TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UM PROTOCOLO DE COMUNICAÇÃO SOBRE UM BARRAMENTO RS-485 PARA DIVERSAS APLICAÇÕES

Roger Keidi Tsugami Saquisaka
Widê Assis de Souza

Brasília, Setembro de 2010

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UM PROTOCOLO DE COMUNICAÇÃO SOBRE UM BARRAMENTO RS-485 PARA DIVERSAS APLICAÇÕES

Roger Keidi Tsugami Saquisaka
Widê Assis de Souza

Relatório submetido como requisito parcial para obtenção
do grau de Engenheiro de Controle e Automação.

Banca Examinadora

Prof. Lélio Ribeiro Soares Júnior, UnB/ ENE (Orientador) _____

Prof. Luís Filomeno de Jesus Fernandes, UnB/ ENE _____

Prof. Flávio de Barros Vidal, UnB/ CIC _____

Brasília, Setembro de 2010

FICHA CATALOGRÁFICA

ROGER, SAQUISAKA
WIDÊ, SOUZA
Desenvolvimento de um protocolo de comunicação sobre um barramento RS-485 para diversas aplicações,

[Distrito Federal] 2010.

xii, 89p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2010). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Protocolo de comunicação
2. Barramento RS-485
3. Joystick
4. Entrada e saída de dados
5. Servomotor
6. Motor de passo

I. Mecatrônica/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

SAQUISAKA, R. K. T., SOUZA, W. A., (2010). Desenvolvimento de um protocolo de comunicação sobre um barramento RS-485 para diversas aplicações. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-nº 009/2010, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 89p.

CESSÃO DE DIREITOS

AUTORES: Roger Keidi Tsugami Saquisaka, Widê Assis de Souza.

TÍTULO DO TRABALHO DE GRADUAÇÃO: Desenvolvimento de um protocolo de comunicação sobre um barramento RS-485 para diversas aplicações.

GRAU: Engenheiro ANO: 2010

Roger Keidi Tsugami Saquisaka
QE 34 Conjunto R Casa 42 – Guará II
71065-182 Guará – DF – Brasil.

Widê Assis de Souza
Quadra 18 Conjunto F Casa 13
73050-186 Sobradinho – DF – Brasil.

AGRADECIMENTOS

Aos meus pais, Motokazu Saquisaka e Edna Seiko Saquisaka, por toda dedicação, apoio, carinho e confiança incondicionais que me concederam para que fosse possível alcançar esse objetivo.

Ao meu irmão, Rober, pela compreensão e apoio nos momentos mais difíceis durante esta longa jornada.

Ao meu tio, Marcos Hiroshi Tsugami, pelo exemplo de perseverança e incentivo incondicional aos meus estudos.

À minha avó, Hatsuchiyo Tsugami, que nunca deixou de acreditar em mim e sempre me tratou com carinho.

Ao professor Lélio, pela paciência, dedicação e atenção mostradas não só como orientador do projeto, mas também como pessoa e, sobretudo, por ter acreditado em mim.

Ao meu companheiro de projeto Widê, que se mostrou paciente e dedicado durante o desenvolvimento deste trabalho, além de sempre ter me incentivado a continuar apesar das dificuldades encontradas neste longo caminho.

Roger Keidi Tsugami Saquisaka

Agradeço a Deus por esta conquista, por guiar todos os meus passos e por ter sido a minha sustentação nestes anos de estudo.

Agradeço à minha família, especialmente aos meus pais, Viderval e Maria, que me deram seu carinho, empenharam seus esforços e me impulsionaram para chegar a essa etapa da vida.

Agradeço aos meus irmãos, Marcel e Juliana, pelo carinho e apoio que me deram e pela certeza de poder contar com eles.

Agradeço ao professor e orientador Lélio pelos preciosos ensinamentos, pela paciência e apoio sempre presentes na orientação desse projeto.

Agradeço ao meu companheiro de projeto, Roger, pela sua ajuda, amizade e por sua perseverança e dedicação na realização desse projeto.

Agradeço a todos os professores da UnB do Colegiado de Mecatrônica por seus ensinamentos e experiências que determinaram minha formação.

Widê Assis de Souza

RESUMO

O propósito do presente trabalho consiste no projeto e desenvolvimento de um protocolo de comunicação sobre um barramento que utiliza o padrão de comunicação serial RS-485 para o controle de diversas aplicações. Foram propostas quatro aplicações: o monitoramento do estado de um *joystick*, um dispositivo com função de entrada e saída de dados analógicos/digitais, o controle de um servomotor e o controle de um motor de passo.

O trabalho possui duas partes distintas: o *hardware* e o *software*. O *hardware* foi desenvolvido para atender as especificações do padrão RS-485 com a utilização de um microcontrolador ATmega8, fabricado pela Atmel, para cada aplicação. O *software* para o protocolo de comunicação serial foi baseado no protocolo do controlador de servo inteligente da Logosol e foi desenvolvido com a linguagem C.

Palavras Chave: protocolo de comunicação, barramento RS-485, *joystick*, entrada e saída de dados, servomotor, motor de passo.

ABSTRACT

This work consists in the project and development of a communication protocol on a bus that uses the RS-485 serial communication standard for the of control several applications. Four applications was proposed: the monitoring of a joystick status, a device with the function of input and output of analogic/digital data, the control of a servomotor and the control of a step motor.

This work has two distinct parts: the hardware and the software. The hardware was developed to attend the specifications of RS-485 standard with the use of a microcontroller ATmega8, manufactured by Atmel, for each application. The software for the communication protocol was based on the Logosol Intelligent Servo Drive protocol and was developed with the C language.

Keywords: communication protocol; RS-485 bus; *joystick*; data input and output; servomotor; stepper motor.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Latinos

f_c	Freqüência de corte	[Hz]
S_f	Posição ao término do deslocamento	[passos]
S_0	Posição de início do deslocamento	[passos]
S_a	Posição de término da aceleração	[passos]
S_d	Posição de início da desaceleração	[passos]
v_0	Velocidade inicial	[passos/s]
v	Variável de velocidade	[passos/s]
a	Aceleração	[passos/s^2]
t_f	Instante de tempo ao final do deslocamento	[s]
t_a	Instante de tempo ao témino da aceleração	[s]
t_d	Instante de tempo ao início da aceleração	[s]

Símbolos Gregos

σ	Desvio Padrão
Ω	Ohm

Sobrescritos

‾	Valor Complementar

Siglas

API	Application Programming Interface – Interface de Programação de Aplicativo
AC	Alternating Current – Corrente Alternada
A/D	Analogic/Digital – Analógico/Digital
bps	Bits por Segundo
D/A	Digital/Analogic – Digital/Analógico
DC	Direct Current – Corrente Contínua
EEPROM	Electrically-Erasable Programmable Read-Only Memory – Memória Apenas para Leitura, Programável e Eletricamente Apagável.
GND	Ground – Valor de Referência Comum, Terra
LARA	Laboratório de Robótica e Automação
Mbps	Megabits por Segundo
MIPS	Milhões de Instruções por Segundo

NRZ	Non Return to Zero – Sem Retorno ao Nível Zero
OSI	Open Systems Interconnection – Interconexão de Sistemas Abertos
PC	Personal Computer – Computador Pessoal (Microcomputador)
PWM	Pulse Width Modulation – Modulação por Largura de Pulso
RAM	Random Access Memory – Memória de Acesso Aleatório
RISC	Reduced Instruction Set Computer – Computador com Conjunto Reduzido de Instruções
TTL	Transistor-Transistor Logic – Lógica Transistor-Transistor
UART	Universal Asynchronous Receiver Transmitter – Receptor Transmissor Assíncrono Universal
USART	Universal Synchronous-Asynchronous Receiver Transmitter – Receptor Transmissor Síncrono-Assíncrono Universal
USB	Universal Serial Bus – Barramento Serial Universal

CAPÍTULO 1 - INTRODUÇÃO

O laboratório LARA, na Universidade de Brasília, realiza pesquisas de sistemas de controle e visão computacional. O braço robótico Rhino é um bom exemplo dos equipamentos disponíveis no laboratório para o desenvolvimento das mesmas.

Os motores do braço robótico são comandados por um sistema de controle fornecido pela Jeffrey Kerr. Tal sistema, apesar de ser funcional, é obsoleto e, principalmente, não permite alterações, pois possui arquitetura fechada. Assim, as possibilidades de se explorar o manipulador Rhino são limitadas para fins de projeto e pesquisa.

Devido à essa limitação referente ao braço robótico Rhino e à possibilidade de grande utilidade para o laboratório no que se refere a futuros projetos, este trabalho, em que propõe o desenvolvimento de um protocolo de comunicação sobre um barramento RS-485 para diversas aplicações, foi desenvolvido. Entre elas seria a de controlar os motores do braço robótico, por exemplo. O protocolo a ser implementado deve possibilitar a comunicação entre o sistema mestre (no caso o computador) e os escravos (outros dispositivos). O mestre deve ser capaz de enviar parâmetros de configuração, comandos e de receber informações dos dispositivos. A Figura 1.1 ilustra o sistema proposto.

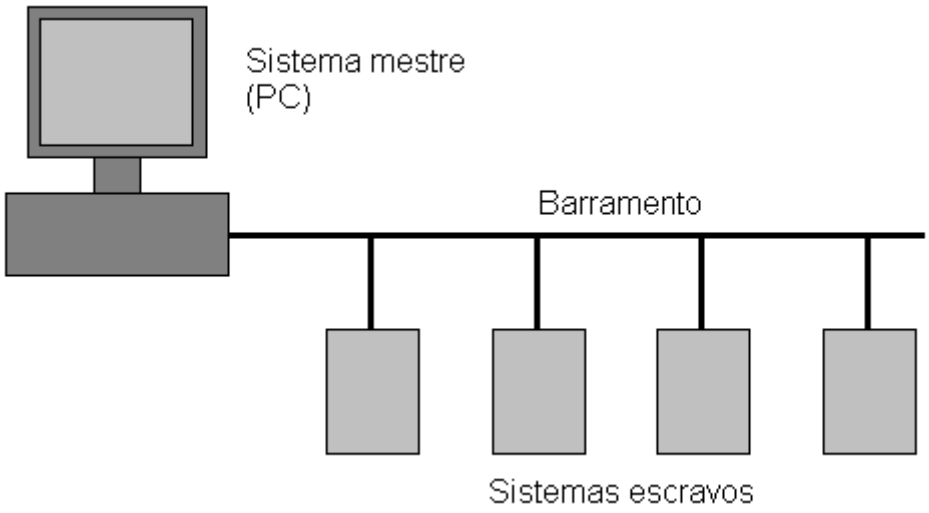


Figura 1.1. Ilustração do sistema a ser implementado.

Primeiramente, o padrão de comunicação serial RS-485 foi escolhido para o barramento devido à sua maior robustez com relação ao ruído e a possibilidade de se conectar vários dispositivos simultaneamente no mesmo barramento, em detrimento do padrão RS-232.

Depois, foi feita a escolha do modelo de microcontrolador a ser utilizado para executar o *software* específico para cada aplicação. O modelo escolhido foi o ATmega8 pelo baixo custo e por atender todas os requisitos exigidos pelo projeto. Além disso, ele foi utilizado em projetos anteriores no mesmo laboratório em que este trabalho foi desenvolvido e o material de leitura para o manuseio do microcontrolador em questão (gravadora, circuito de referência) foi escrito por um professor do LARA e alguns de seus alunos (BORGES *et al.*, 2008).

Em seguida, foi definido que o protocolo a ser desenvolvido seria baseado no utilizado pelo controlador de servomotores LS-174 da Logosol (LOGOSOL, 2002). Foram necessárias adaptações, uma vez que o protocolo da Logosol é específico para o controle de servomotores DC/AC. O *software* do sistema mestre utilizado para a implementação do protocolo teve como base o código do sistema PIC- SERVO da Jeffrey Kerr, LLC, utilizado nas placas de controle dos servomotores do Rhino, adaptado pelo professor Lélio Ribeiro Soares (orientador deste trabalho).

Quanto às aplicações, foram escolhidos quatro dispositivos como sistemas escravos: placa de monitoramento do estado (posicionamento do manche e o acionamento dos botões) do *joystick*, uma placa com entrada e saída de dados analógicos/digitais, uma placa de controle de servomotor e uma placa de controle de motor de passo. Para cada aplicação, foi desenvolvido um *software* embarcado específico, com base no trabalho desenvolvido por Thiago Felippe Kurudez Cordeiro para o controle de um motor DC (CORDEIRO, 2009), que utiliza o mesmo barramento.

O capítulo 2 descreve os conceitos envolvidos no trabalho como as características principais do microcontrolador ATmega8, o padrão RS-485, comparação com o padrão RS-232, o protocolo utilizado e os comandos associados, as características e o funcionamento de cada dispositivo das aplicações utilizadas.

O capítulo 3 cobre o projeto e o desenvolvimento da placa de cada aplicação e o seu *software* específico, explicando os detalhes do *hardware*, as funções envolvidas na comunicação serial pelo *software* e a interpretação de comandos enviados pelo sistema mestre.

O capítulo 4 apresenta os resultados obtidos do sistema desenvolvido. Foram feitos vários testes, como a resposta a vários comandos para cada placa e os problemas encontrados.

O capítulo 5 traz as considerações finais e sugestões para trabalhos futuros a partir dos resultados obtidos neste projeto.

CAPÍTULO 2 - CONCEITOS

Uma questão importante é a escolha do microcontrolador a ser utilizado. O modelo adotado definirá a capacidade de processamento e a quantidade de memória disponível para o *software*. A escolha do modelo também definirá a forma como ocorrerá a interação com o barramento e a obtenção de informações analógicas ou digitais.

A próxima escolha a ser feita é com relação ao barramento de dados a ser utilizado para a comunicação entre os dispositivos e entre um dispositivo e o sistema mestre (no caso um computador). Um critério a ser utilizado para tal escolha é a compatibilidade do barramento com alguma forma de comunicação disponível no microcontrolador escolhido, devendo permitir que vários dispositivos sejam conectados simultaneamente no mesmo barramento. Além disso, deve ser compatível com algum barramento existente no sistema mestre.

Por fim, um protocolo deve ser desenvolvido para que fique definido como se dará a comunicação entre os dispositivos do barramento. Detalhes como estrutura dos dados enviados, como cada dispositivo será endereçado, o limite de dispositivos controlados por um mesmo mestre e a implementação de algum algoritmo de detecção de erros de transmissão devem ser definidos para que a comunicação ocorra sem problemas.

2.1 MICROCONTROLADOR ATMEL ATMEGA8

A razão para se utilizar um microcontrolador é a sua característica de integrar várias funcionalidades em um único circuito integrado. Isso possibilita a simplificação do *hardware*, minimizando as chances de ocorrerem problemas e, principalmente, a redução de custo, quando comparado com a situação de se utilizarem dispositivos discretos mais básicos para atender às funcionalidades requeridas pelo projeto.

O microcontrolador ATmega8, produzido pela ATMEL, foi escolhido para ser utilizado neste trabalho. Uma das razões é o seu custo-benefício, uma vez que esse modelo atende às necessidades para a implementação da proposta desenvolvida neste trabalho. Algumas das diversas funcionalidades integradas (ATMEL, 2010) ao microcontrolador que merecem destaque para o propósito deste trabalho são:

- Um conversor A/D com seis entradas multiplexadas;
- 3 canais PWM que podem ser utilizados como conversores D/A;
- Oscilador interno ajustável entre 1MHz e 8 MHz, com opções de utilizar um cristal externo ou um circuito oscilador externo;

- Memória flash interna programável de 8 Kbytes, com função de memória de programa. É possível reprogramá-la cerca de dez mil vezes;
- Memória RAM interna de 1 Kbyte;
- Memória EEPROM de 512 bytes, para armazenar dados.

Vale ressaltar também as características de desempenho (ATMEL, 2010), consideradas para a escolha do modelo de microcontrolador a ser utilizado:

- Arquitetura RISC de 8 bits;
- A maioria das 130 instruções em assembly necessita de um ciclo de *clock* para a sua execução. Considerando o pipeline, o *throughput* pode atingir até 16 MIPS com a utilização de um cristal de 16 MHz;
- Multiplicação em *hardware*, realizada em dois ciclos de *clock*;
- 32 registradores de uso geral de 8 bits;
- Dois contadores/temporizadores de 8 bits e um de 16 bits;
- Fontes de interrupções internas e externas;
- Lógica de comunicação serial síncrona e assíncrona (USART) em hardware, o que reduz o custo de processamento.

Quanto às características elétricas (ATMEL, 2010):

- Tensão de operação: 4,5 – 5,5 V (ATmega8) / 2,7 – 5,5 V (ATmega8L);
- Consumo de corrente (4MHz, 3 V, 25 ºC): 3,6 mA (modo ativo), 1,0 mA (modo inativo) e 0,5 µA (desligado).

2.2 BARRAMENTO

O barramento é o meio físico pelo qual a informação trafega entre dois ou mais dispositivos. A escolha do barramento leva em conta vários aspectos da transmissão, como a taxa em que os bits são transmitidos e a susceptibilidade a ruídos.

Muitos projetos de sistemas baseados em microprocessadores requerem comunicação entre dispositivos. Esta comunicação pode ser efetuada de forma paralela ou serial, desde que os dispositivos possuam meios para isto (MARTINS e BORGES, 2006). No caso específico de comunicação serial, o dispositivo deve ser dotado internamente de uma UART (Universal Asynchronous Receiver Transmitter), USART (Universal Synchronous-Asynchronous Receiver Transmitter) ou qualquer controlador de comunicação

serial (MARTINS e BORGES, 2006). Estes controladores possuem pinos de entrada/saída pelos quais ocorre o fluxo e controle de informações.

“Em geral, os níveis de tensão nos pinos são baixos, de acordo com a alimentação. Se a distância entre dispositivos for pequena, da ordem de algumas dezenas de centímetros, e o nível de interferência eletromagnética for baixo, pode-se fazer a conexão direta entre os pinos dos dispositivos (em geral ponto-a-ponto). Entretanto, quando se deseja realizar a comunicação para distâncias mais longas, faz-se necessário empregar um padrão físico de comunicação serial. Um destes padrões é o RS-232, que é mais comumente utilizado para a comunicação serial para curtas distâncias, a baixas velocidades de comunicação. Por outro lado, o padrão RS-485 é capaz de prover uma forma bastante robusta de comunicação multiponto, bastante comum na indústria, em controle de sistemas com taxas podendo ultrapassar 10 Mbps, e distâncias podendo alcançar centenas de metros.” (MARTINS e BORGES, 2006).

O microcontrolador ATmega8 possui USART implementada via *hardware*. A USART faz parte da camada de enlace, que é a camada superior à camada física no modelo OSI (TANENBAUM, 2003). Ao escolher a USART como forma da camada de enlace, limita-se a escolha de barramento a algumas opções compatíveis.

2.2.1 RS-232 (EIA-232)

RS é uma abreviação de “Recommended Standard”. Ela relata uma padronização de uma interface comum para comunicação de dados entre equipamentos, criada no início dos anos 60, por um comitê conhecido como “Electronic Industries Association” (EIA) (CANZIAN, 2009), que, desde 1997, passou a se chamar “Electronic Industries Alliance”.

O padrão RS-232 é referenciado ao GND comum (0V). O nível lógico 1 (marca) é interpretado como sendo qualquer tensão no intervalo entre -25 V e -3 V, enquanto que tensões no intervalo entre 3 V e 25 V correspondem ao nível lógico 0 (espaço). Tensões entre -3 V e 3 V não possuem nível lógico definido, sendo considerado uma região de transição. A Figura 2.1 ilustra essas faixas de valores de tensão.

Este tipo de interface é útil em comunicações ponto-a-ponto a baixas velocidades de transmissão, mas, devido à grande faixa de variação dos sinais, faz-se necessário dispor de drivers de alto *slew rate* para se alcançar altas velocidades de comunicação (MARTINS e BORGES, 2006). Deve-se observar que, com o aumento do comprimento do cabo de comunicação, o padrão RS-232 se torna altamente susceptível a interferência eletromagnética e a retorno de sinal.

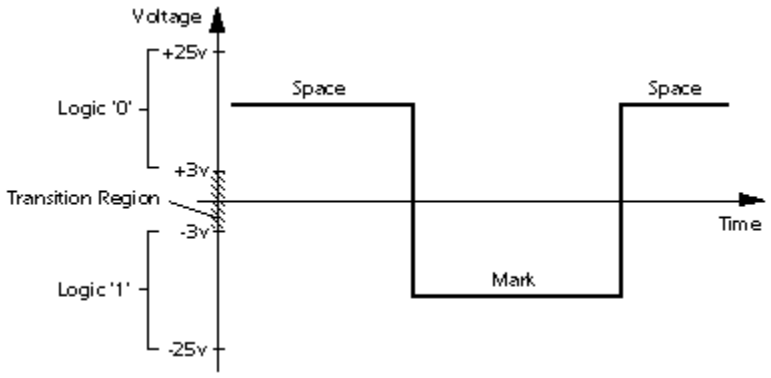


Figura 2.1. Faixas de valores para os níveis lógicos 0 e 1 no padrão RS-232 (CANZIAN, 2009).

Os sinais do padrão RS-232 (RxD e TxD) oscilam entre valores positivos e negativos (MAXIM INNOVATION DELIVERED, 2000), em relação ao GND comum. O dado do RS-232, quando comparado ao TTL/CMOS, possui polaridade invertida. Entretanto, ao converter de TTL/CMOS para RS-232 e, então de volta ao TTL/CMOS, o dado volta à polaridade normal (MAXIM INNOVATION DELIVERED, 2000).

Tabela 2.1. Sumário das principais especificações elétricas do padrão RS-232 para um transceptor MAX3225E (adptado de MAXIM INNOVATION DELIVERED, 2000).

Parâmetro	Condição	Mín.	Máx.	Unidade
Tensão de Saída do Driver, em Circuito Aberto			25	V
Tensão de Saída do Driver, com Carga	3 kΩ < R_L < 7 kΩ	±5	±15	V
Resistência de Saída do Driver, Desligado	-2V < V < 2V		300	Ω
Slew Rate		4	30	V/µs
Máxima Capacitância da Carga			2500	pF
Resistência de Entrada do Receptor		3	7	kΩ
Saída = Marca (1 Lógico)		-3		V
Saída = Espaço (0 Lógico)			3	V

Transmissões RS-232 típicas raramente excedem 30 metros por duas razões. A primeira é a diferença de valores entre a transmissão e a recepção é baixa, e permite

apenas uma pequena faixa de rejeição de tensão modo comum (MAXIM INNOVATION DELIVERED, 2000). A segunda é o fato da capacitância distribuída de um cabo mais longo pode afetar o *slew rate* (a transição entre dois valores de tensão se torna mais lenta) ao ultrapassar o valor máximo permitido pelo padrão (MAXIM INNOVATION DELIVERED, 2000). Como o RS-232 foi designado para conexões ponto-a-ponto ao invés de multiponto, esta suporta apenas cargas simples de 3 kΩ até 7 kΩ (MAXIM INNOVATION DELIVERED, 2000).

A tabela 2.1 mostra o sumário das principais especificações elétricas do RS-232.

2.2.2 RS-485

O padrão RS-485 utiliza um princípio diferente ao RS-232. Um transceptor RS-485 traduz um sinal lógico TTL em dois sinais, denominados de A e B, (ver Figura 2.2). O sinal A possui a mesma lógica do sinal TTL, enquanto que o sinal B é complementar. A informação do sinal de entrada está codificada na forma do sinal A-B, ou seja, da diferença entre os sinais A e B. Se esta diferença for superior a 200 mV, então se tem o nível lógico 1. Caso a diferença seja inferior a -200 mV, então se considera o nível lógico 0. No intervalo de -200 mV a 200 mV, o nível lógico é indefinido, servindo também como meio de detecção de cabo solto para alguns transceptores comerciais. Entretanto, deve-se ter algum cuidado com relação ao compartilhamento de referência de 0 V entre os dispositivos (MARTINS e BORGES, 2006).

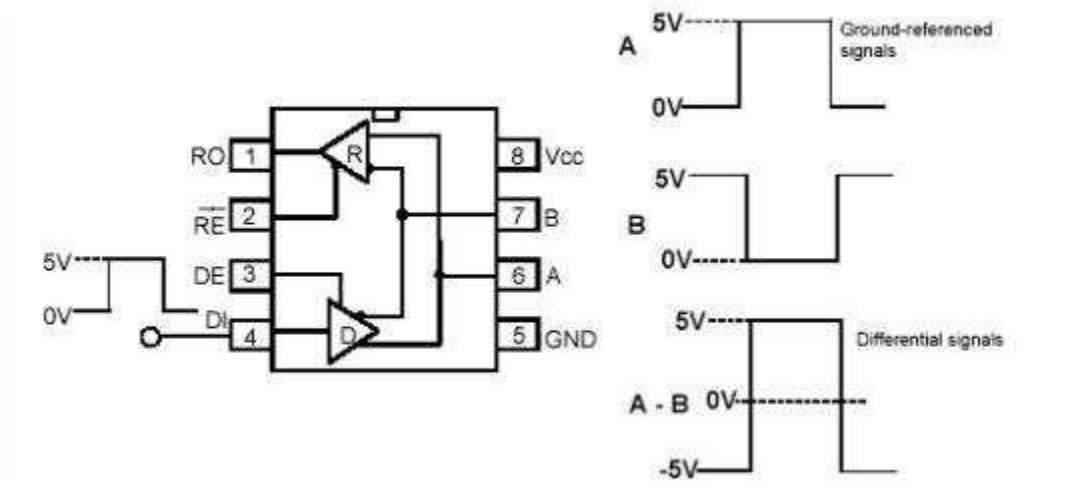


Figura 2.2. Diagrama dos transceptores DS485 e ST485, e sinais diferenciais (MARTINS e BORGES, 2006).

Segundo Martins e Borges (2006), uma das vantagens da transmissão diferencial é a sua robustez a interferência eletromagnética. A conexão entre dispositivos RS-485 é feita por cabos de par trançado, com resistores de terminação para balanceamento. Dessa forma, se um ruído é introduzido na linha, ele é induzido nos dois fios de modo que a diferença entre A e B é quase nula. Outra vantagem da transmissão diferencial é que diferentes potenciais de terra são, até certo ponto, ignorados pelos transmissores e receptores. Isso se torna importante quando se tem que percorrer grandes distâncias ou mesmo em sistemas com 2 diferentes fontes de alimentação. Cabos trançados com terminações corretas que minimizam reflexão do sinal permitem taxas de transferência de 10 Mbps a distâncias de até 1 km (MARTINS e BORGES, 2006).

O RS-485 é um padrão de comunicação multiponto, permitindo a conexão de até 32 dispositivos em um simples cabo de par trançado. Dependendo do transceptor, pode-se conectar mais de 200 dispositivos, seguindo a topologia mostrada na Figura 2.3, que é a arquitetura mais comum de utilização do RS-485.

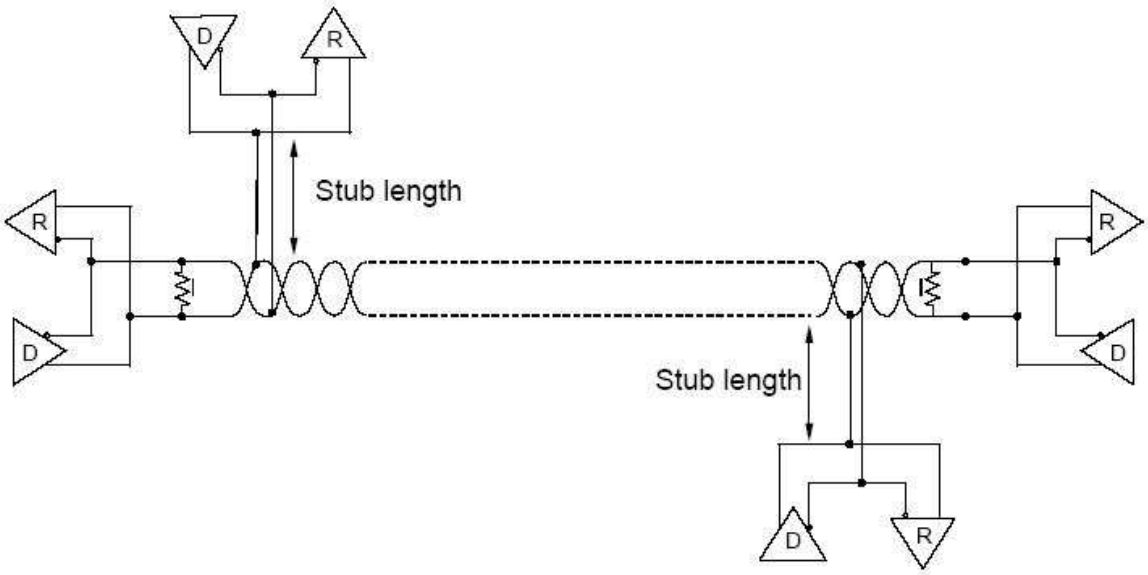


Figura 2.3. Barramento RS-485 típico (MARTINS e BORGES, 2006).

