Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Unidade Acadêmica de Engenharia Elétrica

Projeto de Conclusão de Curso

Desenvolvimento de Plataforma de Experimentos para Redes Industriais ModBus

Aluno: Ravi Agra Marques

Matrícula.: 20311214

Prof. Orientador: Edmar Candeia Gurjão

CAMPINA GRANDE

2008

Biblioteca Setorial do CDSA. Fevereiro de 2021.

Sumé - PB

Sumário

1. Lista de Figuras e Tabelas

2. Lista de Abreviaturas

ASCII – American Standard Code for Information Interchange

CCS – Custom Computer Services

CRC – Cyclic Redundancy Check

EIA – Electronic Industries Alliance

LCD – Liquid Crystal Display

LRC – Longitudinal Redundancy Check

OSI – Open Systems Interconnection

RFC – Request for Comments

RTU – Remote Terminal Unit

TTL – Transistor-Transistor Logic

UART – Universal Asynchronous Receiver/Transmitter

3. Resumo

Com o intuito de fornecer um ambiente onde estudantes do tema de Redes Industriais, especificamente do protocolo ModBus, pudessem colocar em prática os desenvolvimentos teóricos vistos, propôs-se a criação de um conjunto de dispositivos capazes de implementar o referido protocolo. Tais dispositivos, tendo incorporadas funcionalidades de aquisição de dados, permitem o desenvolvimento de experimentos e aulas práticas onde o protocolo ModBus é utilizado como ferramenta de laboratório, dando um enfoque utilitário à esse tema.

Com o fornecimento dos aplicativos de suporte e rotinas de tratamento de dados, libera-se o foco dos estudantes para a análise do protocolo de comunicação, bem como sua realização prática.

Essa estrutura pode ainda ser incrementada e modificada pelos próprios estudantes, dando margem ao desenvolvimento de projetos futuros e fornecendo suporte à pesquisa em diversos âmbitos.

4. Introdução

O presente trabalho consiste na elaboração de uma plataforma de desenvolvimento para suporte ao ensino do protocolo de redes industriais ModBus.

Tal plataforma é constituída por hardware e software, sendo:

- Hardware: placas de experimentos com diversos circuitos de entrada e saída, ligados a um micro-controlador (PIC16F877A), que irá emular um nó de uma rede ModBus. Há também na placa a interface serial para a interligação dos diversos nós.
- Software: Será fornecido, junto com cada placa, o código para o micro-controlador adotado que permita aos alunos utilizar de imediato as funcionalidades da placa, acessando os dados advindos das diversas interfaces sem necessitar desenvolver funções para isso. Será fornecido também o aplicativo que irá emular o mestre da rede ModBus, na forma de um programa para ambiente Windows, a ser executado em um PC conectado à rede formada pelas placas de desenvolvimento através de uma porta serial.

Esse kit de desenvolvimento será usado como suporte à disciplina de Redes de Computadores, podendo também ser aproveitado por outras disciplinas, como Sistemas de Automação Industrial, Sistemas de Aquisição de Dados e Interface, dentre outras, que abordem o tema de redes industriais.

Para que o enfoque no uso dessa plataforma se dê justamente sobre os aspectos da rede, as funções de aquisição de dados e acionamento já são implementadas no código fornecido, devendo os alunos configurar e analisar os pacotes de comunicação trocados entre os nós da rede, identificando os campos característicos do protocolo ModBus.

5. Fundamentos Teóricos

O ModBus é um protocolo de mensagens desenvolvido originalmente pela Modicon (atualmente parte da Schneider Electric) em 1979 para servir de guia para a comunicação entre seus dispositivos (1).

Desde então se tornou um padrão da indústria e é atualmente o modo mais comum de interligação entre dispositivos industriais. Tal sucesso se deve principalmente à simplicidade do protocolo, além de ser um padrão de código aberto, permitindo que os desenvolvedores personalizem sua aplicação. O padrão ModBus também permite a comunicação entre dispositivos de diferentes fabricantes de forma direta, facilitando a integração e manutenção do sistema (2).

Atualmente existe uma organização de usuários e desenvolvedores, a ModBus-IDA, que atua, sem fins lucrativos, no intuito de desenvolver melhorias para o sistema, promover a adoção do protocolo em novos mercados e fornecer uma infra-estrutura para a troca de informações e experiências entre usuários (3).

O ModBus é um protocolo do tipo Mestre-Escravo, com comunicação half-duplex, onde um dispositivo (o Mestre) solicita pelo enlace a realização de alguma função por um determinado dispositivo (o Escravo), este pode então responder ao Mestre, passando dados para ele. Numa rede ModBus padrão só há um Mestre, mas podem existir até 247 escravos, cada um com um endereço único (entre 1 e 247) (4). O *broadcast* é possível, mas mensagens enviadas por broadcast não serão respondidas, ou seja, o mestre pode usá-las para acionar funções de todos os escravos, mas não para ler dados destes.

Tomando como referência o modelo OSI (5), podemos enquadrar as definições do padrão ModBus em duas camadas, a de enlace, onde o protocolo define o enquadramento dos dados e a forma da mensagem, e a de aplicação, onde dispositivos fazem uso das funções definidas pelo padrão. Há diferentes formas de implementação do ModBus, como podemos ver na Figura 1.

