroscópio da Analog Device produz uma tensão analógica de zero a cinco Volts. Utilizamos o módulo de ADC de 10 bits presente no PIC para capturar o sinal do sensor. Isso significa que teremos um sinal de 0 a 5 V para ser codificado em uma escala de 0 a 1023 pontos.

Para a aquisição do sinal pelo microcontrolador, necessitamos das seguintes funcionalidades:

- Iniciar do módulo ADC
- Calibragem
- Iniciar de captura
- Requisição de resultado

A inicialização corresponde em ativar o módulo ADC do PIC para 10 bits e colocar os dois pontos de referência sendo o terra e o V_{CC} do microcontrolador. Também recuperamos da

memória EEPROM o valor de referência para nenhum movimento adquirido na última calibragem. Assim não é necessário calibrar o sensor toda vez que o componente é ligado.

O processo de calibragem consiste em deixar o sensor em uma posição estável (isso significa, completamente parado), e retirar 64 amostras[1]. Depois é só retirar a média e temos o *gyro_zero*. Para futuras utilizações, o valor é armazenado na memória EEPROM e lido novamente sempre que o dispositivo é inicializado.

A conversão de um sinal analógico em digital demanda um determinado tempo calculado conforme a equação abaixo:

$$\begin{aligned} & T = 12 \cdot T_{AD} \\ & \begin{cases} T_{AD} = K \cdot T_{OSC} \\ F_{OSC} = 20/4 = 5MHz \\ K = 64 \end{cases} \\ & T = 12 \cdot K \cdot 1/F_{OSC} = 12 \cdot 64 \cdot 1/5.000.000 = \underline{153{,}6\mu s} \end{aligned} \tag{5.84}$$

Para termos utilizado o maior valor de K, e portanto ter uma precisão maior na medida, tivemos que sacrificar o tempo gasto para a conversão. Com o intuito de não afetar o desempenho de nosso microcontrolador e, considerando que a captura pode ser executada em paralelo com as outras instruções do PIC, decidimos separá-la em duas etapas: iniciar a captura e ler o resultado. Assim, podemos deixar o ADC calcular o valor do giroscópio enquanto fazemos outras ações, como calcular o filtro de *Kalman*. Assim que necessitamos do valor do sensor, fazemos a leitura do registrador, que já contém o valor convertido armazenado, e iniciamos novamente uma nova conversão.

A utiliza utilização do valor medido do giroscópio precisa estar na escala correta, para que os cálculos do *preditor* do filtro de *Kalman* estejam corretos. Segundo as especificações do componente utilizado, o ADXRS150 é capaz de medir uma velocidade angular de até ±150º/s, nos fornecendo uma sensibilidade de 12,5$mV/º/s$ 8. Considerando um conversor A/D de cinco Volts com precisão de 1024 pontos, podemos dizer que:

[1] O número 64 foi escolhido por dois motivos: primeiro, porque é múltiplo de 2 e a divisão pode ser feita com um *shift* para a direita; segundo, porque como o valor máximo do ADC é de 1024, após 64 amostras somadas não irá exceder o valor de 65536, equivalente a uma variável *long* – dois *bytes*.

$$\frac{5V}{0{,}0125} = \frac{1024}{x} \qquad \Rightarrow \qquad x = \frac{1024 \cdot 0{,}0125}{5} = 2{,}56 \tag{5.85}$$

Temos então que $12{,}5mV/^{o}/s$ equivalem a 2,56 pontos de captura por grau por segundo. Isso significa que:

$$1\ \text{ponto de captura} = 0{,}39°/s = \underline{0{,}006817\,rad/s} \tag{5.86}$$

Essa será a constante que, multiplicada pela medida do ADC menos o valor de calibragem, pode ser utilizada pelo filtro de Kalman. Juntando todos os itens, teremos o seguinte resultado:

$$\text{velocidade angular} = \left(gyro\_adc - gyro\_zero\right) \cdot 0{,}006817 \tag{5.87}$$

O código fonte para a captura do sinal do giroscópio pode ser encontrado no Anexo 5.

5.1.5. Montagem do Circuito

Os dois sensores utilizados para se obter a inclinação do robô necessitam de uma montagem especial, pois são sensíveis ao movimento.

Para o acelerômetro, os requisitos são dois: primeiro, que ele esteja montado com seu plano (acelerômetro X *x* acelerômetro Y) perpendicular com o eixo das rodas, coincidindo com o plano no qual o robô se movimenta. Isto significa que o acelerômetro X dentro do circuito integrado deverá ser posicionado paralelo à gravidade. O acelerômetro Y deverá ser posicionado paralelo às rodas. E o segundo requisito é que o CI seja montado no local do robô onde seja menos influenciado pelas acelerações diferentes da aceleração da gravidade.

Para o giroscópio, os requisitos são três: primeiro, que seu eixo de rotação seja posicionado paralelo ao eixo de rotação das rodas. O segundo requisito é que o CI seja montado no local do robô onde seja menos influenciado por velocidades diferentes da velocidade angular de rotação do robô em relação ao eixo de suas rodas. E o terceiro é que o circuito de alimentação seja exclusivo, já que o CI emite um sinal analógico e é, portanto, muito sensível a ruídos em sua alimentação.

Como podemos ver, os requisitos são semelhantes. Decidimos colocar os dois sensores em uma placa de circuito impresso (independente do resto do circuito) com as seguintes características:

- receber como entrada apenas uma tensão não regulada;
- possuir um regulador de tensão e um filtro para evitar ruídos (descrito na seção 5.4.2);

- possuir todos os componentes eletrônicos necessários para o funcionamento dos CI's dos sensores;
- enviar como saída todos os sinais para serem lidos externamente pelo microcontrolador em outra placa de circuito impresso;
- os sensores devem ser montados com precisão na placa, pois caso estejam tortos irão afetar a leitura do ângulo;
- ser montada o mais próximo possível do eixo das rodas;
- ser montada perpendicular ao eixo das rodas.

O esquemático do circuito pode ser encontrados no Anexo 3.

5.2. Sensores de Posição

Para obtermos a posição linear *x(t)* e a velocidade linear *v(t)* do robô, utilizamos codificadores de posição conectados diretamente aos motores DC.

Como já visto, o motor escolhido para o projeto vem acoplado a um *encoder*. Este possui dois canais de saída e através da defasagem entre estes dois canais obtemos a direção e velocidade do motor, como mostrado na Figura 5 .37. O funcionamento do circuito é muito simples, temos dois modos de funcionamento: A ou B. Como nosso motor rodará para frente e para trás, vamos utilizar um circuito com duas saídas, canais invertidos. De acordo com a direção, um gerará uma seqüência de pulsos, enquanto que o outro permanecerá em nível lógico zero. As saídas deste circuito são enviadas para o microcontrolador.

A : o encoder gera uma defasagem entre o canal A e B de acordo com a direção e a velocidade de rotação do motor. Quanto maior a velocidade, menor é a defasagem entre os canais.

B : o encoder gera uma defasagem entre o canal A e B de acordo com a direção. A velocidade de rotação do motor controla a freqüência da onda quadrada gerada. Quanto maior a velocidade, maior é a freqüência do sinal.

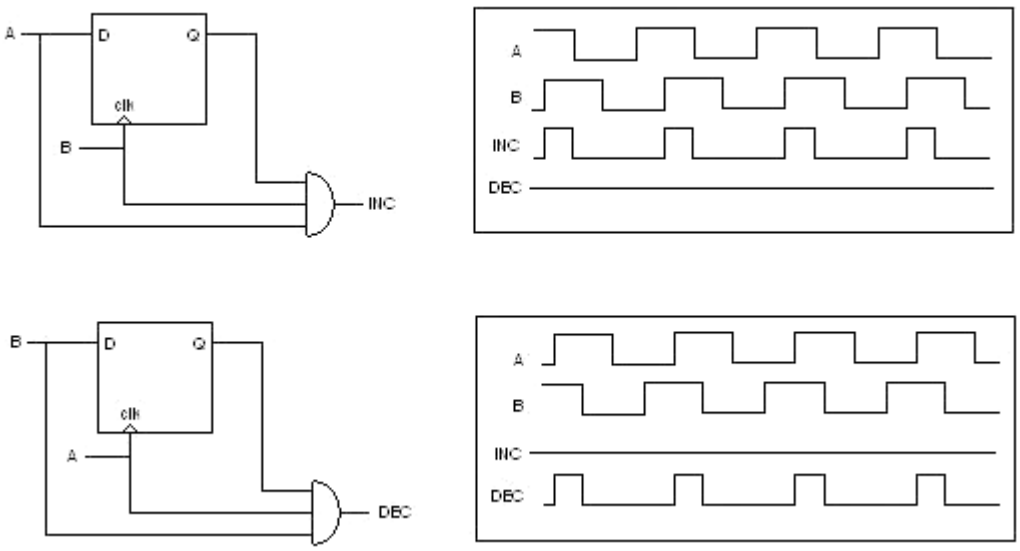


Figura 5.37 – Sinais de saída do sensor de posição.

5.2.1. Aquisição do sinal dos sensores de posição

Como visto acima, a aquisição da posição é feita pelo *encoder* acoplado ao motor. Este possui dois canais de saída. Através da defasagem entre estes dois canais obtemos a direção e através da freqüência obtemos a velocidade do motor. Dois *Flip-Flop* são utilizados para converter o sinal de "defasagem" para "rotação em uma direção" e "rotação na outra direção".

O esquemático do circuito para captura do sinal do *encoder* pode ser encontrado no Anexo 4, na segunda página do esquemático.

5.2.2. Implementação dos sensores de posição

Para manter o robô em posição de estabilidade, necessitamos somente de um sinal de retorno, o ângulo. Por isso não estaremos utilizando os sensores de posição nessa fase do projeto. Eles serão utilizados em uma futura expansão do robô.

5.3. Sistema dos Motores

5.3.1. Motores e Circuitos de Potência

O motor escolhido para o projeto é o PITTMAN GM9434H511R3 – nele vem acoplado um *encoder* que será utilizado para aquisição da posição. Este é um motor DC de 24 volts, com corrente máxima igual 8A e torque de 36,5 x 10^{-3} Nm/A. Mais informações sobre o motor podem ser encontradas em seu *datasheet* 8.

O melhor modo de alimentarmos o motor será através de um sinal PWM. A operação com PWM conserva a potência e reduz o aquecimento do circuito, resultando numa maior confiabilidade. Como não modula a amplitude, e sim a largura de pulso (*dutty-cycle*), ele permite um fino controle da potência fornecida. Além disso, o motor, como a maioria dos dispositivos eletromecânicos, funciona como um *passa-baixas*, isto é, ele filtra o sinal e somente o nível médio é observado.

Para fornecer potência ao motor, necessitamos de um CI que possa fornecer a corrente máxima do motor de 8A na tensão máxima do motor de 24V. Para isso escolhemos um CI de fácil utilização que possui quatro *drivers* que suportam picos de até 2A a uma tensão de 36V, o SN754410 da Texas Instruments 8. Utilizaremos dois desses componentes por motor para manter uma margem de segurança e não queimarmos o componente. Veja as informações do CI na tabela abaixo:

Voltagem de Alimentação	5,0	V
Voltagem de Potência	4,5 – 36,0	Hz
Corrente Nominal por Driver	1.1	A
Pico de Corrente	2	A
Número de Drivers	4	

Tabela 5.10 – Parâmetros do CI SN754410 8.

5.3.2. Sinal PWM

A freqüência de operação do PWM afeta o ruído que é gerado pelo motor. O ouvido humano é sensível a freqüências entre 20Hz e 20kHz, porém freqüências acima de 4kHz são consideradas imperceptíveis pelo o ouvido humano 8. Por isso estaremos utilizando uma freqüência de 19,5 kHz para o PWM.

A variação do *dutty-cycle* é responsável por definir a direção e a intensidade da rotação do motor. O *dutty-cycle* do sinal PWM será controlado pelo módulo ECCP do micro-controlador escolhido, como veremos mais adiante.

Vamos apresentar algumas configurações possíveis do sinal gerado para o motor. Todas foram testadas e a melhor foi escolhida para a versão final.

Primeira Configuração

A primeira opção de configuração do PWM que se poderia utilizar seria com a configuração mostrada na Figura 5 .38. Com *dutty-cycle* de 50% o motor fica parado Com um *dutty-cycle* maior que 50%, o motor roda para uma direção e com o *dutty-cycle* menor, ele roda na direção oposta.

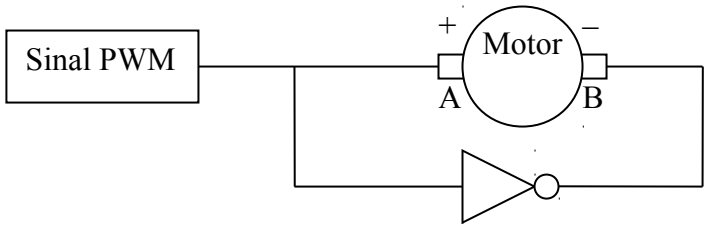


Figura 5.38 – Primeira configuração do sinal enviado ao motor.

Vejamos um exemplo do sinal enviado para o motor e como o *passa-baixas* do motor interpreta o sinal:

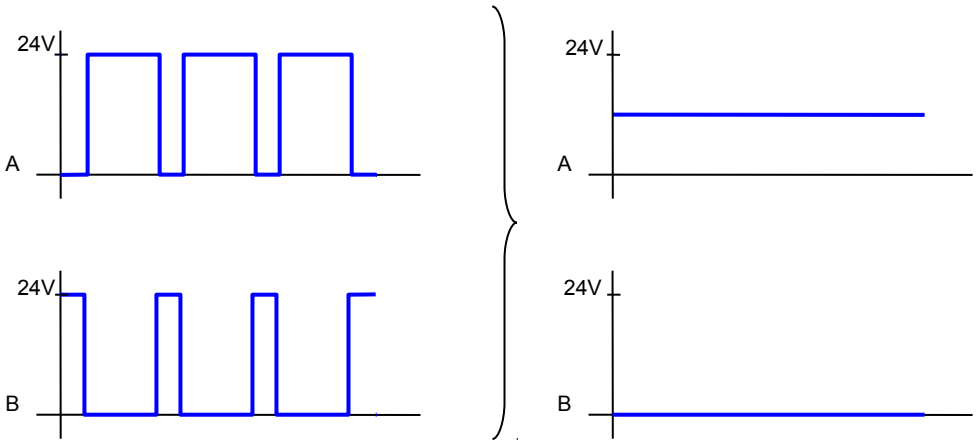


Figura 5.39 – Sinal PWM enviado para o motor e sinal interpretado pelo passa-baixas do motor.

A grande vantagem desse método é a possibilidade de apenas com o sinal PWM poder controlar também a direção em que o motor irá rodar. Porém, com isso, limitamos nossa largura de pulso pela metade. Com essa configuração, estaremos estressando mais o CI de potência, podendo levá-lo a superaquecer.

Segunda Configuração

A segunda opção de configuração do sinal enviado para o motor é mostrada na Figura 5 .40. Agora utilizamos todo o *dutty-cycle* para definir a velocidade do motor e introduzimos mais um sinal vindo do microcontrolador para informar em qual direção o motor deve girar.

Quando o sinal de direção estiver em 1, ele gira para um lado, e quando estiver em 0 gira para o outro.

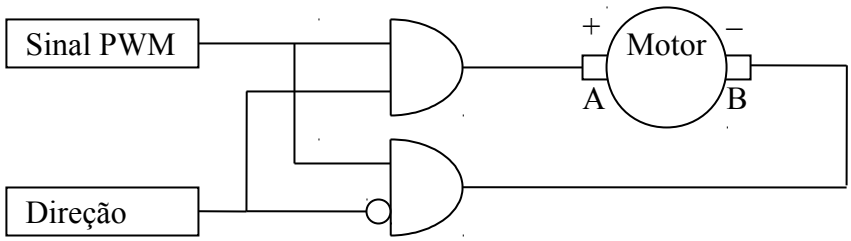


Figura 5.40 – Segunda configuração do sinal enviado ao motor.

