MARCELO SHINMYO
MARCO ANTONIO GROTKOWSKY

PROJETO DE BANCADA DE TESTES PARA SISTEMAS DE CONTROLE DISTRIBUÍDOS UTILIZANDO MOTORES C.C.

São Paulo
2010

MARCELO SHINMYO
MARCO ANTONIO GROTKOWSKY

PROJETO DE BANCADA DE TESTES PARA SISTEMAS DE CONTROLE DISTRIBUÍDOS UTILIZANDO MOTORES C.C.

Monografia apresentada à Escola Politécnica da Universidade de São Paulo.

Orientador:
Prof. Dr. Newton Maruyama

São Paulo
2010

FICHA CATALOGRÁFICA

Shinmyo, Marcelo
Projeto de bancada de testes para sistemas de controle distribuídos utilizando motores C.C. / M. Shinmyo, M.A. Grotkowsky. -- São Paulo, 2010.
66 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1. Sistemas distribuídos 2. Algoritmos de Scheduling 3. Controle (Teoria de sistemas e controle) I. Grotkowsky, Marco Antonio II. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III. t.

DEDICATÓRIA

Dedicamos este trabalho a nossas famílias.

AGRADECIMENTOS

Agradecemos a nossas famílias, ao técnico Carlos Alberto de Souza Lima, e ao professor orientador Newton Maruyama.

RESUMO

Objetiva-se nesse trabalho a construção de uma plataforma de testes para sistemas de controle distribuídos em redes de comunicação com o uso de motores de corrente contínua. A plataforma é composta de um motor a ser controlado e um motor atuando como gerador de distúrbios de torque, ambos acoplados por uma inércia variável. Microcontroladores TINI são aplicados no transporte de informações correspondentes à velocidade angular para uma rede de comunicação CAN e para repassar os sinais correspondentes às ações de controle para os drivers de acionamento. Para as tarefas que calculam as ações de controle, é utilizado um sistema baseado no microcontrolador ARM7. As plataformas foram modeladas e simuladas no microcontrolador TINI, atuando como nós na rede CAN e introduzindo atrasos. Por meio do sistema distribuído, é possível analisar: o desempenho do protocolo de comunicação CAN; a influência dos algoritmos de controle em sistemas operacionais de tempo real (controlador proporcional-integral, controlador com compensação de atraso do tempo de amostragem, controlador utilizando técnica *Subtask Scheduling*).

ABSTRACT

This work deals with the implementation of a testbed for networked control systems using DC motors. The platform is composed by the controlled motor and another motor actuating as a torque disturbance generator, both coupled with a variable inertia. TINI microcontrollers are applied on information transport from the position and torque sensors to the network and to transmit the control action signal to the actuation drivers. For the control action calculation tasks, an ARM7 based system is used. In order to plan the required computational power of a networked controls system, we have devised an experimental setup with a single processing unit node (Keil ARM7 Board) and nodes that simulates distributed plants (TINI). At first, we evaluate network time-delay distributions and packet losses while increasing the number of distributed plant nodes. Later we propose some computational techniques that introduces modifications on the classical control algorithms (like sampling time adaption, subtask scheduling, etc.) that can improve the control performance. At the end of this work, we draw some preliminary conclusions about the resulting controllers performance.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Objetivo

Objetiva-se nesse trabalho a construção de uma plataforma de testes para sistemas de controle distribuídos em redes de comunicação com o uso de motores de corrente contínua.

Analisa-se do uso do protocolo de comunicação *CAN* como meio de transmissão de sinais entre plantas e controlador. Verifica-se a interferência de atrasos determinísticos ou não-determinísticos no desempenho do sistema de controle.

Como terceiro objetivo, estuda-se a influência de algoritmos de controle em sistemas operacionais de tempo real. São estudados controlador Proporcional-Integral, controlador com compensação de atrasos no tempo de amostragem, e controlador utilizando a técnica de *Subtask Scheduling*.

Com isso, esse trabalho visa o estudo de técnicas de co-projeto (*co-design*), que é o projeto do sistema de controle levando-se em consideração o algoritmo de escalonamento.

1.2 Motivação

Produtos modernos como carros, celulares, MP3/DVD players, video games, utilizam sistemas embarcados para implementar muitas de suas funcionalidades. Essas, por sua vez, necessitam de recursos computacionais físicos (*hardware*) que executam operações pré-definidas em um software.

Sistemas de controle embarcado geralmente possuem vários processos sendo executados paralelamente, inclusive vários processos de controle. Dessa forma, o escalonamento de tarefas se torna importante pois decide qual processo será executado em um determinado instante. Desde meados de 1970 o interesse acadêmico em escalonamento tem sido grande, mas pouco trabalho focou nas tarefas de controle. Além disso, a teoria de controle não estuda o problema dos recursos de hardware limitados ou compartilhados. Em vez disso, assume-se que o

controlador utiliza um computador dedicado.

Não há, em geral, incentivo para se estudar a possibilidade de conciliar o desenvolvimento do sistema de controle com a estratégia de escalonamento dos processos, uma vez que o aumento do poder de processamento resolve os problemas de desempenho do sistema. Apesar disso, os fabricantes ainda demonstram dificuldades em alocar tarefas em processadores altamente carregados, considerando que visa-se a utilização do hardware mais econômico possível.

Os sistemas de controle atuais geralmente são distribuídos em uma rede, onde sensores e atuadores se encontram em diferentes nós. São, também, tratados como componentes de software e espera-se que eles possam suportar mudanças de hardware, upgrades online, etc. Dessa forma, esses sistemas apresentam alta flexibilidade. Por outro lado, são não determinísticos, introduzindo variações de tempo que prejudicam o desempenho do controlador.

Consequentemente, para se obter melhor desempenho no controle quando se dispor de recursos de hardware limitados, deve-se integrar o projeto do controlador com o projeto do escalonador de processos [3].

1.3 Organização do Documento

Este documento foi estruturado da seguinte maneira:

- Capítulo 1 (Introdução):

  Apresenta objetivo, motivação, e a forma em que o documento está organizado.

- Capítulo 2 (Conceitos e Tecnologias Envolvidas):

  Conceitos sobre indeterminismo temporal. Descrição das técnicas de escalonamento de processos *Fixed Priority* (FP) e *Earliest Deadline First* (EDF). *Subtask-Scheduling*. Apresentação dos protocolos de comunicação *SPI* e CAN. Descrição dos microcontroladores TINI e ARM7. Sistema operacional de tempo real RTX.

- Capítulo 3 (Metodologia):

  Descreve a concepção da plataforma de testes. Apresentação do kit mecânico e elétrico com os respectivos desenhos esquemáticos e especificações. Modelagem e Identificação da Planta da Plataforma. Configuração da rede CAN e SPI. Técnicas de Controle utilizadas. Aquisição e tratamento dos dados, e simulação do Sistema Distribuído.

- Capítulo 4 (Resultados):

Apresenta os resultados obtidos, como: Plataforma Mecânica, *Bridge* de Rede, *Software* de Controle, Simulação de Controle, Simulação do Sistema Distribuído, Análise dos Controladores e dos atrasos.

- Capítulo 5 (Conclusão):

  Discute brevemente os resultados e apresenta as considerações finais.

- Capítulo 6 (Referências Bibliográficas):

- Capítulo 7 (Apêndice):

  Apresenta uma ferramenta para simular os aspectos temporais de sistemas de tempo real e de redes.

  .

2 CONCEITOS E TECNOLOGIAS ENVOLVIDAS

2.1 Indeterminismo Temporal

Um *loop* de controle consiste principalmente em entrada de dados, cálculo do sinal de controle, e escrita da saída. No caso ideal, o algoritmo de controle é executado periodicamente conforme o período de amostragem, e o tempo entre a leitura e a escrita de dados deve ser desprezível. Na prática, existem diversas formas de atraso presentes do sistema, conforme apresentado na figura 2.1.

O período de amostragem que assume-se constante é h. Porém, existe um atraso L_S entre o momento em que o escalonador coloca a *task* na condição *Ready*, e o momento em que ela de fato é executada. Esse atraso é chamado de latência de amostragem, e ocorre devido à interrupção da *task* em questão por outras de maior prioridade (ou seja, depende da política de escalonamento). O tempo entre a leitura e escrita de dados L_{io} corresponde à latência de entrada e saída.

A variação da amostragem é definida como:

$$J_S = L_S^{max} - L_S^{min}$$

Define-se variação do intervalo de amostragem como:

$$J_h = h^{max} - h^{min}$$

Analogamente, a variação do tempo entre a entrada e saída é:

$$J_{io} = L_{io}^{max} - L_{io}^{min}$$

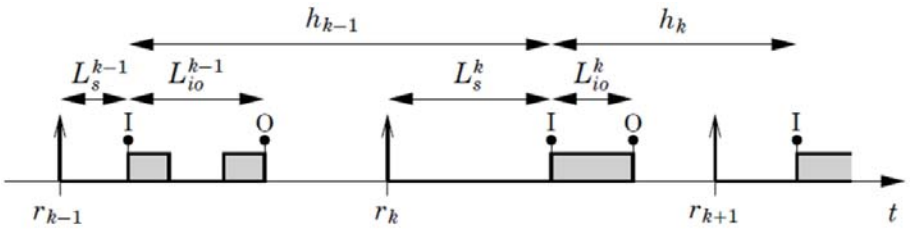


Figura 2.1: Atrasos indeterminísticos no loop de controle.

2.2 Técnicas de Escalonamento de Processos

Sistemas de tempo real possuem *deadlines*, que são limites superiores de tempo aos quais o processo não deve ultrapassar. Nesse tipo de sistema, é essencial que exista uma análise anterior à implementação. O objetivo é que se possa garantir que todas as tarefas sejam executadas antes do seu *deadline*.

A estratégia de escalonamento pode se dar na forma estática ou dinâmica. O primeiro modelo consiste em uma tabela com uma sequência de tarefas a serem executadas ciclicamente. Esta tabela é baseada em algoritmos de otimização previamente elaborados. Porém, este modelo não suporta a modificação da tabela durante a execução, dificultando a integração com o sistema de controle.

A segunda forma na qual a estratégia de escalonamento pode se apresentar é dinâmica, pois o processo a ser executado é definido durante a execução do programa. Os algoritmos de escalonamentos que serão estudados nesse trabalho foram primeiramente apresentados em 1973 por Liu e Layland [2], e diferem na condição suficiente para alterar a prioridade das tarefas. Serão analisados o *Earliest Deadline First* (EDF), e o *Rate Monotonic* (RM). Nesse trabalho, é utilizado o termo *Fixed Priority* ou FP para se referir à técnica RM.

O EDF significa "prazo-mais-curto-primeiro"e dá maior prioridade ao processo em que seu deadline está mais próximo de ser atingido. O FP, prioridade-fixa, considera como tarefa de maior prioridade aquela que possui o menor período de execução.

Para a melhor análise do EDF e RM, [2] propõe as seguintes hipóteses:

- Todos processos são periódicos e possuem um período T_i,
- Todo processo possui um *Worst Case Execution Time* (WCET), que é o tempo que o processador requer para executar determinado processo no pior caso possível, considerando que apenas essa tarefa é realizada. O WCET será representado por C_i,
- Toda tarefa possui um deadline D_i conhecido e igual ao período:$D_i = T_i$,

- Não há comunicação entre processos,
- Operações internas ao sistema operacional não requerem tempo de execução.

Em [2], é apontado que, considerando U a carga da CPU, no caso do EDF:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.1}$$

Dessa forma, temos que para uma utilização de 100 % da CPU, ainda se pode cumprir todos os deadlines.

Para o FP, temos:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.2}$$

Nesse caso, se n tende ao infinito e para os deadlines serem cumpridos, temos que $U \leq 0.693$.

Nos últimos anos a análise do EDF e do FP têm sido refeitas considerando hipóteses menos restritivas.

2.3 Subtask-Scheduling

Uma implementação trivial de múltiplos controladores numa mesma plataforma consiste no uso do algoritmo de escalonamento *First Priority* e da utilização de somente uma *task* para cada controlador. A cada período de amostragem, a entrada é lida, o algoritmo de controle é executado, e a saída é escrita. Quando executado em um sistema de tempo real, atrasos indeterminísticos causados pelo escalonamento são introduzidos. Desse modo, o desempenho do controlador, como um todo, é prejudicado pelo aparecimento de grandes atrasos como: latência de amostragem, latência de entrada e saída, e variações no período de amostragem.

Para uma boa performance do controlador, além do fato de que os cálculos devem ser finalizados antes do *deadline* correspondente, é importante que as operações de amostragem e de atuação sejam regulares, e livres de variações ou latências consideráveis.

Objetivando-se reduzir a latência, pode-se dividir o algoritmo de controle em *tasks* diferentes, e escalona-las individualmente definindo-se diferentes prioridades para cada uma delas. O conceito principal do *Subtask Scheduling*, apresentado em [3], é dividir o algoritmo de controle em duas *tasks*, onde a porção crítica em que se calcula o sinal de controle forma a *task Calculate Output*, e o resto das operações constituem a *task Update State*. Nota-se que a

task que calcula a saída é mais crítica do ponto de vista temporal, pois espera-se que ela seja executada precisamente a cada período de amostragem. A *task* responsável pela escrita da saída deve ser cumprida até o período de amostragem. A redução da latência de entrada e saída pode ser provada pela análise que considera o pior tempo de execução (onde a *task* que calcula o sinal de controle é interrompida por todas *tasks* que possuem prioridade superior à ela), e consequentemente, a maior latência possível.

O caso ideal de implementação do *Subtask Scheduling* utilizando *First Priority* ocorre quando todas as *tasks* de cálculo do sinal de controle possuem prioridades mais altas do que todas as *tasks* de escrita da saída. Porém, esse tipo de definição de prioridades pode levar ao caso de impossibilidade de escalonamento. Para esses casos problemáticos, existem algoritmos que minimizam os *deadlines* (e consequentemente aumentam as prioridades) das *tasks* de cálculo do sinal de controle a partir de modelos escalonáveis.

2.4 Protocolos de Comunicação

2.4.1 O Protocolo CAN

O *Controller Area Network* (CAN) foi criado em meados dos anos 80 pela fornecedora de sistemas automotivos Bosch. O principal objetivo era fazer uma comunicação serial robusta e aumentar assim a confiabilidade, sendo hoje vastamente empregada na automação industrial e sistemas embarcados automotivos [4].

O CAN, juntamente com muitos outros protocolos, utiliza o modelo de camadas padrão, *Open Systems Interconnection* (OSI) da ISO, o que a princípio permite a interoperabilidade de produtos de diferentes fabricantes. No CAN as 2 últimas camadas da OSI já são implementadas, deixando as camadas superiores à disposição ao desenvolvedor para eventuais adaptações ou otimizações. Essas camadas sujeitas à implementação são em geral usadas para padronizar os procedimentos de inicialização, escolher endereços para os tipos de mensagens e os nós participantes, determinar as estruturas das mensagens , prover rotinas para tratamento de erros, entre outras aplicações.

O CAN é um protocolo do tipo CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*). Isso significa que cada dispositivo deve monitorar o bus de comunicação por um período determinado de tempo antes de transmitir alguma informação. Após esse intervalo, todos os dispositivos têm a mesma oportunidade de envio de mensagem. Há, porém, uma prioridade definida para cada nó e um sistema de detecção de colisões. Assim quando duas

transmissões se iniciam ao mesmo tempo, somente a mensagem de maior prioridade é enviada, sem sofrer atraso ou corrompimento.

Nesse protocolo o nível lógico dominante é o 0, consequentemente o recessivo é o 1. Portanto o identificador de mensagem(campo utilizado para a arbitragem de prioridade) que tiver o menor valor será o de maior prioridade. Como foi citado, cada dispositivo monitora o bus para verificar se o bit que está tentando enviar é o que é de fato encontrado no bus. Em um caso de colisão, em algum momento o bit da mensagem de menor prioridade será "dominada"pelo nível lógico 0, cientificando o dispositivo de que há colisão e fazendo esse último parar de transmitir imediatamente.

