Capítulo

3

Protocolos Orientados a Caracter

Protocolos Orientados a Caracter

Protocolo Modbus

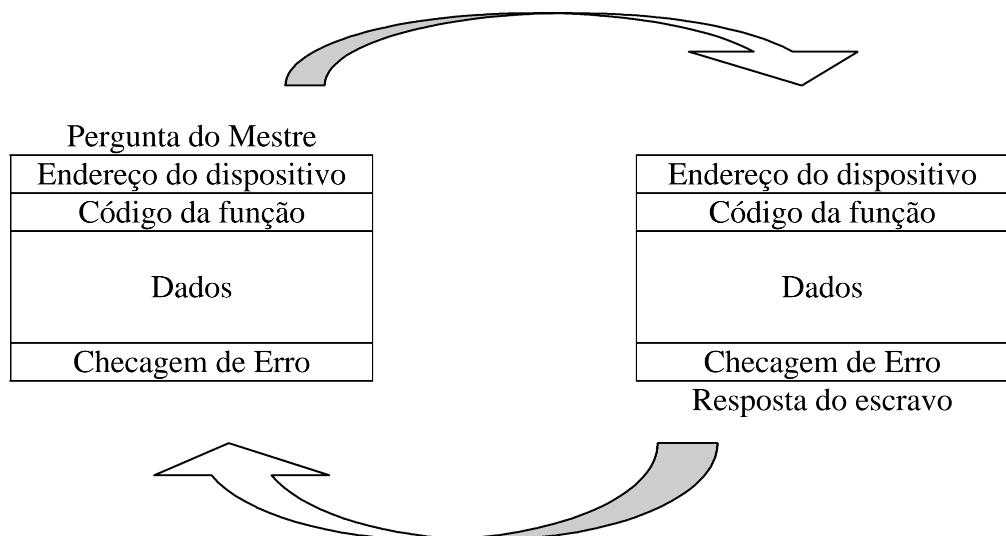
Descrição

O protocolo Modbus foi desenvolvido pela Modicon Industrial Automation Systems, hoje Schneider, para comunicar um dispositivo mestre com outros dispositivos escravos. Embora seja utilizado normalmente sobre conexões seriais padrão RS-232, ele também pode ser usado como um protocolo da camada de aplicação de redes industriais tais como TCP/IP sobre Ethernet e MAP. Este á talvez o protocolo de mais larga utilização em automação industrial, pela sua simplicidade e facilidade de implementação. O texto que se segue foi modificado a partir da referência [Souza 1999]

Modelo de Comunicação

O protocolo Modbus é baseado em um modelo de comunicação mestre-escravo, onde um único dispositivo, o mestre, pode iniciar transações denominadas *queries*. O demais dispositivos da rede (escravos) respondem, suprindo os dados requisitados pelo mestre ou executando uma ação por ele comandada. Geralmente o mestre é um sistema supervisório e os escravos são controladores lógico programáveis. Os papéis de mestre e escravo são fixos, quando se utiliza comunicação serial, mas em outros tipos de rede, um dispositivo pode assumir ambos os papéis, embora não simultaneamente.

O ciclo pergunta-resposta:



Modos de Transmissão

Existem dois modos de transmissão: ASCII (*American Code for Informastion Interchange*) e RTU (*Remote Terminal Unit*), que são selecionados durante a configuração dos parâmetros de comunicação.

<table>
<tr><td>ASCII</td><td colspan="2">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`.</td></tr>
<tr><td></td><td>10 bits por byte</td><td>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</td></tr>
<tr><td></td><td>Campo de Checagem de Erros</td><td>Longitudinal Redundancy Check</td></tr>
<tr><td>RTU</td><td colspan="2">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.</td></tr>
<tr><td></td><td>11 bits por byte</td><td>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</td></tr>
<tr><td></td><td>Campo de Checagem de Erros</td><td>CRC</td></tr>
</table>

Formato de Mensagem

Modo ASCII

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

Modo RTU

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

Como a comunicação geralmente utilizada em automação industrial é em modo RTU, vamos nos concentrar nesta forma de comunicação.

- **Endereço**: A faixa de endereços válidos vai de 0 a 247 (`0x00` a `0xf7` hexadecimal), sendo que os dispositivos recebem endereços de um a 247. O endereço zero é reservado para *broadcast*, ou seja, mensagens com esse valor de endereço são reconhecidas por todos os elementos da rede. Quando há outros protocolos de rede abaixo do Modbus, o endereço é normalmente convertido para o formato utilizado pelos mesmos. Nesse caso, pode ser que o mecanismo de *broadcast* não seja suportado. Quando o master envia uma mensagem para os escravos, este campo contém o endereço do escravo. Quando o escravo responde, coloca seu próprio endereço neste campo.

- **Código de Função**: Varia de 1 a 255 (`0x01` a `0xff`), mas apenas a faixa de um a 127 (`0x01` a `0x7f`) é utilizada, já que o *bit* mais significativo é reservado para indicar respostas de exceção. Normalmente, uma resposta inclui o código de função da requisição que lhe deu origem. No entanto, em caso de falha, o *bit* mais significativo do código é ativado para indicar que o conteúdo do campo de dados não é a resposta esperada, mas sim um código de diagnóstico.

- **Dados**: Tamanho e conteúdo do campo de dados variam com a função e o papel da mensagem, requisição ou resposta, podendo mesmo ser um campo vazio.

- ***Frame Check Sequence***: O campo de *frame check sequence* contém um valor de oito ou dezesseis *bits*, dependendo do modo de transmissão serial, que é utilizado para detecção de erros na mensagem. Quando eventuais camadas de rede subjacentes provêem mecanismos de detecção de erros próprios, o FCS é dispensado. O LRC corresponde ao complemento de 2 da soma de todos os bytes da mensagem excluídos o byte de start e end. O CRC é uma variante do CRC16 já estudado.

Os únicos identificadores através dos quais o dispositivo mestre pode reconhecer a resposta para uma determinada mensagem são o endereço do dispositivo escravo e a função solicitada. Assim, o envio de múltiplas requisições, em que tais parâmetros coincidam, deve ser feito ordenadamente, isto é, cada mensagem só deve ser enviada, depois que a resposta para a mensagem anterior for recebida. Não há problema em se enviar simultaneamente comandos iguais para dispositivos diferentes ou comandos diferentes para um mesmo dispositivo,

embora nesse último caso possam surgir problemas dependendo do equipamento específico.

