PUC
RIO

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

FELIPE MAIMON

PROJETO DE UM SISTEMA ELETRÔNICO PARA O CONTROLE DE MOTORES DE ALTA POTÊNCIA POR PWM

TRABALHO DE FIM DE CURSO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

DEZ/2004

CTC Centro Técnico Científico

Agradecimentos

Gostaria de agradecer a todos que me ajudaram neste projeto, dando novas idéias, ajudando a montar os circuitos ou simplesmente tentando aprender tudo o que eu fiz para que futuramente esse projeto ainda seja melhorado. Entre essas pessoas posso citar especialmente Felipe Scofano, Cláudio Duvivier, Júlio Guedes, Rafael Caram, Felipe Garschagen, Filipe Sacchi, Bruno Favoreto, Leonardo Salvini, Rodger Moulin, Gustavo Lima, Ilana Nigri, Mariana Bystronski e Marcio Barros, todos eles integrantes da equipe RioBotz a qual orgulhosamente fiz parte. Gostaria de agradecer também ao grande Prof. Marco Antonio Meggiolaro, coordenador da equipe e meu orientador, aos professores Prof. Mauro Schwanke, Prof. Mauro Speranza, Prof. Luiz Antonio Gusmão, Prof. Francisco Rubens, Prof. Moisés Szwarcman, por tudo que me ensinaram. Outros agradecimentos vão para meus pais, Zeev Lucyan Maimon e Maria Ines Brito W. Maimon, meus grandes amigos, Felipe Belo, Ricardo Moreno, Rubens, Luiz, Daniel Camerini e Thiago Salcedo. E não poderia esquecer da própria PUC-Rio por possibilitar tal projeto, além de prover toda a ajuda, suporte e infra-estrutura necessários, com o CETUC e ITUC, além do Laboratório de Controle e Automação.

Resumo

Este projeto apresenta a concepção de um conjunto de circuitos eletrônicos de controle e de potência para promover uma solução completa para controle de velocidade de motores de corrente contínua (CC) de alta potência, através de um controle remoto comum de aeromodelos.

O circuito de potência é formado por uma Ponte H, capaz de conduzir até 160A continuamente, podendo suportar picos de mais de 400A, sem dissipadores, sendo refrigerados apenas por um ventilador, tornando o circuito leve. Este circuito recebe sinais de velocidade e direção de acionamento de uma placa de controle e aciona a Ponte H apropriadamente, incorporando medidas de segurança para evitar condições que poderiam danificar o circuito no caso de problemas no circuito de controle. O circuito de potência também possui um regulador de tensão para alimentar a si próprio e o circuito de controle.

O circuito de controle recebe os sinais do receptor de rádio, os interpreta e os converte em sinais para o acionamento de dois motores (saída para duas eletrônicas de potência). O controle de velocidade do motor é feito através de PWM (Pulse Width Modulation), que funciona modificando a tensão média sobre o motor e, portanto, sua velocidade, já que esta depende diretamente da tensão aplicada. O sinal de PWM utilizado possui freqüência de aproximadamente 4 kHz, ou seja, a cada 250μs, o circuito de sinais envia um pulso, juntamente com o sinal de direção, para o circuito de potência.

O conjunto é testado em um sistema robótico com motores de alta potência, apresentando resultados satisfatórios no controle de velocidade, sem aquecimento significativo das placas mesmo sob condições de correntes médias de até 100A.

Abstract

This work presents the development of control and power electronics to provide velocity control of high power direct current (DC) motors, using a standard radio control system as an input.

The control circuit receives signals from a radio control receiver, generating proper signals to drive two DC motors through outputs to two power circuits. The speed control is achieved through PWM (Pulse Width Modulation), which works by changing the average voltage applied to the motor and, therefore, its speed, since they are directly proportional. The PWM signal has an approximate frequency of 4 kHz, meaning that the control circuit sends a pulse to the power circuit every 250 microseconds, along with the direction signal.

The power circuit is built around an H bridge, able to handle up to 160A continuously, and withstanding current peaks higher than 400A. It only uses a cooling fan to dissipate the heat, without the need of heat sinks, resulting in a very light and portable system. The power circuit receives velocity and direction signals from the control board, acting on the H bridge accordingly. It also incorporates safety measures to prevent damage to the circuit caused by unexpected problems in the signals from the control board. The power circuit also includes a voltage regulator to provide low voltage power to both itself and the control board.

The circuits are tested in a robotic system with high power DC motors, presenting satisfactory speed control performance, without any significant heating of the boards for average currents up to at least 100A.

Sumário

Índice de Figuras

Índice de Tabelas

1 – Introdução

Em várias aplicações da robótica moderna, existe a necessidade de motores elétricos potentes, muitas vezes com potência elétrica especificada maior do que 1 kW. Devido a isso, é necessária a utilização de circuitos com maior complexidade para o controle desses motores. Uma simples ponte H feita de chaves ou relés não conseguiria suportar toda a corrente gerada no motor em situações em que o mesmo estivesse em alta velocidade em uma direção sendo revertido completamente na direção oposta. Nesse caso, uma alta corrente elétrica seria gerada e, com isso, uma dissipação de potência que poderia danificar o circuito.

Para essas aplicações de alta potência, foram desenvolvidos, neste projeto, um conjunto de circuitos modulares – circuito de controle e de potência – que, juntos, provêm uma solução completa para controle destes motores, através de um controle remoto comum de aeromodelos.

Esta dissertação está organizada em cinco capítulos. No Capítulo 2 é feita uma introdução à teoria de motores CC, além de mostrar as formas de controlar estes tipos de motores. O Capítulo 3 descreve a parte de potência do circuito de acionamento dos motores. No Capítulo 4 temos toda a descrição do hardware e software do circuito de controle dos. E, finalmente, no Capítulo 5, as conclusões do trabalho.

2 – Motores CC

2.1. Funcionamento de motores CC

Para entender os modos acionamento de motores CC (Corrente Contínua) é necessário entender seus fundamentos. Os motores são normalmente formados por um imã permanente fixo, que forma o estator, e um rotor que possui vários enrolamentos. Esse enrolamento gera um campo magnético que, em conjunto com o campo do imã, gera um torque no rotor. Para que se tenha uma saída com torque constante, os enrolamentos devem ser comutados continuamente, tarefa feita pelo comutador presente no rotor e pelas escovas, presa na carcaça do motor.

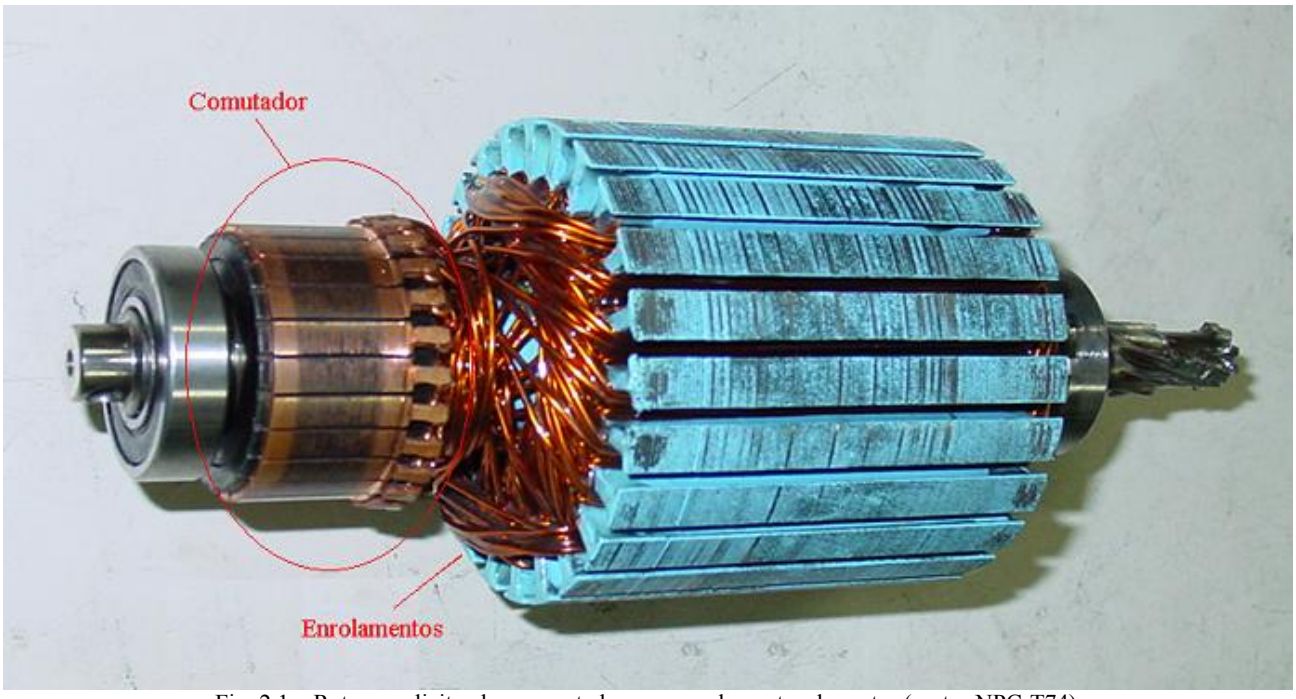

Fig. 2.1 – Rotor, explicitando o comutador e os enrolamentos do motor (motor NPC-T74).

Eletricamente, o motor CC pode ser modelado por uma resistência, uma indutância e uma fonte de tensão, cujo valor é diretamente proporcional à velocidade do motor, todos em série.

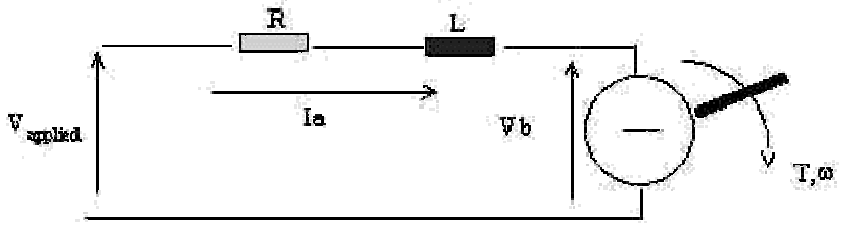

Fig. 2.2 - Modelagem elétrica de um motor DC

Como se pode perceber pela Figura 2.2, a corrente no motor é expressa pela equação (1):

$$V_a = L \cdot \frac{dI_a}{dt} + R \cdot I_a + K \cdot \omega \tag{1}$$

onde,

- V_a é a tensão aplicada nos terminais do motor;
- I_a é a corrente passando através do motor;
- K é uma constante que, multiplicada pela velocidade do eixo, gera a tensão induzida quando o rotor está girando (efeito gerador);
- ω é a velocidade angular do eixo.

2.2. Acionamento dos motores

Ao analisar a modelagem dos motores CC, fica claro que variando a tensão nos terminais do motor, varia-se sua velocidade e caso seja necessário mudar a direção do motor, é necessário inverter os terminais do motor ou gerar uma tensão negativa entre seus terminais. Portanto, um controle simples em que um relé fica encarregado de inverter os terminais do motor e um regulador de tensão linear gera a tensão necessária para o acionamento do motor pode ser facilmente implementado.

Apesar de ser bem simples, essa abordagem tem problemas sérios, sendo o principal deles a baixa eficiência do circuito, especialmente em baixas tensões de acionamento, já que toda a energia que não é utilizada pelo motor é dissipada no regulador linear. Por exemplo, caso esse método fosse utilizado para controlar o motor NPC-T64 [1], que possui uma resistência interna de aproximadamente 0.22 Ω e é acionado por uma tensão de 24 V, fazendo-o girar à metade de sua velocidade máxima, tendo 12 V nos seus terminais, teríamos uma corrente de 55 A passando também pelo regulador de tensão, que dissiparia os 12 V restantes. Isso geraria no total uma dissipação de potência de 660 W, o que seria inviável, já que essa potência seria desperdiçada em forma de calor que provavelmente queimaria o regulador de tensão caso ele não fosse grande o suficiente para dissipar eficientemente o calor para suportar tamanha potência.

Esse método ainda possui outro problema, este devido à indutância do motor. Ao utilizar o relé para inverter os terminais do motor, a corrente estará sendo desligada quase instantaneamente e, devido a isso, uma tensão muito grande será gerada pela indutância, gerando arcos nos terminais do relé, o que diminui sua vida útil.

2.3. Modulação por Largura de Pulso (PWM)

Para conseguir uma eficiência maior e circuitos menores, é necessário utilizar um método diferente para variar a velocidade do motor, conhecido pela sigla em inglês de PWM (*Pulse Width Modulation*, Modulação por Largura de Pulso). Esse método consiste em ligar e desligar o motor, numa freqüência fixa, através de uma chave, normalmente algum tipo de transistor (Bipolar ou MOSFET), fazendo com que o motor gire numa velocidade proporcional à relação entre o tempo ligado (T_{on}) e período (T). Essa relação é chamada de *Duty Cycle* (D) e, se multiplicada pela tensão de pico (tensão de alimentação do motor), temos uma tensão média que equivale à tensão DC que teria que ser aplicada para fazer o motor girar à mesma velocidade.

Fig. 2.3 – (a) PWM com D próximo de 100%; (b) D = 50%; (c) D próximo de 0%.

Na Figura 2.3, temos três formas de ondas geradas com PWM, numa freqüência fixa. A Figura 2.3(a) mostra o *PWM* com um *D* de quase 100%, ou seja, o motor se comporta como se estivesse recebendo quase toda a sua tensão nominal. Já na Figura 2.3(b) o motor estaria girando à aproximadamente a metade de sua velocidade máxima e na 2.3(c) estaria em uma velocidade muito baixa.

A eficiência do circuito é, idealmente, 100%, já que a chave que acionaria o motor não tem perdas. Na prática, isto não ocorre devido às perdas, tanto estática quanto dinâmicas, nos transistores utilizados. Apesar disso, a eficiência de um controlador bem projetado está normalmente acima dos 90%.

2.4. Ponte H

A reversão do motor ainda é um problema utilizando os métodos citados anteriormente. As duas formas apresentadas para implementar a reversão são inverter os terminais do motor ou gerar uma tensão negativa para acioná-lo. Inverter os terminais é muito complexo, envolvendo a utilização de chaves ou relés, o que é impraticável, devido à dificuldade de encontrá-los. Há também o problema do alto preço de componentes qualificados para suportar altas correntes, além do fato de ser necessário desconectar os fios momentaneamente para que os terminais sejam trocados. Já a opção de gerar tensão negativa também é inviável devido à complexidade do circuito necessário para isto, tendo uma bateria como alimentação principal do circuito.

Uma terceira opção é a utilização de uma ponte H. Ela possui este nome devido à forma em que as chaves e a carga estão dispostas no circuito. Ela é a melhor escolha por não ser necessário criar tensões negativas e nem desconectar os terminais para que eles sejam trocados.

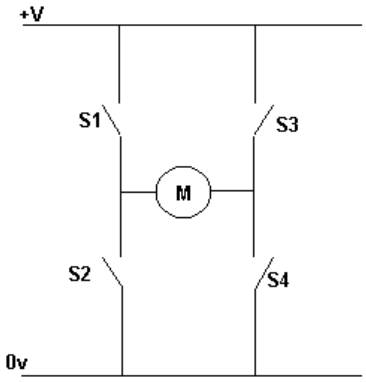

Fig. 2.4 – Ponte H básica

A Figura 2.4 mostra uma ponte H básica. Fechando as chaves S1 a S4 numa ordem fixa, pode-se fazer o motor girar para frente, para trás, assim como frear o motor. Outra vantagem é o fato de as chaves poderem ser transistores que podem ser chaveados rapidamente, suportam facilmente altas correntes, e são baratos e acessíveis.

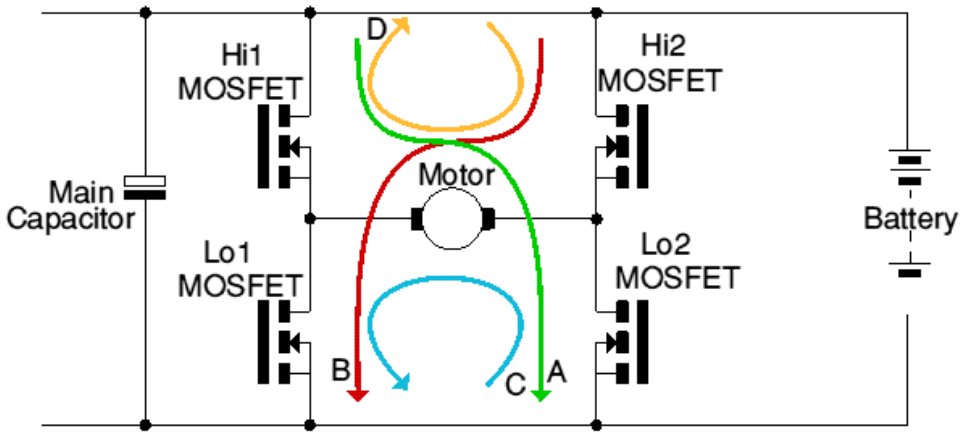

Fig. 2.5 – Ponte H e seus modos de funcionamento.

Na Figura 2.5 as chaves já foram substituídas por transistores tipo MOSFET, mostrando um circuito básico da ponte H. Para fazer o motor girar para frente, a corrente sai da bateria, passa pelo MOSFET Hi1, passa pelo motor e por Lo2, como mostrado na curva A. Para girar para trás, os MOSFETs Hi1 e Lo2 devem ser desativados, ativando-se o Hi2 e Lo1, fazendo a corrente percorrer o caminho B.

Para frear o motor há duas opções, ou usar os MOSFETs Lo1 e Lo2 (curva C), ou usar Hi1 e Hi2 (curva D). Em ambas as opções os terminais do motor são curtados. Esse efeito de frenagem ao curtar os terminais é chamado de freio motor e acontece devido ao fato de toda a energia do motor estar sendo dissipada somente na resistência interna deste, que é, normalmente, muito pequena, fazendo com que a energia se dissipe rapidamente, parando efetivamente o motor.