As letras D e R indicam os drivers de transmissão e recepção de cada dispositivo, respectivamente. Uma observação a ser considerada é a impedância característica do cabo. Se o cabo for utilizado para transmissões com componentes espectrais e altas freqüências, podem ocorrer reflexões do sinal em sua extremidade, provocando inconsistência nos dados transmitidos. Para minimizar este efeito, deve-se adicionar resistores de terminação de valores iguais à impedância característica do cabo para que ele se comporte como um cabo infinito (MARTINS e BORGES, 2006). Os valores típicos para essa resistência é de cerca de

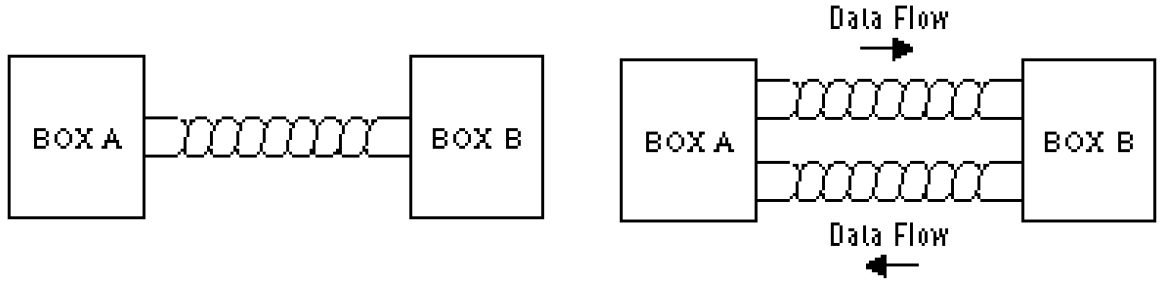
120 Ω para cabos trançados e de 54 Ω para cabos blindados. Apesar de típicos, estes valores podem ser diferentes, pois dependem também dos requisitos de carga mínima dos transmissores. É importante também que haja apenas dois resistores, um em cada extremidade, que podem estar também dentro do último dispositivo conectado. Na prática, porém, para pequenas distâncias e baixas velocidades, a terminação não chega a ser algo crucial e a maioria dos circuitos funciona bem. Um outro artifício para minimizar a influência da reflexão dos sinais é por meio da redução forçada da banda passante. Isto já é feito em vários transceptores, por meio de uma limitação do *slew-rate*. Um sumário das principais especificações elétricas pode ser visto na Tabela 2.2.

Tabela 2.2. Sumário das principais especificações elétricas do padrão RS-485 (MAXIM INNOVATION DELIVERED, 2000).

Parâmetro	Condição	Mín.	Máx.	Unidade
Tensão de Saída do Driver, em Circuito Aberto		1.5 -1.5	6 -6	V V
Tensão de Saída do Driver, com Carga	R_L=100 Ω	1.5 -1.5	5 5	V V
Corrente de Curto-Circuito de Saída do Driver	Por saída para o comum		±250	mA
Tempo de Subida da Saída do Driver	R_L=54 Ω C_L=50 pF		30	% da largura do bit
Tensão de Modo Comum do Driver	R_L=54 Ω		±3	V
Sensibilidade do Receptor	-7 V < V_{MC} < 12 V		±200	mV
Faixa de Tensão de Modo Comum do Receptor		-7	12	V
Resistência de Entrada do Receptor		12		kΩ

A maioria dos sistemas com RS-485 utiliza uma arquitetura mestre/escravo para comunicação, na qual apenas um único dispositivo (geralmente o PC), chamado mestre, envia periodicamente mensagens endereçadas aos escravos. Cada escravo, por sua vez, tem um único endereço e responde apenas a pacotes endereçados a ele. Além do mais, pode-se usar USARTS *half-duplex* ou *full-duplex* como ilustrado pela Figura 2.4. Para tanto, fisicamente as seguintes conexões devem ser feitas:

- *Half-duplex*: utiliza-se um único par trançado, em que todos os dispositivos estão conectados no mesmo cabo trançado. Dessa forma, todos eles devem possuir transceptores com saídas tri-state, incluindo o mestre. A comunicação se dá em ambas direções e é importante evitar por *software* que mais de um dispositivo tenha o seu driver da transmissão habilitado ao mesmo tempo.
- *Full-duplex*: utilizam-se dois pares trançados, em que os escravos transmitem para o mestre por meio do segundo par trançado. Essa solução geralmente permite comunicação multiponto em sistemas que foram originalmente projetados para RS-232 com pequenas modificações no *software* mestre. Outro fato é que o mestre também não precisa colocar a saída do seu transceptor em estado de alta impedância.



(a) half-duplex (b) full-duplex

Figura 2.4. Conexões físicas RS-485 para (a) half-duplex e (b) full-duplex (MARTINS e BORGES, 2006).

2.2.3 TRANSCEPTORES RS-485

No mercado, há uma grande variedade de transceptores RS-485 produzidos por diferentes fabricantes, sendo possível encontrar até circuitos prontos para determinadas aplicações. Cada modelo possui características distintas como taxas de comunicação, o número de dispositivos conectados ao barramento suportado, capacidade de realizar comunicação *half-duplex* ou *full-duplex* e opto-isolamento. A escolha de qual modelo utilizar vai depender dos requisitos exigidos pelo projeto.

A Figura 2.2 mostra o diagrama dos circuitos integrados DS485 e ST485. A diferença básica entre eles está somente nas suas características elétricas, bem como de taxas de velocidade máxima: até 2,5 Mbps para o DS485 e até 30 Mbps para o ST485. O pino RE habilita o driver de recepção R, sendo ativo no nível 0. O pino DE habilita o driver de transmissão D (ativo em 1). Normalmente esses dois pinos são conectados juntos de forma

que o transceptor esteja apenas recebendo ou transmitindo. Os pinos RO e DI representam a saída da recepção e o driver de entrada, respectivamente, e trabalham com níveis lógicos TTL (0 a 5 V). Já as saídas A e B para o barramento operam com tensão diferencial entre seus terminais. O transceptor normalmente é alimentado com 5 V DC.

Para que um dispositivo transmita um dado pelo barramento, é necessário que se aplique 5 V ao pino DE, fazendo com que RE seja desabilitado, para então transmitir a informação necessária pelo pino DI. Com o final da transmissão, deve-se desabilitar DE e reabilitar RE, de forma que o transceptor volte ao modo de recepção. Esta habilitação pode ser feita via *software*, controlando pinos de um microcontrolador ou de uma porta de comunicação de um microcomputador, ou mesmo por meio de um *hardware* construído especialmente para detectar o início de transmissão para o formato de dados utilizado (por exemplo, UART).

De acordo com Martins e Borges (2006), um problema possível de ocorrer com um barramento RS-485 que dispõe apenas de resistores de terminação é que, quando todos os dispositivos estão em modo de recepção, o nível lógico do barramento pode ficar indefinido. Para garantir que o barramento fique sempre em nível lógico 1, que corresponde ao estado *default* NRZ das USARTS conectadas ao barramento, deve-se adicionar um resistor de *pull-up* ao pino A e um de *pull-down* no pino B, conforme ilustra a Figura 2.5. Esses resistores devem ter valores iguais para não alterarem o balanceamento da linha de transmissão.

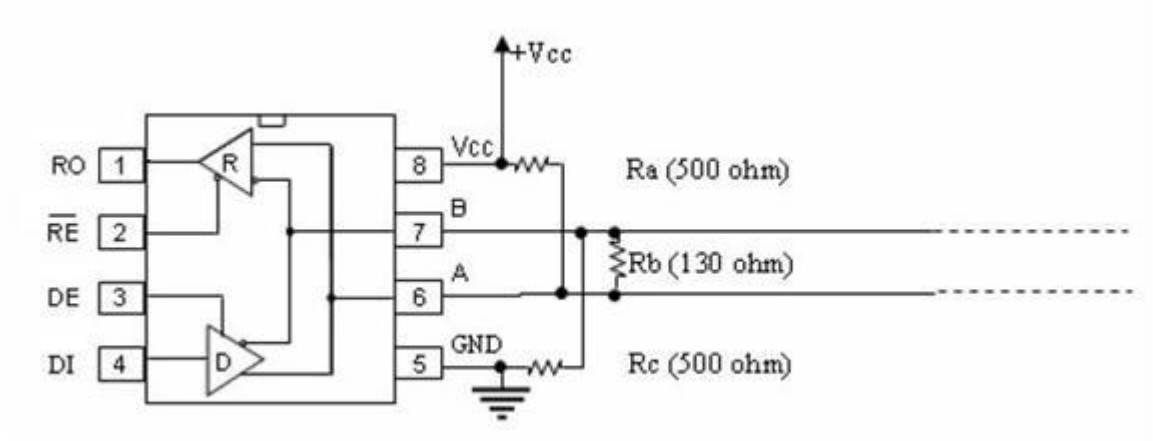


Figura 2.5. Pull-up e pull-down para evitar sinais indefinidos no barramento (MARTINS e BORGES, 2006).

2.2.4 COMPARAÇÃO E ESCOLHA DO BARRAMENTO

Avaliadas as características de ambos os padrões, nota-se que cada uma possui vantagens e desvantagens.

O padrão RS-232 se destaca por ser uma forma comum de um computador pessoal se comunicar com outros dispositivos, embora esteja caindo em desuso devido à popularização da porta USB. Entretanto, permite apenas transmissões a curta distância, é mais suscetível a ruídos do que os outros padrões analisados e, sendo usado da maneira que é especificado, permite apenas a comunicação entre dois dispositivos.

O padrão RS-485 não é utilizado em computadores pessoais. Entretanto, possui melhor desempenho que o RS-232, quando comparado em distância máxima de transmissão e resistência a ruídos. Além disso, permitem a comunicação entre vários dispositivos usando apenas um barramento.

Considerando que muitos sistemas (por exemplo, motores) são controlados em ambiente industrial, que possuem mais ruído e que nem sempre é possível manter um computador perto do maquinário, decidiu-se pelo padrão RS-485. A questão da comunicação com o PC pode ser facilmente resolvida por meio de um conversor RS-232 para RS-485. Desta forma, a conexão RS-232 de um computador se comunica com um sistema RS-485, aproveitando a primeira que já se encontra disponível.

2.3 PROTOCOLO

O protocolo de comunicação a ser desenvolvido se baseia no utilizado pelo controlador de servomotores LS-174 da Logosol (LOGOSOL, 2002). Este permite o controle de até 31 controladores inteligentes sobre um barramento multiponto full-duplex RS-485 em um ambiente de controle de movimento distribuído (LOGOSOL, 2002). No entanto, o protocolo será desenvolvido para várias aplicações, não se limitando a controladores apenas.

A comunicação serial com o controladores do LS-174 é realizada por meio de um protocolo *full-duplex* de 4 fios e 8 bits assíncronos, com um bit de início seguido de 8 bits de dados (bit menos significativo primeiro) e um bit de parada (LOGOSL, 2002).

O protocolo de comando é estritamente um protocolo mestre/escravo, no qual o mestre, nomalmente um computador, envia um pacote de comando para um controlador escravo específico ou a um grupo de controladores. Os dados são armazenados em um *buffer* até o fim do ciclo de servo (0.512 ms no máximo) e então o comando é executado (LOGOSOL, 2002). O controlador ou o líder (no caso de um grupo) então retorna um pacote de estado.

2.3.1 ENDEREÇAMENTO

Quando existem vários controladores conectados em um mesmo barramento, é imprescindível que cada um deles possua um endereço individual para que se possa enviar um comando a ele. Existem dois pinos, ADDR_IN e ADDR_OUT que fazem parte do processo de endereçamento. Para isto, liga-se o ADDR_OUT de um controlador no ADD_IN do próximo (configuração chamada de *daisy-chain*) (adaptado de LOGOSOL, 2002). Deixa-se um controlador com o ADDR_IN ligado ao GND e um ADDR_OUT em aberto.

Ao serem iniciados, todos os controladores começam com o endereço individual 0x00 e com o pino ADD_OUT em nível alto. Entretanto, se um controlador estiver com o pino ADDR_IN em nível alto, irá ignorar qualquer dado enviado pelo mestre.

Como citado anteriormente, um dos controladores está com o pino ADDR_IN conectado ao GND, ou seja, está em nível baixo, enquanto os outros pinos ADDR_IN estão em nível alto por estarem conectados aos pinos ADDR_OUT dos outros controladores que estão em nível alto. O mestre envia então um comando para definir o endereço individual e apenas este controlador responderá.

Depois de ter o endereço alterado, o respectivo controlador tem o seu pino ADDR_OUT levado para o nível baixo e o controlador seguinte irá responder ao endereço 0x00, tendo o seu endereço definido pelo mestre. O processo continua até que todos os controladores possuam um endereço individual distinto.

Dessa forma, outros controladores podem ser adicionados ao barramento RS-485 sem haver a necessidade de mudança no hardware inicial (LOGOSOL, 2002).

2.3.2 COMANDOS INDIVIDUAIS E DE GRUPO

Em complemento ao endereço individual, cada controlador tem um endereço secundário chamado de endereço de grupo. O endereço individual pode assumir um valor entre 0 e 255 (0x00 a 0xFF). Já o endereço de grupo só pode ser definido entre 128 e 255 (0x80 a 0xFF).

O endereço de grupo é útil para enviar comandos que devem ser executados simultaneamente por vários controladores (por exemplo, iniciar movimento, definir *baud rate*, etc.) (LOGOSOL, 2002).

“Quando um controlador recebe um comando enviado para o seu endereço de grupo, ele executará o comando, mas não retorna um pacote de estado. Isso previne colisão de dados na linha compartilhada de resposta. Entretanto, quando se está definindo um endereço de grupo, o mestre pode especificar um membro do grupo como o “líder do grupo”. O líder do grupo irá retornar um pacote de estado como faria para um comando enviado ao

seu endereço individual. O endereço de grupo é definido ao mesmo tempo do endereço individual pelo mesmo comando.” (LOGOSOL, 2002).

2.3.3 COMUNICAÇÃO – MESTRE

O mestre não deve enviar nenhum outro comando até o pacote de estado ter sido recebido para se assegurar que ele não sobrescreva quaisquer dados de comando ainda em uso.

Cada pacote de comando consiste no seguinte (LOGOSOL, 2002):

- Byte de cabeçalho (definido como 0xAA)
- Byte de endereço – individual ou de grupo (0x00 – 0xFF)
- Byte de comando
- 0 – 15 bytes de dados
- Byte de *checksum*

Todo comando enviado pelo mestre começa com o cabeçalho 0xAA. Observando sua forma binária (0b10101010), percebe-se a alternância dos valores 0 e 1. A probabilidade dessa seqüência ser gerada por ruído é muito baixa, minimizando a ocorrência de falsos inícios de comando.

O byte de comando é dividido em nibbles alto e baixo: o nibble baixo (4 bits menos significativos) é o valor do comando; o nibble alto (4 bits mais significativos) é o número de bytes adicionais de dados, o qual seguirá o código do comando. O byte de *cheksum* é a soma de 8 bits do byte de endereço, do byte de comando e dos bytes de dados.

2.3.4 COMANDOS

A Tabela 2.3 lista os comandos definidos no protocolo, traz a quantidade de dados exigidos e uma descrição sucinta da função de cada comando. Na coluna do nome do comando, também é informado o mnemônico do mesmo que será utilizado no *software* desenvolvido neste trabalho.

Além disso, estão especificados quais são os comandos gerais e os específicos de cada módulo ou dispositivo. Os comandos gerais podem ser aproveitados em projetos futuros, assim como os comandos específicos cujo(s) o(s) dispositivo(s) estiver(em) envolvido(s) no projeto em questão.

Tabela 2.3. Lista de comandos do protocolo (adaptado de Logosol, 2002).

Comando (mnemônico)	Módulo	Código do comando	# de bytes de dados	Descrição
Define endereço (setAddress)	Geral	0x1	2	Define os endereços individuais e de grupo
Define estado (defineStatus)	Geral	0x2	1	Define qual dado deve ser retornado em cada pacote de estado
Lê estado (readStatus)	Geral	0x3	1	Faz um dado de estado particular ser retornado apenas uma vez
Lê joystick (readJoystick)	Joystick	0x4	1	Lê o estado do joystick
Configura entrada/saída (configIO)	Entrada e saída AD	0x4	1	Configura os 8 pinos como entrada/saída de dados
Lê dados (readIO)	Entrada e saída AD	0x5	1	Lê os dados nos pinos de entrada
Escreve dados (writeIO)	Entrada e saída AD	0x6	1	Escreve dados nos pinos de saída
Lê ADC (readADC)	Entrada e saída AD	0x7	1	Lê os dados convertidos dos canais do conversor AD
Configura saída analógica (configDAC)	Entrada e saída AD	0x8	0	Configura os contadores para gerar um sinal PWM
Gera um sinal analógico (writeCDA)	Entrada e saída AD	0x9	2 ou 3	Gera um sinal analógico por meio de PWM em 1 ou 2 canais
Configura os servos (configServo)	Servo	0x4	2	Configura as portas que irão acionar os servos
Define a posição de um servo (setPosInd)	Servo	0x5	2	Define a posição de um servo individual
Define a posição dos servos (setPosAll)	Servo	0x6	1	Define a posição de todos os servos
Pára o motor de passo (stopMotor)	Motor de passo	0x4	1	Interrompe o movimento do motor de passo
Inicia motor de passo (startMotor)	Motor de passo	0x5	2	Inicia o movimento do motor de passo
Define o movimento do motor de passo (deslocMotor)	Motor de passo	0x6	4	Define o número de passos do motor e a sua velocidade
Ajusta a velocidade do motor (velocMotor)	Motor de passo	0x7	2	Ajusta a velocidade do motor em um sentido
Modo trapezoidal (trapezoidal)	Motor de passo	0x8	6	Executa o modo trapezoidal do motor de passo

Define baud rate (setBaudRate)	Geral	0xA	1	Define o baud rate (apenas comando de grupo)
Apaga bits (clearBits)	Geral	0xB	0	Apaga todos os bits de status com memória
NoOp (noOp)	Geral	0xE	0	Apenas retorna o pacote de status definido
Reset de hardware (hardReset)	Geral	0xF	0	Reinicia o controlador para seu estado de recém-ligado. Possui a opção de salvar dados na EEPROM antes

2.3.5 COMUNICAÇÃO – ESCRAVO

O sistema escravo sempre inicia a transmissão com uma resposta a um comando recebido. Depois que um comando é recebido e executado, o controlador correspondente envia um pacote de estado consistindo de (LOGOSOL, 2002):

- Byte de estado
- 0 – 16 bytes de dados opcionais
- Byte de checksum

A Tabela 2.4 e a Tabela 2.5 descrevem os bits do byte de estado e do byte auxiliar de estado, respectivamente. Também é informado o mnemônico do bit que será usado no *software* desenvolvido neste trabalho.

Tabela 2.4. Byte de estado (adaptado de Logosol, 2002).

Bit	Nome	Definição
0	Movimento concluído (moveDone)	Não utilizado
1	Erro de *checksum* (checksumError)	Ativado se houver um erro de checksum no pacote de comandos recebido mais recentemente
2	Sobrecorrente (overCurrent)	Não utilizado
3	Ligado (powerOn)	Ativado se a tensão do dispositivo está entre 0,9 V e 4,5 V ou apagado, caso contrário
4	Erro de posição (positionError)	Não utilizado
5	Limite 1 (valorLimit1)	Não utilizado
6	Limite 2 (valorLimit2)	Não utilizado
7	Buscando posição home (homeInProgress)	Não utilizado

Tabela 2.5. Byte auxiliar de estado (adaptado de Logosol, 2002).

Bit	Nome	Definição
0	Index (index)	Não utilizado
1	Estouro na posição (posWrap)	Não utilizado
2	Servo ligado (servoOn)	Não utilizado
3	Acelerando (accelerating)	Não utilizado
4	Slew (slew)	Não utilizado
5	Overrun do ciclo de servo (servoOverrun)	Ativado quando os cálculos exigidos levam mais que 0,512 ms. Apagado com o comando apaga bits
6	Modo trajetória (pathMode)	Não utilizado
7	Não utilizado	Não utilizado

Vale observar que o primeiro byte enviado pelo controlador não é de cabeçalho, mas o byte de estado, que contém informações básicas do estado, incluindo um flag de erro do *checksum* para o comando recebido. Os bytes de dados opcionais devem incluir dados como posição, velocidade, etc. e é programável pelo mestre. O byte de *checksum* é a soma de 8 bits do byte de estado e os bytes adicionais de dados opcionais. Todos dados de 16 bits de 32 bits são enviados com o bit menos significativo primeiro.

Deve-se notar que muitos bits do byte de estado e do byte auxiliar de estado não serão utilizados por serem especificamente para o servocontrolador da Logosol. Entretanto, eles podem ser configurados para outras aplicações específicas, envolvendo o estado de funcionamento do dispositivo ou segurança. Os únicos bits do byte de estado utilizados são de erro no *checksum* e o que indica se o dispositivo está ligado ou não. Quanto ao byte auxiliar de estado, o único bit utilizado é o de *overrun* do ciclo de servo.

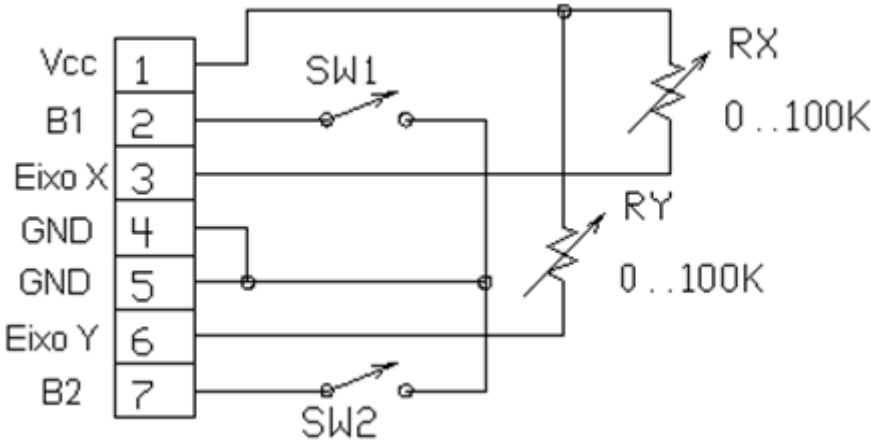
2.4 JOYSTICK

O *joystick* é um dispositivo composto basicamente por botões e um manche. Foi inicialmente concebido para o uso no controle de aviões, mas sua mais famosa aplicação é relacionada a jogos como uma interface de entrada. Entretanto, este dispositivo é utilizado no meio industrial para fins de posicionamento e orientação de ferramentas de robôs (por exemplo, manipuladores).

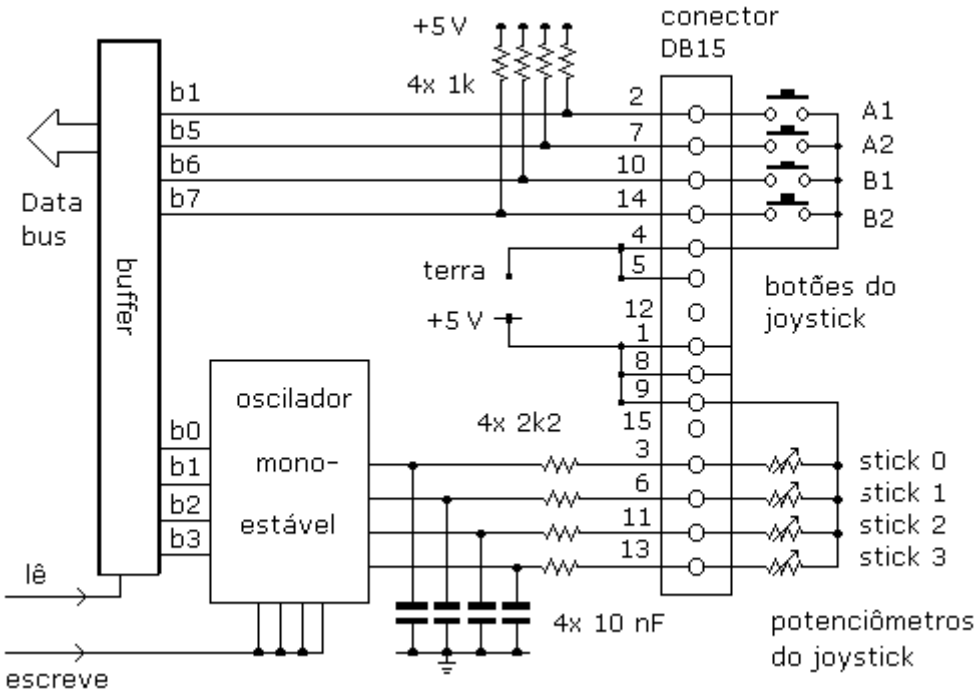
Este dispositivo, na versão mais simples encontrada no mercado, possui dois botões digitais e um manche com dois potenciômetros, um para cada eixo. Assim, a posição do manche é determinada pela tensão de saída, que por sua vez é determinada pelos potenciômetros. O *joystick* utilizado para este trabalho de graduação possuía dois

potenciômetros de 100 kΩ aproximadamente (ENGDAHL, 1996). O circuito interno do *joystick* se conecta ao computador conforme a Figura 2.6. A interface de *joystick* de um microcomputador é ilustrada na Figura 2.7.

Os sensores que fornecem sinais que indicam a posição do manche do *joystick* são potenciômetros nos modelos mais simples e tradicionais. Nos modelos mais modernos, são utilizados encoders incrementais, além de usarem uma interface USB para conexão com o computador. Nos modelos de potenciômetros, eles têm a sua resistência alterada de acordo com a movimentação do manche. Ainda há a possibilidade de se calibrar cada um dos eixos para se ajustar a posição central do manche por meio de cursores na base do *joystick*.

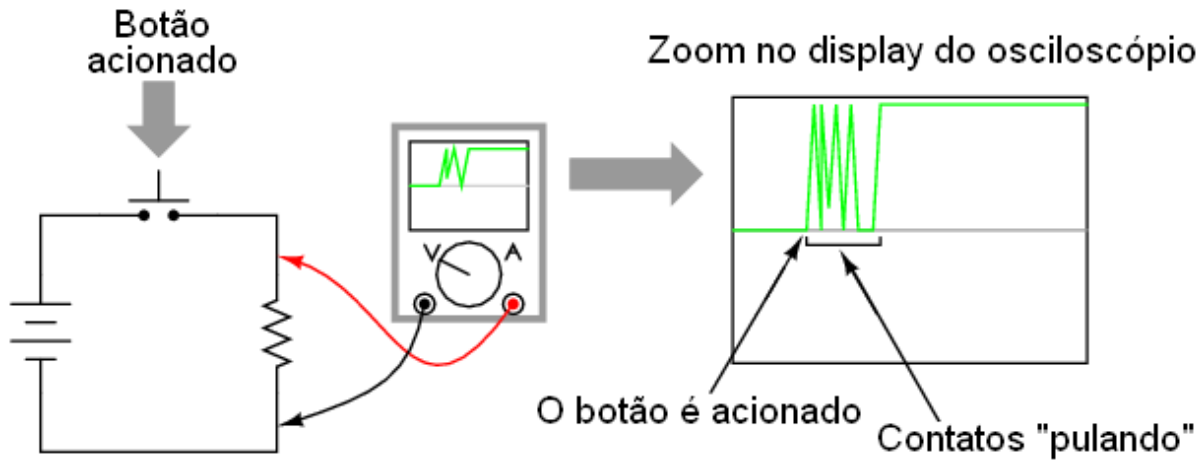


2.6. Circuito interno de um *joystick* (adaptado de Engdahl, 1996).



2.7. Circuito típico de uma interface de *joystick* de um microcomputador (Haag e Veit, 2001).

Os botões são chaves mecânicas que fecham contato ao serem acionadas. Um problema inerente a este tipo de acionamento mecânico é o efeito *bounce*. Tal efeito se dá devido à massa do contato que se move e à elasticidade inerente ao mecanismo e aos materiais, causando “saltos” e, conseqüentemente, fechamentos múltiplos dos contatos durante certo intervalo de tempo até estabilizarem e produzirem um sinal contínuo sem perturbações. Esse efeito é ilustrado na Figura 2.8.



2.8. Ilustração do efeito *bounce* (adaptado de Kuphaldt, 2007).

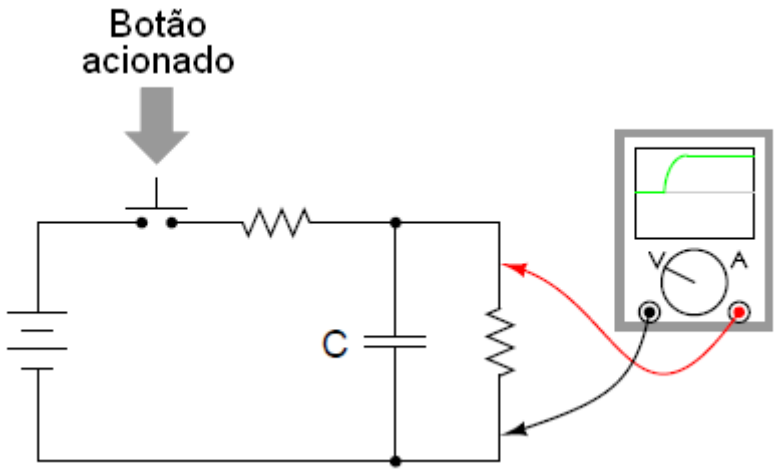


Figura 2.9. Redução das oscilações por meio de um filtro passa-baixa (adaptado de Kuphaldt, 2007).

Para sistemas lentos, este efeito indesejável passaria despercebido, mas circuitos eletrônicos possuem respostas rápidas em geral e, portanto, interpretariam essas oscilações, prejudicando o sinal de saída. A gravidade deste efeito depende da qualidade dos materiais utilizados para os contatos, ou seja, não é possível eliminá-lo por completo,

apenas atenuá-lo com materiais de melhor qualidade. Entretanto, a sua influência em circuitos eletrônicos pode ser tratada com o acréscimo de um filtro passa-baixa na saída da chave, como ilustrado na Figura 2.9.

2.5 SERVO

Servos são motores DC com o mecanismo e o circuito de controle por realimentação embutidos. Além disso, não é necessário o uso de drivers para o seu funcionamento.

Eles são muito utilizados na construção de robôs e aeromodelos, ou seja, na robótica móvel em geral. Geralmente os servomotores podem rotacionar entre 90 e 180 graus, alguns podem chegar a 360 graus ou mais (SOCIETY OF ROBOTS, 2010). Entretanto, os servos não são capazes de rotacionar continuamente, impossibilitando a sua utilização com rodas e outros dispositivos que requerem rotação contínua (a não ser que o mecanismo interno seja modificado), mas a precisão de posicionamento os tornam ideais para uso em membros de robôs, caixa de direção tipo pinhão e cremalheira e sensores de varredura, etc (SOCIETY OF ROBOTS, 2010).

