UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

PROTOCOLO DE COMUNICAÇÃO PARA UMA REDE DE SENSORES SEM FIO: TEORIA E IMPLEMENTAÇÃO EM SOC

GUSTAVO LUCHINE ISHIHARA

ORIENTADOR: ALEXANDRE RICARDO ROMARIZ

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.DM – 290/06
BRASÍLIA/DF: DEZEMBRO – 2006

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

PROTOCOLO DE COMUNICAÇÃO PARA UMA REDE DE SENSORES SEM FIO: TEORIA E IMPLEMENTAÇÃO EM SOC

GUSTAVO LUCHINE ISHIHARA

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

____

ALEXANDRE RICARDO SOARES ROMARIZ, Ph.D. (UnB)

(ORIENTADOR)

____

ADSON FERREIRA DA ROCHA, Ph.D. (UnB)

(EXAMINADOR INTERNO)

____

MARCO ANTÔNIO ASSFALK DE OLIVEIRA, Ph.D. (UFG)

(EXAMINADOR EXTERNO)

DATA: BRASÍLIA/DF, 15 de dezembro de 2006.

FICHA CATALOGRÁFICA

ISHIHARA, GUSTAVO LUCHINE

Protocolo de Comunicação para uma Rede de Sensores sem Fio: Teoria e Implementação em SoC [Distrito Federal] 2006.

xvii, 241 p., 297 mm (ENE / FT / UnB, Mestre, Engenharia Elétrica, 2006).

Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.

Departamento de Engenharia Elétrica.

1. Redes de Sensores	2. Protocolo de Comunicação
3. Programação Assembly	4. Microcontroladores
I. ENE / FT / UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ISHIHARA, G.L. (2006). Protocolo de Comunicação para uma Rede de Sensores sem Fio: Teoria e Implementação em SoC. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGENE.DM-290/06, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 241 p.

CESSÃO DE DIREITOS

AUTOR: Gustavo Luchine Ishihara

TÍTULO: Protocolo de Comunicação para uma Rede de Sensores sem Fio: Teoria e Implementação em SoC.

GRAU: Mestre ANO: 2006

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta dissertação de mestrado pode ser reproduzida sem a autorização por escrito do autor.

Gustavo Luchine Ishihara
SHIN QI 2 Cj.4 Cs.13
71.510-040 – Brasília – DF

DEDICATÓRIA

A Deus.

Aos meus queridos pais, Kogi Ishihara e Waldelice Luchine Ishihara, que sempre me ajudaram nesta caminhada para que eu pudesse chegar até aqui.

Aos meus irmãos, Daniel e Fernanda.

À minha namorada, Iná Caroline.

E aos irmãos que eu escolhi, meus amigos.

AGRADECIMENTOS

Agradeço ao meu orientador Alexandre Ricardo Soares Romariz, por ter me incentivado e depositado a sua confiança. Seus ensinamentos ficarão guardados, não apenas os que pude transferir para esta dissertação, mas também os que levo pra minha vida.

Agradeço ao professor José Camargo da Costa por se empenhar tanto em realizar o sonho da fabricação nacional de chips que possibilitou a concretização deste trabalho.

Agradeço ao meu amigo Regis Proença Picanço, pela co-autoria do artigo publicado na Croácia e pelas muitas explicações que me deu na época da graduação.

Agradeço ao meu amigo Alexandre de Araújo Martins, pelo incentivo e por toda a ajuda. Obrigado por ter sido co-autor do nosso projeto final de graduação. Sua participação foi fundamental naquela e nesta conquista. Começamos este mestrado juntos, mas sei que você não pôde chegar no fim porque havia coisas mais importantes a serem cuidadas, sua esposa e seus filhos.

Agradeço aos meus amigos que participaram diretamente desta conquista: Mário Viktor, Luiz Cláudio, Felipe (primo), Ubirajara Santos, Carolina Sampaio, Rafael Farias e Luiz Anísio.

Agradeço aos meus tios Rui, Pepenha (Mária), Yukie, Napoleão, Welber, Waltinho, Maria Laura e Saina (Orlinda) por todo o apoio que tive deles.

Agradeço ao meu chefe na Anatel, Jairo Antônio Karnas, por ter me proporcionado a oportunidade de ficar em uma função em que eu não viajasse tanto. Obrigado, também, aos meus colegas e amigos de trabalho por terem viajado em meu lugar. Agradeço meu ex-chefe Domingos Aló por ter permitido que eu pudesse ter iniciado esta jornada.

Agradeço também a todos os meus amigos que não pude citar, mas que sempre serão lembrados por terem contribuído com este objetivo que foi alcançado.

RESUMO

PROTOCOLO DE COMUNICAÇÃO PARA UMA REDE DE SENSORES SEM FIO: TEORIA E IMPLEMENTAÇÃO EM SOC

Autor: Gustavo Luchine Ishihara

Orientador: Alexandre Ricardo Romariz

Programa de Pós-graduação em Engenharia Elétrica

Brasília, dezembro de 2006.

Esta dissertação apresenta a especificação, verificação, implementação, testes práticos e análise de *performance* de um protocolo de comunicação desenvolvido a partir do trabalho de Castricini e Marinangeli para o protocolo SNAP.

A motivação deste trabalho nasceu da necessidade de comunicação de um novo microcontrolador que está sendo desenvolvido pela Universidade de Brasília, o RISC16. Para atingir esse objetivo, foi utilizada a técnica de descrição formal da máquina de estados finitos juntamente com a explicação do fluxo de funcionamento, apresentada a estrutura do pacote de dados e realizada a codificação que implementou as funcionalidades necessárias. Como as ferramentas para este novo microcontrolador ainda se encontravam em desenvolvimento, foi utilizada como estratégia, implementar primeiro o protocolo no microcontrolador PIC para depois migrar o código para o RISC16. Essa estratégia permitiu que fossem utilizadas as ferramentas do PIC, incluindo o seu simulador, que sanou muitos comportamentos errôneos difíceis de detectar apenas por análise do código. A tradução para o RISC16 foi tranqüila, necessitando apenas de pequenas modificações por causa das inerentes diferenças entre os microcontroladores.

Para efeito de validação do protocolo de comunicação, foram realizados testes de tráfego de pacotes, onde os resultados foram apresentados no decorrer desta dissertação.

ABSTRACT

COMMUNICATIONS PROTOCOL TO A SENSOR NETWORK: THEORY AND IMPLEMENTATION ON SOC

Author: Gustavo Luchine Ishihara

Supervisor: Alexandre Ricardo Romariz

Electrical Engineering Master Degree Program

Brasília, december of 2006.

This thesis presents the specification, verification, implementation, practical tests and performance analysis of a communication protocol which was developped starting from the work of Castricini and Marinangeli to the SNAP Protocol.

The motivation to this work came from the communication needs of a new microcontroller named RISC16 which is under development by the University of Brasilia. To fulfill this objective, the formal description technique of finite state machine was used jointly with a explanation of the working flux, a presentation of the packet structure and the codes that implemented all the functionalities.

Because all the tools for this new microcontroller were under development, a strategy was formulated. First it was implemented the communication protocol for the PIC microcontroller, and then all the generated code was translated to the RISC16. That strategy made possible the use of all mature tools to the PIC, including its simulator which helped to solve several very hard to find errors that were not visible by only inspecting the code. The translation process to the RISC16 was accomplished with only small changes in the code due to the inherent differences between the two microcontrollers.

SUMÁRIO

1 – INTRODUÇÃO 1

1.1 - OBJETIVOS 1

1.2 - PROTOCOLOS DE COMUNICAÇÃO 1

1.3 - LINGUAGEM ASSEMBLY 4

1.4 - MICROCONTROLADORES 4

1.5 - O MICROCONTROLADOR RISC16 5

1.6 - TERMOS E DEFINIÇÕES 6

1.7 - DIAGRAMAÇÃO DESTA DISSERTAÇÃO 6

2 – REDES DE SENSORES 7

2.1 - HISTÓRICO 7

2.2 - REDES DE SENSORES EM CAMPO 8

2.3 - DESAFIOS E LIMITAÇÕES 9

2.4 - PROJETO DO SISTEMA DE CONTROLE DE IRRIGAÇÃO - SCI 11

2.4.1 - Nó Sensor 14

3 – O PROTOCOLO NO PIC 16

3.1 - CONSIDERAÇÕES INICIAIS 16

3.2 - PREMISSAS BÁSICAS 16

3.3 - METODOLOGIA DE IMPLEMENTAÇÃO 17

3.4 - IMPLEMENTAÇÃO NO PIC 18

3.4.1 - Organização e estrutura dos pacotes 20

3.4.2 - Fluxo do protocolo 23

3.4.3 - Implementação das funções envolvidas 42

3.4.3.1 - Função TX_SNAP 47

3.4.3.2 - Uso de Registradores Temporários 52

3.5 – CONSIDERAÇÕES FINAIS 55

4 – TRADUÇÃO DO PROTOCOLO PARA O RISC16 56

4.1 – CONSIDERAÇÕES INICIAIS 56

4.2 – ORGANIZAÇÃO E ESTRUTURA DOS PACOTES 57

4.3 – FLUXO DO PROTOCOLO NO RISC16 59

4.4 – IMPLEMENTAÇÃO NO RISC16 60

4.4.1 - Função TX_SNAP 67

4.4.2 - Função TEMPO 75
4.5 – CONSIDERAÇÕES FINAIS 78
5 – TESTES DE TRÁFEGO 79
5.1 – CONSIDERAÇÕES INICIAIS 79
5.2 – PRIMEIRA ARQUITETURA – DIRIGIDA POR EVENTOS 80
5.3 – SEGUNDA ARQUITETURA – RELAÇÃO MESTRE/ESCRAVO 86
6 – CONCLUSÃO 88
REFERÊNCIAS BIBLIOGRÁFICAS 93
A – CÓDIGO COMPLETO PARA O PIC 98
A.1 - FUNÇÃO TX_SNAP 98
A.2 - FUNÇÃO SEND_SNAP 103
A.3 - FUNÇÃO SEND_WORD 104
A.4 - FUNÇÃO SEND_BYTE 105
A.5 - FUNÇÃO RX_SNAP 106
A.5 - FUNÇÃO RECEIVE_SNAP 110
A.6 - FUNÇÃO RECEIVE_WORD 114
A.7 – FUNÇÃO RECEIVE_BYTE 115
A.8 - FUNÇÃO COMANDO_EXEC 117
A.9 - FUNÇÃO APPLY_EDM 119
A.10 - FUNÇÃO EDM_BYTE 122
B – FUNÇÕES DO PROTOCOLO PARA O RISC16 124
B.1 - FUNÇÃO TX_SNAP 124
B.2 - FUNÇÃO SEND_SNAP 131
B.3 - FUNÇÃO SEND_WORD 133
B.4 - FUNÇÃO SEND_BYTE 134
B.5 - FUNÇÃO RX_SNAP 139
B.6 - FUNÇÃO RECEIVE_SNAP 145
B.7 - FUNÇÃO RECEIVE_WORD 151
B.8 - FUNÇÃO RECEIVE_BYTE 152
B.9 - FUNÇÃO COMANDO_EXEC 158
B.10 - FUNÇÃO APPLY_EDM 161
B.11 - FUNÇÃO EDM_BYTE 164
B.12 - FUNÇÃO TEMPO 169

C – CÓDIGO COMPLETO PARA O PIC 173
C.1 - LINGUAGEM ASSEMBLY DO PROTOCOLO 173
C.2 - PROGRAMA PRINCIPAL DO SIMULADOR DE TRÁFEGO 196
D – MÁQUINAS DE ESTADOS FINITOS 199
E – MAPAS DE REGISTRADORES 202
F – CÓDIGO COMPLETO PARA O RISC16 204
F.1 - ARQUIVO SNAP16.ASM 204
F.2 - ARQUIVO SNAP16.INC 239
F.3 - ARQUIVO APPL16.INC 241

LISTA DE FIGURAS

Figura 2.1: *Yosemite National Park* – derretimento da neve nas montanhas 8

Figura 2.2: fatores que afetam o tempo de vida de um nó... 11

Figura 2.3: sensoriamento de uma cultura através de redes de sensores sem fio 12

Figura 2.4: diagrama de blocos de um nó sensor 14

Figura 2.5: o microcontrolador RISC16 (fonte [26] (2003))... 14

Figura 2.6: o nó sensor parcialmente montado (à esq.) e o tensiômetro (à dir.) 15

Figura 3.1: pacote de dados do SNAP Modificado 20

Figura 3.2: *byte* que inicia todo pacote de dados e de comando 20

Figura 3.3: primeira parte do cabeçalho do pacote.. 21

Figura 3.4: interpretação dos *bits* de ACK do *byte* HDB2 21

Figura 3.5: segunda parte do cabeçalho do pacote 22

Figura 3.6: endereço de destino do pacote 22

Figura 3.7: endereço de origem do pacote... 22

Figura 3.8: *byte* de dados (DBi, 1<=i<=8) ou comando... 22

Figura 3.9: *byte* de detecção de erro (CRC-8) 23

Figura 3.10: fluxo simplificado do protocolo SNAP Modificado 24

Figura 3.11: diagramas de tempo detalhando o funcionamento do *acknowledge* (ACK)... 25

Figura 3.12: fluxo do protocolo SNAP Modificado considerando a função APPLY_EDM 27

Figura 3.13: Máquina de Estados do Protocolo SNAP Modificado................................... 29

Figura 3.14: ciclo de transmissão de pacotes do protocolo SNAP Modificado 30

Figura 3.15: leituras sendo realizadas no *byte* de sincronismo (SYNC) 34

Figura 3.16: ciclo de recepção de pacotes do protocolo SNAP Modificado....................... 37

Figura 3.17: Inicio do arquivo SNAP.INC contendo o mapa de memória......................... 43

Figura 3.18: trecho do arquivo SNAP.INC contendo o mapa de memória 44

Figura 3.19: trecho do arquivo SNAP.INC contendo outras definições 44

Figura 3.20: trecho do arquivo SNAP.INC contendo as definições dos *bits* de controle.... 45

Figura 3.21: trecho do arquivo SNAP.INC contendo as definições de interface 46

Figura 3.22: trecho do arquivo APPL.INC contendo o endereçamento dos nós................. 47

Figura 3.23: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP........ 47

Figura 3.24: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP........ 48

Figura 3.25: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP............ 49

Figura 3.26: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP............ 49

Figura 3.27: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP 50
Figura 3.28: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP 51
Figura 3.29: trecho do arquivo SNAP.ASM contendo o final da função TX_SNAP 52
Figura 3.30: mapa de registradores em função do fluxo do protocolo na transmissão 53
Figura 3.31: mapa de registradores em função do fluxo do protocolo na recepção 54
Figura 4.1: pacote de dados do SNAP Modificado no RISC16 57
Figura 4.2: *byte* de cabeçalho do pacote no RISC16... 58
Figura 4.3: endereço de destino do pacote no RISC16 58
Figura 4.4: endereço de origem do pacote no RISC16.. 59
Figura 4.5: *byte* de dados (DBi, 1<=i<=8) ou comando no RISC16.................................. 59
Figura 4.6: *byte* de CRC-16 no RISC16.. 59
Figura 4.7: trecho inicial do arquivo SNAP16.INC 60
Figura 4.8: trecho do arquivo SNAP16.INC que define os *bits* de controle do protocolo.. 61
Figura 4.9: trecho do SNAP16.INC que mapeia os *bytes* do pacote na memória............... 62
Figura 4.10: início do trecho do SNAP16.INC que define as constantes do protocolo 62
Figura 4.11: trecho do SNAP16.INC que define os endereços da interface serial............. 63
Figura 4.12: trecho do SNAP16.INC que define os endereços da interface de RF............. 63
Figura 4.13: trecho do SNAP16.INC que define a taxa de 9600 bps para comunicação.... 64
Figura 4.14: trecho do APPL16.INC que define os endereços dos nós 65
Figura 4.15: trecho inicial do SNAP16.ASM que mostra a organização da memória........ 66
Figura 4.16: mapa geral de memória do microcontrolador RISC16 66
Figura 4.17: trecho da função TX_SNAP que desabilita as interrupções do RISC16 68
Figura 4.18: trecho da TX_SNAP que configura os controles RUN_TX e RUN_RTX..... 68
Figura 4.19: trecho da TX_SNAP que configura os *bits* de *acknowledge* 69
Figura 4.20: trecho da TX_SNAP que verifica se é o primeiro pacote de resposta............ 70
Figura 4.21: trecho da TX_SNAP que completa o pacote e o transmite............................ 71
Figura 4.22: trecho da TX_SNAP que aguarda pelo pacote de resposta 72
Figura 4.23: trecho da TX_SNAP que verifica o pacote de resposta recebido 73
Figura 4.24: trecho final da função TX_SNAP.. 74
Figura 4.25: função TEMPO completa que mostra o *loop* de espera com 10 ciclos 75
Figura 4.26: trecho do arquivo SNAP16.INC que é configurado em função do *clock* 77
Figura 5.1: Gerador de Números Aleatórios 81
Figura 5.2: forma de onda do Gerador de Números Aleatórios vista no osciloscópio........ 82
Figura 5.3: código do aplicativo principal para a geração de tráfego na rede.................... 84

Figura 5.4: efeito do aumento do número de nós 85
Figura 5.5: efeito do aumento do tamanho do pacote... 86
Figura A.1: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP......... 98
Figura A.2: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP......... 99
Figura A.3: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP............. 99
Figura A.4: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP........... 100
Figura A.5: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP........... 101
Figura A.6: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP........... 102
Figura A.7: trecho do arquivo SNAP.ASM contendo o final da função TX_SNAP 102
Figura A.8: trecho do arquivo SNAP.ASM contendo parte da função SEND_SNAP...... 103
Figura A.9: parte da função SEND_SNAP que prepara o envio de vários *bytes*............. 103
Figura A.10: parte da função SEND_SNAP que guarda o endereço de destino............... 104
Figura A.11: trecho da função SEND_WORD que ordena os *bytes* para o envio 104
Figura A.12: trecho da função SEND_BYTE que efetivamente transmite os *bits*............ 105
Figura A.13: trecho da função SEND_BYTE que temporiza e decrementa o contador ... 106
Figura A.14: trecho do arquivo SNAP.ASM contendo o inicio da função RX_SNAP 106
Figura A.15: trecho da função RX_SNAP com ênfase no método de detecção de erros.. 107
Figura A.16: parte da função RX_SNAP com ênfase no tratamento do pacote recebido. 108
Figura A.17: trecho da função RX_SNAP com ênfase no reconhecimento de *broadcast* 109
Figura A.18: trecho da função RX_SNAP com ênfase na espera de *broadcast* 109
Figura A.19: trecho do arquivo SNAP.ASM contendo o final da função RX_SNAP 110
Figura A.20: trecho do SNAP.ASM contendo o inicio da função RECEIVE_SNAP 111
Figura A.21: parte da função RECEIVE_SNAP que verifica se o pacote é de *broadcast* 111
Figura A.22: trecho da função RECEIVE_SNAP que verifica se o pacote é deste nó 112
Figura A.23: trecho que verifica se existem dois nós com o mesmo endereço................ 112
Figura A.24: verificação se o SAB1 do pacote recebido é igual ao DAB1 anterior 113
Figura A.25: parte da função RECEIVE_SNAP que recebe CRC1 e configura o *bit* REJ114
Figura A.26: trecho da função RECEIVE_WORD que recebe o número de *bytes* de W. 114
Figura A.27: trecho da função RECEIVE_BYTE que espera pelo *start bit* 115
Figura A.28: trecho da RECEIVE_BYTE que recebe os *bits* na interface de entrada...... 116
Figura A.29: trecho da COMANDO_EXEC que verifica se o pacote é um comando...... 117
Figura A.30: trecho da função COMANDO_EXEC que exemplifica um comando......... 118
Figura A.31: trecho final da função COMANDO_EXEC de exemplo 119
Figura A.32: trecho inicial da função APPLY_EDM.. 120

Figura A.33: trecho da função APPLY_EDM que demonstra a tomada de decisão......... 120
Figura A.34: trecho da função APPLY_EDM que aplica o resultado do CRC calculado 121
Figura A.35: trecho da função APPLY_EDM que compara o CRC-8 com o *byte* CRC1 122
Figura A.36: função EDM_BYTE que calcula a detecção de erro para cada *byte* 123
Figura B.1: trecho da função TX_SNAP que desabilita as interrupções do RISC16 125
Figura B.2: trecho da TX_SNAP que configura os controles RUN_TX e RUN_RTX 125
Figura B.3: trecho da TX_SNAP que configura os *bits* de *acknowledge* 126
Figura B.4: trecho da TX_SNAP que verifica se é o primeiro pacote de resposta 127
Figura B.5: trecho da TX_SNAP que completa o pacote e o transmite........................... 128
Figura B.6: trecho da TX_SNAP que aguarda pelo pacote de resposta........................... 129
Figura B.7: trecho da TX_SNAP que verifica o pacote de resposta recebido 130
Figura B.8: trecho final da função TX_SNAP 131
Figura B.9: trecho da função SEND_SNAP que envia o octeto de sincronismo............. 132
Figura B.10: trecho da SEND_SNAP que guarda o *byte* de endereço DAB1 do pacote.. 132
Figura B.11: trecho da SEND_SNAP que mostra o envio de vários *bytes*...................... 133
Figura B.12: trecho inicial da função SEND_WORD 133
Figura B.13: trecho da função SEND_WORD que contém o *loop* de envio de *bytes* 134
Figura B.14: trecho da SEND_BYTE que verifica se a transmissão é serial ou RF......... 135
Figura B.15: trecho da SEND_BYTE que inicia a transmissão pela interface serial 135
Figura B.16: trecho da SEND_BYTE que mostra o *loop* de envio pela interface serial .. 136
Figura B.17: trecho da SEND_BYTE que decide sobre a transmissão do *byte* de SYNC 136
Figura B.18: parte da SEND_BYTE que decide sobre a transmissão do segundo octeto 137
Figura B.19: trecho da SEND_BYTE que inicia a comunicação pela interface RF......... 137
Figura B.20: trecho da SEND_BYTE que cuida da condição do octeto de sincronismo . 138
Figura B.21: trecho da SEND_BYTE que mostra o *loop* de transmissão de RF............. 139
Figura B.22: trecho inicial da função RX_SNAP 140
Figura B.23: trecho da função RX_SNAP que verifica a validade do pacote.................. 141
Figura B.24: trecho da função RX_SNAP que verifica os *bytes* ERR e RUN_TX 141
Figura B.25: trecho da RX_SNAP que limpa SAB_PARA_DAB e executa comando.... 142
Figura B.26: trecho da função RX_SNAP que verifica a requisição do *acknowledge* 143
Figura B.27: trecho da função RX_SNAP que verifica se o pacote é de *broadcast*......... 143
Figura B.28: trecho da função RX_SNAP que mostra o *loop* de espera de *broadcast*..... 144
Figura B.29: trecho final da função RX_SNAP 145
Figura B.30: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1.................... 145

Figura B.31: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1 146
Figura B.32: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1 147
Figura B.33: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1 147
Figura B.34: trecho da função RECEIVE_SNAP que verifica o endereço de destino 148
Figura B.35: parte da função RECEIVE_SNAP que verifica a duplicidade de endereços 148
Figura B.36: parte da RECEIVE_SNAP que verifica se o nó certo enviou a resposta 149
Figura B.37: trecho da RECEIVE_SNAP que recebe os *bytes* de dados do pacote.......... 150
Figura B.38: trecho da RECEIVE_SNAP que recebe o *byte* da detecção de erro 150
Figura B.39: trecho da RECEIVE_SNAP que trata da rejeição do pacote 150
Figura B.40: trecho inicial da função RECEIVE_WORD 151
Figura B.41: trecho da função RECEIVE_WORD que evidencia o *loop* de recepção 152
Figura B.42: trecho inicial da função RECEIVE_BYTE 152
Figura B.43: trecho da RECEIVE_BYTE que inicializa a recepção pela interface serial 153
Figura B.44: trecho da RECEIVE_BYTE que mostra o *loop* de espera do *start bit*......... 154
Figura B.45: trecho da RECEIVE_BYTE que espera o tempo de um *bit* e meio 154
Figura B.46: trecho da RECEIVE_BYTE que recebe os *bits* pela interface serial 155
Figura B.47: trecho da RECEIVE_BYTE que verifica se é o octeto de sincronismo....... 155
Figura B.48: trecho da RECEIVE_BYTE que verifica qual foi o octeto recebido do *byte* 156
Figura B.49: trecho da RECEIVE_BYTE que inicia a recepção pela interface RF.......... 156
Figura B.50: trecho da RECEIVE_BYTE que verifica se é o octeto de sincronismo....... 157
Figura B.51: trecho da RECEIVE_BYTE que grava valor positivo no TX_RF.............. 157
Figura B.52: trecho da RECEIVE_BYTE que espera o tempo de meio *bit* 158
Figura B.53: trecho da RECEIVE_BYTE que mostra o *loop* de recepção pela RF.......... 158
Figura B.54: trecho da COMANDO_EXEC que verifica se o pacote contém um comando 159
Figura B.55: parte da COMANDO_EXEC que verifica qual comando o pacote contém 160
Figura B.56: trecho da COMANDO_EXEC que mostra como é o último comando da lista 160
Figura B.57: trecho da APPLY_EDM que inicia o método de detecção de erro 161
Figura B.58: trecho da APPLY_EDM que aplica o EDM no HDB1 do pacote recebido. 162
Figura B.59: trecho da APPLY_EDM que mostra o *loop* de aplicação do EDM nos DBi 162
Figura B.60: trecho da APPLY_EDM que mostra o processo de decisão da função........ 163
Figura B.61: trecho da APPLY_EDM que mostra o CRC calculado sendo transferido ... 164

Figura B.62: trecho da APPLY_EDM que compara os CRCs e configura o *byte* ERR ... 164
Figura B.63: função EDM_BYTE que calcula o método de detecção de erro para cada *byte* 169
Figura B.64: função TEMPO completa que mostra o *loop* de espera com 10 ciclos........ 170
Figura B.65: trecho do arquivo SNAP16.INC que é configurado em função do *clock* 171
Figura D.1: máquina de estados do protocolo SNAP Modificado.................................. 199
Figura D.2: máquina de estados do protocolo na transmissão 200
Figura D.3: máquina de estados do protocolo na recepção... 201
Figura E.1: mapa de registradores da transmissão 202
Figura E.2: mapa de registradores da recepção.. 203

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

Água capilar	- Parte da água infiltrada que é armazenada nos poros do solo
CHIP	- Pastilha de silício com circuitos miniaturizados (circuito integrado)
GNA	- Gerador de Números Aleatórios
IDE	- Ambiente de desenvolvimento integrado
Middleware	- Programa que pertence a uma camada intermediária de aplicação
Potencial mátrico	- Energia com que a água capilar é retida por forças superficiais
SoC	- Sistema em circuito integrado (*System on Chip*)
TDF	- Técnica de Descrição Formal

1 – INTRODUÇÃO

1.1 - OBJETIVOS

O presente trabalho tem como principal objetivo fornecer um *software* que funcione como um protocolo de comunicação para o circuito integrado que a Universidade de Brasília está desenvolvendo para sistemas embarcados. Este protocolo foi implementado de forma que possa ser utilizado como uma biblioteca de apoio para as funções de comunicação desse *chip*. A motivação desta dissertação é não apenas cobrir a aplicação desejada, mas que o protocolo de comunicação desenvolvido sirva de *Middleware* para outras aplicações que envolvam o uso do *chip* da Universidade de Brasília em redes de sensores.

1.2 - PROTOCOLOS DE COMUNICAÇÃO

Desde o início dos sistemas digitais, sempre houve uma grande preocupação na correta passagem da informação dentro de um circuito. Se nos sistemas analógicos, as pequenas distorções dos componentes internos geravam apenas conseqüências pouco perceptíveis, como os ruídos sonoros de um disco de vinil, nos sistemas digitais, a informação de um único *bit* incorreto poderia levar todo o circuito para um funcionamento catastrófico. Não se discute a qualidade adquirida com o uso da tecnologia digital, mas sim o aumento da responsabilidade no tratamento e no trânsito da informação dentro de um circuito digital.

Dentro de um mesmo dispositivo, o trânsito da informação é facilitado pela pequena distância entre os circuitos. Barramentos de dados e trilhas em circuito impresso são suficientes para garantir a integridade e confiabilidade das informações que transitam internamente. Entretanto, a crescente necessidade de interligação dos dispositivos eletrônicos levou ao desenvolvimento dos protocolos de comunicação, pois as distâncias entre esses dispositivos passaram a ser críticas, uma vez que os dados passaram a sofrer fortes interferências de fontes externas e atenuações do meio de transmissão.

Segundo Falbriard [1], os protocolos de comunicação definem conjuntos de regras que coordenam e asseguram o transporte de informações úteis entre dois ou mais dispositivos. Nessa tarefa, os protocolos estabelecem regras e métodos de funcionamento, negociando como as informações devem trafegar. Falbriard [1] também acrescenta que os protocolos

tratam de quais sinais devem ser enviados, como fazer o endereçamento das mensagens, quando pode ocorrer o envio das mensagens, quais mensagens podem ser enviadas, qual o significado de uma mensagem e como estabelecer uma conexão.

Held [2] divide o protocolo de comunicação em elementos básicos: um conjunto de símbolos chamado de conjunto de caracteres, um conjunto de regras para definir a seqüência e o sincronismo de mensagens construídas a partir do conjunto de caracteres, e ainda, procedimentos para detectar a ocorrência de um erro na transmissão e informações sobre como corrigir um erro. Held [2] também definiu quatro elementos da comunicação útil: uma mensagem a ser comunicada, um remetente, um meio ou canal sobre o qual a mensagem vai ser enviada e um destinatário. Outros autores enfocam as características funcionais, como o Lai [3] que define as funções básicas sendo: conexão, desconexão, endereçamento, controle de erro, controle de fluxo e sincronização. Enfim, o assunto traz uma riqueza incrível de detalhes e não existe uma fórmula pronta para se descrever um protocolo de comunicação, cada autor descreve seu protocolo de forma diferente e todas as formas de descrição são válidas para tentar cobrir as possibilidades de uso. Nesta dissertação, a forma para descrever o protocolo em questão foi apresentar inicialmente a estrutura do pacote, os diagramas de tempo do ponto de vista de trocas de mensagens, o fluxo de funcionamento da máquina de estados, e finalmente, a codificação que implementou as funções do protocolo em linguagem de máquina.

Na verdade, os protocolos de comunicação são bastante complexos porque envolvem a descrição de um fluxo de controle e um fluxo de dados que atuam em sincronia para que a informação, ou entropia de acordo com Claude Shannon[1], possa ser comunicada de um dispositivo para o outro. O desenvolvimento de novos protocolos de comunicação é uma tarefa desafiante, pois o bom comportamento dos protocolos em redes com poucos nós pode adquirir características não previstas quando o número de nós aumenta muito. A característica do tráfego na rede pode se alterar completamente quando alguns parâmetros do protocolo são alterados.

Segundo Lai [3], desenvolver um novo protocolo de comunicação requer um entendimento profundo de todos os elementos do sistema: serviços, funções, interações e procedimentos

[1] Claude Shannon [4] discutiu a incerteza ou desordem de um sistema, a qual ele chamou de entropia.

escritos em linguagens legíveis para humanos. Ele cita a descrição formal como uma representação simbólica de certos objetos em uma dada linguagem. Essa linguagem pode usar diversos tipos de símbolos, tanto textuais como gráficos. Lai [3] coloca que as principais funções de uma descrição formal são: ter uma especificação não ambígua do protocolo; em conjunto com uma especificação informal de suporte, para prover um entendimento completo do sistema; e permitir que o sistema possa ser analisado a procura de problemas. Com isso, Lai [3] insiste que para cumprir essas tarefas foram elaboradas diversas técnicas de descrição formal (TDF). Ele cita que uma dessas técnicas é a construção de uma máquina de estados do protocolo. Outras técnicas são baseadas em linguagem de desenvolvimento: Estelle (ISO 9074) [5] que é baseado na Máquina de Estados Finitos Estendida; LOTOS (ISO 8807) [6] que é baseado nos Cálculos de Sistemas de Comunicação de Milner (1980) [7]; e SDL (*Specification and Description Language*) desenvolvida pela ITU (ITU Z.100) [8]. Como não foi possível obter essas ferramentas em *software* para descrever o protocolo, esta dissertação utilizou a primeira técnica citada, a máquina de estados, juntamente com as explicações passo a passo de como o fluxo de controle do protocolo funciona. Lai [3] explica que Máquinas de Estados Finitos é um modelo de transição motivado pela observação que os protocolos consistem de processamentos simples em resposta a numerosos eventos como comandos, chegada de mensagens de camadas mais baixas e estouro de intervalos internos.

A preocupação de se utilizar uma TDF para especificar um protocolo de comunicação advém da necessidade de se ter uma base de progressão desde os requerimentos iniciais do protocolo até a sua implementação final. O desenvolvimento dessa metodologia sistemática é chamado de Engenharia de Protocolo (*Protocol Engineering*), que foi apresentada por Sarikaya [9]. A Engenharia de Protocolo pode ser considerada uma subdivisão da Engenharia de *Software*, pois se preocupa com as atividades envolvendo o *design* e a implementação rigorosos de um protocolo usando uma TDF durante os seus estágios de desenvolvimento. Se uma TDF é dada e conhecida, então essa TDF pode ser usada para as seguintes atividades: Especificação; Verificação (processo de checar a presença de erros na especificação que poderia levar a estados impróprios; Corretude (*Correctness*) (significa que o protocolo age de acordo com a especificação); Consistência (significa que seu comportamento pode ser determinado); Implementação (se preocupa com a realização das especificações de um protocolo em um sistema físico); Testes do Protocolo (se preocupa se um dado protocolo implementado se comporta de acordo com o que foi especificado e suas

regras, de forma que a especificação pode então ser usada como base para criar uma rotina de testes que possa detectar algum desvio da especificação); e Análise de *Performance* (que determina quão rápido e quão confiável a transferência da informação pode ser provida pelo protocolo em operação).

Existe um documento denominado ISO/IEC TR 10167 [10] que introduz uma base de comparação entre as técnicas de descrições formais (TDFs) e cita que os principais objetivos de uma TDF são: não permitir ambigüidade nas descrições que devem ser limpas e precisas; permitir estabelecer fundamentos de análise da especificação; prover ajuda para *designers* e programadores com o que deve ser implementado, mas não como implementar, então a técnica em si não limita como a implementação deverá ser feita; e servir de base para os testes de conformidade. Nesta dissertação todos esses objetivos foram cumpridos pela TDF da máquina de estados do protocolo nos capítulos 3 e 4, sendo que os testes de conformidade são apresentados no capítulo 5.

1.3 - LINGUAGEM ASSEMBLY

A linguagem *assembly* é uma notação legível por humanos para o código de máquina que um microprocessador utiliza para executar suas funções. Ao contrário das linguagens de alto nível, a linguagem *assembly* é característica do microprocessador. No entanto, desde que o conjunto de instruções de um microprocessador seja suficientemente completo, não existem problemas que impeçam a tradução de um programa em linguagem *assembly* de um microprocessador para a linguagem *assembly* de outro microprocessador. O capítulo 4 trata desse assunto, comprovando passo a passo que todas as funcionalidades requeridas pelo *software* do protocolo de comunicação do PIC puderam ser plenamente traduzidas para o microcontrolador da UnB.

1.4 - MICROCONTROLADORES

Com o advento do contínuo aumento de complexidade dos circuitos integrados, os fabricantes de tecnologia puderam desenvolver um circuito integrado cuja função seria determinada por um programa, e ele foi batizado de microprocessador. Segundo Matic [11], o primeiro circuito integrado programável que se tem notícia foi o Intel 4004, tendo sido um microprocessador de 4 *bits* e que podia processar 6000 operações por segundo. Depois disso, o mundo acordou para a enorme importância dos microprocessadores, e

desde então, novos microprocessadores mais modernos são lançados em períodos de tempo cada vez menores.

Entretanto, todo microprocessador precisa de componentes externos para funcionar: interfaces de entrada e saída, memória, osciladores e outros periféricos. Os fabricantes perceberam que havia muitas aplicações em que a miniaturização era um fator crítico. Então, eles resolveram criar microprocessadores com todos os componentes necessários em uma única pastilha de silício, surgindo os microcontroladores. Desde a sua invenção até os dias atuais, os microcontroladores têm crescido de popularidade, podendo ser encontrados em máquinas de lavar, fornos de microondas, brinquedos e outros equipamentos eletrônicos. O uso se tornou tão abundante que os microcontroladores são os circuitos integrados mais vendidos atualmente.

Nesta dissertação são utilizados microcontroladores do tipo RISC (*Reduced Instruction Set Computer*) que significa que eles dispõem de uma quantidade reduzida de instruções simples, por meio das quais é possível compor funcionalidades complexas. Os microcontroladores RISC permitem que seu projeto de hardware seja mais simples e mais rápido, mas comprometem um pouco o trabalho do programador de *assembly* porque tornam mais difíceis as implementações das funcionalidades do *software* quando o número de instruções é muito reduzido. Alguns algoritmos exigem verdadeiras manobras de programação em *assembly* porque, muitas vezes, as instruções disponíveis não são eficientes para realizar as funcionalidades requeridas pela tarefa. Inclusive, no início desta dissertação, uma das dúvidas sobre o protocolo de comunicação era sobre a viabilidade técnica de se implementar todas as funcionalidades necessárias com apenas as 16 instruções disponíveis no microcontrolador da UnB, o RISC16.