Um grande problema de pontes H é um efeito chamado *shoot-through*, que acontece ao acionar as duas chaves de um mesmo lado da ponte. Quando isso acontece, a bateria sofre um curto, gerando uma descarga muito grande de corrente que, em geral, faz com que as chaves sejam destruídas completamente. Na Seção 3.3 serão discutidas técnicas para evitar esse efeito.

Adicionar PWM à ponte H é bastante simples. Ao girar para frente, basta manter Hi1 permanentemente ligado e Lo2 ligado durante o T_{on}, não se esquecendo de manter Hi2 e Lo1

desligados, e, durante o resto do período, desligar Lo2. Contudo, devido à indutância existente no motor, a corrente tenta se manter fluindo e constante enquanto Lo2 estiver desligado e, para isso, Hi2 deve ser acionado neste intervalo de tempo (caminho D na Figura 2.5), que, apesar de estar ligando os dois terminais do motor juntos, mantém um caminho de baixa impedância para a corrente do motor, com perdas mínimas de energia. Apesar de isso frear o motor, é esse efeito de ligar e desligar o motor que gera a tensão média vista por ele e, com isso, a velocidade variável. Algo similar deve ser feito para que o motor gire para trás, acionando Hi2 e Lo1 durante o T_{on}, e durante o resto do período desligando Lo1 e ligando Hi1.

3 – Circuito de potência

As chaves que acionam a ponte-H, discutida no capítulo anterior, foram inicialmente consideradas ideais, ou seja, não possuem perda alguma. Como as chaves não são ideais na prática, devem-se analisar as perdas existentes nas chaves reais.

Existem três tipos perdas em chaves reais. São elas:

- Perdas na condução – perdas relativas ao comportamento da chave quando fechada (ligada), devido a algum efeito que gere uma queda de tensão sobre a chave;
- Perdas na comutação – perdas relativas à fase de comutação da chave, ou seja, durante a fase de ligar ou desligá-la;
- Perdas no acionamento – perdas existentes nos circuitos de acionamento das chaves.

As perdas de condução e comutação consomem parte da potência que seria disponível para a carga, que é transformada em calor, diminuindo a eficiência do circuito, além de diminuir a capacidade de acionamento dos circuitos já que todas as chaves reais possuem um limite de temperatura de trabalho. Esse efeito da temperatura normalmente é atenuado com técnicas de refrigeração, mas sempre existirá nos sistemas disponíveis.

Nos tópicos abaixo, os tipos de chaves eletrônicas (transistores) mais comumente utilizadas – transistores bipolares de junção e Mosfets – serão analisadas, descrevendo suas vantagens, desvantagens e as perdas existentes em cada um.

3.1. Transistor Bipolar de Junção (BJT)

O BJT foi o primeiro tipo de transistor a ser desenvolvido e, devido a isso, é o de construção mais simples e barato. O funcionamento dele é baseado na amplificação de correntes, ou seja, ao injetar uma corrente na sua entrada (Base), é permitida a passagem de uma corrente, proporcional à corrente de base, entre os terminais de saída (Coletor e Emissor). A constante de proporcionalidade é chamada de ganho de corrente (β ou h_{fe}). Caso a corrente de base seja nula, o BJT estará em corte, ou seja, chave aberta (desligada) e, caso a corrente de base seja maior ou igual que a necessária para acionar a carga, ele estará na saturação, que é a situação equivalente a uma chave fechada (ligada).

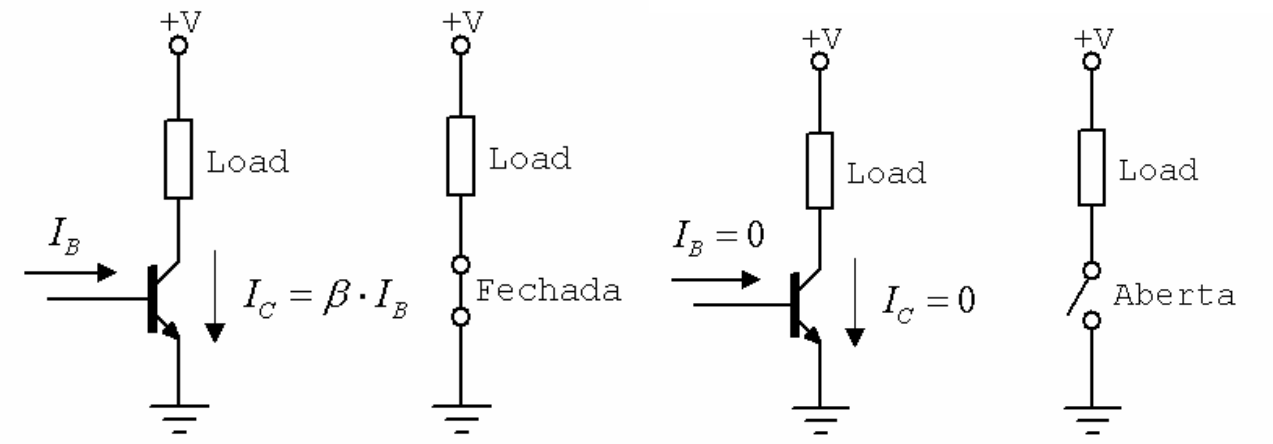

Fig. 3.1 – Modos de operação para chaveamento de um transistor bipolar.

Quando um BJT está na saturação, a tensão entre o coletor e o emissor fica, na maioria dos dispositivos de potência, entre 0,5 V e 1 V. Essa tensão, multiplicada pela corrente que passa pelo circuito, é a potência dissipada durante a condução sendo, portanto, a perda de condução do transistor. No caso de motores de alta corrente, pode-se facilmente demonstrar a impossibilidade do uso de BJTs, já que a corrente a qual o circuito deve suportar continuamente pode chegar a 160 A,

fazendo com que a perda de potência em cada um dos dois dispositivos conduzindo ao mesmo tempo seja de pelo menos 80 W, o que exigiria dissipadores de calor muito grandes e pesados. A melhor forma de contornar esse problema seria utilizar vários transistores em paralelo. Caso fossem utilizados quatro transistores idênticos em paralelo, a corrente seria distribuída igualmente entre todos – 40 A em cada – fazendo com que a potência em cada transistor fosse de pelo menos 20 W, um valor ainda muito grande para o uso sem dissipadores. Além disso, o BJT possui um valor máximo de corrente que não deve ser ultrapassado, podendo ocasionar queima instantânea do transistor caso ocorra.

Outro grande problema do BJT é seu acionamento. Como ele é feito através de corrente, é necessária uma grande corrente em sua base para que ele permita a passagem de altas correntes entre seus terminais, já que os BJTs de potência normalmente possuem β relativamente pequeno, entre 20 e 50. Com isso, a corrente necessária na base para a condução de, e.g., 40 A em cada transistor é de no mínimo 2 A, além de ser necessário que as formas de ondas dessa corrente sejam complexas na transição, fazendo com que o circuito de acionamento se torne extremamente complexo.

Quanto à potência dissipada no transistor durante o acionamento, ela é devido à tensão existente entre a base e o emissor do transistor, aproximadamente 0,7 V, fazendo com que a potência dissipada fique em, pelo menos, 1,4 W seguindo o valor de 2 A do exemplo acima. Apesar de ser muito menor que as perdas na condução, esse valor ainda está muito próximo dos valores máximos permitidos em encapsulamentos comuns de transistores, o que se torna mais um motivo que inviabiliza o seu uso.

Por último, há também a perda na comutação. Ela acontece devido ao transistor não conseguir mudar instantaneamente do estado de corte (sem corrente e toda tensão no transistor) para a saturação (tensão no transistor zero, idealmente, e a corrente fluindo) e vice versa. Nesse intervalo

de tempo haverá uma maior dissipação de calor. As formas de onda vistas no transistor no melhor caso, ou seja, quando o circuito é totalmente resistivo, podem ser vistas na Figura 3.2(a). O pior caso pode ser visto na Figura 3.2(b), caso a impedância complexa (indutores ou capacitores) do circuito seja muito grande.

Pela Figura 3.2, pode-se ver claramente que a perda na comutação depende dos tempos de transição t_{on} e t_{off}, que dependem somente do transistor. A perda também depende da freqüência de operação, que é função somente da aplicação considerada. Num BJT de potência, em geral, os tempos t_{on} e t_{off} ficam numa faixa de 0,5 μs a 2 μs.

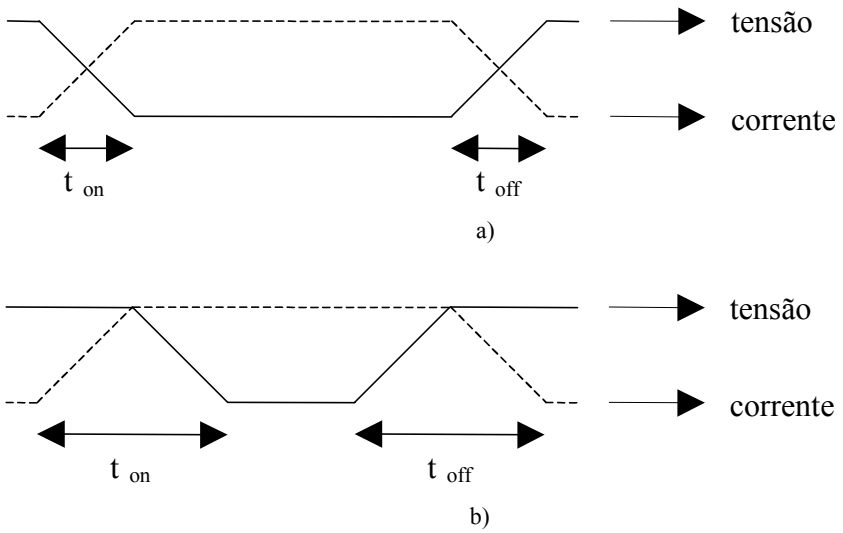

Fig. 3.2 – Formas de onda no transistor. a) melhor caso; b) pior caso.

3.2. Transistor de Efeito de Campo tipo Metal - Óxido - Semicondutor (MOSFET)

O MOSFET, ou abreviadamente FET, é um dispositivo mais complexo que o BJT e, portanto, mais caro. Apesar disso, suas vantagens são muitas. A primeira delas é o fato de ele ser acionado por tensão, facilitando muito o seu acionamento. Basta que a tensão na sua entrada (*Gate*) seja maior que a tensão de limiar (V_{th}) do dispositivo, para que seja permitida a passagem de corrente entre seus terminais (*Drain* e *Source*).

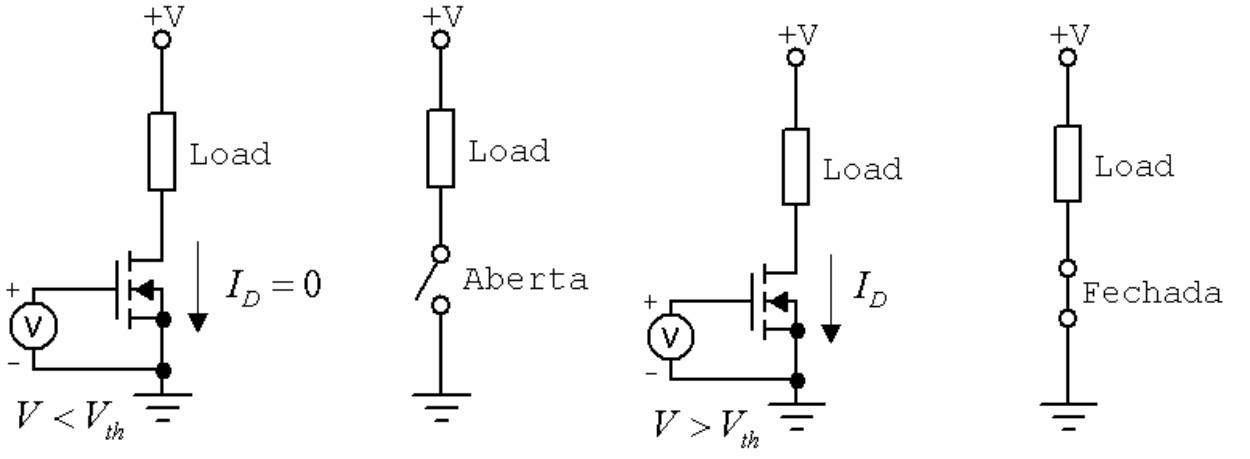

Fig. 3.3 – Modos de operação para um MOSFET.

Quando o FET está conduzindo, ele se comporta como um resistor (r_{on}). Os melhores FETs disponíveis possuem r_{on} perto de 5 mΩ. Assim, como no caso do uso de BJT, um único FET não consegue suportar sozinho todos os 160 A contínuos do exemplo considerado, portanto serão utilizados nesse caso quatro dispositivos, resultando numa potência P igual a:

$$P = I^2 \cdot r_{on} = \left(\frac{160A}{4}\right)^2 \cdot 5m\Omega = 8W \qquad (2)$$

Como se pode ver, a potência dissipada no FET é menos da metade da potência de um BJT similar e, com isso, dentro de uma faixa aceitável para o uso de pequenos dissipadores, juntamente

com ventilação ativa proporcionada por uma ventoinha (*fan*). Caso não se utilize dissipadores, a corrente máxima contínua aceitável num sistema de 4 FETs fica em aproximadamente 100 A. Outra grande vantagem do FET é que ele não possui limitação de corrente, desde que a temperatura do silício fique dentro da máxima permitida, ou seja, que a corrente não seja grande demais por muito tempo (tolerando assim grandes picos de corrente, uma segurança a mais para o circuito).

Como o FET é acionado por tensão, a corrente consumida no *gate*, durante a condução, é zero. Apesar disso, na comutação, existe uma corrente entrando ou saindo do *gate*, devido ao fato de que existe uma capacitância parasítica entre o *gate* e a fonte. Como não existem elementos dissipativos (resistores) dentro do FET no circuito de acionamento, a energia que é armazenada nessa capacitância não será dissipada no FET, e sim no circuito de acionamento.

Apesar disso, essa capacitância parasítica deve ser carregada para que a tensão no *gate*, que é a própria tensão no capacitor, chegue acima de V_{th}. Esse tempo que a tensão leva para chegar ao V_{th} depende somente do circuito de acionamento e da capacitância parasítica, mas é normalmente da ordem de algumas dezenas de nano segundos e é exatamente o t_{on} da comutação. Assim como a tensão tem que subir acima de V_{th} para "ligar" o transistor, ela tem que baixar desse valor para desligá-lo, sendo esse tempo que o transistor leva para entrar em corte, o t_{off}, também da ordem de dezenas de nano segundos. Devido aos pequenos tempos de comutação t_{on} e t_{off}, em relação ao BJT, pode-se afirmar que as perdas de comutação nos FETs são muito menores.

Em suma, as vantagens vistas acima justificam o uso de FETs no acionamento do circuito proposto, apesar de seu custo relativamente alto.

3.3. Circuito de Acionamento da Ponte H

O fato de FETs serem acionados por tensão facilita muito o desenvolvimento de um circuito para seu acionamento, mas uma característica não ideal existente gera alguns problemas. Para o FET entrar em condução, uma carga elétrica deve ser injetada no *gate* do FET para que a tensão entre o *gate* e o *source* chegue a um valor de aproximadamente 10 V, que é, em geral, a tensão em que eles entram em condução máxima, ou seja, menores valores de r_{on}. Essa necessidade de injetar cargas no FET pode ser modelada como um capacitor em paralelo com o *gate* do FET (a capacitância parasítica mencionada anteriormente), vide a Figura 3.4.

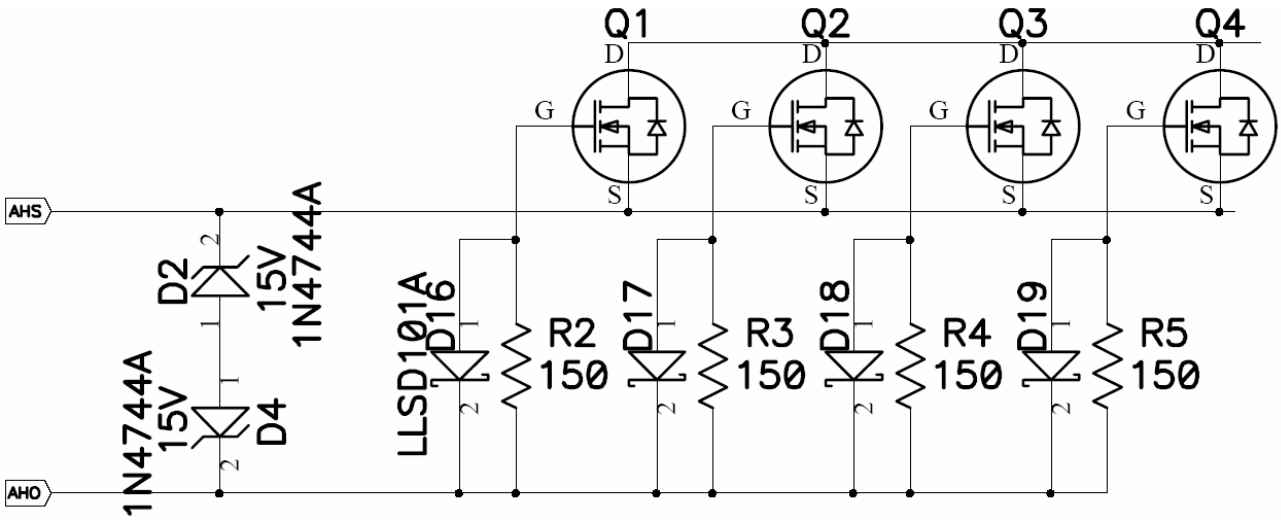

Fig. 3.4 – Circuito de acionamento dos FETs.