Vejamos um exemplo do sinal PWM e o sinal de direção enviados para o motor e como o passa-baixas do motor interpreta o sinal:

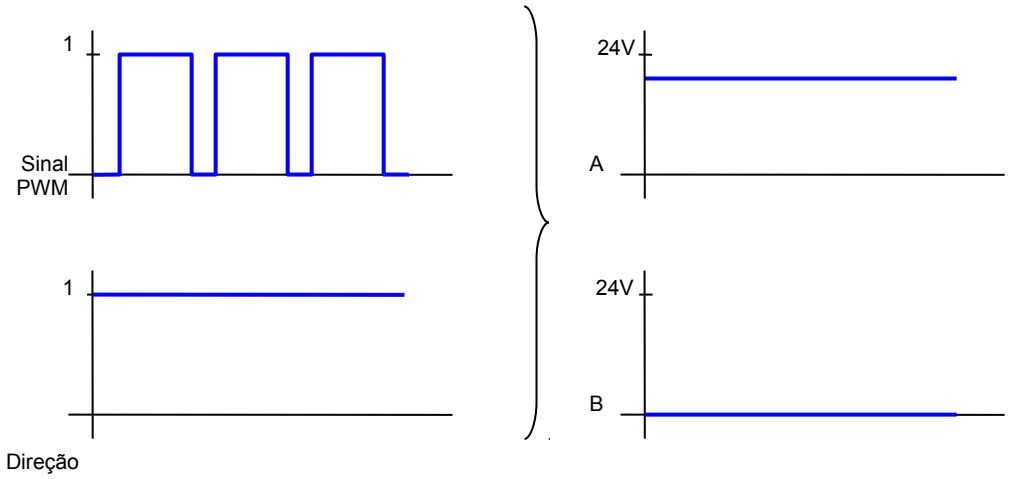


Figura 5.41 – Sinal PWM e direção enviados.

Com essa configuração podemos utilizar todo o *dutty-cycle* do sinal PWM para definir a tensão enviada ao motor. Isso significa que temos uma precisão duas vezes maior que na configuração anterior. Porém, continuamos com o problema de estressar o CI de potência, podendo levá-lo a superaquecer. Isso é resolvido na próxima configuração.

Terceira Configuração

A terceira opção de configuração do sinal enviado para o motor tem o objetivo de diminuir os picos de corrente do motor. Para isso, vamos entender um pouco mais o sinal enviado ao motor:

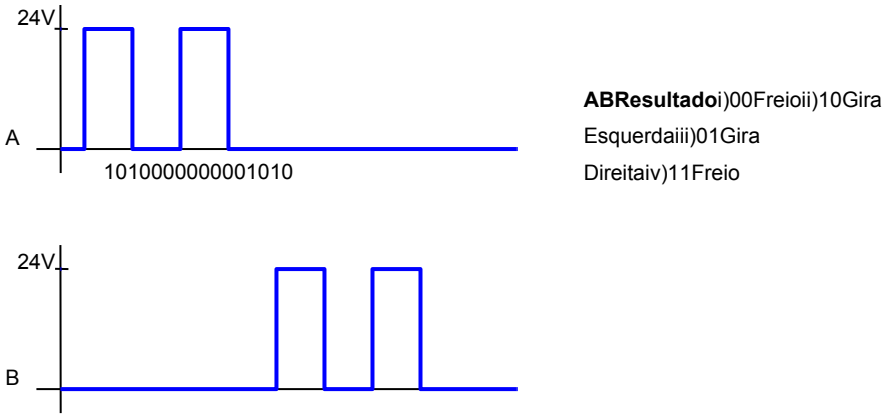


Figura 5.42 – Interpretação do sinal PWM.

Para evitarmos os momentos de frenagem induzida, o que força um gasto maior das baterias e possíveis picos de corrente, vamos fazer com que os momentos de frenagem não ocorram. Para a situação *iv)* isso é automático, pois em nosso circuito isso não pode acontecer. Porém, para a situação *i)*, isso se torna mais complicado. O que vamos fazer é colocar um dos sinais em alta impedância sempre que o sinal for para a situação *i)*. Veja na Figura 5 .43:

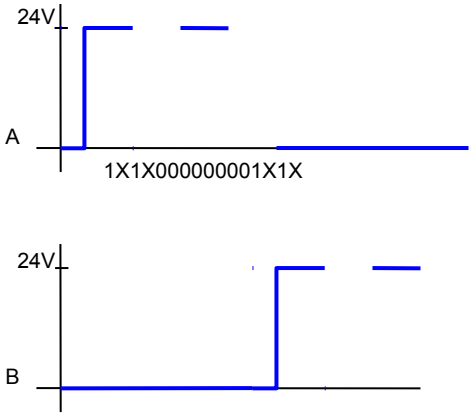


Figura 5.43 – Sinal PWM sem pontos de frenagem.

Abaixo podemos ver o circuito para tal configuração.

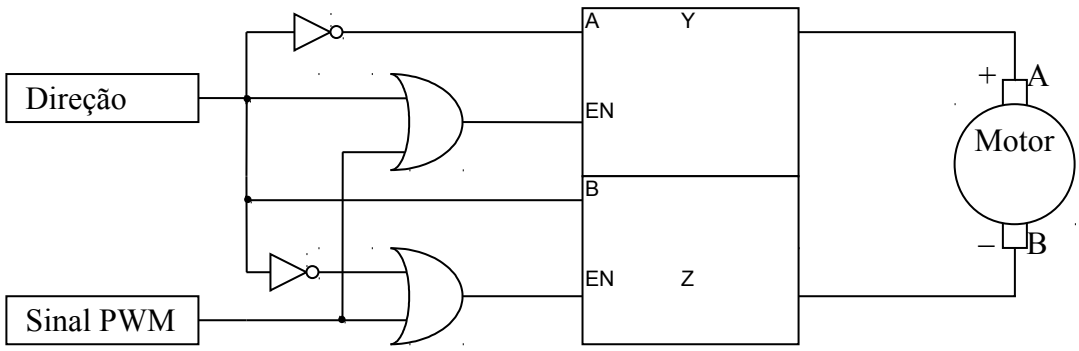


Figura 5.44 – Terceira configuração do sinal enviado ao motor.

Essa é a configuração escolhida para o projeto. O circuito pode ser encontrado no Anexo 4 na página 3 do esquemático.

5.3.3. Isolamento e Proteção do Circuito de Potência

Motores de alta potência normalmente influenciam os circuitos a eles conectados, pois fornecem uma interferência devido à alta corrente que passa por seus indutores. Para isso, estaremos tomando duas ações de precaução.

Isolamento do Circuito

Vamos isolar completamente os circuitos dos microcontroladores e dos sensores, do circuito de potência, o qual consiste do CI de potência e dos CI's de *OR* e do inversor. Para isso vamos utilizar um *Photo-acoplador*. Com ele, estaremos isolando inclusive os terras dos circuitos, já que também serão usadas duas baterias separadas. Veja o esquema na Figura 5 . 45:

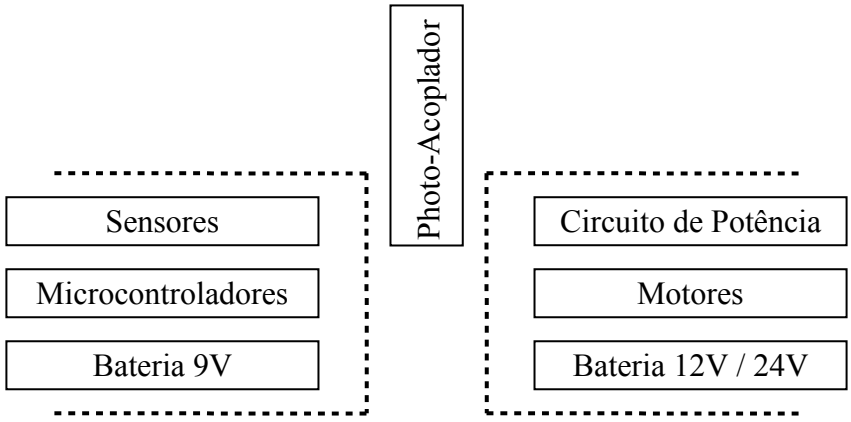


Figura 5.45 – Sistema com Photo-acoplador.

O CI utilizado no sistema foi o PS2501 da NEC 8. Devido ao sinal digital em alta freqüência enviado de um lado do acoplador, tivemos que fazer alguns ajustes no circuito, ou não teríamos uma resposta rápida o suficiente. Na entrada colocamos um transistor para poder baixar o valor do resistor em série com o led interno do acoplador. Assim não puxamos mais corrente do que o PIC suporta. Na saída utilizamos o transistor do acoplador como um Base-Coletor (e não Base-Emissor, como indicado na referência). Assim conseguimos um sinal mais limpo com um resistor mais alto, diminuindo o consumo da bateria, porém, o sinal fica na saída fica invertido, sendo facilmente resolvido inserindo um inversor. O circuito contendo os photo-acopladores pode ser encontrado no Anexo 4 na página 3 do esquemático.

Proteção do Motor

Como proteção estaremos utilizando três itens: dissipadores de calor sobre os CIs de potência para aliviar o superaquecimento, diodos de proteção nas saídas dos CIs de potência e um fusível, para que a corrente não exceda o limite permitido. O objetivo de se utilizar esses diodos é para evitar que os indutores do motor façam com que tenhamos uma tensão menor que 0V ou maior que 24V. Veja seu posicionamento na figura abaixo:

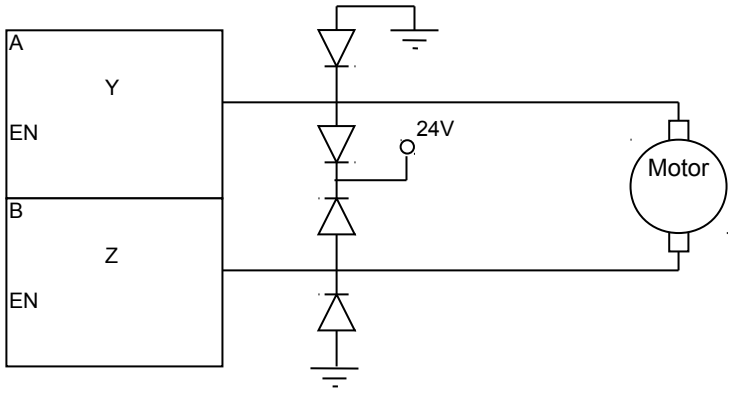


Figura 5.46 – Diodos de proteção.

5.4. Sistema de Controle

O centro de nosso projeto está no módulo de controle. Ele é responsável por juntar todas as partes e fornecer as saídas necessárias. Eis aqui os requisitos para esse sistema:

- Ser capaz de receber dados via protocolo USART (para receber o ângulo do sensor de inclinação);
- Possuir um método para entrar com as constantes de controle PID;
- Fazer ao menos 50 ciclos de controle por segundo;
- Ser capaz de gerar sinais PWM para serem enviados para o Sistema de Motores;
- Ser capaz de capturar dados vindos dos decodificadores de posição.

5.4.1. Microcontrolador

Para fazer as atividade descritas, necessitaremos de um microcontrolador que possua as seguintes características mínimas:

- possuir módulo de PWM com pelo menos dois canais;

- possuir módulo de conversor A/D com pelo menos três entradas, para as constantes Kp, Kd e Ki;
- possuir módulo de transmissão de dados em alta velocidade mas pouco suscetível a erros e compatível com o microcontrolador escolhido para o sensor de inclinação;
- possuir módulo de ponto flutuante, ou aceitar um *clock* de freqüência alta o suficiente para fazer os cálculos necessários;
- possuir algumas portas digitais de I/O para tarefas secundárias;
- ser baixo custo.

Para atender a essas exigências mínimas e para nos mantermos compatíveis com o sensor de inclinação, escolhemos o microprocessador da Microchip PICF876A. Este possui duas entradas para captura de sinais PWM, cinco entradas para conversor A/D e protocolo de transmissão de dados USART entre outros itens. Apesar de não possuir um módulo de cálculo com ponto flutuante, o PIC pode ser utilizado com uma freqüência de operação de até 20MHz, que é rápido o suficiente para os cálculos que desejamos realizar. A maior vantagem oferecida por esse dispositivo é sua fácil programação e custo bem baixo (na escala de unidades de Real).

Comunicação Serial USART

Devido à alta quantidade de dados enviados do sensor de inclinação para o módulo principal, e pensando na modularidade do sensor, decidimos enviar os dados via comunicação serial padrão USART, já que o PIC possui um módulo embutido.

Em nosso caso, o sinal só trafega em uma direção. Quando o sensor de inclinação acaba de calcular um novo ângulo, o envia via USART para o PIC principal. Isso causa uma interrupção de *firmware* no PIC, forçando-o a capturar o valor recebido e então armazenar o novo ângulo. Esse ciclo se estende durante todas as leituras de ângulo feitas pelo sensor de inclinação.

Constantes de Controle PID

Com o intuito de facilitar o ajuste do controle PID, decidimos que as constantes do controle PID podem ser alteradas manualmente. Portanto, três potenciômetros lineares foram conectados ao conversor A/D do PIC para capturar cada uma das constantes.

A leitura das constantes é feita em tempo real, ou seja, podemos alterar os valores de Kp, Kd e Ki durante o funcionamento do sistema.

5.4.2. Controle PID

Agora que detalhamos todos os sinais de entrada e saída do controlador, podemos entrar finalmente no controle PID.

Veja no diagrama da Figura 5 .47 o fluxograma do controle PID implementado.

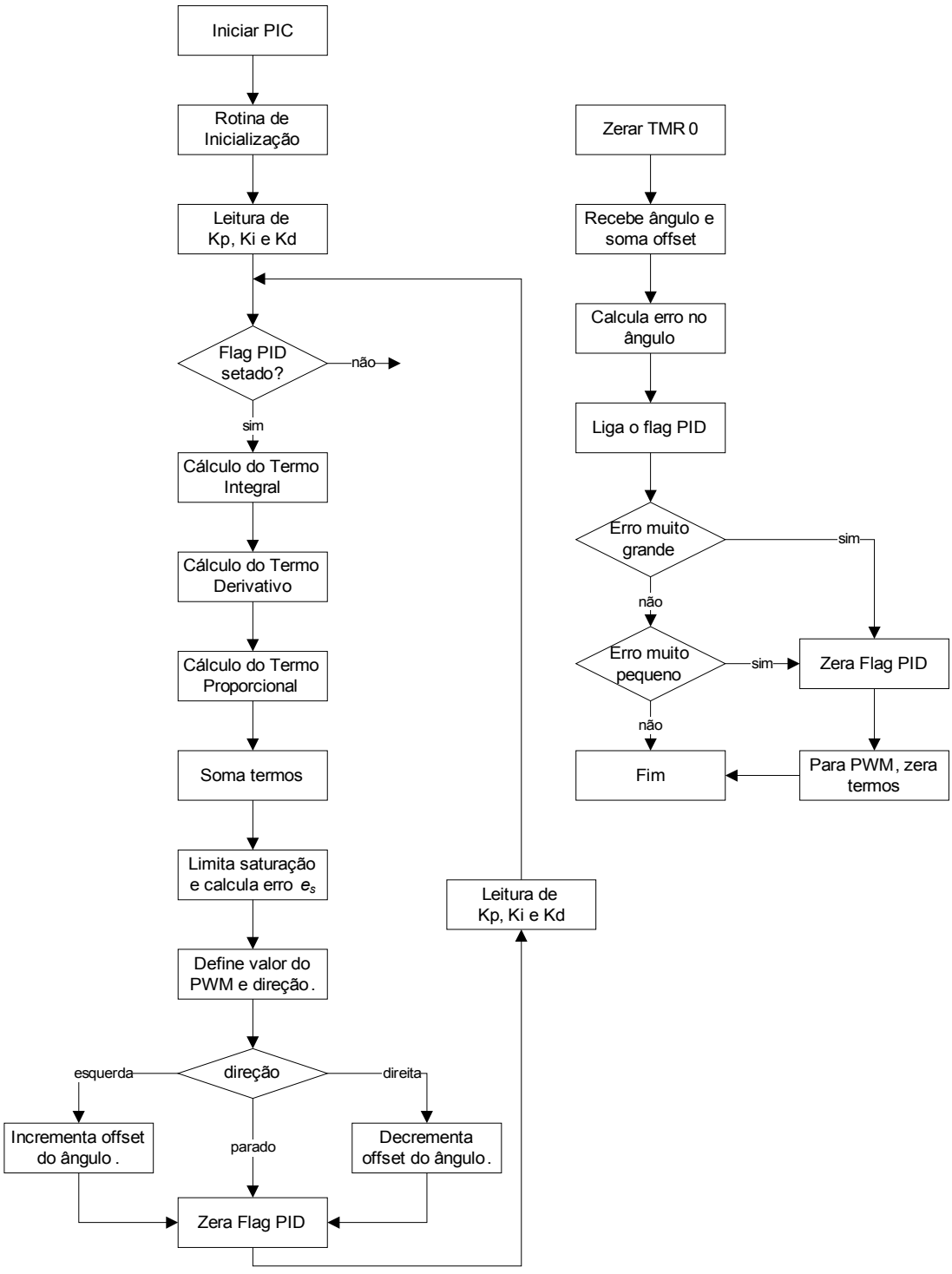


Figura 5.47 – Fluxograma do controle PID no Microcontrolador.