1.5 - O MICROCONTROLADOR RISC16

O microcontrolador utilizado neste trabalho é originário do projeto do Benício [12] e da implementação CMOS da Costa [13], tendo sido feito em conjunto com o Linder [14] que descreveu a linguagem *assembly* estendendo as 16 instruções do RISC16 para mais 60 pseudoinstruções. O programa montador deste microcontrolador, responsável pela tradução da linguagem *assembly* para a linguagem de máquina, foi realizado pelo Rangel [15].

1.6 - TERMOS E DEFINIÇÕES

Outro ponto a ser abordado é que o termo *byte* se refere a menor unidade endereçável em um sistema computacional. Antigamente, alguns computadores usavam *bytes* de seis, sete ou nove *bits*. O microcontrolador PIC utilizado tem *bytes* de oito *bits*, sendo mais adequado que eles sejam chamados de octetos. Já o RISC16 possui *bytes* de dezesseis *bits*, não devendo causar estranheza quando o capítulo 4 estiver sendo lido.

Conforme foi colocado por Hallberg [16], um pacote é qualquer conjunto de dados enviados em uma rede, e o termo é normalmente usado genericamente para descrever unidades de dados enviadas em qualquer camada do modelo OSI. A definição mais formal de pacote deveria ser aplicada somente para mensagens enviadas da camada mais alta do modelo OSI, a camada de aplicação. Unidades de dados da camada de rede são chamados de datagramas, enquanto unidades de dados carregados pela camada de enlace são chamados de *frames* (quadros). Mesmo que o termo correto seja quadros, nesta dissertação o termo pacote é usado genericamente.

1.7 - DIAGRAMAÇÃO DESTA DISSERTAÇÃO

A dissertação foi dividida em 6 capítulos. O capítulo 1 pretende introduzir os assuntos, conceitos, termos e definições utilizados no decorrer deste trabalho. No capítulo 2 são apresentadas as redes de sensores e alguns usos dessa nova tecnologia. O capítulo 3 é dedicado ao desenvolvimento e aperfeiçoamento de um protocolo de comunicação que foi adaptado para a realidade do *chip* desenvolvido pela UnB, mas cujos algoritmos foram montados em código *assembly* do microcontrolador PIC 16F84A. No capítulo 4, temos a tradução de todo o código *assembly* do PIC para o código *assembly* do *chip* da UnB. O capítulo 5 mostra os testes práticos realizados com o protocolo de comunicação e as análises de tráfego de duas arquiteturas de redes de sensores. O capítulo 6 finaliza o trabalho com a apresentação das conclusões obtidas no decorrer desta dissertação. Depois do capítulo 6 são deixadas as referências bibliográficas e apêndices que contém conteúdos relevantes ao tema. O código final do protocolo de comunicação é encontrado no Apêndice F para futuras consultas e melhorias.

2 – REDES DE SENSORES

2.1 - HISTÓRICO

Desde o princípio da eletricidade que o homem estuda como mensurar os fenômenos naturais e traduzi-los em sinais elétricos. Essa necessidade sempre foi motivada pela vontade de automatizar as tarefas humanas repetitivas. A solução foi desenvolver sensores que pudessem contar objetos, medir temperaturas, determinar composição de cores e uma infinidade de outros usos. Nas últimas décadas, houve um enorme avanço na miniaturização desses sensores, que passaram a ter características de baixo custo e baixo consumo de energia.

Embora não exista limite para o número de sensores utilizado em uma determinada aplicação, cada sensor deve ter seu próprio conjunto de fios que conecta ao circuito principal. Muitas vezes, a distância entre o circuito principal e o sensor tornava impeditiva a aplicação. Outras vezes, era preciso dotar os sensores com circuitos auxiliares para que a leitura do sinal pelo circuito principal não fosse prejudicada. Percebeu-se que os problemas apresentados seriam sanados se cada sensor tivesse inteligência própria através do uso dos microcontroladores. Assim, cada sensor poderia ser disposto em um barramento de dados comum a todos os sensores, o que eliminaria a necessidade de fios individuais, levando a uma enorme economia de metais condutores. A comunicação em um barramento também levou ao desenvolvimento de novos protocolos de comunicação, melhorando a comunicação de dados e aumentando as distâncias entre os sensores e o circuito principal, que nesse caso, este se tornou um computador com uma interface de comunicação igual a dos sensores.

Não demorou muito para que esses sensores inteligentes pudessem dispor de interfaces de comunicação sem fio. Afinal, os dispositivos de radiofreqüência e as técnicas de modulação de sinais também evoluíram bastante, diminuindo em tamanho, consumo e preço. Isso possibilitou a criação de uma nova categoria na área de sensoriamento: a de Redes de Sensores Sem Fio (RSSF). Tais redes passaram a viabilizar aplicações até então imaginadas somente na ficção científica.

2.2 - REDES DE SENSORES EM CAMPO

As redes de sensores podem ser vistas em muitas aplicações práticas, como por exemplo:

- A Universidade do Sul da Califórnia [17] (*University of Southern Califórnia*) realiza estudos sobre o monitoramento e detecção de proliferação desordenada de algas e bactérias hostis na água, usando redes de sensores sem fio.

- A Universidade de Tecnologia de Tampere [18] (*Tampere University of Technology*) construiu uma rede de sensores para controlar o tráfego de veículos, cujos nós ficam disfarçados como elementos do trânsito.

- O Professor Doutor Stefan Fischer [19] da Universidade de *Lübeck* está realizando estudos para fazer com que uma rede de sensores, instalada em vários pontos do corpo de um paciente, possa monitorar de forma quantitativa e qualitativa, os seus sinais vitais.

Existem aplicações que são mais bem resolvidas com redes de sensores sem fio, como por exemplo, a aplicação de Lundquist et al. [20] no *Yosemite National Park* (EUA) para recolher dados sobre o derretimento da neve que abastece de água mais da metade do estado da Califórnia (vide figura 2.1). Este tipo de estudo ainda possui poucos dados devido a dificuldade de instalar e fazer manutenção de estações meteorológicas em maiores altitudes.

Fonte: Gustavo Luchine Ishihara

Figura 2.1: *Yosemite National Park* – derretimento da neve nas montanhas

O processo do derretimento da neve é espacialmente complexo, pois a neve ocorre com profundidades e densidades diferentes, além da região geográfica ser bastante ampla. Esta é uma perfeita aplicação para as RSSF, pois os nós da rede são extremamente mais baratos que uma estação meteorológica, podendo ser espalhados para cobrir grandes áreas. Como não existe nenhuma infra-estrutura nas áreas monitoradas, os equipamentos têm que ser alimentados por baterias. O difícil acesso demanda que as informações mensuradas sejam transmitidas para que não necessite que um operador tenha que ficar recolhendo os dados periodicamente. Essa transmissão de dados deve ser realizada por radiofreqüência ou outro meio que não precise de fios, pois é inviável e muito intrusivo passar fiação ou cabeamento por uma área ecológica protegida. A informação coletada é vital para que órgãos de controle possam entender, predizer e informar a quantidade de recursos hídricos potáveis do estado da Califórnia.

2.3 - DESAFIOS E LIMITAÇÕES

Embora as soluções com redes de sensores sem fio sejam bastante elegantes, limpas e aparentemente simples, existem muitos fatores que tornam a escolha de cada especificação técnica, crítica para o funcionamento da aplicação desejada. Mais que isso, cada solução tem que ser cuidadosamente estudada para que sejam definidos como devem ser construídos os nós e como esses nós devem funcionar para coletar os dados relevantes para a aplicação, pois os recursos de cada nó são muito limitados.

Outro aspecto importante é que os nós das RSSF podem ser distribuídos de forma saturada em certo ambiente, para garantir um alto grau de redundância e alta disponibilidade. Nestes pontos, as RSSF podem se parecer com as redes móveis sem fio do tipo *ad hoc*, na qual todos os nós da rede funcionam como roteadores. Mas, de acordo com Karl e Willig [21], as aplicações específicas requerem uma reengenharia de alguns dos paradigmas básicos dos protocolos de comunicação, e por isso, uma rede de sensores é muito mais do que uma forma específica das redes ad hoc. A preocupação com a limitação dos recursos de uma rede de sensores faz com que cada implementação seja otimizada em *hardware* e *software* para tarefa a ser desempenhada.

Muitas oportunidades são abertas pelas RSSF, mas um grande desafio ainda a ser mais bem resolvido é a restrição quanto ao consumo de energia. As baterias e outras fontes de energia

são as maiores partes de um nó de uma RSSF. Segundo Rabaey et al. [22], a comunicação por radiofreqüência é extremamente importante em RSSF porque a energia dedicada nessa atividade, normalmente, supera todos os outros gastos de energia do nó sensor. Wolenetz et al. [23] também coloca que o gasto de energia para que um *bit* seja transmitido é muito maior do que o processamento de uma instrução, mas se muitas instruções são executadas se faz necessário considerar esse tipo de gasto no gerenciamento de energia. De forma similar, memórias cada vez maiores também podem ter um gasto de energia significante. Por tudo isso, a aplicação deve definir as características do nó de uma RSSF. Se a aplicação utiliza processamento intenso, colocar um microcontrolador mais rápido e tentar diminuir a memória e a quantidade de transmissões. Se a aplicação requer muita memória, tentar reduzir a capacidade de processamento e diminuir a quantidade de transmissões. Se a aplicação requer um tráfego intenso de informação, tentar reduzir a memória e a capacidade de processamento. Evidentemente, existem aplicações que requerem todos os requisitos juntos, e assim, deve-se aumentar a capacidade da bateria de forma que o tamanho físico do nó não seja um impedimento para a aplicação.

A gestão da energia em um nó de uma RSSF é tão crítica que levou ao desenvolvimento de inúmeras estratégias para se tentar prolongar o seu tempo de vida. Uma estratégia amplamente utilizada é fazer com que o nó reduza sua capacidade de processamento, baixando o número de ciclos por segundo, e consequentemente, a quantidade de instruções executadas por segundo. Este modo em baixa velocidade é normalmente chamado de *sleep*, consumindo muito menos energia que no modo normal. Outra estratégia é tornar a rede de sensores dirigida por eventos, uma vez que a comunicação deve ser iniciada pelos nós em função do acontecimento de algum evento. Neugebauer e Kabitzsch [24] sugere que além de ser dirigida por eventos a RSSF possa utilizar um esquema FDMA, onde cada nó tenta iniciar a comunicação em um canal, mas caso não haja resposta, o nó vai variando canal por canal até obter a resposta da estação de campo e poder enviar o pacote de dados. O pacote de dados é mantido pequeno para restringir a energia dissipada na transmissão em caso de colisão. Esta abordagem "*Send-On-Delta*" pretende estender o tempo de vida sem manutenção da bateria dos nós em anos, se a ocorrência dos eventos não for muito alta.

Então, o tempo de vida de um nó depende principalmente dos seguintes fatores mostrados na figura 2.2 a seguir.

Figura 2.2: fatores que afetam o tempo de vida de um nó

Um critério que serve para reduzir o gasto de energia é a sazonalidade da aplicação. Segundo Burrell et al. [25], em uma aplicação de RSSF em vinicultura foi percebido que no verão havia uma grande variação nos dados colhidos de dia e pouca variação nos dados colhidos a noite, já no inverno também havia uma grande variação dos dados a noite. Esta sazonalidade serve para reduzir o número de medições e de comunicações em períodos de pouca variação.

Outro grande desafio é sobre a inadequação dos protocolos de comunicação tradicionais de outras redes. As limitações de um nó sensor quanto ao poder de processamento, memória e alimentação elétrica, exigem que as RSSF utilizem protocolos de comunicação mais simples e mais dedicados ao cumprimento dos objetivos da aplicação particular.

2.4 - PROJETO DO SISTEMA DE CONTROLE DE IRRIGAÇÃO - SCI

Este projeto tem como patrocinadores a Empresa Brasileira de Pesquisa Agropecuária – Embrapa e o Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq, através do Instituto do Milênio que tem como objetivo o desenvolvimento de tecnologias multidisciplinares.

O desenvolvimento do Sistema de Controle de Irrigação (SCI) (2003) [26] vem da necessidade de se utilizar racionalmente um recurso natural muito valioso, a água potável. Tanto a falta de água como o seu excesso, trazem prejuízos para a produção de alimentos e

para a conservação do meio ambiente. O SCI busca fazer um monitoramento intensivo sobre as condições do solo, permitindo que sejam irrigadas apenas as micro-áreas que realmente estejam precisando e apenas nas quantidades certas. Este é o conceito da agricultura de precisão, que evita o desperdício de produtos agroquímicos aplicados em função de irrigações desnecessárias que lavam excessivamente o solo e as plantas, reduzindo assim, os custos de produção e a contaminação ambiental.

Para realizar esse monitoramento, será instalada uma rede de sensores sem fio que ficará distribuída em uma área de cultura (figura 2.3). O sistema é composto de uma estação de campo e de nós sensores.

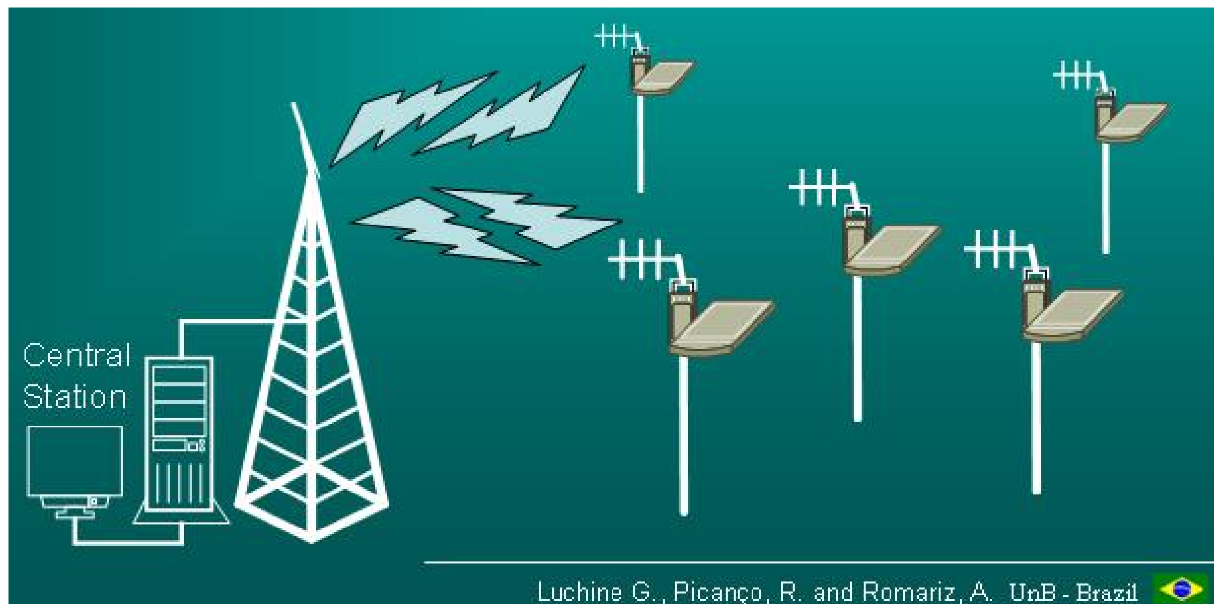

Figura 2.3: sensoriamento de uma cultura através de redes de sensores sem fio

Cada nó da rede de sensores mede a temperatura em três profundidades diferentes e a umidade do solo (através do potencial mátrico da água no solo). A estação de campo realiza a coleta de dados dos nós, aplica algoritmos de processamento desses dados e decide quais micro-áreas devem ser irrigadas. Então, a estação de campo comanda os nós correspondentes às áreas a serem irrigadas para que eles acionem seus atuadores para iniciar a irrigação. A quantidade de água a ser irrigada pode ser determinada dentro do mesmo comando que foi transmitido pela estação de campo para o nó.

Por se tratar da medição de grandezas físicas que não variam com grande velocidade, é aceitável que cada nó permaneça a maior parte do seu tempo no modo *sleep* de baixo consumo de energia. Assim, de tempos em tempos o nó "acordaria", passando para o modo de alto consumo, faria as medições e voltaria ao modo *sleep*. Entre várias medidas, a cada hora, o nó pode avisar a estação de campo que está "acordado" e disponível para receber dados e comandos. A estação de campo enviaria comandos para sincronizar seu relógio de tempo real, recolher os dados armazenados no nó e, se for o caso, mandar o nó executar algum comando. A bateria recarregável e o painel solar instalados em cada nó associados com um baixo consumo de energia prolongam a vida estimada do nó para alguns anos de funcionamento sem intervenção humana.

Devido a grande limitação de poder de processamento, alimentação elétrica e memória intrínseca aos sistemas embarcados, a maior parte do gerenciamento do sistema deve ser implementada na estação de campo. Os nós escravos devem conter o mínimo de condições necessárias para o desempenho de suas tarefas básicas, tais como: coleta de dados de seus sensores, comunicação com a estação de campo e execução de comandos. Como cada nó tem a sua janela de tempo (*time slot*) para se comunicar com a estação central, a sincronização global dos nós é fundamental para se evitar a colisão de pacotes. Este esquema de gerenciamento é chamado de *scheduling* ou agendamento, pois a estação de campo sincroniza os nós da RSSF e designa a janela de tempo no qual o nó deve se comunicar. Do ponto da sincronização global de RSSF, Ping [27] desenvolve uma técnica de sincronização de tempo baseado na medida do atraso (DMTS) que é eficiente no gasto de energia para sincronizar uma RSSF e leve no processamento. A vantagem em relação a outros métodos é a baixa resolução de tempo necessária para sincronizar os relógios dos nós.

2.4.1 - Nó Sensor

Para facilitar o entendimento, um diagrama de blocos do nó sensor pode ser visto na figura 2.4 a seguir.

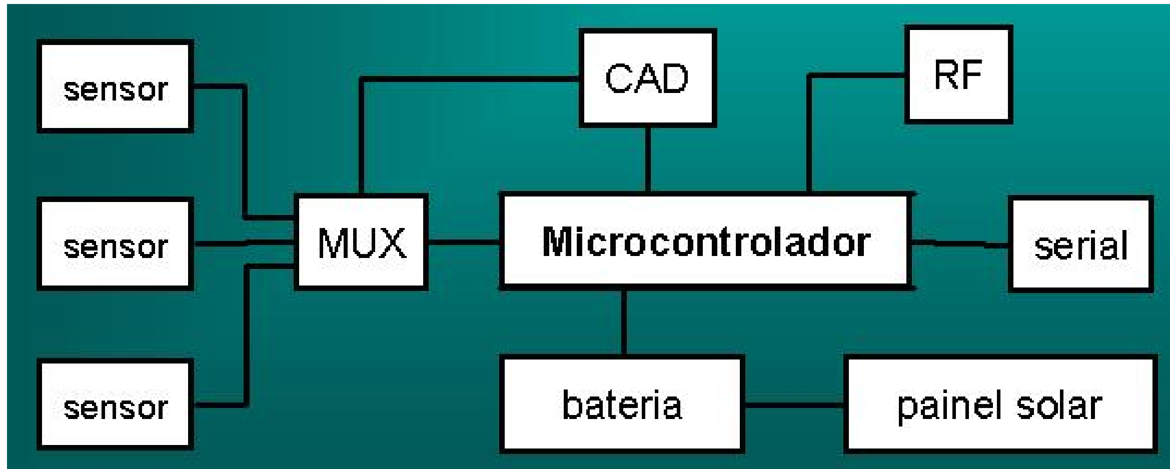

Figura 2.4: diagrama de blocos de um nó sensor

Fica evidente que o coração de um nó sensor é o microcontrolador. Nesta aplicação, o microcontrolador utilizado é o RISC16 que possui palavras de 16 *bits* e 16 instruções disponíveis, tendo sido desenvolvido pela Universidade de Brasília e com capacidade para funcionar em até 250 MHz. Na figura 2.5 são apresentados o esquema do microcontrolador e a sua implementação em silício.

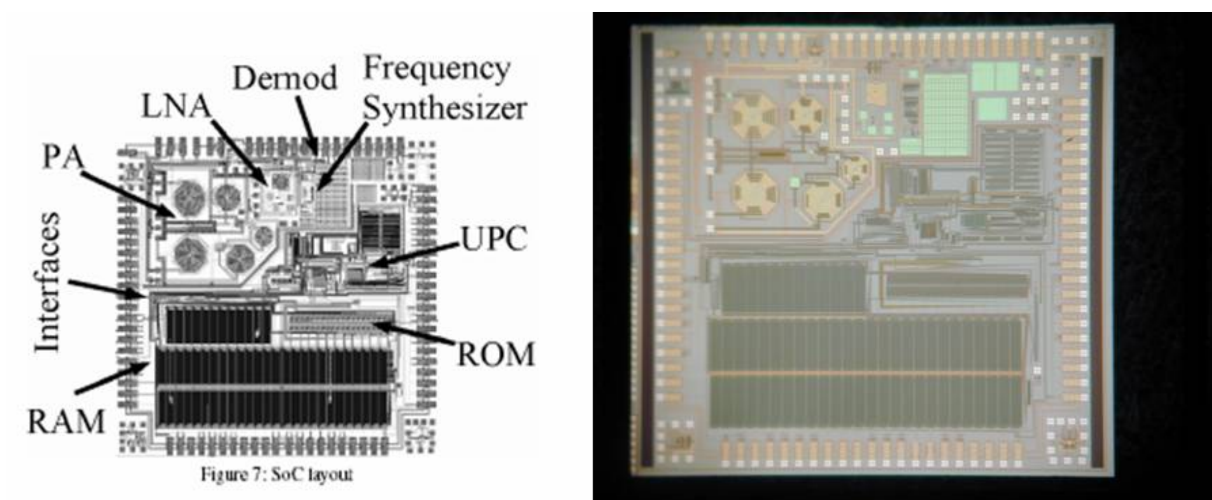

Figura 2.5: o microcontrolador RISC16 (fonte [26] (2003))

A figura 2.6 apresenta o encapsulamento do nó sensor, onde se verifica que o nó é semelhante ao tensiômetro mostrado do lado direito, com a diferença que o manômetro é substituído pelos circuitos digitais e chips, e que são agregados um painel solar e uma antena para comunicação.

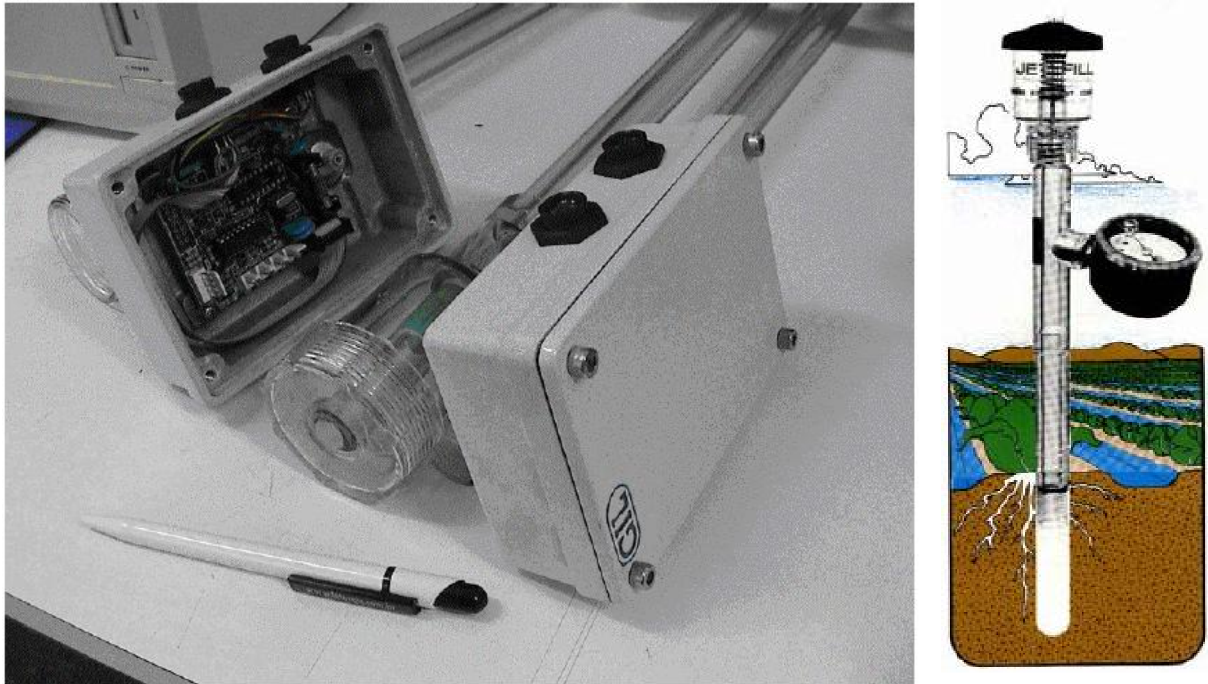

Figura 2.6: o nó sensor parcialmente montado (à esq.) e o tensiômetro (à dir.)

Uma vez apresentado todo o Sistema de Controle de Irrigação (SCI) [26], o seu funcionamento e o nó sensor, ficou assim colocada toda a base necessária para a correta contextualização dos capítulos seguintes que envolvem o protocolo de comunicação desta rede de sensores sem fio.

3 – O PROTOCOLO NO PIC

3.1 - CONSIDERAÇÕES INICIAIS

Para efetivar a comunicação em uma rede de sensores é necessário que seja definido um conjunto de regras e procedimentos padronizados a serem adotados pelos elementos de rede chamados de nós. Também é função do protocolo especificar o formato com que os dados são trafegados dentro de uma rede. Se os protocolos entre dois elementos de rede são diferentes, a comunicação não pode se estabelecer.

O protocolo proposto neste capítulo tem como tarefa principal a transferência de dados e a execução remota de comandos, essencial para o funcionamento de uma rede de sensores. A princípio, este protocolo de comunicação foi customizado para atender uma aplicação específica, mas nada impede que possa ser generalizado para outras aplicações.

3.2 - PREMISSAS BÁSICAS

O protocolo proposto deve caracterizar-se por utilizar relações de comando assimétricas do tipo mestre - escravo, pois apenas a estação de campo pode enviar comandos para os nós escravos. Entretanto, considerando que pode ser muito vantajoso para o gerenciamento do sistema, os nós escravos podem enviar mensagens para a estação de campo mesmo quando não requisitados, desde que não sejam comandos. Então, deve existir uma forma de distinguir as mensagens que são de dados das mensagens que possuem comandos.

O protocolo também deve permitir que o transmissor das mensagens saiba que elas chegaram com ou sem erro no receptor. Isso acontece através de uma mensagem de retorno que informa a condição em que a mensagem anterior foi recebida. Esta forma de funcionamento exige que as interfaces de comunicação funcionem em modo *half-duplex*.

Cada entidade que possuir a capacidade de comunicação nesta rede de sensores deve ter um endereço atribuído único. A única exceção é o endereço de *broadcast*, no qual a estação de campo dissemina uma mensagem para todos os nós escravos de sua rede. A comunicação pode então ocorrer ponto-a-ponto e *broadcast*. No modo de *broadcast*, os nós escravos não podem responder todos ao mesmo tempo, pois haveria colisão de todas as

mensagens. Assim, um critério de ordenação deve ser obedecido na implementação do protocolo.

Como não existe como prever todos os comandos que podem ser obedecidos pelos nós escravos, assim como atender as demandas futuras, a solução é manter esta parte do código bem acessível e de fácil visualização. Dessa forma, novas facilidades e outros usos podem ser adicionados conforme mais dispositivos vão sendo agregados, ficando garantida a evolução do sistema.

Em função das reduzidas capacidades de processamento e memória, o protocolo deve operar nas camadas física e enlace do modelo de referência *Open Systems Interconnection* (OSI) citado por Tanenbaum [28], deixando todas as funções das outras camadas para a estação de campo. Além disso, a tarefa de dividir dados para serem enviados em múltiplas mensagens também é da estação de campo, pois ela tem melhores condições de organizar mensagens perdidas ou duplicadas, e de controlar seu fluxo.

Para a comunicação se efetivar, todos os dados a serem transmitidos e recebidos devem transitar encapsulados em pacotes de dados. Estes pacotes devem ser manejados de forma apropriada ao meio em que será realizada a comunicação. A comunicação serial deve ser feita de forma assíncrona em grupos de dez *bits*, sendo que o primeiro é o *start bit* e o último o *stop bit*. Os oito *bits* do meio de cada grupo serão partes dos pacotes de dados que serão integralmente restituídos na recepção. Na comunicação por radiofreqüência (RF), os pacotes de dados não precisam ser divididos, podendo ser transmitidos do início ao fim de uma só vez.

3.3 - METODOLOGIA DE IMPLEMENTAÇÃO

Dentre os vários protocolos estudados, o *Scaleable Node Address Protocol* (SNAP) [29] se mostrou o protocolo mais adequado para esta aplicação por ser simples e exigir baixo processamento, um requisito de sistemas embarcados, além de seus autores disponibilizarem algumas ferramentas úteis que simplificariam os testes realizados posteriormente. O protocolo SNAP (2002) [29] é de uso livre, permitindo que sejam feitas modificações em suas implementações, e trabalha independentemente do meio de transmissão. Também existem bibliotecas *Dynamic Link Library* (DLL) para o Windows e

para o Linux, o que reduz o tempo necessário para desenvolver aplicações, assim como possibilita a utilização de funções estatísticas embutidas.

Tendo sido eleito para ser o protocolo do projeto da Embrapa, ficou resolvido que ele seria implementado em duas etapas principais: a primeira seria a implementação em microcontroladores Microchip [30] PIC16F84A de 8 *bits* vista neste capítulo, e a segunda seria a tradução do código obtido na primeira etapa para o RISC16 da UnB, abordada no capítulo 4. Este procedimento visou suprir a falta de ferramentas de desenvolvimento que o novo microcontrolador da UnB ainda possui. Na fase de implementação deste protocolo, havia apenas o montador do RISC16, sendo que o simulador estava em desenvolvimento no Linux, para funcionar em modo texto na linha de comando. Enquanto isso, o microcontrolador da Microchip [30] tinha o MPLAB *Integrated Development Environment* (IDE), que possibilitava um ambiente de produção gráfico de geração de código e simulação de execução das instruções, o que ajudou a perceber alguns erros conceituais e de construção das estruturas, corrigindo-os antes que fossem percebidos através de testes práticos de validação do protocolo.

3.4 - IMPLEMENTAÇÃO NO PIC

No sítio do SNAP [29] na Internet foi encontrada uma implementação deste protocolo para o PIC feita por Castricini e Marinangeli [31]. Inicialmente, foi pensado que os arquivos que estes dois autores disponibilizaram resolveriam a primeira etapa, deixando mais tempo para o desenvolvimento da etapa seguinte, a tradução para o RISC16. Mas uma análise mais detalhada no código *assembly* evidenciou a necessidade de um grande número de modificações. Para começar, o código *assembly* de Castricini e Marinangeli [31] foi feito para funcionar com um modem, pois possuía linhas de controle de fluxo e dependia delas para transmitir e receber os *bits*. Implementava os três *bytes* previstos no SNAP para os endereços de origem e destino, consumindo mais memória do que o necessário. No SNAP, existe um mecanismo para detecção de erro que consiste em transmitir o mesmo pacote três vezes seguidas, caso os três pacotes recebidos fossem idênticos, o receptor deveria aceitar o pacote como válido. Mas no código disponibilizado, tanto a transmissão quanto a comparação no receptor era realizada três vezes seguidas *byte* a *byte*, o que não retrata o padrão e não serve para funcionamento com outros dispositivos SNAP. O modo de recebimento das mensagens não era compatível com a aplicação desejada, pois o

programador deveria saber quando estaria chegando um pacote de dados para executar a função de receber. E esta função, uma vez executada, ficava esperando que o modem enviasse o pacote, pois caso contrário ficaria em *loop* até que um pacote fosse recebido. Caso um pacote fosse transmitido, tendo sido requerida a resposta sobre seu recebimento (*acknowledge*), a implementação esperaria durante certo tempo por um único pacote de dados. Caso este pacote viesse após um outro pacote, seria considerado como não recebido pelo receptor. Castricini e Marinangeli [31] também não previram a execução de comandos, pois não havia espaço reservado no código *assembly* para que eles fossem incluídos, além do quê todos os pacotes com comandos seriam rejeitados. Por não considerar que poderiam existir pacotes de comandos, também não previram que um pacote de comando pudesse requerer que mais de um pacote de resposta fosse enviado ao nó de origem. Outro fator preponderante, mas que não pode ser considerado negativo, era que a implementação feita considerava o tamanho do pacote de dados variável na recepção, mas fixo na transmissão. Esta variação no tamanho do pacote tornou o código *assembly* mais extenso e gastou mais memória. Isso também dificulta um pouco o tratamento dos dados recebidos pela aplicação principal, pois esta teria que verificar os *bits* de controle do cabeçalho do pacote recebido para conhecer a quantidade de novos dados que chegou na memória. No mais, foram observados pequenos erros de sintaxe e uma macro[2] que deve ser substituída por uma função, pois o montador do RISC16 não implementa esse tipo de estrutura.

Tendo em vista todo o trabalho já realizado e as modificações necessárias para adequação do protocolo, é apresentada a seguir a estrutura do SNAP Modificado, objeto da primeira etapa deste trabalho.

---

[2] É a substituição de cada ocorrência de uma entrada por um código expandido de um novo programa

3.4.1 - Organização e estrutura dos pacotes

SYNC	HDB2	HDB1	DAB1	SAB1	Dbi (1<=i<=8)	CRC-8

Figura 3.1: pacote de dados do SNAP Modificado

O pacote é composto de partes de oito *bits* (octetos), sendo que a primeira parte é o sincronismo da comunicação (SYNC); as duas seguintes HDB2 e HDB1 são *bits* de configuração e controle do cabeçalho (*header*); o DAB1 contém o endereço de destino (*Destination Address Byte* – DAB); o SAB1 contém o endereço de origem (*Source Address Byte* – SAB); os *bytes* DB1, DB2, DB3, DB4, DB5, DB6, DB7 e DB8 são para transporte de dados; e o último *byte* é para verificação de erros (*Cyclic Redundancy Check* de oito *bits* – CRC-8).

Como pode ser visto na figura 3.1, o pacote contém 14 *bytes* ou 112 *bits* fixos, caso haja menos dados a serem transmitidos, a aplicação principal deve preencher com zeros o restante do pacote. Este será o tamanho fixo desta implementação para que ela continue compatível com o SNAP original, trazendo a vantagem de se aproveitar toda a base de programas e bibliotecas já desenvolvidas para esse protocolo. O valor inicialmente especificado para este projeto era de 128 *bits*, mas isso seria ter 10 *bytes* de dados no pacote, valor que não encontra correspondência para configuração do cabeçalho do protocolo, pois de 8 *bytes* de dados, o SNAP pula para 16 *bytes*.

O *byte* de sincronismo é padrão do protocolo SNAP e será o mesmo adotado para o SNAP Modificado:

Bits	7	6	5	4	3	2	1	0
SYNC	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**

Figura 3.2: *byte* que inicia todo pacote de dados e de comando

O cabeçalho do pacote é dividido em duas partes: HDB2 e HDB1. Note que a numeração é inversa, primeiro é transmitido o HDB2 e depois o HDB1. Neste caso específico, o cabeçalho define através dos *bits* 6 e 7 que será utilizado um *byte* para o endereço de

destino (*Destination Address Byte* - DAB), e que através dos *bits* 4 e 5 será utilizado um *byte* para o endereço de origem (*Source Address Byte* - SAB). Nenhum *byte* foi utilizado para carregar mais *bits* específicos para o uso do protocolo, tendo seus *bits* 2 e 3 do HDB2 iguais a zero. O campo de ACK corresponde aos dois últimos *bits* do HDB2 e que perfazem quatro possíveis estados. Para transmissão, o *bit* 1 do HDB2 é zerado, e o *bit* 0 tem seu valor dependente da variável ACK_PKT_TX ter sido ligado ou não. Esta variável deve ser configurada antes do envio do pacote, pois estabelece se o protocolo deve ou não aguardar um pacote de resposta (*acknowledge*).

Bits	7	6	5	4	3	2	1	0
	DAB		SAB		PFB		ACK	
HDB2	**0**	**1**	**0**	**1**	**0**	**0**	**X**	**X**

Figura 3.3: primeira parte do cabeçalho do pacote

Na recepção, o ACK tem uma função muito importante, pois ele vai informar se o pacote recebido, em resposta ao pacote anterior, teve erros detectados ou se foi recebido sem problemas. Para isso, o *bit* 1 do HDB2 é colocado a 1 para indicar que se trata de resposta, e o *bit* 0 tem seu valor dependente de ter sido encontrado erro ou não no pacote recebido. O formato do ACK e seus *bits* são mostrados a seguir:

Bits	1	0
Não pede ACK (Tx)	**0**	**0**
Pede o ACK (Tx)	**0**	**1**
Resposta do ACK (Rx)	**1**	**0**
Não houve ACK (Rx)	**1**	**1**

Figura 3.4: interpretação dos *bits* de ACK do *byte* HDB2

A segunda parte do cabeçalho contém o *bit* 7 que distingue se o pacote é de dados ou de comando (*command* - CMD); os *bits* 6, 5 e 4 que definem o método de detecção de erro (*Error Detection Method* - EDM); e os *bits* 3, 2, 1 e 0 que informam quantos *bytes* de dados (*Number of Data Bytes* – NDB) existem no pacote. Nesta implementação, temos como segunda parte do cabeçalho:

Bits	7	6	5	4	3	2	1	0
	CMD	EDM			NDB			
HDB1	**X**	**0**	**1**	**1**	**1**	**0**	**0**	**0**

Figura 3.5: segunda parte do cabeçalho do pacote

No HDB1, apenas temos como variar o *bit* 7 de comando, pois o método de detecção de erro implementado é o CRC-8, cujo código no SNAP é definido (011b), podendo ser visto na figura 3.5. A quantidade de *bytes* de dados (NDB) também foi fixada em 8 *bytes* (1000b).