Principais Funções

Read Coil Status

- **Finalidade**: Leitura de estados das saídas discretas do dispositivo escravo, endereços na forma `0nnnn`.
- **Código de Função**: `0x01`.
- **Formato da Requisição**:
  Endereço do dispositivo...1 *byte*
  Código de função............1 *byte*, `0x01`
  Endereço inicial ....2 *bytes* [1]
  Número de saídas............2 *bytes* [1]
  FCS ....1 ou 2 *bytes* [1]
- **Formato da Resposta**:
  Endereço do dispositivo...1 *byte*
  Código de função............1 *byte*, `0x01`
  Tamanho da resposta ....1 *byte*, `n`
  Resposta..........................`n` *bytes* [2]
  FCS ....1 ou 2 *bytes* [1]

[1] Em valores de 2 *bytes*, o *byte* mais significativo é por convenção representado primeiro.

[2] Cada grupo de oito saídas é representado por um *byte*, onde cada saída corresponde a um *bit* individual (`1 =` ligada, `0 =` desligada). Esses *bits* são ordenados de acordo com os endereços das saídas, com o endereço inicial no *bit* menos significativo. Caso haja mais de oito saídas os múltiplos *bytes* de resposta são também ordenados por endereço, com os *bytes* representando os menores endereços primeiro.

Read Input Status

- **Finalidade**: Leitura de estados das entradas discretas do dispositivo escravo, endereços na forma `1nnnn`.
- **Código de Função**: `0x02`.
- **Formato da Requisição**:
  Endereço do dispositivo...1 *byte*
  Código de função............1 *byte*, `0x02`
  Endereço inicial ....2 *bytes* [1]
  Número de entradas ....2 *bytes* [1]
  FCS ....1 ou 2 *bytes* [1]

- **Formato da Resposta**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x02`
Tamanho da resposta ....1 *byte*, `n`
Resposta..........................`n` *bytes* [2]
FCS ....1 ou 2 *bytes* [1]

[1] *Byte* mais significativo representado primeiro.

[2] A representação é semelhante àquela das saídas, ou seja, cada entrada corresponde a um *bit*, sendo esses *bits* agrupados de oito em oito em *bytes*. *Bits* e *bytes* são ordenados por endereço, de forma crescente a partir do *bit* menos significativo do primeiro *byte*.

Read Holding Registers

- **Finalidade**: Leitura de valores dos *holding registers* do dispositivo escravo, endereços na forma `4nnnn`.

- **Código de Função**: `0x03`.

- **Formato da Requisição**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x03`
Endereço inicial ....2 *bytes* [1]
Número de registradores..2 *bytes* [1]
FCS ....1 ou 2 *bytes* [1]

- **Formato da Resposta**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x03`
Tamanho da resposta ....1 *byte*, `n`
Resposta..........................`n` *bytes* [2]
FCS ....1 ou 2 *bytes* [1]

[1] *Byte* mais significativo representado primeiro.

[2] Os registradores são representados por ordem crescente de endereço. Cada registrador armazena normalmente um valor de 16 *bits*, dois *bytes*, dos quais o mais significativo vem representado primeiro. Registradores de 32 *bits* ocupam dois endereços consecutivos, com os 16 *bits* mais significativos contidos no primeiro endereço.

Read Input Registers

- **Finalidade**: Leitura de valores dos *input registers* do dispositivo escravo, endereços na forma `3nnnn`.

- **Código de Função**: `0x04`.

- **Formato da Requisição**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x04`
Endereço inicial ....2 *bytes* [1]
Número de registradores..2 *bytes* [1]
FCS ....1 ou 2 *bytes* [1]

- **Formato da Resposta**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x04`
Tamanho da resposta ....1 *byte*, `n`
Resposta...........................`n` *bytes* [2]
FCS ....1 ou 2 *bytes* [1]

[1] *Byte* mais significativo representado primeiro.

[2] Representação similar à dos *holding registers*.

Force Single Coil

- **Finalidade**: Escrita de um valor de saída discreta, endereço `0nnnn`, no dispositivo escravo. Esse valor permanece constante enquanto não for alterado por uma nova operação de escrita ou pela programação interna do dispositivo.

- **Código de Função**: `0x05`.

- **Formato da Requisição**: Endereço do dispositivo...1 *byte*
Código de função............1 *byte*, `0x05`
Endereço ....2 *bytes* [1]
Valor ....2 *bytes* [1,2]
FCS ....1 ou 2 *bytes* [1]

- **Formato da Resposta**: Cópia da requisição.

[1] *Byte* mais significativo representado primeiro.

[2] Ligado = `0xff00`, desligado = `0x0000`.

Preset Single Register

- **Finalidade**: Escrita de um valor de *holding register*, endereço `4nnnn`. Assim como acontece para as saídas, o valor no registrador permanece constante enquanto não for alterado por operações de escrita ou pelo próprio dispositivo.

- **Código de Função**: `0x06`.

- **Formato da Requisição**: Endereço do dispositivo... 1 *byte*
  Código de função............ 1 *byte*, `0x06`
  Endereço .... 2 *bytes* [1]
  Valor .... 2 *bytes* [1]
  FCS .... 1 ou 2 *bytes* [1]

- **Formato da Resposta**: Cópia da requisição.

[1] *Byte* mais significativo representado primeiro.

Detecção de Erros

Há dois mecanismos para detecção de erros no protocolo Modbus serial: *bits* de paridade em cada caractere e o *frame check sequence* ao final da mensagem.

A verificação de paridade é opcional em ambos os modos de transmissão, ASCII e RTU. Um *bit* extra é adicionado a cada caractere de modo que ele contenha um número par ou ímpar de *bits* ativos, caso sejam adotadas respectivamente paridade par ou ímpar. A principal desvantagem desse método é sua incapacidade de detectar números pares de inversões de *bit*. Caso não seja utilizado, o *bit* de paridade é substituído por um *stop bit* adicional.

O campo de *frame check sequence* no modo ASCII é preenchido com um valor de oito *bits*, o *Longitudinal Redundancy Check* ou LRC, que é o complemento de dois da soma em oito *bits* dos octetos que compõe a mensagem. Os caracteres delimitadores (`:`, `cr` e `lf`) não são considerados no cálculo do LRC.

Já o modo RTU utiliza como *frame check sequence* um valor 16 *bits*, o CRC, utilizando como polinômio, $P(x) = x^{16} + x^{15} + x^{2} + 1$. O registro de cálculo do CRC deve ser inicializado com o valor 0xFFFF.