A transmissão de mensagens não são baseadas em endereçamento, cada mensagem contém a definição da própria prioridade além dos dados a serem enviados. Logo todos os nós da rede recebem as mesmas mensagens, e cabe à elas decidir se as descarta ou se as processa. Uma vantagem é a facilidade para incluir novos nós no sistema por não necessitar reprogramar todos os outros para reconhecer essa ação. Os novos dispositivos recebem todas as mensagens enviadas, bastando somente escolher qual delas processar. Apesar dessas características, o CAN permite também que um nó requeira informação de um outro especificamente. Isso é chamado de RTR, *Remote Transmit Request*.

Utilizam-se 4 tipos de mensagens no CAN: *Data Frame*, o mais comum deles, usado para transmitir dados para algum ou todos os nós; o *Remote Frame*, usado especificamente para RTR; *Error Frame*, o qual é gerado para indicar algum dos erros definidos no protocolo; e *Overload Frame*, o qual também indica erro, porém especialmente em caso de sobrecarga de algum nó, quando uma mensagem é recebida antes de terminar o processamento da anterior.

Um *Data Frame* é formado por *Arbitration Field*, *Control Field*, *Data Field*, *CRC Field*, 2 *bits* de *Acknowledge Field* e o *End of Frame*. O *Arbitration Field* é usado para priorizar as mensagens como já foi comentado, ou então para determinar se a mensagem é um Remote Frame. É composto por 12 ou 32 *bits* dependendo do tipo de *Data Frame*, o qual pode ser um *Standard Frame* ou *Extended Frame*.

O *Control Field* é composto por 6 *bits*, sendo que para um *Standard Frame* o mais significativo será um bit dominante e o segundo é reservado. No caso de um *Extended Frame* os 2 *bits*são reservados. Os 4 últimos *bits* determinam o tamanho dos dados. O *Data Field* são os dados com o tamanho especificado pelos 4 *bits* citados.

O CRC *Field* consiste em 16 bits, 15 para o valor do CRC (*Cyclic Redundancy Check*) e um *bit* delimitador. É usado pelos nós receptores para identificar se ocorreu algum tipo de

corrompimento no dado. O *Acknowledge Field* é usado para indicar se a mensagem foi recebida corretamente, em caso positivo o nó receptor coloca o nível lógico dominante no instante do ACK *Slot bit*. Finalmente, 7 *bits* recessivos compõem o *End of Frame*.

Os nós de uma rede CAN são capazes de verificar o nível de falha existente e mudar para diferentes modos de funcionamento. Podem, por exemplo, desligarem-se completamente dependendo da gravidade da falha. Dessa forma, os nós com erro deixam de ocupar a banda do bus, deixando-o livre para informações críticas.

Existem 5 tipos de erros no CAN:

- *CRC Error*: O valor dos 15 bits de CRC são calculados pelo dispositivo transmissor e é enviado no CRC *field*. Todos os receptores calculam o CRC e verificam se corresponde ao recebido. Em caso negativo, é gerado um *Error Frame* e a mensagem é reenviada depois de um intervalo de tempo apropriado.

- *Acknowledge Error*: Na transmissão de um *Data Frame*, se pelo menos um dispositivo recebeu a mensagem corretamente o ACK *Slot bit* será dominante. Caso o transmissor verifica que esse último é recessivo, um *Error Frame* é gerado e a mensagem é reenviada após um intervalo de tempo.

- *Form Error*: São diversos os bits reservados para serem recessivos, como no *End of Frame*, intervalo entre transmissões, *Acknowledge Delimiter* ou CRC *Delimiter*. Caso seja encontrado o nível lógico dominante em algum desses *bits*, é gerado um *Error Frame* e a mensagem é retransmitida.

- *Bit Error*: Ocorre se o transmissor envia um *bit* dominante e detecta um *bit* recessivo ou envia um recessivo e detecta um dominante. Não se considera um *Bit Error* quando isso é verificado no *Arbitration Field* ou *Acknowledge Slot*, em que isso já faz parte da funcionalidade do protocolo. Em caso de erro, gera-se o *Error Frame* e a mensagem é reenviada.

- *Stuff Error*: O CAN usa o método de transmissão Non-Returnn-to-Zero (NRZ). No método *Return-to-Zero* (RZ), o sinal retorna para 0 para cada bit de informação, assim o sinal é *self-clocking*, isto é, não exige um *clock* adicional para fins de sincronização. No CAN não há, portanto, esse retorno para zero, logo a informação a ser transmitida sempre estará no bus, e os dispositivos somente sincronizam na transição de um bit recessivo para dominante. O CAN automaticamente coloca um bit de nível lógico oposto quando há mais de 5 bits consecutivos iguais, e isso é utilizado pelos dispositivos para sincronização.

Dessa forma, o erro é gerado quando se detecta 6 ou mais bits consecutivos de mesmo valor, nesse caso um *Error Frame* é gerado e a mensagem é repetida.

A detecção de algum tipo de erro se torna pública na rede através dos *Error Frames* ou *Error Flags*. A mensagem com erro é reenviada assim que o *bus* estiver desocupado e o nó vencer novamente a arbitragem. Para diferenciar falhas temporárias de permanentes, o CAN possui dois contadores de erros associados à cada nó, o REC (*Receive Error Counter*) e o TEC (*Transmission Error Counter*). Esses contadores são incrementados a cada falha e decrementados para cada sucesso de transmissão ou recepção. Dependendo do valor desses contadores, um dispositivo pode estar em 3 estados de falha:

- Error-Active: O nó estará nesse estado caso tenha o valor de TEC e REC abaixo de 128 e pode enviar *Active Error Flags*, que são 6 bits dominantes consecutivos. Isso gera um Stuff Error em todos os receptores, fazendo-os enviar seus respectivos *Error Flags*, chamados de *Error Echo Flags*, podendo totalizar mais de 12 bits dominantes. O estado *Error-Active* indica condições normais, permitindo o dispositivo a transmitir e receber mensagens sem restrições.

- Error-Passive: Um nó passará para esse estado quando o TEC ou o REC estiver maior que 127. No lugar do *Active Error Flags*, poderá somente enviar *Passive Error Flags*, 6 bits recessivos consecutivos. Caso haja sucesso na transmissão poderá gerar os *Error Echo Flags*, porém se for interrompido por algum bit dominante, terá de esperar que o bus fique desocupado por 8 bits antes de retransmitir.

- Bus-Off: Estarão nesse estado os nós que tiverem somente o TEC maior que 255. Os dispositivos nessa condição não podem enviar ou receber mensagens ou Error Flags de qualquer tipo. Há, porém, uma sequência de recuperação definida no protocolo para que os nós nesse estado possam retornar à condição de *Error-Active*, podendo reiniciar as transmissões caso a falha tenha sido removida.

O CAN é, pois, um protocolo voltado para a transmissão de dados relativamente pequenos com alta confiabilidade. Pelo fato de não ser baseado em endereçamento, não há necessidade de modificar a mensagem para transmissões de nó para nó ou multicast. O sistema de confinamento de falha não permite que um único dispositivo com falha possa colocar a rede abaixo, garantindo banda para que mensagens críticas sejam enviadas. Todas essas vantagens são o motivo da expansão de seu uso para além dos veículos automotivos, sendo sempre visto como uma alternativa no desenvolvimento de sistemas embarcados.

2.4.2 O Protocolo SPI

Considerando o aumento da complexidade das aplicações da microeletrônica, é muito comum hoje que se conectem diversos dispositivos, nos quais se incluem diferentes microcontroladores, para que se possa obter uma determinada funcionalidade. O SPI é um protocolo que surgiu com o intuito de facilitar essa comunicação entre múltiplos processadores. Uma de suas principais vantagens é que pode ser implementado inteiramente por software, permitindo assim comunicar um processador que possui o hardware do SPI com um que não possui.

Todo sistema composto por esse protocolo possui um mestre e um ou mais escravos. O mestre é o dispositivo que providencia o clock que sincroniza a comunicação, portanto os escravos são os que recebem esse sinal.

Basicamente 4 sinais são necessários para compor uma comunicação SPI: MOSI (*Master-Out / Slave-In*), MISO (*Master-In / Slave-out*), SCK (*Serial Clock*), e SS (*Slave-Select*) .

O pino MOSI é designado como um pino de saída para o mestre e entrada para o escravo. O mestre utiliza esse pino para enviar mensagens ao escravo, sendo que os bits mais significativos são enviados primeiro.

O pino MISO é um definido como entrada no mestre e como saída no escravo, logo o escravo o utiliza para enviar mensagens ao mestre.

Toda comunicação é sincronizada por um clock, e um bit é enviado a cada pulso de clock. Cada mensagem é composta por um byte, logo requere 8 pulsos de clock para ser enviada.

O SS é utilizado pelo mestre para escolher com qual dos escravos se comunicará, logo diversos pinos podem ser utilizados para essa função, sendo que são definidos como saída no mestre e entrada no escravo. Esse sinal é ativo em 0, assim deve ser colocado nesse valor antes do início dos pulsos de clock.

Um dos elementos principais do funcionamento do SPI é o SPI *Data Register*, um registrador de 8 *bits*. O mestre e cada escravo possuem seus próprios registradores, os quais estão ligados pelo MOSI e MISO. Considerando um mestre e um escravo, forma-se um registrador de 16 *bits* contínuos. Quando uma mensagem é enviada, esse registrador é deslocado de 8 *bits*, trocando o conteúdo dos 2 registradores de 8 *bits*.

O SPI permite selecionar a polaridade e a fase do serial clock a partir de 2 parâmetros, o CPOL (*Clock Polarity*) e CPHA (*Clock Phase*), havendo 4 possíveis configurações.

Quando o CPHA é 0, a amostragem do dado é feita na primeira borda de *clock* depois

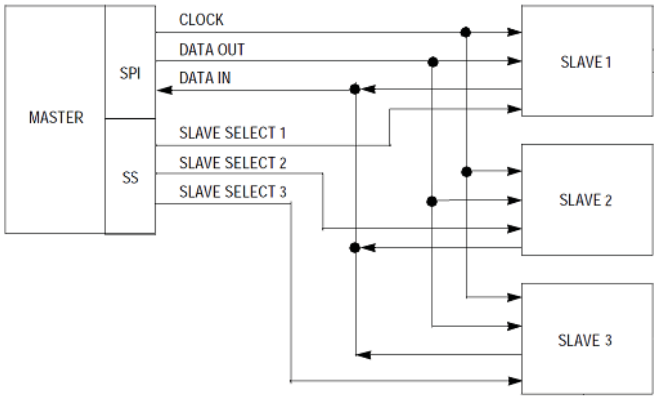


Figura 2.2: Protocolo SPI (Serial Peripheral Interface).

que o SS é colocado em 0. Na segunda borda, ocorre o deslocamento de um bit nos shift registers, e esse processo se repete nas 16 bordas que compõem a transmissão.

Alguns hardwares requerem que primeira borda de SCK ocorra a antes que o dado esteja disponível no pino de saída. Nesse caso o CPHA deve ser 1, e a amostragem ocorre na segunda borda de clock.

Quando CPOL = 0, o nível lógico 0 é produzido em regime pelo pino SCK quando não houver transmissão. Analogamente, para CPOL = 1 o nível lógico de regime será 1.

O SS pode permanecer ativo em transmissões sucessivas no caso de CPHA = 1. Porém, no caso de CPHA =0, o sinal SS é utilizado para disponibilizar o primeiro bit mais significativo na porta.

2.5 O Microcontrolador TINI

O microcontrolador TINI Maxim DS80C400 da Dallas Semiconductor é um dispositivo 8051 que oferece como periféricos um 10/100 Ethernet MAC, 3 portas seriais, um controlador CAN 2.0B, 1-Wire Master e 64 pinos de Entrada/Saída. Apresenta uma pilha TCP Ipv4/6 com capacidade para até 32 conexões simultâneas com taxa máxima de transferência de 5Mb/s. Seu clock máximo é de 75MHz, o que resulta em um tempo mínimo de ciclo de instrução de 54ns.

Suas principais aplicações são:

- Controle e Automação Industrial

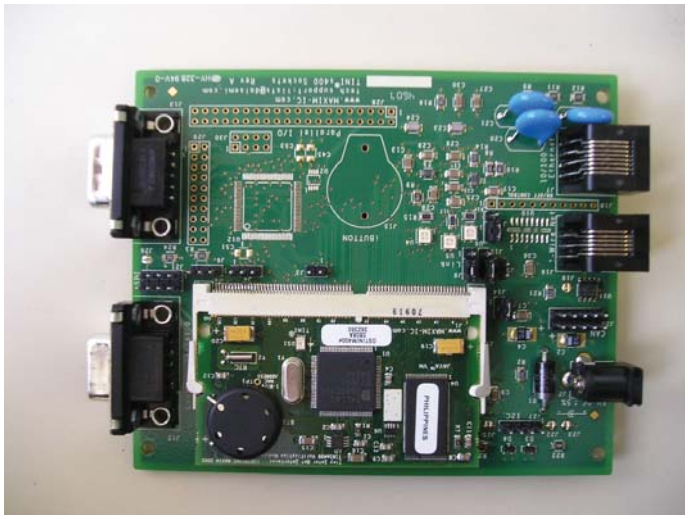


Figura 2.3: Microcontrolador TINI

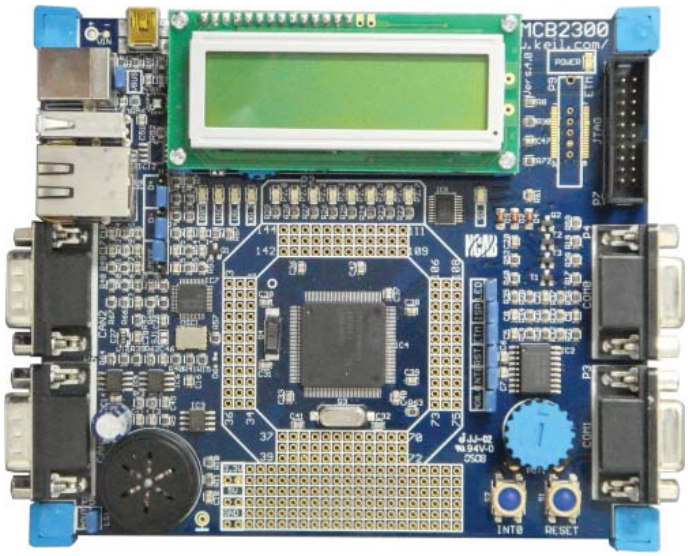


Figura 2.4: Placa de desenvolvimento MCB2300 com Microprocessador ARM 7

- Monitoramento de Ambiente
- Sensores de rede
- Automação residencial ou de escritório

Os aplicativos podem ser programados em 3 linguagens: C, Assembly (8051) ou Java (JDK 1.2.2 a 1.4). Os programas serão desenvolvidos em um PC e carregados no dispositivo por conexão serial, utilizando o software JavaKit fornecido.

2.6 O Sistema Baseado no Microcontrolador ARM7

O ARM7 é um microcontrolador com arquitetura RISC de 32 bits e vem se constituindo como um dos microcontroladores mais utilizados. Vários fabricantes produzem o mesmo microcontrolador mas com diferentes configurações internas de Entrada/Saída [5].

O ARM7 é bastante utilizado na indústria automobilística em aplicações com barramento CAN, e se caracteriza por ter alta conectividade, usualmente contendo protocolos *UART*, *I2C*, CAN e *Ethernet* além de entradas A/D, saídas D/A e Entrada/Saídas digitais (configurações são bastante variáveis de acordo com marca e modelo).

A placa de desenvolvimento escolhida é a MCB2300 da *Keil, Inc* (Figura 2.4). A Placa contêm um microcontrolador ARM 7 2378 NXP com *clock* de 12MHz. O sistema contem 2 portas CAN e uma Porta *Ethernet*.