Tanto o endereço de destino (DAB1) quanto o de origem (SAB1) podem ser preenchidos com qualquer valor. Lembrando que o endereço 0 (zero) corresponde ao endereço de *broadcast* e não deve ser utilizado para representar qualquer dispositivo ou nó.

Bits	7	6	5	4	3	2	1	0
DAB1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 3.6: endereço de destino do pacote

Bits	7	6	5	4	3	2	1	0
SAB1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 3.7: endereço de origem do pacote

Conforme as figuras 3.6 e 3.7 mostram, existem nesta implementação no PIC apenas 255 endereços possíveis nesta rede de sensores. Como esta não será a implementação final, não houve maior preocupação de ampliar estes valores. Outra possibilidade imediata para aumentar o número de nós seria utilização de outra freqüência de comunicação, ou outro barramento físico comum.

Os *bytes* de dados úteis (*Data Bytes* ou *payload*) ou comandos também podem ser preenchidos com qualquer valor sem nenhuma restrição, conforme a figura 3.8 abaixo.

Bits	7	6	5	4	3	2	1	0
DBi	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 3.8: *byte* de dados (DBi, 1<=i<=8) ou comando

O *byte* do método de detecção de erro (CRC-8) vem sempre após todos os *bytes* de dados ou comandos, contendo o resultado de um procedimento matemático, que segundo Hlávka et al. [32], visa distinguir diferenças de até dois *bits* no pacote. Assim, o pacote recebido deve conter o mesmo resultado que foi calculado antes do envio do pacote através do meio de transmissão. Caso estes resultados sejam diferentes, o pacote é assumido como erro de transmissão, não devendo ser considerado para efeitos de processamento; seus dados não são válidos e suas ordens não devem ser obedecidas. Importante salientar que o *byte* de sincronismo (SYNC) não é considerado nestes cálculos de detecção de erros.

Bits	7	6	5	4	3	2	1	0
CRC-8	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 3.9: *byte* de detecção de erro (CRC-8)

Caso o pacote recebido requeira uma confirmação de recebimento, o receptor verifica se o CRC-8 calculado e o colocado no fim do pacote conferem. É assim que o receptor decide se houve erro na transmissão de um determinado pacote. Após a conferência dos resultados da detecção de erros, coloca os *bits* apropriados no campo de ACK do HDB2 e envia outro pacote de volta. O transmissor do pacote que iniciou a comunicação terá um *bit* de controle chamado RECEBIDO levado para 1 se, e somente se, o pacote de retorno não tiver sido rejeitado, não tiver havido erros detectados e se os *bits* de ACK indicarem que houve resposta do *acknowledge*. Qualquer outra condição leva o *bit* RECEBIDO para 0, ficando a cargo da aplicação principal retransmitir o pacote ou ignora-lo, descartando-o.

Uma vez esclarecidas a organização e a estrutura dos pacotes do protocolo SNAP Modificado, pode-se detalhar melhor o fluxo de funcionamento mostrado a seguir.

3.4.2 - Fluxo do protocolo

Para cumprir sua função básica de comunicação com a estação de campo, tanto respondendo a comandos como enviando dados, o protocolo foi desenvolvido em duas partes principais: a de recepção e a de transmissão. As duas partes implementadas são recursivas, de forma que uma pode invocar a outra. Caso o pacote de dados recebido esteja com o *bit* de *acknowledge* ligado, a parte de recepção deve responder a quem emitiu o pacote originalmente se o pacote foi recebido e se houve erro detectado. Caso o pacote de

dados transmitido esteja com o *bit* de *acknowledge* ligado, a parte de transmissão espera pelo pacote de resposta para saber se houve erro na comunicação ou se o pacote foi recebido sem erro. Para que as partes de transmissão e recepção não entrem em *loop* infinito, alguns *bits* de um registrador foram utilizados para controle de estado. O *bit* de transmissão foi chamado de *bit* TX e o de recepção de *bit* RX. Assim, quando o TX_SNAP for executado, o valor 1 é colocado no *bit* TX. Este *bit* somente é colocado a zero depois de terminada a transmissão. O mesmo acontece para o *bit* RX.

A aplicação principal pode executar tanto a função de transmitir um pacote, quanto a de receber (através de uma interrupção). Veja na figura 3.10 que em ambas as alternativas, o *Program Counter* (PC) tem valor inicial igual a X, pois era onde o programa se encontrava antes das chamadas de função TX_SNAP e RX_SNAP, sendo devolvido na posição X+1, com o advento do retorno das funções na instrução RETURN e para continuar executando a aplicação principal.

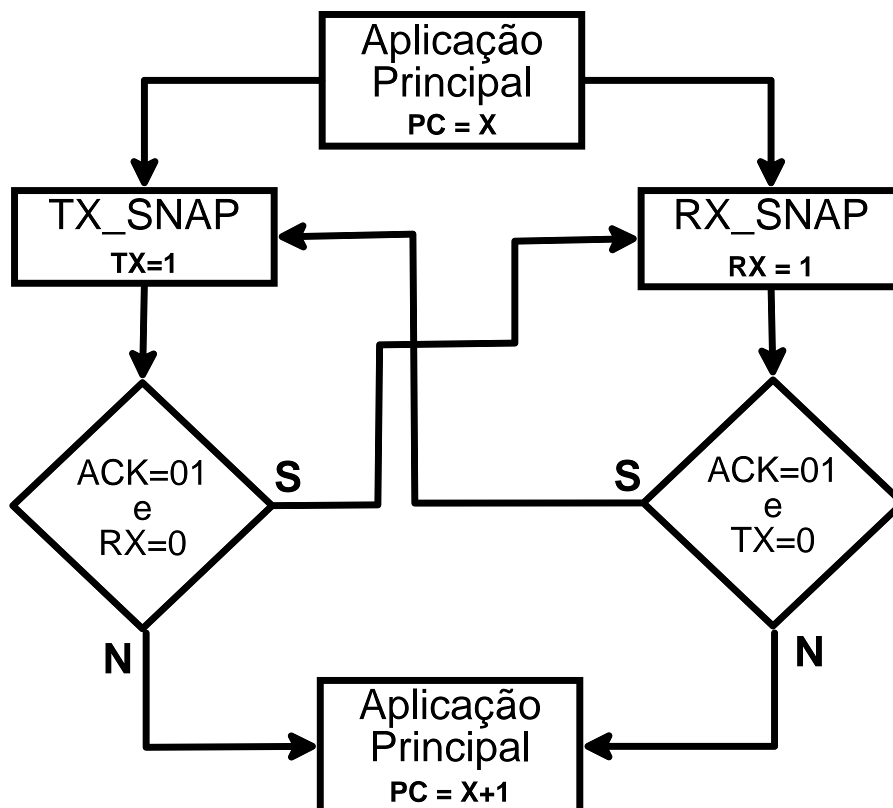

Figura 3.10: fluxo simplificado do protocolo SNAP Modificado

Outro aspecto importante que define o caminho do fluxo é a questão da confirmação de recebimento do pacote (*acknowledge*). Caso o pacote não requeira a confirmação do

recebimento (ACK=00), tanto o TX_SNAP quanto o RX_SNAP são executadas até o final sem que uma função chame a outra. Caso a confirmação de recebimento seja requerida (ACK=01), então o TX_SNAP chama o RX_SNAP e vice-versa, utilizando para isso os *bits* de controle TX e RX para verificar se a outra função não já estava sendo executada, o que evita a recursividade infinita.

Para detalhar melhor como a confirmação de recebimento funciona, foram desenhados na figura 3.11 os seguintes diagramas de tempo entre dois nós da rede de sensores.

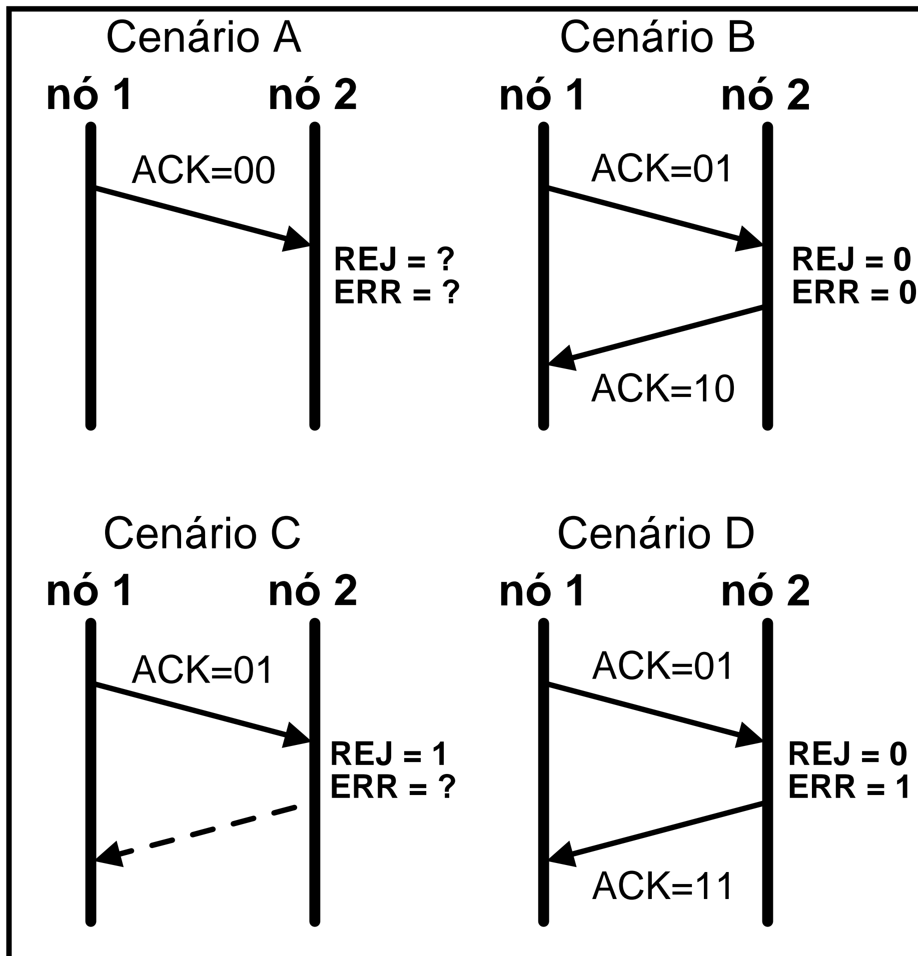

Figura 3.11: diagramas de tempo detalhando o funcionamento do *acknowledge* (ACK)

No Cenário A, o nó 1 enviou um pacote para o nó 2. Como os *bits* de *acknowledge* (ACK) estavam ambos zerados, nenhuma confirmação foi pedida sobre o pacote transmitido de 1 para 2. Então, o nó 1 não tem certeza sobre o recebimento do pacote, assim como não sabe se, depois de recebido o pacote pelo nó 2, houve alguma distorção ou interferência no meio de transmissão que alterou algum *bit* do pacote.

No Cenário B, o nó 1 configurou os *bits* de ACK para exigir uma confirmação do recebimento do pacote e o transmitiu para o nó 2. Esse pacote é recebido no nó 2, pois o pacote não foi rejeitado (REJ=0). O nó 2, após ter recebido o pacote, chama a função que executa o método de detecção de erro que, indicando o valor zero para o *bit* ERR, mostrou que nenhum erro foi detectado e o pacote foi considerado válido. Desta forma, o pacote foi processado, entregando informações ou executando comandos. O nó 2, tendo realizado o processamento necessário, envia um pacote de resposta para o nó 1 com os *bits* no ACK que informa que recebeu o pacote sem erros. O nó 1, tomando conhecimento do recebimento sem erros, torna o *bit* RECEBIDO igual a 1. A função deste *bit* é informar à aplicação principal que o pacote enviado foi recebido pelo destinatário com sucesso.

No Cenário C, o nó 1 colocou a confirmação de recebimento (ACK) dentro do pacote e o transmitiu para o nó 2. O nó 2 não recebeu o pacote que o nó 1 transmitiu, pois, enquanto estava recebendo, aconteceu uma das seguintes situações: os cabeçalhos tinham uma configuração diferente do programado, o pacote não foi endereçado ao nó 2, ou o pacote recebido tem como endereço de origem o endereço do nó 2. Tendo sido rejeitado (REJ=1), o pacote não é considerado para nenhum outro tipo de processamento e é descartado pelo nó 2. Assim, nenhuma resposta é enviada ao nó 1. Então, o nó 1 sabe que, ou o pacote enviado, ou o pacote de resposta foi perdido, pois a resposta não veio depois de decorrido certo intervalo de tempo. O *bit* RECEBIDO é igualado a zero para indicar para a aplicação principal que não existe confirmação de que o pacote foi recebido pelo destino.

No Cenário D, o nó 1 enviou um pacote para o nó 2 com o pedido de confirmação de resposta (ACK). O nó 2, apesar de ter recebido o pacote, detectou ter havido um erro, pois o resultado calculado do CRC-8 está diferente do resultado que veio no final do pacote recebido. Desta forma, o pacote não foi rejeitado (REJ=0), mas teve um erro detectado (ERR=1). Como o pacote requeria confirmação de recebimento, foi enviada uma resposta do nó 2 para o nó 1 com a confirmação de que o pacote foi recebido com erro (ACK=11). O nó 1 sabe que, apesar de o pacote ter sido recebido, houve erro na comunicação. O *bit* RECEBIDO é igualado a zero para indicar para a aplicação principal que o pacote não foi recebido adequadamente e não foi processado.

Como requisito fundamental para detectar a existência ou não de erros nos pacotes em ambos os sentidos de transferência, a função APPLY_EDM é utilizada tanto pela função

TX_SNAP quanto na RX_SNAP, pois todo pacote transmitido tem o resultado encontrado pelo método de detecção de erros aplicado no seu final, e todo pacote recebido tem esse valor embutido no pacote para ser comparado com o novo resultado calculado. A função APPLY_EDM é única, mas foi colocada em dois lugares diferentes na figura 3.12 abaixo apenas por uma questão didática.

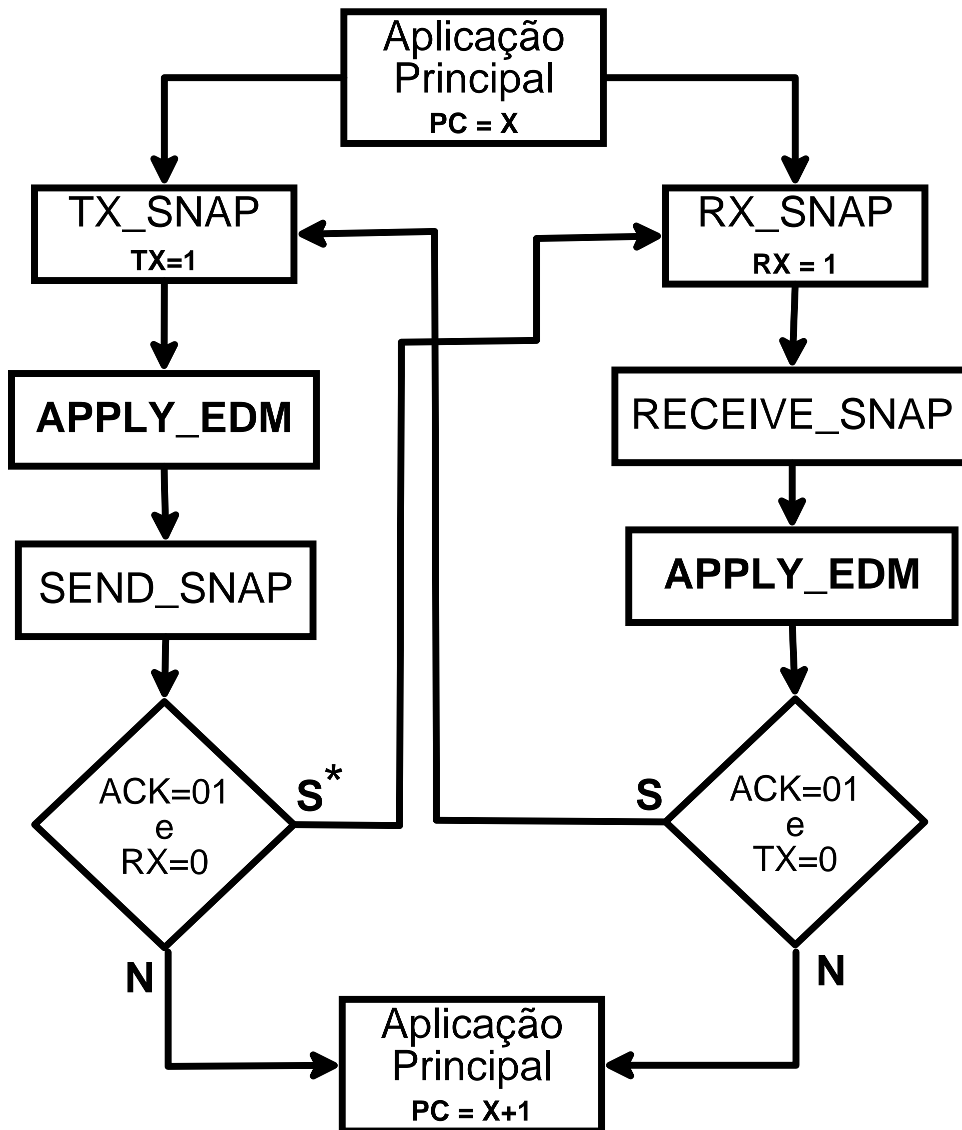

Figura 3.12: fluxo do protocolo SNAP Modificado considerando a função APPLY_EDM

O asterisco (*), colocado na letra "S" do símbolo de decisão da função TX_SNAP, está indicado para lembrar que o recebimento do pacote não é mandatório, mas ocorre por interrupção do microcontrolador, caso o pacote enviado não tenha sido rejeitado ou o pacote de confirmação de recebimento (ACK) não tenha sido perdido.

Repare na figura 3.12 que a função TX_SNAP chama a função APPLY_EDM antes de chamar a função que efetivamente vai transmitir o pacote (SEND_SNAP), pois o resultado do cálculo de detecção de erro tem que ir ao fim do próprio pacote a ser transmitido. A função RX_SNAP funciona diferentemente, primeiro recebe efetivamente o pacote através da função (RECEIVE_SNAP) e depois chama a função APPLY_EDM. A função APPLY_EDM sabe através dos *bits* de controle RUN_TX e RUN_RX o que deve ser feito: calcular o resultado do pacote para aplicar no final dele mesmo, ou calcular o resultado do pacote para comparar com o resultado que veio no fim dele mesmo.

Tendo sido requerida nos pacotes enviados ou recebidos uma confirmação de recebimento (*acknowledge*), a função APPLY_EDM é chamada duas vezes: uma pela função TX_SNAP e outra pela RX_SNAP, não necessariamente nessa ordem. Nesses casos, analisar apenas os *bits* RUN_TX e RUN_RX não é suficiente, pois, ao responder o pacote ou receber um pacote de resposta, ambos estão com o valor 1. A função APPLY_EDM precisa saber quem estava sendo executado antes para tomar uma decisão sobre o que deve ser feito com o CRC-8 calculado do pacote. Se a função RX_SNAP estava sendo executada antes, então será transmitido um pacote de resposta e a função APPLY_EDM deve aplicar o CRC-8 calculado no *byte* correspondente do pacote. Se a função TX_SNAP estava sendo executada antes, então está sendo recebida uma resposta acerca do recebimento do pacote transmitido e a função APPLY_EDM deve comparar o CRC-8 calculado com o que veio no *byte* correspondente do pacote. Assim um *bit* denominado RUN_RTX foi designado para realizar tal controle. O valor desse *bit* depende de qual função foi executada anteriormente: 1 no caso da função RX_SNAP, e zero no caso da função TX_SNAP.

Cada função do protocolo pode chamar outras funções para realizar tarefas repetitivas ou, simplesmente, para separar e organizar outras ações. Essa modularidade permite que o código seja adaptado para funcionar com outros tipos de hardware de transmissão, pois o protocolo fica didaticamente melhor distribuído, o que melhora também a sua manutenibilidade[3]. Para explicar melhor como cada função chama outras funções, foi desenhada a máquina de estados do protocolo que pode ser vista na figura 3.13 a seguir. A figura 3.13 se encontra ampliada no Apêndice D no final deste trabalho.

---

[3] Facilidade de um item em ser mantido ou recolocado em condições de executar suas funções requeridas

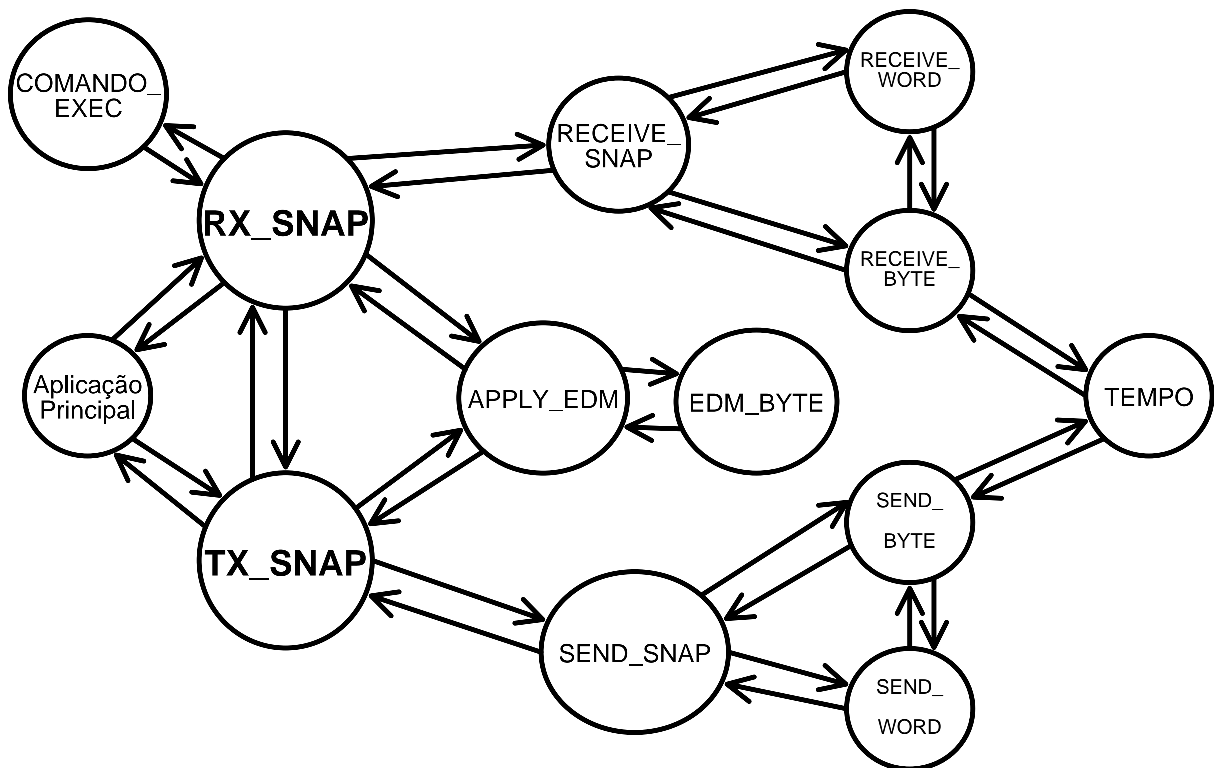

Figura 3.13: Máquina de Estados do Protocolo SNAP Modificado

A máquina de estados do protocolo SNAP Modificado mostra as funções utilizadas e seu fluxo de funcionamento. Note que todas as funções retornam para as funções que as chamaram. Isto se dá por causa do efeito das instruções CALL e RETURN. A instrução CALL realiza a chamada de uma subfunção; a RETURN, o retorno para o ponto seguinte da função que a chamou. Como podem ser vistas, essas chamadas de função são ordenadas, pois não há no fluxo a opção de que a função RX_SNAP chame a função SEND_SNAP diretamente. Assim como a função TX_SNAP não pode chamar a função COMANDO_EXEC diretamente. O início do fluxo se encontra na aplicação principal, que pode chamar tanto a função de transmitir TX_SNAP, quanto pode receber uma transmissão através de uma interrupção no microcontrolador e na conseqüente execução da função RX_SNAP.

Conforme dito na explicação da figura 3.12, a função APPLY_EDM é única, podendo ser acessada tanto pelo TX_SNAP quanto pelo RX_SNAP. Na seqüência, a função APPLY_EDM chama a função EDM_BYTE repetidas vezes, depois retornando a função que originalmente a chamou.

Apesar de parecer, não existe um meio de perfazer circularmente o fluxo, pois as funções devem retornar para a função que as chamou. Cada dupla de setas, uma em cada sentido, está intrinsecamente associada. Uma das setas indica a chamada da função seguinte; a outra, no sentido contrário indica o retorno à função anterior. Exemplificando com a função TEMPO, não existe a possibilidade de que, uma vez sendo chamada pela função SEND_BYTE, ela retorne para a função RECEIVE_BYTE. Também não existe a possibilidade de que a função TEMPO chame qualquer outra função, pois aquela apenas retorna para a função que a chamou através da seta no sentido inverso daquela que a invocou. Esse raciocínio ficará melhor evidenciado nos fluxos desenhados passo a passo para a transmissão e para a recepção dos pacotes que serão mostrados a seguir. Para uma melhor visualização, a figura 3.14 se encontra ampliada no Apêndice D no final desta dissertação.

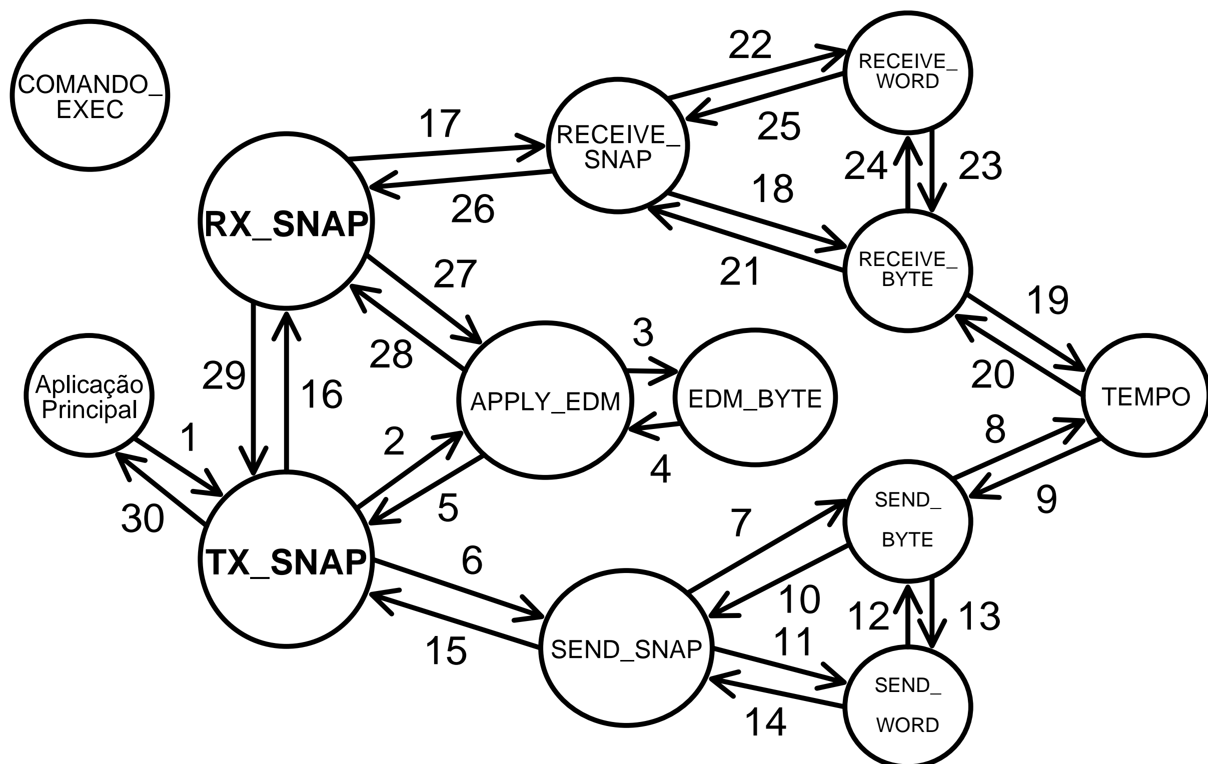

Figura 3.14: ciclo de transmissão de pacotes do protocolo SNAP Modificado

O ciclo de transmissão de pacotes inicia-se na aplicação principal, que através de seu código resolve transmitir algum pacote na rede. Para isso, coloca o endereço do nó que deve receber o pacote no *byte* de destino (DAB1) e os dados nos *bytes* de conteúdo útil (DBi). Após esta preparação inicial, executa a função TX_SNAP no passo 1 mostrado na figura 3.14.

No passo 2, a função TX_SNAP desabilita a ocorrência de todas as interrupções, coloca o *bit* RUN_TX igual a 1 para indicar que está transmitindo um pacote e verifica o estado do *bit* RX para checar se a função RX_SNAP também está sendo executada. Como RX é igual a 0, um terceiro *bit* chamado RUN_RTX é colocado a 0. Conforme já foi visto, este *bit* RUN_RTX tem a utilidade de determinar para a função APPLY_EDM o que deve ser feito, pois ela pode ser invocada por ambos TX_SNAP e RX_SNAP, tendo tarefas diferentes em cada caso. A função TX_SNAP então verifica o *bit* ACK_PKT_TX para checar se a aplicação principal deseja que seu pacote tenha confirmação de recebimento (ACK_PKT_TX=1) ou não (ACK_PKT_TX=0), e coloca os cabeçalhos no pacote. Depois carrega o endereço do nó no *byte* de origem (SAB1) do pacote. Tendo o pacote preenchido, chama a função APPLY_EDM, pois o pacote a ser transmitido precisa ter o resultado do cálculo de detecção de erros (CRC-8) no seu último *byte*.

No passo 3, a função APPLY_EDM limpa o conteúdo do registrador que vai acumular o resultado de todo o cálculo de detecção de erro (CRC-8), movendo *byte* por *byte* todo o pacote para um outro registrador e chamando a função EDM_BYTE a cada novo *byte*.

No passo 4, a função EDM_BYTE retorna para a função APPLY_EDM depois de calcular o resultado de detecção do erro do *byte* corrente e acumular com o resultado do *byte* anterior. Ressalta-se que o *byte* de sincronismo (SYNC) não é considerado neste cálculo.

No passo 5, a função APPLY_EDM retorna para a função TX_SNAP depois de verificar nos *bits* RUN_TX, RUN_RX e RUN_RTX o que deve ser feito. Como RUN_TX=1 e RUN_RX=0, o APPLY_EDM aplica o resultado do cálculo no *byte* de CRC-8 do pacote.

No passo 6, uma vez completamente determinado o pacote de transmissão, a função TX_SNAP chama a função SEND_SNAP para efetivamente enviar o pacote através do meio. A função SEND_SNAP então coloca *byte* por *byte* todo o pacote em um registrador, chamando as funções SEND_BYTE para transmitir um único *byte* ou SEND_WORD para transmitir mais de um *byte*. A primeira tarefa é colocar o *byte* de sincronismo (SYNC) no registrador de transmissão e chamar a função SEND_BYTE.

No passo 7, temos a função SEND_SNAP chamando a função SEND_BYTE. A função SEND_BYTE efetivamente transmite *bit* por *bit* o pacote no meio de transmissão. Assim, o *byte* que foi colocado no registrador de transmissão é rotacionado de forma a alterar o estado lógico de saída (alto ou baixo). Cada alteração de estado lógico deve esperar por um tempo determinado colocado em um registrador que é controlado através da função TEMPO.

No passo 8, a função SEND_BYTE chama a função TEMPO. A função TEMPO usa o valor colocado no seu registrador pela função SEND_BYTE e o utiliza para gerar uma espera, que no caso é de um *bit* na velocidade de transmissão configurada. Esta configuração é feita através de uma constante chamada de TEMPO_BIT tendo os seguintes valores calculados:

Tabela 1: taxa de comunicação e constantes TEMPO_BIT e TEMPO_MEIO_BIT

Velocidade	Tempo de *bit*	TEMPO_BIT	TEMPO_MEIO_BIT
2400 bps	416 μs	104	52
4800 bps	208 μs	52	26
9600 bps	104 μs	26	13

No passo 9, a função TEMPO retorna para a função SEND_BYTE depois de ter gerado uma espera com base na constante TEMPO_BIT configurada. No passo 10, a função SEND_BYTE retorna para a função SEND_SNAP depois de ter transmitido todo o *byte* de sincronismo (SYNC). Repare que os passos 8 e 9 são repetidos o mesmo número de vezes da quantidade de *bits* no *byte*.

No passo 11, a função SEND_SNAP coloca um ponteiro apontando para o primeiro *byte* do cabeçalho (HDB2) e o número de palavras em um registrador, chamando a seguir a função SEND_WORD. No passo 12, a função SEND_WORD coloca o conteúdo do ponteiro que a função SEND_SNAP deixou apontado no registrador de transmissão e chama a função SEND_BYTE. A função SEND_BYTE funciona como explicado anteriormente, rotacionando *bits* e chamando a função TEMPO para gerar os tempos de espera adequados entre cada *bit* (passos 8 e 9).

No passo 13, a função SEND_BYTE retorna para função SEND_WORD, que vai incrementar o ponteiro (apontando para HDB1), colocar seu conteúdo no registrador de transmissão e chamar a função SEND_BYTE novamente até que o número de palavras, configurado pela função SEND_SNAP (passo 11), seja decrementado até zero. No passo 14, a função SEND_WORD retorna para a função SEND_SNAP depois de ter transmitido todas as palavras através da repetição dos passos 12 e 13.

No passo 15, a função SEND_SNAP retorna para a função TX_SNAP após ter realizado todo este procedimento para o pacote. Ao final da transmissão do pacote, a função SEND_SNAP coloca o endereço do nó de destino (DAB1) em um registrador para futuras comparações com os pacotes de resposta, se for o caso. A função TX_SNAP verifica o *bit* ACK_PKT_TX para checar se o pacote transmitido requer a confirmação de recebimento (*acknowledge*). Caso negativo (*bit* ACK_PKT_TX=0), a função TX_SNAP vai para o seu fim, colocando o *bit* TX igual a zero, reabilitando as interrupções e retornando para a aplicação principal (passo 30). Caso positivo (*bit* ACK_PKT_TX=1), a função TX_SNAP vai esperar um determinado tempo para receber o pacote de resposta. Caso esta resposta não seja recebida no tempo determinado, o pacote transmitido é considerado rejeitado (REJ=1). Neste momento de espera pelo pacote de resposta, as interrupções são reabilitadas para permitir que um pacote ao chegar possa executar a função RX_SNAP.

No passo 16, a função TX_SNAP ao reabilitar as interrupções, permitiu que um pacote que está chegando na interface de entrada pudesse executar a função RX_SNAP. A função RX_SNAP vai tornar o *bit* RUN_RX igual a 1 para indicar que está sendo recebido um pacote e chamar outra função para o tratamento da ocorrência.

No passo 17, a função RX_SNAP chama a função RECEIVE_SNAP, que vai receber efetivamente o pacote do protocolo SNAP Modificado. A função do RECEIVE_SNAP é receber *byte* por *byte* o pacote, interpretando o cabeçalho e alterando o estado do *bit* REJ para indicar que o pacote foi todo recebido, ou se houve alguma condição que resultou na rejeição do pacote recebido. A primeira tarefa da função RECEIVE_SNAP é receber o *byte* de sincronismo (SYNC) através da execução da função RECEIVE_BYTE.

No passo 18, a função RECEIVE_SNAP chama a função RECEIVE_BYTE para que esta efetivamente receba *bit* a *bit* o *byte* que está sendo recebido. Assim, *bit* por *bit* recebidos vão sendo rotacionados no registrador de recepção. Cada *bit* recebido deve esperar por um determinado período de tempo configurado nas constantes TEMPO_BIT e TEMPO_MEIO_BIT. A primeira constante é configurada de acordo com a tabela 1 mostrada anteriormente e a segunda é exatamente a metade do valor configurado na primeira. Esta distinção decorre do fato de que o primeiro *bit* é lido no meio de sua duração e os seguintes podem ser lidos a cada tempo de um *bit* inteiro, pois fica garantida a sua leitura da mesma forma que o primeiro *bit* (ver figura 3.15).

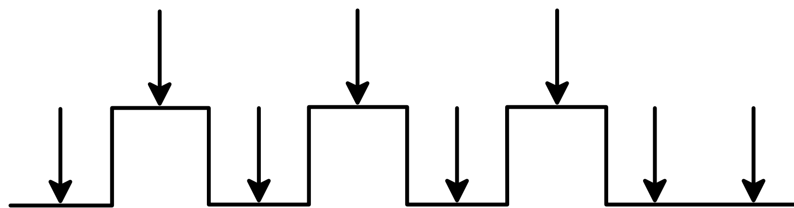

Figura 3.15: leituras sendo realizadas no *byte* de sincronismo (SYNC)

No passo 19, a função RECEIVE_BYTE chama a função TEMPO, que vai gerar uma espera de acordo com o valor configurado pela RECEIVE_BYTE. No passo 20, a função TEMPO retorna para a função RECEIVE_BYTE após ter sido decorrido o tempo de um *bit* ou de meio *bit*.