Tabela sumário de códigos de função [Referência: 984]

Função	Descrição	Query Max	Response Max
1	Leitura de Status de Bobinas	2000 bobs	2000
2	Leitura de Status de Entradas	2000	2000
3	Ler Registradores Internos	125	125
4	Leitura de Registradores de Entrada	125	125
5	Força Bobina Simples	1	1
6	Escrita em Registrador Simples	1	1
7	Leitura de Status de Exceção	N/A	8
8	Diagnósticos	N/A	N/A
9	Programação do CLP 484	Não suportado	Não suportado
10	Poll do CLP 484	Não suportado	Não suportado
11	Busca Contador de Evento de Comunicação	N/A	N/A
12	Busca Relatório de Evento de Comunicação	N/A	70 bytes de dados
13	Programa Controlador	33	33 bytes de dados
14	Poll Controlador	N/A	33
15	Força Múltiplas Bobinas	800	800
16	Escrita em múltiplos registradores	100 regs	100 regs
17	Retorna descrição do Dispositivo Escravo	N/A	N/A
18	Programa 884/M84	Não suportado	Não suportado
19	Reseta link de comunicação	Não suportado	Não suportado
20	Leitura de registros na área de memória estendida.	Length(Msg) <256 bytes	Length(Msg) <256 bytes
21	Escrita de registros na área de memória estendida.	Length(Msg) <256 bytes	Length(Msg) <256 bytes
22	Mascara bits de um registrador		
23	Realiza leitura e escrita de registros numa mesma transação Modbus.		
24	Realiza leitura da fila FIFO		

Protocolos Derivados do BISYNC – Binary Synchronous Communication Protocol.

Este protocolo é muito utilizado na comunicação entre dispositivos mestre e CLPs e na comunicação entre diferentes dispositivos de campo. O protocolo mais conhecido dentro desta classe é o DF1 definido pela Allen Bradley. O quadro de dados consiste em campos delimitados por caracteres especiais. O campo de dados é delimitado pelos caracteres DLE|STX (início) e DLE|ETX (final). Cada quadro é encerrado com dois bytes de CRC, em geral CRC16. O caracter DLE (*Data Link Enquiry*) serve para proporcionar transparência de dados. DLE DLE no campo de dados eqüivale a um dado de valor DLE. Este protocolo foi muito empregado, por ser de fácil implementação, servindo de base da diversos protocolos e aplicativos de comunicação do mercado como o *talk* no S.O. QNX, e o protocolo *Kermit* em UNIX. Kermit é um protocolo de transferência de arquivos textos ou binários, desenvolvido pela Universidade de Columbia em 1981 para transmissão de dados, livre de erros, utilizando quaisquer computadores e sistemas operacionais, em canais hostis.

Formato dos dados:

DLE	STX	DADOS	DLE	ETX	CRC	CRC

O algoritmo de funcionamento deste protocolo é mostrado nas páginas seguintes.

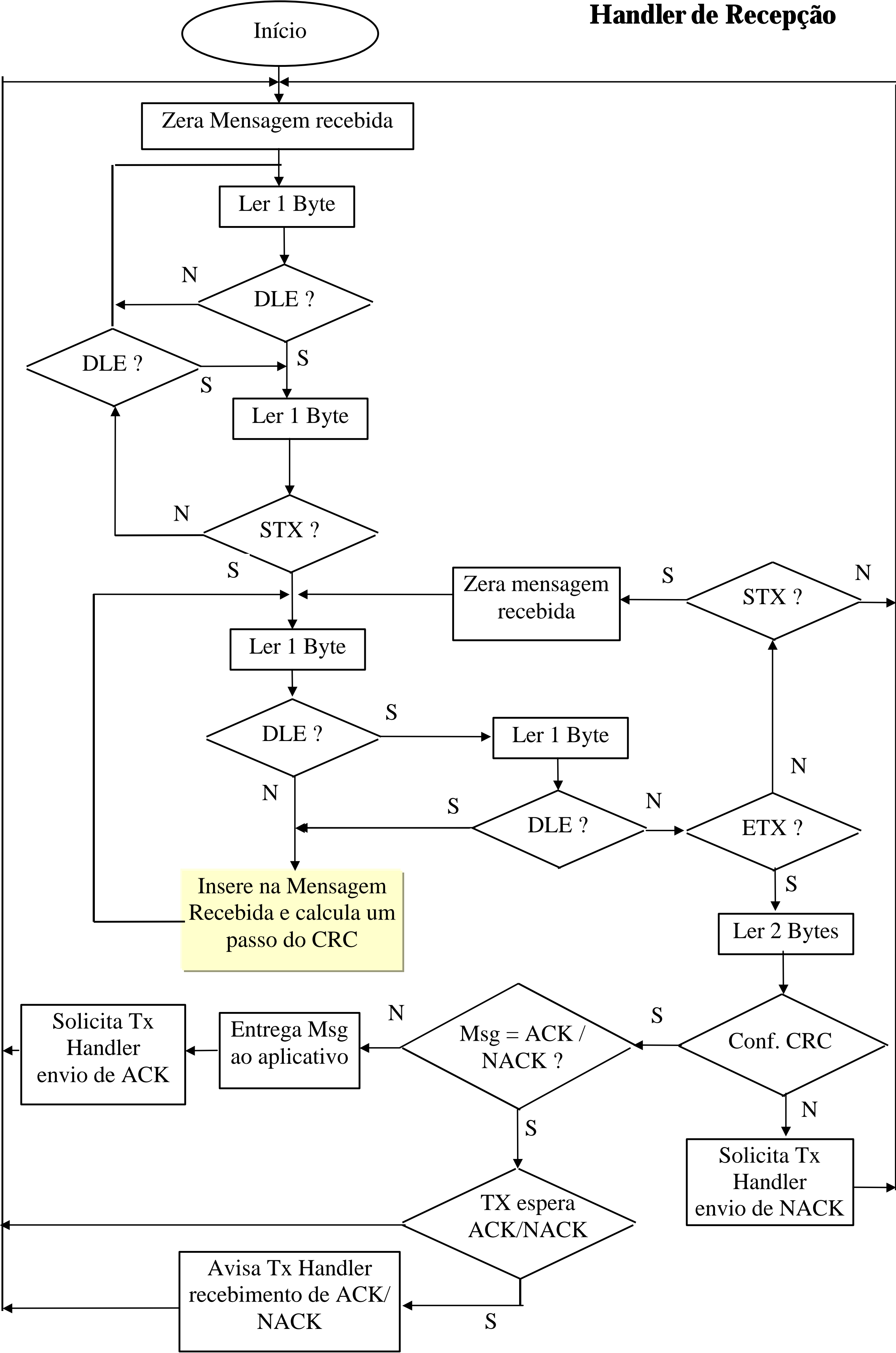
Nós devemos codificar este algoritmo buscando concentrar em duas threads as funções de transmissão e recepção. O protocolo deve funcionar em full duplex.