2.7 O Sistema Operacional de Tempo Real RTX

O software de sistemas embarcados geralmente é desenvolvido com a utilização de interrupções e de um *looping* rodando na função *main* do código fonte do programa. Configuram-se *timers*, e criam-se interrupções para tratar o problema de sensoriamento, atuação, e comunicação com outros dispositivos. Embora muitos programas ainda são implementados dessa forma, para usufruir do avanço das técnicas de projeto e estruturação de software, utilizaremos nesse trabalho um sistema operacional de tempo real no microcontrolador ARM.

O sistema operacional utilizado é o RTX. Com ele, blocos funcionais do projeto podem ser desenvolvidos como processos que são escalonados pelo RTX. Cada processo pode ser desenvolvido e testado isoladamente em relação a outros processos, tornando o trabalho de implementação mais fácil. As vantagens de se usar um sistema operacional de tempo real no microprocessador são uma abordagem orientada a objeto e o suporte a operações multitarefa.

O RTX consiste em um escalonador que suporta os modos *Round-Robin*, preemptivo, e cooperativo para as tarefas. Além disso, o sistema operacional permite o gerenciamento de tempo e memória, e comunicação entre diferentes tarefas com auxílio de *triggering*, semáforos, *Mutex*, e um sistema de caixa de mensagens.

2.7.1 Tasks (Tarefas)

A elaboração de um programa em linguagem C típica é feita por meio de métodos que são chamados para realizar determinada operação, e que retornam para a função que os chamou. No sistema operacional de tempo real, a unidade básica de execução é a "Tarefa", ou *task*.

Uma *task* se parece bastante com um método, porém é necessário que ela tenha um *looping* sem condição de parada. Dessa forma, cada *task* funciona como um pequeno programa que é executado no sistema operacional. Dentro do mesmo, há um laço que é repetido durante um tempo intederminado.

```
__task void task(void) {
for (;;) {
// Código
}
}
```

Um programa baseado no sistema operacional de tempo real possui então várias *tasks*, que são controladas pelo escalonador do sistema. Num dado microprocessador, apenas uma *task* pode estar sendo executada em um instante. Dessa forma, um processo sempre está em um dos quatro estados básicos: *Running*, *Ready*, *Waiting*, ou *Inactive*. O RTX possui alguns métodos para comunicação entre as *tasks*, sendo eles: eventos, samáforos, *Mutex*, e *Mailbox*.

2.7.2 Eventos

Cada *task* é criada com dezesseis *flags* de evento. Estas *flags* estão armazenadas no Bloco de Controle de *tasks*. É possível interromper momentaneamente a execução de uma *task* colocando-a no modo *Waiting* até que um particular grupo de *flags* atinja uma condição necessária. Quando isto ocorre, a *task* volta à condição *Ready*, e é escalonada pelo RTX. É possível definir um período para que a *task* que está no modo *Waiting* retorne à condição *Ready*.

2.7.3 Semáforos

O uso de semáforos permite sincronizar atividades entre duas ou mais *tasks*. Em termos gerais, um semáforo contém um número de *tokens*. Quando uma *task* é executada, ela faz um pedido ao Sistema Operacional para adquirir um *token*. Se o Sistema possui um ou mais

tokens disponíveis, a *task* será executada e o número de *tokens* será decrementado em uma unidade. Se a *task* realiza o pedido ao Sistema, mas não há *tokens* disponíveis, ela entra na condição *Waiting* até que um *token* esteja disponível. Uma tarefa pode também devolver um *token* para o Sistema.

Desse modo, semáforos são usados para controlar o acesso aos recursos do Sistema Operacional. Antes que um processo tenha acesso aos recursos, ele precisa adquirir um *token*. Se nenhum está disponível, ele espera. Quando terminou de utilizar os recursos, ele devolve o *token*.

Para exemplificar uma aplicação do uso de semáforos, pode-se considerar a sincronização de duas *tasks*. Para isso, inicializaremos o Semáforo com um *token*, e executaremos as duas *tasks*. Em certo ponto do programa, uma *task* pedirá o *token*, e continuará sua execução. A segunda *task* também tentará obter o *token*, porém, este já estará ocupado. Entretanto, a primeira *task* devolverá o *token* para o Sistema, e quando isto ocorrer, a segunda *task* irá deixar o estado *Waiting* e entrará na condição *Ready*. Uma vez em *Ready*, o escalonador se encarregará de rodar o processo.

Outras aplicações de Semáforos são: assegurar que determinada *task* execute antes de outra; realizar o papel de multiplexador limitando o número de *tasks* que acessam recursos do sistema; realizar o "encontro"de *tasks*, ou seja, duas ou mais tarefas irão alcançar certo ponto do programa, e irão esperar até que as outras também cheguem neste ponto.

2.7.4 Mutex

Mutex vem de *Mutual Exclusion*, ou seja, exclusão mútua. Esta ferramenta é uma versão mais especializada dos Semáforos, pois possui o mesmo princípio de funcionamento, exceto por ser inicializado com apenas um *token*. Sua principal aplicação é controlar o acesso a um recurso como um periférico. Dessa forma, o *token* de um *Mutex* é binário e limitado.

2.7.5 Mailbox

Os métodos de comunicação entre *tasks* apresentados até agora, visam apenas a sincronização entre diferentes partes de códigos do programa. Porém, existem situações onde é necessária a transferência de dados entre as *tasks*. Esta tarefa pode ser realizada por meio de escrita e leitura em variáveis globais. Entretanto, tal implementação abre precedentes para erros imprevisíveis que não garantem a integridade da informação. Por este motivo, é necessária a formalização na comunicação assíncrona entre *tasks*.

O Sistema Operacional de Tempo Real possui uma caixa de mensagens que possibilita a transferência dos tipos de dados *byte*, inteiro e dados com largura de palavras, com comprimento fixo ou variável. Um *Mailbox* consiste em um bloco de memória formatado como *buffers* de mensagem, e um conjunto de ponteiros para cada *buffer*. Dessa forma, quando se envia uma mensagem para o *Mailbox*, o *slot* correspondente à mensagem fica travado, podendo ser liberado após a leitura dos dados enviados.

A configuração de um *Mailbox* é feito da seguinte forma: Declara-se o ponteiro de mensagem como um *array* de *unsigned integers*, onde se define o número de *slots*. Em seguida, declara-se a estrutura que acomodará os dados a serem transferidos, geralmente utilizando *struct*. Após definir o formato do *slot* de mensagem, deve-se reservar um bloco de memória suficientemente largo para acomodar dezesseis *slots* de mensagem. Esses blocos de memória são formatados com funções disponíveis no Sistema Operacional.

Para realizar a transferência de mensagem entre duas *tasks*, cria-se um ponteiro do tipo da estrutura da mensagem na *task* que envia a informação, e aloca-se o ponteiro no *slot* de mensagem. Desse modo, preenche-se o *slot* de mensagem com os dados a serem transferidos com auxílio do ponteiro. Com este procedimento, o Sistema trava o *slot* de mensagem. Para receber o dado, cria-se um ponteiro do tipo da estrutura mensagem na *task* que recebe a informação, e aloca-se o dado que está no *slot* para o ponteiro. Finalmente, o *slot* é liberado.

2.7.6 Travamento e Destravamento de Tasks

Num software de controle, é necessário assegurar que um trecho de código é executado sem sofrer interrupções causadas pelo escalonador. Em uma aplicação baseada no Sistema Operacional RTX, não se pode garantir que um determinado trecho de código será executado ininterruptamente. Dessa forma, devem-se usar as funções de travamento e destravamento disponíveis no sistema, que permitem ou proíbem o escalonador de interferir na execução de determinada *task*.

```
tsk_lock ();
trecho_de_código_crítico ();
tsk_unlock ();
```

2.7.7 Configuração

A configuração do Sistema Operacional de Tempo Real RTX é realizada pela edição do arquivo RTXConfig.c específico para o microcontrolador LPC2378. Este arquivo vem previamente configurado pelo fabricante, e apresenta todas as opções necessárias para o usuário na forma de um menu de seleção.

2.7.8 Opções de Escalonamento no RTX

O Sistema Operacional de Tempo Real RTX suporta escalonamento Preemptivo, *Round-Robin*, e cooperativo. Entretanto, para a maioria das aplicações, utiliza-se um modelo misto chamado escalonamento *Round-Robin* preemptivo.

2.7.8.1 Escalonamento Preemptivo

O escalonamento preemptivo é configurado desabilitando-se a opção *Round-Robin*, e definindo-se uma prioridade diferente para cada *task*. Dessa forma, um processo será executado até que seja interrompido por outro de maior prioridade, ou pelo próprio Sistema Operacional. Dessa forma, existe uma hierarquia na execução dos processos, com cada tarefa consumindo tempos de execução variáveis.

2.7.8.2 Escalonamento Round-Robin

O escalonamento baseado em *Round-Robin* é configurado habilitando-se a opção *Round-Robin* no arquivo RTXConfig.c, e definindo-se a mesma prioridade para cada *task*. Neste tipo de escalonamento, cada *task* será executada durante o mesmo período de tempo fixo, ou até que seja interrompida pelo Sistema Operacional.

2.7.8.3 Escalonamento Cooperativo

O escalonamento cooperativo é configurado desabilitando-se a opção *Round-Robin*, e definindo-se a mesma prioridade para todas as *tasks*. Dessa forma, cada *task* executará até que seja interrompida pelo próprio sistema operacional, ou utilize uma função para explicitamente passar para outra *task*.

3 METODOLOGIA

3.1 Concepção da Plataforma de Testes

A concepção da plataforma de testes é ilustrada na Figura 3.1. Observa-se a presença de plataformas mecânicas a serem controladas, *drivers* de acionamento, microprocessadores TINI atuando como *bridge* de rede, microcontroladores ARM7 que executam o sistema operacional de tempo real RTX e implementam os controladores, e rede de comunicação CAN.

As plataformas mecânicas são constituídas de dois motores de corrente contínua (DC) juntamente com seus *drivers* de potência. Um deles é o motor que se deseja controlar em malha fechada, e o segundo motor DC é utilizado como um gerador de distúrbios de torque.

O motor principal está acoplado a sensores de torque (embutido no driver) e posição (*encoder*). O *driver* de acionamento possui saídas que permitem o acesso da velocidade de rotação do *encoder* acoplado aos motores. Os sinais dos sensores se conectam a um microcontrolador TINI DS80C400 (Maxim-Dallas Semiconductor). O objetivo da utilização desse microcontrolador é o transporte das informações dos sensores para uma rede de comunicação CAN. Da mesma forma, mensagens CAN do sinal de atuação são convertidos pelo TINI para o protocolo SPI de forma a levar as informações ao *driver*.

As tarefas que implementam os controladores são executadas no microcontrolador ARM7 que possui o sistema operacional de tempo real RTX nele configurado.

Para os experimentos de sistemas de controle distribuído, os módulos poderão ser interconectados em rede. Dessa forma, podemos conectar várias plataformas de teste numa rede de comunicação e distribuir os processos adequadamente em uma CPU ou várias CPUs simultaneamente.

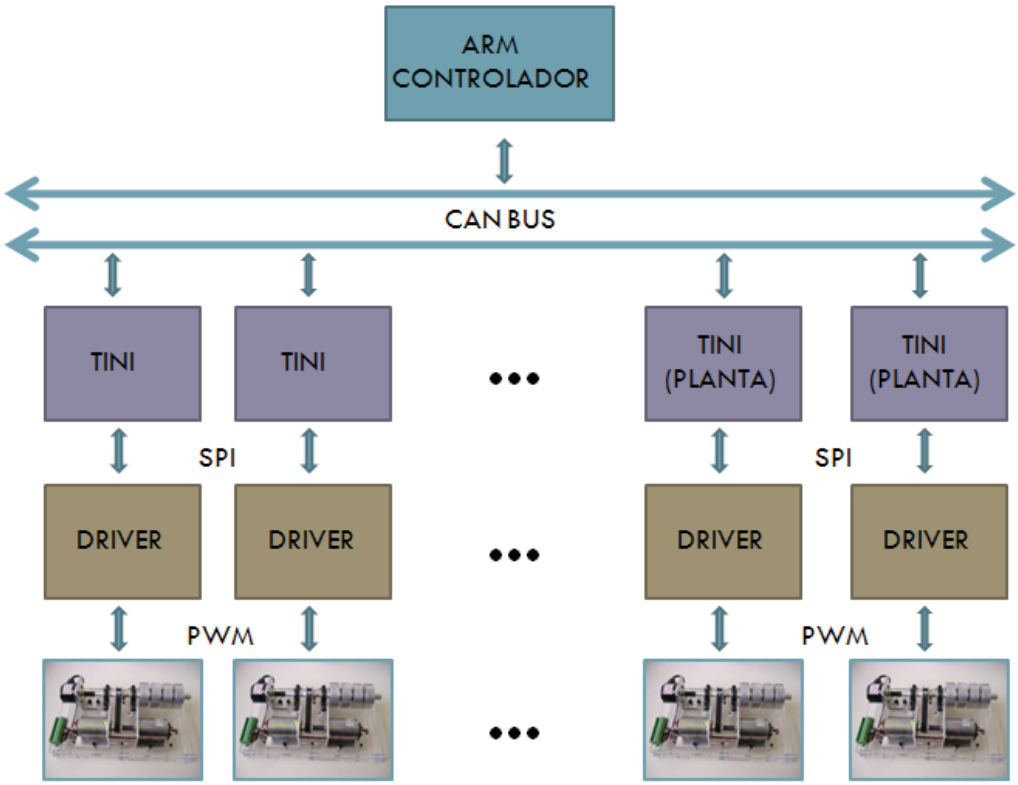


Figura 3.1: Concepção da plataforma de testes

3.1.1 A Plataforma Mecânica

Quatro unidades da plataforma mecânica foram inteiramente projetadas e construídas. A figura 3.7 mostra uma plataforma mecânica em estágio final de construção. As figuras 3.2 e 3.3 apresentam os modelos em *software* CAD do kit. Foram utilizados materiais de baixo custo como acrílico para a base e alumínio para as peças. A tabela da figura 3.9 apresenta os custos relacionados à aquisição de materiais para a construção do kit.

Os motores utilizados são provenientes de *scanners* fabricados pela *Canon*, modelo FN38, e seus gráficos de torque em função da velocidade são apresentados na figura 3.10.

Os mancais utilizados no projeto são fabricados pela Igus, e são do modelo KSTM-10 (figura 3.4).

A polias são fabricadas em alumínio pela Schneider, e proporcionam uma relação de:

$$\frac{D_{primitivoMaior}}{D_{primitivoMenor}} = \frac{28.65}{16.23}$$

conforme (figura 3.5).