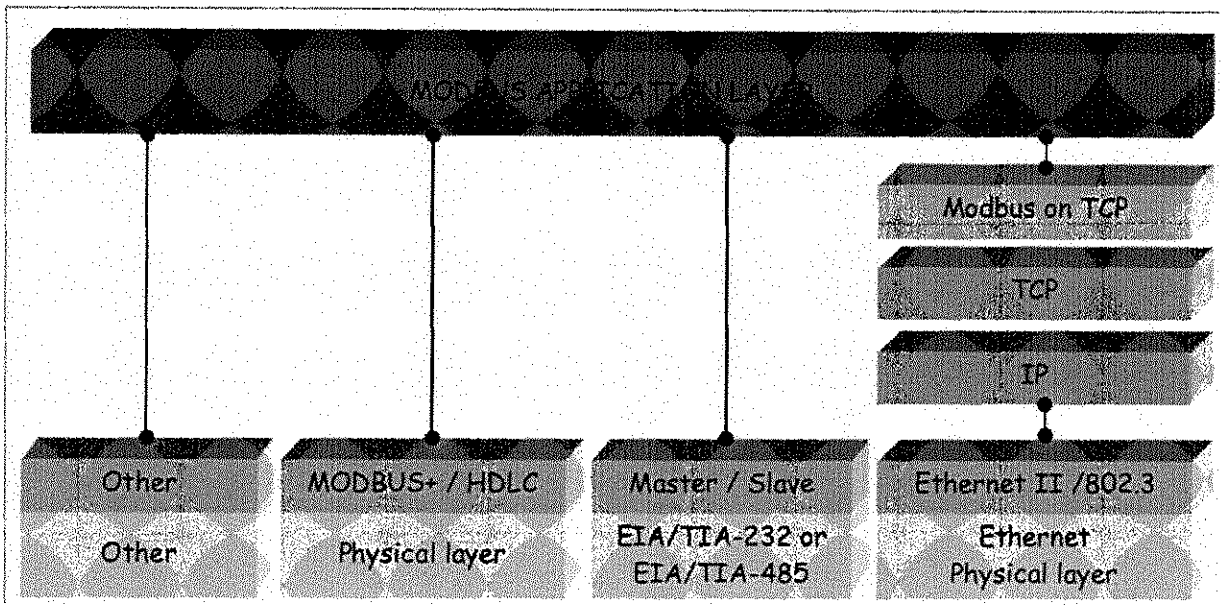

Figura 1 - Camadas de comunicação do ModBus

As diferentes formas de implementação do ModBus se baseiam em distintas formas da camada física. A forma mais comum, e o alvo do presente trabalho é a que usa os protocolos seriais EIA 232 e 485 como camada física. O modo ModBus+ é proprietário da Modicon, esse modo utiliza na camada física cabos em par trançado, sendo os dados codificados por transições de nível de tensão, operando em taxa típica de 1 Mbit/s. A implementação sobre Ethernet traz algumas vantagens, como: a restrição sobre o número de escravos é relaxada, podendo-se alocar mais de 247 dispositivos numa mesma rede, o protocolo Mestre/Escravo pode ser abandonado, permitindo que os dispositivos terminais "reportem por exceção", ou seja, alertem o nó central apenas quando houver mudanças em seus dados, otimizando assim o uso de banda.

Dentro do protocolo serial, existem duas formas de encapsulamento da mensagem: a RTU (Remote Terminal Unit) e o ASCII.

O modo RTU tem as seguintes características (6):

RTU	Cada byte de mensagem é enviado como um byte de dados. A mensagem deve ser transmitida de maneira contínua, já que pausas maiores que 1,5 caractere provocam truncamento da mesma.	
	11 bits por byte	1 start bit 8 bits de dados LSb enviado primeiro { 1 bit de paridade (par/ímpar) + 1 stop bit { 0 bit de paridade + 2 stop bits
	Campo de Checagem de Erros	CRC

E a mensagem tem o seguinte formato:

Start	Endereço	Função	Dados	CRC	END
Silêncio 3..5 chars	← 8 bits→	← 8 bits→	←N x 8 bits→	←16 bits→	Silêncio 3..5 chars

O modo RTU é o mais comum em ambiente industrial, mas por uma questão de facilidade de adaptação da UART do microcontrolador adotado, o modo ASCII é mais abordável, dessa forma, adotamos esse modo na plataforma de desenvolvimento. O modo ASCII possui as seguintes características:

ASCII	Cada byte de mensagem é enviado como dois caracteres ASCII. Durante a transmissão, intervalos de até um segundo entre caracteres são permitidos, sem que a mensagem seja truncada. Algumas implementações fazem uso de tais intervalos de silêncio como delimitadores de fim de mensagem, em substituição à seqüência `cr+lf`.	
	10 bits por byte	1 start bit 7 bits de dados LSb enviado primeiro { 1 bit de paridade (par/ímpar) + 1 stop bit { 0 bit de paridade + 2 stop bits
	Campo de Checagem de Erros	Longitudinal Redundancy Check

Com as mensagens obedecendo ao seguinte formato:

Start	Endereço	Função	Dados	LRC	END
: (0x3A)	2 Chars	2 Chars	N Chars	2 Chars	CRLF

Um processo normal de troca de mensagens entre o mestre e um escravo segue o protocolo mostrado na Figura 2.

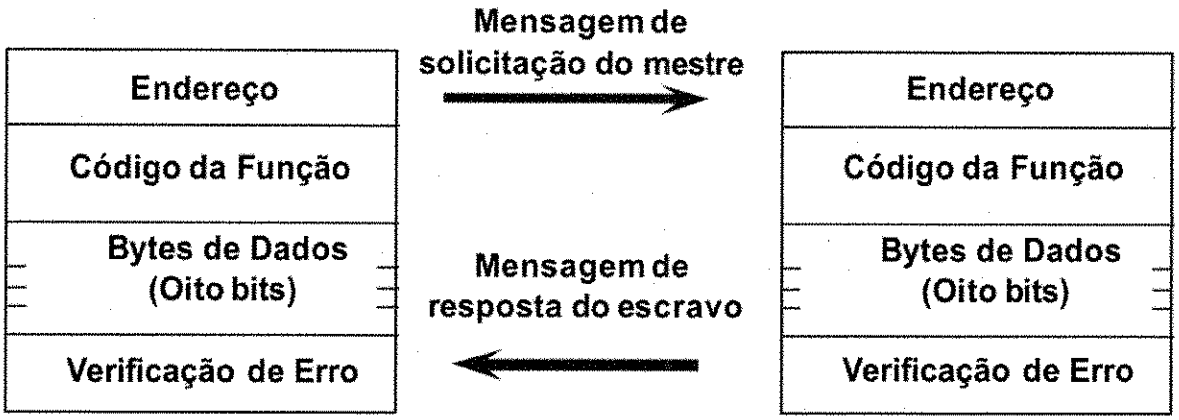

Figura 2 - Protocolo de pedido-resposta

A resposta do escravo repete os campos de endereço e função do pedido (caso a operação seja realizada com sucesso), acrescenta os seus dados (quando pertinente), recalcula o campo de verificação de erro e envia.

No padrão ModBus, as funções são pré-determinadas, tendo seus identificadores registrados na ModBus-IDA, de acordo com a seguinte classificação (7):

- Funções públicas: são funções bem definidas, com um identificador único, garantido pela ModBus-IDA, tendo sido validado pela comunidade de usuários e documentado publicamente. São as faixas de códigos de 1 a 65, de 72 a 100 e de 110 a 127. Alguns desses códigos são reservados para uso futuro.
- Funções definidas por usuários: são as funções nas faixas de 65 a 72 e de 100 a 110, esses identificadores são liberados para que usuários definam neles funções criadas especificamente para suas aplicações, não precisando ser documentadas ou oferecer compatibilidade, como tal, não há garantias de unicidade. Caso um fabricante deseje tornar uma função sua pública, deve iniciar um RFC com a ModBus-IDA, para receber um código público caso seja aceito pela comunidade.

Não há uma exigência para que um dispositivo seja capaz de executar todas as funções públicas, mas, para aquelas das quais ele dispõe, estas devem ser corretamente identificadas, mantendo o padrão. Por exemplo, as funções mais comuns, a escrita de saídas digitais e a leitura das entradas digitais, recebem os códigos 5 e 2, respectivamente, assim, qualquer dispositivo que implementar essas funções deve endereçá-las por esses códigos, a

fim de permitir sua compatibilização com outros dispositivos que porventura lhe solicitem dados em sua rede.