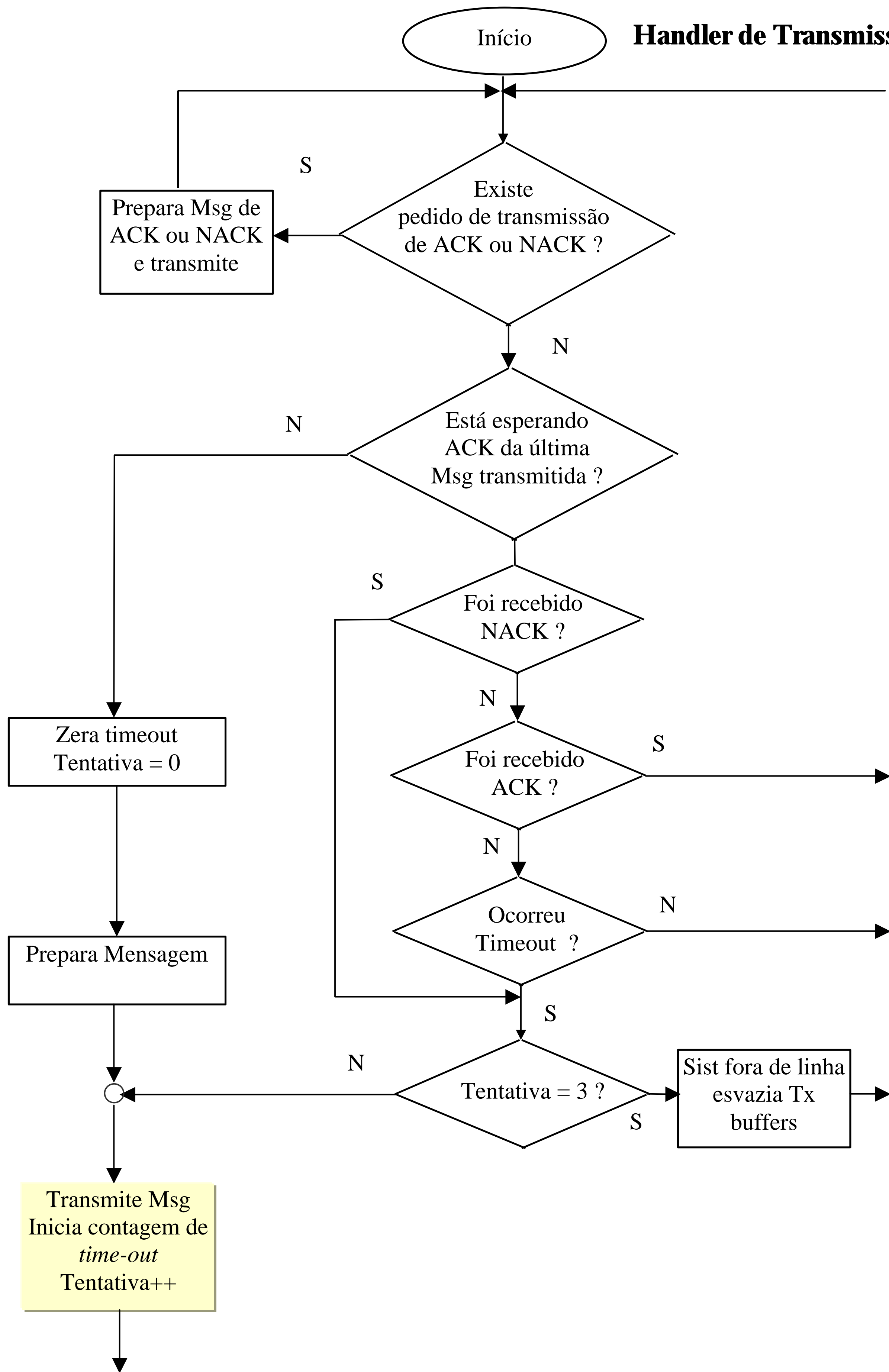
A thread primária irá criar as threads de transmissão e recepção. A partir daí poderá enviar e receber dados utilizando este protocolo.

Para enviar dados irá enviar uma mensagem para ThreadTx indicando o endereço do buffer contendo a mensagem a ser enviada e o tamanho da mensagem em bytes.
Depois ficará à espera do evento de fim de transmissão, que pode retornar SUCESSO ou FRACASSO.
A função *PostMessage()* é utilizada para o envio de mensagens nos dois sentidos. Esta função é assíncrona e coloca a mensagem na fila de entrada da thread receptora. Como apenas GUI threads possuem fila de entrada, a instrução *PeekMessage()* foi incluída no início de cada programa para forçar a criação de uma fila de mensagem em uma “working thread”.

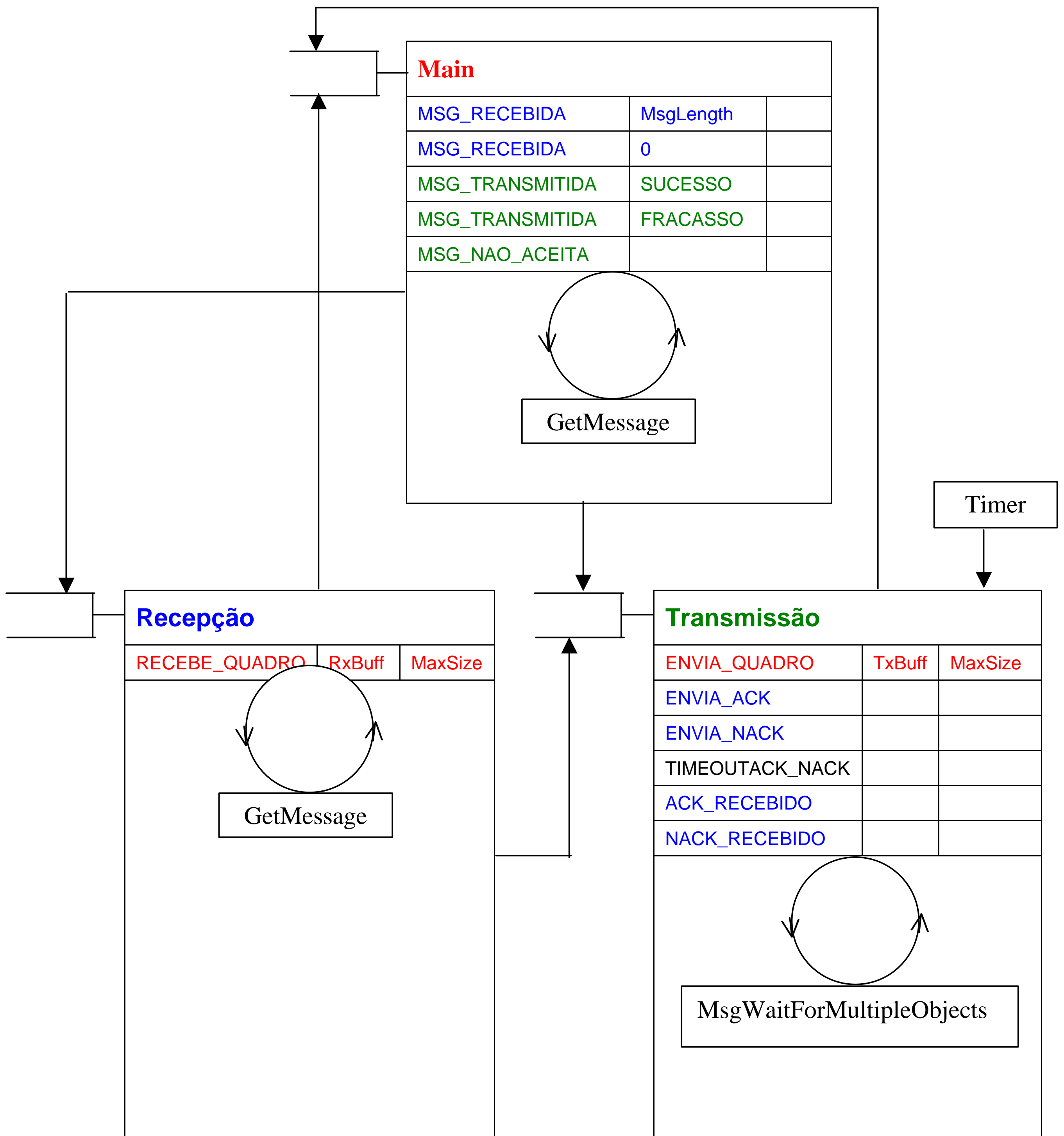
Para receber dados, a thread primária deve informar à ThreadRx, o endereço do buffer de recepção e ficar à espera de um evento de fim de recepção da mensagem. Quando este evento ocorre, a thread recebe o tamanho da mensagem recebida em bytes.