Ciclo de controle : Interrupção

O ciclo de interrupção é utilizado para controlar a velocidade do *loop* de controle PID. O serviço de interrupção deve estourar a interrupção Timer0. O Timer0 é um contador de 8 bits incrementado a cada pulso de instrução, que ao estourar o registrador TMR0, o *flag* de interrupção Timer0 é aceso. Como já comentamos anteriormente, a rotina de controle deve ser executada a cada 7,8 ms (1/128 Hz). Com nosso *Clock* de 20 MHz, o pulso de instrução terá uma freqüência de 5 MHz, significando uma instrução a cada 0,2us. Isso significa, a interrupção deve ocorrer a cada 39000 instruções. Colocando o prescaler em 256 e pré-carregando o TMR0 com 103, a rotina de interrupção irá ocorrer a cada 38912 instruções ((255-103)*256), o que equivale a 7,8 ms.

Sinal PWM

A saída de nosso sistema é constituída de quatro sinais: dois bits digitais e dois canais de modulação PWM, como descrito na sessão 5.3.2. Para gerar o sinal PWM no PIC, tivemos que configurar o Timer 1 para utilizar toda sua magnitude e, assim, conseguirmos uma freqüência de 19,607 kHz mantendo ainda uma resolução de 1024 pontos para cada direção.

Offset do Ângulo

Como estamos controlando um sistema que originalmente deveria ter duas variáveis de realimentação – ângulo e posição, mas implementamos com apenas uma, temos que usar alguns métodos para "enganar" o controle e, assim, manter o robô não somente equilibrado, mas também parado.

Quando o sinal de saída para o motor está com uma direção, uma variável *offset* é incrementada. Quando o sinal de saída para o motor está na direção inversa, a variável é decrementada. Esse *offset* é somado com o ângulo e utilizado na realimentação do controle PID. Com isso, podemos fazer com que a velocidade das rodas influencie no algoritmo do controle. Evitamos assim, que o robô fique equilibrado, porém permaneça se deslocando para um dos lados.

Limitamos o valor máximo que o offset pode chegar em 1/10 do ângulo máximo de 15º, ou seja, 1,5º. É um valor suficientemente grande para evitar que o robô fique andando para os lados, porém é um valor pequeno o suficiente para evitar grandes *overshoots*.

5.5. Sistema de Alimentação

Esse projeto tem a particularidade de possuir várias alimentações diferentes para variados fins. São três baterias, uma de 9 Volts e duas de 12 Volts. Também possuímos três reguladores de tensão para 5 volts.

5.5.1. Bateria de 9 Volts

A bateria de 9 Volts é utilizada por dois reguladores de tensão de 5 Volts. O primeiro está na placa principal e é responsável pela alimentação dos microcontroladores, dos sensores de posição e dos circuitos auxiliares.

O segundo regulador está localizado na placa do sensor de inclinação e é usado exclusivamente para os sensores nele embutidos. Além disso, ele está conectado a um filtro para diminuir ainda mais o ruído.

Essa separação entre os dois circuitos é necessária, pois o sensor de inclinação é extremamente suscetível a ruídos. Como ele está em uma placa separada, enviando um sinal já regulado em 5 Volts de um ponto ao outro do robô, ele adicionaria ruídos indesejados ao sinal.

5.5.2. Baterias de 12 Volts

O principal motivo de estarmos utilizando duas baterias de 12 Volts é para conseguirmos utilizar a tensão *ótima* de nossos motores de 24V. Como também desejamos isolar o circuito principal do circuito de potência, utilizamos uma das baterias de 12 Volts para regular a tensão em 5 Volts e alimentamos os CI's necessários para fornecer potência ao motor.

6. RESULTADOS

Para avaliarmos o sucesso do trabalho, devemos levar em consideração que o projeto foi feito de forma modular. Assim, podemos avaliar os módulos que têm sua importância individual e em seguida o conjunto da obra.

A estrutura física do robô foi modelada utilizando-se de uma ferramenta especializada no assunto. Como resultado, obtivemos um robô de corpo rígido, porém desmontável e até com sua estrutura de bandejas regulável.

O sensor de inclinação se mostrou eficiente e confiável, produzindo 128 leituras por segundo. Este é um item feito de forma modular, pois transmite o resultado do ângulo via comunicação serial.

O módulo dos motores é capaz de controlar um motor de até 8A com sucesso. Um sinal digital PWM pode ser enviado para ele, facilitando o trabalho de micro-controladores.

Os módulos em conjunto possibilitaram a construção de um robô de equilíbrio que permanece sem cair. Alguns testes foram feitos para avaliar a eficiência do sistema:

a) Somente a constante Kp foi utilizada. O robô não foi capaz de permanecer em equilíbrio, caindo após 4 segundos em média.

b) A constante Ki foi adicionada. O robô se equilibrou, porém todas as vezes que se desequilibra demora aproximadamente 4 segundos para voltar à estabilidade.

c) A constante Kd foi adicionada. O robô se equilibrou com menos de 1 segundo. Essa é a configuração final de seu controlador.

Veja na tabela a seguir algumas respostas do sistema a diferentes perturbações:

Perturbação	Tempo de assentamento
2 graus	300 ms
5 graus	1 s
10 graus	3 s

Tabela 6.11 – Resultados do sistema para diferentes perturbações.

7. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou a análise, modelagem e implementação de um robô de equilíbrio. Com este projeto, pudemos aplicar em nossos conhecimentos de controle, eletrônica, computação e programação. Além dos conhecimentos específicos da área de eletrônica e computação, aplicamos também conhecimentos relativos a modelagens de sistemas físicos, o que para nós foi um grande desafio.

No Sistema de Controle fomos capazes de implementar um algoritmo de controle que foi utilizado em um micro-controlador de baixo custo, sem um módulo de ponto flutuante. Isso se apresenta como uma grande vantagem, já que todas outras referências de projetos semelhantes foram utilizando micro-processadores. Ficou clara a utilidade dos micro-controladores modernos, com suas muitas funcionalidades e seu potencial para os mais diversos projetos. Com eles, boa parte da eletrônica que antes devia ser feita com CI's dedicados, pode ser integrada nos micro-controladores utilizando-se de algumas linhas de código.

Criar um sensor de inclinação resistente à trepidação e movimento, e totalmente modular foi uma das maiores conquistas desse projeto. Podemos conectá-lo a qualquer outro projeto utilizando um protocolo padrão de comunicação serial.

7.1. Trabalhos Futuros

A primeira melhoria a ser feita seria dotar o robô de movimento. Para isto, somente necessitamos colocarmos um off-set na leitura do ângulo de inclinação e implementarmos um sistema de controle remoto. Com base nesta mesma idéia, e considerando que nossos motores já são independentes, podemos permitir ao robô realizar curvas, aplicando um controle diferencial no sinal de cada motor.

Pensando mais adiante, poderíamos acoplar uma câmera e um sistema de radiofreqüência para que se possamos controlar o robô remotamente, ou ele mesmo tomar suas próprias decisões.

Sensores de obstáculos podem ser incorporados no projeto, para evitar que quando o robô caminhe, esbarre no que estiver à sua frente.

8. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Khalil Sultan, Ashab Mirza, "Inverted Pendulum – Analysis, Design and Implementation", July 25, 2003 Institute of Industrial Electronics Engineering, Karachi, Pakistan.

[2] Karl Johnn Aström and Tore Hägglung, "Automatic Tuning of PID Controllers", 1988, Instrument Society of America.

[3] http://www.seismicity.segs.uwa.edu.au/welcome/seismic_monitoring/A_history_of_seismology , visitado 15/11/05, Seismicity of Western Australia, University of Western Australia

[4] http://www.segway.com/shop/ ; visitado em 15/11/05

[5] http://www.independencenow.com/ibot/ ; visitado em 15/11/05

[6] BULLETIN LO-COG DC "Gearmotors Series GM8000, GM9000, GM14900", Pittman Company USA.

[7] Datasheet "Low-Cost _2 g Dual-Axis Accelerometer with Duty Cycle Output - ADXL202E*", 2000, Analog Device USA.

[8] Harvey Weinberg, "Using the ADXL202 Duty Cycle Output", Application Note AN604, 2002, Analog Device USA.

[9] John Geen and David Krakauer, "New iMEMS Angular-Rate-Sensing Gyroscope", 2003, Analog Dialog Micromachined Products Division.

[10] Datasheet "±150°/s Single Chip Yaw Rate Gyro with Signal Conditioning - ADXRS150", 2000, Analog Device USA.

[11] Greg Welch and Gary Bishop "An Introduction to the Kalman Filter", 5 de abril 2004, Department of Computer Science, University of North Carolina, USA.

[12] Datasheet "QUADRUPLE HALF-H DRIVER SN754410", November 1986 – revised November 1995, Texas Instruments, USA.

[13] Datasheet "HIGH ISOLATION VOLTAGE SINGLE TRANSISTOR TYPE MULTI PHOTOCOUPLER SERIES - PS2501-1,-2,-4, PS2501L-1,-2,-4", April 2005, NEC, USA.

[14] Steven W. Smith, Ph.D., "The Scientist and Engineer's Guide to Digital Signal Processing" Chapter 22 - Audio Processing / Human Hearing, 1997-2006 by California Technical Publishing.

ANEXO 1 – ARQUIVOS DE SIMULAÇÃO DO MATLAB

TransFunc_openLoop.m

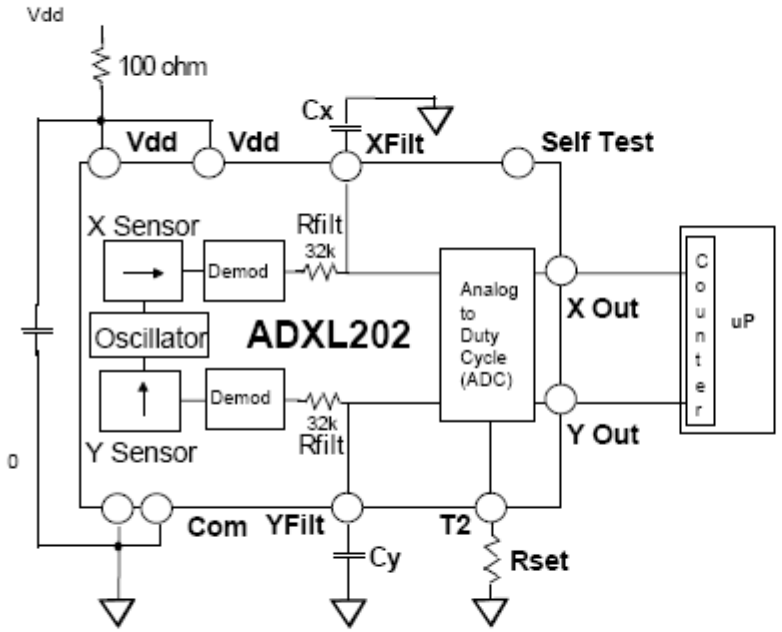
```
%Defining parameters
M = 0.3; %kg
m = 4.3; %kg
l = 0.1533; %m
g = 9.8; %N
I = 0.00047; %kgm2
q = (M+m)*(I+m*l^2)-(m*l)^2;
b = 0.1;

q = (M+m)*(I+m*l^2)-(m*l)^2

% transfer function without b coeficient
%num = [m*l/q]
%den = [1  0  -(M+m)*m*g*l/q ]

% transfer function with b coeficient
num = [m*l/q  0]
den = [1  b*(I+m*(l^2))/q  -(M+m)*m*g*l/q  -b*m*g*l/q]

% Impulse response
t = 0:0.01:5;
%impulse(num,den,t)
%axis([0 1 0 60]);

G = tf(num,den);
H = tf(5);

% Locations of Poles and Zeroes of Open-Loop Transfer Function in Complex Plane
figure
pzmap (G)
title ('Pole-Zero Map of Open-Loop Uncompensated Inverted Pendulum System')

% Impulse Response
figure
impulse (G,t)
axis([0 1 0 60]);
title ('Impulse Response of Open Loop Uncompensated Inverted Pendulum System')

% Step Response
figure
step (G,t)
axis([0 1 0 60]);
title ('Step Response of Open Loop Uncompensated Inverted Pendulum System')

% Locations of Poles and Zeroes of Closed-Loop Transfer Function in Complex Plane
figure
pzmap (feedback (G, H))
title ('Unity Feedback Closed-Loop Uncompensated Inverted Pendulum System')

% Root Locus Plot of Uncompensated System
figure
rlocus (G*H)
title ('Root Locus of Uncompensated Inverted Pendulum System')

zeta=1.3;
Wn=3.1;
sgrid(zeta, Wn)
```

TransFunc_closedLoop_PID.m

```
%Defining parameters
M = 0.3; %kg
m = 4.3; %kg
l = 0.1533; %m
g = 9.8; %N
I = 0.00047; %kgm2
q = (M+m)*(I+m*l^2)-(m*l)^2;
b = 0.1;

q = (M+m)*(I+m*l^2)-(m*l)^2

% transfer function without b coeficient
%num = [m*l/q]
%den = [1  0  -(M+m)*m*g*l/q ]

% transfer function with b coeficient
num = [m*l/q  0]
den = [1  b*(I+m*(l^2))/q  -(M+m)*m*g*l/q  -b*m*g*l/q]

% Impulse response
t = 0:0.01:5;
%impulse(num,den,t)
%axis([0 1 0 60]);

G = tf(num,den);
H = tf(5);
K = 15;
C = tf([1 60 900],[1 0]);

% Locations of Poles and Zeroes of Open-Loop Transfer Function in Complex Plane
figure
pzmap (K*C*G)
title ('Pole-Zero Map of Open-Loop Uncompensated Inverted Pendulum System')

% Impulse Response
figure
impulse (K*C*G,t)
axis([0 1 0 60]);
title ('Impulse Response of Open Loop Uncompensated Inverted Pendulum System')

% Step Response
figure
step (K*C*G,t)
axis([0 1 0 60]);
title ('Step Response of Open Loop Uncompensated Inverted Pendulum System')

% Locations of Poles and Zeroes of Closed-Loop Transfer Function in Complex Plane
figure
pzmap (feedback (K*C*G, H))
title ('Unity Feedback Closed-Loop Uncompensated Inverted Pendulum System')

% Root Locus Plot of Uncompensated System
figure
rlocus (K*C*G*H)
title ('Root Locus of Compensated Inverted Pendulum System')
```

The XL202 Interactive Designer

Enter values below. When your design is complete the values for your design will print out on this page.