O encoder série 40 é fabricado pela Hohner e possui uma resolução de 1024 pulsos por

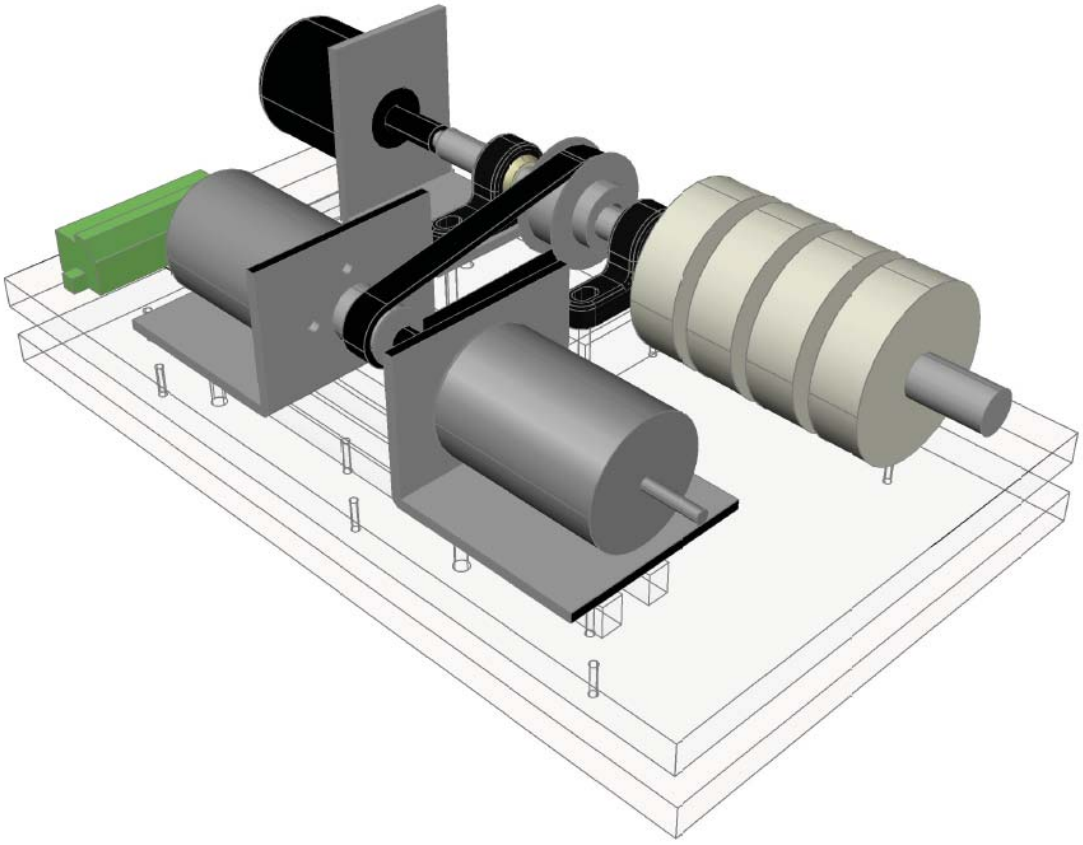


Figura 3.2: Modelo em três dimensões do kit mecânico.

revolução. Sua construção é feita em alumínio e possui eixo com diâmetro de 6mm (figura 3.6).

3.1.2 A Plataforma Elétrica

A plataforma elétrica (figura 3.11) foi adquirida pelo Laboratório de Sistemas Embarcados do PMR (Departamento de Engenharia Mecatrônica), e consiste na fonte de alimentação, drivers de acionamento, microcontroladores TINI e ARM7.

O driver de acionamento (figura 3.12) foi adquirido separadamente, é do modelo PN420, e pode ser utilizado em dois modos de operação: Analógico e Digital. O modo de operação é selecionado pelo *jumper* JP1, onde a presença do *jumper* configura o modo analógico, e a sua ausência aciona o modo digital. Há um *led* vermelho que pisca com uma frequência maior quando o *driver* está configurado para operar no modo analógico.

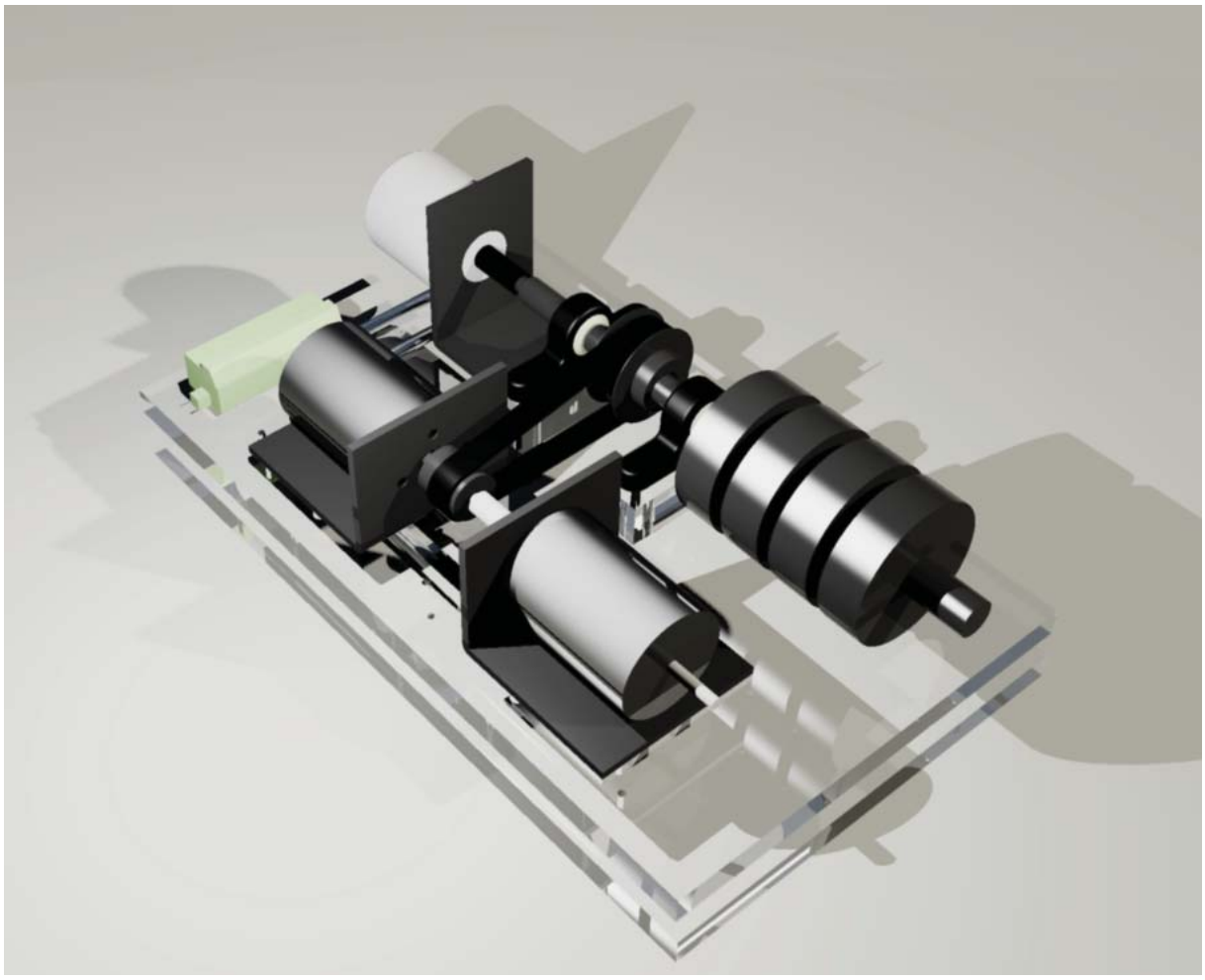
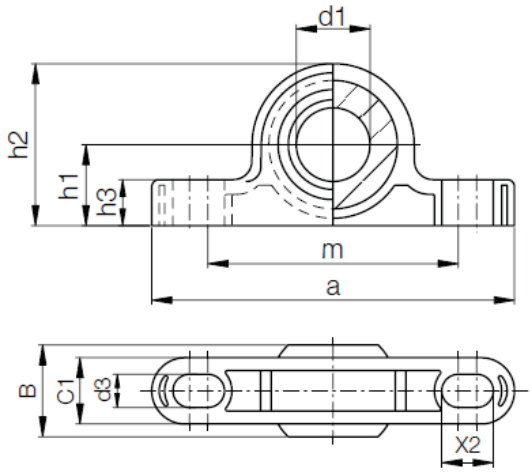


Figura 3.3: Modelo renderizado em três dimensões do kit mecânico.

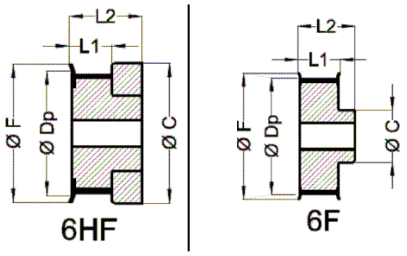


Dimensions [mm]

igubal® - Pillow Block Bearing KSTM

Part Number	a	d1 E10	B	C1	h1	h2	m	h3	d3	X2	Max. Pivot
KSTM-10	62	10	14	10,5	14	26	46	7,5	5,5	8	25°

Figura 3.4: Mancal KSTM-10 fabricado pela Igus.



Referência da Polia	Numero de Dentes	Tipo de Polia	Dia. Primitivo Ø Dp	Dia. Flanges Ø F	Dia. do Cubo Ø C	L1	L2	Furo Min. Ø	Furo Max. Ø	Peso Aprox. (Kg)
17 3M 9	17	6HF	16,23	17,0	17,0	11,7	17,5	3,0	5,0	0,010
30 3M 9	30	6F	28,65	32,0	20,0	13,2	19,2	6,0	11,0	0,028

Figura 3.5: Polias modelos 173M9 e 303M9 fabricados pela Schneider.

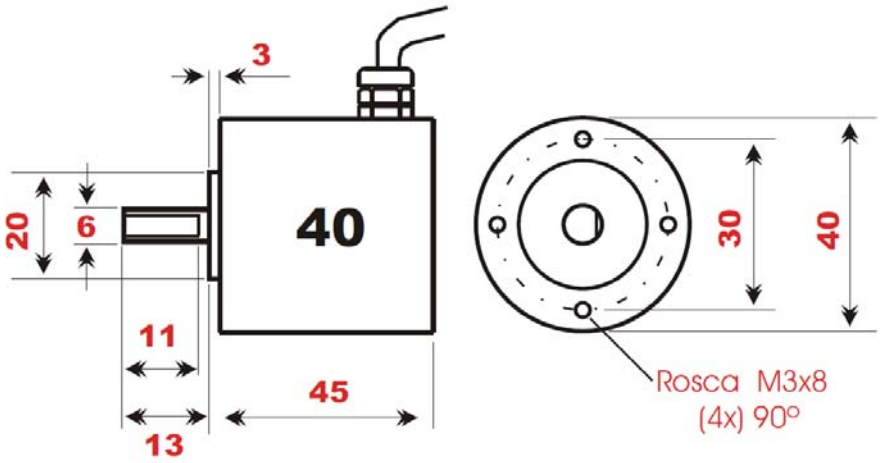


Figura 3.6: Encoder série 40, fabricado pela Hohner.

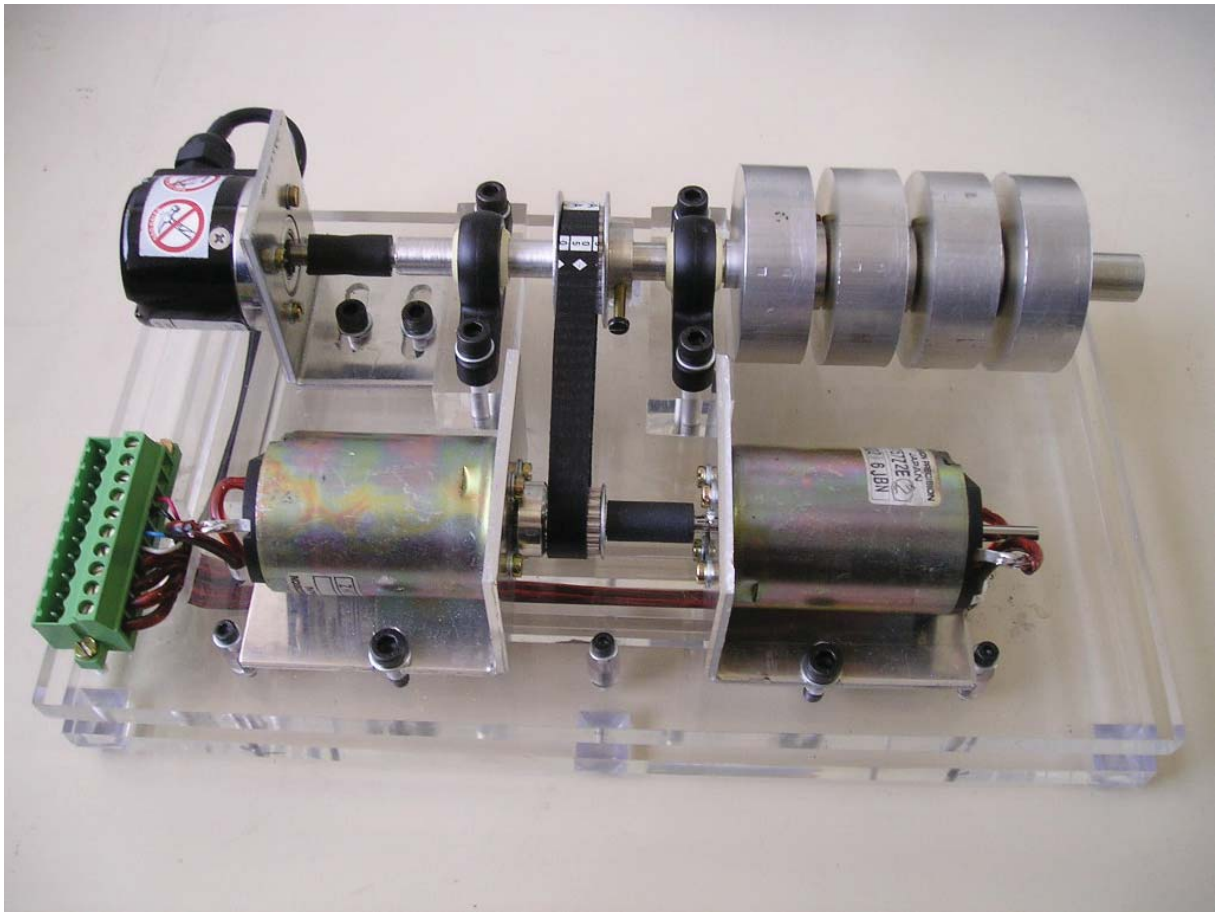


Figura 3.7: Plataforma de testes.

Figura 3.8: Conjunto de quatro plataformas de testes.

Material	Custo (R$)
Mancais	16,00
Polias	73,50
Correia	20,00
Acrílico	95,00
Conectores	31,90
Alumínio	20,00
Outros	23,60
TOTAL	**280,00**

Figura 3.9: Tabela de custos relacionados à matéria prima.

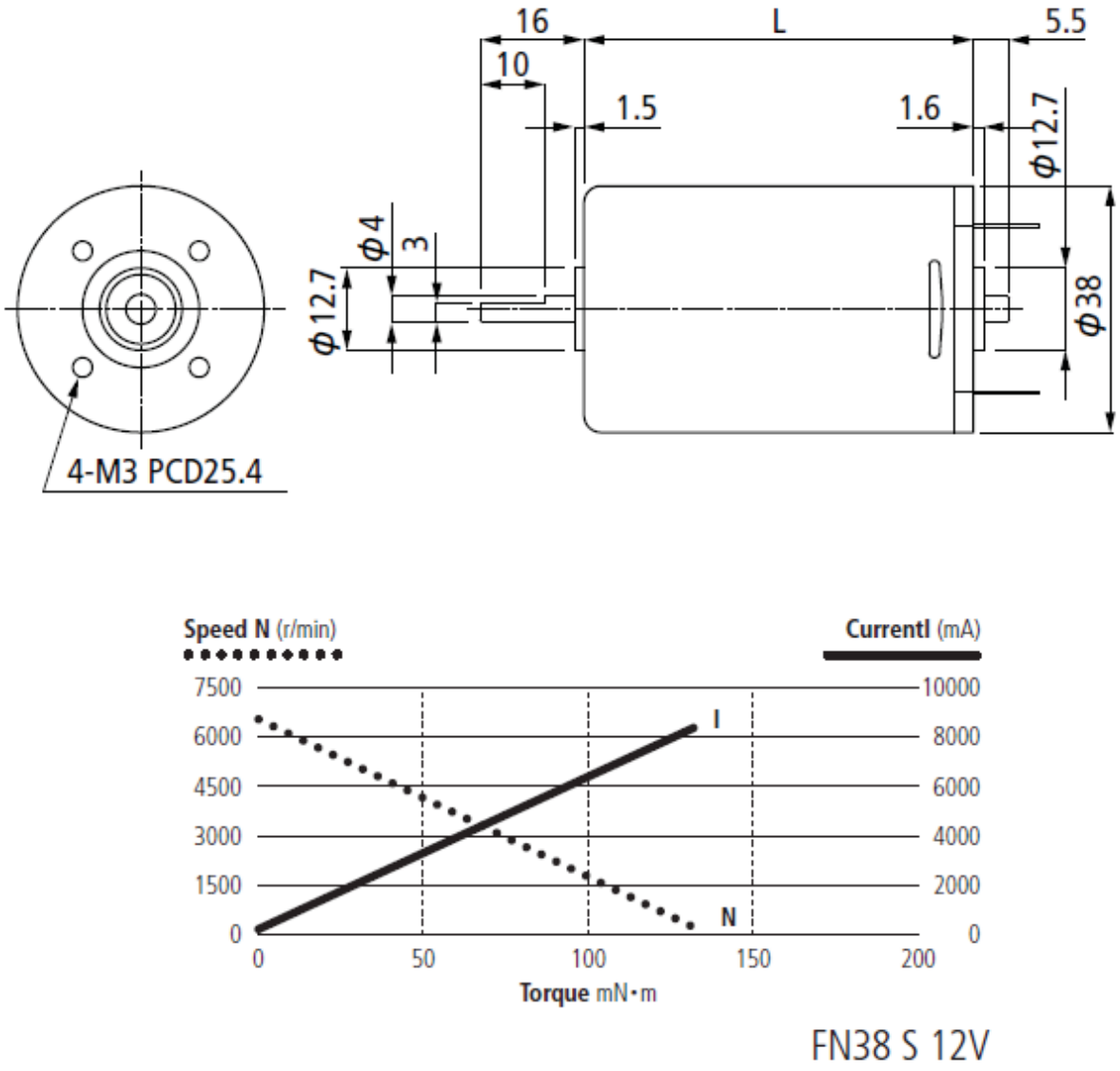


Figura 3.10: Desenhos e curva do motor *Canon* FN38 utilizado no projeto.