Existem ainda os códigos acima de 127, que são reservados para identificar falhas de comunicação/execução, permitindo que o sistema realize diagnósticos de erros. Esses códigos são enviados pelo escravo em substituição ao código da função requisitada, quando há algum erro.

O campo de dados é utilizado para passar informações adicionais à execução da função, ou enviar os dados de resposta. O comprimento desse campo é variável, a depender da necessidade da função em questão, podendo até mesmo ser vazio. No modo ASCII, os dados são formatados como caracteres, enquanto no modo RTU como bytes (8 bits). As variáveis representadas no padrão normalmente são ou variáveis discretas, de 1 bit, representando o *status* de uma entrada/saída, ou registradores de 16 bits, com uso diverso.

No campo de verificação de erro, existem dois métodos de cálculo, o CRC, usado pelo RTU e o LRC, usado pelo ASCII. No presente trabalho, foi usado o modo ASCII, logo, implementamos as rotinas de checagem de erro baseadas no LRC. Uma descrição do método está presente na seção de Desenvolvimento da Plataforma.

6. Desenvolvimento da Plataforma

O alvo principal da plataforma é permitir um primeiro contato com as funções e com o protocolo de comunicação ModBus. Para examinar o protocolo, é mostrado, no aplicativo do mestre, os campos que compõem a mensagem a ser enviada, devendo o usuário preencher esses campos de forma a requisitar a função desejada, incluindo os dados necessários.

Uma vez que o tópico do simples estabelecimento da comunicação seja dominado, o uso das funções em si pode ser explorado. Para tal, não há necessidade de escrever nenhum código, bastando acessar as funções pré-programadas nos dispositivos escravos.

As funções dos escravos são acessadas como em qualquer rede ModBus, devendo-se passar o endereço do dispositivo a atender o pedido, o código válido da função dentre aqueles implementados, e, se necessário, dados adicionais. O campo de verificação de erro é gerado automaticamente pelo programa.

Para permitir a verificação real das funções, foram colocados na placa dos escravos, circuitos de aquisição, condicionamento e acionamento, bem como elementos de interface com o usuário, de forma que este possa ter uma experiência interativa com o sistema, observando os resultados de suas ações. Descreveremos a seguir esses circuitos, bem como a interligação destes ao elemento central dos dispositivos, o micro-controlador.

6.1.Hardware

6.1.1. Entradas Digitais

Cada placa possui 8 entradas digitais, sendo 4 delas ligadas diretamente a um *dipswitch*, para ser usado como elemento de testes e simulação pelo usuário. As 4 restantes possuem um circuito de condicionamento que permite a utilização de sinais em tensão, tanto em 12Vdc como em até 240Vac para servir como entradas.

A Figura 3 mostra esse circuito.

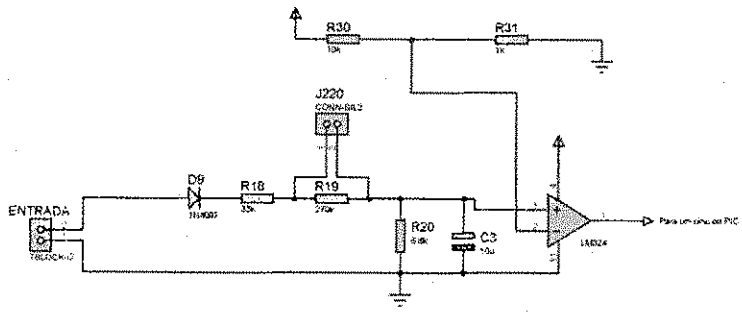

Figura 3 - Circuito de condicionamento de uma entrada digital

Como podemos ver, o Amplificador Operacional age como um comparador, em sua entrada inversora é colocada uma pequena tensão, 0.09 Vcc, ou cerca de 0.5V. Qualquer tensão presente na entrada não-inversora maior que essa irá levar a saída do operacional a Vcc, permitindo que o PIC reconheça aquela entrada como ativa. Uma tensão menor que essa deixará a saída em 0V, sendo interpretada como entrada inativa.

Para obter essa tensão, temos duas possibilidades, caso nossa entrada seja utilizada para ler um sinal DC, normalmente de 12V, devemos colocar um *cap* no *jumper* indicado por J220, o que irá efetivamente remover o resistor de 270kΩ do circuito. Com isso o circuito se resume a um divisor resistivo simples, sendo a relação entre o sinal oferecido à entrada do AmpOp e a entrada de 0.171, assim, uma tensão presente na entrada de 12V irá oferecer ao AmpOp 2.05V, o suficiente para acioná-lo. Tal relação também permite que se use outras tensões na entrada, como 9V (que estabelece 1.54V no AmpOp) ou mesmo 5V (ficando 0.85V). Tensões acima de 12V também são possíveis, podendo-se até mesmo usar o nível, também comum em ambiente industrial, de 24V, o que impõe na entrada do AmpOp uma tensão de 4.1V.

Caso pretenda-se utilizar tensão alternada para acionar as entradas digitais da placa, deve-se retirar o *cap* do *jumper* J220, o que insere o resistor de 270kΩ no circuito. Nesse caso devemos analisar o papel do diodo como retificador de meia-onda, e mencionar que o capacitor colocado ao final do circuito tem o papel de minimizar o efeito do *ripple*, efetivamente oferecendo um nível praticamente constante à entrada do amplificador. Nesse modo de operação, podemos utilizar a tensão da rede, de 220 Vac, para acionar a entrada, ficando no caso uma tensão de cerca de 2.2V na entrada do amplificador (valor obtido em simulação). Outras tensões podem ser utilizadas, como 110 Vac.

No caso das entradas digitais ligadas ao *dipswitch*, a análise é direta, as chaves do *dipswitch* simplesmente ligam ou não o terra às entradas do PIC, como mostrado na Figura 4, quando estas se encontram flutuando, o PIC interpreta o estado como ligado, e, quando aterradas, como desligado. Essas entradas podem ser utilizadas como testes, antes de se passar para usar as entradas reais, ligadas a fontes externas de tensão.

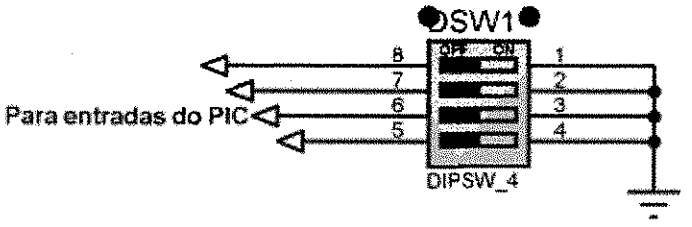

Figura 4 - Entradas com dipswitch

6.1.2. Saídas Digitais

Cada placa possui 8 saídas digitais, sendo 4 delas ligadas a LEDs, para serem usadas em testes e simulações, e as 4 restantes sendo saídas a relay, cada uma com o circuito de acionamento. Essas saídas podem ser usadas para acionar cargas reais, como lâmpadas, pequenos motores, etc. Os relays empregados suportam uma carga de até 1200W, podendo ser usados em tensão alternada. A corrente máxima recomendada é de 10A, então deve-se tomar cuidado ao ligar cargas a elas, determinando-se antes se esta corrente não será excedida pela carga em funcionamento.