Parameters		
Supply voltage	5,0	V
Analog Bandwidth	25	Hz
Acquisition Rate	54	readings per second
Resolution (g's)	0,013	g
Resolution (deg of tilt)	0,74	deg of tilt
Microcontroller counter rate	1	MHz
T2	3,12963	mS
Power cycling %	100%	on time
Tmax	40	deg C
Tmin	15	deg C
Zero g drift Tmax	0,03	g
Zero g drift Tmin	0,02	g

Component Values		
Supply Decoupling	0,1	uF
Xcap, Ycap	0,20	uF
Rset	391,2037	kohm

You will be asked to enter variety of design parameters important to your applications. This will include such issues as how fast is the signal you need to measure, what is the required update or acquisition rate, what is the counter speed on your microcontroller. After entering target values (inputs) the spreadsheet will calculate outputs such as the resolution of the accelerometer. You can then iterate the input values and trade off parameters as necessary to meet your design goals. Only enter values that are in bold.

1. Enter your nominal supply voltage

The XL202 will operate from 3.0V to 5.25V. Enter your nominal supply voltage here.

Vdd 5,0 V

2. What is the fastest signal you want to be able to observe?

In this step you will determine the bandwidth for the analog stage of the accelerometer. The bandwidth generally determines the noise floor and thus the resolution of the accelerometer. In a later section you will also calculate digital noise sources from the PWM stage; the combination of these two noise sources determines the total noise floor. You will be measuring a real world acceleration, such as human or vehicle motion. What part of the signal content is important? If the signals are transient, such as shock or impulse, you may want to set a higher bandwidth. Human motion can often be measured at 10Hz or less. Don't forget to consider filter delays that could result in a lag between a stimulus and a response by the accelerometer, (dominated by the filter). Component values for the Xfilt and Yfilt capacitor are calculated below. You will probably want to iterate to a standard capacitor value.

Enter desired Bandwidth 25 Hz Value for Cx, Cy 0,20 uF Component Value!

3. Estimate P-P noise

The peak to peak noise of the accelerometer is the best indicator of resolution of the accelerometer. Noise is a statistical process, and is best described by an RMS measurement, (available on the datasheet). P-P noise is then estimated using a statistical estimation. You need to select a RMS to P-P estimation. The table below tells you how various RMS to P-P noise multipliers, predict the amount of time the actual signal will EXCEED the estimated P-P noise. The lower the multiplier, the more likely it is that a noise event will exceed the P-P limit.

Enter RMS to P-P multiplier 4 X of RMS

RMS Multiplier	% of time a signal will exceed the P-P estimate
2X	32,00%
4X	4,60%
6X	0,27%
8X	0,01%

Calculated noise at the analog output Xfilt and Yfilt

Noise(rms) at Xfilt, Yfilt	0,003 g (max RMS)	
Noise(P-P) at Xfilt, Yfilt	0,012 g (max P-P) @4X RMS	
Noise(P-P) at Xfilt, Yfilt	0,72 Deg of tilt (max P-P)	At 17mg/deg of tilt

Note: Noise level is inversely proportional to supply voltage

Note Decrease Noise (increase resolution) by decreasing BW.

3A. Iterate

Look at the P-P noise estimate; this is the noise limited resolution, (the smallest signal you can resolve). Is this acceptable for your application? If not you should consider adjusting the bandwidth down to reduce P-P noise and improve resolution.

4. How fast would you like to acquire the signals?

In this section we will begin the design of the digital output, and the microcontroller interface. You will input an acquisition rate, i.e. how many times per second you want a new reading from the accelerometer. You are also asked how long the part should be powered each second. Note that if you only want a few samples per second, but intend to keep the part powered all of the time, then you will need to set a faster acquisition rate in order to get reasonable values for the PWM output. The program requests that you input the time required to do the multiplies and divides to calculate the acceleration. 3.0ms is the time required for a Microchip 16C83 running at 4 Mhz. This section generates a component value for the Rset resistor.

Enter desired acquistion rate 54 Each Channel per second

% of time part will be powered per second 100%

Calculate Acquisition Time

Maximum time available to acquire 2 channels	9,3	ms
Time required to calculate two channels	3,0	ms
Time left for signal acquisition	6,3	ms (two channels)

This implies a requirement for the value of the PWM period T2

Thus, T2 = 3,12962963 mS or 320 Hz This is the Sample Rate

Value for Rset 391,203704 kohm Component Value!

5. Enter the counter rate of your Microcontroller and calculate the resolution of the digital output.

In Section 2, we calculated the resolution of the analog section. In this section we will calculate the resolution of the digital output; a fuction of the PWM rate T2 (calculated in section 4), and the counting rate of your microcontroller. Please note that the counting rate is different, and usually slower than the microcontroller clock rate. The output of this calculation is a measure of the quantization error of the counter. In some cases it may limit the ultimate resolution; we will explore this in section

Counter Rate 1 **Mhz**

Note: you will need a counter of size 6259,259 counts or 13 bits
To avoid overflowing the counter

Resolution	391,2 Counts per g	
Resolution	0,003 g	Quantization bit size
Resolution	0,15 Deg of Tilt	Based on 17mg/deg of tilt

Note: Increase resolution by increasing counter rate or decreasing samples per second

6. Check for aliasing and other errors in sampling:

In all cases the sample rate (1/T2) needs to be faster than the bandwidth of the analog section by a factor of at least 2 in order to meet the requirements of Nyquist. Nyquist notwithstanding, a ratio of at least 10 is recommend to minimize dynamic errors that are endemic to PWM sampling techniques. If your ratio is low, you can improve it by either increasing the sample rate (by increasing the acquistion rate in section 4) or decreasing the analog bandwidth (in section 2).

Ratio of sample rate (1/T2) to analog BW: 12,8 Good!

7. Estimate of total resolution (iterate to meet design objective)

We are now in a position to bring together the various calculation above to determine the resolution of the complete analog and digital design. The ultimate resolution is determined by both the noise at the analog output (Xcap and Ycap) and the quantization bit size of the PWM + counter system. At this point check the total system resolution to see if it meets your requirements. If it does not, then revisit bandwidth at Xfilt and Yfilt, acquistion rate or counting rate to reduce noise. You may also want to consider digital filtering, (oversampling) to reduce noise at the expense of sampling rate as discussed in the next section.

Noise due to analog section	0,012 g (max P-P) @4X RMS	This is the noise contribution at the analog output Xcap, Ycap
Resolution of digital output- counter	0,003 g	This is the quantization noise of the digital output
Estimated Total Noise (resolution):	0,013 g P-P	This is the total P-P noise, which is the root sum square of the analog and digital noise.
Estimated Total Noise (resolution):	0,7 deg of tilt @ 17mg/deg	
Noise (Resolution) is limited by:	Bandwidth at Xfilt, Yfilt; reduce bandwidth if lower noise desired (section 2)	

8. Option: Reduce noise by oversampling (at expense of bandwidth)

Another design option is to use digital filtering (averaging) in order to reduce noise, at the expense of bandwidth. By averaging several samples you are in effect filtering the signal. Implementing averages of 2,4 8, 16 samples are simple right shifts in microcontroller code (very efficient). For oversampling to work, samples need to be taken at a rate no faster than 10 times the analog bandwidth. Note: make sure oversampling is set to 1 sample if you don't want to use oversampling!

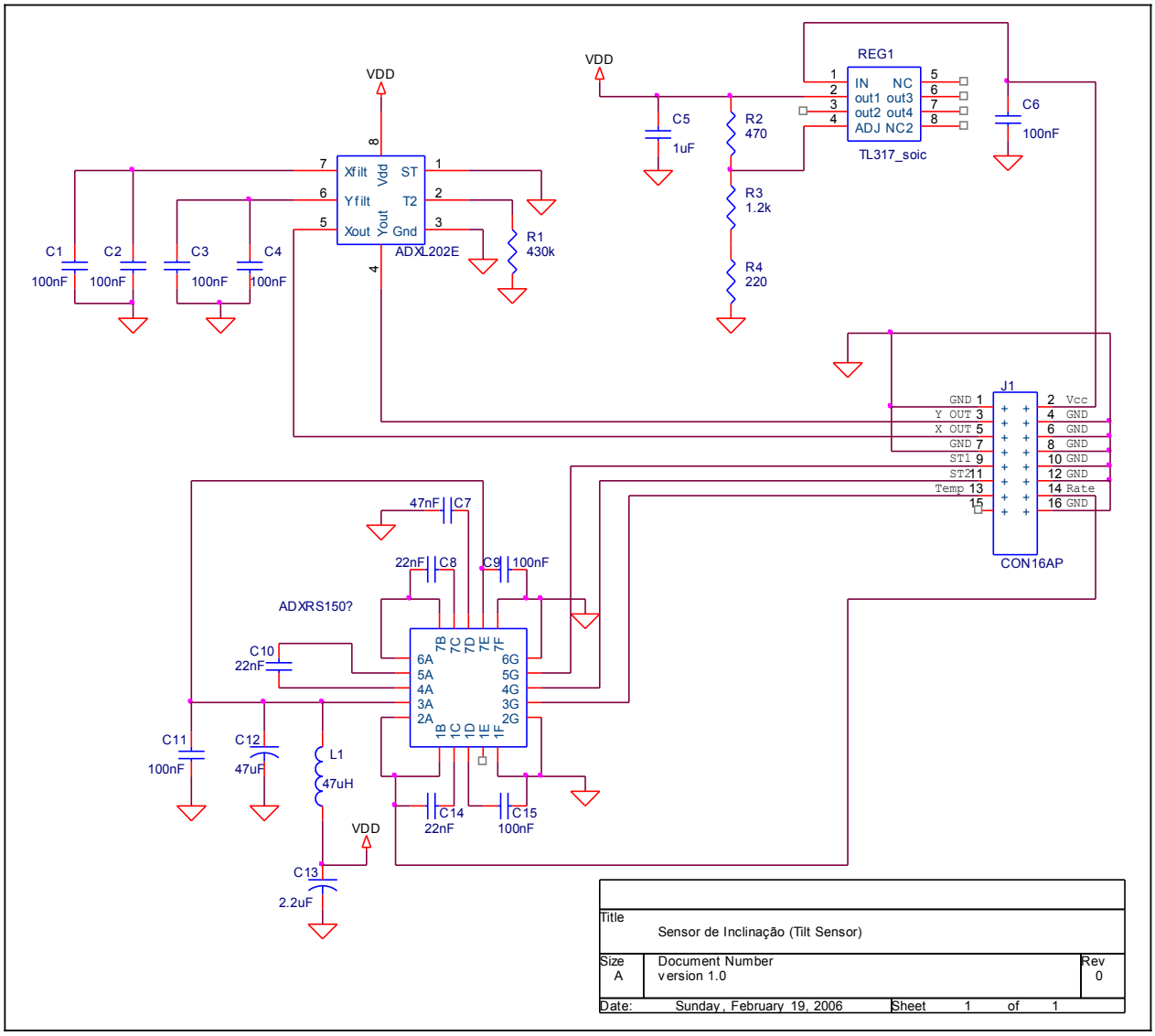
Estimated Noise (resolution) with average of:	1 Samples	Noise before oversampling	0,013 g P-P	
Note: Samples should be taken about:	40 mS apart	Noise after oversampling	0,013 g P-P	0% Reduction
		Bandwidth before oversampling	25 Hz	
		Bandwidth after oversampling	25 Hz	

9. Estimated Drift of Zero g point

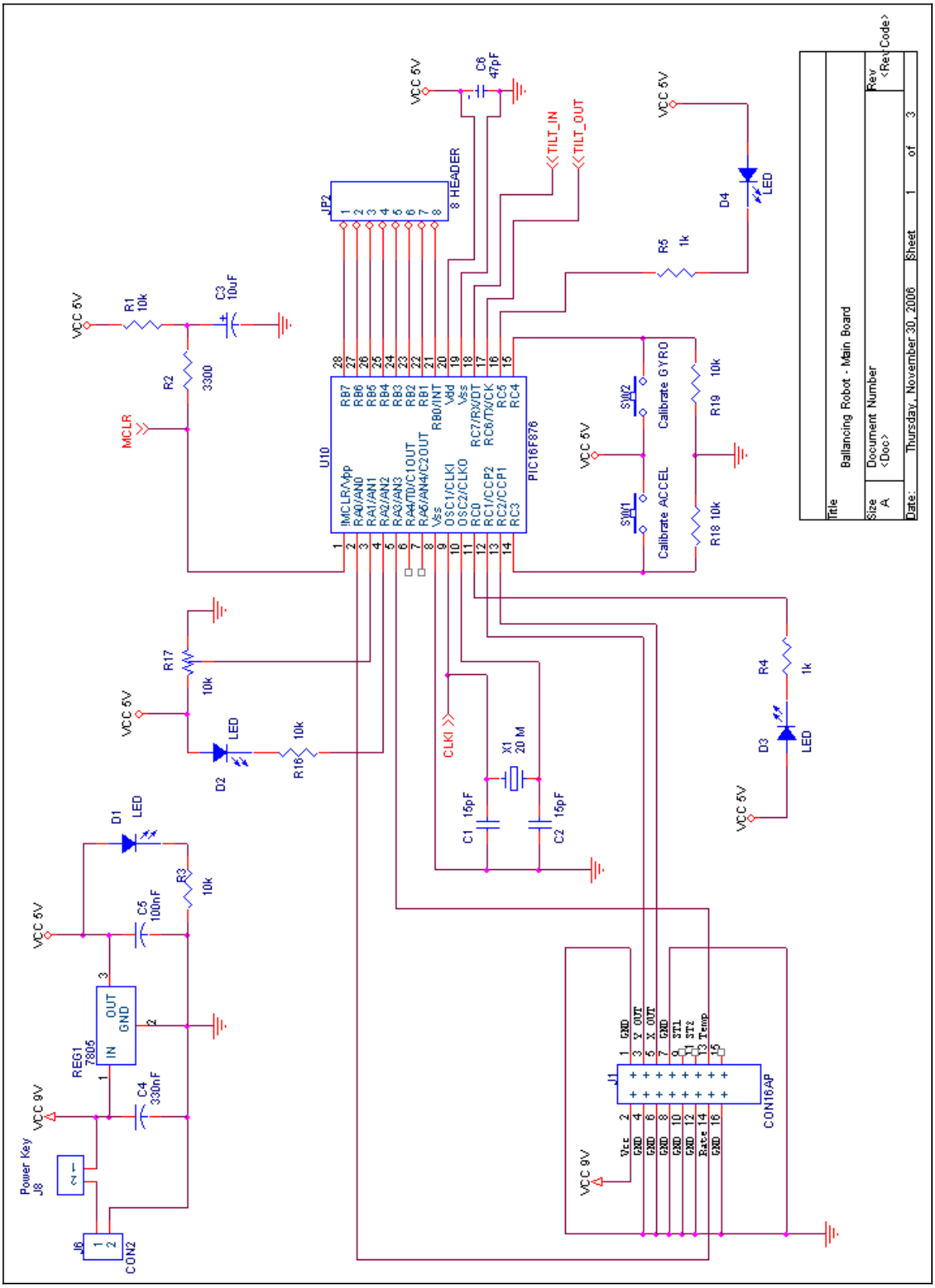
You can estimate the zero g temperature shift by entering your expected temperature range below and an estimate of the drift in mg/C (from the data sheet). Note that zero g drift can be positive or negative, but in general is very linear. X axis and Yaxis drift are uncorrelated.