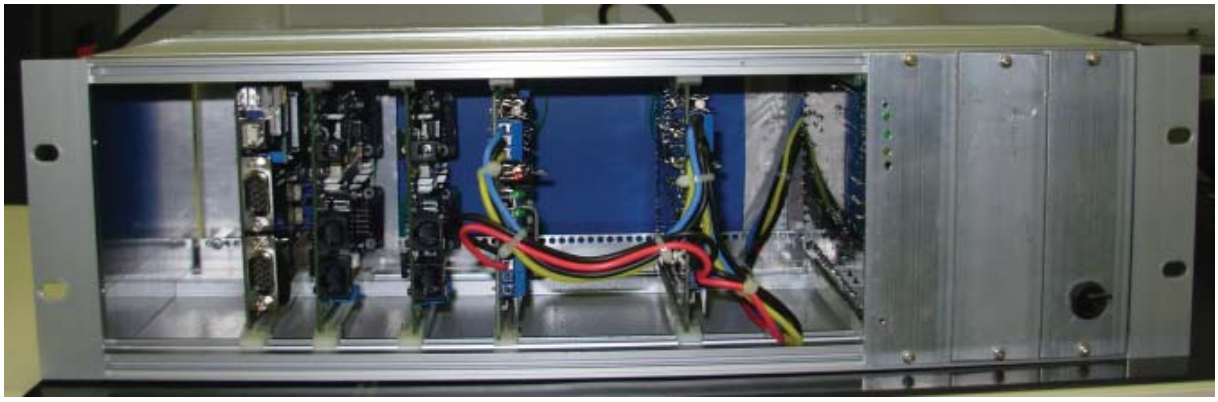


Figura 3.11: Plataforma Elétrica com fonte, drivers de acionamento, e microprocessadores TINI.

3.1.2.1 Driver de Acionamento: Modo Analógico

O modo analógico permite que o usuário entre com uma tensão correspondente ao sinal PWM e à direção de rotação no par de pinos CN6. O sinal deve estar dentro dos limites de tensão -2,5v e +2,5, sendo que o sinal determina a direção de rotação.

A velocidade de rotação do encoder, em RPM, é acessada diretamente no par de pinos CN5. Semelhantemente ao pino CN6, a tensão proporcional à rotação do eixo do encoder pode variar entre -2,5v e +2,5v.

3.1.2.2 Driver de Acionamento: Modo Digital

O modo digital se comunica com o meio externo por meio do protocolo SPI. Nesse caso, utiliza-se o par de pinos CN2 e CN8 pra entrada dos sinais de *clock*, *slave select*, *master in slave out*, e *master out slave in*. O funcionamento do protocolo SPI é apresentado num capítulo dedicado.

3.2 Modelagem e Identificação da Planta da Plataforma

3.2.1 Modelagem do Sistema

A plataforma a ser modelada consiste em um amplificador de tensão, um motor C.C., um sistema de engrenagens, uma carga inercial, e um *encoder* óptico. A modelagem permite que uma função de transferência entre a tensão de entrada $e_g(t)$ e a velocidade angular $\omega_{ENC}(t)$ no eixo do *encoder* seja estabelecida.

As funções de transferência do amplificador e do *encoder* são:

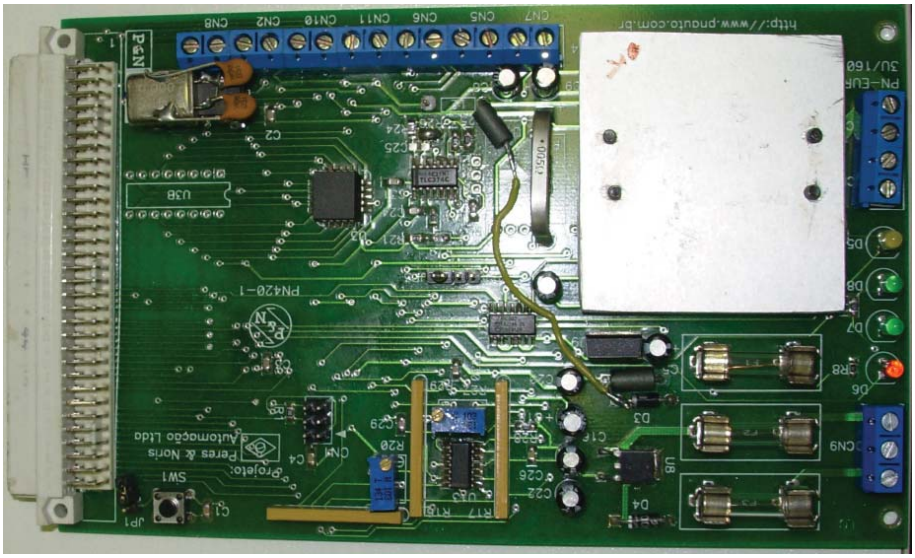


Figura 3.12: Driver modelo PN420 projetado pela Peres e Noris Automação Ltda.

$$G_{AMP}(s) = K_1 \tag{3.1}$$

$$G_{ENC}(s) = K_{ENC} \tag{3.2}$$

Sabemos que o motor utilizado será controlado por armadura, e possui corrente de campo constante. O torque do motor T é proporcional à corrente de armadura i_a.Dessa forma:

$$T = K_2 \cdot i_a \tag{3.3}$$

Sabe-se que a tensão induzida e_b é diretamente proporcional à velocidade angular $\frac{d\theta}{dt}$:

$$e_b = K_3 \cdot \frac{d\theta}{dt} = K_3 \cdot \omega \tag{3.4}$$

Onde K_2 é a constante de torque do motor, K_3 é a constante de força contra-eletromotriz do motor, e_b é a tensão induzida, e θ é o deslocamento angular do eixo de saída do motor.

A velocidade de rotação w do motor é proporcional à tensão de armadura e_a. A equação diferencial do circuito de armadura é dada por:

$$L_a \cdot \frac{di_a}{dt} + R_a \cdot i_a + e_b = e_a \tag{3.5}$$

onde L_a é a indutância da armadura e R_a é a resistência da armadura.

A equação de equilíbrio de torque no eixo do motor é dada por:

$$J_0 \cdot \frac{d^2\theta}{dt^2} + b_0 \cdot \frac{d\theta}{dt} = T = K_2 \cdot i_a \tag{3.6}$$

ou, em termos de ω:

$$J_0 \cdot \frac{d\omega}{dt} + b_0 \cdot \omega = T = K_2 \cdot i_a \tag{3.7}$$

onde J_0 e b_0 são a inércia e o atrito viscoso do eixo do motor, das massas inerciais, e das engrenagens.

Finalmente, com o auxílio das equações 3.1, 3.2, 3.4, 3.5, e 3.7, determina-se a função de transferência entre a saída correspondente à velocidade angular do eixo do *encoder* ω_{ENC} , e a entrada do gerador de funções e_g:

$$G_\omega(s) = \frac{K_1 \cdot K_2 \cdot K_{ENC}}{(L_a \cdot s + R_a) \cdot (J_0 \cdot s + b_0) + K_2 \cdot K_3} \tag{3.8}$$

Pode-se desprezar a indutância da armadura L_a, de forma a representar a equação 3.8 como uma função de transferência de primeira ordem:

$$G_\omega(s) = \frac{K_1 \cdot K_2 \cdot K_{ENC}}{R_a \cdot (J_0 \cdot s + b_0) + K_2 \cdot K_3} \tag{3.9}$$

Uma função desse tipo pode ser apresentada como:

$$G_\omega(s) = \frac{K_\omega}{T \cdot s + 1} \tag{3.10}$$

Desse modo, podemos modificar a equação 3.9 para a forma da equação 3.10, onde:

$$K_\omega = \frac{K_2 \cdot K_{ENC}}{R_a \cdot b_0 + K_2 \cdot K_3} \tag{3.11}$$

e,

$$T = \frac{R_a \cdot J_0}{R_a \cdot b_0 + K_2 \cdot K_3} \tag{3.12}$$

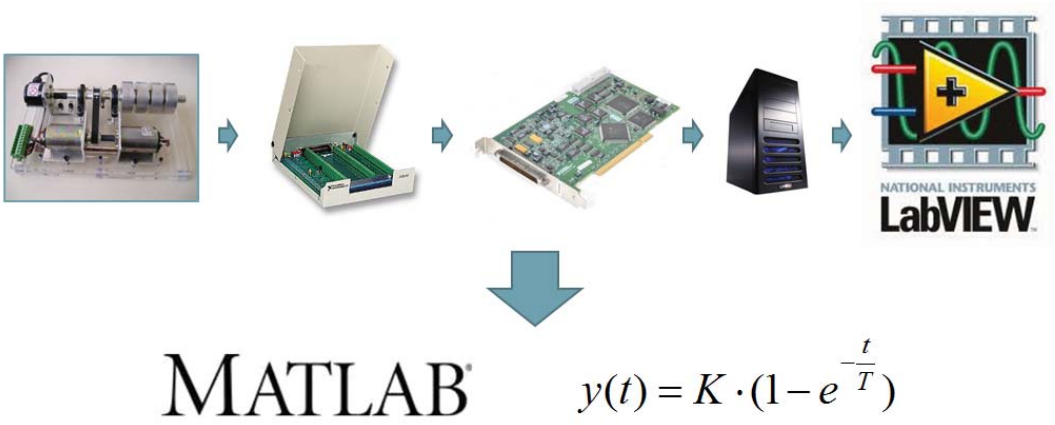


Figura 3.13: Esquema para identificação do sistema.

Figura 3.14: Placa de aquisição de dados PCI-6024E da National Instruments

3.2.2 Identificação do Sistema

O projeto do controlador a ser utilizado necessita primeiramente de um modelo da planta das plataformas construídas. Neste capítulo, será abordado o método de identificação utilizado. Um sistema de aquisição de dados é constituído de computador pessoal, placa de aquisição de sinais, condicionador de sinais, sensores e atuadores, e software de aquisição e tratamento de sinais.

O computador pessoal utilizado possui um AMD Athlon XP 1700+ 1.47Ghz e sistema operacional Windows XP. Dessa forma, é possível cumprir a função de aquisição dos dados em uma taxa de 250 escaneamentos por segundo.

A placa de aquisição de dados disponível no laboratório é a PCI-6024E (figura 3.14), fabricada pela *National Instruments*, e possui dezesseis canais de entradas analógicas, dois canais de saídas analógicas, oito pinos de entrada e saída digital, e conector externo de 68 pinos (figura 3.15).

Para a aquisição dos sinais foi utilizado o software *LabView* (*Laboratory Virtual Instrument Engineering Workbench*) com um Instrumento Virtual (VI) disponível no Laboratório de

Figura 3.15: Conector externo de 68 pinos da placa de aquisição.

Sistemas Embarcados. O tratamento dos dados adquiridos foi feito com o software MATLAB.

A montagem feita é esquematizada na figura 3.13, e consiste em ligar um gerador de funções com uma tensão fixa correspondente ao sinal de controle na entrada analógica do *driver* de acionamento. A saída do *driver* é uma tensão proporcional à velocidade de rotação no *encoder*, em RPM. Um canal da placa de aquisição é afixado na entrada do *driver* junto ao gerador de funções, e outro canal é afixado na saída do *driver*.

O gerador de funções é configurado para gerar uma onda quadrada entre os níveis de tensão 0 e 2.5v com um período suficientemente grande para o sistema entrar em regime estacionário. Os dados obtidos pelo software Labview compreendem desde o instante de tempo em que o degrau de entrada é aplicado até uma região em que o sinal já entrou em regime estacionário.

O modelo do sistema obtido na seção anterior é de primeira ordem. Dessa forma, utiliza-se o software MATLAB para traçar uma curva exponencial que aproxime os dados obtido e permita a identificação dos parâmetros K_ω e T da função de transferência 3.10.

3.3 Redes de comunicação

3.3.1 Configuração da rede CAN

3.3.1.1 O Protocolo CAN aplicado no TINI

Para configurar a rede CAN no TINI, foi utilizada uma biblioteca disponibilizada pela própria fabricante. Programou-se usando a versão em linguagem C, especificamente para o ambiente de desenvolvimento μVision. A implementação é feita por interrupção, permitindo que o CPU processe outras tarefas enquanto se comunica por esse protocolo.

O código abaixo se refere à função de inicialização da rede CAN. Nela é feita a escolha da taxa de transmissão de mensagem, no caso 50Kbps. Posteriormente, determina-se o tipo de mensagem como sendo *standard* no lugar de *extended*, pelo fato de que não há necessidade de envio de uma grande quantidade de dados por mensagem. Configura-se também o filtro de IDs para a recepção de *frames* de um determinado dispositivo.

```
int startupCAN()
 {
...
   //escolha do baud-rate
   if ((retval = can_settseg1(CAN_CONTROLLER,CAN_TSEG1)) != 0)
return retval;
   if ((retval = can_settseg2(CAN_CONTROLLER,CAN_TSEG2)) != 0)
return retval;
   if ((retval = can_setbaudrateprescaler(CAN_CONTROLLER,CAN_PRESCALER))
!= 0 ) return retval;

   if ((retval = can_setsynchronizationjumpwidth(CAN_CONTROLLER,CAN_SJW))
!= 0 ) return retval;
   if ((retval = can_setrxwriteoverenable(CAN_CONTROLLER,1)) != 0 )
return retval; //permite overwriting quando o buffer de mensagens esta
cheio

   junk32 = 0x00;
   if ((retval = can_set11bitglobalidmask(CAN_CONTROLLER,&junk32)) != 0)
return retval;

   config.ExtendedID = FALSE; //utilização de standard frame
   config.ID = CAN_RECEIVE_ID; //determinação de ID para recepção de
mensagem
   config.MemeEnable = TRUE;
   config.MdmeEnable = FALSE;

...

   //início de comunicação
```

```
   if ((retval = can_enablecontroller(CAN_CONTROLLER)) != 0)
return retval;

   return 0;
 }
```

Para o envio de mensagens é utilizada a função *can_sendframe*, cujas entradas são o número do controlador CAN e um *structure* do tipo *CanFrame*, o qual representa a mensagem a ser transmitida. Esse, como pode ser observado no código abaixo, possui parâmetros como o ID e o vetor de bytes para os dados.

```
typedef struct
{
    boolean RemoteFrameRequest;
    boolean ExtendedID;
    uint32_t ID;
    uint8_t Length; //tamanho dos dados
    char Data[8]; //dados
} CanFrame;
```

O valor numérico do sinal de atuação e de feedback é alocado em uma variável do tipo *float*, de 32 bits. No entanto, esse último deve ser convertido em um vetor de 4 *bytes* para ser associado ao vetor de dados no *CanFrame*. Como solução foi utilizado uma estrutura do tipo *union*, em que aloca-se 2 variáveis em mesmas posições de memória.