A Figura 5 mostra o circuito de acionamento dos relays.

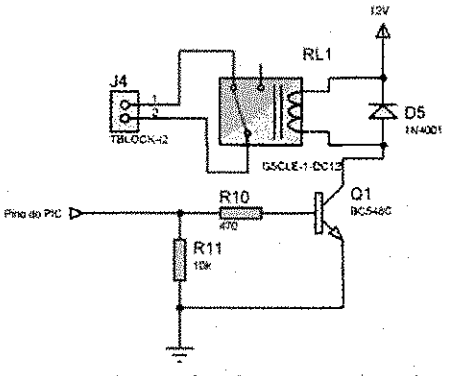

Figura 5 - Circuito de acionamento dos relays

Esse circuito utiliza um transistor para drenar a corrente necessária para excitar a bobina do relay, já que uma saída direta do micro-controlador não tem capacidade de

drenar tal corrente. Utilizamos também um diodo em anti-paralelo com a bobina para oferecer um caminho à corrente na abertura do relay.

As saídas a LED são ligadas diretamente a saídas do PIC, com o uso de resistores para limitar a corrente. A Figura 6 mostra essa ligação.

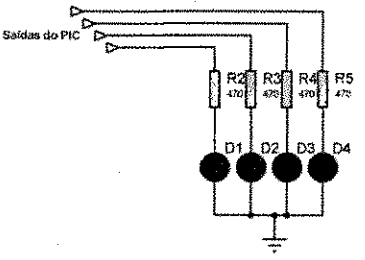

Figura 6 - Saídas digitais com LEDs

6.1.3. Entradas Analógicas

Cada placa possui 5 entradas analógicas, sendo 4 delas disponíveis para ligações externas, e 1 internamente dedicada à leitura de temperatura, através de um sensor colocado na própria placa.

Para as entradas disponibilizadas externamente, nenhum circuito foi adicionado, apenas um borne de ligação foi disponibilizado, permitindo que o usuário conecte seu sinal de tensão às portas de entrada analógica do PIC. Deve-se somente atentar para não utilizar sinais que possam exceder 5Vdc, a fim de evitar danos ao micro-controlador.

Para a entrada de leitura de temperatura, foi utilizado um sensor LM35, em sua configuração básica, como mostrado na Figura 7. Com a saída do sensor ligada a uma entrada analógica do PIC. A tensão de alimentação do sensor utilizada foi 5Vdc.

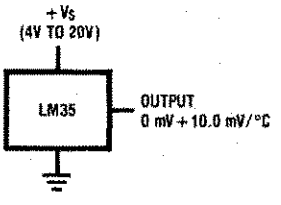

Figura 7 - Ligação do sensor de temperatura LM35

6.1.4. Circuito de comunicação

Para o estabelecimento da rede, foi disposto nas placas dos dispositivos escravos um circuito capaz de converter os sinais TTL da UART do micro-controlador para o padrão EIA-485, formando uma rede half-duplex.

Também é possível utilizar esse mesmo circuito para operar em modo 232, usando um conversor para compatibilizar os sinais advindos da porta serial do PC (dispositivo mestre).

O circuito utilizado nos escravos está mostrado na Figura 8.

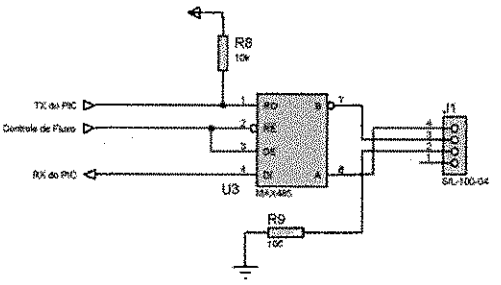

Figura 8 - Circuito de comunicação dos escravos

6.2.Software

O desenvolvimento do software para a plataforma de desenvolvimento tem duas etapas: a elaboração do aplicativo mestre, a ser executado em um PC, em ambiente Windows, e a elaboração do programa embarcado nos dispositivos escravos, para a arquitetura dos micro-controladores PIC16F877A.

Descreveremos agora as funções e rotinas elaboradas para cada um desses códigos, detalhando seu funcionamento e inserindo-as no contexto da aplicação.

6.2.1. Programa dos Dispositivos Escravos

O programa utilizado pelos dispositivos escravos monitora constantemente a UART do micro-controlador, aguardando dados seriais serem recebidos. Ao detectar uma recepção, ele captura os dados na forma de caracteres ASCII, compondo assim uma

mensagem no formato previsto pelo padrão ModBus. Na presente implementação, as mensagens tem um tamanho padrão de 12 caracteres, suficientes para todas as funções implementadas.

Tal mensagem, como já ilustrado, tem o seguinte formato:

Start	Endereço	Função	Dados	LRC	END
: (0x3A)	2 Chars	2 Chars	N Chars	2 Chars	CRLF

Ao encerrar a recepção (por contagem dos 12 caracteres esperados) o programa inicia a análise do pedido. O primeiro campo é sempre fixo, logo o programa analisa os dois caracteres seguintes, que formam o campo de endereço do dispositivo escravo solicitado.

Todos os dispositivos ligados à rede recebem os mesmos dados, mas cada um deles possui um endereço específico, inserido na programação de seu micro-controlador. Este checa o campo de endereço recebido com o seu endereço próprio. Somente caso haja a igualdade, é que este dará continuidade à análise, caso contrário simplesmente retorna a apenas monitorar a porta serial.

Estabelecendo-se que o endereço solicitado é o seu, aquele escravo irá continuar a avaliação do pedido, com a checagem do código de detecção de erros incluído na mensagem. Para isso, é usado o método LRC (Longitudinal Redundancy Check), descrito mais à frente. Caso a correta recepção dos dados seja verificada, o programa lê o campo de identificador da função, caso seja um código reconhecido como uma das funções que aquele dispositivo implementa, ele irá seguir para a rotina que executa aquela função. Caso contrário responderá ao mestre alertando-o que a função requisitada é desconhecida.

Durante a execução de cada função específica, é montada a mensagem de resposta ao mestre, usando como base uma repetição do pedido original, já que o padrão ModBus prevê que o código da função e o endereço do dispositivo na resposta devem ser os mesmos do pedido original. A função selecionada insere nessa mensagem (no campo de dados) os seus resultados.

Ao final, o programa calcula o campo de checagem de erros de sua resposta. Com isso ele completa a mensagem, podendo então enviá-la pela porta serial para a rede, de forma que esta possa então chegar ao seu destino, o dispositivo mestre.