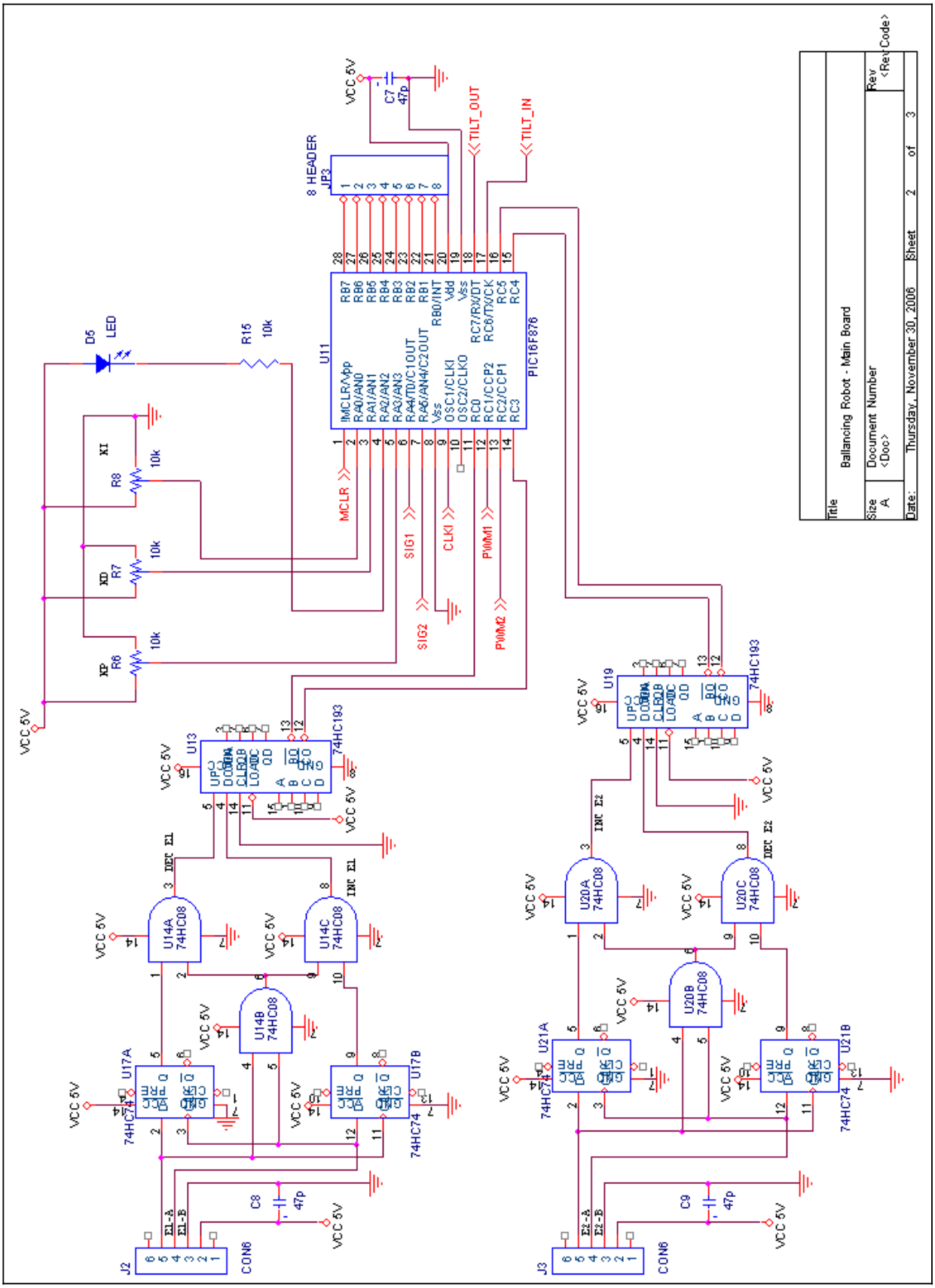
			Drift from 25C Value		
			Drift in mG	Drift in deg of tilt	
Drift in mG per degree C	0,002 g/C				
Max Temp	40 C	at Tmax	0,03 mg	1,8 deg of tilt	at 17mg/deg C
Min Temp	15 C	at Tmin	0,02 mg	1,2 deg of tilt	at 17mg/deg C

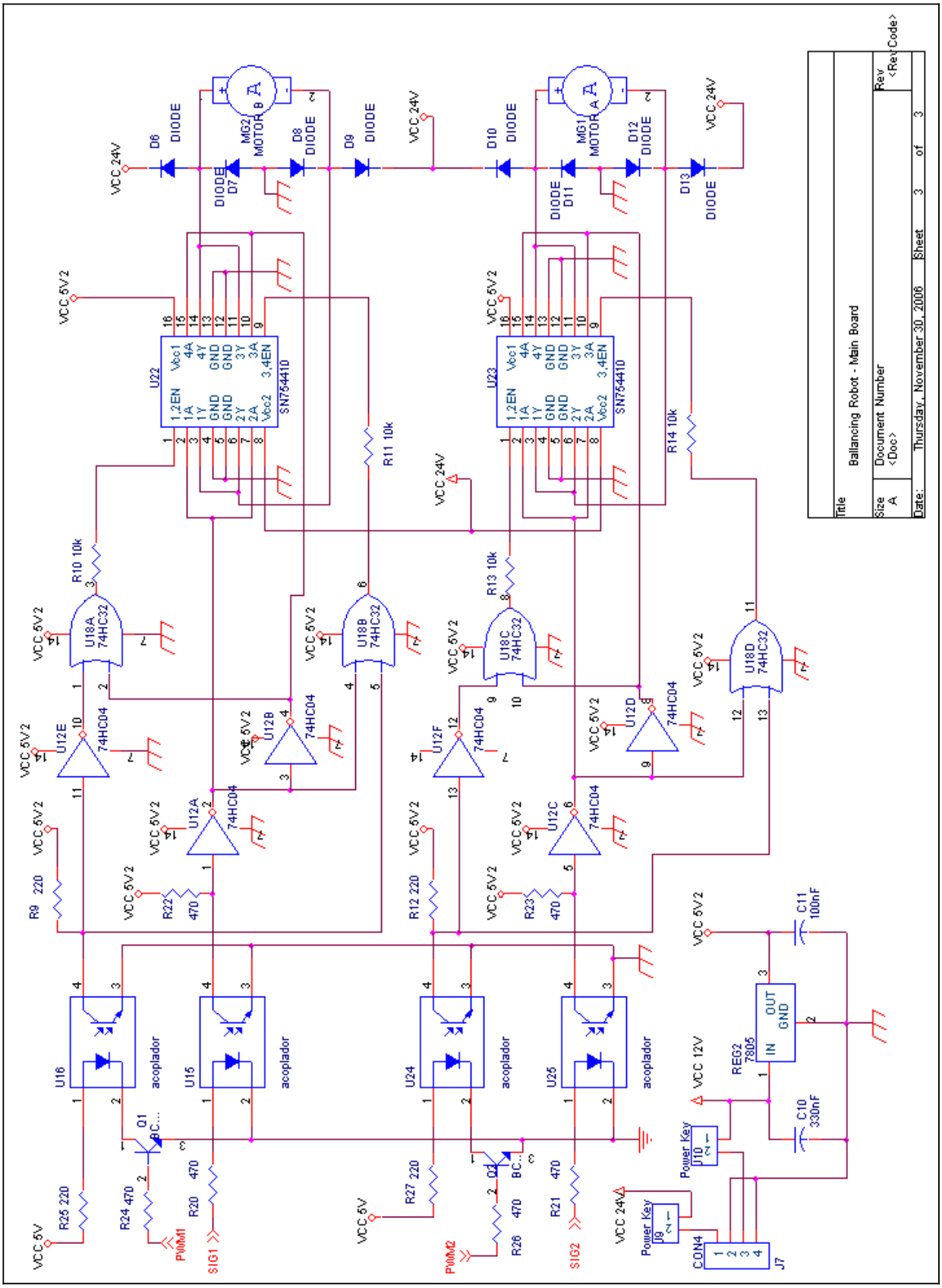
ANEXO 3 – ESQUEMÁTICO DO SENSOR DE INCLINAÇÃO



ANEXO 4 – ESQUEMÁTICO DA PLACA PRINCIPAL




Title
Balancing Robot - Main Board
Size
A
Document Number
<Doc>
Rev
<RevCode>
Date:
Thursday, November 30, 2006
Sheet
2
of
3
PIC16F876
U11
8 HEADER
JP3
TILT_OUT
TILT_IN
LED
D5
R15
10k
KI
KD
KP
R8
R7
R6
MCLR
SIG1
SIG2
CLK1
PWM1
PWM2
74HC193
74HC08
74HC74
CON6
J2
J3
C7
C8
C9
47p
VCC 5V


Title
Balancing Robot - Main Board
Size
A
Document Number
<Doc>
Rev
<RevCode>
Date:
Thursday, November 30, 2006
Sheet
3
of
3
VCC 24V
VCC 5V2
VCC 5V
VCC 12V
D6 DIODE
D7 DIODE
D8 DIODE
D9 DIODE
D10 DIODE
D11 DIODE
D12 DIODE
D13 DIODE
MG2 MOTOR B
MG1 MOTOR A
U22 SN754410
U23 SN754410
R10 10k
R11 10k
R13 10k
R14 10k
U18A 74HC32
U18B 74HC32
U18C 74HC32
U18D 74HC32
U12A 74HC04
U12B 74HC04
U12C 74HC04
U12D 74HC04
U12E 74HC04
U12F 74HC04
R9 220
R12 220
R22 470
R23 470
U16 acoplador
U15 acoplador
U24 acoplador
U25 acoplador
Q1 BC...
Q2 BC...
R24 470
R20 470
R26 470
R21 470
R25 220
R27 220
PWMM1
SIG1
PWMM2
SIG2
Power Key
REG2 7805
C11 100nF
C10 330nF
CON4
J7

ANEXO 5 – FIRMWARE DO SENSOR DE INCLINAÇÃO

Abaixo se encontra o firmware para o sensor de inclinação. O código foi feito utilizando o compilador de C para PIC PCW verão 3.43. O microcontrolador alvo é o PIC 16F876A da Microchip.

tiltsens.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: tiltSensor
 *  Students: André Futuro
 *
 *  Master: Carlos José D'Ávila (Casé)
 *
 *  File: tiltSens.c
 *  Date: 19/08/2005
 *
 *  Description:
 *
 * This function reads the accelerometer sensor, reads the gyro sensor
 * and passes the information through a kalman filter.
 * After that, the value is passed with rs232 to the other PIC.
 */

//#define DEBUG

//including the header file

#include ".\tiltsens.h"

#include "math.h"
//#include <stdlib.h>

#include ".\util.h"
#include ".\accel.h"
#include ".\gyro.h"
#include ".\tilt.h"

signed char accel_angle, angle_byte;

boolean bStateUpdate = false;

boolean a=0, c=0;

////////////////////////////////////////////////////
// main function
////////////////////////////////////////////////////
void main()
{
   // aux variables
       signed long x, y;
       float temp;
       boolean print=false;

signed long b=0;
output_low(PIN_B6);
```

```
	// initialization Routine
	Init();

	// loop
	while(1)
	{

		// checking calibration
		checkCal();


		// does a state update, only if it is time
		if (bStateUpdate)
		{
			// updating state with the angular speed
			state_update();

			// starting a new gyro capture
			readGyro();
			// disabling the bStateUpdate
			bStateUpdate = false;

			// setting print flag
			print = true;

// Test flag RA4 to measure the speed of the loop
if (a)
    output_low(PIN_B1);
else
    output_high(PIN_B1);
a = !a;

		//}


		// uses the accelerometer, only if it has finished reading
		// this reading is happening with frequency: 340,4 Hz
		if (bAccelReady)
		{
			// reading acceleration
			x = getAccelX();
			y = getAccelY();

			// calibrating the state with the acceleration
			kalman_update(x, y);

			// clearing AccelReady flag
			bAccelReady = false;

			// setting print flag
			//print = true;
		}

/*
printf("| %ld | %d | %ld | %d | %c \r\n", (signed long)(temp*180/PI), accel_angle,
getGyro(), angle_byte, angle_byte);
// calculating the angle only with the accelerometers
temp = atan2( (float) (y) , (float)(x) );

accel_angle = (signed char)(temp * 159);

//write_eeprom(0x00, accel_angle);
//writeShortEE(0x10, T1x);
//writeShortEE(0x12, T1y);
//writeShortEE(0x14, x);
//writeShortEE(0x16, y);

//printf("| %d | %ld | %ld | %ld | %ld \n\r", accel_angle, T1x, T1y, x, y);
```

```
//delay_ms(300);
//*/
            }

            // printing result
            //printf("%f  |  %f  |  %d  |  X:  %ld  |  Y:  %ld  \n\r",  ((float)-y  /
(float)x), temp, accel_angle, x, y);

//printf("%ld | %ld | %ld | %ld | %f \n\r", b, x, y, (signed long)(angle * 159),
angle);

//    printf("%ld | %f \n\r", (signed long)(angle * 159), angle);

//delay_ms(30);  //  there  is  a  problem  aquiring  the  data  too  fast.  need  to
investigate.
//printf("%ld \n\r", (signed long)(angle * 159));

            // debuging outputs
            #ifdef DEBUG
                  // printing result
                  printf("Angle: %f | Rate: %f | Accel_X: %ld | Accel_Y: %ld |
Gyro:  %ld",
                        angle, rate, x, y, getGyro());
                  printf("\n\r");
            #endif

            // checking if it is time to send data
            if (print)
            {
                  // conveting to one byte
                  // goes from -PI/4 to PI/4 => PI/2 resolution needed.
                  //    since I'm using one byte only -> max = 256.
                  //    -> angle = temp * (250 / (PI/2)) = temp * 159
                  // goes from -PI/6 to PI/6 => PI/3 resolution needed.
                  //    since I'm using one byte only -> max = 256.
                  //    -> angle = temp * (250 / (PI/3)) = temp * 239

//printf("| %d \n\r", angle_byte);
                  // sending the data to the other PIC
                  putc(angle_byte);
//write_eeprom(0x01, angle_byte);
                  angle_byte = (signed char)(angle * 239);


// Test flag RA4 to measure the speed of the loop
if (c)
    output_low(PIN_A2);
else
    output_high(PIN_A2);
c = !c;
            }

            print = false;
      }
}

//***************************************************************************
// Initialization Routine
//***************************************************************************
void Init()
{
      // setting the pin to input
      set_tris_a(0b11111011);

      // setting the pin to input
      set_tris_b(0b10111111);
```

```
	// setting the pin RC0, RC5, RC6 to output, rest input
	set_tris_c(0b10011110);

	// initializing Tilt variables
	zeroTilt();

	// initializing Gyro and Accelerometer
	initGyro();
	initAccel();

	CLEAR_CALIBRATING_ACCEL;
	CLEAR_CALIBRATING_GYRO;

    /* Setting Timer 2:
     * This timer will be used to read the gyro bias. every time the timer
	 * overflows I will read the gyro and calculate the tilt.
	 * This means the TIMER2 value is how many times I'll send my angle to be
	 * used by the control algorithm.
	 * As estimated, the minimum frequency that should be used is 32 Hz (uma
	 * potencia de 2, para facilitar os calculos). To be sure to have secure
	 * number I'll use a 4 times larger frequency:
	 * => 128 Hz
	 *
	 * The cycle time will be (1/clock)*4*t2div*(period+1)*postscaler
	 * In this program clock=20000000 and period=256 (8 bit counter)
	 * For the three possible selections the cycle time is:
	 *     (1/7072000)*4*16*256*16 = 37.067 ms or 26.98 hz
	 *     (1/7072000)*4*16*215*4  = 7.819 ms or 127.89 hz
	 *     (1/20000000)*4*16*256*16 = 13.107 ms or 76.29 hz
	 *     (1/20000000)*4*16*153*16 = 7.834 ms or 127.66 hz
	 * when changed here, please change also in "tilt.c"
	 */
	setup_timer_2(T2_DIV_BY_16, 153, 16);

	// enabling interruption on timer2
	enable_interrupts(INT_TIMER2);

	// enabling interruption on timer2
	//enable_interrupts(INT_RDA);

	// enabling global interruption, so any interruption may work
	enable_interrupts(GLOBAL);
}

////////////////////////////////////////////////////////
#int_timer2
timer2_handler()
{
	// updating state with the angular speed
	//    state_update();
	// starting a new gyro capture
	//readGyro();