No passo 21, a função RECEIVE_BYTE retorna para a função RECEIVE_SNAP depois de ter recebido todo o *byte*. Assim, os passos 19 e 20 são repetidos o mesmo número de vezes da quantidade de *bits* no *byte*.

No passo 22, a função RECEIVE_SNAP aponta o ponteiro para a posição do primeiro *byte* do cabeçalho (HDB2), coloca o número de palavras igual a 2 em um registrador e chama a função RECEIVE_WORD. No passo 23, a função RECEIVE_WORD chama a função RECEIVE_BYTE, que vai efetivamente receber o *byte*.

No passo 24, a função RECEIVE_BYTE retorna para a função RECEIVE_WORD depois de ter recebido *bit* por *bit* do hardware de recepção e aguardado o tempo de cada *bit* através das repetições dos passos 19 e 20. No passo 25, a função RECEIVE_WORD retorna para a função RECEIVE_SNAP após ter incrementado seu ponteiro (apontando para HDB1) o

mesmo número de vezes colocado em um registrador no passo 22 e repetidos os passos 23 e 24 até que este número fosse decrementado a zero. Neste momento, a função RECEIVE_SNAP vai verificando, na medida em que os *bytes* vão sendo recebidos, se o cabeçalho do pacote está configurado de forma apropriada, se o endereço de destino (DAB1) do pacote coincide com o endereço do nó que está recebendo o pacote, se o endereço de origem do pacote (SAB1) é diferente do endereço do nó que está recebendo o pacote e se o endereço de origem do pacote (SAB1) é o mesmo endereço de destino (DAB1) do pacote anterior. Após o pacote ter passado pelos primeiros testes de recebimento, o *bit* REJ é igualado a zero para indicar que o pacote cumpriu seus requisitos básicos de recepção, não sendo rejeitado.

No passo 26, a função RECEIVE_SNAP retorna para a função RX_SNAP, pois todo o pacote do protocolo SNAP Modificado foi recebido. A função RX_SNAP verifica se o pacote foi rejeitado através do *bit* REJ. Caso positivo (REJ=1), limpa o *bit* RX e retorna para a função TX_SNAP. A função TX_SNAP vai continuar aguardando outro pacote, retornando ao passo 16, pois o pacote rejeitado pode ter partido de outro nó ou ter sido resultado de uma colisão entre pacotes. Caso negativo (REJ=0), o RX_SNAP verifica se houve erro na recepção do pacote através do passo seguinte.

No passo 27, a função RX_SNAP chama a função APPLY_EDM para verificar se houve erro (ERR=1) ou não (ERR=0) na recepção. A função APPLY_EDM funciona como descrito nos passos 3 e 4, repetindo até que o pacote seja calculado por inteiro. A APPLY_EDM também analisa os *bits* de controle (RUN_TX=1, RUN_RX=1 e RUN_RTX=0) para verificar se, depois de calculado o resultado do método de detecção de erros, precisa conferir o resultado calculado com o CRC-8 recebido no pacote (veja na tabela 2).

Tabela 2: *bits* de controle de transmissão e recepção do SNAP Modificado

RUN_ TX	RUN_ RX	RUN_ RTX	Situação
0	0	0	A aplicação principal está sendo executada
1	0	0	TX_SNAP está sendo executado
0	1	0	RX_SNAP está sendo executado

RUN_ TX	RUN_ RX	RUN_ RTX	Situação
1	1	0	TX_SNAP estava sendo executado quando deixou que uma interrupção chamasse a função RX_SNAP para receber um pacote de resposta (*acknowledge*)
1	1	1	RX_SNAP estava sendo executado quando chamou a função TX_SNAP para transmitir um pacote de resposta (*acknowledge*)

No passo 28, a função APPLY_EDM retorna para a função RX_SNAP informando se houve erro (ERR=1) ou não (ERR=0) no pacote recebido.

No passo 29, a função RX_SNAP retorna para a função TX_SNAP, pois ao analisar os *bits* de controle (RUN_TX=1, RUN_RX=1 e RUN_RTX=0), constata que estava recebendo um pacote de confirmação de recebimento (*acknowledge*), não havendo mais nada a ser realizado pela função RX_SNAP.

No passo 30, a função TX_SNAP desabilita novamente todas as interrupções, analisa os *bits* ERR, REJ e ACK, e checa o cabeçalho do pacote recebido para se certificar do recebimento do pacote enviado. Caso o *bit* REJ seja igual a 1, ou *bit* ERR seja igual a 1, ou o cabeçalho do pacote não contém o valor esperado (ACK=10), o *bit* RECEBIDO é zerado para indicar para a aplicação principal que o pacote enviado anteriormente não foi recebido pelo nó de destino, ou que não se tem o conhecimento de que o pacote enviado anteriormente tenha chegado no nó de destino. Caso os *bits* REJ e ERR estejam zerados (pacote recebido sem nenhum erro detectado) e o cabeçalho do pacote contenha o valor esperado (ACK=10), o *bit* RECEBIDO é igualado a 1 para indicar para a aplicação principal que o pacote foi recebido adequadamente. Após configurar o valor do *bit* RECEBIDO, a função TX_SNAP limpa o *bit* RUN_TX, verifica se o *bit* RUN_RX está zerado para reabilitar as interrupções e retorna para a aplicação principal.

Tendo percorrido o fluxo focado na transmissão, todo o seu percurso para a recepção de um pacote SNAP Modificado será refeito na figura 3.16 a seguir. O pacote escolhido contém um comando e requer uma confirmação de recebimento (*acknowledge*). Outros pacotes poderiam ter sido escolhidos, mas por ter um fluxo completo, este tipo de pacote serve também para explicar os fluxos mais simples.

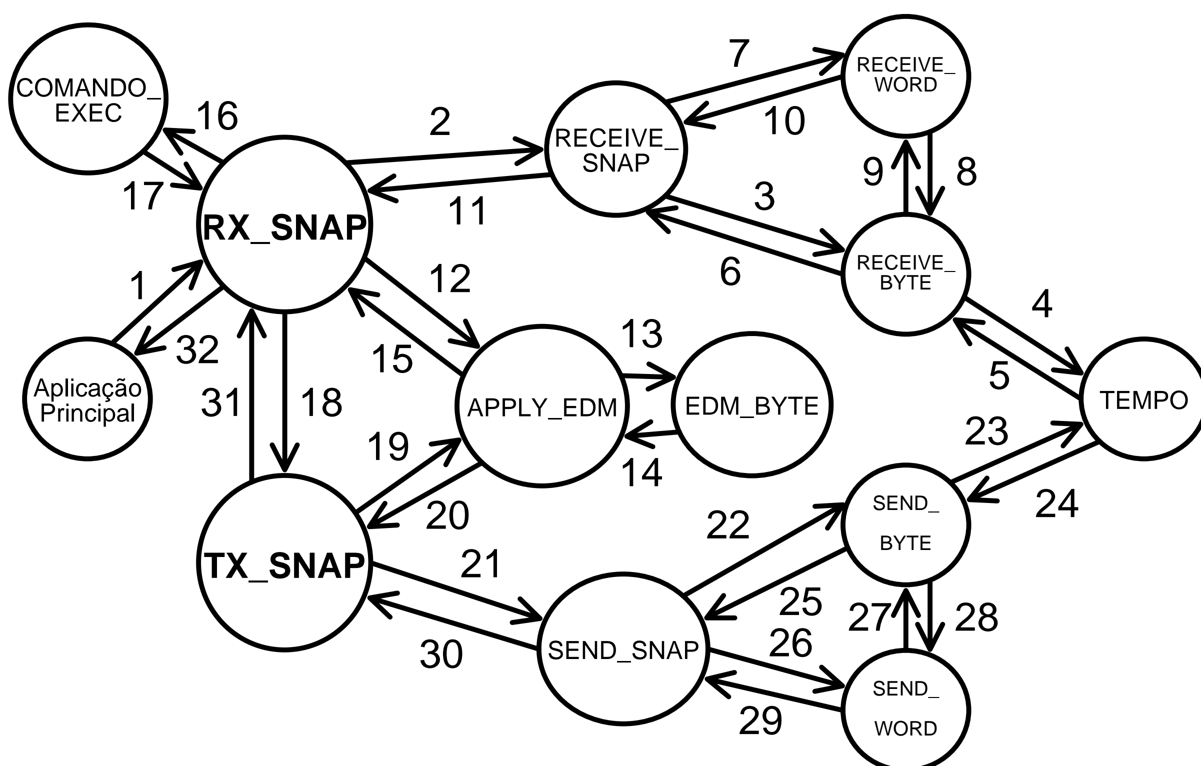

Figura 3.16: ciclo de recepção de pacotes do protocolo SNAP Modificado

Desta vez, a aplicação principal está executando as suas tarefas quando uma interrupção no microcontrolador é feita por causa do início de recebimento de um pacote do protocolo SNAP Modificado. O tratador de interrupção verifica que a interrupção ocorreu pelo recebimento de um pacote através das interfaces de entrada, executando a função RX_SNAP no passo 1.

No passo 2, a função RX_SNAP coloca o *bit* RUN_RX igual a 1 para indicar que a função de recepção está funcionando e chama a função RECEIVE_SNAP para efetivamente receber o pacote. No passo 3, a função RECEIVE_SNAP chama a função RECEIVE_BYTE para receber o primeiro *byte* do pacote, o *byte* de sincronismo (SYNC).

No passo 4, a função RECEIVE_BYTE efetivamente recebe os *bits* da interface de entrada no registrador de recepção, rotacionando-os a cada intervalo de tempo gerado ao se colocar um valor em um registrador de tempo e chamar a função TEMPO. O primeiro intervalo corresponde ao tempo de meio *bit* configurado na constante TEMPO_MEIO_BIT de acordo com a velocidade de comunicação escolhida (ver tabela 1 da página 32). Depois os intervalos correspondem ao tempo um *bit* inteiro visto nessa mesma tabela.

No passo 5, a função TEMPO retorna para a função RECEIVE_BYTE após decorrido o tempo determinado que havia sido colocado em um registrador no passo anterior. No passo 6, a função RECEIVE_BYTE retorna para a função RECEIVE_SNAP depois de ter recebido o *byte* no registrador de recepção.

No passo 7, a função RECEIVE_SNAP aponta o ponteiro para o primeiro *byte* do cabeçalho (HDB2), coloca o número 2 em um registrador de número de palavras e chama a função RECEIVE_WORD. No passo 8, a função RECEIVE_WORD chama a função RECEIVE_BYTE que vai efetivamente receber o *byte*, rotacionando o registrador de recepção e esperando os intervalos de tempo entre os *bits* com a repetição dos passos 4 e 5.

No passo 9, a função RECEIVE_BYTE retorna para a função RECEIVE_WORD, incrementa o endereço do ponteiro na mesma quantidade que se encontra no registrador de número de palavras e chama a função RECEIVE_BYTE (passos 8 e 9) novamente até que o número de palavras, que é decrementado em uma unidade a cada iteração, seja igual a zero. No passo 10, a função RECEIVE_WORD retorna para a função RECEIVE_SNAP que vai realizar uma verificação no cabeçalho recebido. Caso não estejam no formato do SNAP Modificado, esses *bytes* de cabeçalho serão descartados, o pacote é considerado rejeitado e o processo de recebimento é finalizado, retornando para a aplicação principal (passos 11 e 32). Caso o cabeçalho do pacote esteja no formato adequado, a função RECEIVE_SNAP verifica: se o pacote é de *broadcast* ou se o endereço de destino (DAB1) pertence ao nó que está recebendo o pacote, e se o endereço de origem do pacote (SAB1) não coincide com o endereço do nó que está recebendo o pacote. Se todas estas verificações forem verdadeiras, o *bit* REJ é levado para zero. Se alguma delas for falsa, o *bit* REJ é levado para 1, pois o nó não deve receber um pacote que não foi endereçado a ele, e também, não deve receber um pacote cujo endereço de origem é idêntico ao nó que o está recebendo, pois isso significaria que existem dois nós com o mesmo endereço, quebrando uma das premissas básicas no início deste capítulo.

No passo 11, a função RECEIVE_SNAP retorna para a função RX_SNAP que vai verificar se o pacote foi rejeitado. Caso tenha sido rejeitado (REJ=1), o fluxo do protocolo é desviado para o fim da função RX_SNAP, limpando o *bit* RUN_RX e retornando para a aplicação principal (passo 32). Caso tenha sido aceito (REJ=0), a função RX_SNAP prossegue na verificação de erros do pacote.

No passo 12, a função RX_SNAP chama a função APPLY_EDM, que se inicia limpando o registrador que vai acumular o resultado do método de detecção de erro (CRC-8), movendo *byte* por *byte* o pacote para um outro registrador e chamando a função EDM_BYTE para refazer os cálculos a cada novo *byte*. No passo 13, a função APPLY_EDM chama a função EDM_BYTE para que esta calcule o resultado de detecção de erro considerando o *byte* corrente.

No passo 14, a função EDM_BYTE retorna para a função APPLY_EDM com o resultado do cálculo do *byte* corrente acumulado no registrador de detecção de erro. A função APPLY_EDM repete os passos 13 e 14 até que todos os *bytes* do pacote tenham passado pelo método *byte* a *byte*, sendo que o resultado final é o acumulado de todas as iterações, menos a do *byte* de sincronismo (SYNC). Tendo chegado ao valor final do cálculo, a função APPLY_EDM interpreta os *bits* de controle (TX=0, RX=1 e RTX=0), verificando na tabela 2 que apenas a recepção está sendo executada e que o resultado calculado de CRC-8 deve ser conferido com o CRC-8 que veio no pacote. A comparação altera o estado do *bit* ERR.

No passo 15, a função APPLY_EDM retorna para a função RX_SNAP informando se houve erro (ERR=1) ou não (ERR=0) no pacote recebido. Caso tenha havido erro (ERR=1), o fluxo do protocolo é desviado para o fim da função RX_SNAP, limpando o *bit* RUN_RX e retornando para a aplicação principal (passo 32). Caso o pacote não contenha erros detectados (ERR=0), o RX_SNAP prossegue na verificação de comandos dentro do pacote.

No passo 16, a função RX_SNAP, chama a função COMANDO_EXEC que vai verificar se o *bit* de comando no cabeçalho informa que o pacote se trata de um comando ou dado. Caso não seja um comando, nada é executado, a função é desviada para o fim e retorna para o RX_SNAP (passo 17). Caso seja um comando, a função COMANDO_EXEC vai interpretar o conteúdo do primeiro dado DB1 e executar um comando específico.

No passo 17, a função COMANDO_EXEC retorna para a função RX_SNAP informando um código de execução de cada comando através do *byte* DB1. Assim, por exemplo, um comando de código 2 pode retornar o valor 3 no DB1 para indicar que o comando foi executado com sucesso. E, por exemplo, o valor 55 poderia estar no *byte* DB1 caso

nenhum dos comandos fosse executado. Este método é interessante por permitir que, tendo a confirmação de recebimento ligada (*acknowledge*), o nó de origem saiba se o comando foi executado, ou se o comando não é suportado pelo nó de destino. Depois de verificar se o pacote se trata de um comando ou dado, a função RX_SNAP analisa os *bits* de *acknowledge* do cabeçalho (ACK). Caso os *bits* sinalizem que o pacote não requer confirmação de recebimento (ACK=00), a função RX_SNAP limpa o *bit* RUN_RX e retorna para a aplicação principal (passo 32). Caso os *bits* sinalizem que o pacote requer resposta (ACK=01), a função RX_SNAP verifica se o endereço de destino do pacote recebido é de *broadcast*. Caso o pacote recebido não seja de *broadcast*, a função RX_SNAP continua normalmente e chama a parte de transmissão do SNAP Modificado. Caso o pacote recebido seja de *broadcast*, a função RX_SNAP vai esperar um determinado tempo de acordo com o número do nó e depois chamar a parte de transmissão do SNAP Modificado. Assim, nós com números de endereços menores respondem antes que os nós com números de endereços maiores e cada nó encontra seu *time slot* para transmitir uma resposta ao nó de origem. Nenhuma colisão é obtida, pois todos os endereços são únicos dentro da rede.

No passo 18, a função RX_SNAP chama a função TX_SNAP para enviar um pacote de resposta ao nó de origem. A função TX_SNAP desabilita todas as interrupções e leva o *bit* RUN_TX para 1, indicando que a parte de transmissão está sendo executada. Ao testar se o *bit* RUN_RX está igual a 1, o *bit* RUN_RTX também é levado para 1 (conforme tabela 2). Esta situação indica que a parte de recepção estava sendo executada antes e que a parte de transmissão foi executada depois. Na situação contrária os *bits* de controle seriam TX=1, RX=1 e RTX=0. A função TX_SNAP então carrega os cabeçalhos no pacote, verifica se houve erro no pacote recebido através do *bit* ERR e configura os *bits* de *acknowledge* do cabeçalho HDB2 de acordo com a figura 3.4. Outro aspecto a ser verificado é que determinados comandos podem fazer com que o nó tenha que enviar de volta, vários pacotes. Isso é feito através do *bit* de controle chamado SAB_PARA_DAB. Quando este *bit* está zerado, indica que ainda não houve a troca do endereço de origem do pacote recebido para o endereço de destino do pacote a ser enviado. Quando este *bit* está igual a 1, indica que já houve a troca de endereços para o primeiro pacote de resposta e que está pronto para enviar outros pacotes de resposta para o nó de origem. Dessa forma, a função TX_SNAP analisa o *bit* SAB_PARA_DAB para realizar a troca do endereço de origem do

pacote recebido com o endereço de destino do pacote a ser enviado e transfere seu endereço para o *byte* de origem do pacote a ser enviado.

No passo 19, a função TX_SNAP chama a função APPLY_EDM para determinar o resultado do CRC-8 em todo o pacote. Isto é feito *byte* a *byte* através da chamada repetitiva da função EDM_BYTE nos passos 13 e 14. A função APPLY_EDM analisa os *bits* de controle (RUN_TX=1, RUN_RX=1 e RUN_RTX=1) e decide aplicar o resultado total calculado no *byte* do CRC-8 do pacote a ser enviado. Esta decisão se baseia no fato de que os *bits* de controle indicam que, se a parte de transmissão estava sendo executada antes e a de transmissão foi executada depois, então um pacote de resposta está sendo enviado, devendo conter um novo cálculo de CRC-8 para ser aceito pelo nó que originou a resposta.

No passo 20, a função APPLY_EDM retorna para a função TX_SNAP que vai se preparar para enviar o pacote. No passo 21, a função TX_SNAP chama a função SEND_SNAP para efetivamente enviar o pacote de resposta de volta ao nó que originou a comunicação.

No passo 22, tendo o pacote preenchido com o CRC-8 que faltava, a função SEND_SNAP coloca *byte* por *byte* em um registrador de transmissão, chamando a função SEND_BYTE para transmitir um único *byte* ou a função SEND_WORD para transmitir mais de um *byte*. No início do pacote, a função SEND_SNAP carrega o *byte* de sincronismo (SYNC) no registrador de transmissão e chama a função SEND_BYTE.

No passo 23, a função SEND_BYTE alterna os *bits* na interface de saída, transfere uma constante (TEMPO_BIT) do valor do tempo de um *bit* (vide tabela 1 da página 32) em um registrador de tempo e chama a função TEMPO. No passo 24, a função TEMPO gera um tempo de espera baseado no valor da constante que estava no registrador de tempo e retorna para a função SEND_BYTE.

No passo 25, tendo a função SEND_BYTE repetido os passos 23 e 24 para todos os *bits* do *byte* de sincronismo, ela retorna para a função SEND_SNAP. No passo 26, a função SEND_SNAP vai apontar um ponteiro para o primeiro *byte* do cabeçalho (HDB2), colocar o número 2 em um registrador e chamar a função SEND_WORD. O número 2 indica para a função SEND_WORD que existem dois *bytes* a serem transmitidos.

No passo 27, a função SEND_WORD chama a função SEND_BYTE para enviar o primeiro *byte* (passos 23 e 24), incrementa o endereço do ponteiro, decrementa o registrador que continha o número de *bytes* a ser transmitidos e chama função SEND_BYTE novamente. Este procedimento continua até que este registrador de *bytes* transmitidos seja zero.

No passo 28, a função SEND_BYTE retorna para a função SEND_WORD depois de ter efetivamente transmitido todos os *bits* do *byte* apontado pelo ponteiro na interface de saída.

No passo 29, a função SEND_WORD depois de ter transmitido todos os *bytes* através do método descrito no passo anterior, retorna para a função SEND_SNAP. A função SEND_SNAP, por sua vez, vai apontando o ponteiro para todos os endereços dos *bytes* do pacote e chamando as funções de transmissão SEND_BYTE e SEND_WORD, repetindo os passos de 22 a 29 até o *byte* de CRC-8 ter sido enviado.

No passo 30, a função SEND_SNAP retorna para a função TX_SNAP, que verifica, através dos *bits* de controle (RUN_TX=1, RUN_RX=1 e RUN_RTX=1), que a parte de recepção estava sendo executada antes da parte de transmissão. A função TX_SNAP não reabilita, por esse motivo, uma nova interrupção.

No passo 31, a função TX_SNAP retorna para a função RX_SNAP após zerar os *bits* de controle RUN_RTX e RUN_TX. No passo 32, a função RX_SNAP retorna para a aplicação principal depois de zerar seu *bit* de controle RUN_RX.

Assim termina o fluxo do ciclo de recepção de um pacote de comando com confirmação de recepção (*acknowledge*).

3.4.3 - Implementação das funções envolvidas

Para explicar como as funções mostradas no fluxo foram implementadas, foi escolhido o método de apresentação de código em pedaços para que os detalhes ficassem mais evidentes. Todo o código da implementação pode ser encontrado nos apêndices finais desta dissertação.

Como o microcontrolador PIC da Microchip [30] tem como filosofia possuir um único registrador de trabalho chamado de W (*work register*), um mapa de memória foi criado para possibilitar o tratamento das informações e a organização do pacote de dados. Este arquivo foi chamado de SNAP.INC e será apresentado a seguir.

```
            ORG   11h         ;16F84A = 11h / 16F88 = 20h
TEMP        RES   10          ;Registradores temporários

CONTROL     EQU   1Bh         ;Registrador que controla os flags = 1Bh

DAB_MAX     EQU   1Ch         ;Endereço de DAB MSB  = 1Ch
SAB_MAX     EQU   1Dh         ;Endereço de SAB MSB  = 1Dh
DB_MAX      EQU   1Eh         ;Endereço de DB MSB   = 1Eh
EB_MAX      EQU   1Fh         ;Endereço de CRC MSB  = 1Fh
```

Figura 3.17: Inicio do arquivo SNAP.INC contendo o mapa de memória

O arquivo começa pela instrução ORG (na figura 3.17) para indicar que se origina na posição de memória 11h (11 em hexadecimal), pois é uma das primeiras posições da memória RAM. A seguir, um nome (*label*) é atribuído a cada endereço, através da diretiva “EQU", para facilitar a tarefa do programador de lembrar cada posição de memória. Isto também é útil para facilitar mudanças e portar para outros microcontroladores. O “ponto e vírgula”, visto nas figuras que contenham código *assembly*, corresponde ao caractere de comentário, não sendo considerado pelo montador.

```
HDB2        EQU   24h         ;  Header2
HDB1        EQU   25h         ;  Header1

DAB1        EQU   26h         ;  1° byte DAB, n° via software
SAB1        EQU   27h         ;  1° byte SAB, n° via software

DB8         EQU   40h         ;  08° byte data
DB7         EQU   41h         ;  07° byte data
DB6         EQU   42h         ;  06° byte data
```

```
DB5         EQU   43h          ;  05° byte data
DB4         EQU   44h          ;  04° byte data
DB3         EQU   45h          ;  03° byte data
DB2         EQU   46h          ;  02° byte data
DB1         EQU   47h          ;  01° byte data

CRC1        EQU   48h          ;  1° byte CRC
```

Figura 3.18: trecho do arquivo SNAP.INC contendo o mapa de memória

Na figura 3.18, temos o mapa de memória em que o pacote SNAP foi mapeado e completamente definido. Observe que nesta implementação temos 8 (oito) *bytes* de dados. A palavra *byte* no microcontrolador PIC referencia 8 *bits* ou um octeto, pois este é um microcontrolador de 8 *bits*.

Outro trecho que consta no arquivo SNAP.INC diz respeito às definições de constantes, como, por exemplo, a palavra de sincronismo do SNAP que é representada pelos *bits* 01010100b. Assim como o cabeçalho padrão do pacote, que dependendo da situação pode ter alterações em seus *bits*, mas que é carregado pelas partes do programa como uma constante. Assim, não existe a necessidade de que cada parte do programa gere um cabeçalho completo, podendo carregar o cabeçalho padrão e apenas modificar um ou outro *bit* de seu interesse. Essas definições podem ser vistas na continuação do arquivo SNAP.INC na figura 3.19.

```
SYNC              EQU   01010100B   ;  Label

HEADER2           EQU   01010001B   ;  Label (via software)
HEADER1           EQU   00111000B   ;  Label (via software)

MSB               EQU   07h         ;  Label: Bit MSB
LSB               EQU   00h         ;  Label: Bit LSB
```

Figura 3.19: trecho do arquivo SNAP.INC contendo outras definições

O *bit* menos significativo (*Least Significant Bit* – LSB) e o *bit* mais significativo (*Most Significant Bit* – MSB) também foram definidos neste arquivo e aparecem nas funções através dos rótulos (*labels*) MSB e LSB vistos na figura 3.19.

O registrador de controle definido na figura 3.17 assume outras definições mais específicas, armazenando os *bits* de controle do protocolo (*flags*) que podem ser vistos na figura 3.20 a seguir.

```
#define    RUN_RX      CONTROL,0     ; = 1 se rx_snap esta sendo executado

#define    RUN_TX      CONTROL,1     ; = 1 se tx_snap esta sendo executado

#define    RUN_RTX     CONTROL,2     ; = 1 se rx_snap and tx_snap estao
                                  ; sendo executados (primeiro RX, depois TX)

#define    REJ         CONTROL,3     ; REJ = 1 se o pacote for rejeitado

#define    RECEBIDO    CONTROL,4     ; Pacote recebido OK

#define    SAB_PARA_DAB    CONTROL,5     ;SAB=>DAB: 0=NAO ou 1=SIM

#define    ERR         CONTROL,6     ;Foi encontrado erro no pacote recebido

#define    ACK_PKT_TX      CONTROL,7         ;ACK do pacote a ser
                                              ;transmitido
```

Figura 3.20: trecho do arquivo SNAP.INC contendo as definições dos *bits* de controle

Conforme mostrado na figura 3.20, o *byte* CONTROL possui oito *bits* de controle. Os *bits* de controle ficam definidos: o *bit* zero do *byte* CONTROL é zero se a função RX_SNAP não estiver sendo executada e 1 se ela estiver; o *bit* 1 funciona da mesma forma para a função TX_SNAP; o *bit* 2 é igual a 1 se o nó tiver recebido um pacote e está para transmitir uma resposta (*acknowledge*); o *bit* 3 é zero se o pacote for recebido adequadamente pela função RECEIVE_SNAP e 1 se o pacote tiver sido rejeitado; o *bit* 4 é igual a 1 se o pacote que foi enviado teve resposta recebida do nó de destino informando

que não houve erro; o *bit* 5 é igual a 1 se os endereços de origem e destino foram trocados; o *bit* 6 é igual a 1 se houve erro detectado no pacote recebido e zero se nenhum erro foi detectado; e o *bit* 7 é igual a 1 se a aplicação principal tiver requerido que o próximo pacote a ser enviado contenha um pedido de confirmação de resposta (*acknowledge*) e zero se isto não for necessário.

```
#define     RXD   PORTB,0        ;RB0 = Entrada de dados (interrupcao)
#define     TXD   PORTB,2        ;RB2 = Saida de dados

TEMPO_BIT          EQU           104     ;2400bps e fosc=4MHz
TEMPO_MEIO_BIT     EQU   52
```

Figura 3.21: trecho do arquivo SNAP.INC contendo as definições de interface

Na figura 3.21, temos as definições dos pinos de entrada e saída, denominados RXD e TXD, respectivamente. Também é configurada a taxa de comunicação de dados pela porta, que, no caso de 2400 bps, corresponde a 104 contagens da função TEMPO mostrados na tabela 1 da página 32. A constante TEMPO_MEIO_BIT é apenas a metade do valor configurado na constante TEMPO_BIT, útil para que a RECEIVE_BYTE possa ler os *bits*, de acordo com a figura 3.15 da página 34.

A seguir temos a definições dos endereços dos nós feitas no arquivo APPL.INC que está listado na figura 3.22.

```
NODE1     EQU     00000001b     ; Mestre
NODE2     EQU     00000010b     ; Escravo 1
NODE3     EQU     00000011b     ; Escravo 2
NODE4     EQU     00000100b     ; Escravo 3
NODE5     EQU     00000101b     ; Escravo 4
NODE6     EQU     00000110b     ; Escravo 5
NODE7     EQU     00000111b     ; Escravo 6
NODE8     EQU     00001000b     ; Escravo 7
NODE9     EQU     00001001b     ; Escravo 8
NODE10    EQU     00001010b     ; Escravo 9
```

```
NODE11      EQU     00001011b     ;  Escravo 10

MY_ADDR1  EQU     NODE1          ;Endereço atual
```

Figura 3.22: trecho do arquivo APPL.INC contendo o endereçamento dos nós

Uma vez montado o mapa de memória, realizadas as definições necessárias e atribuídos os rótulos adequadamente, fica preparado todo o ambiente para a comunicação de dados e comandos através das funções que estão mostradas no apêndice A. Todas as funções foram colocadas dentro do arquivo SNAP.ASM que inclui também um espaço reservado para a aplicação principal e para a programação dos comandos. A seguir, será colocado a título de exemplo, a explicação da função TX_SNAP.

3.4.3.1 - Função TX_SNAP

A primeira ação da TX_SNAP é desabilitar todas as interrupções para não permitir que outra tarefa possa impedir a transmissão do pacote com o sincronismo correto, pois a função de tempo leva em consideração a execução em tempo real do microcontrolador. Na figura 3.23, podemos ver a desabilitação global das interrupções e o *bit* RUN_TX ser colocado em 1 para indicar que a parte de transmissão está sendo executada. Além disso, a instrução BTFSC testa se o *bit* RUN_RX é igual a 1. Caso positivo, coloca 1 no *bit* RUN_RTX também.

```
BCF         INTCON,GIE        ;DEsabilitacao global de interrupcoes

BSF         RUN_TX      ;RUN_TX=1

BTFSC       RUN_RX      ;Se RUN_RX=0, entao pula 1 linha
BSF         RUN_RTX     ;Se RUN_RX=1, entao RUN_RTX=1 (RX->TX)
```

Figura 3.23: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP

Depois, a função TX_SNAP carrega os cabeçalhos padrão que foram definidos no arquivo SNAP.INC. A seguir, o *bit* RUN_RX é testado, conforme mostrado na figura 3.23. Se RUN_RX for 1, o nó está respondendo um pacote que foi recebido anteriormente. O *bit* mais significativo do ACK é ligado (1) para indicar que o pacote a ser transmitido é de

resposta. O *bit* menos significativo tem seu estado lógico colocado em conformidade com o estado do *bit* ERR. Caso o *bit* ERR seja zero, o pacote recebido não teve erro detectado e o ACK fica com o valor 10b (ver figura 3.4). Caso o *bit* ERR seja 1, o pacote foi recebido com erro e o ACK fica com o valor 11b.

```
BTFSSRUN_RX              ;Testa o bit RUN_RX, se for 1 pula uma linha
GOTO        carrega_ack          ;Se RUN_RX=0 entao checa se ACK é requerido
BSF         HDB2,1               ;Liga o bit de resposta do ACK=1X

BTFSSERR                 ;Testa o bit ERR gerado na recepção, se for 1 pula
BCF         HDB2,0               ;Zerar o bit zero se estiver certo, pois 1 é o padrão
```

Figura 3.24: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP

Caso o *bit* RUN_RX tenha sido igual a 1, o TX_SNAP verifica se já houve a troca do endereço de origem do pacote que foi recebido anteriormente para o endereço de destino do próximo pacote que será transmitido. Isso permite que um comando enviado ao nó possa requerer vários pacotes de resposta, como, por exemplo, se a estação de campo pedir todos os dados coletados na memória pelo nó. Esta troca só poderá ocorrer uma única vez, caso contrário o nó poderia ficar enviando pacotes endereçados para si mesmo, e por isso o *bit* SAB_PARA_DAB é ligado para realizar este controle.

Caso o *bit* RUN_RX tenha sido igual a zero, o TX_SNAP decide continuar normalmente, pois está transmitindo um pacote que não se trata de uma resposta, mas que pode requisitar uma resposta do nó de destino. Assim, caso a aplicação principal tenha ligado o *bit* ACK_PKT_TX, uma resposta foi requerida e o *bit* menos significativo de ACK é mantido no valor 1 que se encontra na constante HEADER2 carregada anteriormente. Caso o *bit* ACK_PKT_TX esteja desligado, nenhuma resposta foi requerida e o *bit* menos significativo do ACK é igualado a zero, conforme a figura 3.4 mostrada na estrutura do pacote (3.4.1) e mostrado na figura 3.25 a seguir.

```
carrega_ack                          ;Apenas na situação RUN_TX=1 e RUN_RX=0
        BTFSS       ACK_PKT_TX       ;Se ACK_PKT_TX=1, pula 1 linha
        BCF         HDB2,0           ;Se ACK_PKT_TX=0, zera ACK=X0
```

```
charge_sab
        MOVLW       MY_ADDR1
        MOVWF       SAB1            ;Se vou TX, colocar endereço do nó no SAB

        CALL apply_edm      ;CALCULA O EDM PARA O PACOTE A SER ENVIADO

        CALL send_snap      ;TRANSMITE O PACOTE
```

Figura 3.25: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Após carregar o endereço do próprio nó, o TX_SNAP assume que a aplicação principal já configurou o endereço de destino DAB1 e os dados a serem enviados nos *bytes* DBi (1<=i<=8) e chama a função APPLY_EDM para calcular o CRC-8 do pacote. Tendo o pacote sido completamente determinado, a TX_SNAP chama a função SEND_SNAP para efetivamente enviar o pacote.

Depois de ter enviado o pacote, o TX_SNAP faz uma nova verificação no *bit* RUN_RX para decidir sobre o fluxo do protocolo a ser tomado. Caso RUN_RX seja igual a 1, então o TX_SNAP foi invocado para responder um pacote para o nó que originou a comunicação. Uma vez respondido (situação do *acknowledge* enviado), a TX_SNAP vai para o seu fim, pois nada mais deve ser feito por ela. Caso RUN_RX seja zero, então o TX_SNAP foi chamado pela aplicação principal para transmitir um pacote. Enviado o pacote, o TX_SNAP verifica se foi pedido uma confirmação sobre o recebimento para o nó de destino. Em caso negativo, também vai para o fim, pois a ação já foi tomada.

```
BTFSS       RUN_RX      ;Se RUN_RX=1, pula 1 linha
BTFSS       HDB2,LSB    ;Se ACK=01, entao pula 1 linha
GOTO        end_txsnap  ;Se RX estiver rodando vai para o fim
```

Figura 3.26: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Caso tenha sido pedida a confirmação (*acknowledge*), o TX_SNAP deve esperar um tempo pelo recebimento do pacote de resposta. Este tempo foi arbitrado em 8 segundos, se decorrido mais do que isso, o pacote é considerado perdido e o *bit* RECEBIDO é igualado

a zero para indicar que o nó de destino não recebeu o pacote ou o pacote de resposta se perdeu na comunicação.

O TX_SNAP coloca 1 no *bit* REJ para indicar que o pacote não foi recebido, habilita as interrupções e inicia o contador de tempo na figura 3.27.

```
        BSF         REJ             ;Inicia com REJ=1
        BSF         INTCON,GIE      ;Habilitação global de interrupções (RX=0)

wait_loop2
        CLRWDT
        BTFSS       REJ             ;Se REJ=1, então pula 1 linha
        GOTO        exit_wait       ;Se REJ=0, então um pacote foi recebido

        MOVF        TMR0,W
        BTFSS       STATUS,Z        ;Quando TMR=0, sai do LOOP2
        GOTO        wait_loop2

exit_wait
        BCF         INTCON,GIE      ;DEsabilitacao global de interrupções
```

Figura 3.27: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Note na figura A.25 da página 114 que quando o RX_SNAP receber adequadamente um pacote, o *bit* REJ vai para o valor zero e isso interrompe a espera do TX_SNAP. Outro fator importante, que vai poder ser visto na função RX_SNAP quando esta for apresentada, é que caso um pacote que não seja de resposta (ACK<>1X) tenha sido recebido, o *bit* REJ continuará sendo 1, pois o protocolo está somente aguardando pacotes de resposta. Caso um outro pacote de resposta tenha sido recebido vindo de outro nó, além daquele que foi o destinatário do pacote originalmente enviado, o *bit* REJ continuará sendo 1. Então, conclui-se que a espera do TX_SNAP somente é interrompida por um recebimento do pacote de resposta vindo do nó para o qual o pacote foi transmitido, o que elimina falsas confirmações advindas do tráfego de pacotes dentro da rede de sensores.