6.2.1.1. Leitura das Entradas Digitais

Corresponde à função identificada pelo código 02 no padrão ModBus. As entradas são codificadas como dígitos binários, com ON = 1 e OFF = 0. Elas são então agrupadas em bytes (8 entradas). O Mestre deve incluir nos campos de dados, após o identificador da função, o endereço de início das entradas que deseja ler e a quantidade dessas entradas.

O dispositivo escravo deve então compor a resposta, incluindo nos campos de dados quantos bytes forem necessários para informar as entradas pedidas. No kit de desenvolvimento, no entanto, as placas dos escravos possuem apenas 8 entradas, ficando desnecessário tal procedimento. Assim sendo, os escravos sempre respondem com apenas 1 byte aos pedidos de leitura de entradas digitais, esse byte contendo o valor lógico das suas 8 entradas. Dessa forma os campos de dados do mestre, nessa função, são ignorados, e qualquer dispositivo que solicite as entradas da placa de desenvolvimento receberá sempre os 8 bits.

Para o código utilizado na implementação dessa função, referimos às linhas 84-94 do Apêndice A.

6.2.1.2. Leitura das Saídas Digitais

O modelo da função para leitura das saídas digitais é idêntico ao da de leitura de entradas digitais, tendo sua implementação seguido os mesmos critérios. A função corresponde ao identificador 01 do padrão ModBus.

Por uma característica das portas de entrada e saída do micro-controlador, a execução de uma operação de leitura das saídas automaticamente transforma esses pinos em entradas, logo, os valores presentes serão lidos, mas as saídas conectadas a eles serão desativadas. Logo, se é de interesse manter os valores anteriores, deve-se re-escrever esses valores imediatamente após a operação de leitura. Tal procedimento pode inclusive ser acoplado à própria função.

Para o código utilizado na implementação dessa função, referimos às linhas 73-83 do Apêndice A.

6.2.1.3. Escrita de Saída Digital

Equivale ao código identificador 05. É usada quando o mestre deseja alterar o estado de uma saída digital em determinado escravo. O mestre passa em seus dados o endereço da saída desejada (endereço referente à posição daquela saída no dispositivo) e o estado a que essa deve ser forçada.

Logo, o escravo, ao identificar o pedido de uma função 05, deve ler os próximos dois campos, sendo o imediato o endereço da saída a ser alterada, e o posterior o estado lógico a que essa deve ser levada.

Para o código utilizado na implementação dessa função, referimos às linhas 95-149 do Apêndice A.

6.2.1.4. Leitura de Entrada Analógica

A placa de desenvolvimento possui o recurso de leitura de sinais analógicos, através de 4 entradas do micro-controlador disponibilizadas diretamente. Para a utilização desses recursos, foi desenvolvida uma função definida pelo usuário, já que o padrão ModBus não prevê entradas analógicas em suas funções públicas.

Assim sendo, adotamos o código 65 para identificar essa função. Para sua utilização o mestre deve enviar no primeiro campo de dados o número da entrada analógica que deseja ler. O escravo então seleciona aquele canal de conversão analógico-digital e realiza a leitura do sinal presente no pino correspondente.

O escravo então já converte o valor binário obtido (Valor de 10 bits, 0 a 1023) em dois caracteres ASCII (para acomodar os dados no modo adotado de comunicação serial) e os insere nos campos de dados da resposta enviada para o mestre. Adicionalmente, o valor convertido para uma representação em tensão do sinal presente (0 a 5V) é exibida no LCD ligado à placa de desenvolvimento.

Para o código utilizado na implementação dessa função, referimos às linhas 150-157 do Apêndice A.

6.2.1.5. Leitura de Temperatura

Conectado a uma quinta entrada analógica, cada placa possui um sensor de temperatura, um LM35, ajustado em sua configuração padrão para associar temperaturas entre 0 e 100°C com valores de tensão entre 0 e 1V.

Para o acesso a essa informação, foi disponibilizada outra função ModBus definida pelo usuário, identificada pelo código 66. Para utilizar essa função o mestre precisa apenas solicitá-la, não havendo campos de dados necessários.

O escravo irá então ler a entrada analógica associada ao sensor de temperatura, e enviar o dado obtido para o mestre, decompondo os 10 bits originais em dois caracteres ASCII. Adicionalmente o valor da temperatura é calculado a partir do valor lido e exibido no LCD.

Para o código utilizado na implementação dessa função, referimos às linhas 165-170 do Apêndice A.

6.2.1.6. Método para a Checagem de Erros na comunicação

No modo ASCII, o método usado para checagem de erros é o LRC (Longitudinal Redundancy Check). Este método permite que um receptor detecte se uma mensagem transmitida serialmente contém algum erro.

Para gerar o campo de checagem de erros presente no final da mensagem, por este método, o dispositivo soma todos os campos variáveis da mensagem, tomando apenas o byte menos significativo do resultado. Calcula-se então o complemento de 2 desse byte. Esse é então o byte utilizado no campo de checagem de erro.

No outro ponto da comunicação, está a necessidade de checar a mensagem recebida, como nesta está presente o valor do LRC calculado pelo transmissor, se o receptor recalcular esse campo com base no restante da mensagem e obtiver um valor diferente daquele presente no campo de checagem de erro da mensagem, significa que a mensagem foi corrompida. Caso os valores coincidam, a mensagem deve estar correta (há a possibilidade

desta ter sido corrompida, mas para isso devem ter ocorrido um número par de erros, na forma de troca de bits, com cada par se cancelando).

As rotinas de cálculo e checagem do LRC estão nas linhas 64-71 e 171-177 do Apêndice A.

6.2.1.7. Configurações do Micro-Controlador

Algumas configurações são necessárias para o funcionamento do micro-controlador da forma prevista. A maioria delas são deixadas como o *default* quando se usa o Project Wizard do compilador PCW v4.038 da CCS. As exceções são: o Power Up Timer é habilitado, evitando que problemas surjam durante a inicialização do código, e as entradas analógicas são todas habilitadas, permitindo que o programa possa selecionar a entrada desejada em tempo de execução.

Além disso, o conversor AD do dispositivo é configurado para operar com valores de 10 bits, e a rotina de inicialização do LCD é chamada.

6.2.1.8. Rotinas de comunicação

Na configuração da UART, os seguintes parâmetros são adotados: taxa de transmissão de 9600, 8 bits, sem paridade.

A comunicação é transparente para o microcontrolador, sendo a transmissão de dados feita pela familiar função *printf*, e a recepção feita por uma captura de um stream de caracteres através de *getc*.