Considerando o código a seguir, uma vez que uma das variáveis é alocada, automaticamente a outra também será, podendo ser acessada para uso. Dessa forma converte-se o float em um vetor de bytes para envio e o contrário para recepção de mensagens.

```
typedef union{

float f;
unsigned char c[4];

}CanframeData;
```

3.3.1.2 O Protocolo CAN aplicado no ARM7

O sistema operacional de tempo real RTX possui um driver CAN que permite uma implementação de uma rede de forma rápida e fácil, e com uma enorme compatibilidade com outros dispositivos CAN. O sistema RTX possui bibliotecas com diversas funções que auxiliam o programador.

Para a configuração do controlador CAN, foi escrita a seguinte função:

```
void startup_CAN(){

//Seta o baudrate para o controlador CAN número 1 em 50k
CAN_init (1, 50000);

// CAN_rx_object ativa o recebimento de mensagens com determinado ID.
CAN_rx_object (1, 2,  ID1_IN, DATA_TYPE | STANDARD_TYPE);
CAN_rx_object (1, 2,  ID2_IN, DATA_TYPE | STANDARD_TYPE);
CAN_rx_object (1, 2,  ID3_IN, DATA_TYPE | STANDARD_TYPE);
CAN_rx_object (1, 2,  ID4_IN, DATA_TYPE | STANDARD_TYPE);
CAN_rx_object (1, 2,  ID5_IN, DATA_TYPE | STANDARD_TYPE);
CAN_rx_object (1, 2,  ID6_IN, DATA_TYPE | STANDARD_TYPE);

//Inicia o controlador CAN especificado, e o introduz na rede CAN
CAN_start (1);
}
```

A tarefa de receber mensagens CAN merece uma atenção especial, pois foi desenvolvida em uma *task* independente das correspondentes aos controladores. Dessa forma, utiliza-se semáforos para acessar as variáveis globais que acodomodam o conteúdo das mensagens recebidas.

De forma semelhante ao caso apresentado na seção anterior, foi utilizada a estrutura *Union* para permutar entre um dado float (32 bits), e um vetor de 4 bytes (4 x 8 bits).

A função escrita para recebimento de mensagens CAN é na forma:

```
if(CAN_receive (1, &msg_rece, 0x0000) == CAN_OK){
    in_id   = msg_rece.id;
```

```
if(in_id == ID1_IN){
if( OS_R_TMO != os_mut_wait(mutex1,0)){
in1.c[0] = msg_rece.data[3];
in1.c[1] = msg_rece.data[2];
in1.c[2] = msg_rece.data[1];
in1.c[3] = msg_rece.data[0];
os_mut_release(mutex1);
}
situation1 = 1;
}

//if(in_id == ){
// De forma semelhante para outros ID's
//}
}
```

O envio de mensagens para a rede é feita pelo seguinte trecho de código:

```
(...)
msg_send1.data[0] = U.c[3];
msg_send1.data[1] = U.c[2];
msg_send1.data[2] = U.c[1];
msg_send1.data[3] = U.c[0];
CAN_send (1, &msg_send1, 0x0000);
```

3.3.1.3 Conexões Elétricas

O cabo de conexão CAN é um par trançado com resistores terminadores de 120Ω. No *bus* CAN, o nivel lógico zero é representado pela máxima diferença de voltagem chamada de "dominante", e o nível lógico zero é representado pelo *idle state* chamado de "recessivo". A figura 3.16 ilustra a ponta do *bus* CAN com o resistor terminador.

3.3.2 Comunicação SPI

Tanto a biblioteca em Java como a em C para o SPI são importações de uma implementação nativa em *assembly*. Existe uma função para inicialização e outra para o envio e leitura

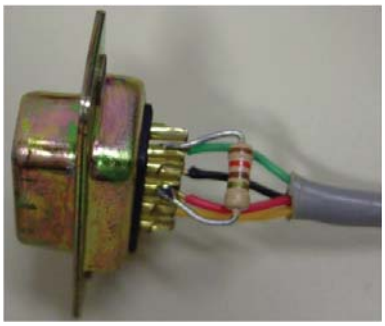


Figura 3.16: Ponta do cabo CAN com resistor terminador).

simltânea, cuja declaração pode ser vista a seguir. A vantagem é que, por ser implementada em baixo nível, atinge-se uma excelente precisão nas constantes de tempo de envio. Perde-se, porém, versatilidade com relação às diversas constantes de tempo que o protocolo poderia assumir.

```
void spi_xmit  (
unsigned char *   dataptr,
int   length,
unsigned char   delay,
unsigned char   options
)
```

Por problemas que serão citadas depois, como alternativa à biblioteca original, foi feito uma implementação do SPI em linguagem C utilizando interrupção. As constantes de tempo estavam imprecisas devido à problemas na periodicidade das interrupções, assim posteriormente foi feito uma customização da biblioteca nativa a partir da edição do programa em assembly.

3.4 Controle do Sistema

Nesse trabalho são avaliados três algoritmos de controle, sendo eles:

1. Controlador PI
2. Controlador PI com Compensação
3. Controlador PI dividido em Subtasks

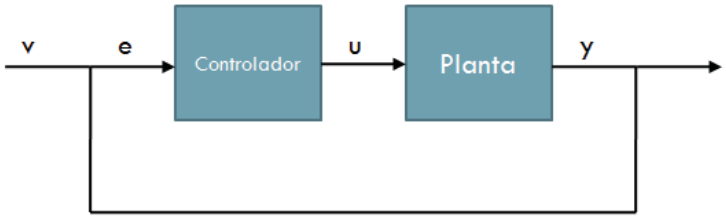


Figura 3.17: Looping de controle utilizado.

3.4.1 Controlador PI

Considerando-se a função de transferência de primeira ordem obtida na seção sobre modelagem e identificação de sistemas, utilizaremos controladores do tipo proporcional-integral (PI) para o controle das plantas, pois se pode demonstrar que controladores desse tipo são suficientes para controlar sistemas onde a dinâmica predominante é de primeira ordem [6].

$$G_c = K_p + \frac{K_i}{s}$$

O controlador é projetado no domínio do tempo contínuo e é calibrado com o auxílio do software MATLAB. Após isso, é discretizado utilizando-se o método de aproximação de Tustin, onde:

$$s = \frac{2 \cdot (z-1)}{T_a \cdot (z+1)}$$

A equação de diferença obtida e implementada no microprocessador ARM7 é:

$$u(k) = u(k-1) + (K_P + \frac{K_i \cdot T_a}{2}) \cdot e(k) + (\frac{K_i \cdot T_a}{2} - K_P) \cdot e(k-1) \tag{3.13}$$

Onde K_P, K_i, T_a, $u(K)$ e $e(K)$ são, respectivamente: o ganho da planta, o ganho da porção integral do controlador, o tempo de amostragem, o sinal de controle num instante K, e o sinal de erro num instante K.

3.4.2 Controlador PI com Compensação

A introdução de indeterminismos temporais, sobretudo a variação do período de amostragem, fazem com que o desempenho de um controlador seja bastante reduzido. A figura 3.18 mostra em linha tracejada o gráfico de Bode para o caso em que um período de amostragem é menor

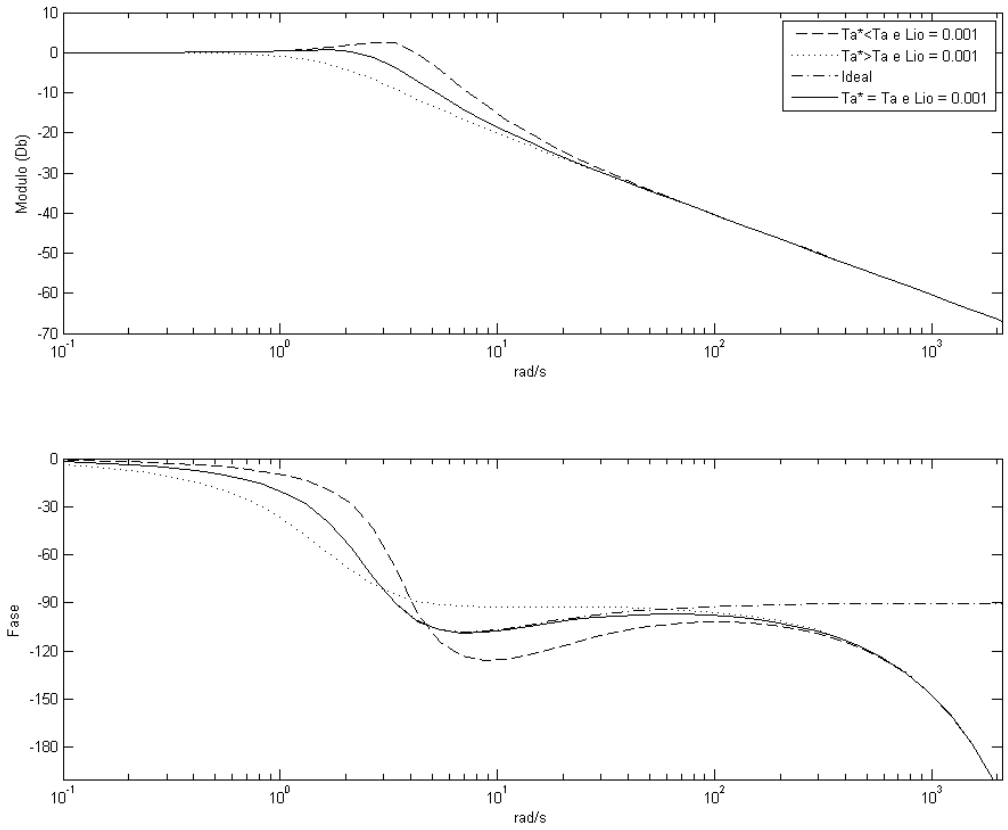


Figura 3.18: Diagrama de Bode ilustrando efeito de atrasos no período de amostragem.

do que o ideal, enquanto que o de linha pontilhada apresenta o mesmo gráfico para um período maior do que o estabelecido. A linha contínua corresponde ao traçado do gráfico para o tempo de amostragem ideal. Os três traçados anteriores possuem latência de entrada e saída constantes e diferente de zero. A linha traço-ponto mostra o caso ideal onde não há variação no período de amostragem, e não há latência de entrada e saída. Com base nesse gráfico, percebe-se a introdução de oscilações devido à variações no tempo de amostragem.

O controlador com compensação de atraso implementado foi baseado no controlador PI apresentado na seção anterior. A idéia básica é utilizar um dos *timers* do microprocessador ARM para efetuar *timestamps*. Subtraindo-se dois valores de tempo consecutivos no recebimento de mensagens CAN, utiliza-se esse período de amostragem para recalcular os coeficientes A e B da equação de diferenças apresentada na figura 3.19.

$$u(k) = u(k-1) + \underbrace{(K_P + \frac{K_i \cdot T_a}{2})}_{A} \cdot e(k) + \underbrace{(\frac{K_i \cdot T_a}{2} - K_P)}_{B} \cdot e(k-1)$$

Figura 3.19: Coeficientes da equação de diferenças do controlador compensado.

3.4.3 Controlador PI dividido em Subtasks

Para a implementação do controlador PI dividido em *Subtasks*, o algoritmo do controlador PI comum é utilizado. A *task* de controle é dividida em duas partes: A primeira possui maior prioridade, e basicamente calcula o sinal de controle, com base no erro do sistema. A segunda parte envia a mensagem CAN.

3.5 Dificuldades encontradas

Mesmo após tentativas de estabelecer a comunicação por SPI entre o microcontrolador TINI e o driver de acionamento, isso não foi possível devido à falta de confiabilidade na transmissão de mensagens principalmente por existirem interferências de ruídos. Além disso, a maior velocidade que a conexão possivelmente proporcionaria não era suficientemente rápida para que fosse efetuado um controle das plataformas mecânicas.

Planejou-se como solução de longo prazo a utilização do modo analógico do driver, e a introdução de dispositivos externos, conversores D/A e A/D. Porém, para isso seria necessário a aquisição de conversores capazes de se comunicar de forma serial, algo que impossibilitou a realização dessa alternativa.

Como solução de curto prazo, simulou-se o conjunto driver-kit-mecânico diretamente no microcontrolador TINI, para dar continuidade ao projeto.

3.6 Aquisição e Tratamento de Dados

Após a fase de implementação do *software*, foi feita a aquisição dos dados que trafegavam pela rede CAN para sua posterior análise. Para isso, a placa PCAN PCI (figura 3.20) da Peak System Technik foi utilizada em conjunto com o *software* PCAN Explorer 4 (figura 3.21). Este *software* permite o armazenamento de todas mensagens completas que trafegam pela rede, com seus respectivos *timestamps*. Desse modo, é possível elaborar um histograma

Figura 3.20: Placa Peak Pcan Pci utilizada para monitorar a rede CAN.

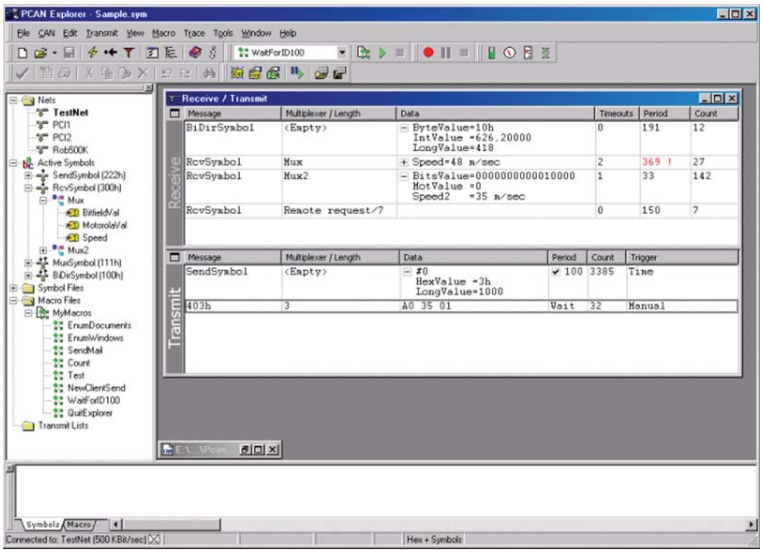


Figura 3.21: Software Pcan Explorer 4 utilizado para aquisição dos dados que trafegam na rede CAN.

com a distribuição dos tempos de amostragem. Além disso, pode-se utilizar os dados de uma mensagem para traçar gráficos do sinal de atuação e do sinal de saída de cada sistema planta-controlador inserido na rede CAN.

Para o tratamento dos dados obtidos pelos softwares de aquisição, foi desenvolvido um aplicativo em JAVA capaz de manipular o arquivo de saída do PCAN Explorer, de forma a tornar fácil o seu posterior estudo.

O arquivo de saída do PCAN Explorer destina uma linha para cada mensagem que trafega na rede. Cada mensagem é então caracterizada pelo seu ID de destino, pelos seus dados (a informação proprimente dita, na forma de quatro bytes em hexadecimal), e seu *timestamp*. A aplicação em JAVA gera arquivos independentes para cada conjunto de mensagens com o mesmo ID em comum. Os dados da mensagem na forma de quatro bytes são interpretados, e o aplicativo salva no arquivo gerado o valor do tipo ponto flutuante correspondente ao vetor de bytes.

Após a fase de organização dos dados aquisitação, uma rotina em Matlab foi implementada

para traçar os gráficos de sinal de saída e de atuação para cada conjunto planta-controlador, e para criar o histograma de variação do tempo de amostragem.