	// just enable to state update
	bStateUpdate = true;
}


////////////////////////////////////////////////////////
// Check Calibration Routine
// this function verifies if has to do a calibration routine.
// If yes, stores the calibration in E2PROM
////////////////////////////////////////////////////////
void checkCal()
{
	char i;
	// checks if needs acceleration calibration bit is on.
	if (NEED_CALIBRATION_ACCEL)
```

```
{
      // disabling interruptions
      disable_interrupts(INT_TIMER2);
      //disable_interrupts(INT_RDA);

      #ifdef DEBUG_CAL
            printf("Calibrating\n\r");
      #endif

      // blinking with total delay of 4000
      for (i=0; i<5; i++)
      {
            SET_CALIBRATING_ACCEL;           // on LED
            delay_ms(400);                         // introducing a delay.
            CLEAR_CALIBRATING_ACCEL;  // off LED
            delay_ms(400);                         // introducing a delay.
      }

 // initializing Accelerometer calibration
 SET_CALIBRATING_ACCEL;

      #ifdef DEBUG_CAL
            printf("Accelerometer Calibration\n\r");
      #endif

      // Calibrating accelerometer
      calibrateAccel();

      #ifdef DEBUG_CAL
            printf("End Accelerometer Calibration\n\r");
      #endif

      // restart tilt
      zeroTilt();

      // blinking with total delay of 4000
      for (i=0; i<5; i++)
      {
            CLEAR_CALIBRATING_ACCEL;  // off LED
            delay_ms(400);                         // introducing a delay.
            SET_CALIBRATING_ACCEL;           // on LED
            delay_ms(400);                         // introducing a delay.
      }

      // end Accelerometer calibration
      CLEAR_CALIBRATING_ACCEL;

      // re-enabling interruptions
   enable_interrupts(INT_TIMER2);
      //enable_interrupts(INT_RDA);
}

// checks if needs gyro calibration bit is on.
if (NEED_CALIBRATION_GYRO)
{
      // disabling interruptions
      disable_interrupts(INT_TIMER2);
      //disable_interrupts(INT_RDA);

      #ifdef DEBUG_CAL
            printf("Calibrating\n\r");
      #endif

      // blinking with total delay of 4000
      for (i=0; i<5; i++)
      {
            SET_CALIBRATING_GYRO;            // on LED
            delay_ms(400);                         // introducing a delay.
```

```
                CLEAR_CALIBRATING_GYRO;          // off LED
                delay_ms(400);                         // introducing a delay.
            }

            // initializing Gyroscope calibration
            SET_CALIBRATING_GYRO;

            #ifdef DEBUG_CAL
                printf("Gyro Calibration\n\r");
            #endif

            // Calibrating Gyroscope
            calibrateGyro();

            #ifdef DEBUG_CAL
                printf("End Gyro Calibration\n\r");
            #endif

            // restart tilt
            zeroTilt();

            // blinking with total delay of 4000
            for (i=0; i<5; i++)
            {
                CLEAR_CALIBRATING_GYRO;          // off LED
                delay_ms(400);                         // introducing a delay.
                SET_CALIBRATING_GYRO;            // on LED
                delay_ms(400);                         // introducing a delay.
            }

            // end Gyroscope calibration
            CLEAR_CALIBRATING_GYRO;

            // re-enabling interruptions
          enable_interrupts(INT_TIMER2);
            //enable_interrupts(INT_RDA);
      }
}
```

tiltsens.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Tilt Sensor
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: tiltSensor.h
 *  Date: 22/03/2005
 */

#ifndef _tiltSensor_h_
#define _tiltSensor_h_

//#include <.\16F628.h>
#include <16F876A.h>
#device ADC=10

//#use delay(clock=7072000)
#use delay(clock=20000000)
#fuses HS, NOWDT, NOPROTECT, NOLVP, PUT

//#use rs232(baud=57600,parity=N,xmit=PIN_C6,rcv=PIN_C7)
```

```
//#use rs232(baud=38400,parity=N,xmit=PIN_C6,rcv=PIN_C7)
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)
//#use rs232(baud=9600,parity=N,xmit=PIN_C6,rcv=PIN_C7)

// ACCEL Calibration Defines
#define NEED_CALIBRATION_ACCEL   input(PIN_C3)
#define SET_CALIBRATING_ACCEL    output_low(PIN_C0)
#define CLEAR_CALIBRATING_ACCEL output_high(PIN_C0)

// GYRO Calibration Defines
#define NEED_CALIBRATION_GYRO    input(PIN_C4)
#define SET_CALIBRATING_GYRO     output_low(PIN_C5)
#define CLEAR_CALIBRATING_GYRO   output_high(PIN_C5)

#bit RCIF = 0x0C.5
#bit TXIF = 0x0C.4

#bit CREN = 0x18.4

void checkCal();
void Init();

#endif
```

util.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Util
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: util.c
 *  Date: 19/08/2005
 *
 *  Description:
 *
 */

//including the header file
#include ".\util.h"


////////////////////////////////////////////////////////
// Function to write a Short in the EEPROM
// this funtion will write a short content into the EEPROM
// sequentially.
////////////////////////////////////////////////////////
void writeFloatEE( int address, float data)
{
 write_eeprom(address, (long)*(&data + 3));
 write_eeprom(++address, (long)*(&data + 2));
 write_eeprom(++address, (long)*(&data + 1));
 write_eeprom(++address, (long)*(&data + 0));
}


////////////////////////////////////////////////////////
// Function to write a Short in the EEPROM
// this funtion will write a short content into the EEPROM
// sequentially.
////////////////////////////////////////////////////////
void writeShortEE(int address, long data)
{
```

```
       write_eeprom (address,data>>8);
       write_eeprom (++address,data);
}


////////////////////////////////////////////////////////
// Function to read a Short from the EEPROM
// this funtion will read a short content from the EEPROM
// sequentially.
////////////////////////////////////////////////////////
long readShortEE(int address)
{
       long ret;
       ret = read_eeprom(address) << 8;
       ret += read_eeprom(++address);
       return ret;
}
```

util.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Util
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: util.h
 *  Date: 22/08/2005
 */

#ifndef _util_h_
#define _util_h_

//#define PI 3.1416

// functions declaration.
extern void writeFloatEE( int address, float data);
extern void writeShortEE(int address, long data);

extern long readShortEE(int address);

#include ".\util.c"

#endif
```

accel.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: tiltSensor - accelerometer
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: accel.c
 *  Date: 19/08/2005
 *
 *  Description:
 *
 */
```

```
//including the header file
#include ".\accel.h"

#include ".\tiltsens.h"
#include ".\util.h"

////////////////////////////////////////////////////////
// This initializes the capture module.
// We will be using a Free running timer. So the states have to be
// checked for an overflow of the clock
////////////////////////////////////////////////////////
void initAccel()
{
    // setting timer.
    // We will be using a Free running timer. So the states have to be
    // checked for an overflow of the clock
    setup_timer_1(T1_INTERNAL|T1_DIV_BY_2);

       // no values ready for reading
       bAccelReady = false;

       // calibration off
       bAccelCalibrate = false;

       // enabling the accelerometer
       enableAccel();

       //////////////
       // reading the T2 value from EEPROM
       T2x = readShortEE(EE_T2x);
       T2y = readShortEE(EE_T2y);

       // testing if T2 value from EEPROM is not valid.
       // in that case, use default value
       if (T2x == 0xFFFF)
       {
              #ifdef DEBUG
                     printf("T2x: %LX | T2y: %LX\n\r", T2x, T2y);
              #endif
              // default values already tested
              T2x = 0x1E78; //15BD | 1D91 | 1EBC | 3D90
              T2y = 0x1F19; //160D | 1F13 | 1F2E | 3E37
           // writing into the EEPROM T2Cal
       writeShortEE( EE_T2x, T2x);
       writeShortEE( EE_T2y, T2y);
       }

       #ifdef DEBUG
              printf("T2x: %LX | T2y: %LX\n\r", T2x, T2y);
       #endif

}

////////////////////////////////////////////////////////
// This treats the CCP1 state change event
////////////////////////////////////////////////////////
#int_ccp1
ccp1_handler()
{
       // aux variable to calculate T2
       long T1x_new;

       // disabling ccp1 interruptions
       disable_interrupts(INT_CCP1);

       // checking if the event was a rising edge or a falling edge
       if (T1x_rise) // rising edge
```

```
	{
		// clearing Timer1
		//set_timer1(0);

		// getting the CCP1 value.
		T1x_new = CCP_1;

		if (bAccelCalibrate)
		{
			// checking if new mesure is smaller then the first one
			if (T1x_new < T1x_init)
				T2x = (long)(T1x_new + 0xFFFF - T1x_init);
			else
				T2x = (long)(T1x_new - T1x_init);
		}

		// getting the CCP1 value.
		T1x_init = T1x_new;

		// setting the next event to be a falling edge.
		setup_ccp1(CCP_CAPTURE_FE);
	}
	else // falling edge
	{
		// getting the CCP1 value.
		T1x_end = CCP_1;

		// checking if new mesure is smaller then the first one
		if (T1x_end < T1x_init)
			T1x = (long)(T1x_end + 0xFFFF - T1x_init);
		else
			T1x = (long)(T1x_end - T1x_init);

		// setting the next event to be a rising edge.
		setup_ccp1(CCP_CAPTURE_RE);

		// verifying if whole capture is complete:
		// X has completed the capture.
		// Checking if Y has completed the capture. If so, whole capture is
complete.
		if (T1y_rise)
			bAccelReady = true;
	}

	// changing which will be the next event type (rising or falling edge)
	T1x_rise= !T1x_rise;


	// enabling interruptions again.
	enable_interrupts(INT_CCP1);
}


////////////////////////////////////////////////////////
// This treats the CCP2 state change event
////////////////////////////////////////////////////////
#int_ccp2
ccp2_handler()
{
	// aux variable to calculate T2
	long T1y_new;

	// disabling ccp2 interruptions
	disable_interrupts(INT_CCP2);

	// checking if the event was a rising edge or a falling edge
	if (T1y_rise) // rising edge
	{
```

```
		// clearing Timer1
		//set_timer1(0);

		// getting the CCP2 value.
		T1y_new = CCP_2;

		if (bAccelCalibrate)
		{
			// checking if new mesure is smaller then the first one
			if (T1y_new < T1y_init)
				T2y = (long)(T1y_new + 0xFFFF - T1y_init);
			else
				T2y = (long)(T1y_new - T1y_init);
		}

		// getting the CCP2 value.
		T1y_init = T1y_new;

		// setting the next event to be a falling edge.
		setup_ccp2(CCP_CAPTURE_FE);
	}
	else // falling edge
	{
		// getting the CCP2 value.
		T1y_end = CCP_2;

		// checking if new mesure is smaller then the first one
		if (T1y_end < T1y_init)
			T1y = (long)(T1y_end + 0xFFFF - T1y_init);
		else
			T1y = (long)(T1y_end - T1y_init);

		// setting the next event to be a rising edge.
		setup_ccp2(CCP_CAPTURE_RE);

		// verifying if whole capture is complete:
		// Y has completed the capture.
		// Checking if X has completed the capture. If so, whole capture is
complete.
		if (T1x_rise)
			bAccelReady = true;
	}

	// changing which will be the next event type (rising or falling edge)
	T1y_rise= !T1y_rise;


	// enabling interruptions again.
	enable_interrupts(INT_CCP2);
}


//////////////////////////////////////////////////////
long getAccelX()
{
	// return (((float)T1x / (float)T2 - 0.5) / 12.5);
	return (2*T1x - T2x);
}

//////////////////////////////////////////////////////
long getAccelY()
{
	// return (((float)T1y / (float)T2 - 0.5) / 12.5);
	return (2*T1y - T2y);
}

//////////////////////////////////////////////////////
void enableAccel()
```

```
{
    // Configure CCP1 and CCP2 as a Capture RisigEdge
    // this command will put 0100 into register CCP1CON,
       // meaning that Capture mode is on.
    setup_ccp1(CCP_CAPTURE_RE);
    setup_ccp2(CCP_CAPTURE_RE);

    // waiting for a rising edge
    T1x_rise = true;
    T1y_rise = true;

    // clearing variables
    T1x_init = 0;
    T1x_end = 0;
    T1y_init = 0;
    T1y_end = 0;

    // enabling interuption on capture CCP1
    enable_interrupts(INT_CCP1);
    enable_interrupts(INT_CCP2);
}

////////////////////////////////////////////////////////
void disableAccel()
{
      // disabling ccp1 and ccp2 interruptions
      disable_interrupts(INT_CCP1);
      disable_interrupts(INT_CCP2);
}

////////////////////////////////////////////////////////
// Accelerometer Calibration Routine
// this function calibrates the acceleration module.
// It calculates the T2 value and writes it into EEPROM
////////////////////////////////////////////////////////
void calibrateAccel()
{
      long i;

      // local variables
      long T1x_max = 0, T1x_min = 0x7000;
      long T1y_max = 0, T1y_min = 0x7000;

      // iterations to find the highest and the lowest value.
      for (i = 0; i<1000; i++)
      {
            // reading the accelerometer values
            // readAccel();

            // getting T1x maximum value
            if (T1x > T1x_max)
                  T1x_max = T1x;

            // getting T1x maximum value
            if (T1x < T1x_min)
                  T1x_min = T1x;

            // getting T1x maximum value
            if (T1y > T1y_max)
                  T1y_max = T1y;

            // getting T1y minimum value
            if (T1y < T1y_min)
                  T1y_min = T1y;

            // introducing a delay
            delay_ms(25);
```

```
	}

//writeShortEE( 0x50, T1x_max);
//writeShortEE( 0x52, T1x_min);
//writeShortEE( 0x60, T1y_max);
//writeShortEE( 0x62, T1y_min);

	// getting T2x
	T2x = T1x_max + T1x_min;
	//T2 = T2x;
	// getting T2y
	T2y = T1y_max + T1y_min;

	#ifdef DEBUG
		printf("T1x_max: %LX | T1x_min: %LX | T1y_max: %LX | T1y_min: %LX | ",
				T1x_max,T1x_min,T1y_max,T1y_min);
		printf("T2x: %LX | T2y: %LX\n\r",T2x, T2y);
	#endif

    // writing into the EEPROM T2Cal
    writeShortEE( EE_T2x, T2x);
    writeShortEE( EE_T2y, T2y);
}

//////////////////////////////////////////////////////////
// Accelerometer Calibration Routine
// this function calibrates the acceleration module.
// It calculates the T2 value and writes it into EEPROM
//////////////////////////////////////////////////////////
void calibrateAccel2()
{
	long i;

	// local variables
	int32 T2x_SUM = 0;
	int32 T2y_SUM = 0;

	  bAccelCalibrate = true;
    // introducing a delay
    delay_ms(10);

	// iterations to find the highest and the lowest value.
	for (i = 0; i<0b00111111; i++)
	{
		// reading the T2x value
		T2x_SUM += T2x;

		// reading the T2y value
		T2y_SUM += T2y;

		// introducing a delay
		delay_ms(25);

	}

	  bAccelCalibrate = false;
    delay_ms(10);

	// getting T2x
	T2x = (long) (T2x_SUM >> 6);
	// getting T2y
	T2y = (long) (T2y_SUM >> 6);

	#ifdef DEBUG
		printf("T2x: %LX | T2y: %LX\n\r",T2x, T2y);
	#endif

    // writing into the EEPROM T2Cal
```

```
    writeShortEE( EE_T2x, T2x);
    writeShortEE( EE_T2y, T2y);
}
```

accel.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Tilt Sensor - accelerometer
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: accel.h
 *  Date: 22/03/2005
 */

#ifndef _accel_h_
#define _accel_h_

// Definitions
#define XOUT input(PIN_C2)
#define YOUT input(PIN_C1)

// Defining variables
long T1x; // contains the time spent for Xout
long T1y; // contains the time spent for Yout
//long T2; // contains the period
long T2x; // contains the period for Xout
long T2y; // contains the period for Yout

long T1x_init, T1x_end;
long T1y_init, T1y_end;

boolean T1x_rise; // this variable is true when is waiting for a rising edge
boolean T1y_rise; // this variable is true when is waiting for a rising edge

const byte EE_T2x = 0x30;
const byte EE_T2y = 0x32;

boolean bAccelReady;

boolean bAccelCalibrate;

// functions declaration.

void initAccel(void);
void readAccel(void);

void disableAccel(void);
void enableAccel(void);

long getAccelX(void);
long getAccelY(void);

#include ".\accel.c"

#endif
```

gyro.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: tiltSensor - gyro
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: accel.c
 *  Date: 19/08/2005
 *
 *  Description:
 *
 */