A recepção dos dados utiliza um buffer de recepção, com 12 caracteres, que são preenchidos um a um, a partir da execução de uma rotina de interrupção acionada pela detecção de um dado recebido presente na UART. Nessa interrupção, *getc* é chamada e captura aquele dado presente na UART para o buffer de entrada, incrementando o índice deste. Ao concluir os 12 caracteres de uma mensagem típica, a rotina aciona as funções de tratamento da mensagem. Esses procedimentos podem ser vistos nas linhas 55-57 do Apêndice A.

A transmissão de dados utiliza diretamente a função *printf*, normalmente utilizada em c para a impressão de strings num dispositivo qualquer de interface, como monitores ou impressoras. No microcontrolador usado, o dispositivo de saída padrão é a porta serial. Logo comandos passados por *printf* serão naturalmente enviados pela UART. A transmissão de dados é feita ao final da rotina de tratamento da mensagem, após a mensagem de resposta ser composta. No caso de se usar a rede 485, o pino de controle de fluxo é ativado antes de iniciar a transmissão, permitindo que aquele dispositivo acesse o canal para transmitir. O procedimento de envio dos dados de resposta pode ser visto nas linhas 178-183 do Apêndice A.

6.2.2. Programa do Mestre (PC)

O mestre utiliza uma abordagem de interface com o usuário, ficando suas ações inteiramente dependentes dos comandos passados através da interface gráfica. Para seu desenvolvimento foi utilizado o Borland C++ Builder 6. O programa consta de diversos controles de usuário que permitem que este defina a mensagem de pedido ModBus, selecionando a função e passando os dados necessários.

A interface do programa, em ambiente Windows, é mostrada na Figura 9.

O procedimento do programa é simples. De início o usuário deve selecionar a porta serial à qual se encontra conectada a rede ModBus de destino. Uma vez isso feito, o programa irá abrir essa porta e aguardar que o usuário lhe passe algum comando.

O usuário deve então selecionar o dispositivo escravo com o qual deseja se comunicar, através da caixa de seleção apropriada. Uma vez isso feito, ele pode escolher a função que deseja requisitar, e, caso necessário, selecionar os parâmetros que devem ser passados para a execução dessa função. Tudo feito, o pressionar do botão *"Send Request"* irá fazer com que o programa envie a mensagem de pedido formada para o dispositivo selecionado e aguarde a resposta. Pressionando o botão *"Read Response"* fará com que o programa leia o buffer de entrada e exiba na tela os resultados.

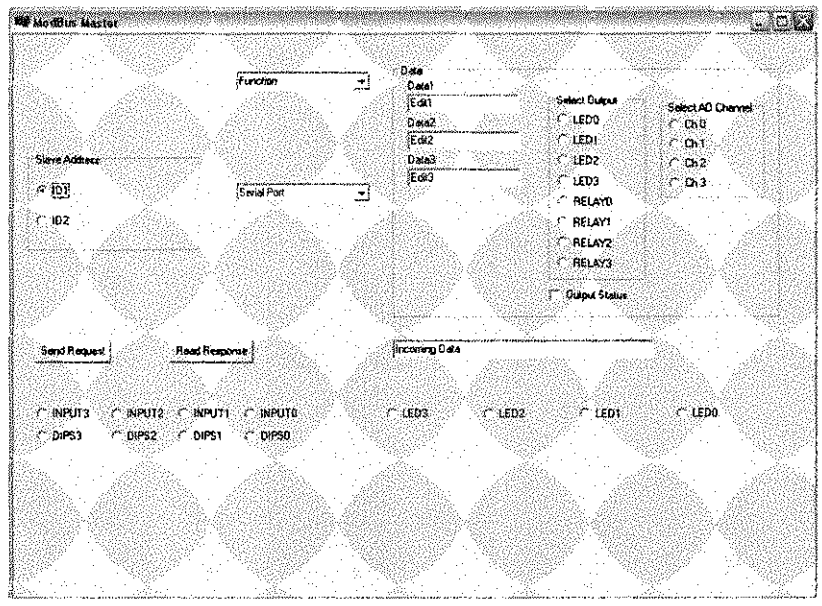

Figura 9 - Interface do programa Mestre

Quanto às funções de checagem de erros, estas são idênticas às implementadas no programa dos escravos, e portanto dispensam revisões.

Para o acesso à comunicação serial, foi utilizada uma biblioteca de componentes proprietária, a TMS Async 32, e portanto, para detalhes de seu funcionamento, referimos ao site do desenvolvedor (8).

6.2.2.1. Leitura de Entradas Digitais

Para a leitura de entradas digitais, o usuário só precisa selecionar essa função na Combo Box das funções e enviar o pedido, não sendo necessário informar nenhum parâmetro.

A função associada à mudança do valor dessa Combo Box irá apenas escrever no campo de função da mensagem a ser enviada o código identificador da função de leitura de entradas digitais, no caso, 02.

Tendo enviado o pedido, ao ler a resposta o programa irá identificar, na mensagem de resposta, os campos de dados, que contêm os estados das entradas digitais do escravo endereçado e irá então associar os Radio Buttons da interface aos valores lidos pelo escravo.

6.2.2.2. *Leitura de Saída Digitais*

De forma semelhante à função anterior, o pedido de leitura de saídas digitais requer apenas a seleção da função, e, ao final da recepção, o programa irá identificar nos Radio Buttons dos Leds, aqueles que se encontram ativos no escravo endereçado.

6.2.2.3. *Leitura de Entradas Analógicas*

Para a chamada da função de leitura de entradas analógicas, o usuário deverá, além de selecionar a respectiva função, identificar qual o canal de conversão analógico-digital que deseja utilizar, dentre os 4 disponibilizados em cada escravo.

Uma vez isso feito, o programa comporá a mensagem de pedido, incluindo no campo de dados o número do canal AD a ser lido. Na resposta, o programa irá exibir no campo de texto identificado por Incoming Data o valor do sinal lido, em uma representação em número real, de 0 a 5, correspondendo ao valor da tensão lido pelo escravo naquela porta.

6.2.2.4. *Escrita de Saídas Digitais*

Para a chamada à função de escrita de saídas digitais o usuário deve, além de selecionar a devida função, informar qual saída deseja modificar e para qual estado esta deve ser levada, através dos Radio Buttons e da Check Box disponíveis.

Uma vez selecionada, o programa irá converter essa seleção no valor numérico da saída e incluir tal campo na mensagem a ser enviada.

6.2.2.5. *Leitura de Temperatura*

A seleção da função de leitura de temperatura é suficiente para o pedido ser completado, sendo o campo da função da mensagem de pedido preenchido com o código adequado, no caso, 66.

Após o recebimento da resposta, o programa irá converter os dados presentes na mensagem recebida para o valor adequado, consistente com uma medida de temperatura, e informar o resultado no campo de Incoming Data.

7. Conclusão

O presente trabalho serviu de valiosa ferramenta para o estudo aprofundado de uma tecnologia predominante na indústria, o protocolo de comunicação para redes industriais ModBus.