Uma vez que os servos são totalmente independentes, o controle por realimentação do ângulo ou da velocidade são fáceis de serem implementados, enquanto que os preços permanecem acessíveis (SOCIETY OF ROBOTS, 2010).

Para se usar um servo, basta simplesmente conectar o fio preto/marrom ao terra comum, o vermelho a uma fonte de 4.8-6 V e o fio amarelo/laranja/branco a um gerador de sinais (no caso, o de um microcontrolador) (SOCIETY OF ROBOTS, 2010). Variando a largura do pulso de onda quadrada de 1-2 ms, tem-se um controle de posição/velocidade do servo.

Todos os servos possuem 3 fios, onde o fio preto/marrom é o terminal para o terra comum, o vermelho é para a alimentação (~4.8-6 V) e o fio amarelo/laranja/branco é para a aplicação de um sinal de controle (3-5 V) (SOCIETY OF ROBOTS, 2010). A razão para se utilizar essa faixa de operação (fio vermelho) é o fato de que a maioria dos microcontroladores e receptores de rádio operam nesta faixa (SOCIETY OF ROBOTS, 2010). A não ser que se tenha uma limitação de tensão/corrente/potência, deve-se operar com 6 V, pois motores DC fornecem um maior torque para maiores tensões (SOCIETY OF ROBOTS, 2010).

O fio de sinal é o utilizado para comandar o servo. O conceito geral é de simplesmente enviar uma onda quadrada comum de um determinda largura para o servo e ele rotaciona um determinado ângulo (ou varia a velocidade, no caso do servo ter sido modificado) (SOCIETY OF ROBOTS, 2010). A largura do pulso define diretamente o ângulo de posicionamento do servo.

Se o sistema é controlado remotamente, o receptor RC aplicará a onda quadrada adequada ao servo (SOCIETY OF ROBOTS, 2010). Caso o controle seja feito por um microcontrolador, o procedimento para aplicar a onda quadrada deve ser seguir a seqüência:

- Colocar um pino em nível alto;
- Esperar entre 1-2 ms;
- Colocar o mesmo pino em nível baixo;
- Repetir o ciclo algumas dúzias de vezes por segundo.

Vale observar que o maior número de servos possíveis de serem controlados simultaneamente é definido pelo número de pinos digitais disponíveis para serem utilizados (SOCIETY OF ROBOTS, 2010).

A relação entre a largura do pulso aplicado e a posição angular do servo é mostrada na Figura 2.10.

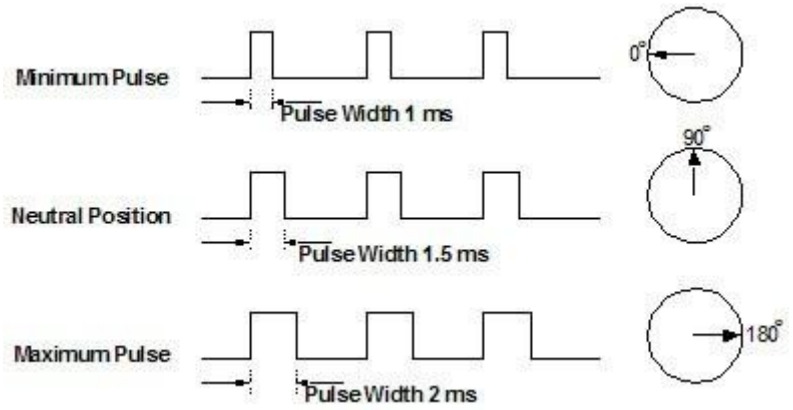


Figura 2.10. Relação entre a largura do pulso aplicado e a posição angular do servo (SOCIETY OF ROBOTS, 2010).

2.6 MOTOR DE PASSO

O motor de passo é um dispositivo eletromecânico que converte pulsos elétricos em movimentos mecânicos que geram variações angulares discretas (Brites e Almeida, 2008). O rotor ou eixo é rotacionado em pequenos incrementos angulares chamados de “passos”, quando pulsos elétricos são aplicados em uma determinada seqüência nos terminais do motor de passo.

Um motor de passo pode ser utilizado em aplicações que exigem movimentos precisos como ângulo de rotação, velocidade, posição e sincronismo (impressoras, scanners, robôs, câmeras de vídeo, brinquedos, automação industrial, etc.). No entanto, ele

possui as desvantagens de não fornecer um torque alto e também sua capacidade limitada para desenvolver altas velocidades (Brites e Almeida, 2008).

O funcionamento básico do motor de passo é dado pelo uso de solenóides alinhados dois a dois que atraem o rotor quando energizados, fazendo ele se alinhar com o eixo determinado pelo par de solenóides, gerando um deslocamento angular cuja unidade é o passo. O número de passos em uma rotação completa é definido pelo número de alinhamentos possíveis entre o rotor e as bobinas (Brites e Almeida, 2008). A ordem de energização dos solenóides determina o sentido de rotação, a velocidade entre cada energização determina a velocidade de rotação e a quantidade de energizações em seqüência determina o ângulo total (número de passos total) rotacionado (Brites e Almeida, 2008).

2.6.1 TIPOS DE MOTORES DE PASSO

Há basicamente três tipos de motores de passo: Relutância Variável, Ímã Permanente e Hibrído.

- **Relutância Variável**

Este tipo de motor de passo consiste de um rotor de ferro com múltiplos dentes e um estator com enrolamentos. Quando os enrolamentos do estator são energizados com corrente DC, os pólos ficam magnetizados. A rotação ocorre quando os dentes do rotor são atraídos para os pólos do estator energizado devido à força magnética que surge, para que o sistema tenha o circuito com a menor relutância (Brites e Almeida, 2008). A Figura 2.11 ilustra o referido tipo de motor.

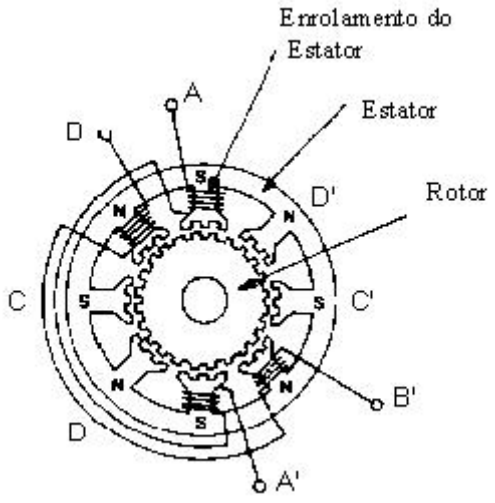


Figura 2.11. Seção transversal de um motor de relutância variável (Brites e Almeida, 2008).

- **Ímã Permanente**

Motores de ímã permanente têm baixo custo e baixa resolução, com passos típicos de 7,5º a 15º (48-24 passos/revolução). O rotor é construído com ímãs permanentes e não possui dentes. Os pólos magnetizados do rotor provêm uma maior intensidade de fluxo magnético e por isto este tipo de motor exibe uma melhor característica de torque, quando comparado ao de relutância variável (Brites e Almeida, 2008). A Figura 2.12 ilustra este tipo de motor.

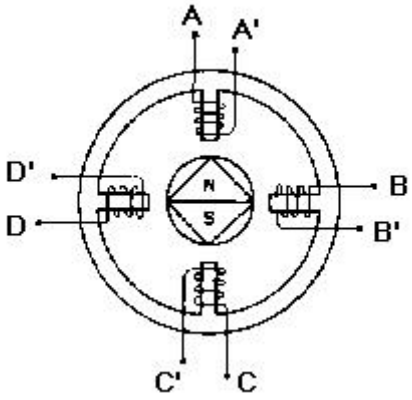


Figura 2.12. Motor de passo de ímã permanente (Brites e Almeida, 2008).

- **Híbrido**

O motor de passo híbrido é mais caro do que o de ímã permanente, mas tem melhor desempenho quanto à resolução de passo, torque e velocidade. Ângulos de passo típico de motores híbridos estão entre 3,6º a 0,9º (100-400 passos/revolução). Ele combina as melhores características dos motores de ímã permanente e motor de relutância variável, possuindo dentes e um ímã permanente ao redor do seu eixo. Os dentes provêm um melhor caminho que ajuda a guiar o fluxo magnético para locais preferidos no GAP de ar (Brites e Almeida, 2008). A Figura 2.13 traz a ilustração do motor em questão.

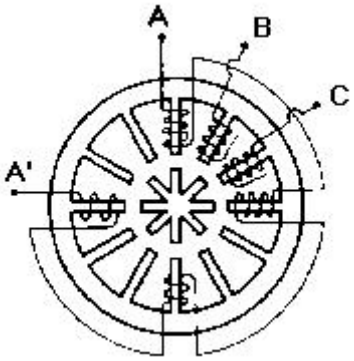


Figura 2.13. Motor de passo híbrido (Brites e Almeida, 2008).

2.6.2 TIPOS DE ENROLAMENTO

Quanto ao tipo de enrolamento, o motor de passo pode ser basicamente de dois tipos: unipolar ou bipolar.

- **Unipolar**

Um motor de passo unipolar tem dois enrolamentos por fase, um para cada sentido da corrente. Desde que neste arranjo um pólo magnético possa ser invertido sem comutar o sentido da corrente, o circuito da comutação pode ser feito de forma muito simples (por exemplo, um único transistor) para cada enrolamento. Tipicamente, dado uma fase, um terminal de cada enrolamento é feito como comum: dando três ligações por fase e seis ligações para um motor bifásico típico. Freqüentemente, os terminais em comum das duas fases são ligados internamente, assim o motor tem somente cinco ligações.

A resistência entre o fio comum e o fio de excitação da bobina é sempre a metade da resistência entre os fios de excitação da bobina, devido ao fato de que há somente meio comprimento do centro (o fio comum) à uma das extremidades. Os motores de passo unipolares com seis ou oito fios podem ser conduzidos com excitadores bipolares, deixando os terras em comum da fase desconectadas e conduzindo os dois enrolamentos de cada fase junto. É igualmente possível usar um excitador bipolar para conduzir somente um enrolamento de cada fase, deixando a metade dos enrolamentos não utilizada (Brites e Almeida, 2008). A Figura 2.14 ilustra este tipo de motor.

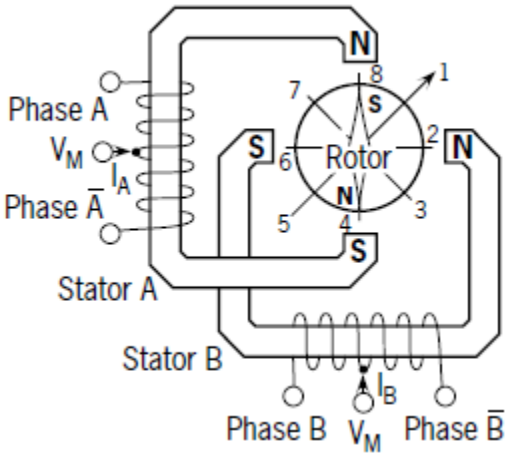


Figura 2.14. Motor de passo unipolar (adaptado de Ericsson (2010)).

- **Bipolar**

Este tipo de motor tem um único enrolamento por fase. A corrente em um enrolamento precisa ser invertida a fim de inverter um pólo magnético, assim o circuito de condução é um pouco mais complexo. Há duas ligações por fase e nenhuma está em

comum. Como os enrolamentos são melhores utilizados, são mais poderosos do que um motor unipolar de mesmo peso (Brites e Almeida, 2008). A Figura 2.15 traz a ilustração do motor em questão.

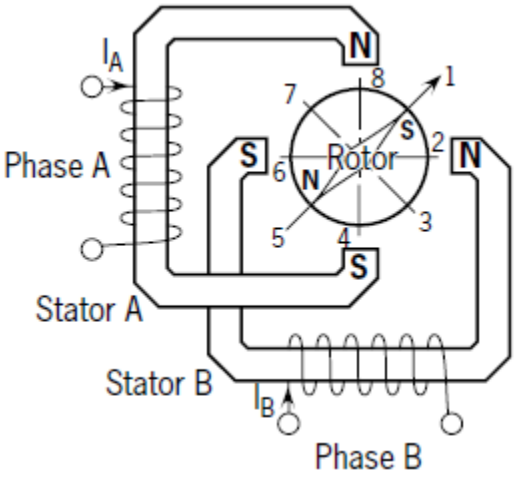


Figura 2.15. Motor de passo bipolar (adaptado de Ericsson (2010)).

2.6.3 MODOS DE PASSO

Quanto aos modos de passo, o motor de passo pode operar de duas maneiras: passo completo (full-step) ou meio-passo (half-step), dependendo da forma como as bobinas são polarizadas.

- **Passo completo (full-step)**

No modo passo completo, o rotor se movimenta tendo suas pás alinhadas com as bobinas ou entre as últimas. A Tabela 2.6 e a Tabela 2.7 mostram as seqüências de alimentação das bobinas do modo de passo completo para motores unipolares e bipolares respectivamente. A Figura 2.16 e a Figura 2.17 ilustram o movimento do motor de passo unipolar e bipolar respectivamente, baseado nos diagramas da Figuras 2.14 e da Figura 2.15.

Tabela 2.6. Seqüência de alimentação de um motor de passo unipolar para se obter o modo de passo completo (sentido horário).

Fase / Passo	$V_M\bar{A}$	V_MB	V_MA	$V_M\bar{B}$
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

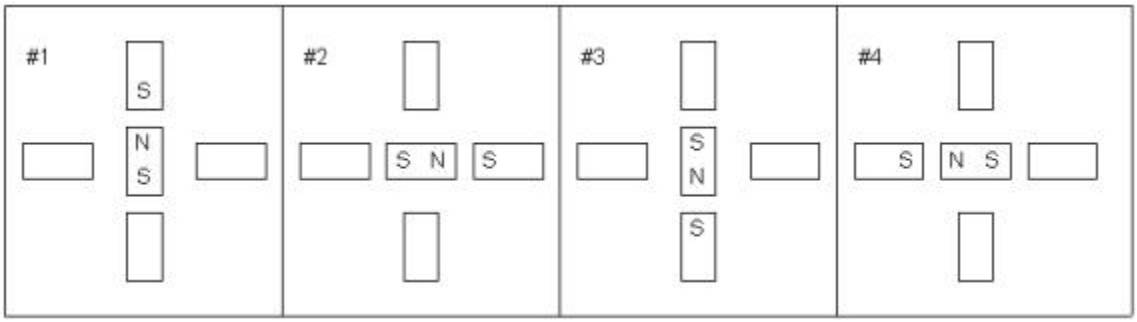


Figura 2.16. Modo de passo completo para um motor unipolar (Brites e Almeida, 2008).

Tabela 2.7. Seqüência de alimentação de um motor de passo bipolar para se obter o modo de passo completo (sentido horário).

Fase / Passo	A	$\overline{A}$	B	$\overline{B}$
1	−	+	+	−
2	+	−	+	−
3	+	−	−	+
4	−	+	−	+

+ = fluxo de corrente positivo, − = fluxo de corrente negativo

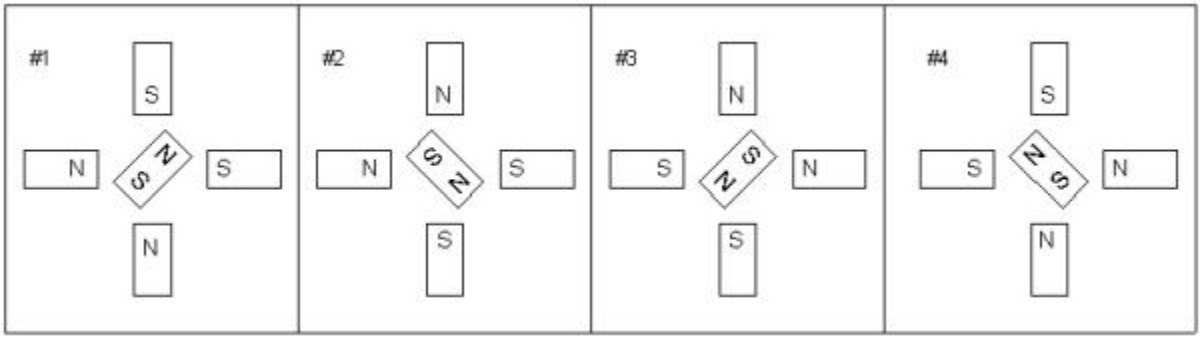


Figura 2.17. Modo de passo completo para um motor bipolar (Brites e Almeida, 2008).

- **Meio-passo (half-step)**

No modo meio-passo, o rotor se movimenta tendo suas pás alinhadas com as bobinas e entre as últimas durante a revolução. A Tabela 2.8 e a Tabela 2.9 mostram as seqüências de alimentação das bobinas do modo de passo completo para motores unipolares e bipolares respectivamente. A Figura 2.18 e a Figura 2.19 ilustram o movimento do motor de passo unipolar e bipolar respectivamente, baseado nos diagramas da Figura 2.14 e da Figura 2.15.

Tabela 2.8. Seqüência de alimentação de um motor de passo unipolar para se obter o modo de meio-passo (sentido horário).

Fase / Passo	$V_M\bar{A}$	V_MB	V_MA	$V_M\bar{B}$
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

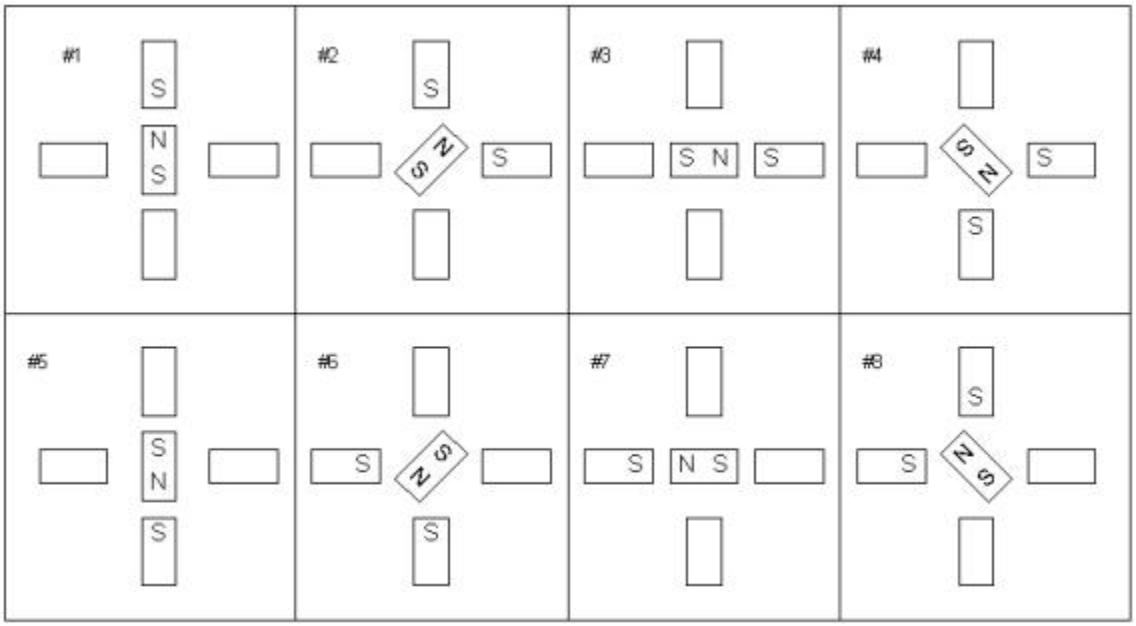


Figura 2.18. Modo de meio-passo para um motor unipolar (Brites e Almeida, 2008).

Tabela 2.9. Seqüência de alimentação de um motor de passo unipolar para se obter o modo de meio-passo (sentido horário).

Fase / Passo	A	$\bar{A}$	B	$\bar{B}$
1	−	+	0	0
2	−	+	+	−
3	0	0	+	−
4	+	−	+	−
5	+	−	0	0
6	+	−	−	+
7	0	0	−	+
8	−	+	−	+

+ = fluxo de corrente positivo, − = fluxo de corrente negativo

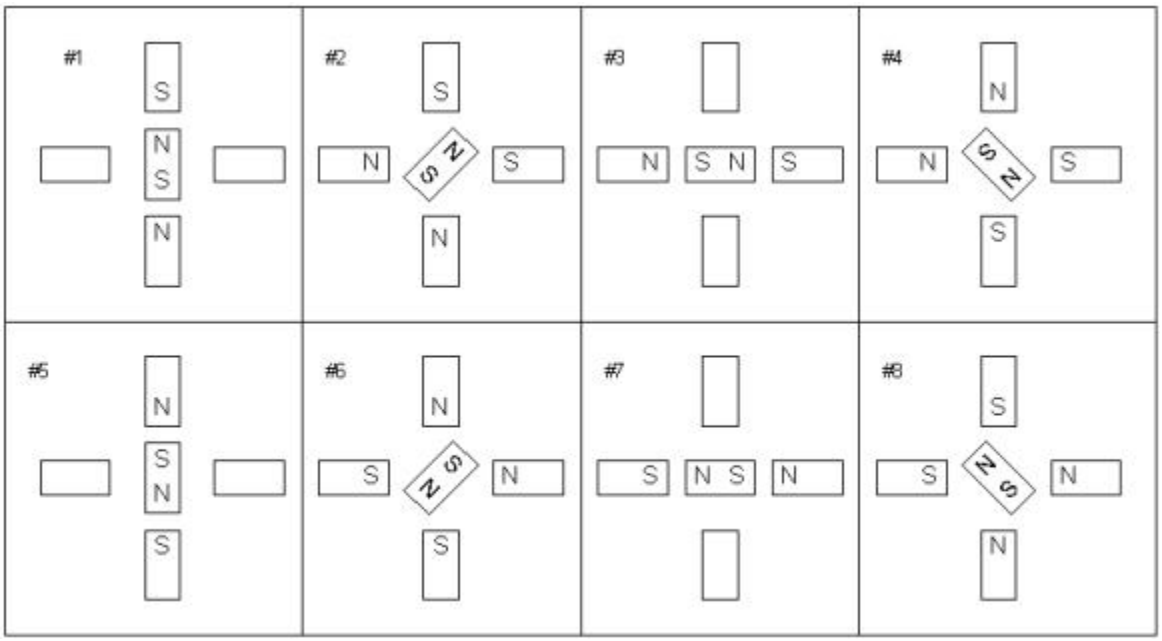


Figura 2.19. Modo de meio-passo para um motor bipolar (Brites e Almeida, 2008).

CAPÍTULO 3 – IMPLEMENTAÇÃO DA PROPOSTA

No capítulo anterior, discutiu-se os conceitos a respeito dos dispositivos utilizados para a montagem do *hardware*, o barramento de dados a ser usado e o protocolo de comunicação que o *software* deve seguir, assim como algumas de suas funcionalidades. Neste capítulo, o *hardware* e o *software* serão projetados e descritos.

3.1 HARDWARE

O *hardware* foi montado com a utilização de uma placa perfurada e uma *protoboard*. A placa perfurada foi utilizada para servir como uma placa genérica para ser utilizado com cada dispositivo proposto, sendo que ela possuía dois transceptores MAX-485 (ver Figura 3.3), um para recepção e outro para transmissão dos dados do microcontrolador, para converterem o sinal diferencial do barramento em níveis TTL. A *protoboard* foi utilizada para o caso em que foi necessária a montagem de circuitos auxiliares como filtros, *buffers* e outros circuitos de proteção.

3.1.1 MICROCONTROLADOR ATMEL ATMEGA8

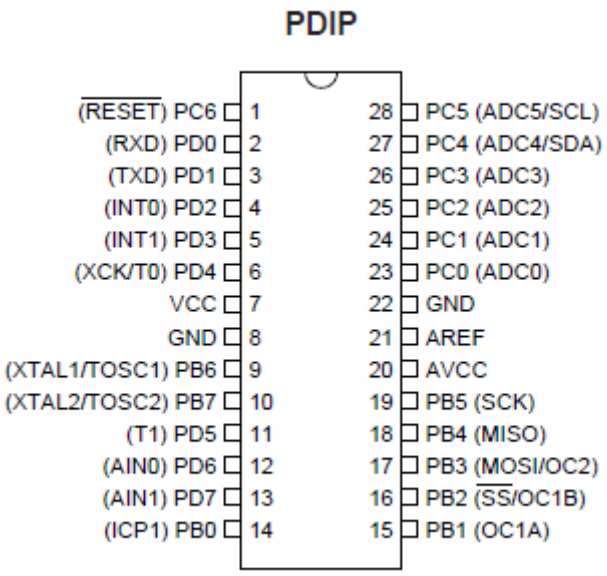


Figura 3.1. Pinagem do microcontrolador ATmega8 (ATMEL, 2010).

A Figura 3.1 mostra a pinagem do microcontrolador ATmega8. Deve-se observar a presença de duas entradas GND e duas VCC (VCC e AVCC). Os pinos 7 e 8 são usados para alimentar o circuito digital do sistema, enquanto que os pinos 20 e 22 são usados para

alimentar o circuito de conversão analógica/digital. Os pinos 20 e 22 devem ser conectados, mesmo que o conversor A/D não seja utilizado. Caso seja usado, é importante conectar um filtro passa-baixa na entrada do pino 20 (AVCC) para reduzir o ruído durante a conversão (ATMEL, 2010).

Outro detalhe importante é que muitos pinos são multifuncionais. Uma função comum a quase todos eles é ser um bit de entrada ou de saída de dados digitais de uma porta. Existem 3 portas, B, C e D com 8, 7 e 8 bits respectivamente. A nomenclatura utilizada para os pinos é P seguido da porta e o bit da porta, por exemplo, PD2 (pino 4) corresponde ao bit 2 da porta D. Outras funções são: RXD e TXD, que são utilizadas como recepção e transmissão da comunicação serial; INT0 e INT1, que geram interrupções externas de *hardware*; XTAL1 e XTAL2, que recebem o sinal oscilante de um cristal (se utilizado); ADC0-ADC5, que são canais multiplexados do conversor A/D e os pinos OC1A, OC1B e OC2, que podem ser saídas de sinal PWM.

Os pinos 12 e 13 (PD6 E PD7) são utilizados no processo de endereçamento automático e correspondem aos sinais ADDR_IN e ADDR_OUT respectivamente. O ADDR_OUT de uma placa deve ser conectado ao ADDR_IN da próxima. A primeira placa deve conectar o pino correspondente ao ADDR_IN ao GND e a última placa deve conectar o pino correspondente ao ADDR_OUT em aberto.

No entanto, a utilização ou não das funções vai variar conforme a aplicação envolvida, uma vez que os projetos das placas se diferem entre si.

3.1.2 CONVERSOR RS-232/RS-485 E TRANSCEPTOR MAX485

O conversor RS-232/RS-485 utilizado neste projeto já estava disponível no laboratório. A Figura 3.2 mostra o esquemático do circuito do conversor e a Figura 3.4 mostra a foto do mesmo.

No primeiro transceptor das placas, os pinos 2 e 3 ($\overline{RE}$ e DE) foram conectados ao GND, de forma que ele esteja sempre em modo de recepção. O pino RO é conectado ao RXD do microcontrolador, de forma que o mesmo receba os dados do barramento por meio do primeiro transceptor, enquanto que o pino DI foi conectado ao GND.

No segundo transceptor das placas, os pinos 2 e 3 ($\overline{RE}$ e DE) foram conectados ao V_{CC}, de forma que ele fique apenas no modo de transmissão. O pino DI foi conectado ao TXD do microcontrolador, de forma que o mesmo possa enviar dados ao barramento por meio do segundo transceptor. O pino RO foi deixado em aberto.

Os pinos 6 e 7 (A e B) de ambos transceptores foram ligados aos conectores RS-485a e RS-485b. As saídas Z e Y do circuito integrado LTC491 (LINEAR TECHNOLOGY, 1992) foram ligadas aos pinos 6 e 7 do transceptor MAX-485 (MAXIM, 2009) de recepção

das placas. As saídas A e B foram ligadas aos pinos 6 e 7 do transceptor MAX-485 de transmissão das placas.