Para a comparação de diferentes algoritmos de controle, além da inspeção visual dos gráficos traçados, uma análise matemática também foi levada em consideração.

$$\overline{e^2} = \frac{\int_0^{T_{final}} (y_{ideal}(t) - y_{simulado}(t))^2 \, dt}{T_{final}}$$

O erro quadrático médio acima é calculado para cada algoritmo de controle. O vetor de dados aquisitados é comparado a um vetor do sinal de saída ideal, simulado no matlab.

3.7 Simulação de Sistema Distribuído

A partir da modelagem da planta real como um sistema de primera ordem, obteve-se o modelo discretizado pelo método de Tustin. Esse, por sua vez, pode ser representado pela equação de diferenças apresentada a seguir, a qual é utilizada no algoritmo de simulação do TINI.

$$u(k) = \frac{A \cdot u(k) + B \cdot u(k-1) - C \cdot y(k-1)}{T_a + 2 \cdot T}$$

Para se obter uma melhor aproximação com relação à planta real, foi feito uma compensação dos parâmetros da equação de forma semelhante ao que foi feito no controlador PI. Uma interrupção foi configurada especificamente para fins de temporização, sendo utilizada para medir o período de amostragem real, isto é, o intervalo de tempo entre duas mensagens consecutivas recebidas do controlador. Dessa forma, a planta permite simular os efeitos causados pela variação do período de atuação, mostrando-se sensível aos possíveis atrasos que surgem no sistema.

Para cada mensagem de controle recebido pela rede CAN, o microcontrolador TINI calcula a saída conforme equação acima e logo depois retorna a mensagem de *feedback* pela rede. O cálculo relativo à simulação da planta é feita pela função a seguir.

```
float simulatePlant(){

oldcount = count;
count = 0;
```

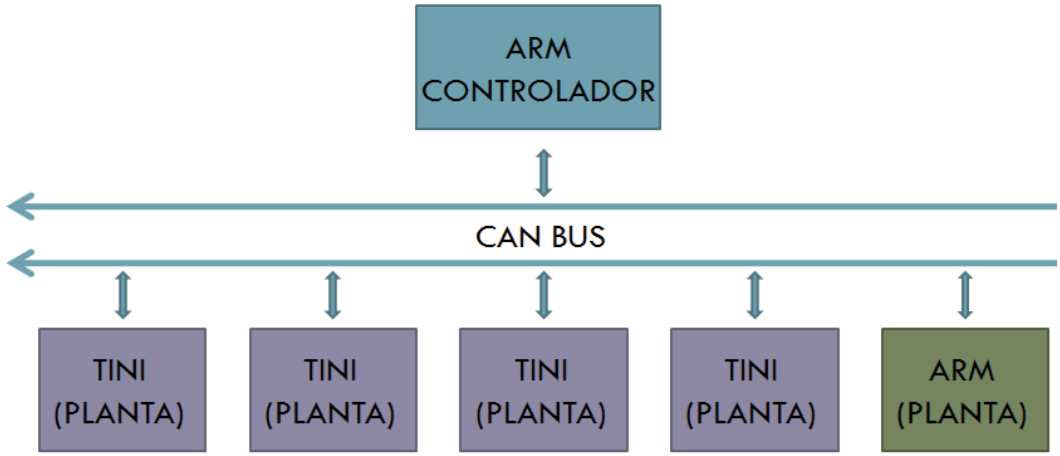


Figura 3.22: Esquema da simulação do sitema distribuído.

```
stime = (float)oldcount/19200.0; //cálculo do período para compensação
pA = PLANT_K*stime; //atualização dos parâmetros
pB = pA;
resetwdtimer(); //reseta o timer do Watchdog
pC = stime - 2*PLANT_T;
y = ( pA*u + pB*u_1 - pC*y_1)/(stime + 2*PLANT_T); //eq de diferenças
resetwdtimer(); //reseta o timer do Watchdog
y_1 = y; //armazena os valores atuais das variáveis
u_1 = u;

return y;
}
```

A simulação é feita em cinco microcontroladores, quatro TINIs e um ARM7. Cada um possui uma planta ligeiramente diferente, com controladores específicos. A configuração final pode ser vista na figura 3.22

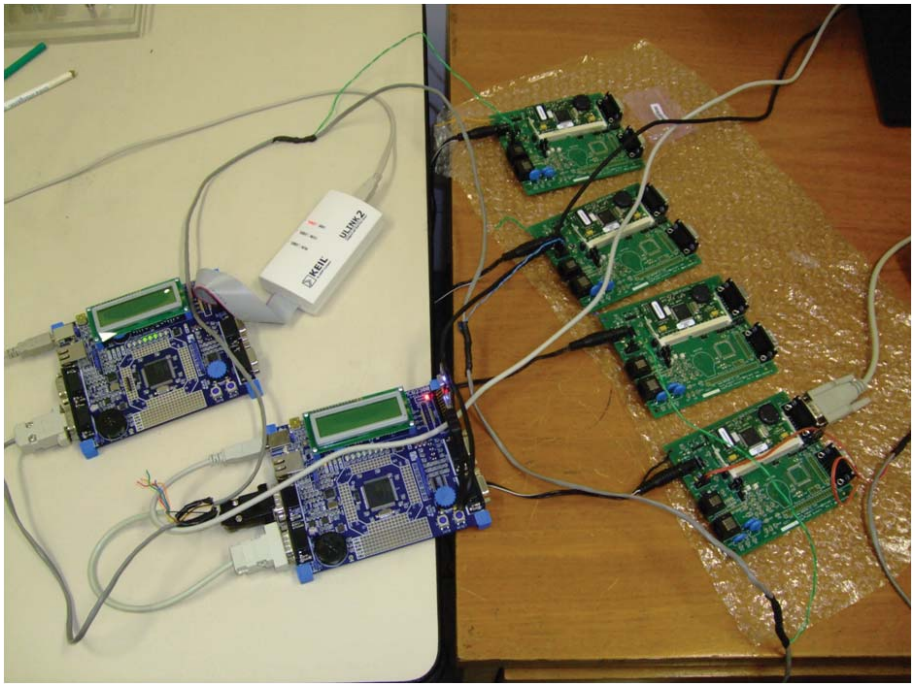


Figura 3.23: Simulação do Sistema Distribuído.

4 RESULTADOS

4.1 Plataforma Mecânica

O kit mecânico foi inteiramente projetado, pois um produto comercial semelhante possui custo muito alto. Foram feitos desenhos em programas de CAD, e o desenvolvimento foi documentado.

A construção do protótipo da plataforma mecânica foi feita utilizando-se materiais de fácil aquisição, como acrílico e alumínio. Para a fabricação das peças, foram utilizados os equipamentos da oficina do Departamento de Engenharia Mecatrônica da USP. O kit foi testado no Laboratório de Sistemas Embarcados.

Fotos das plataformas mecânicas podem ser vistas na figura 3.8.

4.2 Bridge de Rede

Foi implementada a rede CAN no microcontrolador TINI, bem como foi configurada a comunicação SPI, o que permitiria o uso do dispositivo como um *bridge* de rede. Porém, pelas dificuldades abordadas nas seções anteriores, a comunicação SPI não foi utilizada, e o driver de acionamento e o kit mecânico foram simulados no TINI.

4.3 Software de Controle

Foram desenvolvidos três tipos de algoritmos de controle para sistemas de primeira ordem. O primeiro consiste num controlador proporcional-integral comum. No segundo, os coeficientes da equação de diferença são recalculados conforme o último tempo de amostragem medido pelo sistema de tempo real no ARM7. O último, por sua vez, é dividido em *subtasks*, de forma a priorizar a periocididade na amostragem e cálculo do sinal de controle, enquanto que o envio do mesmo para a rede CAN possui uma relevância menor.

4.4 Simulação do Sistema Distribuído

Foi desenvolvido um *main loop* em linguagem C para o TINI e para o ARM o qual recebe a mensagem de controle, simula a planta real, e retorna a resposta da mesma pela rede. Foi feito uma compensação no período de amostragem para aprimorar a aproximação do modelo com relação à planta real.

4.5 Análise dos Controladores

O conjunto de aquisição de dados permitiu a análise dos dados que trafegaram pela rede CAN. Com base nessas informações, pôde-se avaliar o desempenho dos três algoritmos de controle.

A planta do caso estudado nas análises seguintes, possui ganho igual a 1.8 e constante de tempo de 1.8 segundos. O período de amostragem utilizado foi de 8ms. O controlador projetado no domínio contínuo possui parâmetro proporcional igual a 0.262884, e tempo de integração aproximadamente igual a 0.2s.

4.5.1 Controlador PI

A figura 4.1 apresenta o gráfico de sinal de saída e de atuação do controlador simples da planta simulada no microcontrolador ARM7, quando a rede de comunicação era compartilhada por cinco nós. A linha contínua represena os sinais ideais simulados em MATLAB, enquanto que a linha tracejada apresenta os dados experimentais.

Observa-se claramente que o congestionamento da rede CAN prejudica o desempenho do controlador, a ponto de tornar o sistema instável. Esse efeito ocorre devido principalmente à perda de mensagens, o que introduz atrasos consideráveis no sinal de controle.

O erro quadrático médio calculado chegou à ordem de 10^8 para esse caso da figura apresentada. Esse valor alto é explicado pela crescente amplitude de oscilação devido à instabilidade do sistema.

4.5.2 Controlador PI com Compensação

O controlador PI compensado apresentou um melhor desempenho com relação ao caso anterior. Os gráficos (figura 4.2) mostram que o sistema convergiu, porém com considerável

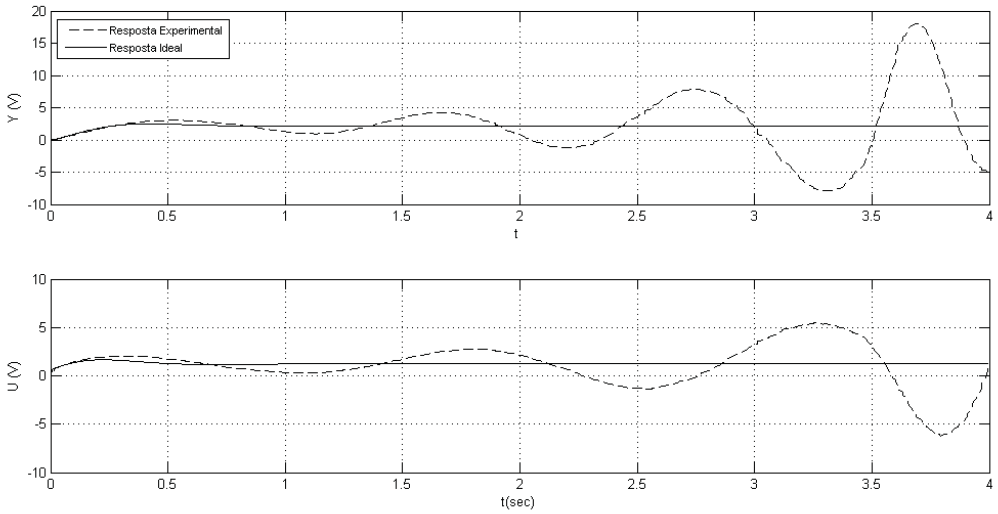


Figura 4.1: Controlador PI: Sinal de saída e de atuação para um conjunto planta-controlador

oscilação em relação à curva teórica.

O erro quadrático encontrado foi de 6.1592 para o exemplo da figura 4.2.

4.5.3 Controlador PI dividido em Subtasks

O controlador PI dividido em subtasks mostrou um desempenho melhor do que os dois casos acima. Com base nos gráficos traçados (figura 4.3), o desempenho é semelhante ao caso do controlador PI com compensação. Porém, o erro quadrático médio mostra que esse controlador obteve melhores resultados.

O valor do erro quadrático médio foi de 1.1524.

4.6 Análise da Variação do Tempo de Amostragem

Foram coletados os intervalos entre mensagens de um mesmo conjunto planta-controlador, sendo possível criar histogramas e comparar a distribuição dos tempos de amostragem para diferentes cargas sobre a rede CAN.

O conjunto planta-controlador do caso estudado possui ganho igual a 1.8 e constante de tempo de 1.8 segundos. O período de amostragem utilizado foi de 8ms.

Com base nos histogramas, pode-se constatar a enorme amplitude na variação do período de amostragem. Houve casos onde o atraso foi maior do que um período de amostragem, o

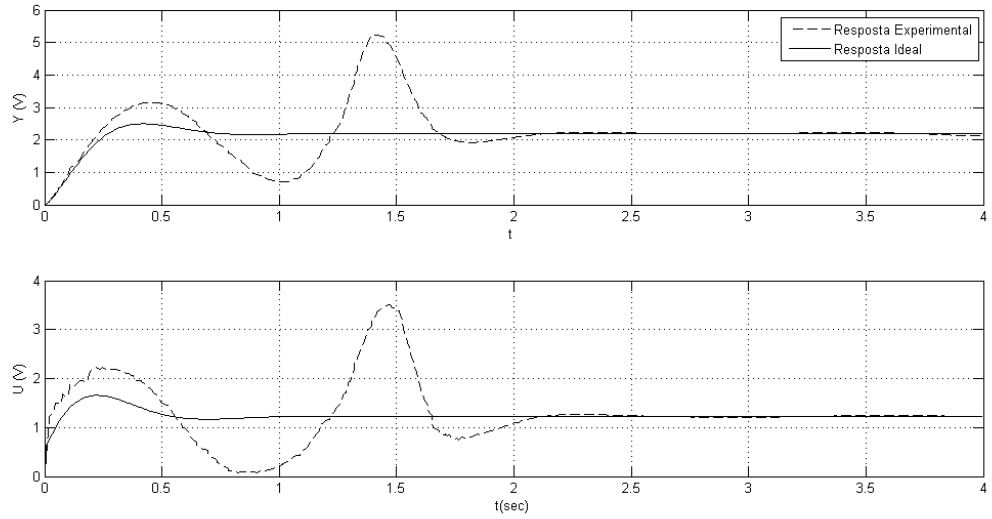


Figura 4.2: Controlador PI com compensação: Sinal de saída e de atuação para um conjunto planta-controlador

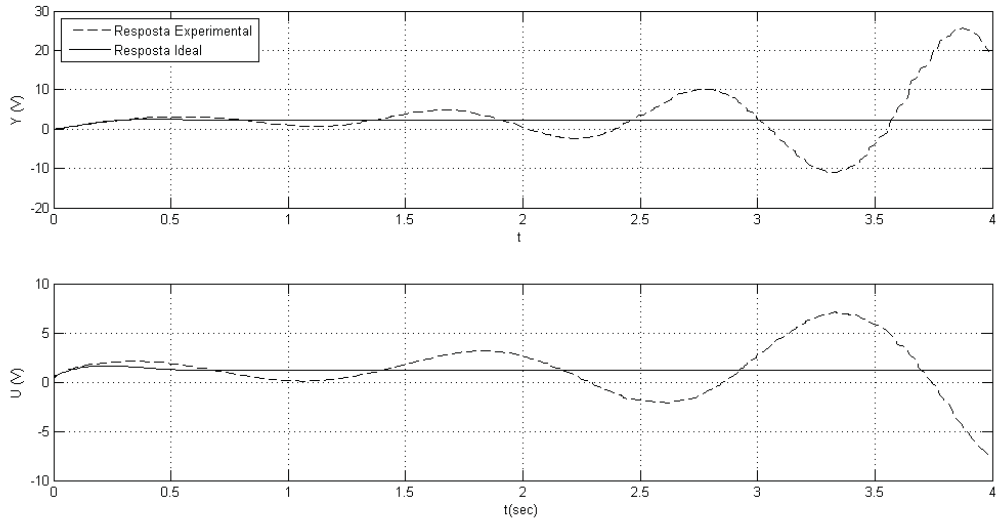


Figura 4.3: Controlador PI divido em Subtasks: Sinal de saída e de atuação para um conjunto planta-controlador