O código a seguir é apenas um esboço da solução completa.


Início
Zera Mensagem recebida
Ler 1 Byte
DLE ?
N
S
DLE ?
S
Ler 1 Byte
STX ?
N
S
Zera mensagem recebida
STX ?
S
N
Ler 1 Byte
DLE ?
S
Ler 1 Byte
N
DLE ?
S
N
ETX ?
N
S
Insere na Mensagem Recebida e calcula um passo do CRC
Ler 2 Bytes
Solicita Tx Handler envio de ACK
Entrega Msg ao aplicativo
Msg = ACK / NACK ?
N
S
Conf. CRC
S
N
Solicita Tx Handler envio de NACK
TX espera ACK/NACK
Avisa Tx Handler recebimento de ACK/ NACK
S


Início
Handler de Transmissão
S
Existe
pedido de transmissão
de ACK ou NACK ?
Prepara Msg de
ACK ou NACK
e transmite
N
N
Está esperando
ACK da última
Msg transmitida ?
S
Foi recebido
NACK ?
N
Zera timeout
Tentativa = 0
Foi recebido
ACK ?
S
N
Ocorreu
Timeout ?
N
Prepara Mensagem
S
N
Tentativa = 3 ?
S
Sist fora de linha
esvazia Tx
buffers
Transmite Msg
Inicia contagem de
time-out
Tentativa++

Mensagens entre as Threads:



Exemplo de Implementação:

```
// TxRx.c
//
#define WIN32_LEAN_AND_MEAN
#define WIN32_WINNT  0x0400  // Necessário para ativar novas funções da versão 4
#include <windows.h>
#include <process.h>          // _beginthreadex() e _endthreadex()
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "CheckForError.h"
#define _CHECKERROR    1        // Ativa função CheckForError

// Casting para terceiro e sexto parâmetros da função _beginthreadex
typedef unsigned (WINAPI *CAST_FUNCTION)(LPVOID);
typedef unsigned *CAST_LPDWORD;

// Protótipos de funções
DWORD WINAPI ThreadRxFrame(LPVOID);
DWORD WINAPI ThreadTxFrame(LPVOID);

#define CRC_16                  0xA001

#define MAX_MSG_SIZE            512

// Caracteres de controle ASCII
#define         DLE             0x10
#define         STX             0x02
#define         ETX             0x03
#define         ACK             0x06
#define         NACK            0x15

// Estados do Autômato de Recepção
#define INICIALIZA              0
#define ESPERA_DLE              1
#define ESPERA_STX              2
#define LE_DADOS                3
#define LE_CRC                  4
#define PROCESSA_MSG_OK         5
#define PROCESSA_MSG_NOK        6
#define PROCESSA_ACK            7
#define PROCESSA_NACK           8

typedef struct {
        char *lpBuffer;         // Apontador para buffer de transmissão ou recepção
        DWORD dwMsgLength;      // Número de bytes a transmitir ou receber
} MsgData;

#define WM_TXRX_MSG                     WM_APP + 0x0000

// Mensagens da thread de recepção para a de transmissão
#define ENVIA_ACK                       0
#define ENVIA_NACK                      1
#define ACK_RECEBIDO                    2
#define NACK_RECEBIDO                   3

// Mensagens da thread primária para a de Recepção/Transmissão
#define         ENVIA_QUADRO            4
```

```
#define         RECEBE_QUADRO         5

// Mensagem da thread de transmissão para ela mesma para codificar evento do timer
#define TIMEOUT_ACK_NACK  6        // Timeout do timer de espera de ACK/NACK

// Mensagens das threads de transmissão/recepção para a aplicação (thread primária)
#define MSG_TRANSMITIDA              7
#define MSG_RECEBIDA                 8
#define MSG_NAO_ACEITA               9

#define SUCESSO                       0
#define FRACASSO                     -1

char ReadByte (void); // Le um byte da entrada serial. Só retorna se caracter estiver disponível.
char WriteByte        (char ); // Escreve um byte na saída serial
unsigned CalcCRC(char *, int, unsigned);
void TxQuadro         (char *, int);
void FormatAndPostMsg(DWORD, int , DWORD, DWORD );
char ReadByte         (void);

char RxBuffer[MAX_MSG_SIZE];
char TxBuffer[MAX_MSG_SIZE];

char Msg1[]= "Primeira Mensagem a ser transmitida";

BOOL bEsperaAck; //Flag: Autômato de transmissão está a espera de ACK/NACK

HANDLE hTx;
HANDLE hRx;
HANDLE hEventTx;
HANDLE hEventRx;

// Id das threads de transmissão e recepção
// Importante para mandar mensagem para tarefas filhas
DWORD       dwTxId, dwRxId;
// Id da thread primária
DWORD       dwCommId;

int main(VOID)
{
   DWORD   dwExitCode;
        DWORD       dwStatus;
        MSG         msg;
        DWORD       dwFunction;
        DWORD       dwRxBytes;   // Número de bytes recebidos
        BOOL bRet;

        dwCommId = GetCurrentThreadId();
        // Cria fila de mensagens
        PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

        // Cria um objeto do tipo evento para cada thread de comunicação
        // O evento será sinalizado quando a thread estiver apta a receber mensagens
        hEventTx = CreateEvent(NULL, TRUE, FALSE, NULL);
        hEventRx = CreateEvent(NULL, TRUE, FALSE, NULL);

        // Cria thread de recepção de dados
        hRx = (HANDLE) _beginthreadex(
                    NULL,
                    0,
```

```
                    (CAST_FUNCTION) ThreadRxFrame,     // casting necessário
                    0,
                    0,
                    (CAST_LPDWORD)&dwRxId              // casting necessário
          );
CheckForError(hRx);
if (hRx) printf("Thread RX criada Id= %0x \n", dwRxId);

// Cria thread de Transmissão de dados
hTx = (HANDLE) _beginthreadex(
                    NULL,
                    0,
                    (CAST_FUNCTION) ThreadTxFrame,     // casting necessário
                    0,
                    0,
                    (CAST_LPDWORD)&dwTxId              // casting necessário
          );
CheckForError(hTx);
if (hTx) printf("Thread TX criada Id= %0x \n", dwTxId);

// Espera sincronismo para iniciar comunicação
WaitForSingleObject(hEventTx, INFINITE);
WaitForSingleObject(hEventRx, INFINITE);

// Prepara recepção de Mensagem
FormatAndPostMsg(dwRxId, RECEBE_QUADRO, (char *)RxBuffer,MAX_MSG_SIZE);

// Envia Mensagem
ZeroMemory(TxBuffer, MAX_MSG_SIZE);
strcpy(TxBuffer, Msg1);
FormatAndPostMsg(dwTxId, ENVIA_QUADRO, (char *)TxBuffer, sizeof(Msg1)+1);
/ / Transmite inclusive caracter fim de string

// Espera comunicação de fim de transmissão ou recepção
bRet = GetMessage(&msg, NULL, 0, 0);
dwFunction = (DWORD) ((msg).wParam);

switch (dwFunction) {
          case MSG_TRANSMITIDA:
                    dwStatus  = (DWORD) ((msg).lParam);
                    if (dwStatus == SUCESSO)
                              printf ("\nPAI Mensagem transmitida com Sucesso\n");
                    else printf ("\nPAI Erro na transmissao da Mensagem\n");
                    break;

          case MSG_RECEBIDA:
                    dwRxBytes  = (DWORD) ((msg).lParam);
                    printf ("\nPAI: Mensagem recebida: %d bytes: \n %s\n", dwRxBytes,
                    RxBuffer);
                    break;

          case MSG_NAO_ACEITA:
                    printf ("\nPAI: Mensagem não aceita. Existe outra mensagem em processo de
                    transmissao\n");
                    break;

          default:
                    printf ("\nPAI: Mensagem desconhecida\n");
                    break;
}  // switch
```

```
	// Encerra threads de comunicação
	PostThreadMessage(dwTxId, WM_QUIT, NULL, NULL);

	// Thread de recepção pode não estar no estado inicial onde está apta a receber
	// msg da aplicação
	TerminateThread(hRx, 0);	// Termina thread com exit_code 0

	// Espera threads de comunicação terminarem
	WaitForSingleObject(hTx, INFINITE);
	GetExitCodeThread(hTx, &dwExitCode);
	CloseHandle(hTx);

	WaitForSingleObject(hRx, INFINITE);
	GetExitCodeThread(hRx, &dwExitCode);
	CloseHandle(hRx);

	CloseHandle(hEventRx);
	CloseHandle(hEventTx);

	return EXIT_SUCCESS;
}	// main



// Rx Frame é uma thread independente que recebe frames de dados.
// Inputs: RxFrame recebe da aplicação o endereço do buffer de dados
// Outputs: RxFrame reporta o número de bytes recebidos e
//	o status da transação: 0 se OK e -1 se houve problema.
//
DWORD WINAPI ThreadRxFrame(LPVOID n)
{
	char	cInput;			// Byte lido da linha serial
	int		nState;		// Estado do FSA
	int		nNextState;		// Próximo estado do FSA
	int		nCRCLow;		// Byte menos significativo do CRC
	int		nCRCHigh;		// Byte mais significativo do CRC
	int		nCRC;			// CRC calculado
	int		nCRCrx;			// Valor recebido do CRC
	int		nMsgLength;		// Tamanho da mensagem recebida
	DWORD	dwMaxMsgLength;	// Tamanho máximo da mensagem a receber
	MSG		msg;
	char	*pRxBuffer;
	MsgData *pMsgData;
	BOOL bRet;

	// Cria fila de mensagens
	PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);
	bRet = SetEvent(hEventRx);		// Thread de transmissão pronta
	CheckForError(bRet);

	nState = INICIALIZA;
	while(1) {  // Parser
		switch(nState) {
		case INICIALIZA:
			// Fica a espera do endereço do buffer de recepção
			bRet = GetMessage(&msg, NULL, 0, 0);

			// única Mensagem recebida é RECEBE_QUADRO
			pMsgData=	(MsgData *)((msg).lParam);
```

```
        pRxBuffer=  (char *)((*pMsgData).lpBuffer); // End Buffer
        // Tamanho Máximo da Msg a ser aceita
        dwMaxMsgLength = (DWORD)((*pMsgData).dwMsgLength);
        free(pMsgData);      // libera memória alocada

        ZeroMemory(pRxBuffer, MAX_MSG_SIZE);
        nMsgLength = 0;
        nNextState = ESPERA_DLE;
        printf("\nRX: Inicializa");
        break;

    case ESPERA_DLE:
        while ((cInput = ReadByte()) != DLE);    // espera DLE
        nNextState = ESPERA_STX;
        break;

    case ESPERA_STX:
        cInput = ReadByte();          // le um byte da entrada
        if (cInput == STX)
            nNextState = LE_DADOS;
        else if (cInput == DLE)
            nNextState = ESPERA_STX;
        else nNextState = ESPERA_DLE;
        break;

    case LE_DADOS:
        while ((cInput = ReadByte()) != DLE) {    // espera DLE
            *pRxBuffer = cInput;          // insere caracter no buffer
            pRxBuffer++;
            nMsgLength++;
        } // end_while
        cInput = ReadByte();
        switch(cInput) {
            case ETX:
                nNextState = LE_CRC;
                break;
            case DLE:
                *pRxBuffer = cInput; // insere dado = DLE no buffer
                pRxBuffer++;
                nMsgLength++;
                nNextState = LE_DADOS;
                break;
            case STX:
                ZeroMemory(pRxBuffer, MAX_MSG_SIZE);
                nMsgLength = 0;
                nNextState = LE_DADOS;
                break;
            default:
                nNextState = INICIALIZA;
        }  // switch
        break;

    case LE_CRC:
        nCRCLow  = ReadByte();   // Le CRC low