Para carregar essa grande capacitância, foi utilizado o chip HIP4081A [2], capaz de fornecer até 2 A para os quatro FETs que se encontram em cada saída, em paralelo. Mas como os FETs se comportam como uma capacitância, a corrente inicial em cada ciclo seria muito grande, caso não houvesse alguma limitação, podendo danificar o chip. Para evitar isso, foi colocado um resistor em série com o *gate* de cada FET, o que limita a corrente total, apesar de fazer com que os FETs demorem mais para conduzirem. Um outro beneficio do resistor é o fato de ajudar a balancear

o tempo de T_{on} e T_{off} de todos os FETs em paralelo, ao fazer com que as constantes RC do conjunto resistor - capacitância sejam semelhantes.

Apesar dessas vantagens, a inclusão deste resistor aumenta a chance de ocorrer um *shoot-through*, pelo fato de que o T_{off} é maior do que o T_{on}, ou seja, um FET pode entrar em condução enquanto o outro no mesmo lado da ponte H ainda estiver conduzindo. Duas proteções existem para evitar essa condição. A primeira é um tempo programável no HIP4081A que faz com que ambos os FETs fiquem em estado de corte e a segunda é a adição de diodos extremamente rápidos em paralelo com os resistores, de forma que durante o T_{off} toda a corrente seja descarregada por eles, fazendo com que o T_{off} seja muito mais rápido, eliminando qualquer chance de ocorrer *shoot-through* no circuito.

Uma última preocupação é o fato de o *gate* dos FETs serem muito sensíveis a tensões muito altas, que podem destruir o FET, mesmo caso ocorram muito rapidamente. Para proteger os FETs, dois diodos zener são colocados para cortar qualquer ruído que ultrapasse o valor da tensão do zener (15 V, vide Figura 3.4).

3.4. Circuito de Completo de Potência

O componente principal do circuito de potência projetado é o chip HIP4081A, discutido na seção anterior. Ele tem a função de acionar os FETs, tanto os FETs inferiores quanto os superiores, já incluindo um circuito que aumenta a tensão para acionar estes últimos. Ele aceita tensões de bateria de 12 V até 80 V, gerando todos os sinais e tensões necessários para acionar os FETs. Para funcionar corretamente, o HIP4081A precisa de uma tensão de alimentação nominal de 12 V, podendo variar de 9,5 V até 15 V. Caso a tensão de alimentação seja menor que o valor mínimo, uma proteção interna desliga as saídas para os FETs superiores, desligando efetivamente o circuito. Deve ser lembrado que caso a tensão de alimentação exceda 16 V, o chip pode ser danificado. Na figura 3.5, o chip está sendo alimentado por uma tensão de 12 V e uma tensão de bateria de 80 V está sendo aplicada à carga.

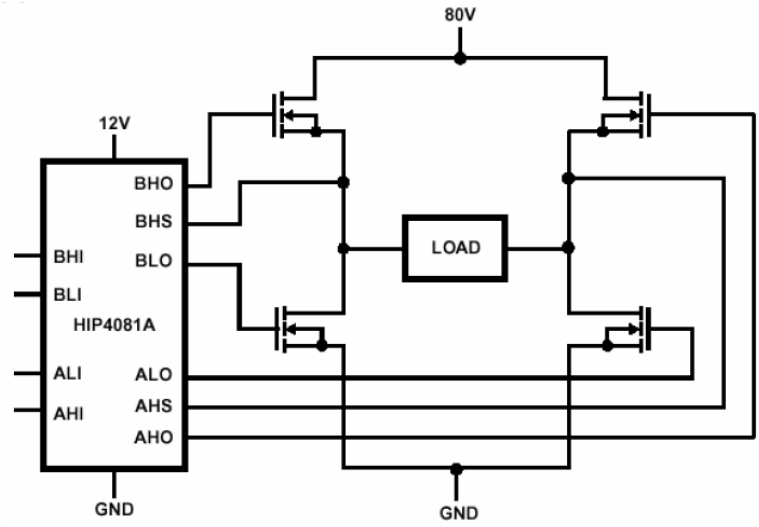

Fig. 3.5 – Diagrama de blocos do HIP4081A

O HIP4081A tem quatro entradas digitais, AHI, ALI, BHI e BLI, cada uma correspondendo às saídas que acionam cada conjunto de FETs, AHO, ALO, BHO e BLO, respectivamente, ou seja, quando uma entrada for ativada, a sua saída correspondente faz com que os FETs entrem em condução. Uma eletrônica externa deve enviar os sinais de PWM e direção para o HIP4081A de

forma a acionar a ponte H. Esses sinais digitais são compatíveis com a lógica TTL, mas qualquer tensão na entrada acima de 3 V é reconhecida como sinal de nível alto. Além disso, o HIP4081A possui uma proteção na lógica interna contra *shoot-through,* que desliga os FETs superiores quando os FETs inferiores do mesmo lado da ponte forem acionados, independentemente do estado da entrada superior. Essa proteção é feita com portas lógicas AND em todas as entradas do HIP4081A, conforme mostrado na figura 3.6. As portas têm como entrada os valores de entrada de AHI, ALI, BHI e BLI, o complemento do pino DIS (Disable, ou desabilitar), fazendo com que ao ter um nível baixo, desliga todos os FETs. Além dessas, as portas superiores, de AHI e BHI, tem outras duas entradas, uma para a proteção contra tensões de alimentações baixas, já citada e outra sendo o complemento das entradas inferiores, o que faz com que as saídas superiores, AHO e BHO, sejam desligadas, caso as entradas inferiores respectivas estiverem ativadas.

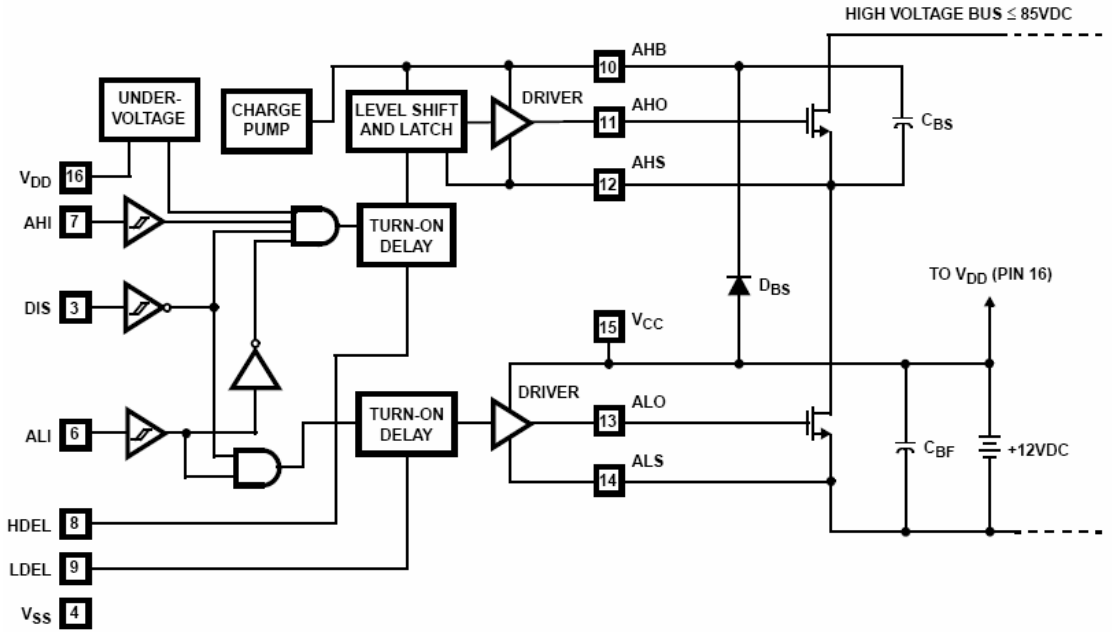

Fig. 3.6 – Diagrama funcional de metade do HIP4081A

Resistores também são colocados no HIP4081A, entre os pinos de entrada e o terra, forçando o chip a um estado padrão, de forma que, caso não exista nada conectado à placa de potência, todos os FETs estejam desligados, formando uma proteção adicional contra o acionamento não desejado dos motores.

Devido à natureza dos FETs utilizados, a tensão no *gate* deve ser aproximadamente 10 V maior do que a tensão de bateria para acionar os FETs superiores. Para gerar essa tensão maior, o HIP4081A possui um sistema de charge-pump que, com a ajuda de um diodo e um capacitor externo, D_{BS} e C_{BS} na figura 3.6, gera a tensão necessária nas saídas AHO e BHO, possibilitando o acionamento dos respectivos FETs. Devido à existência desse sistema, não é necessário mais nada para acionar os FETs.

Para proteger o circuito de picos de tensões causados pelas escovas e comutadores de motores DC, é utilizado um componente chamado Supressor de Transientes de Tensão (TVS – *Transient Voltage Supressor*, em inglês). Eles funcionam exatamente como um diodo zener, ou seja, quando a tensão entre o dispositivo ultrapassa um valor especificado ele entra em condução, "absorvendo" toda a tensão excedente. Além disso, o TVS é otimizado para agüentar picos de tensão com altas correntes. O TVS é colocado no circuito de forma a absorver os picos de tensão entre os terminais da bateria e proteger os FETs de tensões que possam vir a superar o limite da tensão entre os terminais *drain* e *source*. Além dos TVS, uma rede RC entre os terminais do motor provê uma proteção adicional contra picos de alta freqüência gerados pelas escovas do motor e, finalmente, grandes capacitores eletrolíticos são colocados o mais próximo possível da ponte H para reduzir os efeitos causados pelas impedâncias dos fios da bateria até o circuito.

A última parte do circuito é composta pela fonte de alimentação chaveada, que converte a tensão da bateria para 12 V, através de um regulador chaveado de alta eficiência, fazendo com que não seja necessário o uso de um dissipador para o regulador. Esses 12 V também alimentam o circuito externo à placa de potência, o que elimina a necessidade de uma alimentação externa para o circuito de sinais.

O circuito completo se encontra no anexo A.

4 – Circuito de controle

O circuito de potência visto no Capítulo 3 não é capaz de receber diretamente os sinais de um receptor de rádio-controle sem que estes sejam primeiramente tratados. Esse tratamento de sinais é feito pelo circuito de controle, formado principalmente por um micro-controlador PIC [3] capaz de executar 1 milhão de instruções por segundo, caso esteja funcionando com um cristal de 4 MHz.

O circuito da placa de sinais proposta nesse trabalho será capaz de decodificar até quatro sinais do receptor do controle remoto e, de acordo com estes, acionar duas eletrônicas de potência – um sinal de entrada para cada – com controle de velocidade e direção, além de poder acionar um relé auxiliar utilizando uma lógica com um ou os dois sinais restantes.

4.1. Sinais de Entrada e Saída

Antes de analisar o projeto do circuito de controle, deve-se entender quais sinais devem entrar e sair deste circuito.

O único sinal de entrada do circuito é o que se origina no receptor do rádio-controle. Esse sinal digital possui um período que pode variar entre 18 ms e 25 ms, com o sinal variando entre 1 ms (*low*) e 2 ms (*high*).

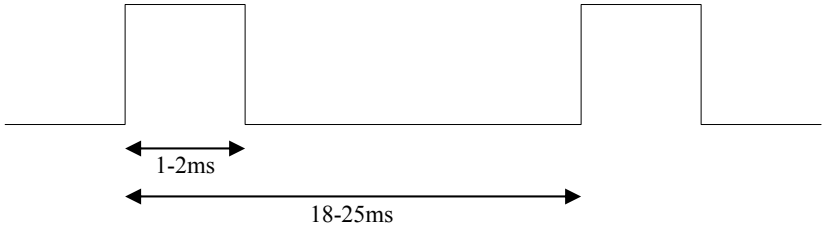

Fig. 4.1 – Sinal do Receptor

Quanto aos sinais de saída, há cinco sinais que devem ser enviados à placa de potência, o AHI, ALI, BHI, BLI e o sinal de *Disable*. Esses sinais correspondem a cada um dos sinais de entrada do HIP4081A da placa de potência. Devido à proteção contra *shoot-through* do HIP4081A, pode-se fazer uma simplificação, mantendo tanto o sinal AHI quanto o BHI em nível lógico alto. Já o *Disable* só é usado caso se queira desligar a ponte H. Com isso, a Tabela 4.1 mostra os sinais sugeridos para o correto funcionamento do circuito de potência.

Tabela 4.1 – Sinais sugeridos para a placa de potência

AHI	BHI	ALI	BLI	Disable	Função
1	1	0	PWM	0	Para frente
1	1	PWM	0	0	Para trás
1	1	0	0	0	Freio
1	1	1	1	0	Freio
X	X	X	X	1	Desligado

X: não importa; 1: 5 V; 0: 0 V;

Apesar de já ser necessário utilizar somente dois sinais, já que o AHI e o BHI são fixos em nível lógico 1, ainda não há no circuito um sinal somente para o PWM e outro independente somente para a direção, o que seria necessário já que o sinal gerado pelo PIC estaria fixo num pino específico. A melhor solução é modificar a tabela para que os sinais fiquem fixos numa única posição. Essa modificação é mostrada na Tabela 4.2.

Tabela 4.2 – Sinais sugeridos modificados para a placa de potência

AHI	BHI	ALI	BLI	Disable	Função
1	1	0	PWM	0	Para frente
1	1	1	$\overline{\text{PWM}}$	0	Para trás
1	1	0	0	0	Freio
1	1	1	1	0	Freio
X	X	X	X	1	Desligado

X: não importa; 1: 5 V; 0: 0 V;

Com essa nova tabela, fica claro que o sinal BLI será o sinal de PWM, apesar de ser necessário alguma medida (seja por *software* ou por *hardware*) para inverter o sinal caso o motor esteja girando para trás. Nesse caso o sinal ALI será o sinal de direção, sendo de nível baixo (“0”) quando se deseja ir para frente e de nível alto (“1”) quando se deseja ir para trás. Com isso, considerando o PWM em 100%, ou seja, em “1” sempre, no caso para frente a corrente passa pelos FETs conectados em AHO e BLO e, como no caso para trás o PWM está invertido (em “0”) no caso para trás, a corrente passa pelos FETs em BHO e ALO.

4.2. Descrição do Hardware

O *hardware* do circuito de controle é extremamente simples. É composto de um micro-controlador PIC16F876A, de um *buffer* para isolar os sinais gerados pelo PIC em relação aos sinais da placa de potência, protegendo-o de quaisquer problemas que possam ocorrer. Há também um circuito (opcional) de acionamento de relé de alta potência, também isolado, mas dessa vez por um optoacoplador, que provê um isolamento completo entre o relé e o circuito de controle em si. Além disso, ainda existe um circuito regulador de tensão que utiliza os 12 V disponíveis através da placa de potência e os transforma em 5 V através de um regulador linear. Esses 5 V são utilizados para alimentar todo o circuito de controle, incluindo o receptor de rádio controle. O circuito ainda inclui dois botões, um para *reset* do PIC e outro para o acionamento do modo de calibragem.

O micro-controlador PIC possui opção de gravação *in-circuit*, ou seja, não é necessário retirar o chip para gravar o software de controle. Para isso, basta conectar o gravador no conector disponível no circuito.

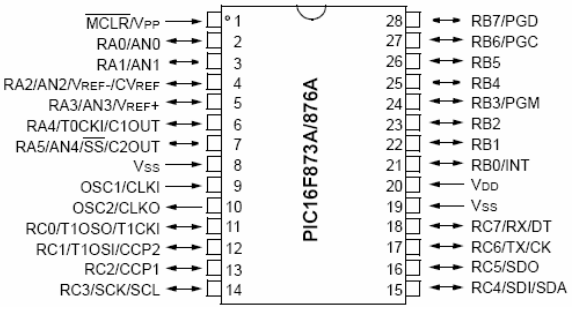

Fig. 4.2 – Pinagem do PIC16F876A.

As entradas dos sinais do receptor do rádio-controle são conectadas através de resistores para os pinos RB4 até o RB7 do PIC. Esses pinos geram uma interrupção (desvio de programa) quando a entrada muda de estado, sendo assim perfeitos para a leitura do sinal do receptor.

O buffer utilizado em ambas as saídas é o chip 74HCT244, composto por dois conjuntos de quatro buffers. É possível também utilizar outros chips equivalentes ao 74HCT244, como o

74HC244, mas deve-se atentar aos valores de tensão de saída que estes apresentam, caso se utilize algum outro além desses dois apresentados, já que existem valores mínimos a serem aplicados às entradas do HIP4081A da placa de potência. Por exemplo, o chip da série LS (74LS244), possui uma saída em nível lógico alto com a tensão mínima especificada de 2 V, enquanto a tensão mínima de entrada do HIP4081A para reconhecimento de nível lógico alto é de 2,5 V. Apesar disso o 74LS244 pode ser utilizado, mas não é recomendado, já que a tensão mínima de saída só é alcançada com altas correntes de saída, que não ocorrem no circuito.

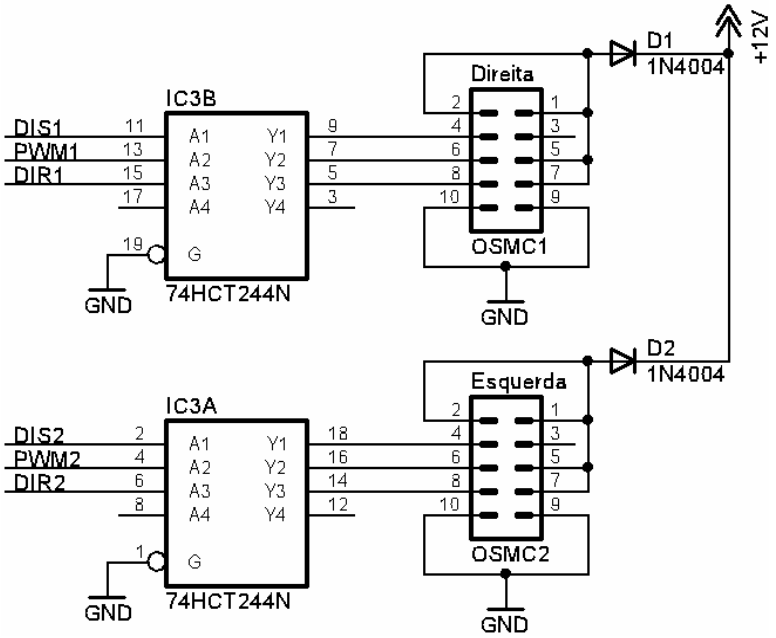

Fig. 4.3 – Circuito de saída da placa de controle.