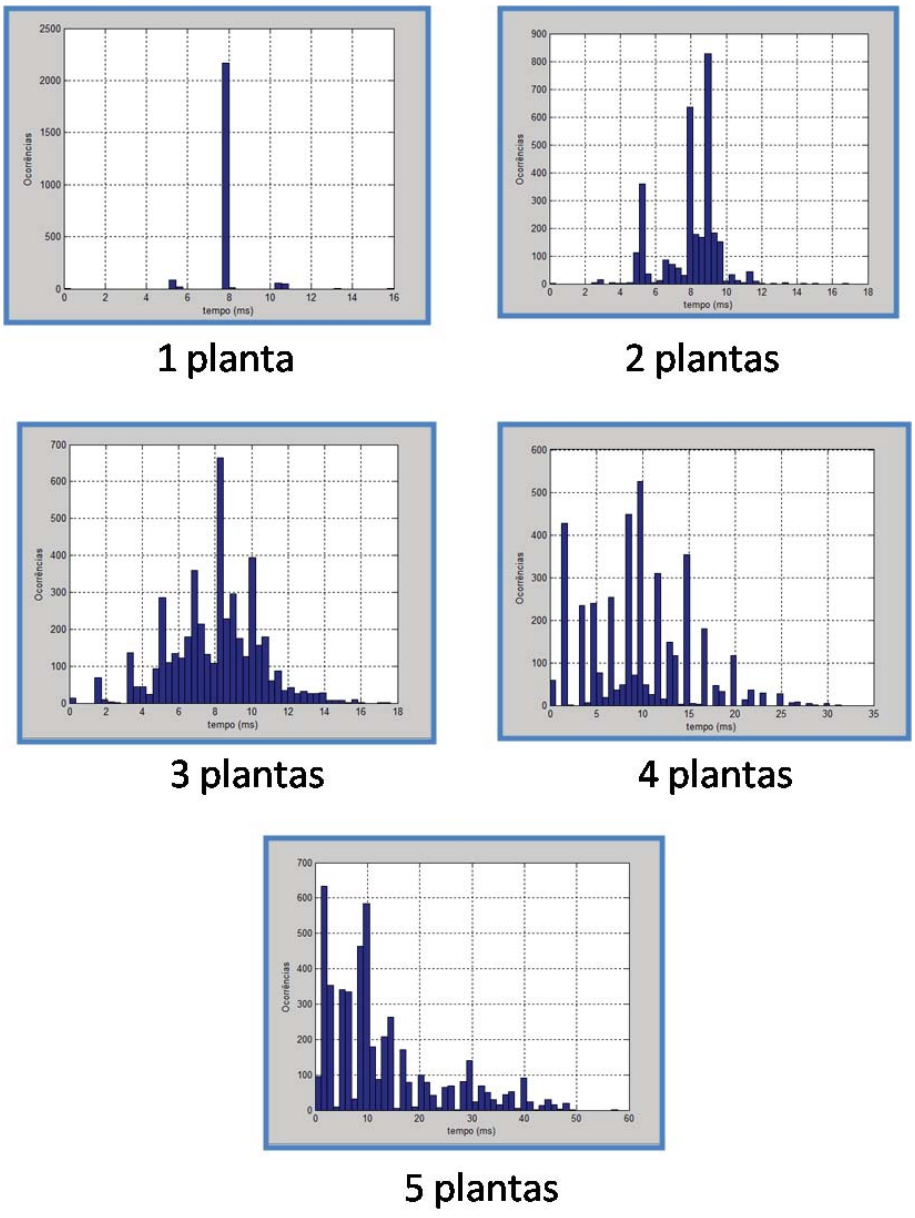


Figura 4.4: Histogramas para os Tempos de Amostragem

que é apresentado no gráfico como incidências com valores menores do que 8ms (Período de amostragem ideal).

Os resultados obtidos foram o projeto, construção, e teste da plataforma mecânica, desenvolvimento do software que implementa o protocolo CAN, e o desenvolvimento do software de controle.

O kit mecânico foi inteiramente projetado, pois um produto comercial semelhante possui custo muito alto. Foram feitos desenhos em programas de CAD, e o desenvolvimento foi documentado.

A construção do protótipo da plataforma mecânica foi feita utilizando-se materiais de fácil aquisição, como acrílico e alumínio. Para a fabricação das peças, foram utilizados os equipamentos da oficina do Departamento de Engenharia Mecatrônica da USP. O kit foi testado no Laboratório de Sistemas Embarcados.

O software que implementa o protocolo CAN foi desenvolvido, permitindo a integração dos microprocessadores ARM 7 e TINI na rede de comunicação.

O algoritmo de escalonamento implementado utilizado o sistema operacional de tempo real RTX.

5 CONCLUSÃO

O projeto de Sistemas de Controle Distribuídos de Tempo Real são, essencialmente, um problema de co-projeto. Foi possível por meio dos resultados constatar a existência de distúrbios de rede. Para isso, foi desenvolvida uma rede com números de nós variáveis representando modelos de plantas, de forma a congestionar gradativamente a mesma. Com isso, obteve-se uma relação entre a utilização da banda de rede e o desempenho do controlador, o qual é prejudicado principalmente pelos atrasos ou perdas de mensagens. Pode-se afirmar que o bom aproveitamento de um controlador depende do período de amostragem e da variação da latência de entrada e saída. Além disso, analisou-se três tipos de algoritmos de controle, cada um com suas respectivas peculiaridades. Baseado nos dados experimentais, recomenda-se que o controlador leve em conta os atrasos no cálculo de seus parâmetros, ajustando como consequência o sinal de atuação. Considerando isso, é possível ressaltar a importância do co-projeto, ou seja, da necessidade de se levar em conta aspectos de *hardware* e *software* no projeto de um controlador, para que este tenha um aproveitamento melhor dos recursos do sistema embarcado.

6 BIBLIOGRAFIA

[1] Wakamoto, E. *Sistemas de controle distribuídos em redes de comunicação*, São Paulo, 2009.

[2] Liu, C.L. and Layland, J.W., *Scheduling algorithms for multi-programming in a hard real-time environment*, Journal of the ACM, Vol 20, No. 1, pp. 40–61, 1973.

[3] Cervin, A. *Integrated control and real-time scheduling*, Ph.D. Thesis, Lund Institute of Technology, 2003.

[4] Pazul, K. *Controller Area Network (CAN) Basics*, Microchip Technology Inc., 1999.

[5] ARM, Ltd. *Getting Started: Building Applications with RL-ARM*,ARM Ltd e ARM Germany GmbH, 2009.

[6] Ogata, K. *Modern Control Engineering*, Prentice Hall, 2001.

[7] Andersson, M., Henriksson, D., Cervin, A. *Truetime 1.5 Reference Manual*, Lund Institute of Technology, 2007.

[8] Andersson, M., Henriksson, D., Cervin, A.,Årzén,K.E. *Truetime: Simulation of Networked Computer Control Systems*, Lund Institute of Technology, 2006.

7 APÊNDICE

7.1 Uma ferramenta de análise

7.1.1 A ferramenta TrueTime

O não determinismo temporal introduzido pela rede de comunicação acarreta uma perda de desempenho significativa no sistema de controle. Nesse contexto, a ferramenta TrueTime é utilizada nesse trabalho para analisar os efeitos causados pelos atrasos no controlador. Para uma descrição completa da ferramenta, consultar [8], [9]. TrueTime está disponível em "http://www.control.lth.se/user/dan/truetime".

A ferramenta em questão permite simular os aspectos temporais de sistemas de tempo real e de redes com ou sem fio dentro do ambiente Simulink, junto com a dinâmica em tempo contínuo da planta a ser controlada. A abordagem permite uma simulação com um riqueza de detalhes próxima ao sistema real.

O programa consiste em uma biblioteca de blocos com as funções de representar o *kernel*, uma rede convencional, uma rede *wireless*, e a bateria. O bloco *kernel* executa processos pré-definidos pelo usuário, e lida com interrupções causadas por algoritmos de controle, processos de entrada e saída, interface de rede, além de possuir conversores A/D e D/A. A política de escalonamento utilizada pelo *kernel* é definida pelo usuário. Para simular o sistema de controle em tempo real, os blocos da biblioteca TrueTime são conectados com os convencionais blocos do Simulink, sendo estes últimos utilizados para construir graficamente o algoritmo de controle.

Os blocos de rede distribuem as mensagens entre os nós de acordo com o modelo de rede escolhido. O TrueTime 1.5 suporta seis dos principais protocolos de controle acesso: (CSMA/CD (*Ethernet*), *switched Ethernet*, CSMA/CA (CAN), *token-ring*, FDMA, e TDMA). O bloco de rede sem fio proporciona a simulação dos padrões IEEE 802.11 WLAN e IEEE 802.15.4 ZigBee. Somente as interações mais relevantes para o comportamento do sistema em relação aos atrasos que são modelados. Algumas delas são: atrasos de pré e pós processamento, mecanismos de detecção de colisão e de prevenção de ocorrência de colisão, e a

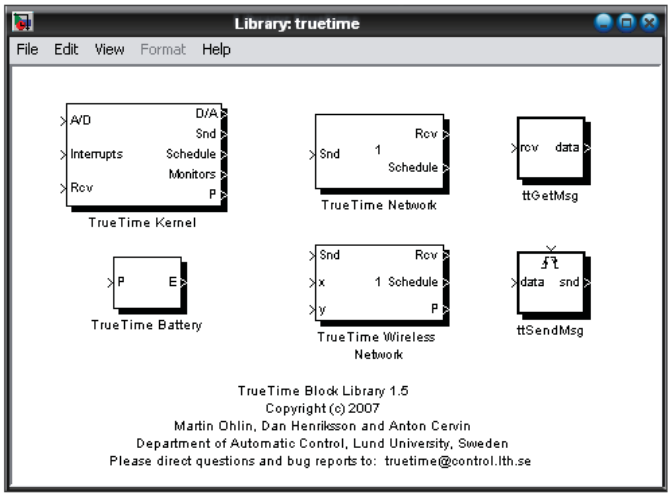


Figura 7.1: Biblioteca de blocos do TrueTime

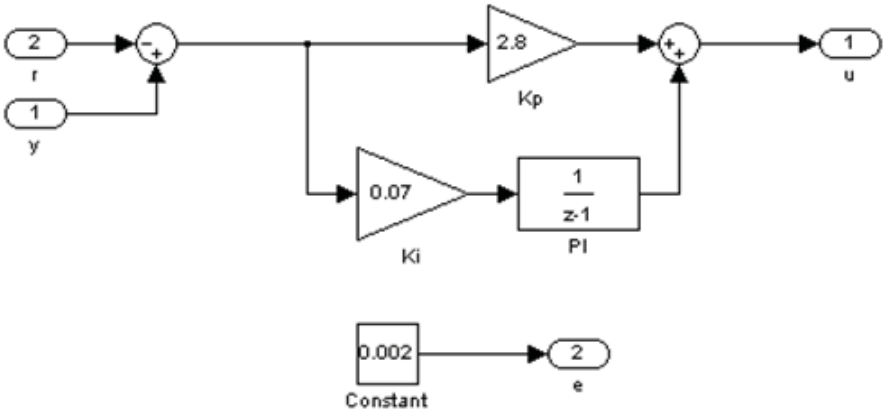


Figura 7.2: Controlador PI implementado em Simulink-Matlab

probabilidade de se perder pacotes.

Os códigos das funções para os processos ou comandos de inicialização podem ser feitos em C++ ou em linguagem MATLAB.

7.1.1.1 Exemplo

Para uma melhor apresentação da ferramenta TrueTime, foi realizada uma simulação do controle de dois motores de corrente contínua que possuem a função de transferência a seguir:

$$G(S) = \frac{1}{0.5s + 1} \tag{7.1}$$

O controlador utilizado é um PI (Proporcional Integral) com $K_p = 2.8$ e $K_i = 0.07$ conforme a figura 7.2. Monta-se o esquema mostrado na figura 7.3 utilizando um bloco *kernel*

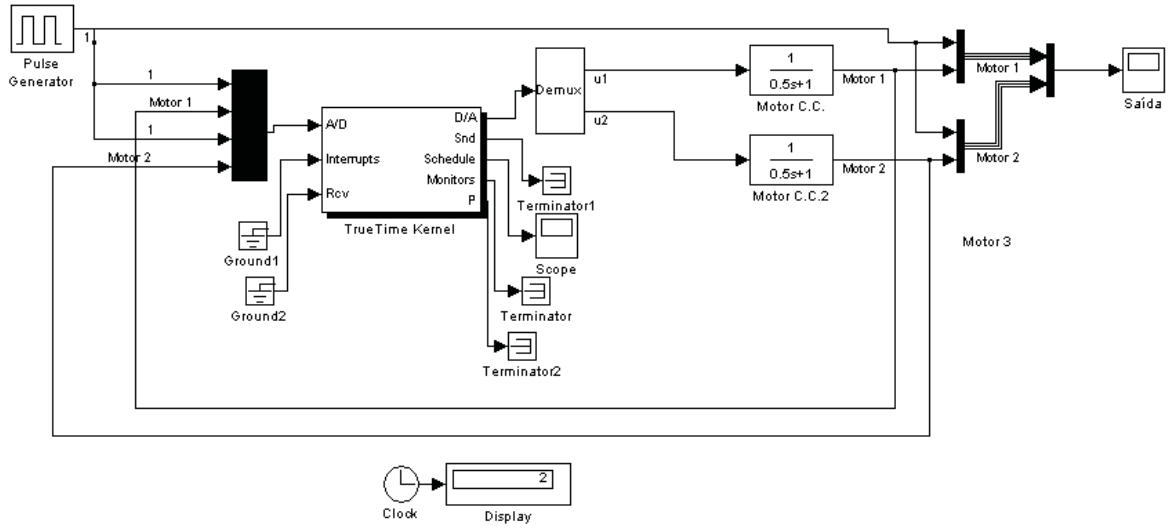


Figura 7.3: Esquema utilizando biblioteca TrueTime.

que é inicializado pelo arquivo *doismotoresTF.m*. Este arquivo define o número de entradas e saídas do bloco *kernel*, bem como a prioridade e período de cada processo. Além disso, permite escolher a estratégia de escalonamento e os controladores a serem utilizados. Destaca-se o fato de que foram utilizados controladores diferentes para cada uma das tasks, permitindo-se o uso de tempos de execução diferentes, sendo 3ms para a primeira, e 2ms para a segunda task.

```
% doismotoresTF.m
function dois_motoresTF

ttInitKernel(4, 2, 'prioRM');       %Inicia o kernel com 4 entradas e duas
%saídas. Pode-se escolher 'prioRM' e 'prioEDF'
periods = [0.006 0.005];          %Período de cada task
prio = [1 1];                   %Prioridade de cada task
names = {'motor1', 'motor2'};
Pc = {'Pcontroller1','Pcontroller2'}; %Controladores diferem no WCET das
    tasks
rChans = [1 3];                 %Seleciona entrada + do kernel
yChans = [2 4];                 %Seleciona entrada - do kernel
uChans = [1 2];                 %Seleciona saída do kernel

for i = 1:2
data.h = periods(i);
data.u = 0;
offset = 0;
```

```
data.Iold = 0;
data.Dold = 0;
data.yold = 0;
data.rChan = rChans(i);
data.yChan = yChans(i);
data.uChan = uChans(i);

ttCreatePeriodicTask(names{i}, offset, periods(i), prio(i), Pc{i}, data);
%cria task
end
```

```
% PI_controller1.m
function [exectime, data] = Pcontroller(segment, data)

switch segment,
case 1,
inp(1) = ttAnalogIn(data.rChan);        %Lê entrada positiva do kernel
inp(2) = ttAnalogIn(data.yChan);        %Lê a entrada negativa do kernel
outp = ttCallBlockSystem(2, inp, 'PI2'); %Calcula a ação de controle
%(WCET = 2ms)

data.u = outp(1);
exectime = outp(2);



case 2,
ttAnalogOut(data.uChan, data.u);        %Envia o sinal de controle
exectime = -1; % finished
end
```