        nCRCHigh = ReadByte();   // Le CRC High
        nCRC = CalcCRC(pRxBuffer, nMsgLength, CRC_16);
        nCRCrx = (nCRCHigh << 8) + nCRCLow;
        if (nCRCrx != nCRC)         // Erro de CRC
            nNextState = PROCESSA_MSG_NOK;
        else { // CRC_OK
```

```
                        if (nMsgLength == 1)
                                if (*pRxBuffer == ACK)
                                        nNextState = PROCESSA_ACK;
                                else if (*pRxBuffer == NACK)
                                        nNextState = PROCESSA_NACK;
                                        else nNextState = PROCESSA_MSG_OK;
                        else nNextState = PROCESSA_MSG_OK;
                        }  // else
                break;

        case PROCESSA_MSG_OK:
                // Transmite ACK
                PostThreadMessage(dwTxId,  WM_TXRX_MSG,  (WPARAM)ENVIAACK,
                (LPARAM)0);
                // Informa Transação OK para aplicação e retorna tamanho da mensagem
                PostThreadMessage(dwCommId, WM_TXRX_MSG,
                (WPARAM)MSG_RECEBIDA, (LPARAM)nMsgLength);
                printf("\nThreadRX: Msg recebida com sucesso: %d bytes", nMsgLength);
                nNextState = INICIALIZA;
                break;

        case PROCESSA_MSG_NOK:
                // Transmite NACK
                PostThreadMessage(dwTxId, WM_TXRX_MSG,
                (WPARAM)ENVIA_NACK, (LPARAM)0);
                // Informa Transação NOK para aplicação e retorna 0
                PostThreadMessage(dwCommId, WM_TXRX_MSG,
                (WPARAM)MSG_RECEBIDA, (LPARAM)0);
                printf("\nThreadRX: Erro na recepção de mensagem");
                nNextState = INICIALIZA;
                break;

        case PROCESSA_ACK:
                if (bEsperaAck) {
                        // Informa ThreadTx que ACK foi recebido
                        printf("\nThreadRX: ACK Recebido");
                        PostThreadMessage(dwTxId, WM_TXRX_MSG,
                        (WPARAM)ACK_RECEBIDO, (LPARAM)0);
                        nNextState = INICIALIZA; }
                        break;

        case PROCESSA_NACK:
                if (bEsperaAck) {
                        // Informa ThreadTx que NACK foi recebido
                        printf("\nThreadRX: NACK Recebido");
                        PostThreadMessage(dwTxId, WM_TXRX_MSG,
                        (WPARAM)NACK_RECEBIDO, (LPARAM)0);
                        nNextState = INICIALIZA;
                        }
                        break;
        default:break;
        } // switch
        nState = nNextState;
    } // while
    _endthreadex((DWORD) 0);

    return(0);
} // RxFrame
// TxFrame é uma thread independente que transmite frames de dados.
// Inputs:      RxFrame recebe da aplicação através de PostMessage, o endereço
```

```
//              do buffer de dados e o número de bytes a serem transmitidos
//              As mensagens da aplicação e da thread RX (pedido de transmissão
//              de ACK/NACK) ou aviso de recepção de ACK/NACK são inseridas na
//              fila de mensagem da thread.
//              O timeout na temporização de espera de ACK/NACK é sinalizado
//              através de um evento de reset manual.
// Outputs:      Status da transação: 0 se OK e -1 se houve problema.
//
DWORD WINAPI ThreadTxFrame(LPVOID n)
{
        int     nTentativas;  // Conta as tentativas de retransmissão
        int     nCRC;         // CRC calculado
        char    FrameBuffer[MAX_MSG_SIZE+6];  // Buffer para formação do quadro a transmitir
        char    *pchar;
        DWORD       dwRet;
        int     cont;
        int     nMsgLength;
        MSG     msg;
        UINT    MsgId;              // Tipo da Mensagem Windows recebida: WM_XX
        DWORD       dwFunction;
        char    *pBuffer;
        MsgData     *pMsgData;
        BOOL bRet;

        // Programação do Timer
        LARGE_INTEGER Preset;
        HANDLE hTimer;                  // Handle para Timer
        // Define uma constante para acelerar cálculo do atraso e período
        const int nMultiplicadorParaMs = 10000;
        // Programa o temporizador para que a primeira sinalização ocorra 2s
        // depois de SetWaitableTimer
        // Use - para tempo relativo
        Preset.QuadPart = -(2000 * nMultiplicadorParaMs);

        // Cria fila de mensagens
        PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);
        bRet = SetEvent(hEventTx);       // Thread de transmissão pronta
        CheckForError(bRet);

        // Define timer para gerar evento a cada 1s
        // Cria timer com reset automático
        hTimer = CreateWaitableTimer(NULL, FALSE, "MyTimer");
        CheckForError(hTimer);

        for (; ; ) {
        // Recebe evento de tempo ou mensagem com solicitação de transmissão
        dwRet    =    MsgWaitForMultipleObjects(1,    &hTimer,    FALSE,    INFINITE,
        QS_POSTMESSAGE);
        if (dwRet == WAIT_OBJECT_0)  // foi o Timer
                dwFunction = TIMEOUT_ACK_NACK;  // Timeout na espera de ACK/NACK
                else { // chegou uma mensagem
                        PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
                        MsgId=              (UINT)    ((msg).message);
                        dwFunction= (DWORD)   ((msg).wParam);
                }

                if (MsgId == WM_QUIT) break; // WM_QUIT: Abort Thread

                switch (dwFunction) {
```

```
case ENVIA_QUADRO:
if (bEsperaAck) { // Não aceita outra mensagem se houver mensagem pendente
                printf("\nTX: Existe mensagem pendente");
                FormatAndPostMsg(dwCommId, MSG_NAO_ACEITA, 0, 0);
                break;
        }

        printf("\nTX  Envia Quadro");
        pMsgData=   (MsgData *)((msg).lParam);
        pBuffer=        (char *) ((*pMsgData).lpBuffer);
        nMsgLength= (int )   ((*pMsgData).dwMsgLength);
        free(pMsgData);       // libera memória alocada
        printf("\nTX: MsgLength %d \n %s", nMsgLength, pBuffer);

        pchar = FrameBuffer;
        *(pchar++) = DLE;
        *(pchar++) = STX;
        cont = nMsgLength;
        while (cont--) *(pchar++) = *(pBuffer++);
        *(pchar++) = DLE;
        *(pchar++) = ETX;
        nCRC = CalcCRC(FrameBuffer, nMsgLength, CRC_16);
        *(pchar++) = nCRC & 0x00FF;
        *(pchar++) = nCRC >> 8;
        TxQuadro(FrameBuffer, nMsgLength + 6);

        nTentativas = 3;
        // Dispara temporizador 2s
        SetWaitableTimer(hTimer, (const LARGE_INTEGER *)&Preset, 0, NULL,
        NULL, FALSE);
        // Avisa thread de recepção que está esperando ACK
        bEsperaAck = TRUE;

        break;

case ENVIA_ACK:
        pchar = FrameBuffer;
        *(pchar++) = DLE;
        *(pchar++) = STX;
        *(pchar++) = ACK;
        *(pchar++) = DLE;
        *(pchar++) = ETX;
        nCRC = CalcCRC(FrameBuffer, nMsgLength, CRC_16);
        *(pchar++) = nCRC & 0x00FF;
        *(pchar++) = nCRC >> 8;
        TxQuadro(FrameBuffer, 7);
        break;

case ENVIA_NACK:
        pchar = FrameBuffer;
        *(pchar++) = DLE;
        *(pchar++) = STX;
        *(pchar++) = NACK;
        *(pchar++) = DLE;
        *(pchar++) = ETX;
        nCRC = CalcCRC(FrameBuffer, nMsgLength, CRC_16);
        *(pchar++) = nCRC & 0x00FF;
        *(pchar++) = nCRC >> 8;
        TxQuadro(FrameBuffer, 7);
        break;
```

```
			case TIMEOUT_ACK_NACK:
				printf("\nTX: timeout de comunicacao");
			case NACK_RECEBIDO:
				if (bEsperaAck)
				  if (nTentativas--) {  // Retransmite quadro
					TxQuadro(FrameBuffer, nMsgLength + 6);
					SetWaitableTimer(hTimer, (const LARGE_INTEGER *)&Preset, 0,
					NULL, NULL, FALSE);
				  }
				  else {
					ZeroMemory(FrameBuffer, MAX_MSG_SIZE);
					// Retorna fracasso
					PostThreadMessage(dwCommId, WM_TXRX_MSG,
					(WPARAM)MSG_TRANSMITIDA, (LPARAM)FRACASSO);
					bEsperaAck = FALSE;
					printf("\nTX: Transmissao de dados falhou");
				  }
				break;

			case ACK_RECEBIDO:
				if (bEsperaAck) {
					// Retorna sucesso
					PostThreadMessage(dwCommId, WM_TXRX_MSG,
					(WPARAM)MSG_TRANSMITIDA, (LPARAM)SUCESSO);
					bEsperaAck = FALSE;
					printf("\nThreadTX: Transmissao de dados OK");
				}
				break;

			default:  break;

			} // switch

	} // for

	_endthreadex((DWORD) 0);

	return(0);
} // TxFrame


unsigned CalcCRC(char *pch, int nBytes, unsigned Polinomio)
{
	unsigned CRC;
	int bit0;

	CRC = 0;			// Inicializa shift register para zero
	for (int cByte = nBytes; cByte >0; --cByte) {
		CRC ^= (*pch++ & 0x00FF);	// Assegura que trabalhará com byte
		for (int cBit=8; cBit >0; --cBit) {
			bit0 = 1 & CRC;
			CRC >>= 1;
			if (bit0 == 1) CRC ^= Polinomio;
		}
	}
	return (CRC);
}	// CalcCRC
```

```
void FormatAndPostMsg(DWORD dwThreadId, int Function, char *pBuffer, DWORD
dwLength)
{
	MsgData *pMsgData;		// Apontador para MsgData

	pMsgData = (MsgData *)calloc(1, sizeof(MsgData));
	// Não se esqueça de desalocar a memória após o uso
	(*pMsgData).lpBuffer =	pBuffer;
	(*pMsgData).dwMsgLength = dwLength;
	PostThreadMessage(dwThreadId, WM_TXRX_MSG, (WPARAM)Function,
	(LPARAM)pMsgData);
}  // FormatAndPostMsg

void TxQuadro(char *pBuffer, int nLenght)
{
	while (nLenght--)
		printf("%02x ", *pBuffer++); // Envia byte via linha serial
}	// TxQuadro


char ReadByte(void)
{
	char cInputByte;

	cInputByte = 0x31;   // Le um byte da linha serial
	return(cInputByte);
}
```