Na figura 4.3 estão mostrados dois conectores chamados OSMC1 e OSMC2. Eles são os conectores para as placas de potência, chamadas assim devido ao circuito base utilizado no projeto chamado *Open Source Motor Controllers* ou, abreviadamente, OSMC [4].

Os sinais de saída AHI e BHI dos conectores das placas de potência, pinos 5 e 7 respectivamente, já estão conectados na linha de 12 V da placa de potência, pinos 1 e 2, ou seja,

serão reconhecidos pelo HIP4081A como sendo nível lógico 1, exatamente como descrito na Tabela 4.2.

Outro detalhe que pode ser visto na Figura 4.2 é o fato de termos diodos entre a linha de 12 V que vem dos conectores e a linha de 12 V da placa de controle. Isso é feito para se utilizar a alimentação das duas eletrônicas de potência e, com isso, ganhar confiabilidade, já que existirá alimentação no circuito de controle mesmo caso uma das eletrônicas de potência queime.

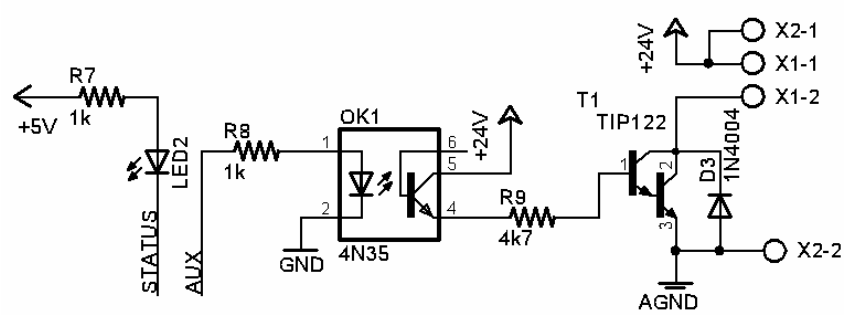

Fig. 4.4 – Circuito do relé e LED de status.

Já na Figura 4.3, pode-se ver o circuito que aciona o relé auxiliar (opcional). Ele é formado por um optoacoplador que, ao ser ativado, faz com que o transistor T1 entre em condução, acionando o relé. Para isso, os terminais do relé devem ser posicionados nos terminais X1-1 e X1-2 e uma alimentação de 24 V deve ser colocada nos terminais X2-1 e X2-2. Isso, é claro, assume que o relé seja de 24 V. O transistor T1 pode conduzir até 1 A, caso não se utilize um dissipador de calor, ou até 3,5 A caso contrário. Os transistores TIP120, TIP121 e TIP122 podem ser utilizados nesse circuito [5].

Na Figura 4.3 também é mostrado o LED de status, que serve para mostrar o estado do programa que está sendo executado pelo PIC. Nesse projeto ele só foi utilizado para indicar se o circuito está no modo de calibragem ou no modo normal.

4.3. Software do Circuito de Controle

Todo o software foi escrito na linguagem de programação C, que facilita muito a programação, pois o programa utiliza várias equações matemáticas que seriam complicadas de serem implementadas em linguagem de máquina (*assembly*). O programa segue o diagrama de blocos mostrado na Figura 4.4.

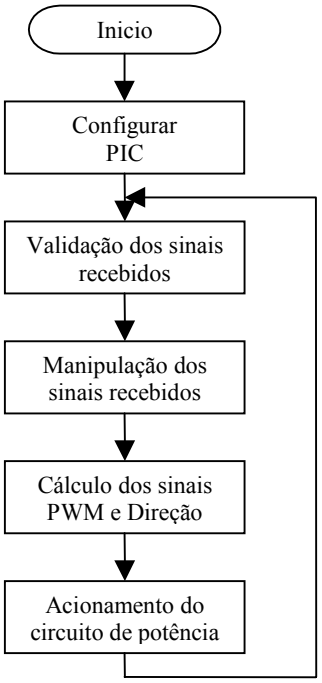

Fig. 4.5 – Diagrama de blocos do software

Como pode ser observado na figura acima, o programa principal não aquisita os valores dos sinais do receptor. Isso é feito pela rotina de interrupção, que é executada sempre que uma interrupção ocorre, independentemente do que está sendo processado no momento da interrupção. No caso, ocorre uma interrupção em todas as transições dos quatro sinais do receptor. Isso permite que os tempos lidos sejam os mais precisos possíveis.

Um outro detalhe não mostrado na Figura 4.4 é o fato de o programa rodar a cada 10 ms, para facilitar as temporizações existentes no software.

4.3.1. Configuração do PIC

Antes de poder executar qualquer programa, deve-se rodar alguma rotina que configure todos os periféricos do micro-controlador que serão utilizados no sistema. Também é necessário configurar as portas (pinos) do PIC para entrada ou saída, além de colocar os seus respectivos valores iniciais, e de inicializar as variáveis. Todas essas operações são feitas escrevendo valores específicos em registradores de configuração do PIC. A Tabela 4.3 mostra todos os registradores que devem ser configurados, suas funções no sistema, além da configuração necessária e o valor, em hexadecimal, a ser escrito. A ordem em que esses valores são escritos é importante, então devem ser colocados na ordem mostrada na tabela.

Tabela 4.3 – Configuração do micro-controlador

Registrador	Função	Configuração	Valor
TRISA	Define se os pinos da PORTA serão entradas ou saídas	Todos os pinos são saída	0x00
TRISB	Define se os pinos da PORTB serão entradas ou saídas	Pinos RB0 e RB4 a 7 são entrada	0xF1
TRISC	Define se os pinos da PORTC serão entradas ou saídas	Somente o pino RC7 é entrada	0x80
PORTA	Define a saída nos pinos da PORTA	–	0x00
PORTB	Define a saída nos pinos da PORTB	–	0x00
PORTC	Define a saída nos pinos da PORTC	–	0x00
ADCON1	Configura parte do conversor AD e pinos da PORTA	Configura os pinos da PORTA como digital (padrão é analógico)	0x07
OPTION	Configuração do Timer 0	Clock interno, dividido por 4. A configuração é feita pelo macro	0xC1
T1CON	Configuração do Timer 1	Clock interno, dividido por 8 e já habilitado. A configuração é feita por macro `Setup_Counters();`	0x31
PR2	Configura freqüência do PWM	F_{pwm} = 3,906 kHz ≈ 4 kHz	0xFF
CCPR1L	Configura o duty cycle do PWM1	Deve ser inicialmente zero, para não acionar inadvertidamente os motores. Configurado pelo macro `set_ccp1();`	0x00
CCPR2L	Configura o duty cycle do PWM2	Deve ser inicialmente zero, para não acionar inadvertidamente os motores. Configurado por macro `set_ccp2();`	0x00
CCP1CON	Configura o modo do modulo CCP1	Deve ser configurado como PWM Configurado pelo macro `setup_ccp1();`	0x0C
CCP2CON	Configura o modo do modulo CCP2	Deve ser configurado como PWM Configurado pelo macro `setup_ccp2();`	0x0C

T2CON	Configuração do Timer 2	Prescaler e postscaler do timer 2 desabilitados e timer 2 ligado Configurado pelo macro `setup_timer_2();`	0x04
RBPU	Bit de ativação dos resistores de pull-up da PORTB	Resistores devem ser ativados, para que o PIC leia nível lógico alto, caso algum canal do receptor não esteja ligado.	0
INTCON	Configuração das interrupções	Interrupção on-change (mudança de estado) da PORTB e de overflow do timer 0 devem ser ativadas. Configurada pelo macro `enable_interrupts();`	0x28
GIE	Habilita todas as interrupções	–	1

Além desses registradores, as variáveis também devem ser inicializadas. A rotina de inicialização completa é mostrada abaixo. Todas as macros são definidas no arquivo "commdefs.h".

```
/*****************************************************************************************/
/* Rotina de Inicialização do processador                                                */
/*****************************************************************************************/
void init_proc (void)
{
       TRISA = 0x00;                        // PORTA é toda saída
       TRISB = 0xF1;                        // RB<4:7> são entradas (4 canais) e
                                            // RB0 é entrada do botão de calibragem
       TRISC = 0x80;                        // PORTC é toda saída, exceto RC7 (RX)

       PORTA = 0x00;
       PORTB = 0x00;
       PORTC = 0x00;

       Status_Led_Off();

       sts.tmr_1ms =
       sts.tmr_10ms =
       sts.invalid_rc1 =
       sts.invalid_rc1 =
       sts.invalid_rc1 =
       sts.invalid_rc1 = true;
       sts.Disable_All = true;
       sts.MixMode = EEPROM_READ(eeaddMixMode);
       sts.AuxMode = EEPROM_READ(eeaddAuxMode);

       Slew = SR_100ms;

       ticks = 0;

       ADCON1 = 0x07;                       // Configura toda a PORTA como digital

       Setup_Counters( RTCC_INTERNAL, RTCC_DIV_4 );// Configura TMR0 (RTCC) para clock
                                            // interno com pre-scaler em 1:4 para
                                            // ter um Time-out de 1,024 ms ~ 1 ms

       setup_timer_1( T1_DIV_BY_8 | T1_ENABLED );  // Timer 1 incrementa a cada 8 us
                                            // e já está rodando

       PR2 = 0xFF;                          // Fpwm = 3,906 kHz
       set_ccp1(0);                         // Duty Cycle = 0 (sem potência nos
       set_ccp2(0);                         // Motores)
       setup_ccp1( CCP_PWM );               // CCP1 em modo PWM
```

```
        setup_ccp2( CCP_PWM );               // CCP2 em modo PWM
        setup_timer_2( T2_DIV_BY_1, 1);              // Seta o Prescaler e o Postscaler do
                                             // Timer 2 em 1:1 e liga o Timer 2

        RBPU = 0;             // Ativa os Pull-ups na PORTB para o caso de não haver sinal

        enable_interrupts( RB_CHANGE | INT_TIMER0 );// Habilita as interrupções:
                                             // RB on-change -> sinal rádio
                                             // TMR0 -> int a cada 1ms

        GIE = 1;                                     // Habilita todas as interrupções
}
```

4.3.2. Rotina de Interrupção

Ao ocorrer uma interrupção, desde que o bit GIE (que habilita todas as interrupções) e o bit que habilita a interrupção de um periférico específico estiverem em 1, a rotina de interrupção será executada. Como na configuração os bits relativos à interrupção da *RB on-change* e de *overflow* do timer 0 estão habilitados, assim como o GIE, sempre que ocorrer o evento relativo a essas interrupções a rotina de interrupção será executada.

Como todas as interrupções que ocorrem executam a mesma rotina, deve-se testar os flags de interrupção para descobrir quem a gerou. Caso o flag de um módulo esteja em 1, a interrupção ocorreu nele e, então, é executado algum código específico. Não se deve esquecer de zerar o flag da interrupção que ocorreu, caso contrário a rotina de interrupção será executada num loop infinito, já que o fato de o flag de alguma interrupção habilitada estar em 1 é que gera a interrupção. O fato de ter que testar os flags por software permite que se tenham prioridades entre as interrupções, bastando testar a interrupção com maior prioridade primeiro e ir testando as próximas com prioridade decrescente. Nesse sistema, a medição do tempo dos pulsos do receptor deve ser feita com a menor latência possível, portanto a interrupção do RB on-change terá a maior prioridade, sendo testada primeiro, seguido pela interrupção de temporização, gerada pelo timer 0 que, por ter freqüência baixa, pode ter prioridade menor.

A medição de tempo é feita através da interrupção *RB on-change*. Como ela é gerada sempre que há uma mudança na entrada dos pinos RB4 até o RB7, é necessário identificar qual entrada mudou e qual é seu novo valor. Isso é feito guardando o último valor da PORTB e comparando com o valor atual, através de uma operação lógica chamada "OU exclusivo" ou, abreviadamente, XOR, vide Figura 4.5. Ao fazer essa operação em dois bytes obtém-se na saída outro byte, com bits igual a 1 somente nas posições em que existam diferenças entre os dois bytes.

$$\begin{array}{ll} 01101001 & 0x69 \\ \underline{10101010} = & \underline{0xAA} \\ 11000011 & 0xC3 \end{array}$$

Fig. 4.6 – Exemplo da operação XOR em dois bytes

Após a operação de XOR, ainda é necessário saber se o bit do pino que se deseja testar é 1. Para isso, basta zerar todos os outros bits e comparar com o valor que a PORTB teria caso só aquele bit estivesse em 1.

Com isso, sabe-se se foi esse pino que mudou de estado, então basta ler o pino em si para saber se ele mudou para 1 ou para 0. Caso seja 1, o tempo lido do timer 1 (múltiplos de 8 µs), logo no início da rotina de interrupção, é o tempo inicial do pulso e, caso contrário, é o tempo final. Como é recomendado manter a rotina de interrupção com o menor tamanho possível, a subtração dos tempos lidos é feita em outra rotina, fora da interrupção. A rotina que implementa essa operação para o pino RB4 é mostrada no quadro abaixo:

```
time = TMR1L;                          // Lê o valor do timer
rb_value = (old_portb ^ PORTB);              // Compara o valor da PORTB antigo com o atual
old_portb = PORTB;                           // Salva o novo valor (proximo antigo)

if ((rb_value & 0x10) == 0x10) {             // Se RB4 mudou

       if (RB4 == 1){                 // Subida de Pulso
              servo1.start = time;    // valor inicial do pulso é time
       }
       else {                         // Descida do pulso
              servo1.end = time;             // valor final do pulso é time
              servo1.done = TRUE;            // Terminou o pulso
       }
}
```

Após ler o tempo de quando o pulso vai para zero, a rotina avisa que possui os últimos valores de início e fim do pulso do receptor através do campo *done* da variável *servo1*. Após ler os valores novos, o programa principal zera este campo. Todos os outros pinos possuem código semelhante a esse, mudando apenas os valores de comparação no primeiro condicional *if*.

Além da leitura dos pulsos do receptor, a rotina de interrupção também aciona o campo *tmr_10ms* da variável de status *sts*, que será utilizado para a temporização do programa, a cada 10 ms, durante a interrupção do timer 0. Como o timer 0 foi programado para ter overflow a cada 1,024 ms e o tempo de 10 ms não é crítico, a variável pode ser acionada a cada dez contagens da interrupção. Além disso, os 1,024 ms também são utilizados para incrementar um contador – *ticks* –, que conta quanto tempo passou desde o último reset. Essa rotina é mostrada abaixo:

```
        if (T0IF)
        {
                T0IF = 0;
                if (++count > 9) {
                        sts.tmr_10ms = 1;
                        count = 0;
                }
                ticks++;
        }
```

4.3.3. Normalização e validação do pulso do receptor

Os valores de tempo dos quatro pulsos do receptor lidos pela rotina de interrupção ficam armazenados nas variáveis *servo1*, *servo2*, *servo3* e *servo4*. Essas variáveis possuem uma estrutura com vários campos, listados na Tabela 4.4, cada um com uma função específica dentro do programa.

Tabela 4.4 – Campos de dados das variáveis servo1 a servo4

Campo	Descrição	Função
start	Tempo de inicio do pulso	Tempo de início de pulso lido na interrupção
end	Tempo de fim do pulso	Tempo de fim de pulso lido na interrupção
low	Menor valor de pulso	Fator de escala para normalizar pulso – pode ser modificado na calibragem
high	Maior valor de pulso	Fator de escala para normalizar pulso – pode ser modificado na calibragem
rcpulse	Largura do pulso	É a subtração de *end* por *start*.
width	Largura do pulso normalizado	É o valor de rcpulse, normalizado entre 0 e 1023.
drive	Valor a ser aplicado no motor	
done	Flag de fim de pulso	Indica que o pulso acabou de ser lido na interrupção.
valid	Flag de validação	Indica se o último pulso é válido ou não.

Além desses campos, valores definidos no arquivo *motor.h* são utilizados no processo de validação do pulso, sendo os principais deles os valores que delimitam os tamanhos máximo e mínimo do pulso sem que programa o considere inválido. São eles *MINVALIDWIDTH* e *MAXVALIDWIDTH*.