Decorrido o tempo arbitrado ou recebido o pacote de resposta esperado, as interrupções são novamente desabilitadas. Na figura 3.28 podem ser vistas as instruções que decidem a sobre o recebimento correto do pacote pelo nó de destino.

```
        BTFSC       REJ                 ;Se REJ=0, pula 1 linha
        GOTO        nao_recebido        ;Se não houve resposta --> time out
        BTFSS       ERR                 ;Se REJ=0 e ERR=1, pula 1 linha
        BTFSC       HDB2,LSB            ;Se REJ=0, então ACK=10 OU 11
        GOTO        nao_recebido        ;Houve erro detectado na resposta
        BSF         RECEBIDO            ;REJ=0, ERR=0, ACK=10, então OK
        GOTO        end_txsnap

nao_recebido
        BCF         RECEBIDO            ;Pacote não foi recebido adequadamente
```

Figura 3.28: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

O TX_SNAP verifica o estado lógico do *bit* REJ, caso seja 1 temos duas possibilidades: ou não foi recebido o pacote de resposta do nó para o qual ele foi transmitido, ou nenhum pacote foi recebido. Em ambas, o protocolo considera que não existe certeza sobre o recebimento do pacote pelo nó de destino e coloca o valor zero no *bit* RECEBIDO. Caso REJ tenha valor zero, então foi recebido um pacote de resposta do nó para o qual um outro foi transmitido anteriormente e a questão é verificar se houve ou não erro neste pacote de resposta. Tendo havido erro (ERR=1), a hipótese anterior se repete e o *bit* RECEBIDO recebe o valor zero. Caso o *bit* ERR tenha valor zero, então o pacote de resposta foi recebido corretamente, mas falta checar se o cabeçalho deste pacote indica que houve a confirmação de recebimento correto pelo nó de destino do pacote anterior. Isso é verificado no *bit* menos significativo do ACK (ver figura 3.4). Caso ACK=11, então houve erro na recepção e o *bit* RECEBIDO recebe o valor zero. Caso ACK=10, então todas as condições foram cumpridas, o pacote teve confirmação de ter sido recebido corretamente pelo nó de destino e o valor 1 é colocado no *bit* RECEBIDO. Repare que, após transmitir um pacote, só existem duas situações para o *bit* RECEBIDO no fluxo apresentado: ou ele recebe o valor 1, ou recebe o valor zero. A questão é que, sempre após a transmissão de um pacote de dados, a aplicação principal, ao testar o *bit* RECEBIDO, tem uma confirmação sobre o real estado da comunicação.

```
BTFSSRUN_RX             ;Se RUN_RX=1, então pula 1 linha
BSF         INTCON,GIE          ;Habilitação global de interrupções
```

Figura 3.29: trecho do arquivo SNAP.ASM contendo o final da função TX_SNAP

Na figura 3.29 pode ser observado o cuidado de verificar que a função RX_SNAP não está sendo executada através do *bit* RUN_RX, para depois realizar a reabilitação de todas as interrupções. Caso esse cuidado não fosse tomado, a função TX_SNAP estaria invocando um nova interrupção para uma outra instância da função RX_SNAP, provocando a perda dos valores guardados pelo tratador de interrupções que poderia colocar o microcontrolador em estados não previstos e causaria a perda de controle do nó.

Para terminar, a função TX_SNAP limpa os *bits* RUN_RTX e RUN_TX e retorna para a aplicação principal.

3.4.3.2 - Uso de Registradores Temporários

Pela própria arquitetura e funcionamento do microcontrolador PIC, esta implementação do protocolo SNAP Modificado utiliza vários registradores globais temporários. Eles foram denominados de TEMP, TEMP+1, TEMP+2, TEMP+3, TEMP+4, TEMP+5, TEMP+6 e TEMP+7, tendo como finalidade possibilitar que sejam realizados os controles de estado, contagens de eventos e passagens de dados entre as funções.

Para garantir que o conjunto de todas as funções que compõem o protocolo não utilize os mesmos registradores temporários, enquanto estes abrigam dados que serão utilizados por outras funções no decorrer do seu fluxo, foram elaborados dois mapas de registradores, um para cada função fundamental, a transmissão e a recepção. O mapa da função de transmissão pode ser vista na figura 3.30 a seguir.

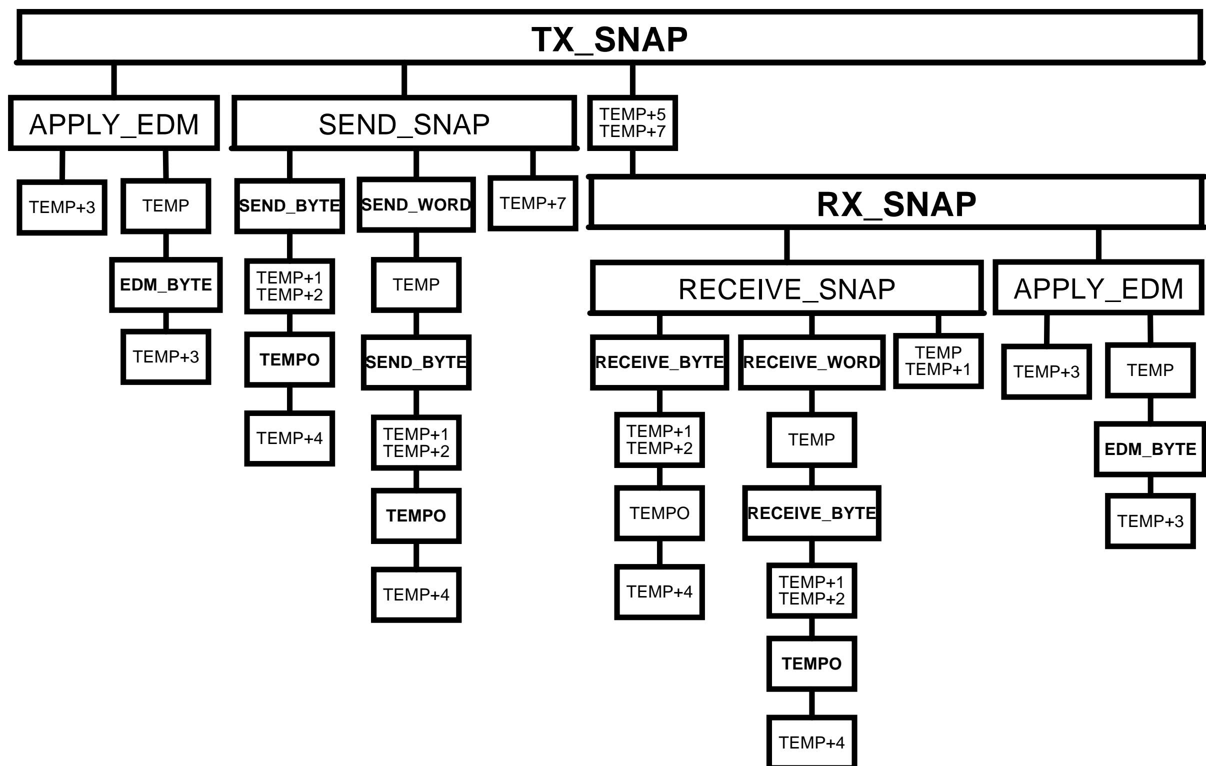

Figura 3.30: mapa de registradores em função do fluxo do protocolo na transmissão

Cada nível indica a utilização de um registrador temporário ou a chamada de uma função. Desta forma, pode ser lido no mapa que a função TX_SNAP chama a função APPLY_EDM, que por sua vez utiliza o registrador TEMP+3. Repare que a função APPLY_EDM termina de usar o registrador TEMP+3 antes de chamar a função EDM_BYTE. Entretanto, um dado é guardado no registrador TEMP e depois a função EDM_BYTE é invocada. Isso acontece porque a função APPLY_EDM precisa memorizar quantos *bytes* de dados ainda precisam ser computados com a função EDM_BYTE. Fica, assim, criada uma dependência na qual a função EDM_BYTE não pode alterar o registrador temporário TEMP, sob pena de causar mau funcionamento da função APPLY_EDM. Como a função EDM_BYTE utiliza apenas o registrador TEMP+3, conforme consta no mapa, não existe o risco do conteúdo de TEMP ser alterado e o correto funcionamento do protocolo fica assegurado. Da mesma forma, podemos visualizar o mapa da função de recepção na figura 3.31 a seguir.

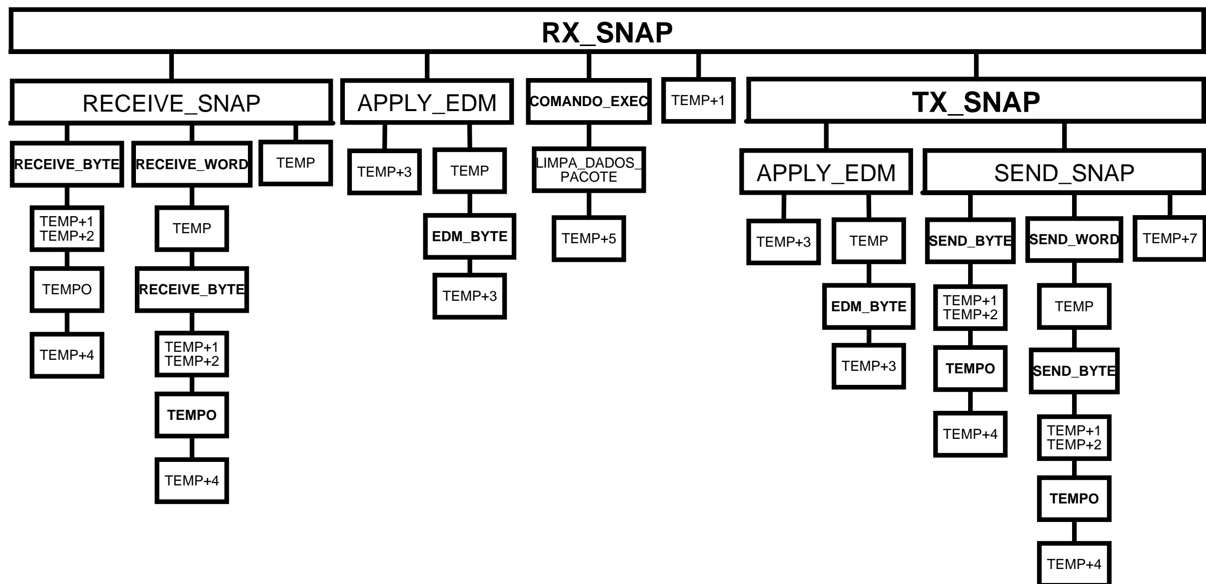

Figura 3.31: mapa de registradores em função do fluxo do protocolo na recepção

Conforme pode ser visto no mapa de registradores da função de recepção, todas as funções envolvidas respeitam os registradores que contém dados das funções da hierarquia superior. Do mesmo modo que foi mostrado no mapa anterior, pode ser visto como exemplo, que a função RX_SNAP chama a função RECEIVE_SNAP. Esta por sua vez chama a função RECEIVE_WORD que vai utilizar o registrador TEMP e chamar a função RECEIVE_BYTE. A função RECEIVE_BYTE utiliza dois registradores TEMP+1 e TEMP+2 para guardar dados temporários e chama a função TEMPO. A função TEMPO apenas usa o registrador TEMP+4. Fica evidente a seguinte conclusão: nenhuma função chamada altera os dados dos registradores temporários das funções anteriores que estejam guardando informações que serão posteriormente utilizadas.

Como representação gráfica, os mapas de registradores são boas ferramentas para auxiliar o programador não cometer violações em conteúdos de outras funções, uma vez que a linguagem *assembly* não acusa este tipo de acesso como nas linguagens de alto nível. Este tipo de recurso também serve para iniciar a validação do código, pois demonstra que os registradores temporários foram utilizados de forma correta na implementação do protocolo, o que eleva os níveis de confiança no seu funcionamento.

Caso seja desejado, ambos os mapas de registradores se encontram no formato de página inteira no Apêndice E no final desta dissertação.

3.5 – CONSIDERAÇÕES FINAIS

A implementação do protocolo no PIC foi muito interessante para que o trabalho pudesse ser testado na prática. As muitas modificações necessárias feitas no código inicial tornaram o código final completamente diferente. Mesmo as inúmeras simulações realizadas com o código final não deixariam a certeza obtida com os testes práticos. Isso se deve principalmente pela oportunidade de testar diversos nós em rede, impossível através das ferramentas computacionais obtidas no sítio da Microchip [30].

A estratégia de se implementar o protocolo no PIC se mostrou correta, pois a experiência adquirida com as conseqüências das alterações no código para os resultados práticos obtidos em uma rede de sensores, se mostraram essenciais para um amplo entendimento do assunto, o que facilitou enormemente a fase seguinte de tradução do código para o RISC16 que será abordada no capítulo seguinte.

4 – TRADUÇÃO DO PROTOCOLO PARA O RISC16

4.1 – CONSIDERAÇÕES INICIAIS

Nesta segunda etapa do trabalho, é realizada a tradução de todo o código gerado para o PIC no capítulo 3. Esse código teve como base a implementação de Castricini e Marinangeli [31], que depois de ter passado por todas as modificações necessárias a fim de atender as especificações desejadas, culminou no protocolo SNAP Modificado, objeto desta dissertação. Todas as premissas básicas e a metodologia da implementação foram as mesmas do capítulo anterior.

O novo código traduzido deve realizar as mesmas tarefas do PIC no microcontrolador RISC de 16 *bits*, que está sendo desenvolvido pela Universidade de Brasília (UnB). Para tal, as duas grandes dificuldades encontradas foram: a diferença entre os comprimentos das palavras, pois o PIC é um microcontrolador de 8 *bits* e o microcontrolador que está sendo desenvolvido é de 16 *bits*; e a pequena quantidade de instruções disponível para que o novo microcontrolador execute suas tarefas, pois o PIC possui mais que o dobro de instruções do RISC16.

A primeira grande dificuldade considera que todos os registradores do PIC têm 8 *bits* e todos do novo microcontrolador têm 16 *bits*. Isso também se aplica ao mapeamento do pacote do SNAP Modificado na memória, obrigando que algumas modificações tivessem que ser feitas para não subaproveitar o potencial do RISC16, assim como, para não desperdiçar a tão escassa memória.

A segunda grande dificuldade foi que o microcontrolador PIC possui 35 instruções contra as 16 instruções do RISC16. Mesmo para um RISC, esta quantidade de instruções é bastante reduzida, o que contribuiu para tornar o código mais extenso. No entanto, apesar das diferenças de arquiteturas e das instruções de cada microcontrolador, todas as tarefas desempenhadas pelo protocolo no PIC puderam ser plenamente contempladas no RISC16, como pode ser visto a seguir.

4.2 – ORGANIZAÇÃO E ESTRUTURA DOS PACOTES

No microcontrolador RISC16, o pacote é composto de partes de dezesseis *bits* (dois octetos), com exceção do SYNC que é composto de um único octeto. Essa adaptação foi realizada porque as palavras do RISC16 têm 16 *bits*, ao contrário do PIC que tem palavras de 8 *bits*. Isso significa que, deste ponto em diante nesta dissertação, toda vez que a palavra "*byte"* aparecer no texto, ela se refere a dois octetos. Então, seguindo as definições apresentadas, o pacote do protocolo ficou com a estrutura vista na figura 4.1 abaixo.

SYNC[4]	**HDB**	**DAB1**	**SAB1**	**Dbi (1<=i<=8)**	**CRC16**

Figura 4.1: pacote de dados do SNAP Modificado no RISC16

Não houve grandes mudanças do pacote utilizado no PIC, até porque, segundo a metodologia aplicada, o protocolo SNAP Modificado deveria ser apenas traduzido do PIC para o RISC16, não permitindo que quaisquer grandes mudanças funcionais ou estruturais fossem introduzidas. Esta preocupação é necessária porque o RISC16 não estava disponível para os testes práticos. Assim, uma vez que foi comprovado o perfeito funcionamento do protocolo no PIC, o código traduzido para o RISC16 também herdou essa garantia.

A primeira parte do pacote é o octeto SYNC de sincronismo. Conforme explicado anteriormente, esta parte contém apenas um octeto por motivo de compatibilidade com toda a base existente do protocolo SNAP. Apesar das palavras no RISC16 serem representadas com 16 *bits*, nada impede que o octeto de sincronismo possa ser colocado em uma palavra de dois octetos. O único cuidado foi tomado no momento de transmitir ou receber apenas metade da palavra de 16 *bits* que estivesse no início do pacote. Assim, nenhuma biblioteca DLL teve que ser alterada para que o protocolo continuasse operacional. O primeiro octeto foi definido como a seqüência de *bits* 01010100b, a mesma do protocolo SNAP original (vide figura 3.2). O segundo octeto não chegou a ser definido, pois não é efetivamente transmitido nem recebido.

---

[4] O SYNC é padronizado no protocolo SNAP como tendo apenas um octeto.

O cabeçalho do pacote no PIC era dividido em duas partes, HDB2 e HDB1. Estes dois octetos foram unidos em um único *byte* do RISC16, que foi chamado de HDB, conforme pode ser visto na figura 4.2.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	DAB		SAB		PFB		ACK		CMD	EDM			NDB			
HDB	**1**	**0**	**1**	**0**	**0**	**0**	**X**	**X**	**X**	**1**	**0**	**0**	**1**	**0**	**0**	**1**

Figura 4.2: *byte* de cabeçalho do pacote no RISC16

Apesar de manter sua estrutura do cabeçalho igual a da implementação do PIC, os valores numéricos sofreram alterações para que fosse mantida a compatibilidade com o SNAP original. Assim, seguindo as regras do protocolo base que se encontra nos documentos do SNAP [28], temos os *bits* 10b para definir dois octetos de endereço de destino DAB e dois octetos para o endereço de origem SAB. Os *bits* 00b definem que nenhum octeto será usado para os *bits* de controle específico PFB. Os *bits* dos campos ACK e CMD são definidos no decorrer da utilização do protocolo, tendo seu funcionamento idêntico ao explicado anteriormente. Os *bits* 100b definem que o método de detecção de erro é o CRC-16. E finalmente, os *bits* 1001b definem que o pacote contém 16 octetos de dados, conforme consta nos documentos do SNAP [28].

Note que, as alterações feitas em relação ao cabeçalho da implementação no PIC têm como objetivo tratar com 16 *bits* o que antes era tratado com 8, mas evitando o desperdício de memória. Essas alterações melhoraram muitos aspectos do protocolo implementado no PIC. Como pode ser visto nas figuras 4.3 e 4.4, o número de endereços possíveis aumentou de 255 para 65535 (pois o zero corresponde ao endereço de *broadcast*). A quantidade de dados transportados por pacote passou de 8 para 16 octetos. O método de detecção de erros evoluiu do CRC-8 para o CRC-16.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DAB	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 4.3: endereço de destino do pacote no RISC16

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAB	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 4.4: endereço de origem do pacote no RISC16

Assim como na implementação do PIC, os *bytes* de dados úteis (*Data Bytes* ou *payload*) ou comandos também podem ser preenchidos com qualquer valor sem nenhuma restrição, como pode ser visto na figura 4.5 abaixo.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DBi	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 4.5: *byte* de dados (DBi, 1<=i<=8) ou comando no RISC16

A figura 4.6 a seguir ilustra a última parte do pacote do SNAP Modificado no RISC16, o *byte* do método de detecção de erro. Como o *byte* do RISC16 tem dois octetos, utilizar o CRC-8 seria subutilizar a capacidade do microcontrolador. Além de que, segundo Tanenbaum [28], o CRC-16 é muito superior, pois é capaz de detectar todos os erros de um *bit*, dois *bits*, qualquer número ímpar de *bits* e rajadas de erros de comprimento inferior a 16 *bits*. Para as rajadas de erros superiores a 16 *bits*, a chance de reconhecer erros é de 99,998%. Como no CRC-8 da implementação anterior, o cálculo do CRC-16 não considera o octeto de sincronismo.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CRC16	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figura 4.6: *byte* de CRC-16 no RISC16

4.3 – FLUXO DO PROTOCOLO NO RISC16

O fluxo do protocolo no RISC16 segue idêntico ao fluxo mostrado do protocolo para o PIC. Nenhuma função foi acrescida ou retirada, além de que elas permaneceram com os mesmos nomes, não havendo necessidade de repetir toda a explicação do item 3.3.2 novamente. A única consideração que mudou foi que, a palavra do PIC tem 8 *bits* e a do RISC16 tem 16 *bits*. Assim, na tradução do código *assembly* do PIC para o código

assembly do RISC16, as tarefas desempenhadas foram as mesmas, embora as instruções fossem diferentes. Houve uma enorme preocupação em realizar uma tradução que fosse fiel *bit* por *bit* nas execuções das tarefas. Essa abordagem de trabalho garantiu que o protocolo pudesse ser testado na prática, uma vez que o PIC estava disponível e o RISC16 não.

4.4 – IMPLEMENTAÇÃO NO RISC16

Da mesma forma que foi feito para o PIC, algumas funções serão explicadas neste capítulo, enquanto as outras partes do código estão explicadas no Apêndice B. O código final na sua forma completa pode ser encontrado no apêndice F desta dissertação.

O PIC tem como filosofia possuir um único registrador de trabalho W, tendo que construir um mapa de memória para tratar o pacote e organizar as suas informações. Já o RISC16 tem 16 registradores de trabalho, o que facilita em muito as operações feitas pelo microcontrolador, pois ele não tem que ficar o tempo todo carregando dados em um único registrador para conseguir trabalhar. No entanto, o pacote do SNAP Modificado também não cabe inteiramente nesses registradores do RISC16. Até porque alguns são para uso do próprio microcontrolador e não estão disponíveis para o usuário colocar dados. A solução foi criar um mapa de memória no RISC16 igual ao realizado para o PIC. O arquivo do PIC que contém esse mapa foi chamado de SNAP.INC, então o arquivo do RISC16 ficou com o nome de SNAP16.INC que se encontra na figura 4.7 abaixo.

```
ORG   $0002
TEMP         EQU   $02            #Endereços para registros temporários
TEMP3        EQU   $03
TEMP7        EQU   $04
```

Figura 4.7: trecho inicial do arquivo SNAP16.INC

A primeira instrução informa ao montador que o mapa de memória criado se inicia do endereço 2h. A diretiva “EQU" atribui um rótulo (*label*) para as posições de memória. Assim, o registrador temporário TEMP corresponde ao endereço 2h, o registrador TEMP3 corresponde ao endereço 3h e o registrador TEMP7 corresponde ao endereço 4h. Note que os números possuem o caractere “$” na sua frente para que o montador possa interpretar

corretamente. E o caractere “#” indica que qualquer texto que vier após a sua ocorrência é um comentário, não devendo ser considerado como parte integrante do código *assembly*. Essa sintaxe foi definida na dissertação de mestrado de Benício [12].

Seguindo a montagem do mapa de memória, a figura 4.8 a seguir mostra os endereços dos *bits* de controle do SNAP Modificado.

```
RUN_RX       EQU  $05       #RX_SNAP esta sendo executado
RUN_TX       EQU  $06       #TX_SNAP esta sendo executado
RUN_RTX      EQU  $07       #primeiro RX_SNAP, depois TX_SNAP
REJ          EQU  $08       #pacote rejeitado
RECEBIDO     EQU  $09       #pacote recebido pelo nó de destino
SAB_PARA_DAB      EQU  $0A  #houve troca de endereços
ERR          EQU  $0B       #pacote recebido com erro
ACK_PKT_TX        EQU  $0C  #pede confirmação no pacote a ser TX
OCTSYNC      EQU  $0D       #Octeto de sincronismo
TX_RF        EQU  $0E       #Transmissão por RF se TX_RF>0
TX_RF2       EQU  $0F       #Backup temporário do registrador TX_RF
```

Figura 4.8: trecho do arquivo SNAP16.INC que define os *bits* de controle do protocolo

No PIC esses *bits* de controle ficavam agrupados em um único *byte*, pois a verificação de um *bit* para a tomada de uma decisão, ou a mudança de qualquer *bit* dentro de um *byte*, era facilmente conseguida através das instruções BCF, BSF, BTFSC e BTFSS. No RISC16, essas instruções simplesmente não existem. Colocar todos os *bits* de controle mostrados na figura 3.20 em um único *byte* seria possível, mas causaria um enorme aumento do código, pois toda a vez que fosse preciso operar com *bits*, seria necessário carregar máscaras combinadas com as instruções AND e OR. Se considerado que, nenhuma instrução do RISC16 consegue carregar uma máscara de 16 *bits* para um registrador em uma única operação, sendo preciso de pelo menos duas instruções para realizar tal tarefa, fica evidente que o código aumentaria ainda mais. Na verdade, esse aumento no código seria proporcional ao número de eventos que utilizassem os *bits* de controle. Sendo assim, por serem amplamente utilizados no controle do fluxo do protocolo, optou-se pela consignação de um *byte* inteiro para cada *bit* de controle. Dessa forma, as instruções LW, SW, BEQ e

BLT cumpriram as mesmas tarefas do PIC com o uso de poucas linhas de código, tornando-o mais otimizado, como poderá ser visto a seguir.

Na figura 4.9 abaixo está mostrado como ficou mapeado o pacote do SNAP Modificado na memória.

```
HDB1      EQU  $10      #Cabeçalhos do pacote (HDB2 e HDB1)
DAB1      EQU  $11      #Endereço do nó de destino
SAB1      EQU  $12      #Endereço do nó de origem
DB8       EQU  $13      #Palavras de dados
DB7       EQU  $14
DB6       EQU  $15
DB5       EQU  $16
DB4       EQU  $17
DB3       EQU  $18
DB2       EQU  $19
DB1       EQU  $1A
CRC1      EQU  $1B      #Resultado do Método de detecção de erro
```

Figura 4.9: trecho do SNAP16.INC que mapeia os *bytes* do pacote na memória

Interessante notar que os rótulos designam um endereço de 8 *bits*, ao invés de 16. Isso ocorre porque os endereços estão no início da memória e seus 8 primeiros *bits* são iguais a zero.

Depois de mapeado o pacote na memória, existe outro trecho que define as constantes utilizadas (vide figura 4.10).

```
SYNC      EQU  $54      #01010100b    Octeto de sincronismo
HEADER2   EQU  $A1      #10100001b  #Cabeçalho SNAP Modificado
HEADER1   EQU  $49      #01001001b
```

Figura 4.10: início do trecho do SNAP16.INC que define as constantes do protocolo

O octeto de sincronismo é igual ao definido no SNAP original e realmente nunca muda, já os *bits* do cabeçalho são carregados como uma constante, mas podem ter os *bits* do ACK e CMD alterados antes do pacote ser transmitido. Na verdade, o cabeçalho na figura 4.10 continua dividido em duas constantes de um octeto cada. Como não existe no RISC16 uma instrução que consiga carregar os 16 *bits* de uma só vez para um registrador, todas as constantes com esse comprimento devem ser divididas para serem carregadas em grupos de 8 *bits*. Uma vez carregada a constante na memória, ela fica armazenada em um único endereço de 16 *bits*.

A seguir, foram definidos os endereços das interfaces de entrada e saída, conforme pode ser visto nas figuras 4.11 e 4.12.

```
SERup    EQU  $FF    #Endereço FFFAh de configuração da serial
SERlo    EQU  $FA
RXDup    EQU  $FF    #Endereço FFF7h de recepção da serial
RXDlo    EQU  $F7
TXDup    EQU  $FF    #Endereço FFF8h de transmissão da serial
TXDlo    EQU  $F8
```

Figura 4.11: trecho do SNAP16.INC que define os endereços da interface serial

```
RF1up    EQU  $FF    #Endereços FFFFh e FFFEh que
RF1lo    EQU  $FF    #configuram o módulo de RF
RF2up    EQU  $FF
RF2lo    EQU  $FE
RXFup    EQU  $FF    #Endereço FFFBh de recepção da RF
RXFlo    EQU  $FB
TXFup    EQU  $FF    #Endereço FFFCh de transmissão da RF
TXFlo    EQU  $FC
```

Figura 4.12: trecho do SNAP16.INC que define os endereços da interface de RF

Ambas as interfaces possuem rótulos, que referenciam os endereços reais das interfaces, para possibilitar que mudanças feitas no *hardware* do microcontrolador possam ser rapidamente assimiladas por este protocolo. Como todas as referências a endereçamentos

de interfaces no código do protocolo são feitas através dos rótulos, caso haja alguma mudança no endereço dessas interfaces, basta trocar os valores definidos para elas no arquivo SNAP16.INC. Lembrando que as constantes de 16 *bits* são divididas em octetos para carregamento nos registradores, e que se convencionou nesta divisão colocar os subscritos "up" e "lo" para se referir aos octetos mais e menos significativos, respectivamente.

Repare que a interface serial é configurada com um único *byte*, mas a interface de RF é configurada com dois *bytes*. Essa característica tem explicação na construção do hardware do microcontrolador, que precisou de mais *bits* do que a interface serial para que a comunicação via RF pudesse ficar bem determinada.

No último trecho do código do SNAP16.INC foram definidas as constantes calculadas para uma comunicação em uma taxa constante de 9600 bps. Os cálculos serão mostrados posteriormente na função TEMPO. Como a velocidade do microcontrolador altera o tempo de espera de uma determinada contagem, foram calculadas constantes para o funcionamento tanto em 200 MHz quanto em 250 MHz. Na figura 4.13 podem ser vistas as constantes para 200 MHz.

```
TEMPO_BITup          EQU   $08   #Configurado para 200MHz
TEMPO_BITlo          EQU   $23
TEMPO_MEIO_BITup     EQU   $04   #Metade do valor de um bit
TEMPO_MEIO_BITlo     EQU   $11
```

Figura 4.13: trecho do SNAP16.INC que define a taxa de 9600 bps para comunicação

Note que são configuradas as constantes do tempo de um *bit* e as constantes do tempo de meio *bit*. O tempo de meio *bit*, como já foi explicado anteriormente, é utilizado na leitura dos *bits* que chegam pela interface, para que o reconhecimento dos *bits* possa ser feito no meio de um *bit*. O valor da constante do tempo de meio *bit* é configurado para que o microcontrolador não precise realizar a operação de divisão.

A seguir será mostrado as definições de endereço dos nós feitas no arquivo APPL16.INC, conforme está mostrado na figura 4.14 a seguir.

```
NODE1up     EQU   $00    #Estação de Campo
NODE1lo     EQU   $01

NODE2up     EQU   $00    #Nó 1 = endereço 0002h
NODE2lo     EQU   $02

NODE3up     EQU   $00    #Nó 2 = endereço 0003h
NODE3lo     EQU   $03

MY_ADDR1up        EQU   NODE2up     #Endereço do Nó 1 (0002h)
MY_ADDR1lo        EQU   NODE2lo
```

Figura 4.14: trecho do APPL16.INC que define os endereços dos nós

O endereço 0001h ficou designado para a estação de campo, o 0002h para o nó 1 e o 0003h para o nó 2. E assim por diante, podem ser definidos 65535 endereços. Lembrando que os endereços de 16 *bits* são separados em octetos porque o RISC16 carrega seus registradores em blocos de 8 *bits*.

Da mesma forma que foi feita no PIC, todas as funções foram colocadas dentro do arquivo SNAP16.ASM que inclui um espaço para que seja desenvolvida a aplicação principal e para a programação dos comandos. Logo no início do arquivo SNAP16.ASM pode ser vista como foi organizada o resto da memória. Na figura 4.15 estão listadas as primeiras instruções e diretivas.

```
        ORG   $0000          #Endereço chamado quando interrompido
        J     tratador       #Pula para o tratador das interrupções

        ORG   $0001
        J     start          #Pula para as configurações iniciais
#////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        ORG $001C            #TRATADOR DE INTERRUPCAO
tratador                     #Mapa de memória termina no end. 1Bh (snap16.inc)
```

```
...       (instruções do tratador)
#////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
start                               #Inicio das configurações do protocolo
```

Figura 4.15: trecho inicial do SNAP16.ASM que mostra a organização da memória

Como o microcontrolador, ao ser ligado, começa executando a instrução do endereço 0001h, este teve que receber a instrução J (*jump*) de salto incondicional, pois um espaço de memória teve que ficar reservado para o tratador de interrupções. Isto ocorre porque a arquitetura do RISC16 reservou apenas um *byte* para o tratador de interrupções no endereço 0000h. Assim, é indispensável que a instrução deste endereço também seja a de desvio incondicional. Como o mapa de memória do pacote começou pelo endereço 0002h e terminou no endereço 001Bh, o endereço do tratador de interrupções ficou sendo 001Ch. Para ilustrar melhor o mapa geral de memória, foi montada na figura 4.16.

Endereço (h)	
0000	J tratador
0001	J start
0002 a 001B	Mapa pacote
001C a XXXX	tratador
XXXX + 1 a YYYY	“start” Habilita inter. e configura protocolo
YYYY + 1 a ZZZZ	Aplicação Principal
ZZZZ + 1 em diante	Funções do protocolo

Figura 4.16: mapa geral de memória do microcontrolador RISC16

Os endereços identificados como XXXX, YYYY e ZZZZ são endereços relativos que serão calculados pelo montador no momento em que o arquivo binário executável é gerado. Eles foram incluídos no mapa geral de memória para proporcionar uma visão geral da

organização da memória, não tendo a intenção de mostrar os endereços exatos da implementação.

Uma vez apresentado o mapa de memória e definidas as constantes necessárias, ficou preparado o ambiente do protocolo para receber as funções que serão mostradas a seguir.

4.4.1 - Função TX_SNAP

Da mesma forma que a implementação do PIC, a primeira tarefa da função TX_SNAP é desabilitar todas as interrupções para não permitir que outras tarefas possam impedir a transmissão do pacote com o sincronismo correto. Caso as interrupções não fossem desabilitadas, qualquer atendimento aos eventos que ocorressem destruiria o pacote, faria com que o microcontrolador não percebesse que o pacote não teria sido enviado e depois ocuparia a interface de saída com *bits* que não fariam mais sentido. Sem mencionar o tempo a mais que o microcontrolador teria perdido, pois estaria executando uma função que não conseguiu cumprir com sua tarefa.

Na figura 4.17 a seguir pode ser visto a desabilitação das interrupções. Note que esta tarefa que consumiu uma única instrução no PIC, teve que ser desempenhada por muito mais instruções no RISC16. Primeiro, porque o RISC16 não possui um *bit* de configuração que possa desligar ou ligar todas as interrupções simultaneamente. Segundo, porque o RISC16 precisa desligar apenas um *bit* de cada *byte* e não possui instruções que tratem de *bits*.

```
LUI     $s1, $FE            #DESABILITA INTERRUPCAO SERIAL
ADDI    $s1, $FF
LUI     $s2, SERup
ADDI    $s2, SERlo
LW      $s0, $s2, $zero     #Carrega conteúdo config. serial em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da serial
SW      $s0, $s2, $zero     #Salva conteúdo na config. serial

LUI     $s1, $FF            #DESABILITA INTERRUPCAO RF
ADDI    $s1, $BF
LUI     $s2, RF2up
```

```
ADDI    $s2, RF2lo
LW      $s0, $s2, $zero     #Carrega conteúdo config. RF em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da RF
SW      $s0, $s2, $zero     #Salva conteúdo na config. RF
```

Figura 4.17: trecho da função TX_SNAP que desabilita as interrupções do RISC16

A primeira vista se tem a impressão que o protocolo SNAP Modificado no RISC16 deve ficar com um número muito maior de instruções. No entanto, apesar de haver muito menos instruções no RISC16 do que existem no PIC, esse efeito de expansão fica muito mais aparente quando o código precisa lidar com *bits* isolados, não caracterizando um crescimento demasiado, além do previsto, no tamanho total do código.

Depois de desabilitar as interrupções, o *byte* RUN_TX precisa ter seu valor maior que zero para indicar que a função TX_SNAP está sendo executada. Além disso, a função testa se o *byte* RUN_RX é maior que zero. Caso afirmativo, qualquer valor positivo é gravado no *byte* RUN_RTX para indicar que a função RX_SNAP foi executada antes da função TX_SNAP. Ambas as tarefas podem ser vistas na figura 4.18 a seguir.

```
AND     $t0, $zero, $t1     #RUN_TX>0 para indicar que TX esta
ADDI    $t0, RUN_TX         #sendo executado
SW      $t1, $t0, $zero

AND     $t0, $zero, $t1     #SE RUN_RX>0, então RUN_RTX>0
ADDI    $t0, RUN_RX
LW      $t1, $t0, $zero
AND     $t2, $zero, $t0
ADDI    $t2, RUN_RTX
BEQ     $t1, $zero, $1      #$t1 tem o valor de RUN_RX
SW      $t0, $t2, $zero
```

Figura 4.18: trecho da TX_SNAP que configura os controles RUN_TX e RUN_RTX

Repare que as figuras 4.17 e 4.18 acima realizam as mesmas tarefas que foram vistas na figura 3.23 no PIC. Novamente, o que o PIC fez com 4 instruções, o RISC16 precisou de

24 instruções para implementar. Por possuir um número muito menor de instruções, o RISC16 realmente gasta muito mais memória para implementar a mesma função. Entretanto, no decorrer desta dissertação, poderá ser visto que esse aumento foi significativo, mas não impeditivo de que o código completo coubesse na escassa memória do RISC16.