Exercícios

1) Discuta porque o protocolo Modbus permite apenas a comunicação síncrona.

2) Desenvolva os algoritmos *bitwise* e *bytewise* para o cálculo do CRC do protocolo Modbus, operando em modo RTU.

3) Desenvolva o algoritmo para cálculo do LRC para o protocolo Modbus, operando em modo ASCII.

4) Implemente o protocolo Modbus modo RTU no WNT. O PC deve funcionar como mestre de uma rede de controladores e aceitar comandos do usuário para realizar as operações básicas de leitura e escrita de bobinas e registradores.

5) Desenvolva um gateway que leia dados de uma rede Modbus e permita leitura assíncrona de dados sobre protocolo TCP/IP. Este gateway deve permitir também o envio de *unsolicited messages* para clientes na rede Ethernet.

6) Complete a implementação do protocolo derivado do BISYNC apresentado.

7) Escreva o fluxograma de recepção na forma de autômato de estados finitos. Escreva o fluxograma de transmissão na forma de um SFC (*Sequential Function Chart*).

8) Modifique o esboço de programa apresentado para a transmissão BISYNC, de tal forma a que as mensagens enviadas, quando a thread de transmissão está ocupada, sejam mantidas na fila de mensagem e processadas posteriormente.

9) Ainda considerando o último exemplo, modifique o programa para que apenas mensagens menores que o tamanho máximo indicado pelo programa principal seja recebidas.

10) Altere o programa para eliminar a morte da thread de recepção através da instrução TerminateThread.

11) Discuta uma forma alternativa de implementar a flag bEsperaAck.

Bibliografia

[Modicon 1993] Modicon Modbus Protocol Reference Guide, AEG, PI-MBUS-300 Rev. E, March 1993

[Souza 1999] Luiz Cláudio Andrade Souza, Desenvolvimento de um drive padrão OPC para Controlador Lógico Programável, em ambiente Windows NT, Dissertação de mestrado, UFMG, 1999.

[Mc Namara 88] John McNamara. Technical Aspects of Data Communication, 3rd edition, 1988, Digital Equipment Corporation.

[Allen Bradley] DF1 Protocol and Command Set - Reference Manual, October, 1996

Sites a visitar:

Protocolo Modbus - especificações	www.modicon.com/techpubs/toc7.html
Site da Schneider dedicado a Modbus	www.modbus.org
Site Lantronix	www.lantronix.com
Site Moxa	www.moxa.com
Melhor site Modbus : componentes, servidores para Windows, etc.	members.tripod.com/~mbserver/index.htm
Protocolo DF1	www.ab.com/manuals/cn/17706516.pdf