Com o desenvolvimento de uma plataforma completa de experimentos, alia-se ao conhecimento específico buscado durante a pesquisa que levou a esse projeto, a utilização de diversas áreas de estudo do curso de Engenharia Elétrica, como o desenvolvimento de sistemas de aquisição de dados, interface com o usuário, programação, eletrônica básica e redes de comunicação.

Considera-se ainda o objetivo do projeto em si, que traz a oportunidade de desenvolver uma ferramenta que irá proporcionar melhores condições de ensino e pesquisa na estrutura da graduação.

8. Referências Bibliográficas

1. **Souza, Vitor Amadeu.** Cerne-tec. *www.cerne-tec.com.br.* [Online] [Citado em: 19 de Janeiro de 2009.] www.cerne-tec.com.br/Modbus.pdf.

2. Wikipedia. *en.wikipedia.org.* [Online] [Citado em: 19 de Janeiro de 2009.] http://en.wikipedia.org/wiki/Modbus.

3. **CEFET-RN.** Apostila de Redes Industriais - Aula 4.

4. Simply Modbus. *www.simplymodbus.ca.* [Online] [Citado em: 19 de Janeiro de 2009.] http://www.simplymodbus.ca/FAQ.htm#Modbus.

5. **Zimmermann, Hubert.** OSI Reference Model. *www.comsoc.org.* [Online] [Citado em: 20 de Janeiro de 2009.] http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf.

6. **Filho, Constantino Seixas.** Protocolos Orientados a Caracter. *www.eletronica.org.* [Online] [Citado em: 21 de Janeiro de 2009.] www.eletronica.org/arq_artigos/ProtocolosCaracterMBUS.PDF.

7. **ModBus-IDA.** ModBus Application Protocol Specification V1.1b. 2006.

8. TMS Software. *TMS Async 32.* [Online] [Citado em: 02 de Fevereiro de 2009.] http://www.tmssoftware.com/site/async32.asp.

9. Modbus Driver. *www.modbusdriver.com.* [Online] [Citado em: 19 de Janeiro de 2009.] http://www.modbusdriver.com/doc/libmbusmaster/modbus.html.

Apêndices

Apêndice A – Código dos Dispositivos Escravos

```
#include <16F877A.h>
#device adc=10

#FUSES NOWDT                    //No Watch Dog Timer
#FUSES XT                       //Crystal osc <= 4mhz
#FUSES PUT                      //Power Up Timer
#FUSES NOPROTECT                //Code not protected from reading
#FUSES NODEBUG                  //No Debug mode for ICD
#FUSES NOBROWNOUT               //No brownout reset
#FUSES NOLVP                    //No low voltage prgming, B3(PIC16) or B5(PIC18) used for I/O
#FUSES NOCPD                    //No EE protection
#FUSES WRT_50%                  //Lower half of Program Memory is Write Protected

#use delay(clock=4000000)
#use rs232(baud=9600,parity=N,xmit=PIN_C6,rcv=PIN_C7,bits=8)

#include <lcd_radio.c>

char request[12];
char response[12];
int i=0;
int j=0;
int16 LRC = 0;
char cLRC_lo;
char cLRC_hi;
long value = 0;
float temp = 0;
int1 p[8] = {0};

#define LED0   PIN_D0
#define LED1   PIN_D1
#define LED2   PIN_D2
#define LED3   PIN_D3

#define REL0   PIN_C3
#define REL1   PIN_C2
#define REL2   PIN_C1
#define REL3   PIN_C0

#define IN0    PIN_B4
#define IN1    PIN_B5
#define IN2    PIN_B6
#define IN3    PIN_B7

#define IN4    PIN_B3
#define IN5    PIN_B2
#define IN6    PIN_B1
#define IN7    PIN_B0

#define REN    PIN_E2

#int_RDA
void  RDA_isr(void)
{
  request[i] = getc();
  i++;
  if(i == 12){
    i = 0;
    printf(LCD, "\f%x %c %c %c %c %c\n%c %c %x %x %x %x",request[0], request[1],
request[2],request[3],request[4],request[5],request[6],request[7],request[8],request[9],request[10],request[11]);
    for(j = 0;j<12;j++)
      response[j] = request[j];                    //repete a mensagem
      if((request[1] == '0')&&(request[2] == '1')){    //checa endereço
      for(j = 1;j<8;j++)
        LRC += request[j];
      LRC = (~LRC) + 1;      //calcula o complemento de 2
```

```
cLRC_lo = LRC;
cLRC_hi = LRC>>8;
LRC = 0;
if((cLRC_hi == request[8]) && (cLRC_lo == request[9]))   //checa o LRC do request
  printf(LCD, "\fLRC correto");
delay_ms(800);
if((request[3] == '0')&&(request[4] == '1')){        //realiza a leitura das saídas digitais
  p[0] = input(LED0);
  p[1] = input(LED1);
  p[2] = input(LED2);
  p[3] = input(LED3);
  p[4] = input(REL0);
  p[5] = input(REL1);
  p[6] = input(REL2);
  p[7] = input(REL3);
  response[5] = p[0]+2*p[1]+4*p[2]+8*p[3]+16*p[4]+32*p[5]+64*p[6]+128*p[7];
}
else if((request[3] == '0')&&(request[4] == '2')){          //realize a leitura das entradas digitais
  p[0] = input(PIN_B4);
  p[1] = input(PIN_B5);
  p[2] = input(PIN_B6);
  p[3] = input(PIN_B7);
  p[4] = input(PIN_B3);
  p[5] = input(PIN_B2);
  p[6] = input(PIN_B1);
  p[7] = input(PIN_B0);
  response[5] = p[0]+2*p[1]+4*p[2]+8*p[3]+16*p[4]+32*p[5]+64*p[6]+128*p[7];
}
else if((request[3] == '0')&&(request[4] == '5')){          //realiza a escrita de uma saída digital
  switch(request[5]){
    case '0':
      if(request[6] == '0')
        output_low(LED0);
      else
        output_high(LED0);
    break;
    case '1':
      if(request[6] == '0')
        output_low(LED1);
      else
        output_high(LED1);
    break;
    case '2':
      if(request[6] == '0')
        output_low(LED2);
      else
        output_high(LED2);
    break;
    case '3':
      if(request[6] == '0')
        output_low(LED3);
      else
        output_high(LED3);
    break;
    case '4':
      if(request[6] == '0')
        output_low(REL0);
      else
        output_high(REL0);
    break;
    case '5':
      if(request[6] == '0')
        output_low(REL1);
      else
        output_high(REL1);
    break;
    case '6':
      if(request[6] == '0')
        output_low(REL2);
      else
        output_high(REL2);
    break;
    case '7':
      if(request[6] == '0')
        output_low(REL3);
      else
```

```
            output_high(REL3);
          break;
          default:
            printf(LCD, "\fInvalid Output");
          break;
        }
      }
      else if((request[3] == '6')&&(request[4] == '5')){          //realiza a leitura de uma entrada analógica
        set_adc_channel(request[5]-48);
        delay_ms(20);
        value = READ_ADC();
        printf(LCD, "\fKIT ModBus\nAD: %.4f ", 0.0048828*value);
        response[5] = (char)value;
        response[6] = (char)((value>>8)&0x03);
      }
      else if((request[3] == '6')&&(request[4] == '6')){          //realiza a leitura da temperatura
        value = READ_ADC();
        temp = 0.48828*value;
        printf(LCD, "\fKIT ModBus\nTEMP: %.2f ",temp);
        response[5] = (char)value;
        response[6] = (char)((value>>8)&0x03);
      }
      else{                                                       //resposta default para uma função desconhecida
        response[5] = 0xFF;
        response[6] = 0xFF;
        response[7] = 0xFF;
        printf(LCD, "\fFuncUnknown");
      }
      for(j = 1;j<8;j++)
        LRC += response[j];
      LRC = (~LRC) + 1;                   //calcula o LRC da response
      response[9] = LRC;                  //LRC_lo
      response[8] = LRC>>8;               //LRC_hi
      LRC = 0;
      response[10] = 0x0D;
      response[11] = 0x0A;
      output_high(REN);
      delay_ms(10);
      for(j = 0;j<12;j++)
        printf("%c",response[j]);    //envia a resposta para o mestre
      output_low(REN);
    }
  //  printf(LCD, "\f");
  }
}


void main()
{

  setup_adc_ports(ALL_ANALOG);
  setup_adc(ADC_CLOCK_INTERNAL);
  setup_psp(PSP_DISABLED);
  setup_spi(SPI_SS_DISABLED);
  setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
  setup_timer_1(T1_DISABLED);
  setup_timer_2(T2_DISABLED,0,1);
  setup_comparator(NC_NC_NC_NC);
  setup_vref(FALSE);
  LCD_init();
  printf(LCD, "\f->");
  enable_interrupts(INT_RDA);
  enable_interrupts(GLOBAL);

  set_adc_channel(6);

  output_low(REN);

  while(1);

}
```