O primeiro passo da rotina é simples, basta conferir que o pulso terminou de ser lido, através do campo *done* e, caso tenha terminado, subtrair *end* de *start* e colocar o valor em *rcpulse*. O valor de *rcpulse* ainda é subtraído por 15 e o resultado armazenado numa variável local – *tWidth* – deslocando os valores lidos e, com isso, permitindo que valores muito perto do limite superior, que está perto do maior valor possível que pode ser armazenado em um byte, ainda possam ser lidos. Após isso, deve-se verificar que *tWidth* esteja entre os valores válidos e, caso estejam, deve-se ainda verificar se o pulso está entre os valores de *low* e *high*, já que esses são os maiores valores que o

pulso pode ter para ser normalizado corretamente. Caso seja menor ou maior que estes valores, basta igualar a *low* ou *high*, respectivamente. Essa operação é mostrada no código abaixo:

```
        if (MINVALIDWIDTH < tWidth && tWidth < MAXVALIDWIDTH)
               {

            if (tWidth > p->high)           // valores limites do pulso
                tWidth = p->high;
            else if (tWidth < p->low)
                tWidth = p->low;
```

Após ter os valores corretos do pulso, deve ser feita uma normalização para transformar o valor de *tWidth* para algum valor entre 0 e 1023.

$$tWidth = \frac{(tWidth - low) \cdot 1023}{high - low} \tag{3}$$

Como a normalização envolve multiplicações e divisões, o código gerado é muito grande e lento, então é necessário otimizá-lo. Quanto à divisão, não existe nada que possa ser feito para diminuir o código, já que os valores de *high* e *low* são valores que podem ser modificados durante a calibragem. Porém a multiplicação pode ser otimizada, já que o multiplicador é constante, além de ser um número muito próximo de 1024, que é múltiplo de 2. Pode-se demonstrar que a operação lógica de deslocar um número binário para a esquerda é o equivalente a multiplicar por 2 e, portanto, deslocar o número 10 vezes para a esquerda é o mesmo que multiplicar por 1024. Essa operação é extremamente rápida, já que o PIC possui uma instrução que a executa.

$$x = 00001111 = 15$$
$$x << 1 = 00011110 = 30$$

Fig. 4.7 – Operação de deslocamento à esquerda

Portanto, para multiplicar por 1023, basta deslocar o número para a esquerda e, depois, subtrair o multiplicando do resultado obtido. Deve-se lembrar que a variável *tWidth* deve ser de 16 bits para suportar o valor de 1023, já que com 8 bits o valor máximo é 255. Além disso, a função deve utilizar uma variável temporária de 32 bits, já que o maior valor que pode surgir durante a

multiplicação é $248 \cdot 1024$ ou 253952, que é maior do que o valor máximo de um número de 16 bits. Esse fato faria com que a multiplicação, caso não fosse otimizada, fosse lenta demais, inviabilizando o funcionamento do circuito.

Apesar de essa otimização já ter diminuído muito o tempo de processamento gasto na multiplicação, outra técnica simples ainda pode ser empregada. Como o PIC é um processador de 8 bits, um número de 16 bits é formado por 2 bytes. Deslocar o número 8 vezes para a esquerda é o mesmo que colocar o byte menos significativo no mais significativo, o que é mais rápido do que deslocar o número para a esquerda várias vezes. Então a função final deve deslocar o número 8 vezes para a esquerda, depois mais dois deslocamentos e, após isso, subtrair o multiplicando do resultado. A operação completa é mostrada no quadro abaixo:

```
temp = tWidth - p->low;
temp = ((temp << 8) << 2) - temp;
p->width = temp / (unsigned)(p->high - p->low);
```

Essa rotina ainda deve retornar se o pulso foi considerado válido ou não.

4.3.4. Inversão do PWM

Como visto na Seção 4.1, é necessário inverter o sinal do PWM quando o motor estiver girando para trás. Como não se deseja fazer uma solução por hardware, é necessário encontrar um método simples para fazer a inversão. A forma mais simples de cumprir esta tarefa é utilizar números inteiros com sinais, ou seja, podem ser positivos ou negativos. Na prática, essa inversão do PWM faz com que a corrente que vá da bateria para o motor (curvas A ou B da figura 2.5) durante o T_{off}, em vez de acontecer durante o T_{on}, como é o caso do motor estar indo para frente. Os números negativos são tratados com uma operação chamada complemento a 2, vide Figura 4.7.

$$
\begin{aligned}
& & 0101 &= 5 \\
-5 \Rightarrow \quad & & \overline{0101} &= 1010 \\
& & 1010 + 1 &= 1011 = -5
\end{aligned}
$$

Fig. 4.8 – Operação de complemento a 2

É essa operação que permite com que o sinal de PWM seja invertido quando o motor gira para trás. Ela funciona fazendo o complemento do número positivo e somando 1 ao resultado. Utilizando esse método, o bit mais significativo do número se torna o sinal do número e os outros bits o número em si. A Tabela 4.5 mostra todos os valores possíveis de números com sinal para uma palavra de 4 bits.

Tabela 4.5 – Números positivos e negativos de 4 bits

Número	Binário	Número	Binário
0	0000_2	-1	1111_2
1	0001_2	-2	1110_2
2	0010_2	-3	1101_2
3	0011_2	-4	1100_2
4	0100_2	-5	1011_2
5	0101_2	-6	1010_2
6	0110_2	-7	1001_2
7	0111_2	-8	1000_2

Usando o exemplo dado na Tabela 4.5, se o número for 7, desconsiderando o bit de sinal, temos o número 111_2, que corresponderia com a saída de PWM sempre em nível alto. Isso combinado com a saída de direção em 0 (para frente), teríamos a maior velocidade para frente, com a corrente passando pelo motor através de AHI e BLI.

Caso o número fosse −1, o valor do PWM ainda seria 111_2 e a sua saída ainda estaria sempre em nível alto, ou seja, BLI sempre fechado (ligado). Combinando isso com a saída de direção em 1, o motor fica parado, pois todos os quatro AHI, BHI, ALI e BLI estariam fechados (ligados), impedindo o fluxo da corrente da bateria para o motor devido à proteção de *shoot-through* do HIP4081A. Se o número fosse −8, o PWM teria 000_2 o que, juntamente com a direção em 1, fará

com que o motor gire em toda sua velocidade para trás, pois BLI estaria aberto (desligado), com a corrente passando pelo motor através de BHI e ALI. Nos casos intermediários, o PWM "negativo" em BLI funcionaria impedindo o fornecimento de corrente ao motor durante o seu tempo T_{on} em que estiver ligado, e fornecendo corrente através de BHI e ALI no tempo restante em que estiver desligado.

Portanto, só o fato de se trabalhar também com números negativos é o suficiente para resolver o problema, desde que se tenha um bit a mais do que originalmente necessário para guardar o sinal do número. Como o PWM é de 10 bits, são necessários 11 bits para a velocidade, ou seja, os valores utilizados ficam entre −1023 e 1024, sendo que os positivos fazem o motor girar para a frente e os negativos para trás.

4.3.5. Inicialização da Placa de Potência

Antes de poder utilizar a placa de potência, é necessário inicializar o chip HIP4081. O procedimento é bem simples, necessitando apenas manter as entradas ALI e BLI em nível baixo por algum tempo e depois aplicar um pulso de aproximadamente 200 μs no pino de *Disable*. Como a rotina de inicialização do PIC já mantém todas as saídas em zero, basta esperar aproximadamente 500 ms e aplicar o pulso. A rotina utiliza a variável *ticks*, que marca quanto tempo passou (em múltiplos de 1 ms) desde o último reset, ou seja, ela espera o valor de *ticks* aumentar de 500 para prosseguir com o programa.

Além de inicializar o HIP4081, ainda é necessário ler da memória de dados não volátil (EEPROM) os valores de *high* e *low* de cada canal do receptor.

O código completo é mostrado abaixo:

```
void init_motor(void)
{
	int time = ticks + 500;

	while (ticks < time);		//espera 1/2 segundo antes de inicializar tudo!

	LEFT_ENABLE();
	RIGHT_ENABLE();
	DelayUs(200);
	LEFT_DISABLE();
	RIGHT_DISABLE();

	servo1.low = EEPROM_READ(eeaddRC1low);
	servo1.high = EEPROM_READ(eeaddRC1high);
	servo2.low = EEPROM_READ(eeaddRC2low);
	servo2.high = EEPROM_READ(eeaddRC2high);
	servo3.low = EEPROM_READ(eeaddRC3low);
	servo3.high = EEPROM_READ(eeaddRC3high);
	servo4.low = EEPROM_READ(eeaddRC4low);
	servo4.high = EEPROM_READ(eeaddRC4high);

	DeadBand = EEPROM_READ(eeaddDeadBand);
}
```

4.3.6. Limitação da taxa de variação da saída

Motores elétricos não toleram variações bruscas de velocidade, ou seja, grandes taxas de variação da velocidade. Quando uma alta taxa ocorre, grandes correntes nos motores aparecem, podendo danificar a eletrônica de potência. Para evitar esse efeito, o programa inclui a opção de limitar a taxa de variação, ou *Slew Rate* (SR), da saída. Um efeito negativo é que o torque no motor é menor, somente enquanto durar a variação da saída, devido à menor corrente no motor.

Essa limitação é feita somando ou subtraindo uma constante ao valor atual da saída a cada 10 ms, até o valor atual ser maior ou igual do que o desejado, ou seja, quanto maior a constante, mais rápido a saída muda, proporcionando uma maior taxa de variação e menor tempo para a mudança completa da saída. Quando essa condição ocorrer e o valor atual for maior do que o necessário, basta igualá-lo ao valor desejado.

A constante pode ser facilmente calculada através da seguinte fórmula:

$$SlewRate = \frac{1023 \cdot 10}{t(ms)} \tag{4}$$

onde *t* é o tempo que a saída levará para sair de 0 para 1023 e *SlewRate* é valor inteiro mais próximo do calculado. Como o programa é executado em ciclos de 10 ms, esse é o menor tempo que a saída pode demorar pra ir de 0 até seu valor máximo. No arquivo *motor.h* já está definido valores para tempos do SR de 10 ms, 50 ms, 100 ms, 150 ms, 200 ms, 250 ms, 300 ms, 400 ms, 500 ms, 680 ms e 1000 ms. Outros valores podem ser facilmente adicionados.

O código da rotina de Slew Rate é mostrado abaixo:

```
int DoSlew(int RequestedDrive, pRCcontrolBlock p, char SlewRate)
{

    int CurrentDrive = p->Drive;

    if (RequestedDrive > CurrentDrive)
    {
        CurrentDrive += SlewRate;
        if (CurrentDrive >= RequestedDrive)
        {
            CurrentDrive = RequestedDrive;
        }
    }
    else
    {
        CurrentDrive -= SlewRate;
        if (CurrentDrive <= RequestedDrive)
        {
            CurrentDrive = RequestedDrive;
        }
    }
    p->Drive = CurrentDrive;
    return CurrentDrive;
}
```

4.3.7. Área morta (Dead Band)

Quando o controle remoto está na sua posição central, ou seja, saída com valor zero, um valor constante é enviado para o circuito de controle. Apesar disso, o valor lido pelo PIC nesta posição não é constante devido principalmente ao fato de existirem latências variáveis para o início da execução da rotina de interrupção. Isso faria com que o motor se movesse (lentamente) mesmo quando o sinal enviado fosse zero. Para evitar isso e permitir que o motor fique parado, a saída é mantida em zero enquanto o valor a ser colocado for menor do que a variável *DeadBand*. O código da rotina é mostrado abaixo:

```
int DoDeadBand(int drive)
{
    if (((0 - DeadBand) < drive) && (drive < DeadBand))
    {
        drive = 0;
    }
    return (drive);
}
```

4.3.8. Modos de controle

Dependendo de quais canais do receptor são utilizados, é possível utilizar dois modos completamente diferentes para o controle das duas placas de potência.

O primeiro é o modo normal, em que cada canal do rádio controle controla uma placa de potência individualmente. Para esse modo, a única operação que se deve fazer é subtrair o pulso normalizado por 512, e depois multiplicá-lo por 2. Após esses passos, o valor final é comparado com 1022 para que, caso seja igual, seja então igualado a 1023, já que este último número jamais seria atingido numa multiplicação por 2 pelo fato de ser ímpar. Com esse procedimento, valores de pulso menores do que 512 serão considerados valores negativos, fazendo com que o motor gire para trás, enquanto valores maiores do que esse fazem o motor girar para a frente. Ao multiplicar por 2

adicionamos um bit ao número, o que é necessário para o correto funcionamento do PWM, como visto na Seção 4.3.4.

O código que executa o controle em modo normal é mostrado abaixo:

```
Right_Drive = LimitDrive(servo1.width - 0x200);
Left_Drive = LimitDrive(servo2.width - 0x200);

Right_Drive = Right_Drive * 2;
Left_Drive = Left_Drive * 2;

if (Right_Drive == 0x03FE)
       Right_Drive = 0x03FF;
if (Left_Drive == 0x03FE)
       Left_Drive = 0x03FF;
break;
```

A função *LimitDrive()* no código acima limita os valores de *Right_Drive* e *Left_Drive* entre −512 e 512.

O segundo tipo de controle é o chamado modo mixado. Ele é muito útil, e.g., quando os dois motores controlados estiverem acionando independentemente duas rodas de um veículo de movimentação diferencial. No modo mixado, os dois canais são misturados de forma a um deles controlar a velocidade média dos motores e o outro controlar a diferença de velocidades entre os motores. Para fazer isso, basta fazer os cálculos mostrados abaixo:

$$Motor1 = canal1 - canal2$$
$$Motor2 = canal1 + canal2 - 1024$$

onde *canal1* e *canal2* são os valores de pulso normalizados (entre 0 e 1023). Nesse caso o canal1 estaria relacionado à velocidade média do veículo, e o canal 2 estaria relacionado a mudanças de direção, ou seja, canal 2 igual a 512 faria o veículo se locomover em linha reta, e qualquer outro valor geraria diferença de velocidades entre as rodas, gerando curvas.

O código desse modo é mostrado abaixo:

```
Right_Drive = LimitDrive(servo1.width - servo2.width);
Left_Drive = LimitDrive(servo1.width + servo2.width - 1024);
```

4.3.9. Rotina de Calibragem

Devido a diferenças entre sistemas de rádio controle, o tamanho mínimo (*low*), aproximadamente 1 ms, e máximo (*high*), aproximadamente 2 ms, do pulso do receptor, podem variar, fazendo com que apareçam pequenos erros de acionamento ao normalizar os pulsos com valores errados. Para corrigir esse problema, foi criada uma rotina de calibragem que pode ser ativada, através de um botão, durante os 0,5 s após o circuito ser ligado ou resetado. Esse limite de tempo para o acionamento dessa rotina é para evitar que o programa entre em modo de calibragem caso o botão seja pressionado acidentalmente durante o funcionamento normal do circuito.

A rotina de calibragem compara os pulsos lidos com os valores de *low* e *high* atuais e caso algum pulso seja menor ou maior do que estes valores, ele será o novo valor. Ainda assim, esses novos valores de *low* e *high* ainda devem obedecer aos valores *MINVALIDWIDTH* e *MAXVALIDWIDTH*, que valem 102 e 248, respectivamente. Para que a operação de calibragem funcione corretamente, todos os canais do controle que estão conectados ao circuito devem ser variados em toda a sua excursão para que os valores de máximo e mínimo reais sejam lidos corretamente.

Para que esses novos valores lidos não se percam ao desligar o circuito, eles são gravados na memória de dados não volátil (EEPROM) do PIC.

A rotina de calibragem para o *servo1* é mostrada no quadro abaixo. Cada canal de servo tem um código idêntico a esse.

```
        if (servo1.done) {
                servo1.done = false;

                servo1.rcpulse = servo1.end - servo1.start - 15;

                if ((MINVALIDWIDTH < servo1.rcpulse) && (servo1.rcpulse < MAXVALIDWIDTH))
                {
                        if (servo1.rcpulse > Canal1.high) {
                                Canal1.high = servo1.rcpulse;
                        }
                        if (servo1.rcpulse < Canal1.low) {
                                Canal1.low = servo1.rcpulse;
                        }
                }
        }
```

4.3.10. Acionamento do relé

O circuito proposto permite a utilização de um relé opcional, independente do controle de velocidade dos motores, para acionar outros sistemas. Existem dois modos de acionamento do relé, sendo eles o modo simples e o modo duplo. No modo simples apenas um canal é utilizado para o acionamento, enquanto no modo duplo dois canais são utilizados.

Para que a saída seja acionada no modo simples, é necessário que o canal tenha um pulso maior do que o valor de disparo superior – AUX_HIGH_TH. A saída só é desligada caso o pulso seja menor do que o valor de disparo inferior – AUX_LOW_TH.

No modo duplo, a saída só é ativada caso ambos os canais tenham pulsos maiores do que AUX_HIGH_TH, dois terços do valor máximo ou 682, e desativada se pelo menos um for menor do que AUX_LOW_TH, um terço do valor máximo ou 341.

O código de acionamento do relé é mostrado abaixo:

```
	switch (sts.AuxMode)
	{
		case AUX_SINGLE: {
			if (servo3.valid) {
				if (servo3.width > AUX_HIGH_TH) {
					Aux_On();
				}
				else if (servo3.width < AUX_LOW_TH) {
					Aux_Off();
				}
			}
			break;
		}

		case AUX_DOUBLE: {
			if ((servo3.valid) && (servo4.valid)) {
				if ((servo3.width > AUX_HIGH_TH) && (servo4.width > AUX_HIGH_TH)) {
					Aux_On();
				}
				else if ((servo3.width < AUX_LOW_TH) || (servo4.width < AUX_LOW_TH)) {
					Aux_Off();
				}
			}
			break;
		}
	}
```

4.3.11. Rotina principal (main)

A rotina principal, ou *main*, é a primeira rotina a ser executada após um reset ou ligar o micro-controlador e, por isso, ela controla todo o sistema.

Portanto a *main* deve chamar as rotinas de inicialização, tanto a do micro-controlador quanto a do motor, e conferir se o botão de calibragem foi pressionado durante os 500 ms iniciais, preparando todo o sistema para o correto funcionamento.

Após toda a incialização, o programa entra num *loop* infinito, sendo executado todos os passos necessários para o acionamento dos motores e do relé, incluindo a interpretação dos sinais, manipulação matemática destes e limitação de taxa de variação da saída. Além disso, antes do acionamento dos motores, ainda é verificado se os pulsos recebidos foram válidos e, caso não tenham sido, a saída correspondente ao pulso inválido é desligada.