Depois, a função TX_SNAP carrega o cabeçalho padrão que foi definido no arquivo SNAP16.INC. Conforme pode ser visto na figura 4.19 abaixo, o *byte* RUN_RX é testado para saber se ele contém um valor maior que zero. Caso RUN_RX seja positivo, o nó está respondendo um pacote que foi recebido anteriormente. O *bit* mais significativo do ACK (*bit* 9 do HDB) tem seu valor alterado para 1, indicando que o pacote a ser transmitido é de resposta. O *bit* menos significativo do ACK (*bit* 8 do HDB) tem seu estado lógico definido de acordo com o estado lógico do *bit* ERR.

```
LUI     $a0, $02            #LIGA O BIT DE RESPOSTA DO ACK
OR      $t2, $a0, $t2       #$t2 continua tendo o valor do header
SW      $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1

AND     $a0, $zero, $t1     #SE ERR=1 NAO ZERA ULTIMO BIT ACK
ADDI    $a0, ERR
LW      $a1, $a0, $zero
BLT     $zero, $a1, $4      #Se ERR>0, então pula 4 linhas
LUI     $a0, $FE            #ZERA O ULTIMO BIT DO ACK (ACK=10)
ADDI    $a0, $FF
AND     $t2, $a0, $t2       #$t2 continua tendo o valor do header
SW      $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1
```

Figura 4.19: trecho da TX_SNAP que configura os *bits* de *acknowledge*

Caso o *byte* RUN_RX seja maior que zero, a função TX_SNAP verifica se o pacote que está sendo enviado de resposta é o primeiro. Este teste é importante porque apenas no primeiro pacote de resposta que o endereço de origem do pacote recebido anteriormente deve ser transferido para o endereço de destino do pacote a ser enviado. Esse mecanismo previne que o endereço de origem do pacote que iniciou a comunicação seja perdido por um comando que exigiu múltiplos pacotes de resposta. Dessa forma, o endereço é trocado

apenas no primeiro pacote de resposta, pois no resto permanece o mesmo endereço de destino. O trecho da função que garante a troca apenas no primeiro pacote de resposta é mostrado na figura 4.20 a seguir.

```
AND     $a0, $zero, $t1     #Verifica SE JA' HOUVE SAB=>DAB (1º)
ADDI    $a0, SAB_PARA_DAB
LW      $t0, $a0, $zero
BEQ     $t0, $zero, $1      #Se SAB_PARA_DAB=0, então pula
J       charge_sab          #Se SAB_PARA_DAB>0 então carrega SAB
AND     $t0, $zero, $t1
ADDI    $t0, SAB1
LW      $t1, $t0, $zero     #Carrega SAB1 em $t1
AND     $t0, $zero, $t1
ADDI    $t0, DAB1
SW      $t1, $t0, $zero     #SAB1 => DAB1
ADDI    $t1, $0F            #para garantir que $t1>0
SW      $t1, $a0, $zero     #Grava SAB1+0Fh p/ SAB_PARA_DAB>0
J       charge_sab
```

Figura 4.20: trecho da TX_SNAP que verifica se é o primeiro pacote de resposta

Caso o *byte* RUN_RX seja igual a zero, o pacote a ser transmitido não é de resposta. A função TX_SNAP vai carregar os *bits* de *acknowledge* padrão e verificar se a aplicação principal ligou o *byte* ACK_PKT_TX. Este *byte* indica se a aplicação principal deseja ou não que o *bit* de *acknowledge* seja ligado. A função TX_SNAP vai ser coerente com a aplicação principal e alterar os *bits* de *acknowledge* do pacote a ser transmitido.

Neste ponto, é suposto que a aplicação principal tenha configurado o *byte* do endereço de destino DAB1. Pois, como o pacote a ser transmitido não é de resposta, a aplicação principal deve saber para quem transmitir o pacote.

Depois, pode ser visto na figura 4.21 que a função TX_SNAP se encarrega de carregar o endereço do nó corrente no endereço de origem do pacote SAB1. Tendo quase todo o pacote definido, o próximo passo é calcular o método de detecção de erro (EDM) e incluir

no final do pacote a ser transmitido. Assim, com o pacote completamente definido, a função TX_SNAP envia o pacote efetivamente através da função SEND_SNAP.

```
charge_sab
        LUI         $t0, MY_ADDR1up   #SE VOU TX, COLOCAR END. NO SAB1
        ADDI        $t0, MY_ADDR1lo
        AND         $t1, $zero, $t0
        ADDI        $t1, SAB1
        SW          $t0, $t1, $zero     #MYADDR1(up+lo)=>SAB1

        JAL         apply_edm           #CALC. EDM P/ O PKT A SER ENVIADO

        JAL         send_snap           #TRANSMITE O PACOTE
```

Figura 4.21: trecho da TX_SNAP que completa o pacote e o transmite

Se o *byte* RUN_RX for positivo, então termina a execução da função TX_SNAP. Caso o *byte* RUN_RX seja igual a zero, a função TX_SNAP vai verificar se houve requisição de resposta do pacote transmitido. Caso tenha havido, a função TX_SNAP vai aguardar pelo pacote de resposta do nó de destino. Caso não tenha havido pedido de confirmação de recebimento (*acknowledge*), a função TX_SNAP retorna para a aplicação principal.

Para aguardar o pacote de resposta, a função TX_SNAP coloca algum valor no *byte* REJ para que este indique inicialmente que a resposta não foi recebida adequadamente. Caso algum pacote tenha sido recebido pelos critérios de recebimento, o valor do *byte* REJ é igualado a zero para indicar que houve coerência do pacote recebido com os filtros básicos para recebimento de pacotes pela função RX_SNAP. O segundo passo é habilitar as interrupções serial e RF (operação contrária do que foi visto na figura 4.17).

A função TX_SNAP inicia dois contadores conjugados para não ficar aguardando indefinidamente por um pacote de resposta, como pode ser visto na figura 4.22 a seguir.

```
wait_loop
        LW          $s3, $s1, $zero
        BLT         $zero, $s3, $1        #Se REJ>0, entao pula 1 linha
        J           exit_wait             #SE REJ=0, um pacote foi recebido, SAI

        ADDI        $s0, $01              #Incrementa o contador1
        BEQ         $s0, $zero, $1        #Se $s0=zero então pula 1 linha
        J           wait_loop

        ADDI        $s2, $01              #Incrementa o contador2
        BEQ         $s2, $zero, $1
        J           wait_loop
exit_wait
```

Figura 4.22: trecho da TX_SNAP que aguarda pelo pacote de resposta

Decorrido o tempo dos contadores, a função TX_SNAP desabilita novamente as interrupções serial e RF tendo ou não recebido um pacote de resposta. Caso o pacote de resposta tenha sido recebido, o fluxo de execução da função não espera até o final das contagens e termina o *loop* de recepção.

Caso o pacote não tenha sido recebido, a função TX_SNAP desvia seu fluxo para o rótulo NAO_RECEBIDO que vai colocar o valor zero no *byte* RECEBIDO. Caso o pacote tenha sido recebido, a função TX_SNAP verifica se houve erro no pacote recebido através dos cálculo de detecção de erro do pacote de resposta. Se houve erro, a função desvia para o rótulo NAO_RECEBIDO. Caso não tenha havido erro, a função TX_SNAP testa se os *bits* do *acknowledge* do pacote recebido confirmam que o pacote é de resposta e se indica que o destinatário recebeu o pacote que iniciou a comunicação sem erros. Caso o pacote recebido seja de resposta e que indique que nenhum erro foi detectado no outro nó quanto a recepção do pacote que iniciou a comunicação, a função TX_SNAP colocar algum valor positivo no *byte* RECEBIDO para indicar para a aplicação principal que a mensagem chegou ao nó de destino sem erros. Todas as instruções que implementam o comportamento descrito podem ser vistas na figura 4.23 a seguir.

```
        AND     $t1, $zero, $t0
        ADDI    $t1, REJ            #End. de REJ=>$t1
        LW      $t0, $t1, $zero
        BEQ     $t0, $zero, $1      #SE HOUVE RESPOSTA (REJ=0), PULA
        J       nao_recebido        #Se não houve resposta --> time out
        AND     $t1, $zero, $t0
        ADDI    $t1, ERR            #End. de ERR=>$t1
        LW      $t0, $t1, $zero
        BLT     $zero, $t0, $7      #Se houve erro, VAI PARA nao_recebido
        AND     $t1, $zero, $t0
        ADDI    $t1, HDB1
        LW      $t0, $t1, $zero
        LUI     %t1, $01
        ADDI    $t1, $00
        AND     $t0, $t1, $t0
        BEQ     $t0, $zero, $1      #SE REJ=0 e ERR=0, TESTAR SE ACK=10
        J       nao_recebido
        AND     $t0, $zero, $t1
        ADDI    $t0, RECEBIDO
        SW      $t1, $t0, $zero     #Se REJ=0, ERR=0, ACK=10, então
                                    # RECEBIDO>0
        J       end_txsnap

nao_recebido
        AND     $t0, $zero, $t1
        ADDI    $t0, RECEBIDO
        SW      $zero, $t0, $zero   #Pacote não foi recebido adequadamente
```

Figura 4.23: trecho da TX_SNAP que verifica o pacote de resposta recebido

O trecho apresentado da função TX_SNAP para o RISC16 (figura 4.23) ficou três vezes maior que o mesmo trecho para o PIC (figura 3.28). Isso aconteceu porque foram necessárias muitas operações sobre *bits* e o RISC16 precisa de mais instruções para gerar o mesmo resultado, pois precisa criar máscaras que possam ser aplicadas sobre os *bytes*.

Na figura 4.24 a seguir, a função TX_SNAP coloca o valor zero nos *bytes* RUN_RTX e RUN_TX. Depois, testa o *byte* RUN_RX para verificar se a função RX_SNAP está sendo executada. Caso negativo, a função TX_SNAP reabilita as interrupções para permitir que novos pacotes possam ser recebidos. Caso positivo, nada é realizado, pois a reabilitação das interrupções causaria um novo atendimento por parte do tratador de interrupções e forçaria que a função RX_SNAP fosse novamente chamada sem ter sido terminada. Fica claro que essa reabilitação indevida tiraria toda a aplicação do fluxo normal, podendo esse comportamento em um efeito recursivo provocar o travamento do nó.

```
end_txsnap
        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RTX
        SW      $zero, $t0, $zero    #RUN_RTX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_TX
        SW      $zero, $t0, $zero    #RUN_TX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RX
        LW      $t1, $t0, $zero
        BEQ     $t1, $zero, $1       #Habilita interrup. se RX não estiver rodando
        J       $ra, $zero
```

Figura 4.24: trecho final da função TX_SNAP

Assim, fica claro nas últimas cinco linhas da figura 4.24 que caso o *byte* RUN_RX contiver algum valor positivo, a instrução BEQ vai deixar que a instrução seguinte seja executada. Caso o valor contido no *byte* RUN_RX seja igual a zero, então a instrução BEQ vai pular a linha que retorna para a aplicação principal e executar a reabilitação das interrupções. Após as interrupções terem sido reabilitadas, uma nova instrução “J $ra, $zero” fará com que a função TX_SNAP termine e obrigatoriamente retorne para a aplicação principal.

4.4.2 - Função TEMPO

A função TEMPO desempenha um dos mais importantes papéis do protocolo de comunicação. Sem ela, os *bits* do pacote não poderiam ser lidos corretamente, e se fossem transmitidos, os *bits* não poderiam ser reconhecidos. A função TEMPO tem como tarefa a geração dos atrasos entre as leituras do *bits* nas interfaces de entrada, ou entre a variação dos *bits* nas escritas dos *bits* nas interfaces de saída.

Como pode ser vista na figura 4.25, a função TEMPO é bem simples e com poucas instruções.

```
tempo               #REGISTRADOR $a0 TEM O VALOR DA ESPERA
        AND         $t1, $zero, $t0
        ADDI        $t1, $01

loop_bit
        SUB         $a0, $t1, $a0       #Decrementa 1 unid do valor lido (4ciclos)
        BEQ         $a0, $zero, $1      #(3ciclos)
        J           loop_bit            #(3ciclos)

        J           $ra, $zero          #RETURN - fim da função TEMPO
```

Figura 4.25: função TEMPO completa que mostra o *loop* de espera com 10 ciclos

Repare que as primeiras duas instruções servem apenas para gravar o valor 1 para que a instrução SUB possa realizar um decremento unitário do valor recebido como parâmetro de espera. Então, o loop de espera tem exatamente 10 ciclos de processamento, de acordo com os dados das especificações da dissertação do Benício [12], pois a função SUB consome 4 ciclos do microcontrolador e as instruções BEQ e J consomem 3 ciclos cada. Em outras palavras, a cada 10 ciclos do microcontrolador é feito um decremento unitário do valor do contador. Como o relógio do microcontrolador funciona a 250 MHz, ele consegue fazer 25 milhões de decrementos unitários por segundo. Se for considerado que a taxa de *bits* fixada para a comunicação é de 9600 bps, o tempo de cada *bit* é 104,166μs. As regras de três, mostradas pelas equações 4.1 e 4.2 a seguir, definiram os valores para o tempo de um *bit* e o tempo de meio *bit* que constam no arquivo SNAP16.INC.

Para 250 MHz:

$$25 * 10^6 \text{ dec} \quad \text{-----------------} \quad 1 \text{ s}$$
$$X \quad \text{-----------------} \quad 104{,}166 * 10^{-6} \text{ s} \qquad (4.1)$$

X = 2604 decrementos

Como o microcontrolador também poderá funcionar a 200 MHz, os mesmos cálculos são feitos a seguir.

Para 200 MHz:

$$20 * 10^6 \text{ dec} \quad \text{-----------------} \quad 1 \text{ s}$$
$$X \quad \text{-----------------} \quad 104{,}166 * 10^{-6} \text{ s} \qquad (4.2)$$

X = 2083 decrementos

Dessa forma, foram calculados os valores das constantes TEMPO_BIT e TEMPO_MEIO_BIT do arquivo SNAP16.INC. Caso a velocidade do relógio do microcontrolador seja alterada, novos valores tem que ser configurados. Lembrando que os valores calculados são da base decimal, e os valores que devem ser configurados devem ser hexadecimais. No caso, para 250 MHz o tempo de um *bit* fica 0A2Ch e para 200 MHz o tempo de um *bit* fica 0823h, que são os equivalentes em hexadecimal dos valores 2604 e 2083, respectivamente.

A figura 4.26 a seguir mostra como os valores são configurados no arquivo SNAP16.INC.

```
#///////////////////////////////////////////////////////////////////////////
#CONFIGURA A VELOCIDADE DA TRANSMISSAO DE ACORDO COM O CLOCK:
#      250 MHz:  9600bps --> Tbit=104us --> TEMPO_BIT= 2604D ou 0A2Ch
#      200 MHz:  9600bps --> Tbit=104us --> TEMPO_BIT= 2083D ou 0823h
#///////////////////////////////////////////////////////////////////////////
TEMPO_BITup          EQU   $08          #Configurado para 200MHz
```

```
TEMPO_BITlo           EQU   $23
TEMPO_MEIO_BITup      EQU   $04          #Metade do valor de um bit
TEMPO_MEIO_BITlo      EQU   $11
```

Figura 4.26: trecho do arquivo SNAP16.INC que é configurado em função do *clock*

Note que os valores são separados em octetos porque não existem instruções no RISC16 que carreguem 16 *bits* de uma única vez.

Depois de terminado o *loop* de espera quando o contador tem seu valor decrementado para zero, a função TEMPO retorna para a função que a chamou.

4.5 – CONSIDERAÇÕES FINAIS

Apesar da dificuldade inicial de traduzir um código de um microcontrolador que possui um maior número de instruções para o RISC16 que possui menos da metade do número de instruções, ficou comprovado que o conjunto de instruções do RISC16 é bastante completo, pois todas as tarefas do protocolo desempenhadas pelo PIC foram plenamente traduzidas para o RISC16 sem perda de funcionalidades.

Embora o gasto de memória tenha sido notavelmente mais elevado para implementar as mesmas funções no RISC16, também deve ser levado em consideração que esse gasto foi dentro do previsto. Era de se esperar que o RISC16 ficaria com seu código pelo menos duas vezes maior que o código do PIC, pois possui menos da metade do número de instruções. Isso apenas não foi verdade quando o protocolo precisou fazer operações com *bits* isolados, situação na qual o RISC16 ficou com o código muito mais extenso por não possuir instruções específicas para lidar com *bits*, sendo necessário constituir máscaras e usar operações lógicas para que pudesse realizar o mesmo trabalho do PIC.

5 – TESTES DE TRÁFEGO

5.1 – CONSIDERAÇÕES INICIAIS

Foi visto no Capítulo 3 como a implementação do protocolo SNAP Modificado foi realizada, assim como o Capítulo 4 mostrou como as instruções foram traduzidas do PIC para o RISC16, mas nenhuma ferramenta computacional estava disponível para simular os testes de tráfego de pacotes entre os nós da rede de sensores. Nem mesmo as ferramentas avançadas do PIC possuem essas funcionalidades.

Para realizar um estudo de tráfego com o protocolo SNAP Modificado, foi necessário desenvolver toda uma metodologia de testes. Primeiro porque a flexibilidade do protocolo permite arquiteturas diferentes para o funcionamento do sistema, e segundo porque algumas idéias que tiveram de ser colocadas em prática culminaram na construção de hardware adicional para que os testes ganhassem credibilidade científica.

Além de ser um protocolo de comunicação que pode ser utilizado em uma infinidade de aplicações, o SNAP Modificado foi testado em duas principais arquiteturas de utilização: a primeira considera o constante monitoramento do ambiente no qual uma mudança significativa nos sensores leva o nó a informar a estação de campo; a segunda usa relações assimétricas de comunicação cujo comportamento funcional é conduzido no modelo mestre/escravo. Nesta segunda arquitetura, a estação de campo organiza o modo como a comunicação é feita através de janelas de tempo bem definidas, o nó apenas transmite para responder a estação de campo. Embora a segunda arquitetura pareça ser mais lógica, existem aplicações em que uma resposta mais rápida é necessária e a primeira arquitetura é mais adequada nessa tarefa. Testes práticos foram feitos em ambas as arquiteturas e serão apresentadas a seguir.

Dada a presente dificuldade de realizar testes com o nó em sua montagem final, devido ao fato dele ainda está em desenvolvimento, foram utilizados processadores PIC16F84A nos resultados que seguem. Como a base do protocolo de comunicação é a mesma, pois possuem as mesmas funcionalidades e a mesma organização, os resultados do PIC podem ser estendidos para o RISC16. Afinal, todos os algoritmos foram refinados e reescritos no

PIC, ganhando as funcionalidades necessárias para o desempenho de suas tarefas, e depois traduzidas para o RISC16 instrução por instrução e *bit* por *bit*.

Kawahara et al. [33] apresentou um estudo semelhante de tráfego de pacotes que mostrou que um meio compartilhado é facilmente saturado com o aumento do número de nós, embora tenha sido conduzido sem o GNA e implementado com o algoritmo *Carrier Sense Multiple Access* (CSMA). Os testes mostrados a seguir foram realizados no intuito de comprovar se os resultados ratificam o estudo de Kawahara [33] ou se apresentam outras características.

5.2 – PRIMEIRA ARQUITETURA – DIRIGIDA POR EVENTOS

Para simular uma rede de sensores da primeira arquitetura, em que os nós comunicam a estação central de mudanças em seus sensores, foi necessário um gerador de tráfego que representasse uma aproximação muito boa de uma rede real de tráfego intenso, causado por condições ambientais em constantes mudanças. Esse gerador de tráfego foi construído baseado na aleatoriedade com que um nó detecta mudanças em seus sensores e transmite um pacote para informar a estação de campo.

Uma importante utilidade que um gerador de tráfego de rede deve possuir é a habilidade de gerar bons números aleatórios. Segundo Knuth [34] (pag. 176), muitas das bibliotecas de sub-rotinas instaladas nos computadores para a geração de números aleatórios são extremamente simples. Programar um bom gerador de números aleatórios pode parecer ser uma tarefa que qualquer programador pode fazer facilmente, mas na realidade não é. A verdade é que os números aleatórios são simulados por algoritmos determinísticos e a exata implementação do conceito é uma tarefa muito árdua.

Toda seqüência de números gerada por *software* em uma máquina de estados finita é pseudoaleatória. Isto foi verificado com muito mais facilidade quando consideramos o microcontrolador de recursos limitados utilizado. Foram testados diversos algoritmos e todos tiveram resultados semelhantes. Ao se repetir os testes práticos, todos os algoritmos testados repetiam os resultados, o que causou um desastre estatístico para a validação dos testes de tráfego. Na verdade, todos os nós sempre transmitiam pacotes nos mesmos momentos porque os números gerados eram pseudoaletórios. Diversas mudanças foram

testadas na tentativa que os nós tivessem transmissões mais aleatórias entre si. Uma dessas tentativas envolveram mudanças nos números sementes dos algoritmos, que ficaram diferenciadas para cada nó individualmente. Essa abordagem pareceu funcionar no início, mas bastou uma análise mais detalhada para perceber que os resultados continuavam a se repetir, haviam colisões de pacotes sempre em determinados momentos, e em outros momentos nunca haviam colisões. Ficou evidente que nenhum algoritmo baseado em *software* poderia garantir que os números gerados fossem realmente aleatórios.

A solução foi construir um gerador de números aleatórios em hardware. Navegando pela Internet não foi difícil encontrar dezenas de projetos, desde os mais simples até os baseados em fenômenos quânticos, como os de decaimento de isótopos radioativos ou de efeitos térmicos. Testando alguns dos mais simples com a ajuda de um osciloscópio, foi selecionado um gerador de números aleatórios baseado no ruído do efeito avalanche em uma junção PN polarizada reversamente. Esse circuito foi apresentado por Logue [35] e pode ser visto na figura 5.1 a seguir.

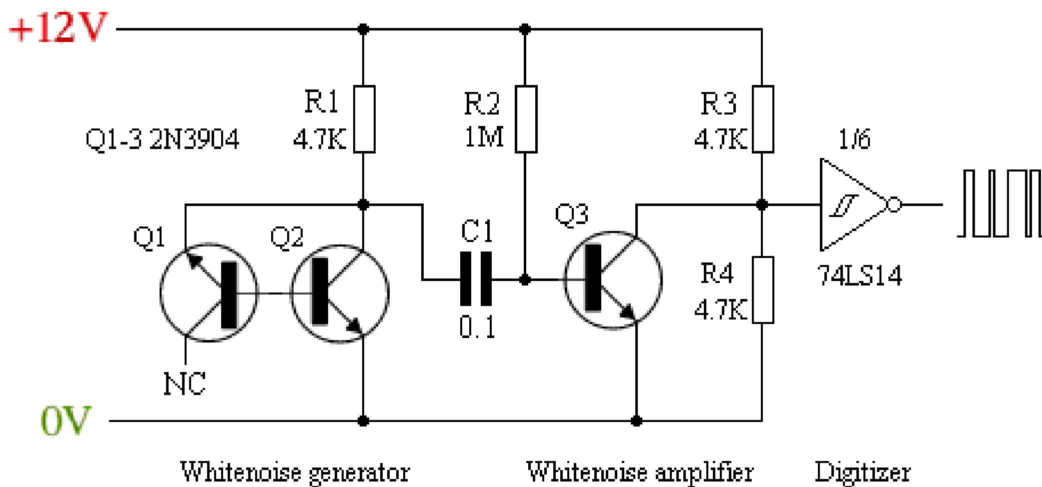

Figura 5.1: Gerador de Números Aleatórios

Este circuito gera a onda mostrada na figura 5.2 a seguir. Apesar de não ser quadrada, a forma da onda pode ser amostrada *bit* por *bit* em um registrador de deslocamento que irá definir um intervalo de tempo imprevisível entre a transmissão de um pacote para o outro. A amostragem *bit* por *bit* é feita com intervalos de tempo para garantir que os *bytes* formados sejam fracamente correlacionados.

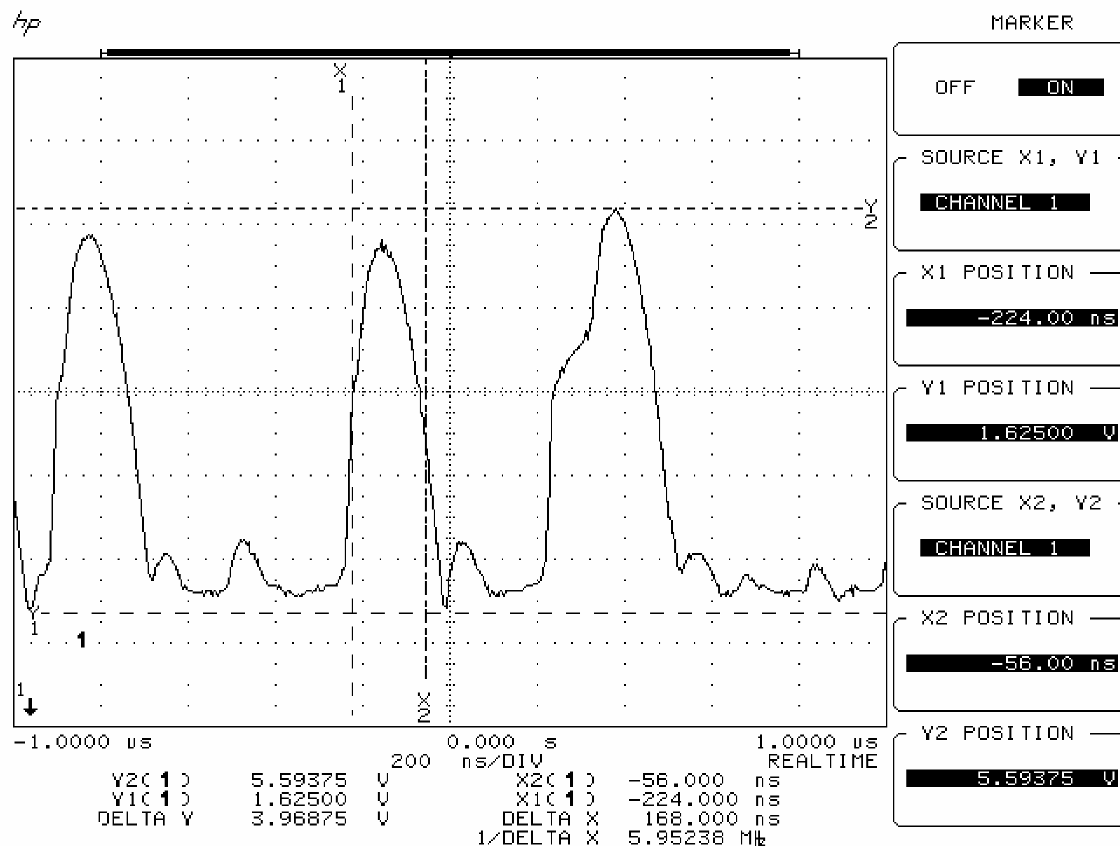

Figura 5.2: forma de onda do Gerador de Números Aleatórios vista no osciloscópio

De acordo com Marsaglia [36], os números gerados pelo circuito foram aprovados quanto a sua aleatoriedade em uma bateria de testes estatisticamente comprovada.

Assim, quando um nó é ligado, ele lê um *byte* através do Gerador de Números Aleatórios (GNA), espera um período de tempo baseado nesse *byte* e transmite um pacote de dados para a estação central. Este procedimento foi codificado na aplicação principal cujo código é mostrado na figura 5.3 a seguir.

```
        MOVLW     255          ;Seqüência de X pacotes para simular trafego
        MOVWF     TEMP+8
envia_pacotes
        MOVLW     1                  ;Transmite um pacote para o nó 1
        MOVWF     DAB1

        CLRF      TEMP+2             ;GNA Recebe byte no TEMP+2 => DB1
```

```
        MOVLW     8
        MOVWF     TEMP+1
le_numero_aleatorio
        CLRWDT
        BCF       STATUS,C          ;Coloca zero no próximo bit
        RRF       TEMP+2,F
        BTFSC     GNA               ;Se a entrada for zero, não faz nada
        BSF       TEMP+2,MSB        ;Se a entrada for um, seta o bit
        MOVLW     TEMPO_BIT         ;Tempo de um bit
        CALL      tempo
        DECFSZ    TEMP+1,F          ;Enquanto houver bits para ler, volta
        GOTO      le_numero_aleatorio
        MOVFW     TEMP+2            ;TEMP+2 = Numero Aleatório
        MOVWF     DB1               ;FIM GNA / joga dados no DB1
;---------------
        BSF       STATUS,RP0        ;Banco 1
        MOVF      OPTION_REG,W
        MOVWF     TEMP              ;OPTION_REG => TEMP
        MOVLW     11000000b
        ANDWF     OPTION_REG,F      ;Limpa seis primeiros bits do byte
        BSF       OPTION_REG,PS2    ;Seta pre-scaler 1/32 no timer
        BCF       STATUS,RP0        ;Banco 0

;  Espero X segundos (X = ALEATORIO vezes 250 vezes 32 / 1MHz)
espera_aleatorio1
        MOVLW     6                 ;faz a contagem do TMR0 começar de 6
        MOVWF     TMR0              ;assim teremos 250 contagens no byte

espera_aleatorio2
        CLRWDT
        MOVF      TMR0,W
        BTFSS     STATUS,Z          ;Pula quando timer for zero
        GOTO      espera_aleatorio2
```

```
        DECFSZ     TEMP+2,F          ;Decrementa numero aleatório
        GOTO       espera_aleatorio1

        BSF        STATUS,RP0        ;Banco 1
        MOVF       TEMP,W
        MOVWF      OPTION_REG        ;Restaura OPTION_REG
        BCF        STATUS,RP0        ;Banco 0
;--------------
        BTFSS      TEMP+8,0          ;ALTERNA ESTADO DO ACK
        BSF        ACK_PKT_TX
        BTFSC      TEMP+8,0
        BCF        ACK_PKT_TX

        CALL       tx_snap;
        DECFSZ     TEMP+8
        GOTO       envia_pacotes
;------------------------------------
begin
        CLRWDT
        NOP        ;Loop infinito para terminar de gerar trafego na rede de sensores

        GOTO       begin   ;  Gera loop infinito
```

Figura 5.3: código do aplicativo principal para a geração de tráfego na rede

Um computador executa um programa que escuta todos os pacotes da rede de sensores e os captura. Esse programa é chamado de *sniffer* e apresenta dados detalhados sobre os pacotes recebidos. Os testes foram repetidos 50 vezes e um valor médio foi tomado. Os resultados são vistos nas figuras 5.4 e 5.5 a seguir.

Note na figura 5.4 que o experimento mostrou que, em média, apenas 2% dos pacotes foram perdidos por colisão quando a rede era composta apenas por dois nós. Quando a rede foi composta por oito nós se comunicando no mesmo meio, 30,5% dos pacotes colidiram e foram perdidos. A curva tem natureza exponencial em função do aumento do número de nós, indicando que um meio compartilhado é facilmente saturado.

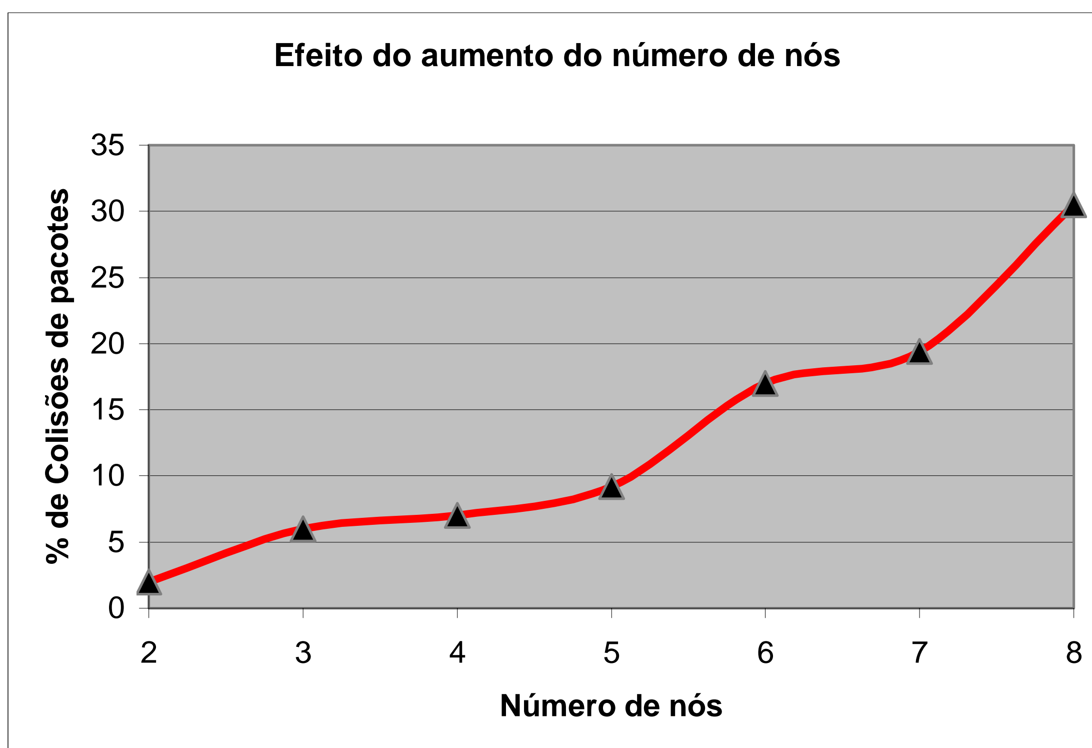

Figura 5.4: efeito do aumento do número de nós

O tamanho do pacote tem um efeito diferente na quantidade de colisões na rede de sensores. O tamanho mínimo do pacote é de 7 *bytes* devido a necessidade de cabeçalhos e do método de detecção de erros para se transmitir apenas 1 *byte* de dados. O tamanho máximo está limitado pela memória RAM do nó, sendo possível até 38 *bytes* por pacote, sendo que 32 *bytes* são de dados. Na medida em que o tamanho do pacote aumenta, a porcentagem de colisões também aumenta, mas uma grande mudança no tamanho dos pacotes apenas reflete uma pequena mudança na porcentagem de colisões. Na figura 5.5 a seguir, pode ser vista essa natureza logarítmica da curva para uma rede composta pelos 8 nós que estavam disponíveis.

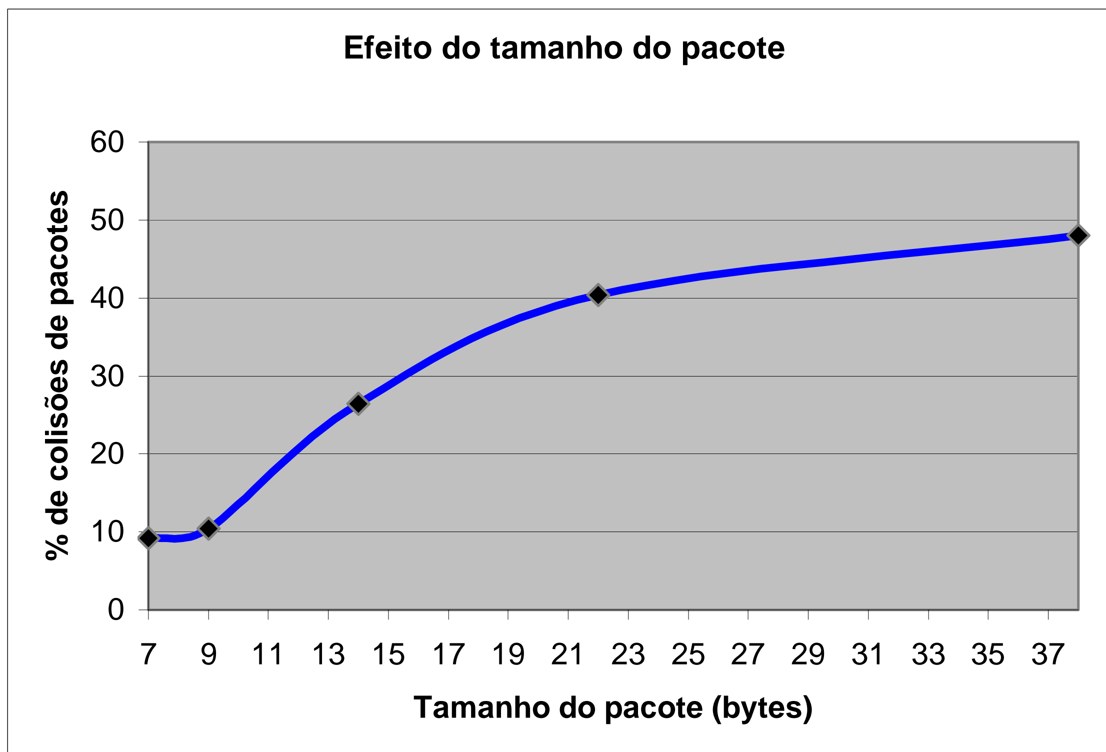

Figura 5.5: efeito do aumento do tamanho do pacote

Analisando os dois gráficos experimentais em conjunto, fica provado que é mais importante quando transmitir um pacote do que o seu tamanho. Esses dados sugerem o uso de estratégias para organizar o modo como um meio compartilhado é acessado pelos nós de uma rede de sensores, no intuito de otimizar a eficiência da comunicação, e consequentemente, aumentar o tempo de bateria de cada nó.