//including the header file
#include ".\gyro.h"

////////////////////////////////////////////////////////
// This Function initializes the Gyro Capture Module
////////////////////////////////////////////////////////
void initGyro()
{
   // enabling ADC
   setup_adc_ports(AN0_AN1_AN3);
   setup_adc(ADC_CLOCK_DIV_64);

   //setup_adc( ADC_CLOCK_INTERNAL );
   set_adc_channel( 0 );

       //////////////
       // reading the gyro_zero value from EEPROM
       gyro_zero = readShortEE(EE_GYRO);

       // testing if T2 value from EEPROM is not valid.
       // in that case, use default value
       if (gyro_zero == 0xFFFF)
       {
              #ifdef DEBUG
                     printf("gyro_zero: %LX\n\r",gyro_zero);
              #endif
              // default value already tested
              gyro_zero = 0x021F; // 0223 | 021F
           // writing into the EEPROM T2Cal
       writeShortEE(EE_GYRO, gyro_zero);
       }


       #ifdef DEBUG
              printf("gyro_zero: %LX\n\r",gyro_zero);
       #endif
}

////////////////////////////////////////////////////////
// This Function starts to acquires the gyro
////////////////////////////////////////////////////////
void readGyro()
{
       Read_ADC(ADC_START_ONLY);
}

////////////////////////////////////////////////////////
// This Function returns the acquired the gyro
////////////////////////////////////////////////////////
signed long getGyro()
{
```

```
	gyro_m = Read_ADC(ADC_READ_ONLY);

	return (signed long)(gyro_m - gyro_zero);
}

//////////////////////////////////////////////////////////
// This Function returns the acquired the gyro in float
//////////////////////////////////////////////////////////
float getGyroFloat()
{
//	return (getGyro() * (5/6 * PI) / gyro_zero);
	return ((float)getGyro()  * 0.006817 );
}

//////////////////////////////////////////////////////////
// Gyroscope Calibration Routine
// this function calibrates the acceleration module.
// It calculates the T2 value and writes it into EEPROM
//////////////////////////////////////////////////////////
void calibrateGyro()
{
	long i;

	// iterations to find the highest and the lowest value.
	for (i = 0; i < 0b00111111; i++)
	{
		// reading the gyro value
		gyro_zero += Read_ADC();

		// introducing a delay
		delay_ms(20);
	}

	#ifdef DEBUG
		printf("gyro_sum: %LX | ",gyro_zero);
	#endif

	// dividing by K
	gyro_zero = (long) gyro_zero >> 6;

	#ifdef DEBUG
		printf("gyro_zero: %LX\n\r",gyro_zero);
	#endif

    // writing into the EEPROM gyro_zero
    writeShortEE( EE_GYRO, gyro_zero);
}
```

gyro.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Tilt Sensor - gyro
 *  Students: André Futuro
*  Master: Carlos José D'Ávila (Casé)
 *
 *  File: gyro.h
 *  Date: 22/03/2005
 */

#ifndef _gyro_h_
#define _gyro_h_

const byte EE_GYRO = 0x20;
```

```
// Defining variables
long gyro_m; // contains the gyro mesurament

long gyro_zero;

#include ".\gyro.c"

#endif
```

tilt.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: tiltSensor - tilt
 *  Students: André Futuro
 *  Master: Carlos José D'Ávila (Casé)
 *
 *  File: tilt.c
 *  Date: 19/08/2006
 *
 *  Description:
 *
 * 1 dimensional tilt sensor using a dual axis accelerometer
 * and single axis angular rate gyro.  The two sensors are fused
 * via a two state Kalman filter, with one state being the angle
 * and the other state being the gyro bias.
 *
 * Gyro bias is automatically tracked by the filter.  This seems
 * like magic.
 *
 * Please note that there are lots of comments in the functions and
 * in blocks before the functions.  Kalman filtering is an already complex
 * subject, made even more so by extensive hand optimizations to the C code
 * that implements the filter.
 *
 *
 */

#include ".\tilt.h"

/*
 * Our update rate.  This is how often our state is updated with
 * gyro rate measurements.
 * For now, we do it every time an 8 bit counter running at CLK/(16*4) expires
 * and a counter counts 16. You will have to
 * change this value if you update at a different rate.
 *
 */
//float      const dt = ( 1024.0 * 256.0 ) / 8000000.0;
#define dt 0.007834 // 1/127.66
#define dti 127.66


/*
 * Our covariance matrix.  This is updated at every time step to
 * determine how well the sensors are tracking the actual state.
 */
float P[2][2] = {
 { 1, 0 },
 { 0, 1 },
};
```

```
/*
 * Our two states, the angle and the gyro bias.  As a byproduct of computing
 * the angle, we also have an unbiased angular rate available.   These are
 * read-only to the user of the module.
 */
float angle = 0;
float q_bias = 0;
float rate = 0;


/*
 * R represents the measurement covariance noise.  In this case,
 * it is a 1x1 matrix that says that we expect 0.3 rad jitter
 * from the accelerometer.
 */
//static float const R_angle = 0.3;
#define R_angle 0.3


/*
 * Q is a 2x2 matrix that represents the process covariance noise.
 * In this case, it indicates how much we trust the acceleromter
 * relative to the gyros.
 */
//static float const Q_angle = 0.001;
//static float const Q_gyro = 0.003;
#define Q_angle 0.001
#define Q_gyro      0.003

//#define Q_gyro_dt 0.111201 //0.003 * 37.067 = 0.111201

/*
 * state_update is called every dt with a biased gyro measurement
 * by the user of the module.  It updates the current angle and
 * rate estimate.
 *
 * The pitch gyro measurement should be scaled into real units, but
 * does not need any bias removal.  The filter will track the bias.
 *
  * pitch : degree of deviation from a horizontal plane; "the roof had a steep
pitch"
 * bias : In a statistical bias context, a systematic error in a test score.
 *
 * Our state vector is:
 *
 * X = [ angle, gyro_bias ]
 *
 * It runs the state estimation forward via the state functions:
 *
 * Xdot = [ angle_dot, gyro_bias_dot ]
 *
 * angle_dot = gyro - gyro_bias
 * gyro_bias_dot = 0
 *
 * And updates the covariance matrix via the function:
 *
 * Pdot = A*P + P*A' + Q
 *
 * A is the Jacobian of Xdot with respect to the states:
 *
 * A = [ d(angle_dot)/d(angle)     d(angle_dot)/d(gyro_bias) ]
 *     [ d(gyro_bias_dot)/d(angle) d(gyro_bias_dot)/d(gyro_bias) ]
 *
 *   = [ 0 -1 ]
 *     [ 0  0 ]
 *
```

c

```
 * Due to the small CPU available on the microcontroller, we've
 * hand optimized the C code to only compute the terms that are
 * explicitly non-zero, as well as expanded out the matrix math
 * to be done in as few steps as possible.  This does make it harder
 * to read, debug and extend, but also allows us to do this with
 * very little CPU time.
 *
 * @param q_m : Pitch gyro measurement
 */
//void state_update(long q_m)
void state_update()
{
	/* Declaring variables */
	//float q_real;
	//float q;
	//float Pdot[2 * 2];

	// variables used for the PIC error
//	float new_angle;
//	float new_rate;
//	float new_P[2][2];

	float aux;

	/* Unbias our gyro */
	//q = q_m - q_bias;

	/*
	 * Compute the derivative of the covariance matrix
	 *
	 * Pdot = A*P + P*A' + Q
	 *
	 * We've hand computed the expansion of A = [ 0 -1, 0 0 ] multiplied
	 * by P and P multiplied by A' = [ 0 0, -1, 0 ].  This is then added
	 * to the diagonal elements of Q, which are Q_angle and Q_gyro.
	      Pdot[2 * 2] = {
	            Q_angle - P[0][1] - P[1][0], // 0,0
	            - P[1][1],  // 0,1
	            - P[1][1],  // 1,0
	            Q_gyro    // 1,1
	            };
	 */

	#define Pdot0  (Q_angle - P[0][1] - P[1][0])  /* 0,0 */
	#define Pdot1  (- P[1][1])  /* 0,1 */
	#define Pdot2  (- P[1][1])  /* 1,0 */
	#define Pdot3  (Q_gyro)  /* 1,1 */

	/* Store our unbias gyro estimate
	 *
	 * We need to read get the Gyro value (already offseted to zero)
	 * and convert it to radians
	 */
	//rate = q;
	//rate = q_m - q_bias;
	//rate = (q_m * (5/6 * PI) / gyro_zero) - q_bias;
	//rate = (getGyro()  * (5/6 * PI) / gyro_zero ) - q_bias;
	//rate = (getGyro()  * 0.006817 ) - q_bias;
	rate = getGyroFloat() - q_bias;

	/*
	 * Update our angle estimate
	 * angle += angle_dot * dt
	 *       += (gyro - gyro_bias) * dt
	 *       += q * dt
	 *            += rate * dt
	 */
	angle += rate * dt;
```

```
        // making a calculation that may help me.
        aux = Pdot1 * dt; // == Pdot2 * dt
        /* Update the covariance matrix */
        P[0][0] += Pdot0 * dt;
        P[0][1] += aux; //Pdot1 * dt;
        P[1][0] += aux; //Pdot2 * dt;
        P[1][1] += Pdot3 * dt;

}


/*
 * kalman_update is called by a user of the module when a new
 * accelerometer measurement is available.  ax_m and az_m do not
 * need to be scaled into actual units, but must be zeroed and have
 * the same scale.
 *
 * This does not need to be called every time step, but can be if
 * the accelerometer data are available at the same rate as the
 * rate gyro measurement.
 *
 * For a two-axis accelerometer mounted perpendicular to the rotation
 * axis, we can compute the angle for the full 360 degree rotation
 * with no linearization errors by using the arctangent of the two
 * readings.
 *
 * As commented in state_update, the math here is simplified to
 * make it possible to execute on a small microcontroller with no
 * floating point unit.  It will be hard to read the actual code and
 * see what is happening, which is why there is this extensive
 * comment block.
 *
 * The C matrix is a 1x2 (measurements x states) matrix that
 * is the Jacobian matrix of the measurement value with respect
 * to the states.  In this case, C is:
 *
 * C = [ d(angle_m)/d(angle)  d(angle_m)/d(gyro_bias) ]
 *   = [ 1 0 ]
 *
 * because the angle measurement directly corresponds to the angle
 * estimate and the angle measurement has no relation to the gyro
 * bias.
 *
 * @param ax_m : X acceleration
 * @param az_m : Z acceleration
 */
boolean kalman_update(signed long ax_m, signed long az_m)
{
        // variables
        // don't need these anymore, cause I'm using aux variable angle_err instead
        // float angle_m;
        float angle_err;
        float K_0, K_1;
        float E;
        // don't need these anymore, cause is equal to P[0][0] and P[0][1]
        //float t_0, t_1;
        // don't need these anymore, cause is equal to P[0][0] and P[1][0]
        //float PCt_0, PCt_1;

        // variables used for the PIC error
//      float new_angle;
//      float new_q_bias;
//      float new_P[2][2];


        /* Compute our measured angle and the error in our estimate */
//      #define       angle_m  (atan( (float)(az_m) / (float)(ax_m) ) )
```

```
#define      angle_m  (atan2( (float)(az_m) , (float)(ax_m) ) )

angle_err = angle_m - angle;

/*
 * C_0 shows how the state measurement directly relates to
 * the state estimate.
 *
 * The C_1 shows that the state measurement does not relate
 * to the gyro bias estimate.  We don't actually use this, so
 * we may comment it out.
 */
#define C_0 1
#define C_1 0

/*
 * PCt[2,1] = P[2,2] * C'[2,1], which we use twice.  This makes
 * it worthwhile to precompute and store the two values.
 * Note that C[0,1] = C_1 is zero, so we do not compute that
 * term.
 */
//const float  PCt_0 = C_0 * P[0][0]; /* + C_1 * P[0][1] = 0 */
//const float  PCt_1 = C_0 * P[1][0]; /* + C_1 * P[1][1] = 0 */
//PCt_0 = (C_0 * P[0][0]); /* + C_1 * P[0][1] = 0 */
//PCt_1 = (C_0 * P[1][0]); /* + C_1 * P[1][1] = 0 */
/* Optimizing last lines, because:
 * PCt_0 == P[0][0]
 * PCt_1 == P[1][0]
 */
 #define PCt_0 (P[0][0])
 #define PCt_1 (P[1][0])


/*
 * Compute the error estimate.  From the Kalman filter paper:
 *
 * E = C P C' + R
 *
 * Dimensionally,
 *
 * E[1,1] = C[1,2] P[2,2] C'[2,1] + R[1,1]
 *
 * Note that C_0 is one, so we do not need to multiply, just take the term.
 * Note that C_1 is zero, so we do not compute the term.
 */
//const float  E = R_angle + C_0 * PCt_0 ; /* + C_1 * PCt_1 = 0 */
//E = (R_angle + C_0 * PCt_0); /* + C_1 * PCt_1 = 0 */
E = (R_angle + PCt_0); // optimised last line

/*
 * Compute the Kalman filter gains.  From the Kalman paper:
 *
 * K = P C' inv(E)
 *
 * Dimensionally:
 *
 * K[2,1] = P[2,2]; C'[2,1] inv(E)[1,1]
 *
 * Luckilly, E is [1,1], so the inverse of E is just 1/E.
 */
//const float  K_0 = PCt_0 / E;
//const float  K_1 = PCt_1 / E;
K_0 = (PCt_0 / E);
K_1 = (PCt_1 / E);

/*
 * Update covariance matrix.  Again, from the Kalman filter paper:
 *
```

```
         * P = P - K C P
         *
         * Dimensionally:
         *
         * P[2,2] -= K[2,1] C[1,2] P[2,2]
         *
         * We first compute t[1,2] = C P.  Note that:
         *
         * t[0,0] = C[0,0] * P[0,0] + C[0,1] * P[1,0]
         *
         * But, since C_1 is zero, we have:
         *
         * t[0,0] = C[0,0] * P[0,0] = PCt[0,0]
         *
         * This saves us a floating point multiply.
         */
        //const float  t_0 = PCt_0; /* C_0 * P[0][0] + C_1 * P[1][0] */
        //const float  t_1 = C_0 * P[0][1]; /* + C_1 * P[1][1]  = 0 */
        //t_0 = PCt_0;    /* C_0 * P[0][0] + C_1 * P[1][0] */
        #define t_0 (PCt_0)
        //t_1 = C_0 * P[0][1];  /* + C_1 * P[1][1]  = 0 */
        #define t_1 (P[0][1])

        P[0][0] -= K_0 * t_0;
        P[0][1] -= K_0 * t_1;
        P[1][0] -= K_1 * t_0;
        P[1][1] -= K_1 * t_1;

        /*
         * Update our state estimate.  Again, from the Kalman paper:
         *
         * X += K * err
         *
         * And, dimensionally,
         *
         * X[2] = X[2] + K[2,1] * err[1,1]
         *
         * err is a measurement of the difference in the measured state
         * and the estimate state.  In our case, it is just the difference
         * between the two accelerometer measured angle and our estimated
         * angle.
         */

        angle += K_0 * angle_err;
        q_bias += K_1 * angle_err;
}

// Function to zero kalman update and tilt
void zeroTilt()
{
        P[0][0] = 1;
        P[0][1] = 0;
        P[1][0] = 0;
        P[1][1] = 1;

        angle = 0;
        q_bias = 0;
        rate = 0;
}
```

tilt.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
```

```
 *  Part: tiltSensor - tilt
 *  Students: André Futuro
 *  Master: Carlos José D'Ávila (Casé)
 *
 *  File: tilt.h
 *  Date: 19/08/2005
 *
 *  Description:
 *
 * 1 dimensional tilt sensor using a dual axis accelerometer
 * and single axis angular rate gyro.  The two sensors are fused
 * via a two state Kalman filter, with one state being the angle
 * and the other state being the gyro bias.
 *
 * Gyro bias is automatically tracked by the filter.  This seems
 * like magic.
 *
 * Please see the file tilt.c for more details on the implementation.
 * This header only has comments for the use of the module, not the
 * inner workings.
 *
 *
 */
#ifndef _tilt_h_
#define _tilt_h_


#include ".\tilt.c"

#endif
```

ANEXO 6 – FIRMWARE DO CONTROLE

Abaixo se encontra o firmware para o microcontrolador principal. O código foi feito utilizando o compilador de C para PIC PCW verão 3.43. O microcontrolador alvo é o PIC 16F876A da Microchip.