Apêndice B – Código do Programa Mestre

```
#include <vcl.h>
#pragma hdrstop

#include "dtr.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "VaClasses"
#pragma link "VaComm"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    VaComm1->SetDTR(false);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormShow(TObject *Sender)
{
    VaComm1->SetDTR(false);
    //Default request (unless you change something on the UI, this will be sent):
    request[0] = 0x3A;    //Start character
    request[1] = '0';     //Slave Address high
    request[2] = '1';     //Slave Address low
    request[3] = '0';     //Function code
    request[4] = '0';     //colocar aqui a função do check link
    request[5] = '0';     //3 chars for data
    request[6] = '0';
    request[7] = '0';
    request[8] = 0;     //calcular o LRC default
    request[9] = 0;
    request[10] = 0x0D;    //Carriage Return
    request[11] = 0x0A;    //Line Feed
    request[12] = '\0';    //String terminator

    LRC = 0;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    VaComm1->PurgeReadWrite();
    VaComm1->Close();
}
//---------------------------------------------------------------------------


void __fastcall TForm1::RadioButton1Click(TObject *Sender)
{
    request[1] = '0';
    request[2] = '1';
}
//---------------------------------------------------------------------------

void __fastcall TForm1::RadioButton2Click(TObject *Sender)
{
    request[1] = '0';
    request[2] = '2';
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ComboBox1Change(TObject *Sender)
{
    function = ComboBox1->ItemIndex;
    switch (function){
        case 0: //Read  Digital Inputs
            request[3] = '0';
```

```
                request[4] = '2';
            break;
            case 1: //Read Digital Outputs
                request[3] = '0';
                request[4] = '1';
            break;
            case 2: //Read Analog Input
                request[3] = '6';
                request[4] = '5';
            break;
            case 3: //Write Digital Output
                request[3] = '0';
                request[4] = '5';
            break;
            case 4: //Read Board Temperature
                request[3] = '6';
                request[4] = '6';
            break;
            default:
                request[3] = '7';
                request[4] = '1';
    }
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button4Click(TObject *Sender)
{
    switch (function){
        case 0: //Read  Digital Inputs

        break;
        case 1: //Read Digital Outputs

        break;
        case 2: //Read Analog Input
            request[5] = (char)RadioGroup2->ItemIndex+48;
        break;
        case 3: //Write Digital Output
            request[5] = (char)RadioGroup3->ItemIndex+48;
            if(CheckBox1->Checked)
                request[6] = '1';
            else
                request[6] = '0';
        //      Edit4->Text = request[5];
        break;
        case 4: //Read Board Temperature

        break;
        case 5: //Read Slave Time

        break;
        case 6: //Read Slave Date

        break;
        case 7: //Set Slave Time

        break;
        case 8: //Set Slave Date

        break;
        case 9: //Check Link

        break;
        default:

        break;
    }

    for(int i=1;i<8;i++)
        LRC+=request[i];
    LRC =(~LRC) + 1;
    request[9] = LRC;
    request[8] = LRC>>8;

    VaComm1->SetDTR(true);
    Sleep(10);
```

```
    VaComm1->WriteBuf(request,12);
    Sleep(10);
    VaComm1->SetDTR(false);

}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button5Click(TObject *Sender)
{
    VaComm1->ReadBuf(response,12);
    Edit5->Text = (int)response[5];

    switch (function){
        case 0: //Read  Digital Inputs
            RadioButton10->Checked = response[5] & 0x01;
            RadioButton9->Checked = response[5] & 0x02;
            RadioButton8->Checked = response[5] & 0x04;
            RadioButton7->Checked = response[5] & 0x08;
            RadioButton6->Checked = response[5] & 0x10;
            RadioButton5->Checked = response[5] & 0x20;
            RadioButton4->Checked = response[5] & 0x40;
            RadioButton3->Checked = response[5] & 0x80;
        break;
        case 1: //Read Digital Outputs
            RadioButton18->Checked = response[5] & 0x01;
            RadioButton17->Checked = response[5] & 0x02;
            RadioButton16->Checked = response[5] & 0x04;
            RadioButton15->Checked = response[5] & 0x08;
        /*      RadioButton14->Checked = response[5] & 0x10;
            RadioButton13->Checked = response[5] & 0x20;
            RadioButton12->Checked = response[5] & 0x40;
            RadioButton11->Checked = response[5] & 0x80;    */
        break;
        case 2: //Read Analog Input
            Edit5->Text = 0.0048828*(256*response[6]+response[5]);
        break;
        case 3: //Write Digital Output

        break;
        case 4: //Read Board Temperature
            Edit5->Text = 0.48828*(256*response[6]+response[5]);
        break;
        default:

        break;
    }
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ComboBox2Change(TObject *Sender)
{
    VaComm1->PortNum = ComboBox2->ItemIndex+1;
    VaComm1->Open();
}
//---------------------------------------------------------------------------
```