Abaixo encontra-se a parte de inicialização da rotina.

```
void main (void)
{
	init_proc();			// Inicializa o PIC

	while (ticks < 500) {		// Espera 500 ms
		if (RB0 == 0) {			// Se o botão foi pressionado
			Calibrate();	// Calibra os canais
			break;
		}
	}

	init_motor();			// Inicializa a placa de potencia

	Status_Led_On();			// Liga LED de status – tudo OK

	RIGHT_ENABLE();			// Habilita eletronicas de potencia
	LEFT_ENABLE();
```

O *loop* infinito, juntamente com a normalização e checagem dos pulsos pela rotina *CheckPulse()*, é mostrado abaixo.

```
while (true) {					// loop Infinito
	if (sts.tmr_10ms) {			// Só executa tudo a cada 10 ms
		sts.tmr_10ms = false;		// zera o flag
		sts.invalid_rc1 = CheckPulse(&servo1); // Normaliza e checa se o pulso esta correto
		sts.invalid_rc2 = CheckPulse(&servo2); // ATENCAO!!! A função CheckPulse retorna
		sts.invalid_rc3 = CheckPulse(&servo3); // verdadeiro se o pulso estiver inválido
		sts.invalid_rc4 = CheckPulse(&servo4);
```

O modo de acionamento dos motores é mostrado abaixo, juntamente com as manipulações efetuadas nos valores de acionamento dos motores.

```
	switch (sts.MixMode)
	{
		case MIXED: {
			Right_Drive = LimitDrive(servo1.width - servo2.width);
			Left_Drive = LimitDrive(servo1.width + servo2.width - 1024);
			break;
		}
		case STRAIGHT: {
			Right_Drive = LimitDrive(servo1.width - 0x200);
			Left_Drive = LimitDrive(servo2.width - 0x200);

			Right_Drive = Right_Drive * 2;
			Left_Drive = Left_Drive * 2;

			if (Right_Drive == 0x03FE)
				Right_Drive = 0x03FF;
			if (Left_Drive == 0x03FE)
				Left_Drive = 0x03FF;
			break;
		}
	}
```

```
	Right_Drive = DoDeadBand(Right_Drive);
	Left_Drive = DoDeadBand(Left_Drive);

	Right_Drive = DoSlew(Right_Drive, &servo1, Slew);
	Left_Drive = DoSlew(Left_Drive, &servo2, Slew);
```

O acionamento dos motores é mostrado abaixo.

```
	if (sts.invalid_rc1) {		// Se for inválido
		RIGHT_DISABLE();		// Desliga motor
		Right_Drive = 0;
	}
	else {				// se for válido
		RIGHT_ENABLE();		// Liga motor
	}
	if (sts.invalid_rc2) {		// Se for inválido
		LEFT_DISABLE();		// Desliga motor
		Left_Drive = 0;
	}
	else {				// se for válido
		LEFT_ENABLE();		// Liga motor
	}

	if (Right_Drive >= 0) {		// Maior que zero
		RIGHT_FOWARD();		// Gira para frente
	}
	else {				// Menor que zero
		RIGHT_REVERSE();		// Gira para trás
	}

	if (Left_Drive >= 0) {		// Maior que zero
		LEFT_FOWARD();		// Gira para frente
	}
	else {				// Menor que zero
		LEFT_REVERSE();		// Gira para trás
	}

	set_pwm1(Right_Drive);		// Coloca novo valor de PWM
	set_pwm2(Left_Drive);		// Coloca novo valor de PWM
```

Por último, foram implementados ambos os modos de acionamento dos relés. A escolha entre eles é feito pela variável *sts.AuxMode*.

```
	switch (sts.AuxMode)
	{
		case AUX_SINGLE: {
			if (servo3.valid) {
				if (servo3.width > AUX_HIGH_TH) {
					Aux_On();
				}
				else if (servo3.width < AUX_LOW_TH) {
					Aux_Off();
				}
			}
			break;
		}
			case AUX_DOUBLE: {
			if ((servo3.valid) && (servo4.valid)) {
				if ((servo3.width > AUX_HIGH_TH) && (servo4.width > AUX_HIGH_TH)) {
```

```
                                Aux_On();
                        }
                        else if ((servo3.width < AUX_LOW_TH) || (servo4.width < AUX_LOW_TH)) {
                                Aux_Off();
                        }
                }
                break;
        }
}
```

5 – Conclusões e Sugestões de Trabalhos Futuros

Nesse trabalho foi apresentado o resultado do desenvolvimento de um conjunto de circuitos para o controle de motores de corrente contínua de alta potência, consistindo de um circuito de controle que recebe sinais de um receptor de rádio controle e os transforma em sinais que podem ser interpretados por um circuito de potência, o qual aciona os motores.

Os testes feitos em dois sistemas robóticos diferentes, ambos com motores que apresentam potências elétricas superiores a 1,5 HP, chegando a mais de 3,5 HP, mostraram que o circuito de potência suporta as cargas sem problemas. Apesar disso, a segunda versão feita do circuito de controle, modificada para acrescentar chaves de segurança, apresentou problemas na gravação do software no micro-controlador, causada pela mudança na disposição das trilhas dos sinais de gravação. Por isso, em próximas versões, é recomendado que essas trilhas sejam mantidas no menor comprimento possível, ligando os pinos do conector com os do PIC por um caminho direto, ininterrupto.

Outro detalhe em relação ao hardware do circuito de controle é a utilização de chips SMD (montagem em superfície), o que dificultou o trabalho de correção de erros nos circuitos que se mostraram defeituosos. Recomenda-se, também, o uso de chips DIP comuns, com o uso de soquetes, para facilitar a troca, caso o micro-controlador queime, apesar de fazer com que o circuito fique um pouco maior.

Além disso, na parte de proteção do software atual existe um *bug* (problema no programa) na parte do código que desativa os motores que deve ser corrigido. Ele se apresenta somente no modo mixado, já que o software relaciona um canal do receptor com uma saída específica. Como no

modo mixado dois canais são utilizados para criar os valores de saída de ambos os motores, os dois devem ser desativados caso pelo menos um dos canais se mostre inválido.

O código incorreto, juntamente com a correção é mostrado abaixo:

Código incorreto	Novo código

`if (sts.invalid_rc1) {`
`  RIGHT_DISABLE();`
`  Right_Drive = 0;`
`}`
`else {`
`  RIGHT_ENABLE();`
`}`
`if (sts.invalid_rc2) {`
`  LEFT_DISABLE();`
`  Left_Drive = 0;`
`}`
`else {`
`  LEFT_ENABLE();`
`}`
	`if ((sts.invalid_rc1) || (sts.invalid_rc2)) {`
	`  RIGHT_DISABLE();`
	`  LEFT_DISABLE();`
	`  Right_Drive = 0;`
	`  Left_Drive = 0;`
	`}`
	`else {`
	`  RIGHT_ENABLE();`
	`  LEFT_ENABLE();`
	`}`

Apesar do sucesso do conjunto, ainda existem várias características que podem ser adicionadas ao circuito de controle. A principal delas é em relação ao reconhecimento de pulsos inválidos. Atualmente somente a largura do nível alto do pulso é medida e comparada com valores limítrofes para verificar se está dentro da faixa de aceitação. O circuito de controle poderia verificar o período desses pulsos, que deve estar entre 18 ms e 25 ms, além de só considerar um pulso válido após receber um número mínimo de pulsos corretos. Essas duas mudanças aumentariam muito a segurança do circuito.

O circuito ainda tem problemas na leitura dos pulsos, com os valores lidos variando um pouco, principalmente devido à latência variável para o início da execução da rotina de interrupção, já que os pulsos são lidos em passos de 8 μs e a latência, com o PIC rodando a 4 MHz, pode variar de 3 μs a 4 μs. Aumentando a freqüência do micro-controlador, o efeito da latência é diminuído. Aumentando a freqüência para 16 MHz, por exemplo, faria com que a latência diminuísse para 0,75 μs a 1 μs, apesar de algumas mudanças no programa terem que ser feitas para manter os *timers*

funcionando nas freqüências certas. Provavelmente, em 16 MHz, os *timers* 0 e 1 deverão trocar de função para que se consiga isso.

Caso os circuitos sejam utilizados para controlar a velocidade de motores com avanço de fase para a locomoção de robôs, pode ser necessário que exista uma limitação da saída, podendo ser só numa direção do motor ou em ambas, para que motores opostos girem à mesma velocidade. Para incluir essa opção, logo antes de atualizar o valor do PWM dos motores deve ser feita uma nova normalização dos valores. Por exemplo, caso o motor deva ser limitado até 50%, ou seja, seu valor máximo é 512, quando girando para frente, nada deve ser feito enquanto o motor gira para trás e ao começar a girar para frente, deve ser feito a seguinte conta:

$$saída = \frac{512 \cdot valor\_atual}{1023}$$

Antes de implementar essa opção, deve ser feito um estudo para saber se o PIC consegue terminar todas as contas adicionais (duas multiplicações e duas divisões para cada motor) entre o recebimento de dois pulsos do receptor de rádio controle. Caso não seja possível, pode ser feita uma diminuição na resolução do PWM, utilizando somente 8 bits para os valores dos sinais lidos, ou seja, −128 a 127, completando os bits do PWM com 0's ou 1's, dependendo do caso, já que são necessários 10 bits para o PWM.

Outras melhorias podem ser feitas para facilitar a utilização dos vários modos de execução existentes. Atualmente é necessário regravar o micro-controlador com os novos parâmetros de utilização. Uma interface poderia ser feita para permitir a programação desses parâmetros pela porta serial disponível no PIC, utilizando o computador para enviar os novos parâmetros. O software atual foi feito pensando nessas futuras atualizações, já que todos os parâmetros (modo do relé, modo de acionamento dos motores, *slew rate*, banda morta e calibragem) são gravados na memória EEPROM

do PIC e lidos durante a inicialização do circuito. Uma outra opção é criar uma interface no circuito utilizando botões e um display de cristal líquido (LCD), fazendo o circuito ficar altamente portátil, sem a necessidade de um computador para efetuar as mudanças necessárias nos parâmetros.

6 – Bibliografia

[1] NPC Robotics Inc., www.npcrobotics.com

[2] Intersil, www.intersil.com

[3] Microchip, www.microchip.com

[4] Grupo de discussão *Open Source Motor Controllers*, groups.yahoo.com/group/osmc

[5] Philips Semiconductors, www.semiconductors.philips.com

[6] Sedra, A.S.; Smith, K.C., Microeletrônica, Ed. Makron Books, 2000.

[7] Pressman, A.I., Switching Power Supply Design 2nd ed, Ed. McGraw-Hill, 1998.

[8] MicrochipC.com, www.microchipc.com

[9] Horovitz, P., Hill, W., The Art of Electronics, Cambridge University Press, 1989.

Anexo A - Esquemáticos

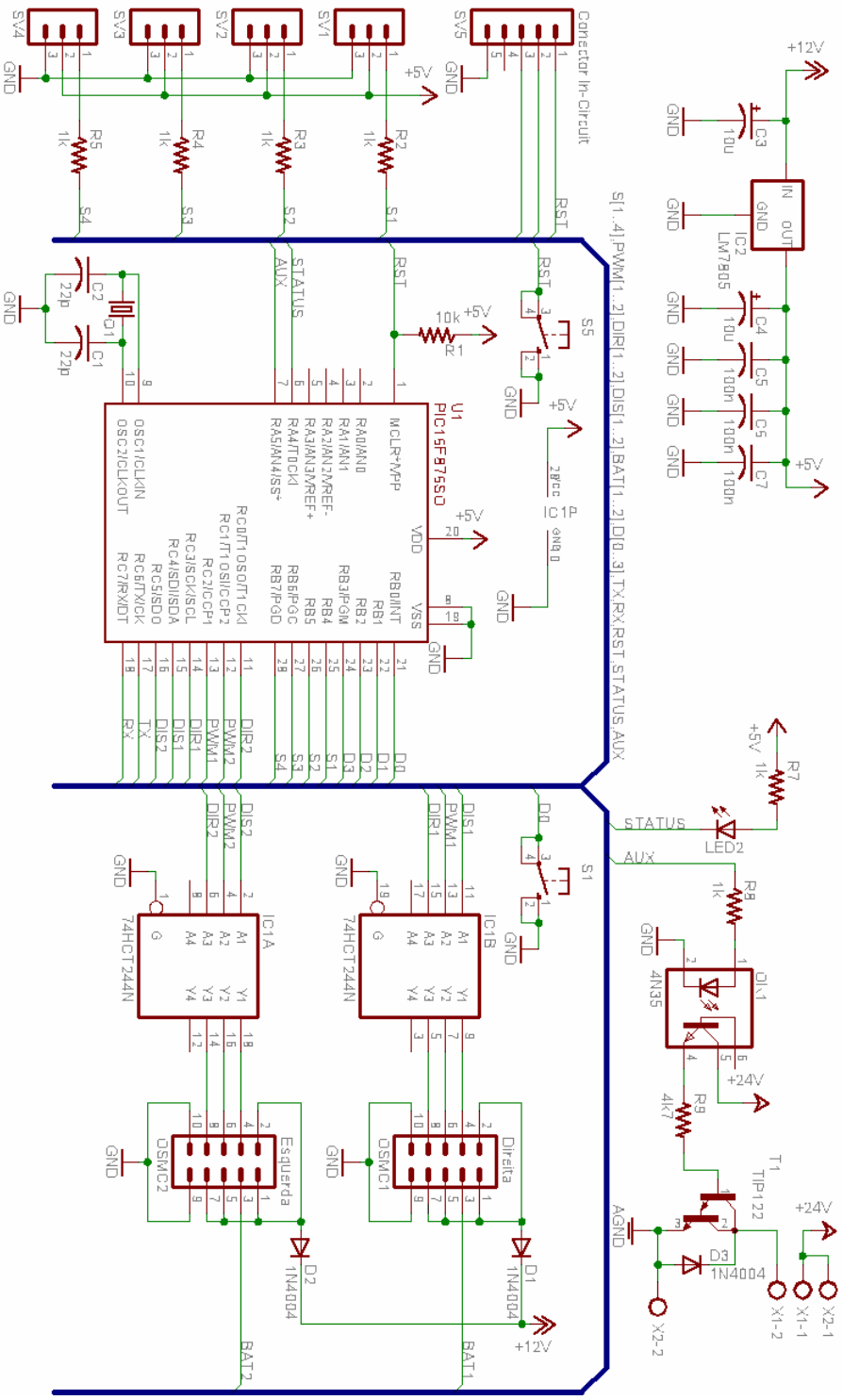

Fig. A.1 – Esquemático do circuito de controle

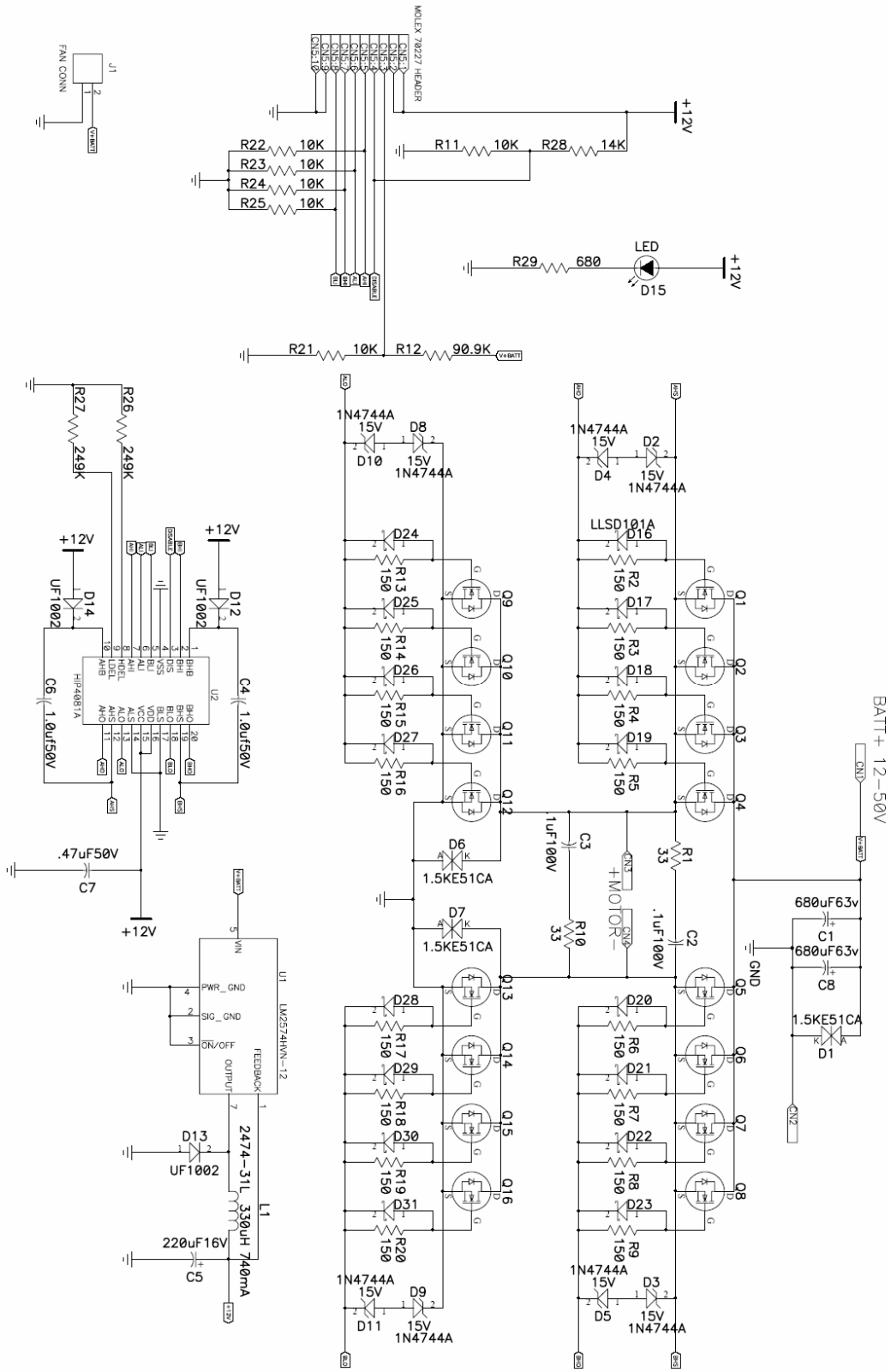

Fig. A.2 – Esquemático da placa de potência.