A Figura 3.3 mostra a pinagem do transceptor MAX485

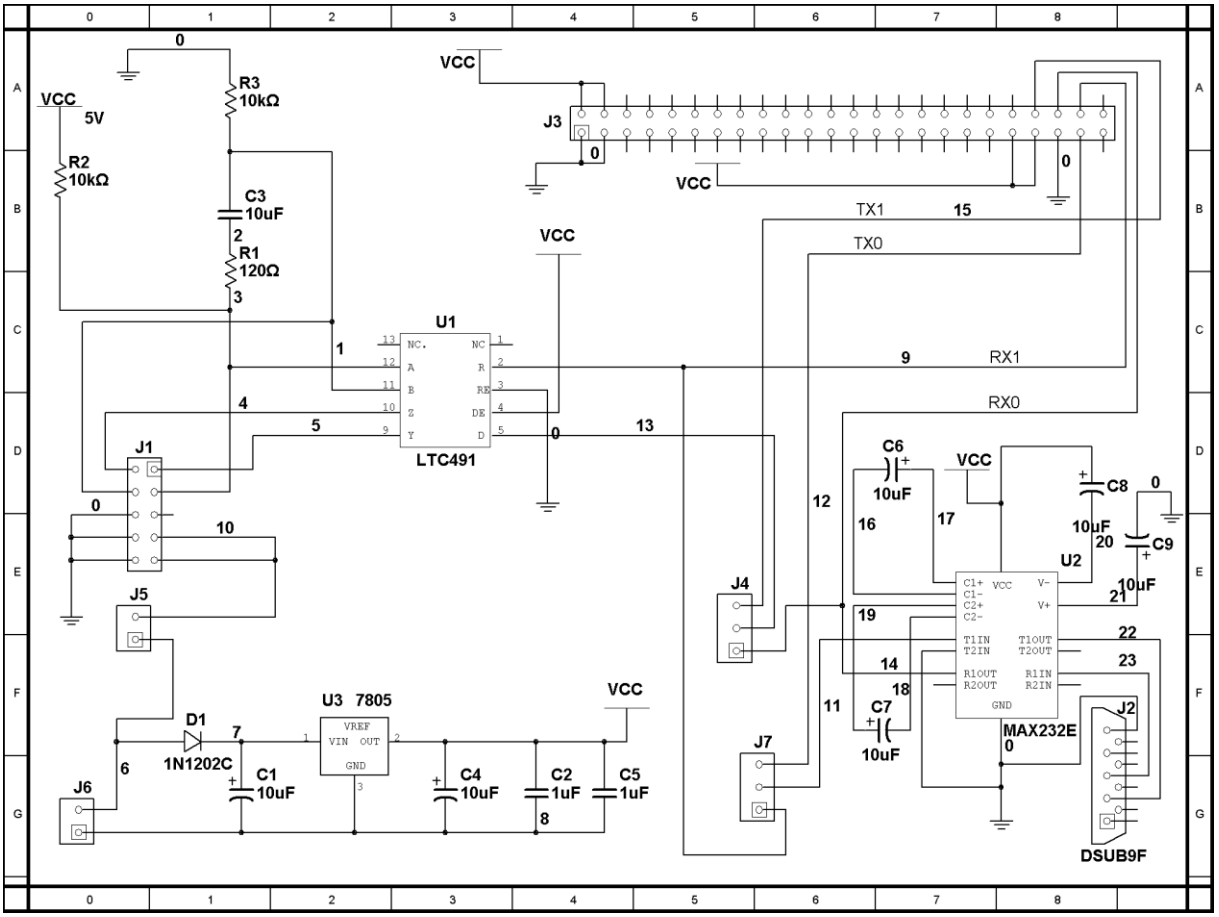


Figura 3.2. Conversor RS-232/RS-485 utilizado no projeto.

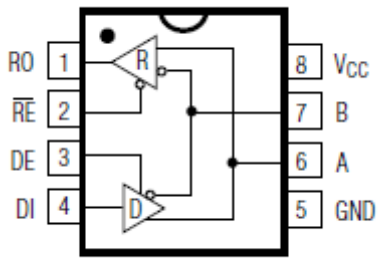


Figura 3.3. Pinagem do circuito integrado MAX485 (MAXIM, 2009).

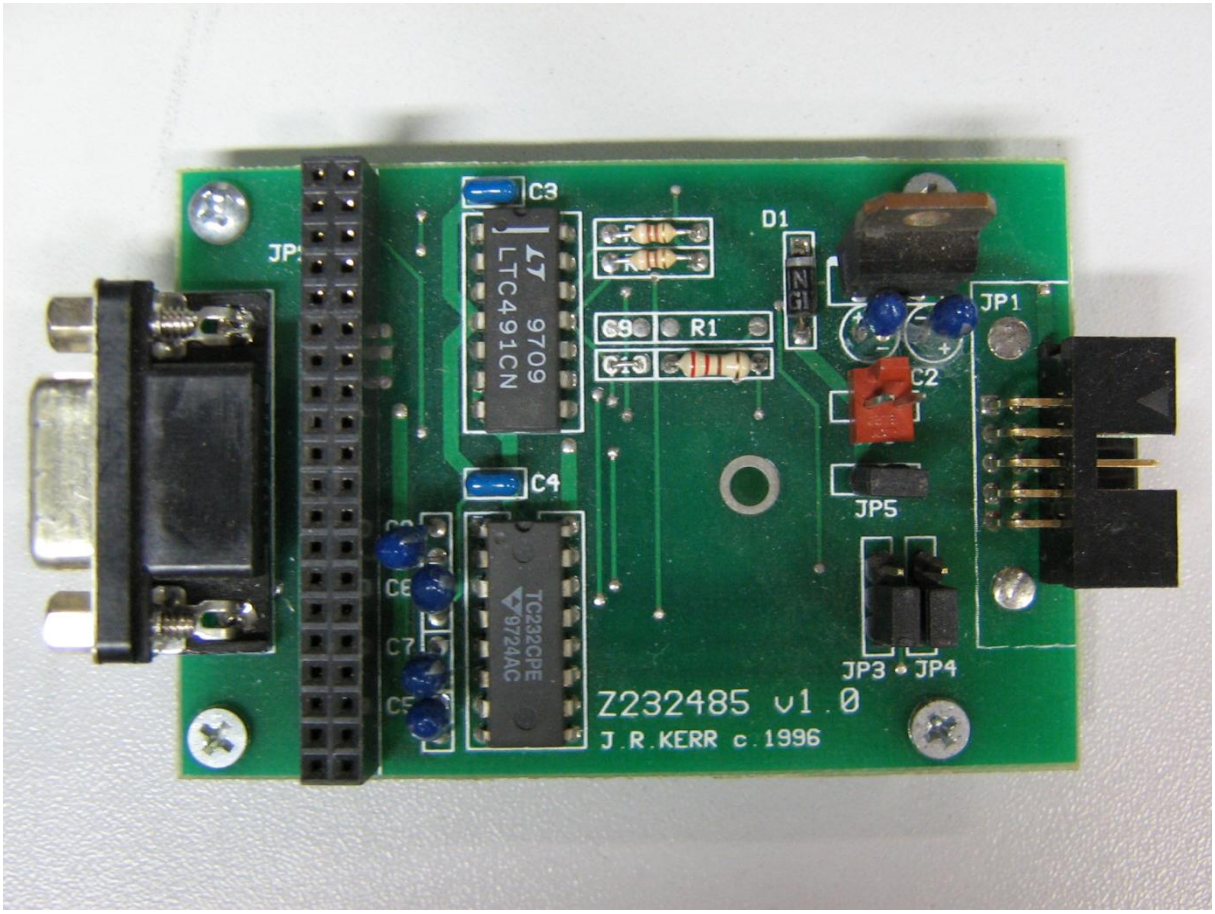


Figura 3.4. Foto do conversor RS-232/RS-485.

3.1.3 PLACA PARA O MONITORAMENTO DO ESTADO DO JOYSTICK

A par dos conceitos envolvendo o funcionamento do joystick, teve-se que modificar o circuito interno do mesmo para que se fosse possível obter sinais de tensão a partir de uma deflexão do manche. Estes sinais de tensão são medidos com a utilização de dois canais do conversor A/D do microcontrolador. Em sua configuração original mostrada na Figura 2.6, uma mudança de posicionamento do manche resultaria em uma mudança na resistência do joystick que seria interpretada pela interface do microcomputador. Com as mudanças mostradas na Figura 3.5, o circuito gera um sinal de tensão como saída.

Outro fator considerado no projeto da placa do joystick foi a presença de ruído que poderia interferir no valor de tensão de saída do circuito interno do dispositivo correspondente a cada eixo. Para contornar este problema, foi projetado um filtro passa-baixa passivo de primeira ordem para cada eixo. O valor da freqüência de corte foi escolhido em função do Teorema de Nyquist, o qual estabelece que as componentes espectrais presentes no sinal não devem possuir freqüências acima da metade da freqüência de amostragem que, no caso, é de 20 Hz (período de 50 ms). O filtro inicialmente projetado possuía uma freqüência de corte f_c = 10 Hz (freqüência de Nyquist), mas não havia

componentes disponíveis para construí-lo exatamente com esse valor. Dessa forma, o filtro foi construído com constante de tempo de 22 ms (freqüência de corte f_c = 7,23 Hz), que atende o Teorema de Nyquist.

Como explicado na seção 2.4, outro efeito indesejável presente no dispositivo é o efeito bounce, inerente ao mecanismo das chaves, representado pelos botões. Para minimizar este problema, foi projetado um filtro passa-baixa passivo com constante de tempo de 100 ms (freqüência de corte f_c = 1.6 Hz) para cada chave cuja finalidade é de ignorar qualquer transição brusca da tensão provocada pelo acionamento/desligamento das chaves. Esse valor foi escolhido pelo fato de que após um intervalo de cinco constantes de tempo o contato da chave se estabiliza, não havendo mais oscilações, ao mesmo tempo em que foi considerado que o tempo de resposta não fosse muito lento. No caso, um tempo de resposta de 500 ms (cinco constantes de tempo), foi adequado para que a transição do estado da chave se desse de maneira suave e suficientemente rápida. O circuito da referida placa é mostrado na Figura 3.6.

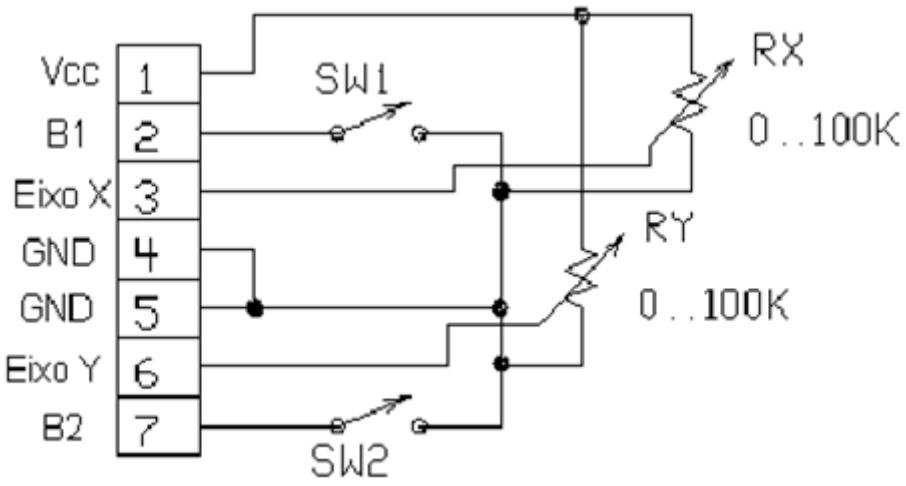


Figura 3.5. Circuito interno do joystick modificado.

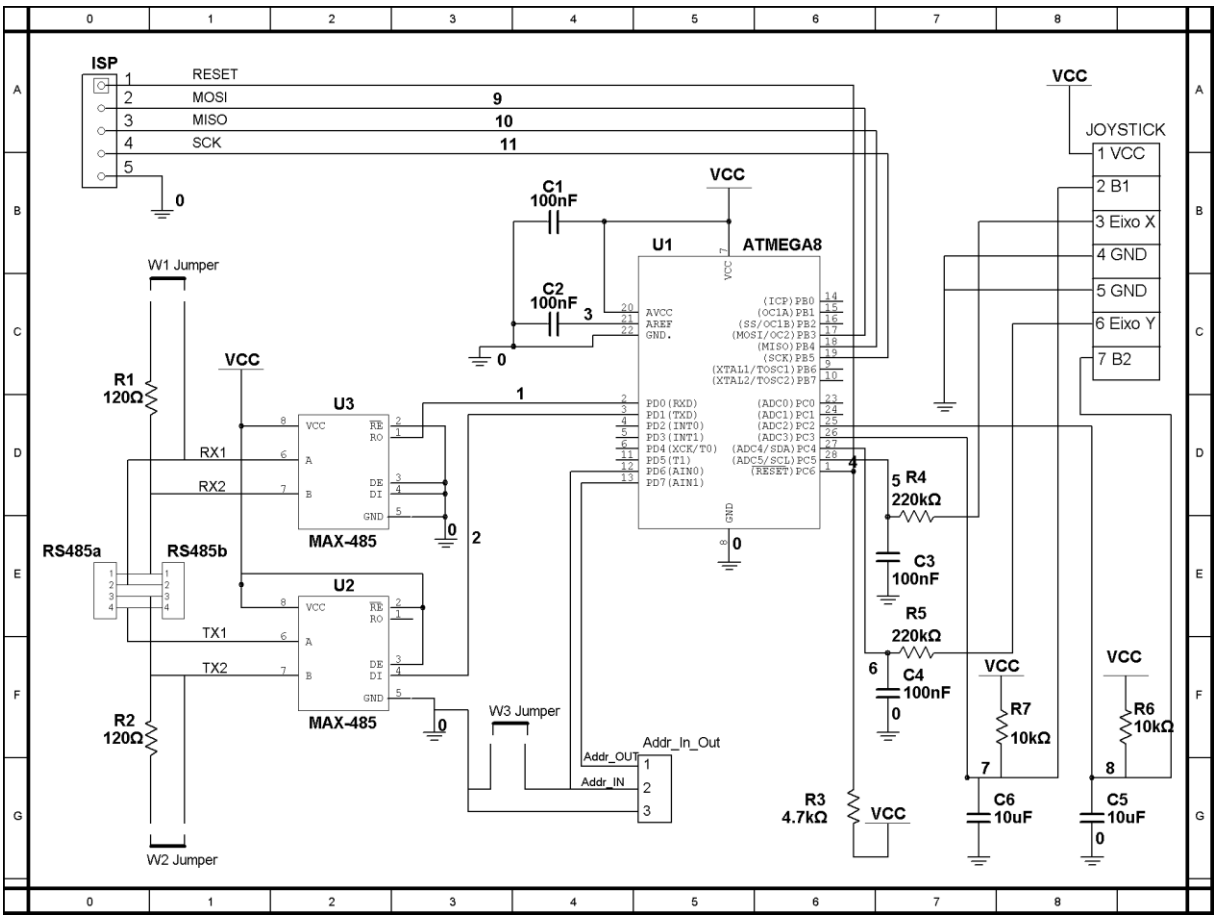


Figura 3.6. Esquemático da placa de monitoramento do joystick.

3.1.4 PLACA DE ENTRADA E SAÍDA DE DADOS AD

A placa de entrada e saída de dados A/D foi projetada para ter duas saídas analógicas por meio do PWM gerado em duas saídas do microcontrolador, 6 entradas analógicas para o conversor A/D e 8 entradas/saídas digitais. Quais e como as entradas/saídas serão utilizadas ficam a cargo da configuração fornecida pelo usuário pelo software mestre (ver seção 3.2.2 e o Anexo I para mais detalhes. A Figura 3.7 ilustra o esquemático do circuito da placa em questão.

Outro aspecto importante é o posicionamento dos pinos de entrada/saída digitais. Os 8 pinos disponíveis não são contínuos (PD2-PD5, PB0, PB1, PB4 e PB5) ou seja, é necessário haver um tratamento dos pinos das portas para utilizá-las como um byte.

Todos os canais do conversor A/D do microcontrolador foram disponibilizados para uso de acordo com a necessidade do usuário.

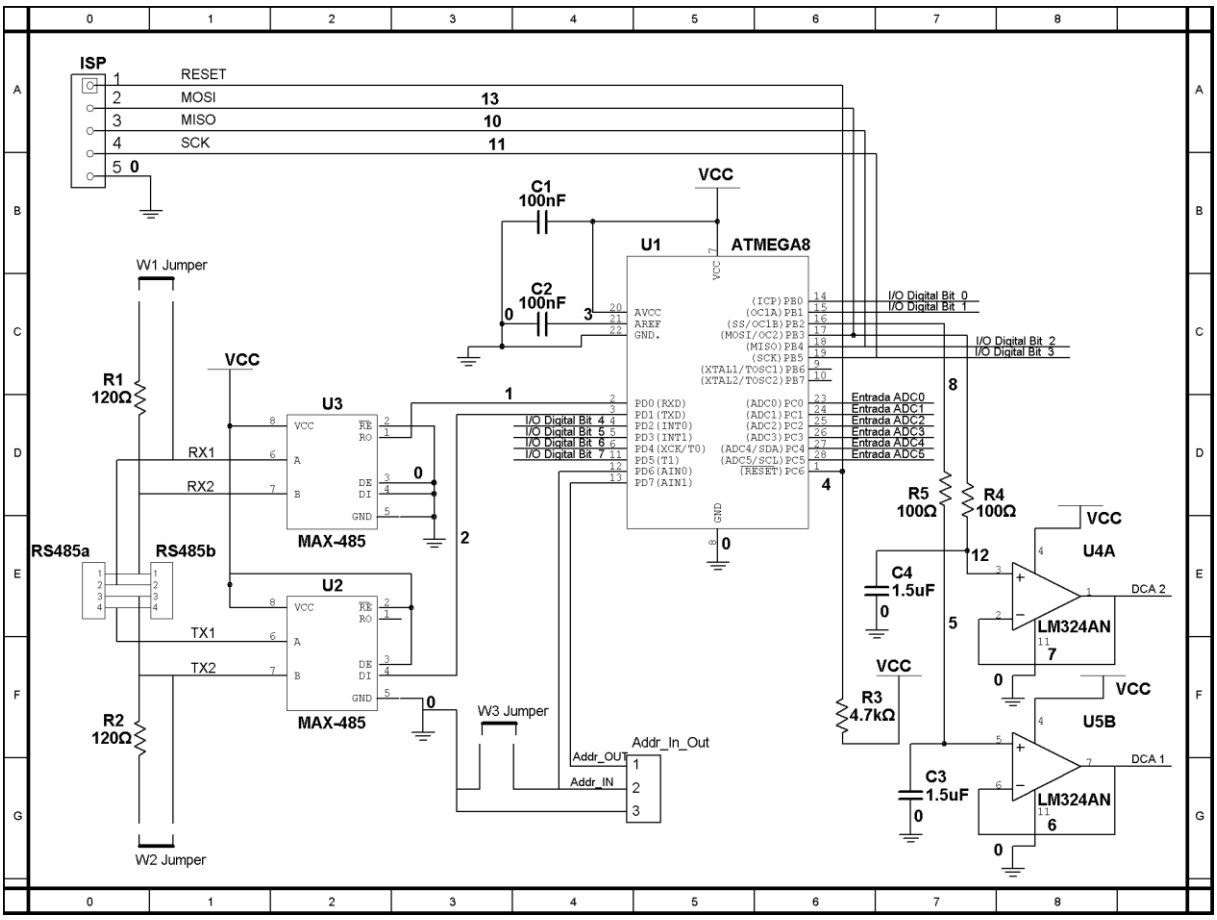


Figura 3.7. Esquemático da placa de entrada e saída de dados AD.

Foram utilizados dois filtros passa-baixas ativos para as saídas PWM para atenuar a influência do ruído. Tais filtros têm constante de tempo de 150 µs (freqüência de corte f_c = 1 kHz). O valor escolhido para a freqüência corte se deve a duas razões. A primeira, é o fato dos sinais PWM gerados nas saídas do microcontrolador possuírem uma freqüência de 31 kHz aproximadamente, então o filtro precisa ter uma freqüência de corte muito menor para que somente o valor médio seja percebido na saída. A segunda é que o valor escolhido permite uma resposta do sistema com freqüência de até 1 kHz, o que é suficientemente ráopido.

O circuito integrado LM324 possui 4 amplificadores operacionais e foi utilizado por sua alimentação poder ser assimétrica (no caso, foi utilizado 0 e 12 V)

3.1.5 PLACA DE CONTROLE DE SERVOMOTOR

A placa de controle de servomotor foi projetada para o controle de 6 servomotores pelos pinos PC0-PC5 da porta C do microcontrolador. A Figura 3.8 mostra o esquemático do circuito completo.

O servomotor utilizado para o desenvolvimento desta aplicação é da marca Motor Tech e já estava disponível no laboratório. Uma vez que não foi possível encontrar um manual para o dispositivo, foram realizados testes com um gerador de funções do laboratório. Os testes mostraram que ele operava adequadamente para qualquer pulso com freqüência na faixa de 50 a 120 Hz (período de 20 e 8,33 ms respectivamente).

O circuito implementado possui capacidade de operar simultaneamente até 6 servomotores.

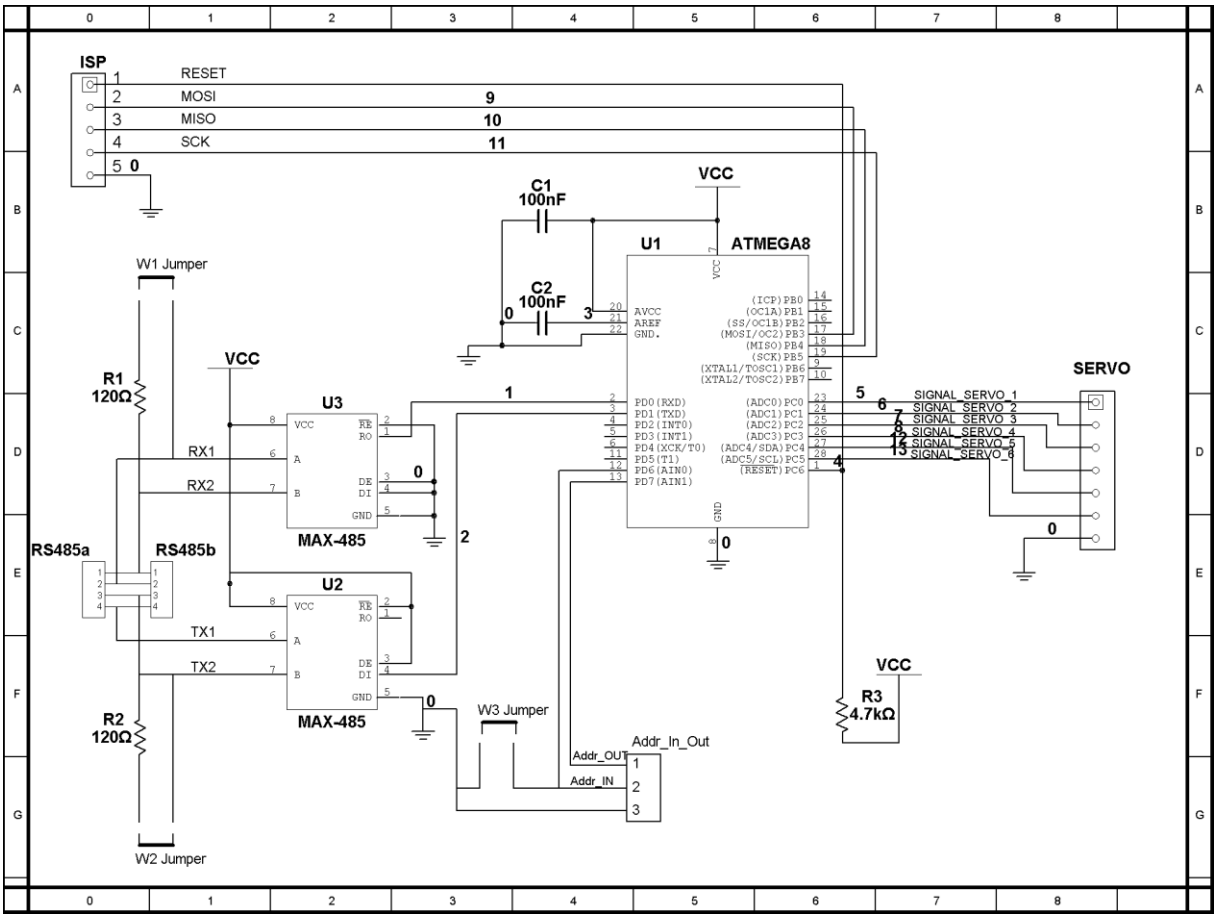


Figura 3.8. Esquemático da placa de controle de servomotor.

3.1.6 PLACA DE CONTROLE DE MOTOR DE PASSO

A placa de controle do motor de passo foi projetada para o controle de dois motores de passo unipolares (possuem 5 fios) de ímã permanente com o modo de operação de passo completo. O esquemático do circuito completo é mostrado na Figura 3.9.

Os motores utilizados foram obtidos no almoxarifado do bloco SG11. Eles não possuíam todas as especificações disponíveis, tanto no dispositivo quanto na internet. Foram realizados testes com o multímetro na função de ohmímetro para se descobrir os terminais de cada bobina.

Também foi testado até que freqüência eles respondiam. O motor de passo da Minibea Co. LTDA., modelo 17BB-H152, respondia até 100 Hz (140 Ω, 12 V, 7,5º graus/passo) e da Mitsumi, modelo M42SP-5P, até 250-300 Hz (50 Ω, 12 V, 7,5º/passo).

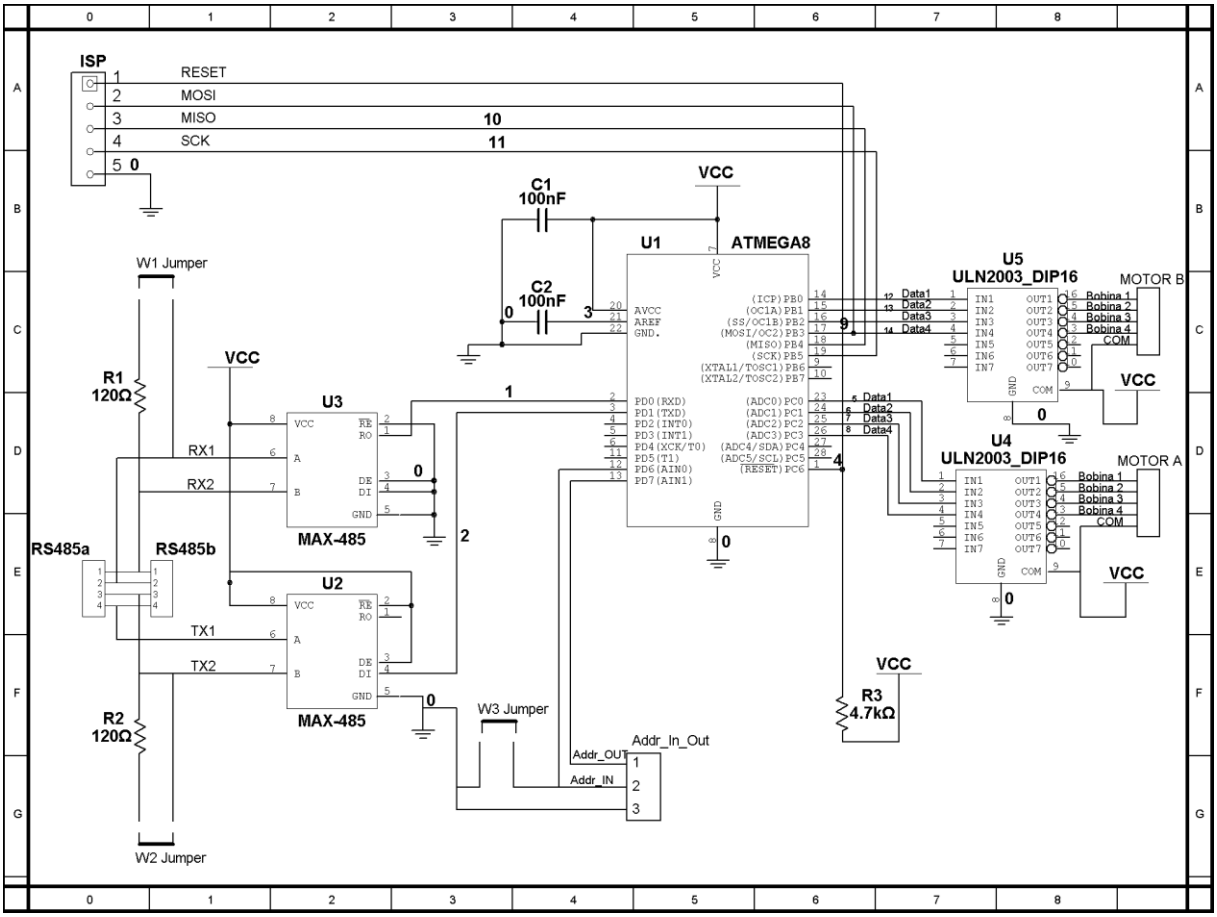


Figura 3.9. Esquemático da placa de controle de motor de passo.

3.2 SOFTWARE

O projeto proposto utiliza uma arquitetura mestre/escravo. Neste caso, o sistema mestre é um computador comum (PC) que utiliza o sistema operacional Windows XP Professional. Para o sistema mestre, o *software* utilizado é baseado no código adaptado, pelo professor Lélio Ribeiro Soares, do sistema PIC-SERVO da Jeffrey Kerr, LLC. Para cada sistema escravo, foi desenvolvido um *software* embarcado específico em linguagem C, baseado no que foi desenvolvido por Thiago Felippe Kurudez Cordeiro para controle de motor DC (CORDEIRO, 2009).

3.2.1 MESTRE

O programa mestre é composto por 2 arquivos: o arquivo principal “main.c” e o arquivo de cabeçalho “mestre.h”.

3.2.1.1 MESTRE.H

Neste arquivo de cabeçalho, são inclusos algumas bibliotecas inicialmente. Em seguida, são estabelecidas várias definições para tornar o código principal mais claro e fácil de ler. A constante do cabeçalho, códigos de comandos e outros valores passam ser referenciados por nomes mais fáceis de serem identificados.

Uma estrutura genérica é definida para armazenar parâmetros advindos das placas correspondentes aos sistemas escravos como, por exemplo, a posição do manche do joystick e o estado de seus botões.

O próximo bloco é composto pelos protótipos das funções utilizadas no arquivo main.c.

A parte final traz as declarações das variáveis globais. O número de módulos a serem conectados no barramento é definido por meio da diretriz #define, o vetor de estruturas com tamanho igual à quantidade de módulos mais 1 é declarado, o endereço de grupo é definido e outras variáveis de uso mais interno ao programa como a porta serial a ser usada são declaradas também.

3.2.1.2 MESTRE.C

Este arquivo corresponde ao código fonte do programa que será executado no sistema mestre (computador) e contém todas as funções de envio de comando e configuração dos sistemas escravos (placas). O usuário deve alterar a função *main()* para que ela chame as funções desejadas e executá-las.

As funções *int SimpleMsgBox(char *msgstr)*, *void ErrorPrinting(BOOL f)* e *int ErrorMsgBox(char *msgstr)* controlam a exibição de mensagens do programa (API do Windows, mais informações em Microsoft). A primeira mostra mensagens genéricas, a segunda controla se as mensagens de erro devem ou não ser exibidas (ao se enviar o valor falso para esta função, a exibição de toda mensagem de erro futura será suprimida) e a terceira mostra mensagens de erro.