Wan et al. [37] mostram como uma simulação que implementa o algoritmo CODA (*Congestion Detection and Avoidance in Sensor Networks*) melhorou a perda de pacotes quando a rede de sensores é dirigida por eventos. Entretanto, a estratégia adotada é muito complexa para ser adotada no nó sensor do Sistema de Controle de Irrigação, devido a escassa quantidade de memória implementada nesta versão.

5.3 – SEGUNDA ARQUITETURA – RELAÇÃO MESTRE/ESCRAVO

De acordo com Luchine, Picanço e Romariz (2006) [38], uma estratégia que aumenta a eficiência e a confiabilidade de uma rede de sensores é organizar o modo com que os nós acessam o meio através de janelas de tempo (*time slots*). Na prática, essa arquitetura se baseia em tornar a estação de campo o mestre da rede de sensores, e seus nós, os escravos.

Assim, toda a capacidade de memória e processamento pode ser utilizada pela estação de campo para organizar as janelas de tempo de cada nó. Como cada nó deve transmitir apenas em resposta à estação de campo, não existem colisões de pacotes. Nem mesmo quando a estação de campo transmite um pacote de *broadcast*, pois caso este pacote exija uma confirmação de recebimento (*acknowledge*), cada nó responderá em intervalos de tempo determinados pelo número de seu endereço na rede.

Entretanto, esta arquitetura mestre/escravo apresenta deficiências quando o tempo de resposta é um fator crítico. Fica claro que em uma batida de automóvel, o tempo de resposta de cada nó para o acionamento dos *airbags* é menor na arquitetura anterior que é dirigida por eventos do que na mestre/escravo. Zhong et al. [39] colocam que protocolos que usam estratégias de reserva ou agendamento têm uma melhor utilização do canal, porém apenas fazem sentido para eventos periódicos, pois eles não são adequados para ambientes dinâmicos ou móveis. Mas, considerando a aplicação pretendida do Sistema de Controle de Irrigação, a arquitetura mestre/escravo é mais adequada.

Diversos testes de *broadcast* foram feitos com o *acknowledge* ligado e nenhuma colisão foi detectada. Para os testes de comando, foram implementados dois comandos nos nós da rede de sensores: um para ligar um *led*, e outro para desligar o mesmo *led*. Cada nó respondeu adequadamente, sendo que apenas o nó comandado acendeu ou apagou o *led* conforme o pacote enviado pela estação de campo. Também foram testados comandos em *broadcast*, assim, todos os nós da rede acenderam ou apagaram seus respectivos *leds*. Os comandos em *broadcast* com o *acknowledge* ligado mostraram que, após terem acendido ou apagado seus respectivos *leds* simultaneamente, os nós esperaram cada um o seu tempo para transmitir de volta um pacote de resposta indicando que o comando da estação de campo foi executado com sucesso.

Em ambas as arquiteturas, os testes comprovaram o funcionamento do protocolo de comunicação SNAP Modificado na prática, o que valida o trabalho realizado nesta dissertação.

6 – CONCLUSÃO

Com base nas premissas iniciais, pode-se afirmar que o desafio de implementar um protocolo de comunicação no RISC16 foi realizado com sucesso. Mais do que apenas codificar funções, o trabalho elaborado permite um melhor entendimento sobre o assunto. Sendo assim, os objetivos foram alcançados, pois o protocolo de comunicação desenvolvido pode ser generalizado para outras aplicações, cobrindo assim, mais funcionalidades do que a aplicação do Sistema de Controle de Irrigação necessita. A generalização do protocolo foi o objetivo principal, pois a grande motivação deste trabalho era implementar um código que poderia ser reutilizado como uma biblioteca de comunicação. Essa abordagem provê uma camada de abstração, na qual os detalhes específicos da transmissão e recepção de dados ficam escondidos do programador do RISC16. Entretanto, mesmo tendo seus objetivos atendidos, algumas considerações ainda precisam ser feitas, de modo que elas são colocadas a seguir.

Conforme colocado como premissa básica no início do capítulo 3, os comandos deverão ser implementados de forma a suprir as necessidades da aplicação principal. Este protocolo se limita a deixar indicações dentro do código, como incluir comandos e o modo como eles são selecionados através do *byte* DB1. Assim, fica a cargo do desenvolvedor da aplicação principal, escrever o código para incluir os comandos necessários, como foi explicado no item 4.4.9.

Outro aspecto relevante na construção da rede é que os endereços dos nós devem começar pelo menor número possível. Assim, de acordo com a metodologia aplicada para responder os pacotes de *broadcast* (figura B.28), os nós de menor endereço responderão mais rapidamente, uma vez que o tempo de espera cresce com o endereçamento do nó.

A abordagem deste trabalho foi bastante sensata ao considerar que, primeiro se deveria modificar o código *assembly* no PIC, para depois traduzi-lo para o código *assembly* do RISC16 (item 3.3). Dessa forma, foram encontrados muitos problemas práticos de implementação que não ficaram evidentes nas modificações do código em linguagem de baixo nível. Inclusive, valiosas ferramentas de desenvolvimento disponíveis para o PIC,

como um ambiente de produção gráfico de geração de código e simulação de execução das instruções, ajudaram a corrigir alguns erros conceituais e de construção das estruturas antes que pudessem chegar ao código final do RISC16.

Outro ponto que merecia ser levado em consideração era a dúvida que pairava sobre a factibilidade de se traduzir as instruções do PIC para o RISC16. Essa dúvida era sustentada pelas seguintes principais diferenças: um microprocessador é de 8 *bits* e o outro de 16; cada microprocessador pertence a uma arquitetura completamente diferente (um Harvard e outro Von Neumann); e o RISC16 possui um conjunto de instruções em número muito inferior ao do PIC. Entretanto, quando foram traduzidas as primeiras instruções, ficou claro que, com organização e atenção, nenhum tipo de malabarismo teria que ser feito para realizar todas as funcionalidades requeridas para o protocolo funcionar. Nem chegou a ser necessário que fossem utilizadas as pseudoinstruções imaginadas por Linder [14].

O número menor de instruções no RISC16 aumentou consideravelmente o seu código. O código do SNAP Modificado no PIC consumiu 411 instruções, enquanto que no RISC16 foram gastas 784, o que representa um aumento de 90,75% no número de instruções. O aumento não foi maior porque o PIC tem um único registrador de trabalho W, que precisa ser alterado constantemente para fazer parte da execução das instruções, enquanto que o RISC16 consegue trabalhar com seus 16 registradores ao mesmo tempo, o que reduziu a necessidade de ficar transferindo dados desnecessariamente.

Tendo em vista a arquitetura do pacote do protocolo, os únicos *bits* que não puderam ser aproveitados na tradução do PIC para o RISC16 são os 8 *bits* do *byte* (dois octetos) de sincronismo. De outra forma, caso o pacote tivesse 16 *bits* de sincronismo, qualquer outro programa que se comunicasse através do protocolo SNAP original entenderia que o primeiro octeto seria para a sincronia, e o segundo, conteria os primeiros 8 *bits* de cabeçalho do pacote. Logicamente, essa abordagem levaria ao descarte do pacote, destruindo a comunicação. A experiência demonstra que a consideração de utilizar apenas oito *bits* para o sincronismo é bastante acertada, uma vez que quanto mais se garanta a interoperabilidade, mais o padrão se firma no mercado. Afinal, este trabalho nunca tendenciou para a implementação de um protocolo de comunicação que fosse único e ficasse isolado de toda uma base disponível para os dispositivos do protocolo SNAP original.

Quanto ao aspecto da energia, várias considerações precisam ser deixadas. O protocolo de comunicação gerado não desperdiça energia em *handshaking*, pois a comunicação é de forma direta, não existe orientação a conexão, nem desconexão. Apesar do protocolo de comunicação não restringir, a arquitetura do sistema não explora a redundância inerente à própria rede de sensores, pois cada nó tem sua antena direcionada para a estação de campo, não permitindo comunicação inter-nós. Também é complicado de se implementar o mecanismo de comparação entre os dados de nós próximos, para que eles se organizem para enviar apenas os dados de um dos nós quando a correlação é muito alta. Outra forma de economizar energia é fazer com que a aplicação não requeira a confirmação de recebimento (*acknowledge*), este critério é opcional no protocolo para que seja utilizado conforme a necessidade da aplicação. Outra alternativa pode ser implementar na estação de campo, um algoritmo que calibrasse a potência de transmissão do nó, pois segundo os trabalhos do Assunção [40] e do Vogel [41], o microcontrolador do nó pode alterar a potência utilizada na comunicação. Para isso, bastaria incluir um comando no nó para alterar o registrador que controla a potência de transmissão. A estação de campo, que possui potência de transmissão suficiente para ser recebida por todos os nós, faria a calibração de qual seria o menor valor de potência do nó que é capaz de transmitir um pacote para a estação de campo. Outra importante colocação que foi vista na figura 2.2, é analisar muito bem a aplicação para manter os nós no modo *sleep* de economia de energia, o máximo de tempo possível, de forma que não prejudique a medição dos dados. Todas essas considerações colocadas trazem uma economia significativa de energia e prolongam a vida dos nós.

Os testes práticos vistos no capítulo 5 foram importantes para validar todo o trabalho de implementação do protocolo SNAP Modificado. Eles foram realizados na etapa inicial do trabalho com apenas um nó, para que ficasse garantido que o comportamento individual de um nó não oferecesse riscos de estabilidade para a rede de sensores como um todo. Depois, os testes foram feitos com vários nós ligados em rede para garantir que o funcionamento da rede de sensores e o tráfego de pacotes estivessem em conformidade com as características esperadas do protocolo de comunicação implementado. Todas as funcionalidades foram testadas: pacotes de dados, pacotes de comandos, pacotes de dados e de comandos com requisição de confirmação de recebimento (*acknowledge*), pacotes de *broadcast* contendo

dados, pacotes de *broadcast* com comandos, e pacotes de *broadcast* contendo dados e comandos com *acknowledge*.

Além disso, testes de tráfego do capítulo 5 foram feitos considerando que o protocolo SNAP Modificado pode ser utilizado para diferentes arquiteturas de rede: uma arquitetura considera que cada nó pode iniciar a comunicação sempre que este desejar informar algum dado novo ou mudança para outros sensores (dirigida por eventos); outra arquitetura considera que a comunicação na rede é gerenciada por um nó central ao qual os outros nós ficam subordinados (mestre-escravo). Em ambas, o protocolo SNAP Modificado cumpriu suas funções de acordo com o esperado, o que demonstrou a flexibilidade de suas aplicações.

Diversas estatísticas puderam ser apresentadas no final do capítulo 5 em razão da natureza do tráfego mensurado na rede de sensores, sendo que seus resultados foram úteis para a conclusão de que é mais importante quando um nó transmite do que a quantidade de informação do pacote transmitido. Essa conclusão torna evidente que a elaboração de estratégias para evitar a colisão de pacotes é extremamente válida para aumentar a confiabilidade da rede de sensores, assim como para diminuir o gasto de energia por parte dos nós.

Como funcionalidades a serem contempladas no futuro, fica a sugestão de incluir roteamento do pacotes para permitir o modo *ad hoc* das redes de sensores sem fio. Este avanço provavelmente deve ser implementado de forma conjunta com o aumento de memória do microprocessador RISC16, pois a quantidade de memória instalada no RISC16 é suficiente apenas para os *softwares* atuais do protocolo de comunicação, a aplicação principal e o registro de dados dos sensores. Outra funcionalidade interessante é a possibilidade de fazer a comunicação implementar o algoritmo *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA), pois Polastre [42] mostrou que seu protocolo baseado neste algoritmo pode incrementar a banda de uma RSSF de 42 para 53 pacotes por segundo.

Outra tendência para o futuro é colocada por Mitchell [43] que propõe nós com maior capacidade de processamento, memória e banda de transmissão para que uma rede de sensores se torne um cluster de processamento com banco de dados distribuído. Assim, o

processamento de um nó colabora com o resultado dos outros para formar uma resposta mais precisa e mais rápida. Esta visão leva a discussão para outros pontos interessantes, onde as redes de sensores serão mais do que sensores colhedores de dados que se comunicam por protocolos. As RSSF serão partes de um único organismo autônomo que poderá desempenhar tarefas que cada nó sozinho não conseguiria. É como um neurônio, que sozinho não faz nada.

Esta dissertação teve o intuito de apresentar o protocolo de comunicação SNAP Modificado na sua estrutura de pacote, fluxo de funcionamento e implementação em código *assembly* do RISC16. Espera-se que a clareza das explicações seja suficiente para possibilitar futuros trabalhos tanto na área de protocolos de comunicação, quanto no aperfeiçoamento de novas funcionalidades para o SNAP Modificado. Este trabalho também espera ter aberto as portas para as muitas aplicações em que o RISC16, microprocessador inteiramente desenvolvido no Brasil, precisa fazer uso de uma rede de comunicação de dados.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] FALBRIARD, C. **Protocolos e aplicações para redes de computadores**. Primeira edição. São Paulo. Érica LTDA, 2002.

[2] HELD, G. **Comunicação de Dados**. Tradução da sexta edição. Rio de Janeiro. Campus LTDA, 1999.

[3] LAI, R.; Jirachiefpattana, A. **Communication Protocol Specification and Verification**. Primeira edição. EUA. Kluwer Academic Publishers, 1998.

[4] SHANNON, C. E. **A mathematical theory of communication**. Bell System Technical Journal. [S.L.]. 1948.

[5] ISO 9074 - Information Processing – Open Systems Interconnection – **Estelle** – A Formal Description Technique Based on an Extended State Transition Model, 1997.

[6] ISO 8807 - Information Processing – Open Systems Interconnection – **LOTOS** – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, 1989.

[7] MILNER, R. **A Calculus of Communicating Systems**. Lecture Notes in Computer Science, Vol. 92 Springer-Verlag, 1980.

[8] ITU Z.100 - **Specification and Description Language**, 1991.

[9] SARIKAYA, B. **Principles of Protocol Engineering and Conformance Testing**. Ellis Horwood Limited, 1993.

[10] ISO/IEC TR 10167 – Information Technology – Open Systems Interconnection – **Guidelines for the application of Estelle, LOTOS and SDL**, 1998.

[11] MATIC, N. **The PIC Microcontroller**. mikroElektronika. 2003. Livro eletrônico. Disponível em <http://www.mikroe.com/en/books/picbook/1_chapter.htm>. Acessado em 20/06/2004.

[12] BENÍCIO G.M.J. **Projeto de Microprocessador RISC 16-Bit para Sistema de Comunicação sem Fio em Chip.** 2002. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[13] COSTA, J. D. **Implementação de um Processador RISC 16-*bits* CMOS num Sistema em Chip**. 2004. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[14] LINDER, R. **Linguagem de máquina para processador num sistema em chip (SoC)**. 2003. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[15] RANGEL, R. M. D. **Modelagem de dispositivos de RF para a Análise de Compatibilidade Eletromagnética em SoCs CMOS**. 2005. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[16] HALLBERG, B. **Networking: A beginner's guide**. Segunda edição. Berkley, California. McGraw-Hill, 2001. Página 88.

[17] MERHAB. **Rapid Analysis of Pseudo-nitzschia & Domoic Acid, Locating Events in near-Real Time**. 2005? Universidade do Sul da Califórnia, Califórnia, EUA. Disponível em http://www.usc.edu/dept/LAS/biosci/Caron_lab/MERHAB/MERHAB_RAPDALERT.html>. Acessado em 20/09/2006.

[18] TUTWSN. **Ultra Low Power Wireless Sensor Network**. 2004? Tampere University of Technology, Institute of Digital and Computer Systems. Disponível em <http://www.tkt.cs.tut.fi/research/daci/ra_tutwsn_overview.html>. Acessado em 19/09/2006.

[19] FISCHER, S. **Sensor Network Settings for Individual Patient Monitoring**. 2006. Universidade de Lübeck. Disponível em <http://www.itm.uni-luebeck.de/theses/tirkawi/os2.html?lang=en>. Acessado em 19/09/2006.

[20] LUNDQUIST, J.D.; CAYAN, D.R.; DETTINGER, M.D. **Meteorology and Hydrology in Yosemite National Park: A sensor network application**. In: Information Processing in Sensor Networks – IPSN 2003, LNCS 2634. Palo Alto, Califórnia, EUA. Páginas 518 a 528.

[21] KARL, H.; WILLIG, A. **A Short Survey of Wireless Sensor Networks**. TKN Technical Reports Series. Berlin, 2003.

[22] RABAEY, J.M.; AMMER, M.J.; SILVA, J.L.; PATEL, D. **PicoRadio supports ad hoc ultra-low power wireless networking**. Computer, 2000. Volume 33, nº 7, páginas 42 a 48.

[23] WOLENETZ, M.; KUMAR, R.; SHIN, J.; RAMACHANDRAN, U. **Middleware Guidelines for Future Sensor Networks**. College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 2003.

[24] NEUGEBAUER, M.; KABITZSCH, K. **A New Protocol for a Low Power Sensor Network**. Institute for Applied Computer Science, Deparment of Computer Science, Dresden University of Technology, Dresden, Alemanha, 2003.

[25] BURRELL, J.; BROOKE, T.; BECKWITH, R. **Vineyard Computing: Sensor Networks in Agricultural Production**. IEEE CS e IEEE ComSoc, 1536-1268/04, 2004. Páginas 38 a 45.

[26] UNIVERSIDADE DE BRASÍLIA. Livro Branco: Sistema de Controle de Irrigação. **Livro_branco_02072003_rdz.PDF**. 2003. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[27] PING, S. **Delay Measurement Time Synchronization for Wireless Sensor Networks**. Intel Research Berkeley Lab, Berkeley, EUA, 2003.

[28] TANENBAUM A.S. **Computer Networks**. Terceira edição. Upper Saddle River New Jersey. Prentice Hall, 1996. Páginas 29 e 30.

[29] SNAP – SCALEABLE NODE ADDRESS PROTOCOL. Desenvolvido pela High Tech Horizon, 1998-2002. Apresenta textos sobre hardware e software de comunicação de dados em instalações elétricas. Disponível em <http://www.hth.com/>. Acessado em 18/03/2004.

[30] MICROCHIP INC. Fabricante de microcontroladores e ferramentas de desenvolvimento de seus produtos. Apresenta datasheets detalhados de seus chips e disponibiliza as ferramentas de desenvolvimento gratuitamente. Disponível em <http://www.microchip.com/>. Acessado em 18/03/2004.

[31] CASTRICINI E.; MARINANGELI. G. **S.N.A.P. in PIC**. Apresenta arquivos em linguagem assembly que implementa o protocolo SNAP através de um modem no microcontrolador PIC da Microchip e um software de configuração do protocolo para o ambiente operacional Windows98. Disponível em <http://www.hth.com/filelibrary/snap/usrcode/PICTOOLS.ZIP>. Acessado em 20/03/2005.

[32] HLÁVKA, P.; KRATOCHVÍLA, T.; REHAK, V.; SAFRÁNEK, D.; SIMECEK, P. ; VOJNAR, T. CRC-64 Algorithm Analysis and Verification. **CESNET technical report** número 27/2005, Dez. 2005. Disponível em <http://www.cesnet.cz/doc/techzpravy/2005/crc64/crc64.pdf>. Acesso em: 15 jan. 2006.

[33] KAWAHARA, Y.; MINAMI, M.; MORIKAWA, H.; AOYAMA, T. **Design and Implementation of a Sensor Network Node for Ubiquitous Computing Environment**. 2003. University of Tokyo, Tokyo, Japão.

[34] KNUTH, D. E. **The Art of Computer Programming**. Segunda edição. Massachussets: Addison-Wesley, 1981. 176p.

[35] LOGUE, A. **Hardware Random Number Generator**. Apresenta um gerador de números aleatórios. Disponível em <www.cryogenius.com/hardware/rng/>. Acessado em 04/04/2006.

[36] MARSAGLIA, G. **The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness**. Florida State University, 1995.

[37] WAN, C.; EISENMAN, S.B.; CAMPBELL, A.T. **CODA**: Congestion Detection and Avoidance in Sensor Networks. 2003. Department of Electrical Engineering, Columbia University, New York, EUA.

[38] LUCHINE, G.; PICANCO, R.; ROMARIZ, A. **Reliability of Traffic on Sensor Networks using Simple Protocols**. In: SoftCOM 2006 – 14th International Conference on Software, Telecommunications & Computer Networks, 2006, Split, Croácia. FESB - Fakultet Elektrotehnike, Strojarstva i Brodogradnje. ISBN 953-6114-87-9.

[39] ZHONG, L.C.; SHAH, R.; GUO, C.; RABAEY, J. **An Ultra-low Power and Distributed Access Protocol for Broadband Wireless Sensor Networks**. 2001. Berkeley Wireless Research Center, Department of EECS, University of California, Berkeley, EUA.

[40] ASSUNÇÃO, L. S. **Amplificador de potência de RF com controle digital em tecnologia CMOS para aplicação em telemetria**. 2004. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[41] VOGEL, P.R.O. **Demodulador em tecnologia CMOS para um transceptor de RF a 900MHz em um sistema em chip**. 2005. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília.

[42] POLASTRE, J. **Sensor Network Media Access Design**. 2003. Computer Science Division, EECS Department, University of California, Berkeley, EUA.

[43] MITCHELL, G. **Data in a Wireless Sensor Network:** A view from the Field. [2005?]. [S.I.: s.n.]. BBN Technologies, EUA.

[44] DATTALO, T. S. **CRC-8 for Dallas iButton Products**. 2003. Apresenta um código em Assembly de CRC-8 altamente otimizado. Disponível em <http://www.dattalo.com/technical/software/pic/crc_8.asm> . Acessado em 03/11/2005.

[45] DATTALO, T. S. **CRC-16 Algorithm**. [2001?]. Apresenta um código assembly de CRC-16 altamente otimizado. Disponível em <http://www.dattalo.com/technical/software/pic/crc16.asm>. Acessado em 19/04/2006.

APÊNDICES

A – CÓDIGO COMPLETO PARA O PIC

A.1 - Função TX_SNAP

A primeira ação da TX_SNAP é desabilitar todas as interrupções para não permitir que outra tarefa possa impedir a transmissão do pacote com o sincronismo correto, pois a função de tempo leva em consideração a execução em tempo real do microcontrolador. Na figura A.1, podemos ver a desabilitação global das interrupções e o *bit* RUN_TX ser colocado em 1 para indicar que a parte de transmissão está sendo executada. Além disso, a instrução BTFSC testa se o *bit* RUN_RX é igual a 1. Caso positivo, coloca 1 no *bit* RUN_RTX também.

```
BCF      INTCON,GIE     ;DEsabilitacao global de interrupcoes

BSF      RUN_TX     ;RUN_TX=1

BTFSC    RUN_RX     ;Se RUN_RX=0, entao pula 1 linha
BSF      RUN_RTX    ;Se RUN_RX=1, entao RUN_RTX=1 (RX->TX)
```

Figura A.1: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP

Depois, a função TX_SNAP carrega os cabeçalhos padrão que foram definidos no arquivo SNAP.INC. A seguir, o *bit* RUN_RX é testado, conforme mostrado na figura A.1. Se RUN_RX for 1, o nó está respondendo um pacote que foi recebido anteriormente. O *bit* mais significativo do ACK é ligado (1) para indicar que o pacote a ser transmitido é de resposta. O *bit* menos significativo tem seu estado lógico colocado em conformidade com o estado do *bit* ERR. Caso o *bit* ERR seja zero, o pacote recebido não teve erro detectado e o ACK fica com o valor 10b (ver figura 3.4). Caso o *bit* ERR seja 1, o pacote foi recebido com erro e o ACK fica com o valor 11b.

```
BTFSS    RUN_RX          ;Testa o bit RUN_RX, se for 1 pula uma linha
GOTO     carrega_ack     ;Se RUN_RX=0 entao checa se ACK é requerido
BSF      HDB2,1          ;Liga o bit de resposta do ACK=1X
```

```
BTFSS   ERR       ;Testa o bit ERR gerado na recepção, se for 1 pula
BCF     HDB2,0    ;Zerar o bit zero se estiver certo, pois 1 é o padrão
```

Figura A.2: trecho do arquivo SNAP.ASM contendo o inicio da função TX_SNAP

Caso o *bit* RUN_RX tenha sido igual a 1, o TX_SNAP verifica se já houve a troca do endereço de origem do pacote que foi recebido anteriormente para o endereço de destino do próximo pacote que será transmitido. Isso permite que um comando enviado ao nó possa requerer vários pacotes de resposta, como, por exemplo, se a estação de campo pedir todos os dados coletados na memória pelo nó. Esta troca só poderá ocorrer uma única vez, caso contrário o nó poderia ficar enviando pacotes endereçados para si mesmo, e por isso o *bit* SAB_PARA_DAB é ligado para realizar este controle.

Caso o *bit* RUN_RX tenha sido igual a zero, o TX_SNAP decide continuar normalmente, pois está transmitindo um pacote que não se trata de uma resposta, mas que pode requisitar uma resposta do nó de destino. Assim, caso a aplicação principal tenha ligado o *bit* ACK_PKT_TX, uma resposta foi requerida e o *bit* menos significativo de ACK é mantido no valor 1 que se encontra na constante HEADER2 carregada anteriormente. Caso o *bit* ACK_PKT_TX esteja desligado, nenhuma resposta foi requerida e o *bit* menos significativo do ACK é igualado a zero, conforme a figura 3.4 mostrada na estrutura do pacote (3.4.1) e mostrado na figura A.3 a seguir.

```
carrega_ack                           ;Apenas na situação RUN_TX=1 e RUN_RX=0
        BTFSS     ACK_PKT_TX          ;Se ACK_PKT_TX=1, pula 1 linha
        BCF       HDB2,0              ;Se ACK_PKT_TX=0, zera ACK=X0


charge_sab
        MOVLW     MY_ADDR1
        MOVWF     SAB1      ;Se vou TX, colocar endereço do nó no SAB

        CALL apply_edm    ;CALCULA O EDM PARA O PACOTE A SER ENVIADO

        CALL send_snap    ;TRANSMITE O PACOTE
```

Figura A.3: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Após carregar o endereço do próprio nó, o TX_SNAP assume que a aplicação principal já configurou o endereço de destino DAB1 e os dados a serem enviados nos *bytes* DBi (1<=i<=8) e chama a função APPLY_EDM para calcular o CRC-8 do pacote. Tendo o pacote sido completamente determinado, a TX_SNAP chama a função SEND_SNAP para efetivamente enviar o pacote.

Depois de ter enviado o pacote, o TX_SNAP faz uma nova verificação no *bit* RUN_RX para decidir sobre o fluxo do protocolo a ser tomado. Caso RUN_RX seja igual a 1, então o TX_SNAP foi invocado para responder um pacote para o nó que originou a comunicação. Uma vez respondido (situação do *acknowledge* enviado), a TX_SNAP vai para o seu fim, pois nada mais deve ser feito por ela. Caso RUN_RX seja zero, então o TX_SNAP foi chamado pela aplicação principal para transmitir um pacote. Enviado o pacote, o TX_SNAP verifica se foi pedido uma confirmação sobre o recebimento para o nó de destino. Em caso negativo, também vai para o fim, pois a ação já foi tomada.

```
BTFSS       RUN_RX       ;Se RUN_RX=1, pula 1 linha
BTFSS       HDB2,LSB     ;Se ACK=01, entao pula 1 linha
GOTO        end_txsnap   ;Se RX estiver rodando vai para o fim
```

Figura A.4: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Caso tenha sido pedida a confirmação (*acknowledge*), o TX_SNAP deve esperar um tempo pelo recebimento do pacote de resposta. Este tempo foi arbitrado em 8 segundos, se decorrido mais do que isso, o pacote é considerado perdido e o *bit* RECEBIDO é igualado a zero para indicar que o nó de destino não recebeu o pacote ou o pacote de resposta se perdeu na comunicação.

O TX_SNAP coloca 1 no *bit* REJ para indicar que o pacote não foi recebido, habilita as interrupções e inicia o contador de tempo na figura A.5.

```
        BSF         REJ              ;Inicia com REJ=1
        BSF         INTCON,GIE       ;Habilitação global de interrupções (RX=0)
```

```
wait_loop2
        CLRWDT
        BTFSS       REJ             ;Se REJ=1, então pula 1 linha
        GOTO        exit_wait       ;Se REJ=0, então um pacote foi recebido

        MOVF        TMR0,W
        BTFSS       STATUS,Z        ;Quando TMR=0, sai do LOOP2
        GOTO        wait_loop2

exit_wait
        BCF         INTCON,GIE      ;DEsabilitacao global de interrupções
```

Figura A.5: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

Note na figura A.25 da página 114 que quando o RX_SNAP receber adequadamente um pacote, o *bit* REJ vai para o valor zero e isso interrompe a espera do TX_SNAP. Outro fator importante, que vai poder ser visto na função RX_SNAP quando esta for apresentada, é que caso um pacote que não seja de resposta (ACK<>1X) tenha sido recebido, o *bit* REJ continuará sendo 1, pois o protocolo está somente aguardando pacotes de resposta. Caso um outro pacote de resposta tenha sido recebido vindo de outro nó, além daquele que foi o destinatário do pacote originalmente enviado, o *bit* REJ continuará sendo 1. Então, conclui-se que a espera do TX_SNAP somente é interrompida por um recebimento do pacote de resposta vindo do nó para o qual o pacote foi transmitido, o que elimina falsas confirmações advindas do tráfego de pacotes dentro da rede de sensores.

Decorrido o tempo arbitrado ou recebido o pacote de resposta esperado, as interrupções são novamente desabilitadas. Na figura A.6 podem ser vistas as instruções que decidem a sobre o recebimento correto do pacote pelo nó de destino.

```
        BTFSC       REJ             ;Se REJ=0, pula 1 linha
        GOTO        nao_recebido    ;Se não houve resposta --> time out
        BTFSS       ERR             ;Se REJ=0 e ERR=1, pula 1 linha
        BTFSC       HDB2,LSB        ;Se REJ=0, então ACK=10 OU 11
        GOTO        nao_recebido    ;Houve erro detectado na resposta
```

```
        BSF         RECEBIDO         ;REJ=0, ERR=0, ACK=10, então OK
        GOTO        end_txsnap

nao_recebido
        BCF         RECEBIDO         ;Pacote não foi recebido adequadamente
```

Figura A.6: trecho do arquivo SNAP.ASM contendo parte da função TX_SNAP

O TX_SNAP verifica o estado lógico do *bit* REJ, caso seja 1 temos duas possibilidades: ou não foi recebido o pacote de resposta do nó para o qual ele foi transmitido, ou nenhum pacote foi recebido. Em ambas, o protocolo considera que não existe certeza sobre o recebimento do pacote pelo nó de destino e coloca o valor zero no *bit* RECEBIDO. Caso REJ tenha valor zero, então foi recebido um pacote de resposta do nó para o qual um outro foi transmitido anteriormente e a questão é verificar se houve ou não erro neste pacote de resposta. Tendo havido erro (ERR=1), a hipótese anterior se repete e o *bit* RECEBIDO recebe o valor zero. Caso o *bit* ERR tenha valor zero, então o pacote de resposta foi recebido corretamente, mas falta checar se o cabeçalho deste pacote indica que houve a confirmação de recebimento correto pelo nó de destino do pacote anterior. Isso é verificado no *bit* menos significativo do ACK (ver figura 3.4). Caso ACK=11, então houve erro na recepção e o *bit* RECEBIDO recebe o valor zero. Caso ACK=10, então todas as condições foram cumpridas, o pacote teve confirmação de ter sido recebido corretamente pelo nó de destino e o valor 1 é colocado no *bit* RECEBIDO. Repare que, após transmitir um pacote, só existem duas situações para o *bit* RECEBIDO no fluxo apresentado: ou ele recebe o valor 1, ou recebe o valor zero. A questão é que, sempre após a transmissão de um pacote de dados, a aplicação principal, ao testar o *bit* RECEBIDO, tem uma confirmação sobre o real estado da comunicação.

```
        BTFSS       RUN_RX           ;Se RUN_RX=1, então pula 1 linha
        BSF         INTCON,GIE       ;Habilitação global de interrupções
```

Figura A.7: trecho do arquivo SNAP.ASM contendo o final da função TX_SNAP

Na figura A.7 pode ser observado o cuidado de verificar que a função RX_SNAP não está sendo executada através do *bit* RUN_RX, para depois realizar a reabilitação de todas as interrupções. Caso esse cuidado não fosse tomado, a função TX_SNAP estaria invocando

um nova interrupção para uma outra instância da função RX_SNAP, provocando a perda dos valores guardados pelo tratador de interrupções que poderia colocar o microcontrolador em estados não previstos e causaria a perda de controle do nó.

Para terminar, a função TX_SNAP limpa os *bits* RUN_RTX e RUN_TX e retorna para a aplicação principal.

A.2 - Função SEND_SNAP

Esta função trata de organizar o envio de cada *byte* do pacote de dados do protocolo, fazendo a passagem de argumentos através do registrador de trabalho W e do ponteiro, para depois chamar as funções corretamente. Quando se tem apenas um *byte*, a função SEND_SNAP coloca o *byte* a ser enviado no registrador de trabalho W e chama a função SEND_BYTE. Isto pode ser visto na figura A.8 abaixo.

```
MOVLW     SYNC          ;Carrega constante no registrador W
CALL      send_byte     ;Envia byte SYNC
```

Figura A.8: trecho do arquivo SNAP.ASM contendo parte da função SEND_SNAP

Quando são vários *bytes* a serem enviados, ela aponta o ponteiro para o primeiro *byte* a ser enviado, coloca o número de *bytes* ou palavras a serem enviadas no registrador W e chama a função SEND_WORD (vide figura A.9).

```
MOVF      DB_MAX,W      ;Apontar ponteiro FSR para DB_MAX
MOVWF     FSR
MOVF      NDB,W         ;e colocar NDB no registrador W
CALL      send_word     ;Envia as palavras pela interface de saída
```

Figura A.9: parte da função SEND_SNAP que prepara o envio de vários *bytes*

A função SEND_SNAP toma esse procedimento *byte* por *byte* do início ao fim do pacote de dados, mas, antes de retornar, grava o endereço para o qual o pacote foi enviado (mostrado na figura A.10). Isto precisa ser feito para que, caso tenha sido requerido a confirmação de recebimento (*acknowledge*), o nó que enviou saiba de qual nó está esperando resposta.

```
        MOVF        DAB1,W              ;Guarda DAB1 do pacote transmitido
        MOVWF       TEMP+7              ;na variável TEMP+7
```

Figura A.10: parte da função SEND_SNAP que guarda o endereço de destino

Havendo terminado sua tarefa e guardado o endereço do pacote transmitido para comparações futuras, a função SEND_SNAP termina e retorna para a função TX_SNAP.

A.3 - Função SEND_WORD

Esta função interpreta os parâmetros passados pela SEND_SNAP de forma a colocar os *bytes* do pacote ordenadamente para a função SEND_BYTE.

```
        MOVWF       TEMP        ;Coloca no TEMP o nº de bytes a serem enviados


sendw_loop
        MOVF        INDF,W      ;INDF e' o conteúdo do ponteiro FSR, coloca em W
        CALL        send_byte   ;Transmite o byte no registrador W

        INCF        FSR,F       ;Incrementa o ponteiro
        DECFSZ      TEMP,F      ;Decrementa o número de bytes, se TEMP=0 pula
        GOTO        sendw_loop
```

Figura A.11: trecho da função SEND_WORD que ordena os *bytes* para o envio

Na figura A.11, a primeira instrução transfere o conteúdo do registrador W para o TEMP, pois este será o contador do número de *bytes* a serem enviados. Ao entrar no *loop* da função, o conteúdo do ponteiro (registrador INDF) é movido para o registrador W e a função SEND_BYTE é chamada para efetivamente enviar o *byte* pela interface de saída. Após o primeiro *byte* ter sido enviado, o endereço do ponteiro FSR é incrementado e o número de *bytes* restantes é decrementado. No caso da instrução DECFSZ, caso o TEMP seja igual a zero nesta operação, a próxima instrução não é executada. Assim, a função sai de seu *loop* e retorna para a SEND_SNAP.

A.4 - Função SEND_BYTE

Esta função efetivamente transmite o *byte* que se encontra no registrador de trabalho W na interface de saída. Para isso, os estados lógicos são alternados pela SEND_BYTE e os intervalos são controlados pela função TEMPO.

As primeiras instruções transferem o conteúdo do registrador W para um registrador temporário denominado TEMP+1 e o valor 8 para outro registrador TEMP+2 que servirá de contador.

```
sendb_loop

        BCF         STATUS,C        ;Zera o carry (C)
        RRF         TEMP+1,F        ;Rotaciona os bits do byte a ser enviado

        BTFSC       STATUS,C        ;Se C=0, então pula 1 linha
        BSF         TXD             ;Liga o pino de saída
        BTFSS       STATUS,C        ;Se C=1, então pula 1 linha
        BCF         TXD             ;Desliga o pino de saída
```

Figura A.12: trecho da função SEND_BYTE que efetivamente transmite os *bits*

Conforme pode ser visto na figura A.12, o *bit* de *carry* (C) é igualado a zero para que a forma de implementação das instruções seguintes esteja correta. A instrução RRF causa uma rotação nos *bits* para a direita, considerando que o *bit* menos significativo vai para o *bit* de *carry* e o valor que estava no *bit carry* vai para o *bit* mais significativo do *byte* rotacionado. Assim, caso a instrução RRF rotacione um *bit* de valor zero para o *bit* de *carry*, a instrução BTFSC vai pular uma linha do código, a instrução BTFSS vai executar a linha seguinte, que desliga (estado lógico zero) o pino de saída da interface. Caso a instrução RRF rotacione um *bit* de valor 1 para o *bit* de *carry*, a instrução BTFSC vai executar a próxima instrução, que liga (estado lógico 1) o pino de saída da interface.

```
        MOVLW       TEMPO_BIT       ;Espera o tempo de um bit
        CALL        tempo
```

```
DECFSZ    TEMP+2,F         ;Decrementa o contador de bits
GOTO      sendb_loop       ;Se TEMP+2=0, esta linha não é executada
```

Figura A.13: trecho da função SEND_BYTE que temporiza e decrementa o contador

A seguir, como pode ser visto na figura A.13, a função joga o valor do tempo de um *bit* no registrador de trabalho W e chama a função TEMPO. Após decorrido este tempo, o contador de *bits* a serem enviados é decrementado em uma unidade e uma instrução de salto incondicional joga o fluxo da função SEND_BYTE de volta ao ponto do rótulo (*label*) chamado sendb_loop. Na verdade, a instrução DECFSZ pula a instrução de salto incondicional GOTO quando o valor de TEMP+2 chegar a zero.