Pendulum.c

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: Motor PWM
 *  Student: André Futuro
 *  Master: Carlos José D'Ávila (Casé)
 *
 *  File: pendulum.c
 *  Date: 09/11/2005
 */

//including the header file
#include ".\pendulum.h"

void Init();
void PID();
void Set_Constants();


//*****************************
// Variables
//*****************************

///////////////
// boolean (bit) variables
// ...
short flag1;
// Flag used to define if the control shall be done.
short do_PID;
// ...
short int_flag;

// angle got from other PIC
signed char angle;

// terms
signed char en0, en1, en2, en3, term1_char, term2_char, off_set;
// temp varible
unsigned char temp;
// 2 bytes (short)
signed long temp_int;
// Constants for integral, derivative and proportional terms
float ki, kd, kp, integral_term;
signed long SumE_Min, SumE_Max, SumE, proportional_term, derivative_term, un, es,
u, v;
signed int32 Cn;

boolean a=0;


//     __CONFIG      _CP_OFF  &  _CPD_OFF  &  _BOD_OFF  &  _MCLRE_ON  &  _WDT_OFF  &
_INTRC_OSC_NOCLKOUT & _FCMEN_ON
```

```
//*************************************************************************
// Positional PID 256 Hz
//*************************************************************************

//*************************************************************************
//Main() - Main Routine
//*************************************************************************
void main()
{
	//////////////
	// setting ports

	//Initialize 16F876 Microcontroller
	Init();

	//Get PID coefficients ki, kp and kd
	Set_Constants();

	//Loop Forever
	while(1)
	{
		if(do_PID){
			PID();
		}
	}
}

//*************************************************************************
//Init - Initialization Routine
//*************************************************************************
void Init()
{
	//////////////
	// setting ports
	PORTA = 0xFF;
	// Set RA5, RA4, RA2 as output, rest input
	set_tris_a(0b00000000);
	PORTB = 0;
	// Set RB as outputs
	set_tris_b(0b11110000);
	PORTC = 0;
	//TRISC = 0b10111001;            // Set RC1, RC2 and RC6 (TX) as outputs,
rest inputs
	set_tris_c(0b10101001);

	//////////////
	// Configure CCP1 as a PWM
	// this command will put 1100 into register CCP1CON, meaning that PWM mode is
on.
	//setup_comparator(NC_NC_NC_NC);
	set_pwm1_duty(0);
	set_pwm2_duty(0);
	setup_ccp1(CCP_PWM);
	setup_ccp2(CCP_PWM);
	// The cycle time will be (1/clock)*4*t2div*(period+1)
	// In this program clock=20000000 and period=255 (below)
	// The cycle time is:
	//     (1/20000000)*4*1*255 =  51.0 us or 19.607 khz
	//setup_timer_2(T2_DIV_BY_1, 0x3F, 1); // for 7072000MHz oscilator
	setup_timer_2(T2_DIV_BY_1, 255, 1);
	set_pwm1_duty(0);
	set_pwm2_duty(0);


	////////////
	// Initializing the Analog Ports, used to get Kp, Kd and Ki
	setup_adc_ports(AN0_AN1_AN3);
	setup_adc(ADC_CLOCK_DIV_2);
```

```
	//////////////////
	// Setting the states variables to 0
	en0 = en1 = en2 = en3 = term1_char = term2_char =0;
	ki = kd = 0;
	kp = off_set = 0;
	temp_int = proportional_term = integral_term = derivative_term = un =0;
	SumE_Max = 32700;
	SumE_Min = 1 - SumE_Max;
	angle = 0;
	// Allowed to do PID function
	do_PID = 1;


	//////////////////
	// Start (request) first angle capture
	//putc(0);

	//////////////////
	// Setting Timer 0, used to define the rate of runing the control
	// Should ocurr every 1/256 Hz.
	// The cycle time will be (1/clock)*4*RTCC_DIV*(256-Preload)
	// For the three possible clocks:
	//     (1/7072000)*4*32*(256-39) = 3.9 ms or 256 hz
	//     (1/18000000)*4*128*(256-118) = 3.9 ms or 256 hz
	//     (1/20000000)*4*128*(256-102) = 3.9 ms or 256 hz
	//     (1/20000000)*4*256*(256-103) = 7.8 ms or 128 hz
	setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // clock = 20000000
	set_timer0(100);
	#define Fs 128 // defining frequency

	//////////////////
	// Setting the interruptions
	enable_interrupts(INT_RDA);      // read data USART
	enable_interrupts(INT_TIMER0);   // start control cicle
	enable_interrupts(GLOBAL);              // global interruptions
}


//***************************************************************************
// Set Constants Kp, Kd, Ki
//***************************************************************************
// This routine reads the contants Kp, Kd and Ki from the Analog inputs
void Set_Constants()
{
	// Read Kp on RA0 (AN0)
	set_adc_channel(0);
	Kp = read_adc() / 24;
//     Kp = 1;

	// Read Kd on RA1 (AN1)
	set_adc_channel(1);
	Kd = read_adc() / 32;
//     Kd = 1;

	// Read Ki on RA3 (AN3)
	set_adc_channel(3);
	Ki = read_adc() / 32;
//     Ki = 1;

}

char Kx=2;

//***************************************************************************
// Set Constants Kp, Kd, Ki
//***************************************************************************
```

```
// This routine reads the contants Kp, Kd and Ki from the Analog inputs
void Set_Constants_with_interval()
{

       switch(Kx)
       {
              case 0:
                     // Read Kp on RA0 (AN0)
                     Kp = (float)((float)read_adc(ADC_READ_ONLY) / 16);
                     // preparing for next reading
                     set_adc_channel(1);
                     Read_ADC(ADC_START_ONLY);
                     Kx = 1;
                     break;
              case 1:
                     // Read Kp on RA1 (AN1)
                     Kd = (float)((float)read_adc(ADC_READ_ONLY) / 16);
                     // preparing for next reading
                     set_adc_channel(3);
                     Read_ADC(ADC_START_ONLY);
                     Kx = 2;
                     break;
              case 2:
                     // Read Kp on RA3 (AN3)
                     Ki = (float)((float)read_adc(ADC_READ_ONLY) / 16);
                     // preparing for next reading
                     set_adc_channel(0);
                     Read_ADC(ADC_START_ONLY);
                     Kx = 0;
                     break;
       }

}



//************************************************************************
// PID Control Routine
//************************************************************************
// The from of the PID is C(n) = K(E(n) + (Ts/Ti)SumE + (Td/Ts)[E(n) - E(n-3)])
//                         C(n) = KpE(n) + (Ki*Ts)SumE + (Kd/Ts)[E(n) - E(n-3)]
//                  where : Ts = 1/128
void PID()
{

       ////////////////
       // Calculate the integral term : (Ts/Ti)*SumE
       // (Ki*Ts)SumE

/*     // SumE is the summation of the error terms
       SumE = SumE + en0;
       // Test if the summation is too big
       if(SumE > SumE_Max)
              SumE = SumE_Max;
       // Test if the summation is too small
       if(SumE < SumE_Min)
              SumE = SumE_Min;

      // Integral term is (Ts/Ti)*SumE where Ti is Kp/Ki and Ts is the sampling
period
    // Actual equation used to calculate the integral term is Ki*SumE/(Kp*Fs*X)
       // where X is an unknown scaling factor and Fs is the sampling frequency
       integral_term = SumE * Ki;                     // Multiply Ki
       integral_term = integral_term / 128;    // Divide by the sampling frequency
*/
       ////////////////
       // Calculate the integral term : I(n+1) = I(n) + Ki*(1-es(n))*E(n)/Fs
```

```
//printf("| %f | %ld | %d | %f |\r\n",integral_term,(signed long)(Ki * (float)/*(1
- es) */ en0 / (float)Fs), en0, Ki);

      integral_term = integral_term + (float)(Ki * (float)(1 - es) * en0 /
(float)Fs);


      /////////////////
      // Calculate the derivative term
      // D = (Kd/Ts)[E(n) - E(n-3)] = (Kd*(Fs/3))[E(n) - E(n-3)]

      derivative_term = (signed long) (kd * (float)(Fs * (en0 - en3)) / 3);


      /////////////////
      // Calculate the proportional term
      // KpE(n)
      proportional_term = (kp * en0);

      ///////////////////
    // C(n) = K(E(n) + (Ts/Ti)SumE + (Td/Ts)[E(n) - E(n-1)])

      Cn = proportional_term + (signed long) integral_term + derivative_term; //
Sum the terms
//    Cn = Cn / 512;                                        // scale

      // getting the real value for the control
      v = Cn;

      // C(n) = KpE(n) + (Ki*Ts)SumE + (Kd/Ts)[E(n) - E(n-3)]
      //Cn = proportional_term + integral_term + derivative_term;// Sum the terms
      // scaling the signal
      //Cn = Cn / 128;

      // Used to limit duty cycle not to have punch through
      if(Cn >= 1022) {
            Cn = 1022;
      }
    if(Cn <= -1022) {
            Cn = -1022;
      }

      // getting the adjusted value for the control.
      u = Cn;

      // calculating es for the retro-alimentation
      es = u-v;

      ///////////////
      // Set the speed of the PWM

      // Position = 0
      // Motor should stay still and set the duty cycle to 0
      if (Cn == 0)
      {
          set_pwm1_duty(0);
          set_pwm2_duty(0);
      }

      // Position > 0
      // Motor should go forward and set the duty cycle to Cn
      if(Cn > 0)
      {
            // Motor is going forward
            MOTOR_A_FORWARD;
            MOTOR_B_FORWARD;
            set_pwm1_duty(Cn);
            set_pwm2_duty(Cn);
```

```
            // Used to stop the pendulum from continually going around in a circle
            off_set = off_set ++;      // the offset is use to adjust the angle of
the pendulum to slightly
            if(off_set > 55) {         // larger than it actually is
                  off_set = 55;
            }
      }

      // Position < 0
      // Motor should go backwards and set the duty cycle to Cn
      else
      {
            // Motor is going backwards
            MOTOR_A_BACKWARD;
            MOTOR_B_BACKWARD;
            set_pwm1_duty(-Cn);
            set_pwm2_duty(-Cn);

            // Used to stop the pendulum from continually going around in a circle
            off_set = off_set --;
            if(off_set < -55){
                  off_set = -55;
            }
      }

      ////////////////////
      // Shift error signals
      en3 = en2;
      en2 = en1;
      en1 = en0;
      en0 = 0;
      do_PID = 0;                      // Done
      // Test flag RA4 to measure the speed of the loop
      //output_low(PIN_A4);

      Set_Constants_with_interval();

      return;
}


//////////////////////////////////////////////////////////////////////////
// Interrupt function on TIMER 0
//////////////////////////////////////////////////////////////////////////
// This routine is called when it is time that PID control has to be done
#INT_TIMER0
void timer0_handler()
{
      // Preload value in timer0: 101
      set_timer0(100);

    // Clear Int Flag
      disable_interrupts(INT_TIMER0);

// Test flag RA2 to measure the speed of the loop
if (a)
    output_low(PIN_A4);
else
    output_high(PIN_A4);
a = !a;



      // Store to error function asuming no over-flow
//    en0 = angle + off_set/8;
    en0 = angle;
```

```
	// Allowed to do PID function
	do_PID = 1;

	//Check if error is too large (positive) or
	//Check if error is too large (negative)
	if((en0 > 125) || (en0 < -125))
	{
		// Stop PWM
//		setup_ccp1(CCP_OFF);
//		setup_ccp2(CCP_OFF);

//		PWM_OFF;

		// Clear all PID constants
//		en0 = en1 = en2 = en3 = term1_char = term2_char = off_set = 0;
//		Cn = integral_term = derivative_term = SumE = /*RA4 =*/ 0;
		// Stop doing PID
//		do_PID = 0;
	}

	// cleaning if there is an error in the USART data reading
	if (OERR == 1)
	{
		CREN = 0;
		CREN = 1;
	}

	enable_interrupts(INT_TIMER0);
}

//***************************************************************************
// Interrupt function on USART RS232 - RX
//***************************************************************************
// This routine is called when receiving the angle from the other PIC
#int_rda
void serial_isr()
{
	// disabling interruptions
	disable_interrupts(INT_RDA);
	// receiving angle
	angle = getc();
//getc();
//temp_int++;
//angle = temp_int>>8;
//angle=10;
	// re-enabling interruptions
	enable_interrupts(INT_RDA);
}
```

Pendulum.h

```
/*  Universidade Federal do Rio de Janeiro
 *  Engenharia Eletronica e de Computação
 *  Projeto Final: Ballancing Robot
 *  Part: PENDULUM
 *  Student: André Futuro
 *  Master: Carlos José D'Ávila (Casé)
 *
 *  File: pendulum.h
 *  Date: 08/11/2005
 */

#ifndef _pendulum_h_
#define _pendulum_h_

#include <16F876A.h>
```

```
#device adc=10

//#use delay(clock=7072000)
#use delay(clock=20000000)
#fuses HS, NOWDT, NOPROTECT, NOLVP, PUT

//#use rs232(baud=57600,parity=N,xmit=PIN_C6,rcv=PIN_C7)
//#use rs232(baud=38400,parity=N,xmit=PIN_C6,rcv=PIN_C7)
#use rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)
//#use rs232(baud=9600,parity=N,xmit=PIN_C6,rcv=PIN_C7)

// declaring registers
#byte PORTA = 0x05
#byte PORTB = 0x06
#byte PORTC = 0x07

#byte TRISA = 0x85
#byte TRISB = 0x86
#byte TRISC = 0x87

#byte CMCON0 = 0x9C

#bit RCIF = 0x0c.5
#bit TXIF = 0x0c.4

#bit OERR = 0x18.1
#bit CREN = 0x18.4

#bit TOC = 0x81.5


#define MOTOR_A_FORWARD   output_low(PIN_A2)
#define MOTOR_A_BACKWARD  output_high(PIN_A2)
#define MOTOR_B_FORWARD   output_low(PIN_A5)
#define MOTOR_B_BACKWARD  output_high(PIN_A5)


#endif
```

ANEXO 7 – ESPECIFICAÇÕES DO SISTEMA

Segue abaixo as especificações do sistema:

Peso	*49,3 kg*
Altura	*50,3 cm*
Largura	*26,4 cm*
Comprimento	*15,2 cm*
Número de placas de circuito impresso	*2*
Bateria 1	*9V*
Bateria 2	*12V, 1,2Ah*
Bateria 3	*12V, 1,2Ah*
Corrente média Bateria 1	*60 mA*
Corrente máxima Baterias 2 e 3 juntas	*400 mA*
Corrente máxima Bateria 1	*100 mA*
Corrente máxima Baterias 2 e 3 juntas	*4 A*
Torque dos Motores	*$36,5x10^{-3}$ Nm/A*
Caixa de redução dos motores	*19,1:1*
Redução das engrenagens	*1:1*