Anexo B – Código completo

3.1. main.c

```
/*****************************************************************************/
/* Programa                  : MAIN.C                                        */
/* Função                    : Programa de Controle do OSMC                  */
/* Autor                     : Felipe Maimon                                 */
/* Linguagem                 : HiTech C 8.02 PL1 (ANSI C)                    */
/* Plataforma                : PIC16F876A @ 4MHz                             */
/* Hardware                  : RioBotz 1.0                                   */
/*                                                                           */
/* Version                   : 0.1                                           */
/*****************************************************************************/

#define _MAIN_C

//#define VERBOSE

#include <pic.h>
#include "commdefs.h"
#include "motor.h"
#include "serial.h"
#include "eeprom.h"
#include "main.h"

__CONFIG( 0x3F71 );                             // XT | PWRTEN | BOREN | LVPDIS


char Slew;
int Left_Drive, Right_Drive;

/*****************************************************************************/
/* Main Program                                                              */
/*****************************************************************************/
void main (void)
{
          init_proc();

          putst("Controlador OSMC\n");

          while (ticks < 500) {
                    if (RB0 == 0) {
                              Calibrate();
                              break;
                    }
          }

          init_motor();

          Status_Led_On();

          RIGHT_ENABLE();
          LEFT_ENABLE();

          while (true) {

                    if (sts.tmr_10ms) {

                              sts.tmr_10ms = false;

                              sts.invalid_rc1 = CheckPulse(&servo1);
                              sts.invalid_rc2 = CheckPulse(&servo2);
                              sts.invalid_rc3 = CheckPulse(&servo3);
                              sts.invalid_rc4 = CheckPulse(&servo4);

                              switch (sts.MixMode)
                              {
                                        case MIXED: {
                                                  Right_Drive = LimitDrive(servo1.width - servo2.width);
                                                  Left_Drive = LimitDrive(servo1.width + servo2.width - 1024);
                                                  break;
                                        }
                                        case STRAIGHT: {
                                                  Right_Drive = LimitDrive(servo1.width - 0x200);
                                                  Left_Drive = LimitDrive(servo2.width - 0x200);

                                                  Right_Drive = Right_Drive * 2;
                                                  Left_Drive = Left_Drive * 2;

                                                  if (Right_Drive == 0x03FE)
                                                            Right_Drive = 0x03FF;
                                                  if (Left_Drive == 0x03FE)
                                                            Left_Drive = 0x03FF;
                                                  break;
                                        }
```

```
                                }

                                Right_Drive = DoDeadBand(Right_Drive);
                                Left_Drive = DoDeadBand(Left_Drive);

                                Right_Drive = DoSlew(Right_Drive, &servo1, Slew);
                                Left_Drive = DoSlew(Left_Drive, &servo2, Slew);

                                if (sts.invalid_rc1) {
                                        RIGHT_DISABLE();
                                        Right_Drive = 0;
                                }
                                else {
                                        RIGHT_ENABLE();
                                }
                                if (sts.invalid_rc2) {
                                        LEFT_DISABLE();
                                        Left_Drive = 0;
                                }
                                else {
                                        LEFT_ENABLE();
                                }

                                if (Right_Drive >= 0) {
                                        RIGHT_FOWARD();
                                }
                                else {
                                        RIGHT_REVERSE();
                                }

                                if (Left_Drive >= 0) {
                                        LEFT_FOWARD();
                                }
                                else {
                                        LEFT_REVERSE();
                                }

                                set_pwm1(Right_Drive);
                                set_pwm2(Left_Drive);

                                switch (sts.AuxMode)
                                {
                                        case AUX_SINGLE: {
                                                if (servo3.valid) {
                                                        if (servo3.width > AUX_HIGH_TH) {
                                                                Aux_On();
                                                        }
                                                        else {
                                                                Aux_Off();
                                                        }
                                                }
                                                break;
                                        }

                                        case AUX_DOUBLE: {
                                                if ((servo3.valid) && (servo4.valid)) {
                                                        if ((servo3.width > AUX_HIGH_TH) && (servo4.width > AUX_HIGH_TH)) {
                                                                Aux_On();
                                                        }
                                                        else {
                                                                Aux_Off();
                                                        }
                                                }
                                                break;
                                        }
                                }

#ifdef VERBOSE1
                                PrintValues();
#endif
                        }
                }
        }
/*********************************************************************************************/

/*********************************************************************************************/
/* Rotina de Inicialização do processador                                                    */
/*********************************************************************************************/
void init_proc (void)
{
        TRISA = 0x00;                                   // PORTA é toda saída
        TRISB = 0xF1;                                   // RB<4:7> são entradas (4 canais) e
                                                        // RB0 é entrada do botão de calibração
        TRISC = 0x80;                                   // PORTC é toda saída, exceto RC7 (RX)

        PORTA = 0x00;
        PORTB = 0x00;
        PORTC = 0x00;

        Status_Led_Off();
```

```
          sts.tmr_1ms =
          sts.tmr_10ms =
          sts.invalid_rc1 =
          sts.invalid_rc1 =
          sts.invalid_rc1 =
          sts.invalid_rc1 = true;
          sts.Disable_All = true;
          sts.MixMode = EEPROM_READ(eeaddMixMode);
          sts.AuxMode = EEPROM_READ(eeaddAuxMode);

// 0 - tempo é 0
// 1 - tempo ~ 50 ms
// 2 - tempo ~ 100 ms
// 3 - tempo ~ 150 ms
// 4 - tempo ~ 200 ms
// 5 - tempo ~ 250 ms
// 6 - tempo ~ 300 ms
// 7 - tempo ~ 400 ms
// 8 - tempo ~ 500 ms
// 9 - tempo ~ 750 ms
// 10 - tempo ~ 1000 ms
          Slew = SR_100ms;

          ticks = 0;

          ADCON1 = 0x07;                                  // Configura toda a PORTA como digital

          Setup_Counters( RTCC_INTERNAL, RTCC_DIV_4 );    // Configura TMR0 (RTCC) para clock
                                                          // interno com pre-scaler em 1:4 para
                                                          // ter um Time-out de 1,024 ms ~ 1 ms

          setup_timer_1( T1_DIV_BY_8 | T1_ENABLED );      // Timer 1 incrementa a cada 8 us
                                                          // e já está rodando

          PR2 = 0xFF;                                     // Fpwm = 3,906 kHz
          set_ccp1(0);                                    // Duty Cycle = 0 (sem potência nos
          set_ccp2(0);                                    // Motores)
          setup_ccp1( CCP_PWM );                          // CCP1 em modo PWM
          setup_ccp2( CCP_PWM );                          // CCP2 em modo PWM
          setup_timer_2( T2_DIV_BY_1, 1);                 // Seta o Prescaler e o Postscaler do
                                                          // Timer 2 em 1:1 e liga o Timer 2
          RBPU = 0;                                       // Ativa os Pull-ups na PORTB para o caso de não haver sinal

          serial_setup();

          enable_interrupts( RB_CHANGE | INT_TIMER0 );    // Habilita as interrupções:
                                                          // RB on-change -> sinal rádio
                                                          // TMR0 -> int a cada 1ms

          GIE = 1;                                        // Habilita todas as interrupções
}
```

3.2. main.h

```
/*****************************************************************************************/
/* Programa                  : MAIN.H                                                    */
/* Função                    : Header do MAIN                                            */
/* Autor                     : Felipe Maimon                                             */
/*                                                                                       */
/*****************************************************************************************/

#ifndef _MAIN_H
#define _MAIN_H

/*****************************************************************************************/
/* Processor speed defines                                                               */
/*****************************************************************************************/
#define  _CLOCK_ 4000000      // Cristal de 4MHz
#define Status_Led_On()       RA4 = 0;
#define Status_Led_Off()      RA4 = 1;
#define Aux_On()              RA5 = 1;
#define Aux_Off()             RA5 = 0;

/*****************************************************************************************/
/* Struct defines                                                                        */
/*****************************************************************************************/

struct ctr_status
{
          unsigned tmr_1ms             :1;               // flag setada a cada 1 ms
          unsigned tmr_10ms            :1;               // flag setada a cada 10 ms
          unsigned MixMode             :1;               // flag do modo de controle (Mixado ou Separado)
          unsigned AuxMode             :1;               // flag do acionamento do motor auxiliar
          unsigned invalid_rc1         :1;               // flag de erro do servo1
```

```
        unsigned invalid_rc2         :1;               // flag de erro do servo2
        unsigned invalid_rc3         :1;               // flag de erro do servo3
        unsigned invalid_rc4         :1;               // flag de erro do servo4
        unsigned Disable_All         :1;               // flag para parar o Robo
}

typedef struct
{
        unsigned char       start;                     // Start time
        unsigned char       end;                       // end time
        unsigned char       low;                       // Scaling factor
        unsigned char       high;                      // scaling factor
        unsigned char       rcpulse;                   // last calculated R/C pulse width
//      unsigned char       pulse;                     // numbers of valid pulses
        unsigned int        width;                     // Normalized pulse width
        unsigned int        Drive;
        unsigned            done     :1;               // pulse flags
        unsigned            valid    :1;               // Marks data as valid or not.
}
RCcontrolBlock, *pRCcontrolBlock;

/*****************************************************************************/
/* Variable and constant declarations                                        */
/*****************************************************************************/
extern volatile struct ctr_status sts;
extern volatile RCcontrolBlock servo1, servo2, servo3, servo4;
extern volatile unsigned int ticks;
/*****************************************************************************/
/* Function prototypes                                                       */
/*****************************************************************************/

void init_proc(void);

#endif // _MAIN_H

// ***** EOF MAIN.H *****
```

3.3. motor.c

```
#define _MOTOR_C

#include <pic.h>
#include "commdefs.h"
#include "delay.h"
#include "main.h"
#include "serial.h"
#include "eeprom.h"
#include "motor.h"

char DeadBand;

void init_motor(void)
{
        int time = ticks + 500;

        while (ticks < time);                          //espera 1/2 segundo antes de inicializar tudo!

        LEFT_ENABLE();
        RIGHT_ENABLE();
        DelayUs(200);
        LEFT_DISABLE();
        RIGHT_DISABLE();

        servo1.low = EEPROM_READ(eeaddRC1low);
        servo1.high = EEPROM_READ(eeaddRC1high);
        servo2.low = EEPROM_READ(eeaddRC2low);
        servo2.high = EEPROM_READ(eeaddRC2high);
        servo3.low = EEPROM_READ(eeaddRC3low);
        servo3.high = EEPROM_READ(eeaddRC3high);
        servo4.low = EEPROM_READ(eeaddRC4low);
        servo4.high = EEPROM_READ(eeaddRC4high);

        DeadBand = EEPROM_READ(eeaddDeadBand);
}

bit CheckPulse(pRCcontrolBlock p)
{
        unsigned char tWidth;
        unsigned long temp;

        if (p->done)
        {
                p->rcpulse = tWidth = p->end - p->start - 15;          // Subtrai 15 para deslocar
                                                                       // o tempo para a esquerda e
                                                                       // conseguir ler até 2.17 ms
```

```
                    p->done = FALSE;            // Reset Pulse Trap

                    if (MINVALIDWIDTH < tWidth && tWidth < MAXVALIDWIDTH)
                    {
                              if (tWidth > p->high)   // limit measured range.
                                        tWidth = p->high;
                              else if (tWidth < p->low)
                                        tWidth = p->low;

                              temp = tWidth - p->low;
                              temp = ((temp << 8) << 2) - temp;
                              p->width = temp / (unsigned)(p->high - p->low);

                              p->valid = true;
                              }
                    else
                    {
                              p->valid = false;
                    }
          }
          return (!p->valid);   // Return True if not valid
}

int DoDeadBand(int drive)
{
          if (((0 - DeadBand) < drive) && (drive < DeadBand))
          {
                    drive = 0;
          }
          return (drive);
}


/*********************************************************************
SlewRate é um inteiro entre 0 e 10. Os valores do SR correspondente estão
na tabela abaixo, em ordem crescente
*/
int DoSlew(int RequestedDrive, pRCcontrolBlock p, char SlewRate)
{

const int SlewTable[] = {     1023,     // 0 - tempo entre máximo e parado é 0
                              204,      // 1 - tempo ~ 50 ms
                              102,      // 2 - tempo ~ 100 ms
                              68,       // 3 - tempo ~ 150 ms
                              51,       // 4 - tempo ~ 200 ms
                              41,       // 5 - tempo ~ 250 ms
                              34,       // 6 - tempo ~ 300 ms
                              26,       // 7 - tempo ~ 400 ms
                              20,       // 8 - tempo ~ 500 ms
                              14,       // 9 - tempo ~ 730 ms
                              10        // 10 - tempo ~ 1000 ms
                              };

    int CurrentDrive = p->Drive;

    SlewRate = SlewTable[SlewRate];

    if (RequestedDrive > CurrentDrive)
    {
        CurrentDrive += SlewRate;
        if (CurrentDrive > RequestedDrive)
        {
            CurrentDrive = RequestedDrive;
        }
    }
    else
    {
        CurrentDrive -= SlewRate;
        if (CurrentDrive < RequestedDrive)
        {
            CurrentDrive = RequestedDrive;
        }
    }
    p->Drive = CurrentDrive;
    return CurrentDrive;
}

int LimitDrive(int Drive)
{
    if (Drive > 511)
        Drive = 511;
    if (Drive < -512)
        Drive = -512;
    return Drive;
}

void Calibrate(void)
{
          struct
          {
                    char high;
```

```
          char low;
        }
        Canal1, Canal2, Canal3, Canal4;
        int time = 0;

        Canal1.high =
        Canal2.high =
        Canal3.high =
        Canal4.high = 0;
        Canal1.low =
        Canal2.low =
        Canal3.low =
        Canal4.low = 255;

        time = ticks + 10000;

        Status_Led_Off();

        while (ticks < time)
        {
          if (servo1.done) {
                    servo1.done = false;

                    servo1.rcpulse = servo1.end - servo1.start - 15;

                if ((MINVALIDWIDTH < servo1.rcpulse) && (servo1.rcpulse < MAXVALIDWIDTH))
                    {
                              if (servo1.rcpulse > Canal1.high) {
                                        Canal1.high = servo1.rcpulse;
                              }
                              if (servo1.rcpulse < Canal1.low) {
                                        Canal1.low = servo1.rcpulse;
                              }
                    }
          }
          if (servo2.done) {
                    servo2.done = false;

                    servo2.rcpulse = servo2.end - servo2.start - 15;

                if ((MINVALIDWIDTH < servo2.rcpulse) && (servo2.rcpulse < MAXVALIDWIDTH))
                    {
                              if (servo2.rcpulse > Canal2.high) {
                                        Canal2.high = servo2.rcpulse;
                              }
                              if (servo2.rcpulse < Canal2.low) {
                                        Canal2.low = servo2.rcpulse;
                              }
                    }
          }
          if (servo3.done) {
                    servo3.done = false;

                    servo3.rcpulse = servo3.end - servo3.start - 15;

                if ((MINVALIDWIDTH < servo3.rcpulse) && (servo3.rcpulse < MAXVALIDWIDTH))
                    {
                              if (servo3.rcpulse > Canal3.high) {
                                        Canal3.high = servo3.rcpulse;
                              }
                              if (servo3.rcpulse < Canal3.low) {
                                        Canal3.low = servo3.rcpulse;
                              }
                    }
          }
          if (servo4.done) {
                    servo4.done = false;

                    servo4.rcpulse = servo4.end - servo4.start - 15;

                if ((MINVALIDWIDTH < servo4.rcpulse) && (servo4.rcpulse < MAXVALIDWIDTH))
                    {
                              if (servo4.rcpulse > Canal4.high) {
                                        Canal4.high = servo4.rcpulse;
                              }
                              if (servo4.rcpulse < Canal4.low) {
                                        Canal4.low = servo4.rcpulse;
                              }
                    }
          }
        }

        EEPROM_WRITE(eeaddRC1low, Canal1.low);
        EEPROM_WRITE(eeaddRC1high, Canal1.high);
        EEPROM_WRITE(eeaddRC2low, Canal2.low);
        EEPROM_WRITE(eeaddRC2high, Canal2.high);
        EEPROM_WRITE(eeaddRC3low, Canal3.low);
        EEPROM_WRITE(eeaddRC3high, Canal3.high);
        EEPROM_WRITE(eeaddRC4low, Canal4.low);
        EEPROM_WRITE(eeaddRC4high, Canal4.high);
}
```

3.4. *motor.h*

```
#ifndef _MOTOR_H
#define _MOTOR_H

#include <pic.h>
#include "main.h"

// Disable Bits
static volatile bit DIS1     @ (unsigned)&PORTC*8+4;
static volatile bit DIS2     @ (unsigned)&PORTC*8+5;
// Direction Bits
static volatile bit DIR1     @ (unsigned)&PORTC*8+3;
static volatile bit DIR2     @ (unsigned)&PORTC*8+0;

#define RIGHT_FOWARD()       DIR1 = 0
#define RIGHT_REVERSE()      DIR1 = 1
#define RIGHT_DISABLE()      DIS1 = 1
#define RIGHT_ENABLE()       DIS1 = 0

#define LEFT_FOWARD()        DIR2 = 0
#define LEFT_REVERSE()       DIR2 = 1
#define LEFT_DISABLE()       DIS2 = 1
#define LEFT_ENABLE()        DIS2 = 0

#define MINVALIDWIDTH        102
#define MAXVALIDWIDTH        248
#define RC_MIN                        135
#define RC_MAX                        210
#define AUX_LOW_TH           (1023 * 1 / 3)
#define AUX_HIGH_TH          (1023 * 2 / 3)

#define SR_0ms               0
#define SR_50ms              1
#define SR_100ms             2
#define SR_150ms             3
#define SR_200ms             4
#define SR_250ms             5
#define SR_300ms             6
#define SR_400ms             7
#define SR_500ms             8
#define SR_750ms             9
#define SR_1000ms            10

#define MIXED                0
#define STRAIGHT             1

#define AUX_SINGLE           0                          // Acionamento com um canal
#define AUX_DOUBLE           1                          // Acionamento com 2 canais

#define DEADBAND             130

void init_motor(void);
bit CheckPulse(pRCcontrolBlock p);
int DoDeadBand(int drive);
int DoSlew(int RequestedDrive, pRCcontrolBlock p, char SlewRate);
int LimitDrive(int Drive);
void Calibrate(void);

#endif _MOTOR_H
```