A seqüência de funções de *HANDLE SioOpen(char *name, unsigned int baudrate)* até *void ErrorShow(void)* estão relacionadas com a comunicação serial do computador, incluindo a inicialização e finalização da porta de comunicação, envio e recebimento de dados, esvaziamento do buffer de dados recebidos e impressão de possíveis erros

relacionados à porta (API do Windows, mais informações em Messias e Microsoft). Estas funções são internas ao funcionamento do programa e normalmente não são chamadas diretamente pela função *main()*. De modo geral, não precisam ser alteradas com a exceção do tempo de *timeout* da comunicação. Quando se altera o valor do *step rate multiplier* (a constante de tempo que multiplica o tempo base do ciclo de servo), o sistema escravo pode demorar mais a responder.

A função *BOOL GET_RESPONSE(byte ADDR)* obtém e interpreta a resposta recebida do sistema escravo após o envio de um comando. A função consulta a estrutura correspondente ao endereço do sistema escravo e verifica quais bytes de dados são esperados na resposta. A quantidade recebida é verificada se coincide com o esperado e se o *checksum* está correto. Em seguida, o valor das variáveis obtidas são armazenados na estrutura. Esta função é utilizada internamente e, geralmente, não é utilizada diretamente na função *main()*.

Seguem-se então as funções genéricas que não estão associadas a nenhuma placa em específico e as funções correspondentes a cada placa utilizada como sistema escravo. Todas elas exigem o fornecimento do endereço de destino e, no caso das funções mais complexas, alguns parâmetros necessários enviados pelo comando. Todas as funções retornam um número booleano que indica se a transmissão e recepção da comunicação ocorreram da forma esperada. O procedimento para o envio é semelhante para todas essas funções. Primeiro verificam se o byte de configuração (se existir) possui valores adequados. Em seguida, montam o vetor de bytes do comando, iniciando com o cabeçalho que é seguido pelo endereço, comando e a quantidade de bytes de dados em um byte, os bytes de dados e o valor do cálculo do *checksum*. O vetor de bytes é enviado e, se for o caso, uma resposta é recebida, ocorrendo a atualização da estrutura correspondente à placa. Por fim, a função retorna um valor booleano, indicando se o programa foi executado com sucesso. Estas funções que devem ser chamadas via *main()* de acordo com a necessidade do usuário.

A função *void InitVars(int N)* inicia o vetor de estruturas com os valores iniciais apropriados. Ela é chamada internamente, durante o processo de inicialização.

Em seguida, tem-se a função *void SYSTEM_RESET(byte ADDR)*, que realiza o procedimento para reiniciar o dispositivo específico do barramento.

A função *BOOL SYSTEM_INIT(char *portname, byte baudrate, int N)* executa o procedimento de inicialização de todo o sistema, incluindo o endereçamento e definição de *baud rate*.

Por fim, tem-se a função *main()*, a qual o usuário deve modificar para executar as funções desejadas. Deve-se definir o *baud rate* desejado e se as mensagens de erro devem

ser mostradas. Em seguida, a função de inicialização (*BOOL SYSTEM_INIT(char *portname, byte baudrate, int N)*) é chamada. A partir deste ponto, as funções de comando são chamadas de acordo com a aplicação da placa. Depois de terminada a execução da função de comando, a porta de comunicação serial é fechada e o programa é encerrado.

3.2.1.3 FUNÇÕES ESPECÍFICAS DE APLICAÇÕES

As funções específicas para aplicações se enquadram em quatro grupos: da placa do *joystick*, da placa de entrada e saída A/D, da placa de controle de servomotores e da placa de controle de motores de passo (para mais detalhes, ver Tabela 2.3).

Para a placa do *joystick*, há apenas a função *BOOL Jk_READ_JOYSTICK(byte ADDR)* de leitura do estado do *joystick*, que corresponde ao posicionamento do manche e o estado dos botões.

Para placa de entrada e saída de dados A/D, na parte de entrada/saída de dados digitais, tem-se a função *BOOL Io_CONFIG_IO(byte ADDR, byte PINOS)*, que configura as portas digitais como entrada/saída, a função *BOOL Io_READ_IO(byte ADDR)*, que lê os dados dos pinos configurados como entrada, a função *BOOL Io_WRITE_IO(byte ADDR, byte ESCRITA)*, que escreve dados nas portas configuradas como saída. Já na parte de conversão A/D tem-se a função *BOOL Io_CONFIG_ADC(byte ADDR)*, que habilita o conversor A/D do microcontrolador da placa, a função *BOOL Io_READ_ADC(byte ADDR, byte num_channels)*, que define quais canais do conversor AD serão lidos. Na última parte de geração de sinal analógico por meio dos canais PWM, tem-se a função *BOOL Io_WRITE_DAC(byte ADDR, byte Channel_PWM, byte WRITE_PWM1, byte WRITE_PWM2)*, que configura os quantos canais de PWM serão usados e qual o ciclo de trabalho para cada um.

Para a placa de controle de servomotor, tem-se a função *BOOL Sv_CONFIG_SERVO(byte ADDR, byte NUM_PORT, int Freq)*, que configura quantos servomotores serão controlados e qual será a freqüência do PWM, a função *BOOL Sv_SET_POS(byte ADDR, byte PORT, float time)*, que define o posicionamento de um servomotor específico e a função *BOOL Sv_SET_POS_ALL(byte ADDR, float time)*, que define a posição de todos os servomotores controlados pela placa.

Para a placa de controle de motor de passo, tem-se a função *BOOL Sm_STOP_MOTOR(byte ADDR, byte MOTOR)*, que interrompe o movimento do motor de passo, a função *BOOL Sm_START_MOTOR(byte ADDR, byte MOTOR, byte MODO)*, que inicia o movimento do motor de passo de acordo com as configurações fornecidas pelo usuário, a função *BOOL Sm_DESLOC_MOTOR(byte ADDR, byte SENSE, int NUM_STEPS, int VEL)*, que configura os parâmetros do modo deslocamento, a função

BOOL Sm_VELOC_MOTOR(byte ADDR, byte MOTOR, byte SENSE, int MAG), que altera a velocidade de deslocamento do motor de passo, a função *BOOL Sm_TRAPE_MOTOR(byte ADDR, byte SENSE, int NUM_STEP, int Step, int ace)*, que configura os parâmetros para o modo trapezoidal.

Para mais detalhes, consultar o Anexo I.

3.2.2 ESCRAVO

Como o projeto propõe várias placas, ou seja, vários sistemas escravos, cada uma possui um microcontrolador com o seu programa escravo distinto. Entretanto, eles têm em comum as funções de comunicação serial e as funções de processamento do comando, se diferindo a partir da interpretação do comando, que direciona o fluxo do programa para o código correspondente ao da aplicação. Todos eles são compostos de 3 arquivos: um arquivo principal ".c" e dois arquivos de cabeçalho ".h", sendo que um deles traz os protótipos das funções e o outro traz definições de constantes e enumerações.

No arquivo de cabeçalho Variaveis.h, são observados três definições no começo: _TESTE_1MHZ, _TESTE_8MHZ e _USAR_STDIO, que podem ou não estar comentados. As duas primeiros definem o *clock* em que a placa está trabalhando. Por exemplo, caso o microcontrolador esteja configurado para operar com um *clock* de 8 MHz, basta descomentar o #define equivalente. Caso esteja configurado para operar com um *clock* de 16 MHz, basta comentar as duas definições. A terceira definição indica se a biblioteca stdio.h e todas as funções relacionadas devem ser compiladas ou não. Esta biblioteca facilita o *debug*, mas ocupa muita memória de programa.

A próxima definição, *tamanhoBufferSerial*, indica o tamanho em bytes dos *buffers* de recepção e envio da comunicação serial. Quanto maior o *buffer*, menor é a chance de ocorrer *overflow*, mas a quantidade de memória RAM utilizada é maior. É importante notar que se deve escolher valores que sejam potências de 2 que possam ser armazenadas em apenas um byte sem sinal, ou seja, 2, 4, 8, 16, 32, 64 ou 128. Foi escolhido o valor 32 por atender todas as necessidades do projeto.

Em seguida, tem-se um grupo de definições que criam novos nomes para os tipos de variáveis, deixando claro qual o tamanho de cada uma. Um detalhe importante para se observar é que as variáveis usuais não possuem o mesmo tamanho comparadas com as da arquitetura x86. Por exemplo, o tipo de variável *int* possui apenas 16 bits (2 bytes), enquanto a variação *long* possui 32 bits (4 bytes). Estas novas definições dos nomes das variáveis são mais claras e facilitam a portabilidade do programa para o outro hardware. Por exemplo, no programa principal, basta usar *int16* sempre que se precisar de uma variável de

16 bits. Quando o hardware possuir uma convenção diferente de nome e quantidade de bits, basta alterar a definição correspondente.

As duas primeiras enumerações *estadoComunicacao* e *destinoComunicacao* funcionam como variáveis de uma máquina de estados durante a interpretação dos bytes recebidos na comunicação serial. A primeira enumera os possíveis estados que a recepção pode assumir e a segunda armazena o tipo de destinatário dos dados. Por exemplo, a função de recepção pode estar no estado recebendo dados e estes dados serem de outro dispositivo. Nesse caso, a ação tomada pela função é descartar os dados.

O próximo bloco contém definições de constantes para facilitar a leitura do código principal.

O segundo arquivo de cabeçalho contém apenas os protótipos das funções existentes no programa principal. Estes protótipos foram adicionados na mesma ordem em que as funções são definidas no arquivo principal e comentários separam os protótipos em grupos, de forma a facilitar a sua localização durante a leitura do código.

O arquivo principal .c contém todo o algoritmo do software da placa. O arquivo é extenso, por isso sua análise foi feita separando o programa em blocos.

3.2.2.1 FUNÇÃO MAIN

A função *main()* começa com a inicialização de variáveis e de parâmetros de *hardware* por meio da chamada a funções com tal propósito. As portas são configuradas como entrada/saída e o temporizador é configurado para gerar os ciclos de servo.

Em seguida, o programa entra em um laço infinito. Neste laço, ocorre a interpretação de comandos provenientes da serial. Primeiramente, verifica-se a existência de um novo byte advindo do barramento. Se houver, o byte é decodificado e o vetor de comando é montado. Caso o byte recebido seja o último byte de um comando, a função ativa uma variável para sinalizar. É importante notar que somente um byte do *buffer* de dados da serial é obtido para que o programa não trave ao receber um conjunto de dados muito extenso. Além disso, em boa parte do tempo, esta será a única tarefa do laço principal, uma vez que as outras tarefas só ocorrem uma vez a cada ciclo de servo.

Após a recepção de um novo byte, ocorre a verificação se o tempo de ciclo foi atingido. Uma variável é ativada na interrupção de tempo, sempre a cada 512 µs (ou um de seus múltiplos, dependendo da aplicação).

Caso o tempo de ciclo não tenha sido atingido, o laço chega ao fim e é reiniciado. Se tal tempo foi atingido, é ativado um bit que indica o *overrun* do servo no byte auxiliar de estado.

Em seguida, é verificado se o controlador recebeu completamente um novo comando. Se for o caso, verifica-se primeiramente o *checksum* e, se estiver correto, o comando é interpretado. Depois, verifica-se se há a necessidade de uma resposta ao comando e, se for o caso, são enviados ao barramento os bytes de resposta.

3.2.2.2 CONFIGURAÇÕES BÁSICAS DE HARDWARE

As funções de inicialização configuram bits dos registradores de configuração do microcontrolador, habilitando ou desabilitando recursos do mesmo.

A função *void iniciaUSART()* configura a comunicação serial. Por meio dessa função, são escolhidos o valor inicial de 19200 para o *baud rate*, 8 bits de dados, um bit de parada e nenhum de paridade. Habilita-se a comunicação em ambos os sentidos (RXD e TXD). Configura-se de forma que seja gerada uma interrupção sempre que chegar dados novos do barramento. Outra interrupção que é utilizada, mas não é inicialmente ativada, é a de *buffer* de envio vazio. Sempre que houver dados a serem enviados, essa interrupção é ativada e , quando não houver mais dados a serem enviados, ela é desativada. Combinando o uso dessas duas interrupções, a comunicação pode ser feita sem a necessidade de *pooling* paralelamente ao bloco do programa principal.

A função *void iniciaTempoServo()* configura o contador/temporizador 0 do microcontrolador para gerar interrupções de *overflow* na contagem de tempo. O *prescaler* foi configurado de acordo com a aplicação.

A função *void iniciaPortas()* ajusta os pinos como entrada ou saída. Os pinos XTAL1, TXD e o PD7 (ADDR_OUT) foram configurados como saída de dados. Os outros pinos são configurados de acordo com a aplicação envolvida, diferindo conforme a placa controlada.

A função *void iniciaVariaveis()* define os valores iniciais das variáveis globais. O endereço individual é definido como 0x00 e o endereço de grupo é definido como 0xFF.

3.2.2.3 FUNÇÕES DE COMUNICAÇÃO SERIAL

Os comandos enviados pelo sistema mestre são transmitidos por comunicação serial e, por isso, é importante que se evite falhas como um byte de dados sobrescrever o anterior no *buffer* de recepção antes de ser lido, ao mesmo tempo em que a comunicação não exija muito tempo de processamento, pois isso atrapalharia outros processos a serem executados ou que já estejam em andamento.

Para garantir a integridade dos dados recebidos, decidiu-se utilizar uma interrupção tanto no envio quanto na recepção de dados. Dois *buffers*, sendo um para envio e outro para a recepção, servem de interface entre a rotina de interrupção e o programa principal.

No envio, o programa principal envia dados ao *buffer*, com o comando *void escreveDadoSerial(char dado)*. O *buffer* ativa a interrupção ao receber tal dado, chamando a rotina de interrupção se o *buffer* de envio da serial estiver disponível. A rotina de interrupção retira um dado do *buffer* de dados de envio, enquanto o programa é executado paralelamente. Quando o envio de um dado é concluído, a rotina de interrupção é chamada novamente. O processo se repete até que a interrupção seja chamada com o *buffer* de dados vazio, quando ela se desabilita. Ela é habilitada novamente quando o *buffer* de dados volta a ser preenchido.

Na recepção, o processo é reverso apesar de semelhante ao envio. A interrupção está sempre habilitada e ocorre quando um dado novo é disponibilizado no *buffer* de recepção da serial. Ativada a interrupção, o dado é enviado ao *buffer* de dados de recepção, de onde é retirado pelo programa principal em momentos apropriados por meio do comando *char leDadoSerial()* e é processado. Um detalhe importante é que o dado será descartado pela interrupção se o pino ADDR_IN estiver em nível alto, de forma a permitir o processo de endereçamento via software pelo método do *daisy chain*.

Ambos os *buffers* possuem o mesmo funcionamento. É utilizado um vetor, onde os dados são armazenados, e duas variáveis inteiras que armazenam as posições deste vetor (ponteiros). Os dois ponteiros apontam para o início do vetor a princípio. Quando um dado novo é recebido, o ponteiro de fim de *buffer* é incrementado, ao passo que o ponteiro de início de *buffer* é incrementado quando um dado é lido. Existem dados no *buffer* enquanto o ponteiro de fim for maior que o ponteiro de início e o *buffer* está vazio quando os ponteiros se igualam. Após um ponteiro alcançar o fim do vetor, o mesmo volta ao início. Uma vez que os ponteiros somente são incrementados, ou zerados se chegarem ao final do vetor, essa configuração caracteriza uma variação da estrutura de dados denominada de fila circular. Desta forma, desconsiderando o caso do *buffer* receber mais dados do que a sua capacidade máxima (*overflow*) e o caso de se ler um dado de um *buffer* vazio (*underflow*), se um ponteiro difere do outro, existem dados. Se o ponteiro de fim do *buffer* é menor que o de início, na verdade ele está uma volta à frente no círculo e, portanto, está à frente do ponteiro de início do *buffer*.

Para facilitar a programação e a leitura do código, a verificação da presença ou não de dados nos *buffers* foi realizada por meio de macros ao invés de funções, pois a lógica simples não justifica o uso da segunda opção. As macros do *buffer* de leitura e do *buffer* de escrita são *existeDadosBufferRX()* e *existeDadosBufferTX()* respectivamente. Tais macros consistem simplesmente do resultado da comparação dos ponteiros do vetor de dados de

cada *buffer*. A Figura 3.10 mostra como o *buffer* de envio recebe um dado do programa principal e ativa a interrupção de envio, enquanto que a Figura 3.11 mostra a interrupção de envio. A Figura 3.12 mostra a interrupção de recepção, enquanto que a Figura 3.13 mostra como o programa principal obtém um dado do buffer de recebimento.

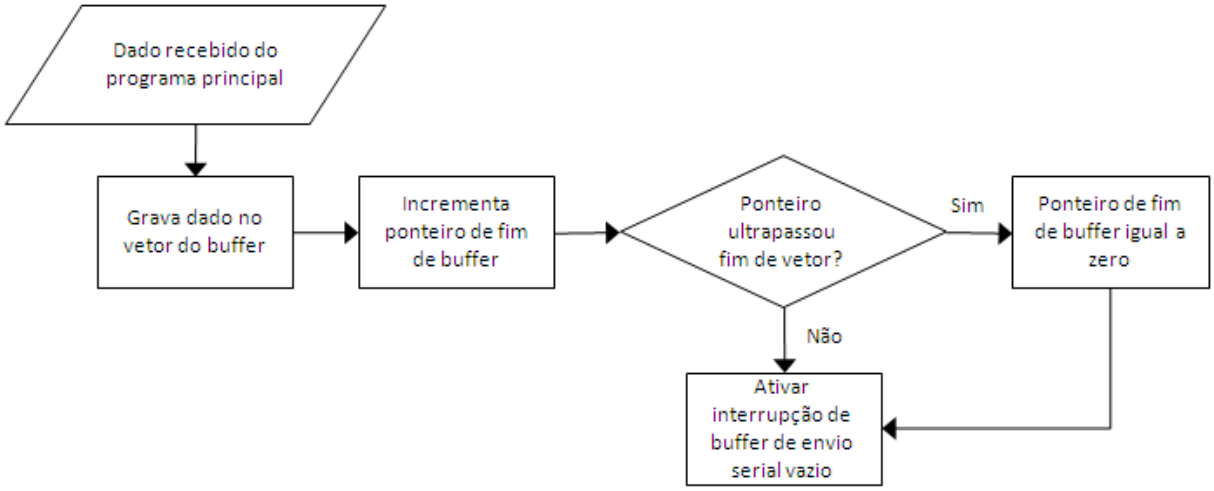


Figura 3.10. Fluxograma do recebimento de um dado pelo *buffer* de envio.

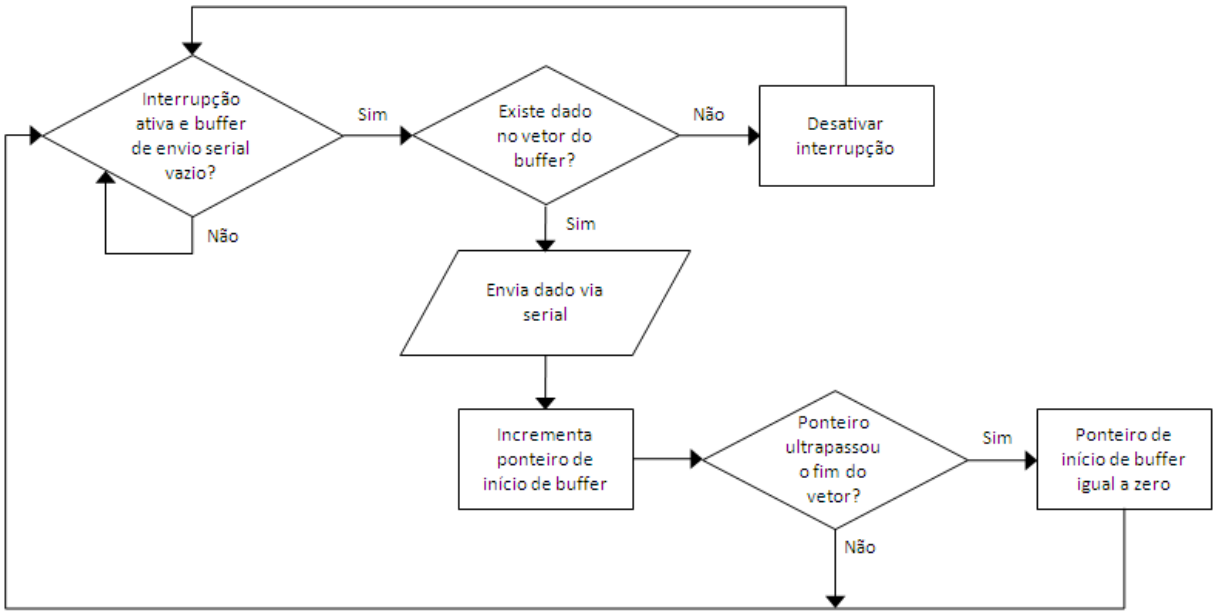


Figura 3.11. Fluxograma da interrupção de envio de dados.

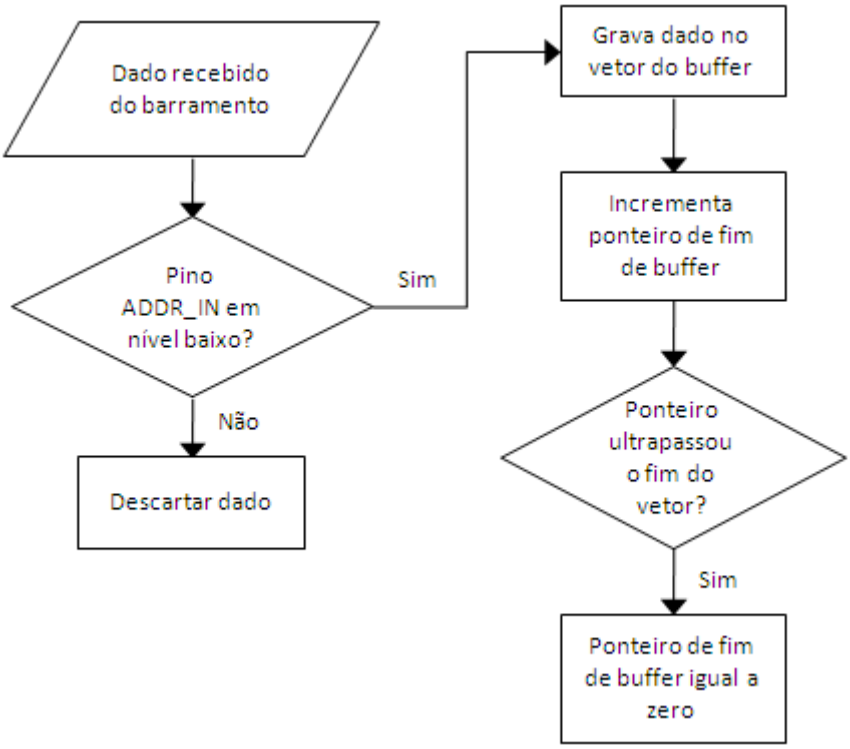


Figura 3.12. Fluxograma da interrupção de recepção de dados.

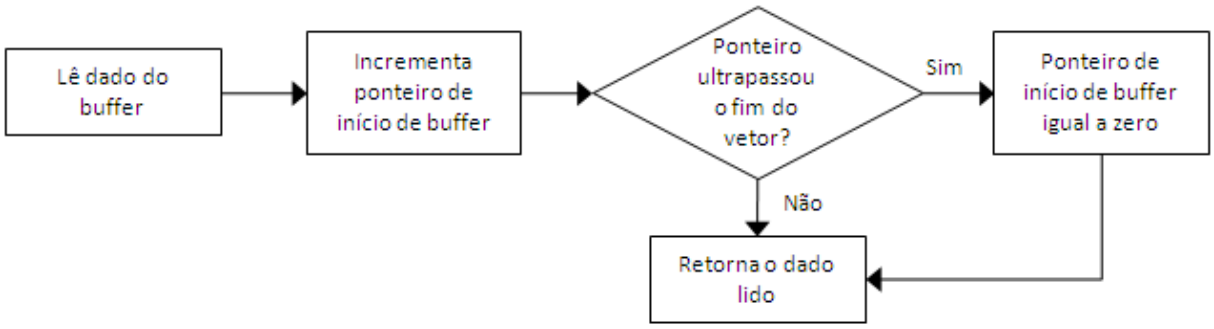


Figura 3.13. Fluxograma do envio de dados do *buffer* de recepção ao programa principal.

3.2.2.4 PROCESSAMENTO DO COMANDO RECEBIDO

As funções envolvidas neste grupo são os responsáveis pela recepção, interpretação e resposta a comandos vindos do barramento.

A função que inicia o processamento do comando recebido pela placa é a *void montaVetorComando(unsigned char dadoRecebido)*. Esta recebe todos os dados presentes no *buffer* de recepção serial e os ordena na forma de um vetor de comando, que começa no cabeçalho e termina no *checksum*. Entretanto, o vetor de bytes que armazena o comando não inclui o cabeçalho e nem o byte de endereçamento, que é armazenado em uma variável separada.

O comportamento desta função é a de uma máquina de estados. Os estados possíveis, que são armazenados em uma variável *enum estadoComunicacao* são: aguardando cabeçalho, aguardando endereço, aguardando comando e aguardando bytes de dados. Outras duas variáveis complementam a definição do estado. A primeira, *enum destinoComunicacao*, define se o endereço de destino do comando é individual, de grupo, o do *reset* global, o endereço de outra placa ou então indefinido. A segunda, *bool recebendoByteOutro*, verifica se o estado é aguardando bytes de dados e o destino é outra placa.

No estado aguardando cabeçalho, a função espera receber o byte de cabeçalho 0xAA. Caso o byte recebido seja diferente, ele é ignorado e o estado permanece o mesmo. Se a função receber um byte de cabeçalho, o sistema vai para o estado aguardando endereço, sendo que o destino ainda é indefinido.

No estado aguardando endereço, o byte que for recebido especifica a placa destino do comando enviado e o estado muda para aguardando comando. Isto ocorre mesmo que o destinatário seja outra placa, uma vez que ainda não se sabe quantos dados serão recebidos até o encerramento da mensagem.

No estado aguardando comando, o byte recebido contém duas informações. A primeira corresponde à quantidade de dados que ainda serão recebidos, não incluindo o *checksum*, e a segunda é o código do comando. O armazenamento dos bytes de dados é feito com o acréscimo do *checksum* e o estado muda para aguardando bytes de dados. Caso o destino seja outra placa, a variável que indica o estado aguardando bytes e o destino como outro é habilitada, gerando um estado diferente do estado aguardando bytes de dados. Se o destino for a própria placa, o comando é armazenado no vetor de comando. O comando é considerado como enderaçado à própria placa em três situações: o endereço do comando é o endereço individual da placa, o endereço de comando é do grupo ao pertence a placa ou o endereço é o endereço de *reset* geral e o comando enviado é o comando é o de *reset*. Se o endereço não for nem o individual, nem o de grupo, mas for o de *reset* geral e o comando for diferente de *reset*, o destino é considerado como sendo outra placa. O endereço de *reset* geral tem menor prioridade sobre o endereço individual e o de grupo, ou seja, só há a verificação se o endereço corresponde ao de *reset* geral se este não for nem o individual e nem o de grupo.

No último estado, correspondente a aguardando bytes de dados, os dados restantes do comando enviado são recebidos, incluindo o *checksum*. Caso o destino seja outra placa, os bytes são descartados logo no início da função. No caso do destino ser a própria placa, os bytes são armazenados no vetor de comando. Ao se verificar que a quantidade de dados recebida corresponde ao definido no estado aguardando comando, a função retorna para o estado aguardando cabeçalho.

A Figura 3.14 ilustra o funcionamento da função por meio de uma máquina de estados.

A função *bool verificaCheckSum()* possui um funcionamento bem simples. Ela realiza a soma de todos os elementos do vetor de comando com exceção do *checksum* e incluindo o endereço ao qual se destina o comando (que fica armazenado em outra variável). Em seguida, tal função realiza a comparação da soma com o valor do *checksum* do vetor de comando e, caso sejam iguais, a função retorna verdadeiro.

A função *void interpretaComando()* decodifica o comando presente no vetor de comando. Um comutador recebe como dado o valor do comando e direciona para o código correspondente.