3.5. *int.c*

```
/*****************************************************************************/
/* Programa                  : INT.C                                         */
/* Função                    : Rotinas de Interrupção                        */
/* Autor                     : Felipe Maimon                                 */
/* Linguagem                 : HiTech C 8.02 PL1 (ANSI C)                    */
/* Plataforma                : PIC16F876A @ 4MHz                             */
/* Hardware                  : RioBotz 1.0                                   */
/*                                                                           */
/* Version                   : 0.1                                           */
/*****************************************************************************/

#define _INT_C
#include "commdefs.h"
#include "motor.h"
#include "main.h"

//#define RB_ADJ


/*****************************************************************************/
```

```
/* Váriaveis externas                                                          */
/*****************************************************************************/
volatile RCcontrolBlock servo1, servo2, servo3, servo4;
volatile struct ctr_status sts;
volatile unsigned int ticks;
/*****************************************************************************/

unsigned char old_portb;
unsigned char count;

void interrupt isr (void)
{
          unsigned char time,
                                    rb_value;

          if (RBIF) {
                    RBIF = 0;

                    time = TMR1L;                                   // Lê sempre o valor do timer e da PORTB
                    rb_value = (old_portb ^ PORTB);                 // para diminuir a latência da leitura
                    old_portb = PORTB;

                    if ((rb_value & 0x10) == 0x10) {                // Se RB4 mudou

                              if (RB4 == 1){                        // Subida de Pulso
                                        servo1.start = time;        // valor inicial do pulso é time
                              }
                              else {                                // Descida do pulso
                                        servo1.end = time;          // valor final do pulso é time
                                        servo1.done = TRUE;         // Terminou o pulso
                              }
                    }
                    if ((rb_value & 0x20) == 0x20) {                // Se RB5 mudou

                              if (RB5 == 1) {                       // Subida de Pulso
                                        servo2.start = time;        // valor inicial do pulso é time
                              }
                              else {                                // Descida do pulso
                                        servo2.end = time;          // valor final do pulso é time
                                        servo2.done = TRUE;         // Terminou o pulso
                              }
                    }
                    if ((rb_value & 0x40) == 0x40) {                // Se RB6 mudou

                              if (RB6 == 1){                        // Subida de Pulso
                                        servo3.start = time;        // valor inicial do pulso é time
                              }
                              else {                                // Descida do pulso
                                        servo3.end = time;          // valor final do pulso é time
                                        servo3.done = TRUE;         // Terminou o pulso
                              }
                    }
                    if ((rb_value & 0x80) == 0x80) {                // Se RB7 mudou

                              if (RB7 == 1) {                       // Subida de Pulso
                                        servo4.start = time;        // valor inicial do pulso é time
                              }
                              else {                                // Descida do pulso
                                        servo4.end = time;          // valor final do pulso é time
                                        servo4.done = TRUE;         // Terminou o pulso
                              }
                    }
          }

          if (T0IF)
          {
                    T0IF = 0;
                    sts.tmr_1ms = 1;
                    if (++count > 9) {
                              sts.tmr_10ms = 1;
                              count = 0;
                    }
                    ticks++;
          }
}
```

3.6. commdefs.h

```
/*****************************************************************************/
/* Programa         : COMMDEFS.H                                             */
/* Função           : Common macros and defines for PIC app's not defined in pic.h      */
/* Autor            : John F. Fitter B.E.                                    */
/* Modificações     :                                                        */
/*                                                                           */
/*        - Retirado o #define TMR1                                          */
/*        - Modificado os #defines de interrupção serial de INT_RDA e INT_TBE para      */
/*        INT_RX e INT_TX, respectivamente                                   */
/*        - Adicionado #defines para interrupções existentes no PIC16F87X    */
/*                                                                           */
/*****************************************************************************/

#ifndef _COMMDEFS_H
#define _COMMDEFS_H

#include <pic.h>

// General PIC macros (additional to pic.h)
#define clrwdt()             CLRWDT()

// Macro to define the ID bytes
#define __ID(a,b,c,d)        asm("\tpsect absdata, abs ovrld, delta=2"); \
                                                    asm("\tglobal id_bytes"); \
                                                    asm("\torg 0x2000"); \
                                                    asm("id_bytes"); \
                                                    asm("\tdb "___mkstr(a)); \
                                                    asm("\tdb "___mkstr(b)); \
                                                    asm("\tdb "___mkstr(c)); \
                                                    asm("\tdb "___mkstr(d))

// General macros
#define HIBYTE(intValue)     ((intValue)>>8)
#define LOBYTE(intValue)     ((intValue)&0xff)
#define HINIBBLE(charValue)  ((charValue)>>4)
#define LONIBBLE(charValue)  ((charValue)&0xf)
#define hibyte(intValue)     ((intValue)>>8)
#define lowbyte(intValue)    ((intValue)&0xff)
#define hinibble(charValue)  ((charValue)>>4)
#define lonibble(charValue)  ((charValue)&0xf)

// Common bit defines
#define B_IN        1
#define B_OUT       0
#define B_HIGH      1
#define B_LOW       0

// Common byte defines
#define W_IN        0xff
#define W_OUT       0
#define W_HIGH      0xff
#define W_LOW       0

// Common peripheral control defines
#define P_ON        1
#define P_OFF       0

// Common user interface defines
#define I_UP        1
#define I_DOWN      0

// Logical defines
#define TRUE        1
#define FALSE       0
#define true        TRUE
#define false       FALSE

// Hardware pullup defines
#define port_b_pullups(flag)  RBPU=flag==0

// ASCII control character defines (usefull for comms)
#define A_NUL       0
#define A_SOH       1
#define A_STX       2
#define A_ETX       3
#define A_EOT       4
#define A_ENQ       5
#define A_ACK       6
#define A_BEL       7
#define A_BS        8
#define A_HT        9
#define A_LF        0xa
#define A_VT        0xb
#define A_FF        0xc
#define A_CR        0xd
#define A_SO        0xe
#define A_SI        0xf
```

```
#define A_DLE       0x10
#define A_DC1       0x11
#define A_DC2       0x12
#define A_DC3       0x13
#define A_DC4       0x14
#define A_NAK       0x15
#define A_SYN       0x16
#define A_ETB       0x17
#define A_CAN       0x18
#define A_EM        0x19
#define A_SUB       0x1a
#define A_ESC       0x1b
#define A_FS        0x1c
#define A_GS        0x1d
#define A_RS        0x1e
#define A_US        0x1f

// Timer 0 defines
#define RTCC_INTERNAL                   0                           // rtcc_state values (OR together)
#define RTCC_EXT_L_TO_H                 0x20
#define RTCC_EXT_H_TO_L                 0x30

#define RTCC_DIV_2                      0                           // ps_state values (OR together)
#define RTCC_DIV_4                      1
#define RTCC_DIV_8                      2
#define RTCC_DIV_16                     3
#define RTCC_DIV_32                     4
#define RTCC_DIV_64                     5
#define RTCC_DIV_128                    6
#define RTCC_DIV_256                    7
#define WDT_18MS                        8
#define WDT_36MS                        9
#define WDT_72MS                        0xa
#define WDT_144MS                       0xb
#define WDT_288MS                       0xc
#define WDT_576MS                       0xd
#define WDT_1152MS                      0xe
#define WDT_2304MS                      0xf

#define Setup_Counters(rtcc_state,ps_state)      OPTION=(OPTION&0xc0)|rtcc_state|ps_state
#define get_rtcc()                      TMR0
#define set_rtcc(tvalue)                TMR0=tvalue
#define get_timer0()                    get_rtcc()
#define set_timer0(tvalue)              set_rtcc(tvalue)

// Timer 1 defines
#define T1_ENABLED                      1                           // timer 1 modes (OR together)
#define T1_INTERNAL                     1
#define T1_EXTERNAL                     7
#define T1_EXTERNAL_SYNC                3
#define T1_CLK_OUT                      8
#define T1_DIV_BY_1                     0
#define T1_DIV_BY_2                     0x10
#define T1_DIV_BY_4                     0x20
#define T1_DIV_BY_8                     0x30

#define setup_timer_1(mode)             T1CON=mode
#define enable_timer_1(state)           TMR1ON=state
#define get_timer1()                    TMR1L
#define set_timer1(tvalue)              TMR1L=tvalue

// Timer 2 defines
#define T2_DISABLED                     0                           // timer 2 modes (OR together)
#define T2_DIV_BY_1                     4                           // T2_DISABLED must be on its own
#define T2_DIV_BY_4                     5                           // postscale is 1 to 16
#define T2_DIV_BY_16                    7

#define setup_timer_2(mode,postscale)  T2CON=mode|((postscale-1)<<3)
#define get_timer2()                    TMR2
#define set_timer2(tvalue)              TMR2=tvalue

// CCP defines
static volatile bit CCP1 @ (unsigned)&PORTC*8+2; ;                  // CCP1 port pin
static volatile bit CCP2 @ (unsigned)&PORTC*8+1; ;                  // CCP2 port pin

#define CCP_OFF                         0
#define CCP_CAPTURE_FE                  4
#define CCP_CAPTURE_RE                  5
#define CCP_CAPTURE_DIV_4               6
#define CCP_CAPTURE_DIV_16              7
#define CCP_COMPARE_SET_ON_MATCH        8
#define CCP_COMPARE_CLR_ON_MATCH        9
#define CCP_COMPARE_INT                 0xa
#define CCP_COMPARE_RESET_TIMER         0xb
#define CCP_PWM                         0xc
#define CCP_PWM_PLUS_1                  0x1c
#define CCP_PWM_PLUS_2                  0x2c
#define CCP_PWM_PLUS_3                  0x3c

#define setup_ccp1(mode)                CCP1CON=mode
#define setup_ccp2(mode)                CCP2CON=mode
```

```
//#define set_ccp1(cvalue)             CCPR1L=cvalue
//#define set_ccp2(cvalue)             CCPR2L=cvalue
#define set_ccp1(cvalue) \
          CCPR1L=((cvalue)>>2)&0x00FF; \
          CCP1CON&=0xCF; \
          CCP1CON|=((cvalue)&0x03)<<4

#define set_ccp2(cvalue) \
          CCPR2L=((cvalue)>>2)&0x00FF; \
          CCP2CON&=0xCF; \
          CCP2CON|=((cvalue)&0x03)<<4

#define set_pwm1(cvalue)             set_ccp1(cvalue)
#define set_pwm2(cvalue)             set_ccp2(cvalue)

#define get_ccp1()                   CCPR1
#define get_ccp2()                   CCPR2

// Interrupt defines

#define GLOBAL                       0x80
#define RTCC_ZERO                    0x20
#define RB_CHANGE                    0x08
#define EXT_INT                      0x10
#define INT_TIMER0                   0x20
#define INT_TIMER1                   0x0100
#define INT_TIMER2                   0x0200
#define INT_CCP1                     0x0400
#define INT_CCP2                     0x10000
#define INT_SSP                      0x0800
#define INT_PSP                      0x8000
#define INT_RX                       0x2000
#define INT_TX                       0x1000
#define INT_CMP                      0x400000
#define ADC_DONE                     0x4000
#define EE_DONE                      0x100000
#define SER_BCL                      0x40000

#define enable_peripheral_interrupts() PEIE=1
#define enable_global_interrupts()   GIE=1
#define enable_interrupts(level) \
          INTCON|=(level)&0xff; \
          PIE1|=((level)>>8)&0xff; \
          PIE2|=(level)>>16
#define enable_rtcc_interrupt()      T0IE=1

#define disable_peripheral_interrupts() PEIE=0
#define disable_global_interrupts()  do GIE=0; while(GIE)
#define disable_interrupts(level) \
          INTCON&=~((level)&0xff); \
          PIE1&=~(((level)>>8)&0xff); \
          PIE2&=~((level)>>16)
#define disable_rtcc_interrupt()     T0IE=0

// USART defines

#define SER_MASTER                   0x80
#define SER_9BIT                     0x40
#define SER_TX_ENABLE                0x20
#define SER_SYNCHRONOUS              0x10
#define SER_HIGH_BAUD                4
#define SER_TSR_EMPTY                2
#define SER_ENABLE                   0x80
#define SER_RX_SGL                   0x20
#define SER_RX_CON                   0x10

#define kbhit()                      RCIF

#define set_lo_baud(baud)            SPBRG=(unsigned char)((2*(long)_CLOCK_/64+(long)baud)/2/ \
                                                           (long)baud-1);TXSTA &= ~SER_HIGH_BAUD
#define set_hi_baud(baud)            SPBRG=(unsigned char)((2*(long)_CLOCK_/16+(long)baud)/2/ \
                                                           (long)baud-1);TXSTA |= SER_HIGH_BAUD

#define setup_usart_async8_lo(baud) \
          TRISC |= 0xc0;\
          TXSTA=SER_MASTER|SER_TX_ENABLE|SER_TSR_EMPTY; \
          RCSTA=SER_ENABLE|SER_RX_CON; \
          set_lo_baud(baud)

#define setup_usart_async9_lo(baud) \
          TRISC |= 0xc0;\
          TXSTA=SER_MASTER|SER_TX_ENABLE|SER_TSR_EMPTY|SER_9BIT; \
          RCSTA=SER_ENABLE|SER_RX_CON|SER_9BIT; \
          set_lo_baud(baud)

#define setup_usart_async8_hi(baud) \
          TRISC |= 0xc0;\
          TXSTA=SER_MASTER|SER_TX_ENABLE|SER_TSR_EMPTY; \
          RCSTA=SER_ENABLE|SER_RX_CON; \
          set_hi_baud(baud)
```

```
#define setup_usart_async9_hi(baud) \
          TRISC |= 0xc0;\
          TXSTA=SER_MASTER|SER_TX_ENABLE|SER_TSR_EMPTY|SER_9BIT; \
          RCSTA=SER_ENABLE|SER_RX_CON|SER_9BIT; \
          set_hi_baud(baud)

static volatile bit txd       @ (unsigned)&PORTC*8+6; ;              // USART serial data out pin
static volatile bit rxd       @ (unsigned)&PORTC*8+7; ;              // USART serial data in pin

// SPI defines
#define SPI_MASTER            0x20
#define SPI_SLAVE             0x24
#define SPI_SS_DISABLED       0x25
#define SPI_L_TO_H            0
#define SPI_H_TO_L            0x10
#define SPI_CLK_DIV_4         0
#define SPI_CLK_DIV_16        1
#define SPI_CLK_DIV_64        2
#define SPI_CLK_T2            3
#define spi_enable(en)        SSPEN=!!en                             // enables the spi
#define spi_data_is_in()      BF                                     // tests the buffer full bit

static volatile bit sdo       @ (unsigned)&PORTC*8+5;                // serial data out pin
static volatile bit sdo_dir   @ (unsigned)&TRISC*8+5;                // serial data out direction bit
static volatile bit sdi       @ (unsigned)&PORTC*8+4;                // serial data in pin
static volatile bit sdi_dir   @ (unsigned)&TRISC*8+4;                // serial data in direction bit
static volatile bit sck       @ (unsigned)&PORTC*8+3;                // serial clock pin
static volatile bit sck_dir   @ (unsigned)&TRISC*8+3;                // serial clock direction bit
static volatile bit ss_dir    @ (unsigned)&TRISA*8+5;                // synchronous slave mode enable bit
static volatile bit BF        @ (unsigned)&SSPSTAT*8+0;              // spi buffer full bit

#endif // COMMDEFS_H
```

3.7. eeprom.c

```
/*

Including this file initialises the 256 EEPROM 8-bit bytes to the correct values.
Works on 16F876, see Hi-Tech C FAQ.

Created for PICuWEB - PIC micros and C
http://www.workingtex.com/htpic
(c)2001 Shane Tolmie

This file is in use for a commercial product, with approx. 7000 lines of C, and
copies of this code are currently running on 1100+ 16F876-based units worldwide.

*/

#include "motor.h"

//[user change]
#define PROFILE 0 //0 for normal setup
                  //1 for EEPROM filled with zeros

//[do not alter]
#if (PROFILE!=0) && (PROFILE!=1)
          #error eep_init 13
#endif

#asm
psect eedata,delta=2,abs,ovrld
          org 2100h
#endasm

#if (PROFILE==0)
#asm
  db STRAIGHT,AUX_DOUBLE,130,SR_50ms,RC_MIN,RC_MAX,RC_MIN,RC_MAX
  db RC_MIN,RC_MAX,RC_MIN,RC_MAX,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

```
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
#endasm
#endif

#if (PROFILE==1)    //nothing in EEPROM
#asm
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
#endasm
#endif
```

3.8. eeprom.h

```
#ifndef _EEPROM_H
#define _EEPROM_H

#define eeaddMixMode          0x00
#define eeaddAuxMode          0x01
#define eeaddDeadBand         0x02
#define eeaddSlewRate         0x03
#define eeaddRC1low           0x04
#define eeaddRC1high          0x05
#define eeaddRC2low           0x06
#define eeaddRC2high          0x07
#define eeaddRC3low           0x08
#define eeaddRC3high          0x09
#define eeaddRC4low           0x0A
#define eeaddRC4high          0x0B

#endif    // _EEPROM_H
```