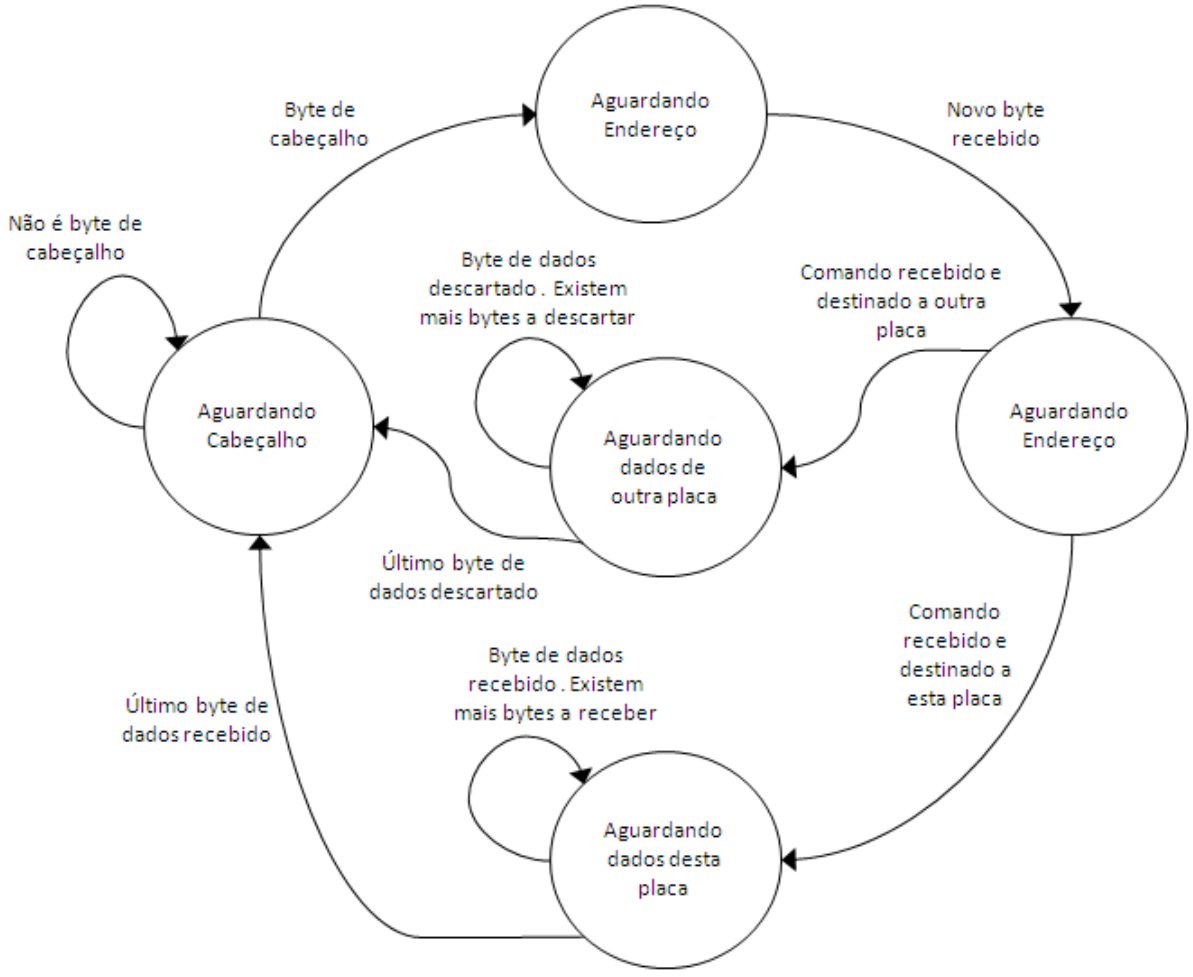


Figura 3.14. Máquina de estados que ilustra o funcionamento da função *montaVetorComando(dadoRecebido)*.

Muitos comandos alteram valores de variáveis e constantes internas do programa. Em alguns casos, aparecem variáveis com mais de um byte. Foram analisadas algumas possibilidades de se juntar os vários bytes do vetor em uma variável. Um dos métodos

testados foi o uso de máscaras *AND* (operação com E lógico) e rotação de bits. Analisando o tamanho do código compilado, verificou-se que o uso de ponteiro para acessar byte a byte uma variável de vários bytes se mostrou bem mais eficiente. Deve-se levar em conta se a arquitetura define o armazenamento da variável do byte menos significativo para o mais significativo. Se for o contrário, é necessário inverter a ordem de acesso da variável, ou seja, do byte final para o inicial.

3.2.2.5 JOYSTICK

No programa embarcado da placa do joystick, a cada ciclo de servo, ocorre a conversão da tensão de saída do joystick para o eixo X e Y, respectivamente, e a leitura do estado dos botões. Como o conversor possui uma resolução de 10 bits, armazena-se o resultado da conversão em duas variáveis de 8 bits, correspondendo à parte alta e à baixa para cada eixo. Assim, são enviadas para o barramento as partes baixas dos valores do eixo X e Y, respectivamente, seguidas de um byte que corresponde à concatenação das partes altas dos valores eixos X e Y e os estados dos botões 1 e 2. Para mais detalhes, ver o programa correspondente contido no CD. A Figura 3.15 mostra o fluxograma do funcionamento do programa desta placa.

3.2.2.6 ENTRADA E SAÍDA DE DADOS A/D

No programa embarcado desta placa, não há muitos detalhes a serem discutidos. A parte de leitura dos canais do conversor A/D é uma extensão do procedimento de leitura da posição dos eixos do *joystick*, envolvendo todos os canais. As saídas PWM (PB1 e PB2) que geram sinais analógicos são configuradas manipulando os contadores/temporizadores do microcontrolador de acordo com parâmetros fornecidos pelo usuário. Quanto às entradas/saídas digitais, são utilizadas máscaras para manipular os 8 pinos disponíveis das portas para tratá-los como uma variável byte. Mais detalhes, ver o programa correspondente à esta placa contido no CD. A Figura 3.16 mostra o fluxograma do funcionamento do programa desta placa.

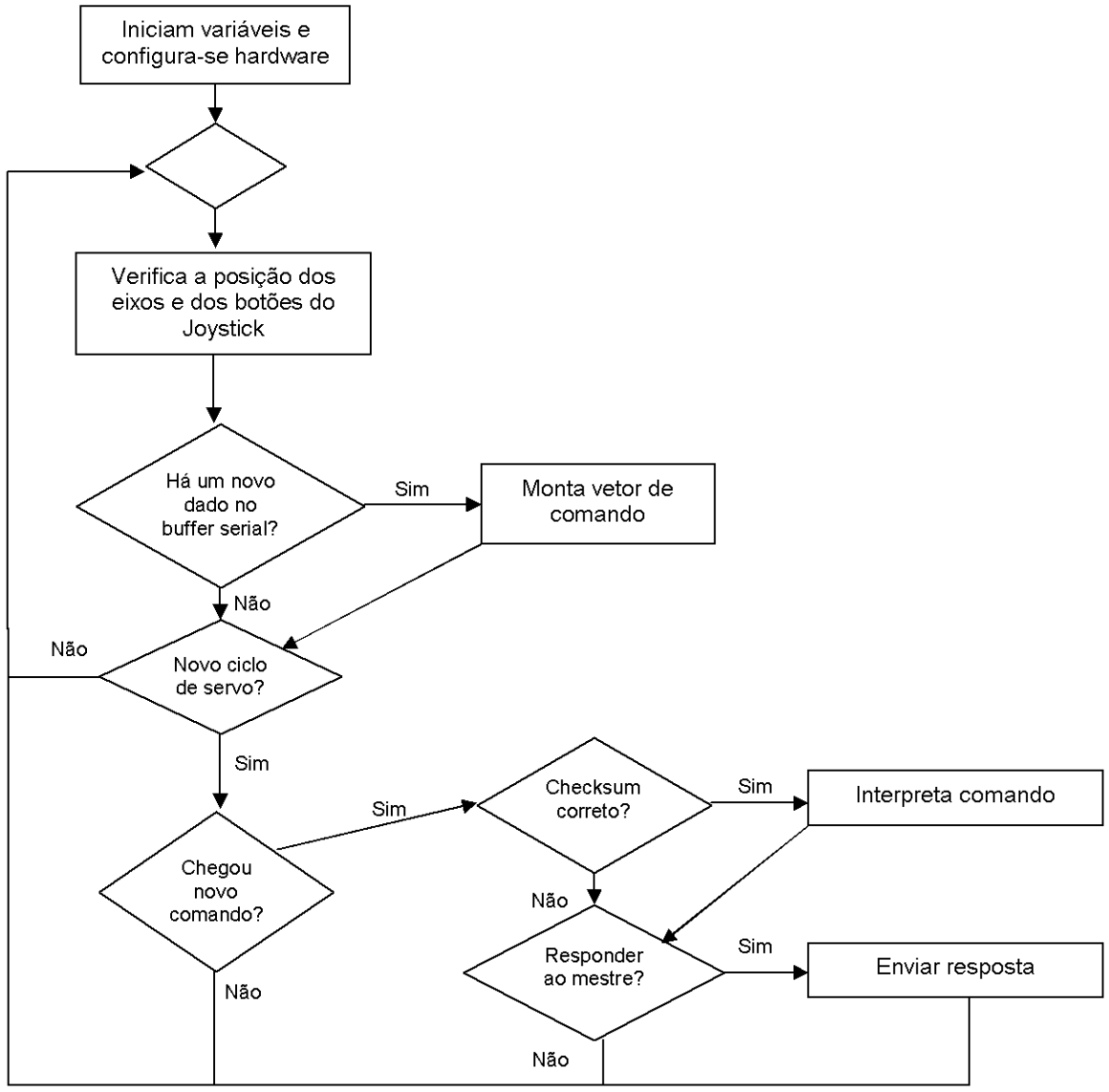


Figura 3.15. Fluxograma que ilustra o funcionamento do programa da placa do *joystick*.

3.2.2.7 SERVOMOTOR

O programa do controlador de servomotor é implementado com o uso de contadores/temporizadores. O usuário escolhe quantos servomotores (até 6) serão utilizados e pode ajustar a posição deles, fornecendo um valor numérico para alterar a largura de pulso do sinal PWM que alimenta os motores. Esse ajuste pode ser feito para cada motor individual ou para todos simultaneamente, dependendo da escolha de função no programa mestre.

Deve-se notar que os sinais PWM são gerados a partir de comparações dos valores fornecidos pelo usuário com os valores dos contadores/temporizadores, que levam as portas para o nível baixo quando o valor do ciclo de trabalho for atingido. O período do pulso deve ser definido pelo usuário, mas normalmente é utilizado um período de 20 ms, que

corresponde a um ciclo de trabalho entre 5% e 10% para uma largura de pulso de 1 a 2 ms, respectivamente (SOCIETY OF ROBOTS, 2010). Portanto, foram utilizadas 6 portas digitais (PC0-PC5) para gerarem os sinais PWM e nenhuma porta com esta função já incluída (PB1, PB2 e PB3) no microcontrolador.

Para ver mais detalhes a respeito do programa mestre, ver Tabela 2.3 e seção 3.2.1.3. O programa da placa se encontra no CD fornecido. A Figura 3.17 mostra o fluxograma do funcionamento do programa desta placa.

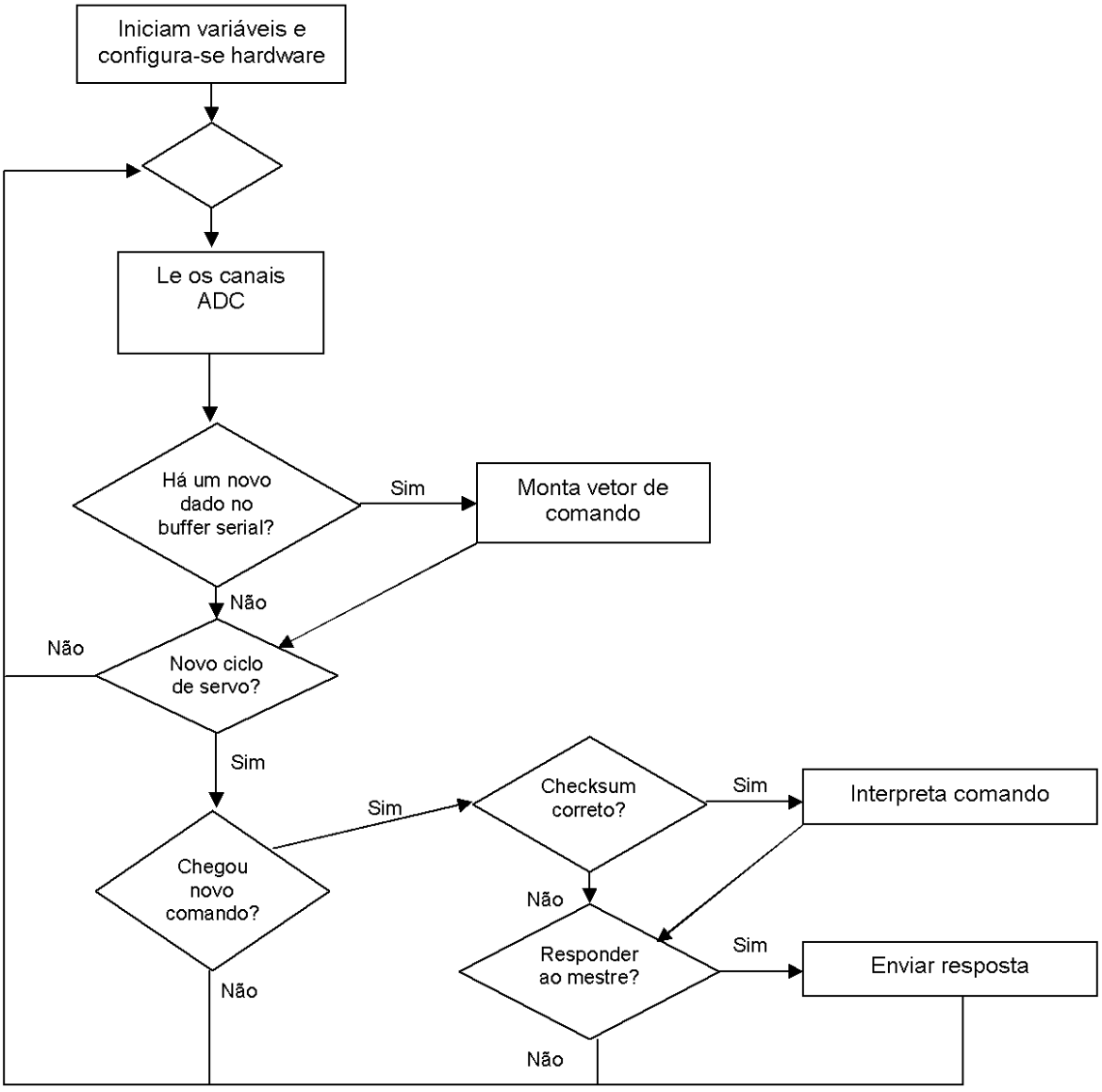


Figura 3.16. Fluxograma que ilustra o funcionamento do programa da placa de entrada e saída A/D.

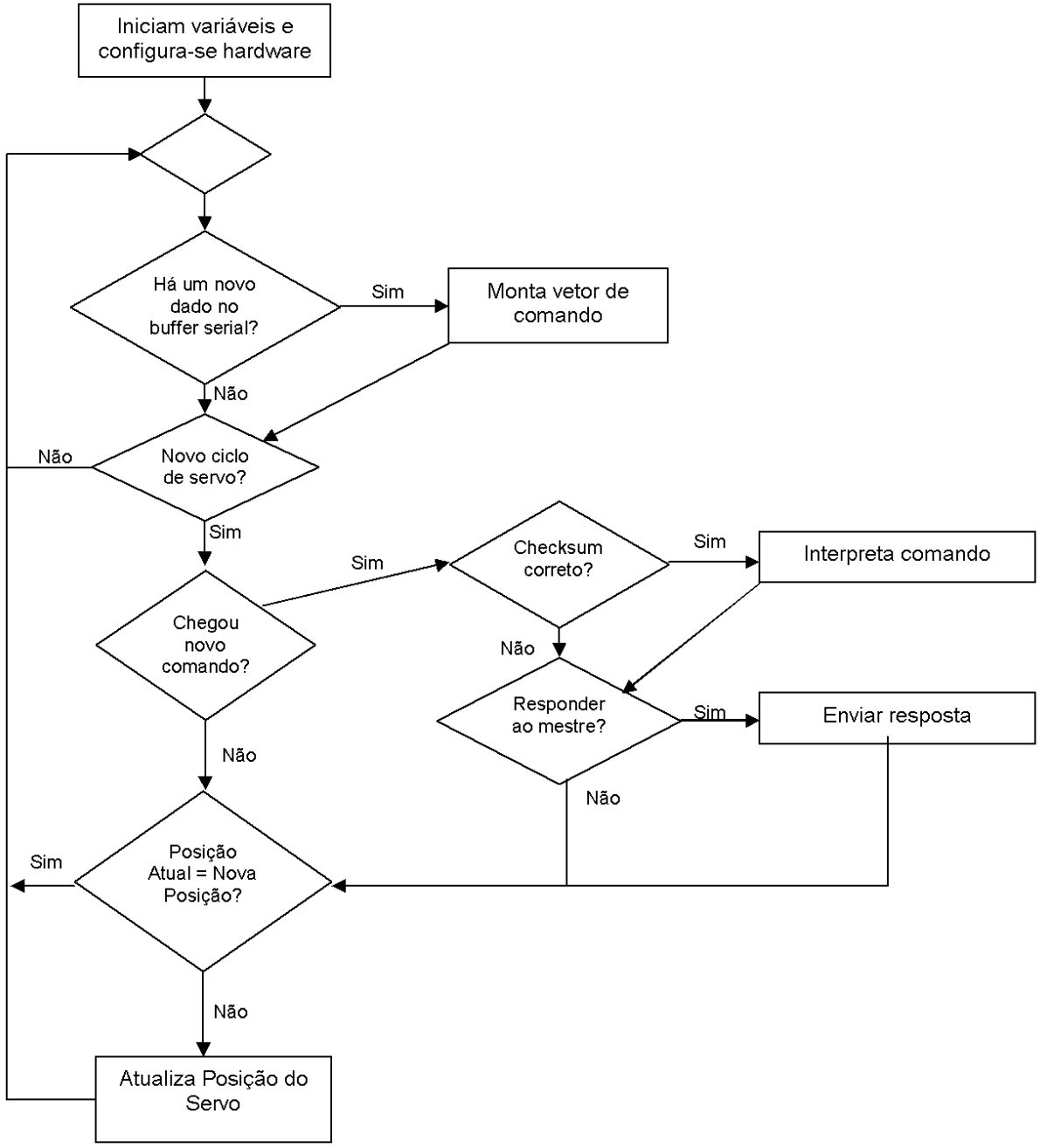


Figura 3.17. Fluxograma que ilustra o funcionamento do programa da placa de controle de servomotor.

3.2.2.8 MOTOR DE PASSO

O programa embarcado da placa de controle de motor de passo disponibiliza várias funcionalidades como iniciar e interromper o movimento do motor, alterar a velocidade do movimento e escolher o modo de operação entre modo deslocamento e modo trapezoidal.

Para o modo deslocamento, o usuário fornece o sentido do movimento, a quantidade de passos e a velocidade desejados. Já no modo trapezoidal, o movimento é realizado com um perfil de velocidade trapezoidal que envolve aceleração e desaceleração. Isto requer

cálculos de velocidade e aceleração pelas equações da cinemática de movimento uniformemente variado (COSTA, 2010). Tais parâmetros são calculados no programa mestre do PC a partir dos dados (quantidade de passos e tempo) fornecidos pelo usuário e são enviados para o microcontrolador da placa que realiza o movimento de acordo com eles. A Figura 3.19 mostra o fluxografo do funcionamento do programa desta placa.

As equações utilizadas para o cálculo dos valores de velocidade e aceleração da função do modo trapezoidal foram baseadas nas encontradas em Costa, 2010. A Figura 3.18 ilustra o deslocamento com perfil de velocidade trapezoidal.

O movimento do motor se inicia a partir do repouso e aceleração constante até chegar a um valor máximo pré-definido, na qual permanece por um certo intervalo de tempo. Em seguida, o motor começa a desacelerar até parar completamente.

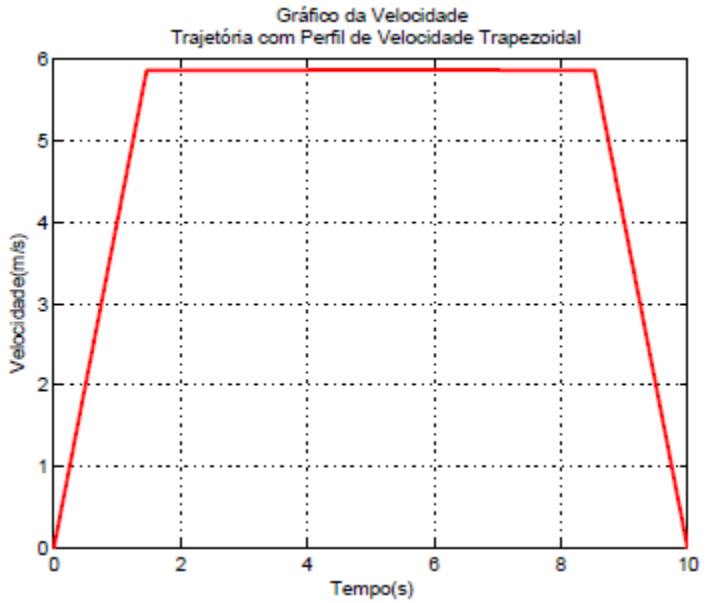


Figura 3.18. Exemplo de deslocamento com perfil de velocidade trapeizodal (COSTA, 2010).

São definidos os parâmetros:

- a – módulo da aceleração constante aplicada ao motor;
- t_a – intervalo de tempo em que o motor acelera positivamente, medido a partir do instante $t = 0$;
- t_d – instante de tempo, medido a partir de $t = 0$, em que o motor inicia a desaceleração até parar, sendo que $t_d = t_f - t_a$;
- t_f – tempo total que o motor leva para completar o movimento;

- S_f – posição final do motor;
- S_0 – posição inicial do motor;
- S_a – posição do motor no qual ele termina a aceleração;
- S_d – posição do motor a partir do qual ele inicia a desaceleração, sendo que $S_d = S_f - S_a$;
- v_0 – velocidade inicial do motor.

Também são definidas as variáveis:

- S – posição do motor em função do tempo;
- v – velocidade do motor em função do tempo;
- t – instante de tempo.

Analisando a Figura 3.18, temos que

$$v = \begin{cases} at, & 0 < t < t_a \\ \dfrac{S_d - S_a}{t_d - t_a}, & t_a < t < t_d \\ -at, & t_d < t < t_f \end{cases} \quad (1)$$

Como a velocidade varia constantemente na aceleração e desaceleração, o movimento se caracteriza como sendo uniformemente variado em ambos os casos. A equação que descreve o movimento durante a aceleração e a desaceleração é dada por

$$S_a = S_0 + v_0 t + \frac{at^2}{2} \quad (2)$$

Como o motor de passo parte do repouso, tem-se que

$$S_a = S_0 + \frac{at^2}{2} \quad (3)$$

Como o perfil de velocidade é simétrico em relação à reta que passa por $t = t_f/2$, obtém-se

$$\begin{cases} S_d = S_f - S_a \\ t_d = t_f - t_a \end{cases} \quad (4)$$

Em $t = t_a$, tem-se que

$$v = \frac{S_f - 2S_a}{t_f - 2t_a} = at_a \tag{5}$$

Susbstituindo a equação (3) na equação (5), obtém-se

$$at_a^2 - at_f t_a + S_f - 2S_0 = 0 \tag{6}$$

O que resulta em

$$t_a = \frac{at_f \pm \sqrt{(at_f)^2 - 4a(S_f - 2S_0)}}{2a} \tag{7}$$

Somente o termo negativo da equação acima é levado em consideração, já que $t_c \leq t_f/2$. Dessa forma, para possibilitar a geração da velocidade com perfil trapezoidal, o módulo da aceleração a deve respeitar a seguinte desigualdade

$$a \geq \frac{4(S_f - 2S_0)}{t_f^2} \tag{8}$$

Para o trecho em que o motor se movimenta com velocidade constante, ou seja, aceleração nula, tem-se que $v = at_a$. Substituindo $a = v/t_a$ na equação (6), obtém-se

$$vt_a - vt_f + S_f - 2S_0 = 0 \tag{9}$$

Portanto,

$$t_a = \left| \frac{vt_f - (S_f - 2S_0)}{v} \right| \tag{10}$$

Como essa velocidade só será constante no intervalo de tempo $0 < t_a < t_f/2$, tem-se como resultado a seguinte restrição para a velocidade

$$\left| \frac{S_f - 2S_0}{t_f} \right| \leq v \leq 2 \left| \frac{S_f - 2S_0}{t_f} \right| \tag{11}$$

Assim, foi utilizado o valor de $1{,}5 \left| (S_f - 2S_0)/t_f \right|$, uma vez que esse valor se encontra dentro da faixa de definida acima, o que possibilita o cálculo dos parâmetros do perfil de velocidade trapezoidal. Além disso, esse valor foi escolhido por estar o mais distante possível dos limites da faixa de operação, minimizando a chance de funcionamento inadequado caso o motor utilizado apresente problemas que comprometam a precisão do movimento.

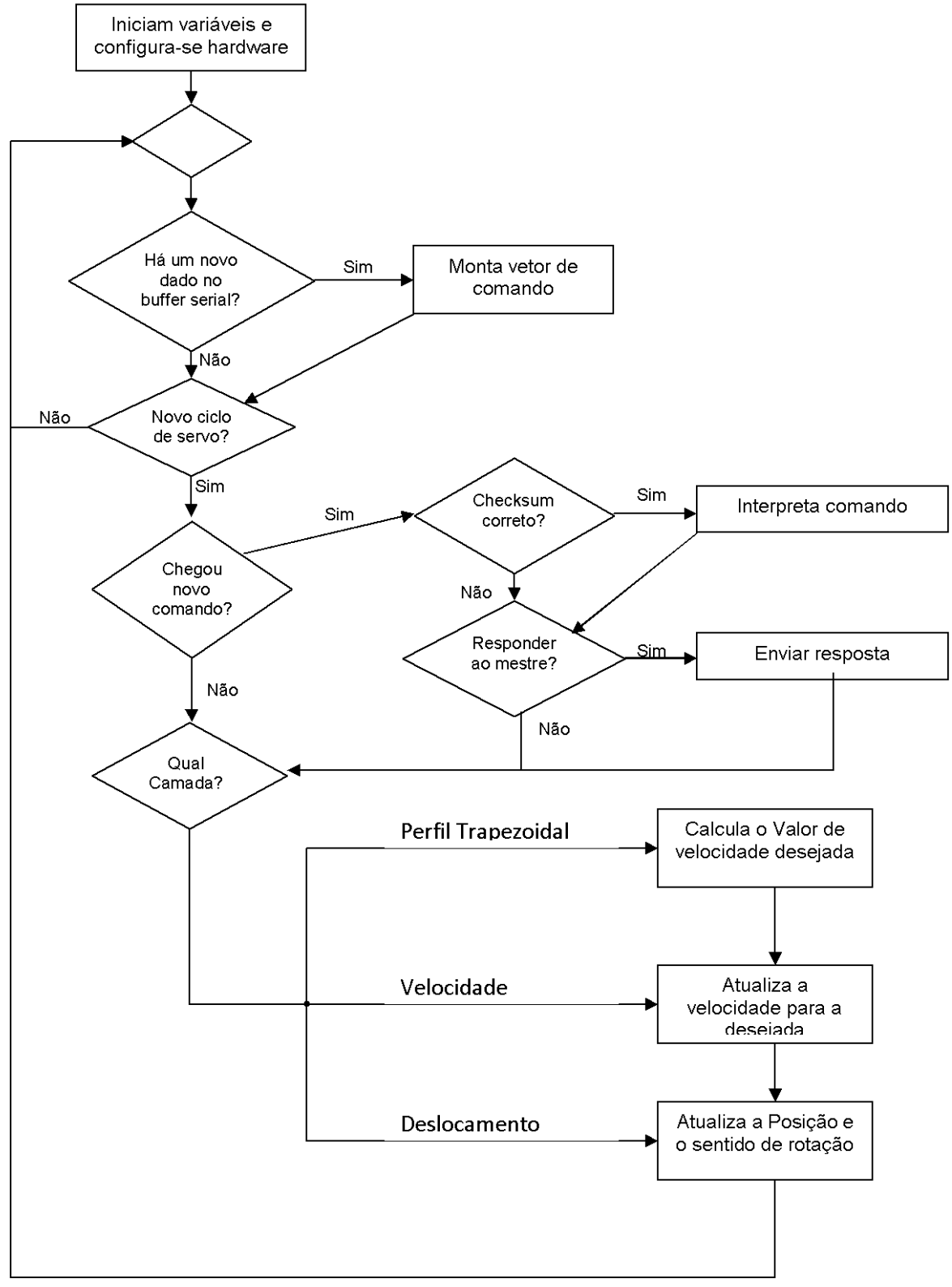


Figura 3.19. Fluxograma que ilustra o funcionamento do programa da placa de controle de motor de passo.

CAPÍTULO 4 – RESULTADOS

A partir do *hardware* e do *software* projetados e desenvolvidos conforme descrito no capítulo 3, vários testes foram realizados para a verificação de todas as funções disponíveis de cada placa.

Com relação à taxa de comunicação, foram testados alguns valores para o *baud rate*. Com o *clock* do microcontrolador ajustado para 8 MHz, foi constatado que as placas deixavam de responder para valores de *baud rate* acima de 19200 bps.

4.1 PLACA DE MONITORAMENTO DO ESTADO DO JOYSTICK

Os testes realizados com a placa do *joystick* mostraram que ela funciona corretamente. O programa mestre foi modificado provisoriamente para mostrar os valores na tela do computador no formato de 0 a 5 V com duas casas decimais de precisão.

Os valores para as posições correspondentes às extremidades de cada eixo variaram de 0 a 5 V como o esperado (depois de fazer a manipulação do valor advindo do conversor AD do microcontrolador). A influência do ruído foi suficientemente minimizada para que o resultado não apresentasse oscilações que o comprometessem.

Os botões funcionaram corretamente, indicando se as chaves estavam abertas (botões soltos) ou fechadas (botões pressionados). Devido à presença de filtros passa-baixa colocados nas saídas do circuito do *joystick* correspondentes aos botões, não houve o efeito *bounce*.

No entanto, notou-se a falta de precisão dos potenciômetros, pois cada vez que o manche era manipulado, os valores de tensão para a posição central eram diferentes de 2,5 V. Esse problema ficou ainda mais evidente quando, mesmo depois de ajustar inúmeras vezes os cursores para se calibrar a posição central, o problema permanecia. Portanto, esse tipo de *joystick* comum de potenciômetro não é recomendado para aplicações que exigem boa precisão.

4.2 PLACA DE ENTRADA E SAÍDA DE DADOS A/D

Os canais de conversão A/D foram testados aproveitando o circuito do *joystick* montado para a sua respectiva placa. O programa principal foi modificado temporariamente para mostrar na tela os valores convertidos no formato de 0 a 5 V com duas casas decimais de precisão. Em suma, os valores obtidos foram os esperados, pois os valores para as posições das extremidades do manche variaram de 0 a 5 V. Todos os 6 canais foram testados de 2 em 2, não havendo nenhum resultado discrepante.

Os pinos digitais foram testados de maneira simples. Configurou-se alguns dos pinos como entrada e o restante como de saída. Então foi fornecido à função de escrita dos pinos um valor binário dentro da faixa disponibilizada pelo número de pinos de saída e se verificou com o multímetro se tais pinos estavam em nível alto/baixo conforme o valor fornecido. Quanto ao restante dos pinos de entrada e saída digital, os pinos foram alimentados de forma aleatória e foi utilizada a função de leitura dos pinos para se conferir os dados lidos pela placa correspondiam ao estado dos pinos de entrada. Não houve erro para nenhum dos valores testados.

O teste para as duas saídas analógicas foi realizado com o fornecimento de vários valores na faixa de 0-255 na função correspondente do programa mestre para ambos canais PWM. O incremento a cada nova chamada foi de 25 unidades, que correspondeu a um incremento na tensão do valor médio do pulso PWM de aproximadamente 0,5 V. Para cada valor, foram realizadas dez medidas para o cálculo da média e do desvio padrão (σ). A Tabela 4.1 lista os valores utilizados no programa mestre e os valores de tensão média para os dois canais PWM, medidos com o osciloscópio e com o multímetro. Os valores obtidos demonstram um pequeno erro percentual.

Tabela 4.1. Valores fornecidos para a função que gera uma tensão analógica para os dois canais simultaneamente.

Valor fornecido (OCR1B/ OCR2)	Valor analógico teórico (V)	Canal 1 (OC1B)				Canal 2 (OC2)			
		Osciloscópio (V)		Multímetro (V)		Osciloscópio (V)		Multímetro (V)	
		Média	σ	Média	σ	Média	σ	Média	σ
0	0	0,035	0,010	0,02	0	0,028	0,013	0,02	0
25	0,49	0,463	0,018	0,51	0	0,472	0,020	0,51	0
50	0,98	0,982	0,008	1,00	0	0,970	0,007	1,00	0
75	1,47	1,463	0,005	1,48	0	1,466	0,005	1,482	0,004
100	1,96	1,957	0,005	1,97	0	1,948	0,004	1,97	0
125	2,45	2,437	0,005	2,46	0	2,438	0,004	2,46	0
150	2,94	2,927	0,006	2,945	0,005	2,926	0,005	2,95	0
175	3,43	3,418	0,004	3,43	0	3,416	0,005	3,43	0
200	3,92	3,905	0,005	3,92	0	3,906	0,009	3,92	0
225	4,41	4,435	0,005	4,405	0,005	4,432	0,008	4,41	0
250	4,90	4,880	0	4,89	0,005	4,884	0,009	4,90	0
255	5,00	4,975	0,005	4,99	0	4,970	0,009	4,99	0

4.3 PLACA DE CONTROLE DE SERVO

O servomotor utilizado para os testes da placa tinha capacidade de rotacionar de 100 a 120 graus. O programa mestre foi modificado provisoriamente para que o servo pudesse varrer todas as posições possíveis. Para isso, foi criado um laço infinito em que o valor da

variável referente ao ciclo de trabalho era incrementado de 0,1 ms de 1 a 2 ms. Como o período do pulso é variável, foram testados os valores da faixa de freqüência para o funcionamento adequado do servomotor constatada na seção 3.1.5. O servomotor respondeu adequadamente.

Todos os 6 canais disponíveis para o controle de servomotores foram testados por meio do osciloscópio e da excitação do servomotor do laboratório. O comportamento para os valores fornecidos foi o esperado.

O teste com o servomotor pelo canal 1 (pino PC0 do microcontrolador) é mostrado no vídeo “servomotor.avi” presente no CD em anexo. O período do pulso utilizado é de 20 ms, correspondendo a uma variação do ciclo de trabalho de 5% a 10% (largura de 1 a 2 ms, respectivamente).

4.4 PLACA DE CONTROLE DE MOTOR DE PASSO

Os testes realizados com a placa do motor de passo envolveram o fornecimento de alguns valores de velocidade e de tempo para o cálculo da aceleração. O movimento foi observado e se constatou que o comportamento foi o esperado. Vale ressaltar que os parâmetros correspondentes à velocidade e à aceleração são calculadas dentro do programa principal do sistema mestre.

O motor da Mitsumi apresentou problemas na desaceleração devido a alguma imperfeição interna, uma vez que o motor da Minibea funcionou normalmente.

A Tabela 4.2 mostra os valores fornecidos e calculados para o modo trapezoidal.

Tabela 4.2. Valores dos testes da placa do motor de passo.

Número de passos (passos)	Tempo total (s)	Velocidade de regime permanente (passos/s)	Aceleração (passos/s^2)
4800	24	250	52
5000	70	100	5
6000	25	300	60
12000	50	300	30

O teste com os parâmetros da segunda linha da tabela, que correspondem aos fornecidos pelos usuário (deslocamento de 5000 passos e tempo total de 70 segundos) e calculados pelo programa do sistema mestre (velocidade de 100 passos/s e aceleração de 5 passos/s^2), é mostrado no vídeo “motor_de_passo.avi” presente no CD em anexo.

CAPÍTULO 5 – CONCLUSÃO

Neste último capítulo são feitas as considerações finais e sugestões para trabalhos futuros que aproveitem o conteúdo deste trabalho.

5.1 CONSIDERAÇÕES FINAIS

Este trabalho cosistiu no desenvolvimento de um protocolo de comunicação sobre um barramento com o padrão RS-485 para aplicações como o monitoramento do estado de um *joystick*, entrada e saída de dados analógicos/digitais, controle de um servomotor e controle de um motor de passo. Todos os módulos das aplicações foram implementadas com sucesso.

O desenvolvimento da placa de monitoramento do estado do *joystick* foi proposto com o intuito de familiarizar o grupo com o equipamento e para dar início ao desenvolvimento do protocolo com uma aplicação mais simples que possuía, a princípio, apenas um comando. Contudo, na fase de testes, ficou evidente a falta de precisão dos potenciômetros, não sendo recomendado para aplicações que exijam precisão.

A placa de entrada e saída de dados A/D foi desenvolvida para incorporar três funcionalidades: entrada/saída de dados digitais, conversor A/D de seis canais e duas saídas analógicas, em que cada sinal é gerado por um PWM. Os resultados se mostraram corretos, embora houve pequena imprecisão nas saídas analógicas.

A placa de controle do servomotor foi desenvolvida para o controle de 6 servomotores por meio de saídas digitais. O usuário tem a opção de escolher quantos servomotores vai utilizar, ajustar o posicionamento individual ou geral deles. O período utilizado para o pulso foi de 20 ms e o ciclo de trabalho varia de 5% a 10% (1 a 2 ms). Na fase de testes, ficou comprovado que o motor rotaciona em uma faixa de 100 a 120 graus.

A placa de controle do motor de passo foi desenvolvida para o controle de 2 motores de passo unipolares de ímã permanente. As funções incorporadas permitem ao usuário escolher o tipo de movimento do motor de passo (modo deslocamento ou modo trapezoidal), iniciar ou parar determinado motor e alterar a velocidade de algum motor durante o seu funcionamento. Durante os testes o motor da Minibea funcionou de acordo com esperado, mas o motor da Mitsumi apresentou problemas na fase de desaceleração do modo trapezoidal provavelmente por alguma imperfeição do mecanismo interno, uma vez que o outro motor de passo não apresentou nenhum problema.

5.2 TRABALHOS FUTUROS

Para trabalhos futuros, uma sugestão seria de expandir a gama de aplicações. Mais dispositivos poderiam ser incluídos ao barramento, uma vez que ele suporta 32 dispositivos e só foram desenvolvidos 4 neste trabalho. Outra sugestão seria de desenvolver placas para o controle dos motores do braço robótico Rhino, utilizando o protocolo desenvolvido neste trabalho. Seria interessante também adaptar o programa mesre para o sistema operacional Linux, pois ele possui um escalonador de processos que trata processos em tempo real comparado ao Windows que não possui essa opção.

O desenvolvimento de projetos futuros, envolvendo o conhecimento fornecido por este trabalho, vai depender das necessidades do laboratório e/ou do interesse do(s) futuro(s) aluno(s) participante(s).

REFERÊNCIAS BIBLIOGRÁFICAS

CORDEIRO, T. F. K. **Controlador de um motor DC sob um barramento de comunicação RS-485 – Algoritmo de controle e protocolo de comunicação**. Julho de 2009. 93p. Trabalho de Graduação – Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília (UnB), Brasília, DF, Brasil.

ATMEL, Corp. ***ATmega8(L) Complete Datasheet***. 2010. Disponível em: <www.atmel.com/dyn/resources/prod_documents/doc2486.pdf>. Acesso em: 31 de julho de 2010.

BORGES, Geovany A.; BO, Antônio Padilha L.; MARTINS, Alexandre S.; COTTA, Leandro C.; FERNANDES, Maurílio; FREITAS, Gabriel; BECKMANN, Ener **Desenvolvimento com Microcontroladores Atmel AVR**. 14 de Junho de 2008. 34p. Departamento de Engenharia Elétrica, Universidade de Brasília (UnB), Brasília, Brasil. Disponível em: <http://www.lara.unb.br/~gaborges/recursos/notas/nt.avr.pdf>. Acesso em: 31 de julho de 2010.

MARTINS, A. S.; BORGES, G. A. **Introdução ao padrão físico RS-485 para comunicação serial**. 5 de Junho de 2006. 13p. Departamento de Engenharia Elétrica, Universidade de Brasília (UnB), Brasília. Brasil. Disponível em: <http://www.lara.unb.br/~gaborges/recursos/notas/nt.rs485.zip>. Acesso em: 31 de julho de 2010.

CANZIAN, Edmur. **Minicurso Comunicação Serial RS-232**. Segundo semestre de 2009. 18p. Disponível em: <http://www.verlab.dcc.ufmg.br/_media/cursos/introrobotica/2009-2/comunicacao_serial.pdf>. Acesso em: 21 de Agosto de 2010.

MAXIM INNOVATION DELIVERED. **APPLICATION NOTE 723 - Selecting and Using RS-232, RS-422, and RS-485 Serial Data Standards**. 29 de Dezembro de 2000. Disponível em: <http://www.maxim-ic.com/app-notes/index.mvp/id/723>. Acesso em: 31/07/2010.

LOGOSOL, Inc. **Logosol AC/DC Intelligent Servo Drive for Coordinated Motion Control LS-174**. Rev. 1.07. 2002. p. 12-40. Disponível em: <http://www.logosolinc.com/download/ls-174.pdf>. Acesso em: 31/07/2010.

TANENBAUM, Andrews S. **Redes de Computadores**. Rio de Janeiro: Editora Campus. 2003. 955p.

ENGDAHL, Tomi. **PC Joystick Interface**. 1996. Disponível em: <http://www.epanorama.net/documents/joystick/pc_joystick.html>. Acesso em: 28/08/2010.

AGUIAR, C. E.; LAUDARES, F. Aquisição de Dados Usando Logo e a Porta de Jogos do PC. **Revista Brasileira de Ensino de Física**. Dezembro de 2001. Instituto de Física, Universidade Federal do Rio de Janeiro (UFRJ): v.23, n.4. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0102-47442001000400003>. Acesso em: 28/08/2010.

HAAG, R.; VEIT, E. A. **Entrada de Joystick**. Agosto de 2001. Instituto de Física, Universidade Federal do Rio Grande do Sul (UFRGS). Disponível em: <http://www.if.ufrgs.br/cref/ntef/aquisicao/joystick.html>. Acesso em: 28/02/2010.

KUPHALDT, Tony R. **Lessons in Electric Circuits, Volume IV – Digital**. 01 de Novembro de 2007. 505f. Disponível em: <http://www.allaboutcircuits.com/pdf/DIGI.pdf>. Acesso em: 28/08/2010.

SOCIETY OF ROBOTS. **ACTUATORS – Servos**, EUA. Data desconhecida. Disponível em: <http://www.societyofrobots.com/actuators_servos.shtml>. Acesso em: 28/08/2010.

BRITES, Felipe Gonçalves; ALMEIDA, Vinicius Puga de. **Motor de Passo**. Julho de 2008. 15p. Escola de Engenharia, Universidade Federal Fluminense, Niterói, Rio de Janeiro. Disponível em: <http://www.telecom.uff.br/pet/petws/downloads/tutoriais/stepmotor/stepmotor2k81119.pdf>. Acesso em: 27/08/2010.

ERICSSON. Stepper Motor Basics. **Circuits Application Note**. Disponível em: <http://www.solarbotics.net/library/pdflib/pdf/motorbas.pdf>. Acesso em: 29/08/2010.

LINEAR TECHNOLOGY. **LTC491 Differential Driver and Receiver Pair**. 1992. Disponível em: <http://www.soselectronic.hu/a_info/resource/c/lin/LTC491.pdf>. Acesso em: 07/09/2010.

MAXIM. **Low Power, Slew-Rate-Limited RS-485/RS-422 Transceivers**. 2009. Disponível em: <http://datasheets.maxim-ic.com/en/ds/MAX1487-MAX491.pdf>. Acesso em: 07/09/2010.

TOSHIBA. **TOSHIBA Bipolar Digital Integrated Circuit**. 2006. Disponível em: <http://www.toshiba.com/taec/components2/Datasheet_Sync/393/22737.pdf>. Acesso em: 07/09/2010.

COSTA, Humberto Cardoso da. **Aplicação de Técnicas de Modelagem e Controle em Sistemas Tipo Ponte Rolante**. 2010. p. 60-64. Tese de Mestrado – Instituto Militar de Engenharia (IME), Rio de Janeiro, Brasil. Disponível em: <http://www.pgee.ime.eb.br/pdf/humberto_costa.pdf>. Acesso em: 07/09/2010.

MESSIAS, Antônio Rogério. **Porta Serial – Controle de Acesso**. Disponível em: <http://www.rogercom.com/PortaSerial/ControleAcesso/ControlePag4.htm>. Acesso em: 04/09/2010.

MICROSOFT. **Functions by Category (Windows)**. Disponível em: <http://msdn.microsoft.com/en-us/library/Aa383686>. Acesso em: 04/09/2010.

ANEXOS

Anexo 1 – Código do sistema mestre (PC).

```
#include <cstdlib>
#include <iostream>

#include "mestre.h"

using namespace std;


// ############################################################################## //
// ############################################################################## //
// ##     FUNCOES DE COMUNICACAO SERIAL                                        ## //
// ############################################################################## //
// ############################################################################## //

//---------------------------------------------------------------------------//
//  Function Name:  SimpleMsgBox (Internal Library Function)                 //
//  Return Value:   0 on failure, non-zero on success                        //
//  Parameters:     msgstr: pointer to null terminated string                //
//  Description:    Displays a simple Windows message box.                   //
//---------------------------------------------------------------------------//
int SimpleMsgBox(char *msgstr)
{
   return MessageBox(NULL, msgstr,"",MB_TASKMODAL | MB_SETFOREGROUND);
}


//---------------------------------------------------------------------------//
//  Function Name:  ErrorPrinting (Internal Library Function)      (*)       //
//  Return Value:   None                                                     //
//  Parameters:     f: 0=disable error printing, 1=enable error printing     //
//  Description:    Controls whether low-level error messages printed by     //
//                  ErrorMsgBox() are printed or suppressed.                 //
//---------------------------------------------------------------------------//
void ErrorPrinting(BOOL f)
{
    printerrors = f;
}


//---------------------------------------------------------------------------//
//  Function Name:  ErrorMsgBox (Internal Library Function)                  //
//  Return Value:   0 on failure or printing disabled, non-zero on success   //
//  Parameters:     msgstr: pointer to null terminated string                //
//  Description:    Displays a Windows message box for error messages under  //
//                  display control using ErrorPrinting().                   //
//---------------------------------------------------------------------------//
int ErrorMsgBox(char *msgstr)
{
   if (printerrors)
     return MessageBox(NULL, msgstr,"",MB_TASKMODAL | MB_SETFOREGROUND);
   else return(0);
}


//---------------------------------------------------------------------------//
//  Function Name:  SioOpen (Internal Library Function)                      //
//  Return Value:   Returns a handle to a COM port stream                    //
//  Parameters:     portname:  name of COM port ("COMn:", where n=1-8)       //
//                  baudrate: 9600,19200,38400,57600,115200,230400           //
//  Description:    Opens a COM port at the specified baud rate.  Set up     //
//                  read and write timeouts.                                 //
//---------------------------------------------------------------------------//
HANDLE SioOpen(char *name, unsigned int baudrate)
{
      BOOL RetStat;
      COMMCONFIG cc;
      COMMTIMEOUTS ct;
      HANDLE ComHandle;
      DWORD winrate;
      char msgstr[50];

      //Open COM port as a file
      ComHandle = CreateFile(name, GENERIC_READ | GENERIC_WRITE ,0, NULL,
```

```
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL , NULL);
while (TRUE)
{
    if (ComHandle == INVALID_HANDLE_VALUE)
          {
      sprintf(msgstr,"%s failed to open",name);
            ErrorMsgBox(msgstr);
      break;
    }

    switch (baudrate)
    {
                case 9600:          winrate = CBR_9600; break;
                case 19200:         winrate = CBR_19200; break;
                case 38400:         winrate = CBR_38400; break;
                case 57600:         winrate = CBR_57600; break;
                case 115200:        winrate = CBR_115200; break;
                case 230400:        winrate = 230400; break;
              default:              ErrorMsgBox("Baud rate not supported - using default of 19200");
                                          winrate = CBR_19200;
    }

    //Fill in COM port config. structure & set config.
    cc.dwSize = sizeof(DCB) + sizeof(WCHAR) + 20;
    cc.wVersion = 1;

    cc.dcb.DCBlength = sizeof(DCB);
    cc.dcb.BaudRate = winrate;
    cc.dcb.fBinary = 1;
    cc.dcb.fParity = 0;
    cc.dcb.fOutxCtsFlow = 0;
    cc.dcb.fOutxDsrFlow = 0;
    cc.dcb.fDtrControl = DTR_CONTROL_DISABLE;
    cc.dcb.fDsrSensitivity = 0;
    cc.dcb.fTXContinueOnXoff = 0;
    cc.dcb.fOutX = 0;
    cc.dcb.fInX = 0;
    cc.dcb.fErrorChar = 0;
    cc.dcb.fNull = 0;
    cc.dcb.fRtsControl = RTS_CONTROL_DISABLE;
    cc.dcb.fAbortOnError = 0;
    cc.dcb.XonLim = 100;
    cc.dcb.XoffLim = 100;
    cc.dcb.ByteSize = 8;
    cc.dcb.Parity = NOPARITY;
    cc.dcb.StopBits = ONESTOPBIT;
    cc.dcb.XonChar = 'x';
    cc.dcb.XoffChar = 'y';
    cc.dcb.ErrorChar = 0;
    cc.dcb.EofChar = 0;
    cc.dcb.EvtChar = 0;

    cc.dwProviderSubType = PST_RS232;
    cc.dwProviderOffset = 0;
    cc.dwProviderSize = 0;

    RetStat = SetCommConfig(ComHandle, &cc, sizeof(cc));
    if (RetStat == 0)
    {
      ErrorMsgBox("Failed to set COMM configuration");
      break;
    }

    //Set read/write timeout values for the file
    ct.ReadIntervalTimeout = 0;                    //ignore interval timing
    ct.ReadTotalTimeoutMultiplier = 2; //2 msec per char
    ct.ReadTotalTimeoutConstant = 50;  //plus add'l 50 msec
    ct.WriteTotalTimeoutMultiplier = 2;//Set max time per char written
    ct.WriteTotalTimeoutConstant = 50;         //plus additional time

    RetStat = SetCommTimeouts(ComHandle, &ct);
    if (RetStat == 0)
    {
      ErrorMsgBox("Failed to set Comm timeouts");
      break;
    }

    break;
}
```

```
    return(ComHandle);
}


//-------------------------------------------------------------------------//
//  Function Name:  SioChangeBaud (Internal Library Function)              //
//  Return Value:   0=Fail, 1=Success                                      //
//  Parameters:     ComPort: COM port handle                               //
//                  baudrate: 9600,19200,38400,57600,115200,230400         //
//  Description:    Change the baud rate to the specified values.          //
//-------------------------------------------------------------------------//
BOOL SioChangeBaud(HANDLE ComPort, unsigned int baudrate)
{
    BOOL RetStat;
    DWORD winrate;
    DCB cs;

    RetStat = GetCommState(ComPort, &cs);
    if (RetStat == false) return RetStat;
    switch (baudrate)
    {
        case 9600:          winrate = CBR_9600; break;
        case 19200:         winrate = CBR_19200; break;
        case 38400:         winrate = CBR_38400; break;
        case 57600:         winrate = CBR_57600; break;
        case 115200:        winrate = CBR_115200; break;
        case 230400:        winrate = 230400; break;
        default:            ErrorMsgBox("Baud rate not supported");
                    return false;
    }
    cs.BaudRate = winrate;
    RetStat = SetCommState(ComPort, &cs);
    if (RetStat == false) return RetStat;
    return true;
}


//-------------------------------------------------------------------------//
//  Function Name:  SioPutChars (Internal Library Function)                //
//  Return Value:   0=Fail, 1=Success                                      //
//  Parameters:     ComPort: COM port handle                               //
//                  stuff: array containing character data to send         //
//                  n: number of characters to send                        //
//  Description:    Write out n chars to the ComPort, returns only after   //
//                  chars have been sent.                                  //
//-------------------------------------------------------------------------//
BOOL SioPutChars(HANDLE ComPort, char *stuff, int n)
{
    BOOL RetStat;
    DWORD nums;

    RetStat = WriteFile(ComPort, stuff,n, &nums, NULL);
    if (RetStat == 0) ErrorMsgBox("SioPutChars failed");
    return RetStat;
}


//-------------------------------------------------------------------------//
//  Function Name:  SioGetChars (Internal Library Function)                //
//  Return Value:   Returns the number of characters actually read         //
//  Parameters:     ComPort: COM port handle                               //
//                  stuff: array to store characters read                  //
//                  n: number of characters to read                        //
//  Description:    Read n chars into the array stuff.                     //
//-------------------------------------------------------------------------//
DWORD SioGetChars(HANDLE ComPort, char *stuff, int n)
{
     BOOL RetStat;
     DWORD numread;

     RetStat = ReadFile(ComPort, stuff, n, &numread, NULL);
     if (RetStat == 0) ErrorMsgBox("SioReadChars failed");
     return numread;
}


//-------------------------------------------------------------------------//
//  Function Name:  SioTest (Internal Library Function)                    //
```

```
//  Return Value:   Returns the number of characters in ComPort's input buf  //
//  Parameters:     ComPort: COM port handle                                  //
//  Description:    Returns the number of chars in a port's input buffer.    //
//-------------------------------------------------------------------------//
DWORD SioTest(HANDLE ComPort)
{
     COMSTAT cs;
     DWORD Errors;
     BOOL RetStat;

     RetStat = ClearCommError(ComPort, &Errors, &cs);
     if (RetStat == 0) ErrorMsgBox("SioTest failed");
     return cs.cbInQue;
}


//-------------------------------------------------------------------------//
//  Function Name:  SioClrInBuf (Internal Library Function)                //
//  Return Value:   0=Fail, 1=Success                                     //
//  Parameters:     ComPort: COM port handle                                //
//  Description:    Purge all chars from a port's input buffer.            //
//-------------------------------------------------------------------------//
BOOL SioClrInbuf(HANDLE ComPort)
{
    BOOL RetStat;

    RetStat = PurgeComm(ComPort, PURGE_RXCLEAR);
    if (RetStat == 0) ErrorMsgBox("SioClrInbuf failed");
    return RetStat;
}


//-------------------------------------------------------------------------//
//  Function Name:  SioClose (Internal Library Function)                   //
//  Return Value:   0=Fail, 1=Success                                     //
//  Parameters:     ComPort: COM port handle                                //
//  Description:    Close a previously opened COM port.                    //
//-------------------------------------------------------------------------//
BOOL SioClose(HANDLE ComPort)
{
    return(CloseHandle(ComPort));
}
//-------------------------------------------------------------------------




//-------------------------------------------------------------------------//
//  Function Name:  ErrorShow  (*)                                        //
//  Return Value:   none                                                  //
//  Parameters:     none                                                  //
//  Description:    Show the error on communication link                   //
//-------------------------------------------------------------------------//
void ErrorShow(void)
{
    char lastError[1024];

    FormatMessage(
              FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
              NULL,
              GetLastError(),
              MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
              lastError,
              1024,
              NULL);
    printf(" \n\n Erro de comunicacao = %s \n\n",lastError);

}


// ########################################################################## //
// ########################################################################## //
// ##     FUNCOES DE COMANDO PICSERVO                                      ## //
// ########################################################################## //
// ########################################################################## //

// ========================================================================== //
// GET_RESPONSE:
// GET STATUS BYTE AND ADDITIONAL BYTES (IF CONFIGURED)
```

```
// ADDR: Address of the module that responds
//
// Returns true or false
// ========================================================================= //
BOOL GET_RESPONSE(byte ADDR) //Testado OK
{
   int     k;
   DWORD   nbytes;
   byte    cksum;

   if (ADDR>=0x80)
   {
      ADDR = Group_Leader;        // Group Leader that returns status
   }

   k = 2;                                  // from 2 to 19 bytes

   if (Module[ADDR].SDEF & SEND_ID_VER){k = k+2;}
   if (Module[ADDR].SDEF & SEND_JOYSTICK){k = k+3;}
   if (Module[ADDR].SDEF & SEND_IO){k = k+2;}
   if (Module[ADDR].SDEF & SEND_PINOS){k = k+2;}
   if (Module[ADDR].SDEF & SendCanais)
   {
     switch(num_canais)
     {
        case 0x01:
            k = k+2;
            break;
        case 0x02:
            k = k+3;
            break;
        case 0x03:
            k = k+4;
            break;
        case 0x04:
            k = k+5;
            break;
        case 0x05:
            k = k+7;
            break;
        case 0x06:
            k = k+8;
            break;
     }
   }

   nbytes = SioGetChars(Com_Port, (char *)comm_string, k);

   if (nbytes != k)                        // No. of bytes test
   {
      SimpleMsgBox("Error on SioGetChars!");
      return false;
   }

   cksum = 0;
   for (k=0; k<=nbytes-2; k++)
   {
      cksum = cksum + comm_string[k];
   }
   if (cksum != comm_string[nbytes-1])          // Checksum test
   {
      SimpleMsgBox("Checksum Error on Module Response!");
      return false;
   }

   k = 0;
   Module[ADDR].STAT = comm_string[k];
   k = k + 1;

   if (Module[ADDR].STAT & COM_ERROR)
   {
      SimpleMsgBox("Checksum Error on Module Command!");
      return false;
   }
}

// ========================================================================= //
// Sv_RESET_POSITION:
// SETS POSITION COUNTER TO ZERO OR RELATIVE POSITION
```

```
// ADDR: Address of a module or Address group
// MODE: ZERO_POS or REL_POS
// Returns true or false
// ========================================================================= //
 BOOL Sv_RESET_POSITION(byte ADDR, byte MODE)
{

}

// ========================================================================= //
// Mo_SET_ADDRESS:
// SETS INDIVIDUAL AND GROUP ADDRESS OF A MODULE
// ADDR: Current address
// ADDR_i and ADDR_g: New individual and group address of a module
// Returns true or false
// ========================================================================= //
BOOL Mo_SET_ADDRESS(byte ADDR, byte ADDR_i, byte ADDR_g)
{
   byte cksum;
   BOOL ckcomm;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x20 | SET_ADDRESS);
   cksum = cksum + comm_string[2];
   comm_string[3] = ADDR_i;
   cksum = cksum + comm_string[3];
   if (Module[ADDR].LEADER) ADDR_g = ADDR_g & 0x7F;
   comm_string[4] = ADDR_g;
   cksum = cksum + comm_string[4];
   comm_string[5] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 6);
   Sleep(100);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR_i);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }
}

// ========================================================================= //
// Sv_SET_STATUS:
// DEFINES ADDITIONAL STATUS DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================= //
BOOL Sv_SET_STATUS(byte ADDR, byte StatusItens)
{
   byte cksum;
   BOOL ckcomm;

   Module[ADDR].SDEF = StatusItens;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x10 | SET_STATUS);
   cksum = cksum + comm_string[2];
   comm_string[3] = StatusItens;
   cksum = cksum + comm_string[3];
   comm_string[4] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
   Sleep(100);
```

```
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}

// ========================================================================== //
// Sv_READ_STATUS:
// READS CONFIGURED ADDITIONAL STATUS DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sv_READ_STATUS(byte ADDR, byte StatusItens) // Testado OK
{
    byte cksum;
    BOOL ckcomm;
    byte temp;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x10 | READ_STATUS);
    cksum = cksum + comm_string[2];
    comm_string[3] = StatusItens;
    cksum = cksum + comm_string[3];
    comm_string[4] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *) comm_string, 5);
    Sleep(100);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }


    temp = Module[ADDR].SDEF;
    Module[ADDR].SDEF = StatusItens;
    ckcomm = GET_RESPONSE(ADDR);
    Module[ADDR].SDEF = temp;
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}


// ========================================================================== //
// Sv_IO_CONTROL:
// SETS THE DIRECTION AND VALUES OF THE LIMIT PINS
// ADDR: Address of a module or Address group
// MODE: I/O mode
//
// Returns true or false
// ========================================================================== //
BOOL Sv_IO_CONTROL(byte ADDR, byte MODE)
{

}

// ========================================================================== //
// Mo_SET_BAUD:
// SETS THE BAUD RATE (Group command only)
// ADDR: Group address
// BR: Baud rate (PB19200, PB57600 e PB115200)
```

```
//
// Returns true or false
// ========================================================================== //
BOOL Mo_SET_BAUD(byte ADDR, byte BR) // Testado OK
{
   byte           cksum;
   unsigned int    br_com;

   switch (BR)
   {
      case PB19200:   br_com = 19200; break;
      case PB57600:   br_com = 57600; break;
      case PB115200:  br_com = 115200; break;
      default :       br_com = 19200; break;
   }

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = 0x10 | SET_BAUD;
   cksum = cksum + comm_string[2];
   comm_string[3] = BR;
   cksum = cksum + comm_string[3];
   comm_string[4] = cksum;
   SioPutChars(Com_Port, (char *) comm_string, 5);
   Sleep(100);
   SioChangeBaud(Com_Port, br_com);
   SioClrInbuf(Com_Port);
   Sleep(100);

   return true; // No status byte is read
}

// ========================================================================== //
// Sv_CLEAR_BITS:
// CLEARS THE STICK STATUS BITS
// ADDR: Address (individual or group)
//
// Returns true or false
// ========================================================================== //
BOOL Sv_CLEAR_BITS(byte ADDR)
{
   byte cksum;
   BOOL ckcomm;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x00 | CLEAR_BITS);
   cksum = cksum + comm_string[2];
   comm_string[3] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *) comm_string, 4);
   Sleep(100);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}

// ========================================================================== //
// Mo_NO_OP:
// NO OPERATION BUT DEFINED STATUS DATA ARE RETURNED
// ADDR: Address (individual or group)
//
// Returns true or false
```

```
// ========================================================================== //
BOOL Mo_NO_OP(byte ADDR)
{
   byte cksum;
   BOOL ckcomm;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x00 | NO_OP);
   cksum = cksum + comm_string[2];
   comm_string[3] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *) comm_string, 4);
   Sleep(100);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}

// ========================================================================== //
// Mo_HARD_RESET:
// RESETS THE CONTROLLER TO ITS POWER-UP STATE
// ADDR: Address (individual or group)
//
// Returns true or false
// ========================================================================== //
BOOL Mo_HARD_RESET(byte ADDR) // Testado OK
{
   byte cksum;
   BOOL ckcomm;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x00 | HARD_RESET);
   cksum = cksum + comm_string[2];
   comm_string[3] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *) comm_string, 4);
   Sleep(100);
   if ((ckcomm == false) && printerrors)
   {
      SimpleMsgBox("Hard Reset: Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte

   ckcomm = GET_RESPONSE(ADDR);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }
}


// ########################################################################### //
// ########################################################################### //
// ##     FUNCÃO DE COMANDO MODULO JOYSTICK                                 ## //
// ########################################################################### //
// ########################################################################### //
```

```
// ========================================================================== //
// Jk_READ_JOYSTICK:
// READS JOYSTICK STATUS DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Jk_READ_JOYSTICK(byte ADDR)
{
    byte cksum;
    BOOL ckcomm;
    byte chave1;
    byte chave2;
    float eixoX;
    float eixoY;
    Module[ADDR].SDEF = SEND_JOYSTICK;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x10 | READ_JOYSTICK);
    cksum = cksum + comm_string[2];
    comm_string[3] = SEND_JOYSTICK;
    cksum = cksum + comm_string[3];
    comm_string[4] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
    Sleep(100);
//    system("PAUSE");
    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }

    eixoX =((comm_string[3] & 0x30)<<4)+ comm_string[1];
    eixoY =((comm_string[3] & 0x0C)<<6)+ comm_string[2];

    chave1 = (comm_string[3] & 0x01);
    chave1 = (~chave1) & 0x01;
    chave2 = ((comm_string[3] & 0x02)>>1);
    chave2 = (~chave2) & 0x01;

    printf("Eixo x: %.2f\n", eixoX);
    printf("Eixo y: %.2f\n", eixoY);
    printf("Chave 1: %x\n",chave1);
    printf("Chave 2: %x\n",chave2);


    Module[ADDR].STAT = comm_string[0];
    Module[ADDR].EIXO_X = eixoX;
    Module[ADDR].EIXO_Y = eixoY;
    Module[ADDR].CHAVE1 = chave1;
    Module[ADDR].CHAVE2 = chave2;

}

// ########################################################################### //
// ########################################################################### //
// ##     FUNCOES DO MODULO ENTRADA E SAÍDA DE DADOS                    ## //
// ########################################################################### //
// ########################################################################### //

// ========================================================================== //
// Io_CONFIG_IO:
// CONFIGS PINS AS DATA INPUT/OUTPUT
// ADDR: Address
```

```
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_CONFIG_IO(byte ADDR, byte PINOS)
{

   byte cksum;
   BOOL ckcomm;

   Module[ADDR].SDEF = SEND_PINOS;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x10 | CONFIG_IO);
   cksum = cksum + comm_string[2];
   comm_string[3] = PINOS;
   cksum = cksum + comm_string[3];
   comm_string[4] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
   Sleep(100);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR);

   Mask_D = comm_string[1];
   Mask_B = comm_string[2];

   printf("MASK_D: %x\n", comm_string[1]);
   printf("MASK_B: %x\n", comm_string[2]);


   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}

// ========================================================================== //
// Io_READ_IO:
// READS IO DATA FROM INPUT PINS
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_READ_IO(byte ADDR)
{
   byte cksum;
   BOOL ckcomm;
   byte LeituraIO,LeituraWR,LeituraAux;

   Module[ADDR].SDEF = SEND_IO;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x10 | READ_IO);
   cksum = cksum + comm_string[2];
   comm_string[3] = SEND_IO;
   cksum = cksum + comm_string[3];
   comm_string[4] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
   Sleep(100);

   if (ckcomm == false)
   {
```

```
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }

    LeituraIO = (comm_string[1]& 0x3C) & ((~Mask_D));
    LeituraAux = (comm_string[2] & 0x30 & (~Mask_B))>>2 | ((comm_string[2] & 0x03) & (~Mask_B));
    LeituraWR = LeituraIO;
    LeituraIO = (LeituraIO<<2) | LeituraAux;
    printf("leitura %x",LeituraIO);
    system("PAUSE");
}

// ========================================================================== //
// Io_WRITE_IO:
// WRITES IO DATA TO OUTPUT PINS
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_WRITE_IO(byte ADDR, byte ESCRITA)
{
    byte cksum;
    BOOL ckcomm;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x10 | WRITE_IO);
    cksum = cksum + comm_string[2];
    comm_string[3] = ESCRITA;
    cksum = cksum + comm_string[3];
    comm_string[4] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
    Sleep(100);

    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }

}

// ========================================================================== //
// Io_READ_ADC:
// READS ADC CONVERTER
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_READ_ADC(byte ADDR, byte num_channels)
{

    byte cksum;
    BOOL ckcomm;
    float channel_0;
    float channel_1;
```

```
float channel_2;
float channel_3;
float channel_4;
float channel_5;

num_canais = num_channels;

Module[ADDR].SDEF = SendCanais;

comm_string[0] = Header;
comm_string[1] = ADDR;
cksum = comm_string[1];
comm_string[2] = (0x10 | READ_ADC);
cksum = cksum + comm_string[2];
num_channels = (0x07 & num_channels) | (0x10 & SendCanais);
comm_string[3] = num_channels;
cksum = cksum + comm_string[3];
comm_string[4] = cksum;

SioClrInbuf(Com_Port);  //Get rid of any old input chars
ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
Sleep(100);
if (ckcomm == false)
{
   SimpleMsgBox("Error on SioPutChars!");
   return false;
}

if (!wait_stat) return true; // No status byte

ckcomm = GET_RESPONSE(ADDR);

if (ckcomm == false)
{
   SimpleMsgBox("Error on GET_RESPONSE!");
   return false;
}

system("cls");

switch (num_canais) {

   case 0x01:

      channel_0 = ((comm_string[2] & 0x03)<<8) + comm_string[1];
      channel_0 = (channel_0*5/1024);

      printf("Canal_1: %.2f\n", channel_0);

      break;


   case 0x02:

      channel_0 = float ((comm_string[3] & 0x03)<<8) + comm_string[1];
      channel_0 = float (channel_0*5)/1024;

      channel_1 = ((comm_string[3] & 0x0C)<<6) + comm_string[2];
      channel_1 = (channel_1*5)/1024;

      printf("Canal_1: %.2f\n", channel_0);
      printf("Canal_2: %.2f\n", channel_1);

      break;

   case 0x03:

      channel_0 = ((comm_string[4] & 0x03)<<8) + comm_string[1];
      channel_0 = (channel_0*5/1024);

      channel_1 = ((comm_string[4] & 0x0C)<<6) + comm_string[2];
      channel_1 = (channel_1*5/1024);

      channel_2 = ((comm_string[4] & 0x30)<<4) + comm_string[3];
      channel_2 = (channel_2*5/1024);

      printf("Canal_1: %.2f\n", channel_0);
      printf("Canal_2: %.2f\n", channel_1);
      printf("Canal_3: %.2f\n", channel_2);
```

```
        break;

    case 0x04:

        channel_0 = ((comm_string[5] & 0x03)<<8) + comm_string[1];
        channel_0 = (channel_0*5/1024);

        channel_1 = ((comm_string[5] & 0x0C)<<6) + comm_string[2];
        channel_1 = (channel_1*5/1024);

        channel_2 = ((comm_string[5] & 0x30)<<4) + comm_string[3];
        channel_2 = (channel_2*5/1024);

        channel_3 = ((comm_string[5] & 0xC0)<<2) + comm_string[4];
        channel_3 = (channel_3*5/1024);

        printf("Canal_1: %.2f\n", channel_0);
        printf("Canal_2: %.2f\n", channel_1);
        printf("Canal_3: %.2f\n", channel_2);
        printf("Canal_4: %.2f\n", channel_3);

        break;

    case 0x05:

        channel_0 = ((comm_string[6] & 0x03)<<8) + comm_string[1];
        channel_0 = (channel_0*5/1024);

        channel_1 = ((comm_string[6] & 0x0C)<<6) + comm_string[2];
        channel_1 = (channel_1*5/1024);

        channel_2 = ((comm_string[6] & 0x30)<<4) + comm_string[3];
        channel_2 = (channel_2*5/1024);

        channel_3 = ((comm_string[6] & 0xC0)<<2) + comm_string[4];
        channel_3 = (channel_3*5/1024);

        channel_4 = ((comm_string[7] & 0x03)<<8) + comm_string[5];
        channel_4 = (channel_4*5/1024);

        printf("Canal_1: %.2f\n", channel_0);
        printf("Canal_2: %.2f\n", channel_1);
        printf("Canal_3: %.2f\n", channel_2);
        printf("Canal_4: %.2f\n", channel_3);
        printf("Canal_5: %.2f\n", channel_4);

        break;

    case 0x06:

        channel_0 = ((comm_string[7] & 0x03)<<8) + comm_string[1];
        channel_0 = (channel_0*5/1024);

        channel_1 = ((comm_string[7] & 0x0C)<<6) + comm_string[2];
        channel_1 = (channel_1*5/1024);

        channel_2 = ((comm_string[7] & 0x30)<<4) + comm_string[3];
        channel_2 = (channel_2*5/1024);

        channel_3 = ((comm_string[7] & 0xC0)<<2) + comm_string[4];
        channel_3 = (channel_3*5/1024);

        channel_4 = ((comm_string[8] & 0x03)<<8) + comm_string[5];
        channel_4 = (channel_4*5/1024);

        channel_5 = ((comm_string[8] & 0x0C)<<6) + comm_string[6];
        channel_5 = (channel_5*5/1024);

        printf("Canal_1: %.2f\n", channel_0);
        printf("Canal_2: %.2f\n", channel_1);
        printf("Canal_3: %.2f\n", channel_2);
        printf("Canal_4: %.2f\n", channel_3);
        printf("Canal_5: %.2f\n", channel_4);
        printf("Canal_6: %.2f\n", channel_5);

        break;

}
```

```
}

// ========================================================================== //
// Io_CONFIG_DAC:
// CONFIGS COUNTERS 1(B) AND 2 TO GENERATE A PWM SIGNAL
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_CONFIG_DAC(byte ADDR)
{

   byte cksum;
   BOOL ckcomm;

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x00 | CONFIG_DAC);
   cksum = cksum + comm_string[2];
   comm_string[3] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 4);
   Sleep(100);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte

   ckcomm = GET_RESPONSE(ADDR);

   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}

// ========================================================================== //
// Io_WRITE_DAC:
// GENERATES PWM SIGNAL IN 1 OR 2 CHANNELS
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Io_WRITE_DAC(byte ADDR, byte Channel_PWM, byte WRITE_PWM1, byte WRITE_PWM2)
{
   byte cksum;
   BOOL ckcomm;
   int num_bytes;


   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   if( (Channel_PWM & Channel_12) == 0)
   {
      comm_string[2] = (0x20 | WRITE_CDA);
      cksum = cksum + comm_string[2];
      comm_string[3] = Channel_PWM;
      cksum = cksum + comm_string[3];
      comm_string[4] = WRITE_PWM1;
      cksum = cksum + comm_string[4];
      comm_string[5] = cksum;
      num_bytes = 6;
   }else
   {
      comm_string[2] = (0x30 | WRITE_CDA);
      cksum = cksum + comm_string[2];
      comm_string[3] = Channel_PWM;
      cksum = cksum + comm_string[3];
      comm_string[4] = WRITE_PWM1;
```

```
        cksum = cksum + comm_string[4];
        comm_string[5] = WRITE_PWM2;
        cksum = cksum + comm_string[5];
        comm_string[6] = cksum;
        num_bytes = 7;
    }

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, num_bytes);
    Sleep(100);

    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }

//    printf("Leitura: %x\n", comm_string[1]);

}

// ########################################################################## //
// ########################################################################## //
// ##     FUNCOES DO SERVO MOTOR                                           ## //
// ########################################################################## //
// ########################################################################## //

// ========================================================================== //
// Sv_CONFIG_SERVO:
// CONFIGS SERVO STATUS DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sv_CONFIG_SERVO(byte ADDR, byte NUM_PORT,int Freq)
{
    byte cksum;
    BOOL ckcomm;
    int Time;

    Time =10000/Freq;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x20 | CONFIG_SERVO);
    cksum = cksum + comm_string[2];
    comm_string[3] = NUM_PORT;
    cksum = cksum + comm_string[3];
    comm_string[4] = Time;
    cksum = cksum + comm_string[4];
    comm_string[5] = cksum;
    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 6);
    Sleep(100);
//    system("PAUSE");
    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
```

```
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}


// ========================================================================== //
// Sv_SET_SERVO:
// SETS SERVO STATUS POSITION DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sv_SET_POS(byte ADDR, byte PORT, float time)
{
   byte cksum;
   BOOL ckcomm;
   int duty_cycle;

   duty_cycle = int(time*10);

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x20 | SET_POS_IND);
   cksum = cksum + comm_string[2];
   comm_string[3] = PORT;
   cksum = cksum + comm_string[3];
   comm_string[4] = duty_cycle;
   cksum = cksum + comm_string[4];
   comm_string[5] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 6);
   Sleep(100);
//    system("PAUSE");
   system("cls");
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }
}


// ========================================================================== //
// Sv_SET_SERVO:
// SETS ALL SERVO STATUS POSITION DATA
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //

BOOL Sv_SET_POS_ALL(byte ADDR,float time)
{
   byte cksum;
   BOOL ckcomm;
   int duty_cycle;

   duty_cycle = int(time*10); //CALCULO DO CICLO DE TRABALHO

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x10 | SET_POS_ALL);
   cksum = cksum + comm_string[2];
   comm_string[3] = duty_cycle;
   cksum = cksum + comm_string[3];
```

```
    comm_string[4] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
    Sleep(100);
//    system("PAUSE");
    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}

// ########################################################################## //
// ########################################################################## //
// ##     FUNCOES DO MODULO MOTOR DE PASSO                                 ## //
// ########################################################################## //
// ########################################################################## //
// ========================================================================== //
// Sm_STOP_MOTOR:
// STOPS MOTOR MOVEMENT
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_STOP_MOTOR(byte ADDR, byte MOTOR)
{
    byte cksum;
    BOOL ckcomm;
    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x10 | STOP_MOTOR);
    cksum = cksum + comm_string[2];
    comm_string[3] = MOTOR;
    cksum = cksum + comm_string[3];
    comm_string[4] = cksum;
    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
    Sleep(100);
//    system("PAUSE");
    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}

// ========================================================================== //
// Sm_RESET_MOTOR:
// RESETS MOTOR PARAMETERS
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_RESET_MOTOR(byte ADDR, byte MOTOR)
{
```

```
    byte cksum;
    BOOL ckcomm;
    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x10 |RESET_MOTOR);
    cksum = cksum + comm_string[2];
    comm_string[3] = MOTOR;
    cksum = cksum + comm_string[3];
    comm_string[4] = cksum;
    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 5);
    Sleep(100);
//    system("PAUSE");
    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}

// ========================================================================== //
// Sm_START_MOTOR:
// STARTS MOTOR MOVEMENT
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_START_MOTOR(byte ADDR, byte MOTOR, byte MODO)
{
    byte cksum;
    BOOL ckcomm;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x20 | START_MOTOR);
    cksum = cksum + comm_string[2];
    comm_string[3] = MOTOR;
    cksum = cksum + comm_string[3];
    comm_string[4] = MODO;
    cksum = cksum + comm_string[4];
    comm_string[5] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 6);
    Sleep(100);

    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}

// ========================================================================== //
// Sm_DESLOC_MOTOR:
```

```
// DEFINES MOTOR NUMBER OF STEPS AND SPEED
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_DESLOC_MOTOR(byte ADDR,byte SENSE,  int NUM_STEPS,  int VEL)
{
   byte cksum;
   byte STEP_HI, STEP_LOW;
   BOOL ckcomm;

   STEP_LOW = (NUM_STEPS & 0xFF);
   STEP_HI  = ((NUM_STEPS & 0xFF00)>>8);

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x40 | DESLOC_MOTOR);
   cksum = cksum + comm_string[2];
   comm_string[3] = SENSE;
   cksum = cksum + comm_string[3];
   comm_string[4] = STEP_HI;
   cksum = cksum + comm_string[4];
   comm_string[5] = STEP_LOW;
   cksum = cksum + comm_string[5];
   comm_string[6] = VEL;
   cksum = cksum + comm_string[6];
   comm_string[7] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 8);
   Sleep(100);

   system("cls");
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on SioPutChars!");
      return false;
   }

   if (!wait_stat) return true; // No status byte


   ckcomm = GET_RESPONSE(ADDR);
   if (ckcomm == false)
   {
      SimpleMsgBox("Error on GET_RESPONSE!");
      return false;
   }

}

// ========================================================================== //
// Sm_VELOC_MOTOR:
// ADJUSTS MOTOR VELOCITY IN ONE SENSE
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_VELOC_MOTOR(byte ADDR, byte MOTOR, byte SENSE,  int MAG)
{
   byte cksum;
   BOOL ckcomm;

   MOTOR = MOTOR | (SENSE<<2);

   comm_string[0] = Header;
   comm_string[1] = ADDR;
   cksum = comm_string[1];
   comm_string[2] = (0x20 | VELOC_MOTOR);
   cksum = cksum + comm_string[2];
   comm_string[3] = MOTOR;
   cksum = cksum + comm_string[3];
   comm_string[4] = MAG;
   cksum = cksum + comm_string[4];
   comm_string[5] = cksum;

   SioClrInbuf(Com_Port);  //Get rid of any old input chars
   ckcomm = SioPutChars(Com_Port, (char *)comm_string, 6);
```

```
    Sleep(100);

    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }

    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }

}

// ========================================================================== //
// Sm_TRAPE_MOTOR:
// CONFIGS TRAPEZOIDAL MODE FOR MOTOR MOVEMENT
// ADDR: Address
// StatusItens: Data itens to be returned
// Returns true or false
// ========================================================================== //
BOOL Sm_TRAPE_MOTOR(byte ADDR,byte SENSE,  int NUM_STEP, int TIME)
{
    byte cksum;
    BOOL ckcomm;
    byte STEP_H, STEP_L, SPACE_L, SPACE_H;
    int velocidade, aceleracao, tempo_aceleracao, espaco_aceleracao;


    velocidade = (3*NUM_STEP)/(TIME*2); //Calculo da velocidade de cruzeiro( Considera a velocidade 1.5 vezes o valor
minimo da velocidade de cruzeiro)
    tempo_aceleracao = ((velocidade*TIME)-NUM_STEP)/velocidade; //Calculo do tempo de aceleração
    aceleracao = velocidade/tempo_aceleracao; //Calculo da aceleracão
    espaco_aceleracao = (aceleracao*tempo_aceleracao*tempo_aceleracao)/2;


    STEP_L  = (NUM_STEP & 0x00FF);
    STEP_H  = (NUM_STEP & 0xFF00)>>8;

    SPACE_L = (espaco_aceleracao & 0x00FF);
    SPACE_H = (espaco_aceleracao & 0xFF00)>>8;

    comm_string[0] = Header;
    comm_string[1] = ADDR;
    cksum = comm_string[1];
    comm_string[2] = (0x60 | PERFIL_TRAPEZ);
    cksum = cksum + comm_string[2];
    comm_string[3] = SENSE;
    cksum = cksum + comm_string[3];
    comm_string[4] = STEP_H;
    cksum = cksum + comm_string[4];
    comm_string[5] = STEP_L;
    cksum = cksum + comm_string[5];
    comm_string[6] = SPACE_H;
    cksum = cksum + comm_string[6];
    comm_string[7] = SPACE_L;
    cksum = cksum + comm_string[7];
    comm_string[8] = aceleracao;
    cksum = cksum + comm_string[8];
    comm_string[9] = cksum;

    SioClrInbuf(Com_Port);  //Get rid of any old input chars
    ckcomm = SioPutChars(Com_Port, (char *)comm_string, 10);
    Sleep(100);

    system("cls");
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on SioPutChars!");
        return false;
    }
```

```
    if (!wait_stat) return true; // No status byte


    ckcomm = GET_RESPONSE(ADDR);
    if (ckcomm == false)
    {
        SimpleMsgBox("Error on GET_RESPONSE!");
        return false;
    }
}



//---------------------------------------------------------------------------//
//  InitVars:
//  Initialize misc network variables
//  N: Number of modules
//
//  Returns nothing
//---------------------------------------------------------------------------//
void InitVars(int N) // Testado OK
{
      int i;

      Module[0].ID = 0;        //Default to a known module type for module 0
      Module[0].VER = 0;
      Module[0].STAT = 0;
      Module[0].SDEF = 0;
      Module[0].GADDR = 0xFF;
      Module[0].LEADER = false;

      for (i=1; i<=N; i++)
      {
         Module[i].ID = 0xFF; // N modules
         Module[i].VER = 0;
         Module[i].STAT = 0;
         Module[i].SDEF = 0;
         Module[i].GADDR = 0xFF;
         Module[i].LEADER = false;
      }
}

// ========================================================================== //
// SYSTEM_RESET:
// RESETS THE SYSTEM
// ADDR: Group Address
//
// Returns nothing
// ========================================================================== //
void SYSTEM_RESET(byte ADDR) // Testado OK
{
    int  k;
    BOOL ws;

    //Send string of 0's to flush input buffers
    comm_string[0] = 0;
    for (k=0; k<20; k++) SioPutChars(Com_Port,(char *) comm_string, 1);

    ws = wait_stat;
    wait_stat = false;
    Mo_HARD_RESET(ADDR);
    wait_stat = ws;

    Sleep(100);                      //wait for reset to execute

    SioChangeBaud(Com_Port, 19200);  //Reset the baud rate to the default
    SioClrInbuf(Com_Port);           //Clear out any random crap left in buffer
}

// ========================================================================== //
// SYSTEM_INIT:
// INITIALIZES THE NETWORK OF CONTROLLERS WITH SEQUENTIAL ADDRESS (1,..., N)
// N: Number of modules
// portname: string indicating serial port (COM1 or COM2)
// baudrate: Baud rate for serial comunication (PB19200, PB57600 e PB115200)
//
// Returns true (OK) or false (not OK)
// ========================================================================== //
BOOL SYSTEM_INIT(char *portname, byte baudrate, int N) // Testado OK
```

```
{
   DWORD          numread;
   byte           addr;

   InitVars(N);

   Com_Port = SioOpen(portname, 19200);        // default rate of 19200
   if (Com_Port == INVALID_HANDLE_VALUE)
      return false;

   SYSTEM_RESET(0xFF);
   SYSTEM_RESET(0xFF);

   for (addr = 1; addr <= N; addr++)
   {
      // Set the address to a unique value:
      comm_string[0] = 0xAA;
      comm_string[1] = 0;                    // default address
      comm_string[2] = 0x20 | SET_ADDRESS;
      comm_string[3] = addr;                 // new address
      comm_string[4] = 0xFF;                 // group address
      comm_string[5] = comm_string[1] + comm_string[2] +
                       comm_string[3] + comm_string[4];

      SioPutChars(Com_Port, (char *) comm_string, 6);

      numread = SioGetChars(Com_Port, (char* ) comm_string, 2);
      if (numread != 2) break;
      if (comm_string[0] != comm_string[1])
      {
         ErrorMsgBox("Status checksum error - please reset the Network");
         SioClose(Com_Port);
         return false;
      }

      //Read the device type
      comm_string[0] = 0xAA;
      comm_string[1] = addr;
      comm_string[2] = 0x10 | READ_STATUS;
      comm_string[3] = SEND_ID_VER;
      comm_string[4] = comm_string[1] + comm_string[2] + comm_string[3];

      SioPutChars(Com_Port, (char *) comm_string, 5);

      numread = SioGetChars(Com_Port, (char *) comm_string, 4);
      if (numread != 4)
      {
          ErrorMsgBox("Could not read device type");
          SioClose(Com_Port);
          return false;
      }
      Module[addr].STAT = comm_string[0];
      Module[addr].ID = comm_string[1];
      Module[addr].VER = comm_string[2];
      Module[addr].SDEF = 0;
      Module[addr].GADDR = 0xFF;
      Module[addr].LEADER = false;

   }

   if (addr == N+1)
   {
      //Mo_SET_BAUD(0xFF, baudrate);
      return true;
   }
   else
   {
      ErrorMsgBox("Could not initialize Modules");
      SioClose(Com_Port);
      return false;
   }

}

// ############################################################################ //
// ############################################################################ //
// ##     PROGRAMA PRINCIPAL                                              ## //
// ############################################################################ //
// ############################################################################ //
```

```
int main(int argc, char *argv[])
{
   float tempo = 1.1;
   BOOL    res;
   byte    baudr = PB19200;    // BAUD RATE
   byte    svmode;

   wait_stat = true;
   ErrorPrinting(true);

   // ================== MODULES CONFIGURATION ============================
   res = SYSTEM_INIT("COM1", baudr, 1);
   if (res == false)
   {
      ErrorMsgBox("Could not initialize Modules");
      SioClose(Com_Port);
      return 1;
   }
   // ====================================================================

   if (Com_Port == INVALID_HANDLE_VALUE)
      ErrorMsgBox("Erro Com_Port");

   // ----------
   printf(" \n\n Versao do controlador 1 = %d \n\n",Module[1].VER);

   // ----------

   Mo_NO_OP(1);

   // ================== Modulo Joystick ==================================
   /* for(;;)
    {
      res =  Jk_READ_JOYSTICK(1);
      res ? : SimpleMsgBox("Erro Sv_READ_JOYSTICK");
    }
   */
   // ====================================================================



   // ================== Modulo Entrada e Saída ===========================
     /*
      res = Io_READ_ADC(0x01, 0x06);
      res ? : SimpleMsgBox("Erro Sv_READ_ADC");

      res = Io_CONFIG_DAC(0x01);
      res ? : SimpleMsgBox("Erro Sv_CONFIG_DAC");
      res = Io_WRITE_DAC(0x01,Channel_2, 0x7F,0x7F );
      res ? : SimpleMsgBox("Erro Sv_WRITE_DAC");


     res = Io_CONFIG_IO(0x01, 0x03);
     res ? : SimpleMsgBox("Erro Sv_CONFIG_IO");
     res = Io_WRITE_IO(0x01, 0x01);
     res ? : SimpleMsgBox("Erro Sv_WRITE_IO");
     res = Io_READ_IO(0x01);
     res ? : SimpleMsgBox("Erro Sv_READ_IO");
    */
   // ====================================================================


   // ================== Modulo Servo =====================================
   /*
      res = Sv_CONFIG_SERVO(1,1,80);
      res ? : SimpleMsgBox("Erro Sv_CONFIG_SERVO");

      res = Sv_SET_POS(1,1,2.0);
      res ? : SimpleMsgBox("Erro Sv_SET_POS");

      res = Sv_SET_POS_ALL(1,2.0);
      res ? : SimpleMsgBox("Erro Sv_SET_POS_ALL");

   */
   // ====================================================================


   // ================== Modulo Motor de Passo ============================
```

```
    /*
        res = Sm_STOP_MOTOR(1,1);
        res ? : SimpleMsgBox("Erro Sv_CONFIG_SERVO");

        res = Sm_DESLOC_MOTOR(1,0,1000,50);
        res ? : SimpleMsgBox("Erro Sv_SET_POS");
        system("pause");
        res = Sm_START_MOTOR(0x01, 0x01, 0x01);
        res ? : SimpleMsgBox("Erro Sv_START_MOTOR");

        res = Sm_TRAPE_MOTOR(1,1,9000,60);
        res ? : SimpleMsgBox("Erro Sv_TRAPE_MOTOR");

        res = Sm_VELOC_MOTOR(1, 1, 1, 50);
        res ? : SimpleMsgBox("Erro Sv_VELOC_MOTOR");
     */

    // ========================================================================

    res = Sm_TRAPE_MOTOR(1,1,9000,60);
    res ? : SimpleMsgBox("Erro Sv_TRAPE_MOTOR");
    system("pause");

    res = Sm_START_MOTOR(0x01, 0x01, 0x02);
    res ? : SimpleMsgBox("Erro Sv_START_MOTOR");

    system("pause");
    SioClose(Com_Port);
    return EXIT_SUCCESS;
}
```