Quando não houver mais *bits* a serem transmitidos, a função retorna para a SEND_WORD ou para a SEND_SNAP, dependendo de qual delas chamou a SEND_BYTE.

A.5 - Função RX_SNAP

Esta função não é executada diretamente pela aplicação principal, pois esta é, na verdade, interrompida pelo microcontrolador através de um evento que acabou de ocorrer. No caso deste protocolo, esse evento ocorreu devido à chegada de algum dado e o microcontrolador deve parar o processamento da aplicação principal com dois objetivos: (a) executar a função RX_SNAP que vai receber o pacote de dados, e (b) tratar os dados que estão chegando, para que não sejam perdidos.

A primeira tarefa a ser realizada é indicar que a parte de recepção do protocolo está sendo executada. Isto é feito através da colocação do valor 1 no *bit* de controle RUN_RX. Então, a função RX_SNAP chama a função RECEIVE_SNAP, que vai efetivamente receber o pacote de dados ou comando.

```
BSF       RUN_RX           ;Indica que rx_snap esta sendo executada
CALL      receive_snap     ;Efetivamente recebe o pacote do protocolo

BTFSC     REJ              ;Se REJ=0, então pula 1 linha
GOTO      end_rxsnap       ;Se o pacote tiver sido rejeitado, termina
```

Figura A.14: trecho do arquivo SNAP.ASM contendo o inicio da função RX_SNAP

Conforme a figura A.14, após a chamada da função RECEIVE_SNAP, a função RX_SNAP verifica o estado lógico do *bit* REJ para checar se o pacote foi rejeitado. Caso tenha sido, a função termina e retorna para a aplicação principal. Caso o pacote não tenha sido rejeitado, a função RX_SNAP chama a função APPLY_EDM que vai computar o resultado do cálculo de detecção de erro e comparar com o resultado que veio embutido ao final do pacote recebido. Na f, pode ser visto que, havendo erro detectado, a função também termina e retorna para a aplicação principal.

```
CALL     apply_edm      ;Calcula o método de detecção de erros


BTFSC    ERR            ;Se ERR=0, então pula 1 linha
GOTO     end_rxsnap     ;Se ocorreu erro (ERR=1), sair do rx_snap
```

Figura A.15: trecho da função RX_SNAP com ênfase no método de detecção de erros

Neste ponto, a função RX_SNAP verifica se a função TX_SNAP estava sendo executada através do *bit* de controle RUN_TX. Caso este *bit* esteja com o valor 1, a função TX_SNAP estava sendo executada anteriormente. Neste caso, foi transmitido um pacote com o pedido de confirmação de recebimento (*acknowledge*). A função TX_SNAP habilitou as interrupções para recebimento do pacote de resposta, sendo que, este, ao ser enviado pelo nó de destino no sentido contrário, levou a execução da função RX_SNAP. Uma vez recebido adequadamente ou não, a função RX_SNAP retorna para a função TX_SNAP, que deverá verificar com que condições o pacote de resposta foi recebido e colocar o estado lógico do *bit* RECEBIDO em conformidade com os *bits* de controle REJ, ERR e ACK.

Se a função RX_SNAP verificar que o *bit* de controle RUN_TX está com o valor zero, a função TX_SNAP não estava sendo executada e um novo pacote está sendo recebido. Esta condição faz com que o fluxo do protocolo continue na função RX_SNAP, limpando o *bit* de controle SAB_PARA_DAB e chamando a função COMANDO_EXEC.

```
BTFSC    RUN_TX         ;Se RUN_TX=0, então pula 1 linha
GOTO     end_rxsnap     ;Se RUN_TX=1, então termina a recepção
```

```
runtx_0                                  ;TX NAO ESTA RODANDO
        BCF         SAB_PARA_DAB

        CALL        comando_exec         ;SE FOR UM COMANDO, EXECUTA

        BTFSS       HDB2,LSB             ;Se pacote recebido exigir resposta, pula
        GOTO        end_rxsnap           ;Se pacote não exigir resposta, termina
```

Figura A.16: parte da função RX_SNAP com ênfase no tratamento do pacote recebido

O *bit* de controle SAB_PARA_DAB tem seu valor igualado a zero para que, caso o pacote recebido exija múltiplos pacotes de resposta, o endereço de origem do pacote recebido seja transferido uma única vez para o endereço de destino do pacote que será transmitido (vide explicação da função TX_SNAP feita anteriormente). Após esta transferência ser efetivada, o *bit* SAB_PARA_DAB tem seu valor igualado a 1.

A chamada para a função COMANDO_EXEC se faz necessária para separar o bloco de código de comandos das instruções do RX_SNAP. Assim fica criada uma certa modularidade que elimina a sensação do programador de estar alterando a função de recepção do protocolo. Note que esta chamada também é incondicional, pois é a função COMANDO_EXEC que vai verificar se o pacote recebido contém ou não um comando a ser realizado e tomar as providências necessárias. Assim, além de responder adequadamente aos comandos, a função COMANDO_EXEC também pode enviar mensagens de que nenhum comando foi realizado.

Após a verificação do conteúdo do pacote, a função RX_SNAP checa se os cabeçalhos do pacote pedem uma confirmação de recebimento. Conforme explicado anteriormente e visto na figura 3.4, isso é feito através do estado lógico do *bit* menos significativo (LSB) do cabeçalho HDB2. Caso o pacote recebido não exija resposta (ACK=00), a função RX_SNAP termina. Caso exija resposta (ACK=01), então a função RX_SNAP identifica o endereço de destino do pacote para constatar se foi um pacote enviado em um *broadcast* (vide figura A.17).

```
        CLRW                                  ;Registrador W = zero
        XORWF       DAB1,W                    ;Testa se DAB1=W=zero
        BTFSS       STATUS,Z                  ;Se DAB1=0, então pula 1 lnha
        GOTO        continua_resposta         ;Não sendo broadcast, vai p/ continua_resp

        MOVLW       MY_ADDR1                  ;Sendo de broadcast, colocar end. no W
        MOVWF       TEMP+1                    ;Gravar end. do nó no registrador TEMP+1
```

Figura A.17: trecho da função RX_SNAP com ênfase no reconhecimento de *broadcast*

Como pode ser visto na figura A.17, se o endereço de destino do pacote recebido for igual a zero, então a instrução BTFSS pula a instrução de salto incondicional GOTO para continuar tratando da resposta como sendo de *broadcast*. Neste caso, a primeira providência é colocar o endereço do nó em um registrador temporário, pois ele vai auxiliar um temporizador a encontrar qual o intervalo de transmissão de resposta. Caso o endereço contido em DAB1 não seja de *broadcast*, o fluxo é desviado para o rótulo continua_resposta no final da função RX_SNAP.

```
espera_broad1
        MOVLW       6                         ;faz a contagem do TMR0 começar de 6
        MOVWF       TMR0                      ;assim teremos 250 contagens no byte

espera_broad2
        MOVF        TMR0,W                    ;Transfere o conteúdo do TMR0 para o W
        BTFSS       STATUS,Z                  ;Confere se W=0, se for pula 1 linha
        GOTO        espera_broad2;

        DECFSZ      TEMP+1,F                  ;Decrementa endereço, se TEMP+1=0 pula
        GOTO        espera_broad1;
```

Figura A.18: trecho da função RX_SNAP com ênfase na espera de *broadcast*

Fica evidente na figura A.18 que o tempo de espera para que o nó responda o pacote de *broadcast* depende do endereço que o nó possui. Isso é uma medida necessária para evitar que todos os nós transmitam pacotes de resposta ao mesmo tempo, o que inevitavelmente

causaria a perda de todos os pacotes por colisão. Assim, cada nó tem um intervalo de tempo bem definido, sendo que os nós de menor endereço responderão antes e os de maior endereço responderão depois. Este é um método simples que foi adotado nesta rede de sensores visando organizar as respostas com o menor gasto de memória possível. Tendo isso em mente, ao se implementar a rede de sensores, os nós devem ser numerados em ordem crescente, evitando que existam endereços não utilizados. Caso algum endereço numérico não esteja sendo usado pela rede, o respectivo intervalo de transmissão continua mantido, pois os outros nós não têm conhecimento de que poderiam estar utilizando-o para aumentar a eficiência do tempo de resposta.

```
continua_resposta
        CALL        tx_snap            ;responde caso TX não esteja rodando

end_rxsnap
        BCF         RUN_RX             ;RUN_RX=0, pois a função terminou
```

Figura A.19: trecho do arquivo SNAP.ASM contendo o final da função RX_SNAP

Na figura A.19, a função RX_SNAP envia o pacote de resposta depois de decorrido o tempo de *broadcast* ou imediatamente após o pacote recebido ter sido processado. Caso tenha havido algum comando que exigiu a resposta de múltiplos pacotes de dados, o último pacote pode ser carregado com um indicador que informa ao nó comandante que o comando foi bem sucedido.

A função RX_SNAP, para finalizar, coloca zero no *bit* de controle RUN_RX para sinalizar que não está mais sendo executado e retorna para a aplicação principal.

A.5 - Função RECEIVE_SNAP

Esta função trata de organizar a recepção dos pacotes de dados ou comandos. Na medida em que o pacote vai sendo recebido, novas verificações são feitas e, caso o conteúdo não obedeça algumas regras pré-estabelecidas, o pacote é sumariamente descartado.

Na figura A.20 abaixo, a função RECEIVE_SNAP primeiramente chama a função RECEIVE_BYTE para receber o *byte* de sincronismo. Depois, aponta o ponteiro para o

primeiro *byte* do cabeçalho, coloca o número 2 no registrador de trabalho W e chama a função RECEIVE_WORD. Este procedimento visa informar a função RECEIVE_WORD que são dois *bytes* que deverão ser recebidos e que devem ser colocados a partir da posição do ponteiro.

```
CALL      receive_byte      ;Recebe o byte de sincronismo (SYNC)


MOVLW     HDB2              ;Transfere endereço de HDB2 para o W
MOVWF     FSR               ;Aponta o ponteiro para o end. de HDB2
MOVLW     2                 ;Numero de bytes a serem recebidos
CALL      receive_word
```

Figura A.20: trecho do SNAP.ASM contendo o inicio da função RECEIVE_SNAP

A função RECEIVE_SNAP verifica se a função TX_SNAP está sendo executada através do *bit* de controle RUN_TX. A diferença é que se a função TX_SNAP estiver sendo executada, o nó está recebendo um pacote de resposta e os *bits* de ACK do cabeçalho devem ser 10b ou 11b para que o pacote não seja rejeitado. Se a função TX_SNAP não estiver sendo executada, então os cabeçalhos são analisados para ver se estão em conformidade com as constantes definidas no arquivo SNAP.INC, com a exceção de poderem ser cabeçalhos de pacotes de comandos e de pacotes que não exijam confirmação de recebimento (*acknowledge*). Qualquer outra configuração de cabeçalho também é prontamente descartada e o pacote não é mais recebido.

Tendo o pacote recebido aprovação nos seus cabeçalhos, a função RECEIVE_SNAP recebe o *byte* de endereço de destino (DAB1) e verifica se corresponde ao endereço de *broadcast* ou ao endereço do nó (vide figura A.21).

```
CLRW                        ;Zera registrador de trabalho W
XORWF     DAB1,W            ;Verifica se DAB1=W=0
BTFSC     STATUS,Z          ;Se DAB1 <> 0, pula 1 linha
GOTO      receive_sab       ;Se DAB1=0, vai receber o end. de origem
```

Figura A.21: parte da função RECEIVE_SNAP que verifica se o pacote é de *broadcast*

Caso o pacote seja de *broadcast*, a função RECEIVE_SNAP é desviada para receber o *byte* de endereço de origem do pacote. De acordo com a figura A.22, o mesmo acontece se o endereço de destino coincidir com o endereço do nó. Caso o endereço de destino não se enquadre em nenhuma das duas possibilidades, o pacote é rejeitado e nenhum *bit* adicional é recebido.

```
MOVLW   MY_ADDR1    ;Transfere MY_ADDR1 para o registrador W
XORWF   DAB1,W      ;Verifica se DAB1= MY_ADDR1
BTFSS   STATUS,Z    ;Se for igual, pula 1 linha
GOTO    reject      ;Se não for igual, rejeita o pacote
```

Figura A.22: trecho da função RECEIVE_SNAP que verifica se o pacote é deste nó

A seguir, a função RECEIVE_SNAP recebe o *byte* SAB1 de origem do pacote, verificando se este endereço contido no SAB1 é diferente do endereço do nó. Caso seja igual, o nó conclui que existem dois elementos de rede com o mesmo endereço e descarta o pacote (vide figura A.23). Essa situação fere a premissa básica que cada nó tem um endereço único em cada rede de sensores. Considerando que, cabe a estação de campo organizar e assegurar o correto funcionamento desta rede, o nó deve apenas ignorar o pacote. Quaisquer mecanismos de detecção e correção de endereços errados que fossem implementados trariam um aumento do código, suficientemente significativo, para justificar a adoção de um procedimento tão simples em detrimento de um comportamento mais elaborado.

```
MOVLW   MY_ADDR1    ;Se SABi for igual ao MY_ADDRi então
XORWF   SAB1,W      ;existem dois nós com o mesmo endereço
BTFSC   STATUS,Z    ;e o presente pacote e' rejeitado
GOTO    reject
```

Figura A.23: trecho que verifica se existem dois nós com o mesmo endereço

Caso o *bit* RUN_TX esteja ligado, a função RECEIVE_SNAP verifica se o endereço de origem do pacote recebido é igual ao endereço de destino do pacote transmitido anteriormente. Esta checagem é útil para distinguir o pacote de confirmação de recebimento (*acknowledge*) que está sendo recebido de outros pacotes de confirmação que

estejam trafegando pela rede de sensores. Ainda que, a estação de campo possa estar coordenando a comunicação entre os nós, este cuidado previne que possa haver uma falsa confirmação de recebimento, pois, a resposta deve ser enviada por quem recebeu o pacote com pedido de confirmação, conforme pode ser visto na figura A.24.

```
BTFSS    RUN_TX       ;Se o bit RUN_TX for igual a 1, então
GOTO     receive_db   ;o protocolo verifica se a resposta veio do
                      ;nó para o qual o pacote foi enviado

MOVF     TEMP+7,W     ;Se TEMP+7 (DAB1 do pacote anterior) for
XORWF    SAB1,W       ;diferente de SAB1, então rejeita o pacote
BTFSS    STATUS,Z     ;recebido
GOTO     reject
```

Figura A.24: verificação se o SAB1 do pacote recebido é igual ao DAB1 anterior

Neste ponto, a função RECEIVE_SNAP recebe os *bytes* de dados do pacote através da chamada de função RECEIVE_WORD, onde o número de palavras a serem recebidas é colocado no registrador de trabalho W, no instante imediatamente anterior. Após terem sido recebidos os *bytes* de dados, a função RECEIVE_SNAP recebe o *byte* de detecção de erro CRC1. Pode ser visto na figura A.25 que, após o recebimento deste *byte*, o *bit* REJ é igualado a zero para indicar que o pacote foi recebido adequadamente. Caso algumas das restrições colocadas anteriormente tivessem sido encontradas, o fluxo seria desviado para o rótulo *reject*, o *bit* REJ teria seu valor igual a 1 para indicar que o pacote foi rejeitado, e a função retornaria para a RX_SNAP.

```
MOVF     EB_MAX,W          ;Recebe o byte de CRC-8 do pacote
MOVWF    FSR               ;Aponta ponteiro para o byte CRC1
MOVF     NEB,W             ;Coloca número de bytes no registrador W
CALL     receive_word
BCF      REJ               ;RECEBIDO PKT A SER VERIFICADO,
GOTO     end_receivesnap   ;OU SEJA, REJ=0
```

```
reject
        BSF         REJ             ;REJ=1 PARA INDICAR QUE O PACOTE
                                    ;FOI REJEITADO

end_receivesnap
        RETURN                      ;Fim da receive_snap
```

Figura A.25: parte da função RECEIVE_SNAP que recebe CRC1 e configura o *bit* REJ

Também pode ser visto na figura A.25 acima que, o *byte* de CRC1 poderia ser recebido através da função RECEIVE_BYTE ao invés da RECEIVE_WORD, pois se trata de uma única palavra. Entretanto, esta forma de receber o CRC1 foi mantida por permitir uma maior flexibilidade, caso o número de palavras de detecção de erro for aumentado, uma única mudança no início do código na constante NEB (*Number of Error Bytes*) faria com que o protocolo recebesse esses *bytes* a mais.

A.6 - Função RECEIVE_WORD

A função RECEIVE_WORD inicia sua execução verificando se o número de *bytes* colocados no registrador de trabalho W é igual a zero. Caso esta verificação seja verdadeira, a função retorna para função anterior, por não haver nada a ser feito. Caso seja verificado que o conteúdo de W é positivo, função RECEIVE_WORD chama a função RECEIVE_BYTE, como pode ser visto na figura A.26, para efetivamente receber os *bits* da palavra que estão chegando à interface.

```
        MOVWF       TEMP            ;Coloca conteúdo de W no registrador TEMP

receivew_loop
        CALL        receive_byte    ;Lê o byte na interface de entrada => W

        MOVWF       INDF            ;Coloca o W no conteúdo do ponteiro
        INCF        FSR,F           ;Incrementa o endereço do ponteiro
        DECFSZ      TEMP,F          ;Decrementa o contador de bytes a receber
        GOTO        receivew_loop   ;Se TEMP=0, pula esta linha
```

Figura A.26: trecho da função RECEIVE_WORD que recebe o número de *bytes* de W

Repare que a função se trata de um *loop* finito, que vai chamar a função RECEIVE_BYTE, tantas vezes quanto tiver sido colocado no conteúdo do registrador W, colocando *byte* por *byte* no registrador indicado pelo ponteiro que tem o seu endereço incrementado a cada iteração.

A.7 – Função RECEIVE_BYTE

Esta função efetivamente recebe o *byte* que está chegando à interface de entrada e o coloca no registrador TEMP+2. Para isso, os estados lógicos são reconhecidos pela RECEIVE_BYTE e os intervalos são controlados pela função TEMPO. O valor 8 é transferido para um registrador temporário denominado TEMP+1 que servirá de contador de *bits*.

Como a transmissão é assíncrona, a função RECEIVE_BYTE espera pelo *start bit* (*bit* de início), recebe cada *byte* e depois deixa passar o *stop bit* (*bit* de término). A figura A.27 mostra a parte da espera do *start bit*.

```
        MOVLW     6               ;faz a contagem do TMR0 começar de 6
        MOVWF     TMR0            ;assim teremos 250 contagens no byte

espera_byte
        CLRWDT
        BTFSS     RXD             ;Enquanto RXD=1, pula 1 linha
        GOTO      continua_byte   ;Se o start bit não vier, continua após tempo
        MOVF      TMR0,W          ;Tempo do timer = 256*250 = 0,064s
        BTFSS     STATUS,Z        ;Se tempo do timer se esgotou, pula 1 linha
        GOTO      espera_byte     ;Caso ainda não tenha se esgotado, volta

continua_byte
```

Figura A.27: trecho da função RECEIVE_BYTE que espera pelo *start bit*

O TMR0 tem seu valor iniciado com o valor 6 para garantir que serão sempre realizadas 250 contagens. Enquanto isso, a função RECEIVE_BYTE fica monitorando o pino de

entrada RXD para verificar quando o *start bit* chegará à interface de entrada. Assim que o *start bit* é recebido, a função passa a receber o *byte* de informação. Enquanto isso não ocorrer, a função fica no *loop* que tem como rótulo, a designação espera_*byte*. Caso o *start bit* não seja recebido no tempo esperado pelo *timer*, a função RECEIVE_BYTE vai assumir que o *byte* pode estar sendo transmitido e começa a receber seus *bits*, na tentativa de, ainda conseguir recuperar a parte do pacote. Este procedimento não causará a corrupção do aplicativo do nó, pois o mecanismo de deteção de erro vai descartar qualquer pacote que tenha sido recebido com problemas.

Na recepção do *start bit*, a função RECEIVE_BYTE espera o tempo de um *bit* e meio para começar a ler o primeiro *bit* do *byte* que está sendo recebido (conforme foi visto na figura 3.15 da página 34). Dessa forma, as leituras são feitas na metade do *bit* e assim determinar o seu estado binário. O *loop* de recepção dos *bits* é mostrado a seguir na figura A.28.

```
receive_loop
        BCF         STATUS,C          ;Coloca zero no próximo bit
        RRF         TEMP+2,F
        BTFSC       RXD               ;Se a entrada for zero, não faz nada
        BSF         TEMP+2,MSB        ;Se a entrada for um, seta o bit
        MOVLW       TEMPO_BIT         ;Espera de tempo de um bit
        CALL        tempo

        DECFSZ      TEMP+1,F          ;Decrementa nº de bits a serem recebidos
        GOTO        receive_loop
```

Figura A.28: trecho da RECEIVE_BYTE que recebe os *bits* na interface de entrada

A função RECEIVE_BYTE inicia esse *loop* colocando zero no *bit* de *carry* do microcontrolador, pois, a instrução que rotaciona o registrador leva esse *bit* em consideração. Dessa forma, caso o *bit* lido na interface de entrada tenha sido zero, nada mais precisa ser feito. Caso o *bit* lido na interface de entrada tenha sido 1, a instrução BSF coloca 1 no *bit* mais significativo do registrador TEMP+2. Depois a função RECEIVE_BYTE decrementa o registrador TEMP+1 que contém o número de *bits* ainda a serem recebidos. Se esse decremento tiver levado seu conteúdo para zero, o fluxo do protocolo vai para a próxima instrução de retorno. Caso o conteúdo de TEMP+1 tenha sido

maior que o valor 1, retorna para o rótulo *receive_loop* e, uma nova iteração deste procedimento é realizada.

Após a função RECEIVE_BYTE ter recebido todos os *bits* do *byte*, o conteúdo do registrador TEMP+2 é transferido para o registrador de trabalho W. O *stop bit* não chega a ser recebido, pois não faz parte do *byte* útil. Entretanto, sua importância é grande para sincronizar o recebimento do próximo *byte*, uma vez que a função RECEIVE_BYTE estabelece seus intervalos de leitura de acordo com o momento que o *start bit* é detectado. Sem o *stop bit*, o nó não pode determinar o instante em que *start bit* chega, correndo o risco de realizar leituras errôneas de *bits* no decorrer do recebimento do pacote de dados ou comandos.

A.8 - Função COMANDO_EXEC

Esta função verifica se o pacote recebido contém um comando a ser realizado. Caso o pacote não contenha um comando, um valor é colocado no *byte* DB1 para indicar que não houve comando executado e a função COMANDO_EXEC retorna para a função RX_SNAP. Caso o pacote contenha um comando, este é executado e um valor referente à execução do comando é colocado no *byte* DB1 para indicar que a tarefa foi cumprida.

Na figura A.29 a seguir, pode ser vista a verificação do *bit* mais significativo do cabeçalho HDB1 que indica se o pacote contém um comando.

```
BTFSS     HDB1,MSB       ;Se é um comando, pula 1 linha
GOTO      fim_comando    ;Se não é comando, vai para fim_comando
```

Figura A.29: trecho da COMANDO_EXEC que verifica se o pacote é um comando

Caso um comando tenha sido reconhecido, a função COMANDO_EXEC deve interpretar qual foi o comando ordenado, executar as suas tarefas associadas e retornar para a função que a chamou. Esta parte do código é altamente dependente da aplicação principal, e por isso, foi arquitetada de forma modular para que o programador possa introduzir os comandos necessários a sua aplicação. Apesar de parecer que isto leva a uma alteração do protocolo, a função COMANDO_EXEC foi colocada de forma independente justamente para separar uma parte da outra.

Neste código para o PIC foram colocados dois comandos de exemplo, um acende um LED ligado em um pino da interface de saída, o outro apaga este mesmo LED, formando um par de comandos antagônicos. Um desses comandos é mostrado na figura A.30 a seguir.

```
comando2
        MOVFW     DB1                 ;Coloca no W o conteúdo do registrador DB1
        XORLW     2                   ;Verifica se o comando é o de número 2
        BTFSS     STATUS,Z            ;Caso verdadeiro, pula 1 linha
        GOTO      comando4            :Caso não seja, vai para o próximo comando
        BSF       NCD                 ;Executa comando 2: Acende LED
        CALL      limpa_dados_pacote
        MOVLW     3
        MOVWF     DB1                 ;Retorna código 3 (executou comando 2)
        GOTO      fim_comando
```

Figura A.30: trecho da função COMANDO_EXEC que exemplifica um comando

O protocolo SNAP estabelece que, se o pacote recebido for um comando, o valor numérico que o representa deve ser colocado no *byte* DB1 obrigatoriamente. Para obedecer esta exigência, a função COMANDO_EXEC transfere o valor do *byte* DB1 para o registrador de trabalho W. Depois, compara se o valor numérico corresponde ao do comando em questão. Caso o valor encontrado corresponda com o valor designado para o comando 2, a função COMANDO_EXEC o executa, limpa os dados do pacote e retorna o valor 3 em DB1, para indicar que o comando 2 foi realizado com sucesso. Caso contrário, a função COMANDO_EXEC pula para o próximo comando. Isso acontece sucessivamente, comando por comando.

Quando chegar ao último comando, e nenhum valor correspondente a ele for encontrado, a função COMANDO_EXEC desvia para o rótulo nenhum_comando, que vai colocar o valor 55 no *byte* DB1 para indicar que o comando não foi encontrado, conforme a figura A.31, retornando o fluxo para a função RX_SNAP.

```
nenhum_comando                              ;Comando não realizado
        CALL      limpa_dados_pacote        ;Limpa os bytes de dados do pacote
        MOVLW     55
        MOVWF     DB1                       ;Retorna código 55

fim_comando
        BTFSS     HDB1,MSB                  ;Verifica se o cabeçalho indica comando
        CALL      limpa_dados_pacote        ;Não sendo comando, limpa os dados

        RETURN
```

Figura A.31: trecho final da função COMANDO_EXEC de exemplo

Neste exemplo, caso o cabeçalho do pacote recebido não indique comando algum, a função COMANDO_EXEC procede com a limpeza dos *bytes* de dados. Como a missão deste protocolo é responder aos comandos da estação de campo, não faz sentido nesse contexto que o nó receba um pacote de dados sem ter transmitido um pacote anteriormente. Caso o nó esteja recebendo um pacote de dados em resposta a um pacote transmitido, a função RX_SNAP não executa a função COMANDO_EXEC, e assim os dados do pacote de resposta permanecem disponíveis para a aplicação principal.

Tendo processado o pacote recebido, a função COMANDO_EXEC retorna para a função RX_SNAP.

A.9 - Função APPLY_EDM

Esta função calcula o CRC-8 do pacote inteiro, que é o método de detecção de erro implementado. Ao se ter o CRC-8 calculado, a função APPLY_EDM deve tomar uma decisão: ou aplica este valor calculado no fim do pacote, ou compara o valor calculado com o valor que veio no fim do pacote. Esta decisão decorre do fato de que esta função pode ser invocada tanto pela função TX_SNAP quanto pela RX_SNAP. Se tiver sido chamada pela TX_SNAP, um novo CRC-8 calculado para o pacote deve ser colocado no *byte* CRC1 antes de ser transmitido. Se tiver sido chamada pela RX_SNAP, o *byte* CRC1 recebido no pacote deve ser comparado com o CRC-8 calculado para verificar se não houve erros na recepção.

Conforme pode ser visto na figura A.32 a seguir, a primeira tarefa da função é limpar o *byte* que vai acumular o cálculo para o pacote atual.

```
CLRF    TEMP+3      ;Limpa o TEMP+3 que vai acumular o EDM

MOVF    HDB2,W      ;Carrega HDB2 e chama o EDM para o byte
CALL    edm_byte
```

Figura A.32: trecho inicial da função APPLY_EDM

Repare que a função EDM_BYTE é chamada sempre que um novo *byte* do pacote é colocado no registrador de trabalho W. Isso acontece sucessivamente até que todos os *bytes* tenham passado pelo cálculo e o resultado final esteja disponível no registrador TEMP+3. A partir do valor final, a tomada de decisão pode ser vista na figura A.33.

```
BTFSS   RUN_RX        ;Se RUN_RX=1, pula 1 linha
GOTO    apply_ass     ;Se RUN_RX=0, aplica CRC-8 em CRC1
BTFSS   RUN_TX        ;Se RUN_TX=1, pula 1 linha
GOTO    apply_confr   ;Se RUN_TX=0, compara CRC-8 com CRC1
BTFSS   RUN_RTX       ;Se RUN_RTX=1, pula 1 linha
GOTO    apply_confr   ;Se RUN_RTX=0,compara CRC8 com CRC1
```

Figura A.33: trecho da função APPLY_EDM que demonstra a tomada de decisão

Esta decisão é baseada nos *bits* de controle RUN_TX, RUN_RX e RUN_RTX. Se o *bit* RUN_RX é igual a zero, significa que um novo pacote está sendo transmitido e o resultado calculado do CRC-8 deve ser aplicado no *byte* CRC1 deste mesmo pacote, para que possa ser conferido pelo nó de destino. Caso o *bit* RUN_RX tenha sido igual a 1, um novo teste será feito no *bit* RUN_TX. Esse teste leva em consideração o resultado do estado do *bit* anterior, portanto, caso RUN_TX seja igual a zero, significa que um novo pacote está sendo recebido e o resultado calculado deve ser comparado com o *byte* CRC1 desse pacote. Caso os valores sejam iguais, não houve erro detectado e o *bit* ERR é igualado a zero. Caso os valores sejam diferentes, houve erro no pacote recebido e o *bit* ERR é igualado a 1. Entretanto, caso o *bit* RUN_TX também seja igual a 1, tanto a função de transmitir como a

de receber foram invocadas. O critério de desempate, que leva a função APPLY_EDM decidir o que será feito, se encontra no estado lógico do *bit* RUN_RTX. Se o estado lógico do *bit* RUN_RTX for zero, indica que a função APPLY_EDM está recebendo um pacote de confirmação de resposta (*acknowledge*). Nesse caso, a função APPLY_EDM desvia para o rótulo apply_confr, e confere se o CRC1 recebido no pacote corresponde ao CRC-8 calculado. Se o *bit* RUN_RTX for igual a 1, indica que um pacote de confirmação de recebimento (*acknowledge*) será transmitido para o nó que iniciou a comunicação. Nesse caso, a função pula a instrução de desvio e recai sobre o rótulo apply_ass, conforme pode ser visto na figura A.34.

```
apply_ass                                  ;Aplica TEMP+3 => CRC1

        MOVF      TEMP+3,W
        MOVWF     CRC1                     ;COLOCA RESULTADO EDM NO CRC1
        GOTO      end_apply_edm            ;Vai para o fim da função APPLY_EDM
```

Figura A.34: trecho da função APPLY_EDM que aplica o resultado do CRC calculado

Este trecho da função APPLY_EDM transfere o valor calculado do CRC-8 para o *byte* CRC1 do pacote. Como o trecho mostrado na figura A.34 ficou colocado após o trecho de decisão (vide figura A.33), uma instrução de desvio incondicional do segundo trecho para o primeiro foi economizada. Assim que o valor do *byte* TEMP+3 for transferido para o *byte* CRC1, a função é desviada para o seu fim.

Caso o trecho da função APPLY_EDM, que realiza a decisão, tenha reconhecido que um pacote foi recebido, o fluxo do protocolo é desviado para o rótulo apply_confr mostrado na figura A.35 a seguir.

```
apply_confr                 ;Confere TEMP+3 com o CRC1 recebido e seta o bit ERR

        BCF       ERR                      ;Inicia o bit ERR para sem erro => ERR=0

        MOVF      TEMP+3,W
        XORWF     CRC1,W                   ;Compara TEMP+3 com o CRC1
```

```
        BTFSS       STATUS,Z            ;Se TEMP+3 = CRC1, pula 1 linha
        BSF         ERR                 ;Houve erro detectado => ERR=1
```

Figura A.35: trecho da função APPLY_EDM que compara o CRC-8 com o *byte* CRC1

A função APPLY_EDM inicia colocando o estado lógico zero no *bit* ERR. Depois, transfere o valor calculado acumulado no *byte* TEMP+3 para o registrador de trabalho W. O *byte* CRC1 do pacote recebido é comparado com o *byte* que se encontra no registrador de trabalho W. Caso os *bytes* sejam iguais, não houve erros detectados no pacote, e o *bit* ERR deve ser deixado no estado lógico inicializado. Caso os *bytes* sejam diferentes, algum erro foi detectado, e o *bit* ERR deve ter seu estado lógico alterado para 1. Após o estado lógico de ERR ser corretamente configurado, a função APPLY_EDM retorna para a função RX_SNAP.

A.10 - Função EDM_BYTE

Esta função calcula o valor do CRC-8 levando em consideração o resultado calculado anterior. Isso é realizado através do registrador temporário TEMP+3 que tem seu conteúdo inicializado em zero no início da função APPLY_EDM, e que após de chamar sucessivamente a função EDM_BYTE, acumula o resultado final do CRC-8 do pacote de dados ou comando.

Existem muitas formas de se implementar o cálculo do CRC-8 em uma linguagem de programação. Mas, ao invés de perder tempo desenvolvendo uma nova implementação, foi resolvido que seria utilizado um código extremamente otimizado da autoria de Dattalo [44]. Desenvolvido para o microcontrolador *iButton*^{TM} da *Dallas Semiconductor & Maxim Integrated Products*, o código implementado por Dattalo [44] funcionou muito bem com o PIC. A excelente otimização alcançada pode ser vista na figura A.36 a seguir.

```
edm_byte
crc8
        xorwf       TEMP+3,f
        clrw


        btfsc       TEMP+3,0
```

```
xorlw     0x5e

btfsc     TEMP+3,1
xorlw     0xbc

btfsc     TEMP+3,2
xorlw     0x61

btfsc     TEMP+3,3
xorlw     0xc2

btfsc     TEMP+3,4
xorlw     0x9d

btfsc     TEMP+3,5
xorlw     0x23

btfsc     TEMP+3,6
xorlw     0x46

btfsc     TEMP+3,7
xorlw     0x8c

movwf     TEMP+3

RETURN
```

Figura A.36: função EDM_BYTE que calcula a detecção de erro para cada *byte*

B – FUNÇÕES DO PROTOCOLO PARA O RISC16

B.1 - Função TX_SNAP

Da mesma forma que a implementação do PIC, a primeira tarefa da função TX_SNAP é desabilitar todas as interrupções para não permitir que outras tarefas possam impedir a transmissão do pacote com o sincronismo correto. Caso as interrupções não fossem desabilitadas, qualquer atendimento aos eventos que ocorressem destruiria o pacote, faria com que o microcontrolador não percebesse que o pacote não teria sido enviado e depois ocuparia a interface de saída com *bits* que não fariam mais sentido. Sem mencionar o tempo a mais que o microcontrolador teria perdido, pois estaria executando uma função que não conseguiu cumprir com sua tarefa.

Na figura B.1 a seguir pode ser visto a desabilitação das interrupções. Note que esta tarefa que consumiu uma única instrução no PIC, teve que ser desempenhada por muito mais instruções no RISC16. Primeiro, porque o RISC16 não possui um *bit* de configuração que possa desligar ou ligar todas as interrupções simultaneamente. Segundo, porque o RISC16 precisa desligar apenas um *bit* de cada *byte* e não possui instruções que tratem de *bits*.

```
LUI     $s1, $FE            #DESABILITA INTERRUPCAO SERIAL
ADDI    $s1, $FF
LUI     $s2, SERup
ADDI    $s2, SERlo
LW      $s0, $s2, $zero     #Carrega conteúdo config. serial em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da serial
SW      $s0, $s2, $zero     #Salva conteúdo na config. serial

LUI     $s1, $FF            #DESABILITA INTERRUPCAO RF
ADDI    $s1, $BF
LUI     $s2, RF2up
ADDI    $s2, RF2lo
LW      $s0, $s2, $zero     #Carrega conteúdo config. RF em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da RF
```

```
        SW          $s0, $s2, $zero     #Salva conteúdo na config. RF
```

Figura B.1: trecho da função TX_SNAP que desabilita as interrupções do RISC16

A primeira vista se tem a impressão que o protocolo SNAP Modificado no RISC16 deve ficar com um número muito maior de instruções. No entanto, apesar de haver muito menos instruções no RISC16 do que existem no PIC, esse efeito de expansão fica muito mais aparente quando o código precisa lidar com *bits* isolados, não caracterizando um crescimento demasiado, além do previsto, no tamanho total do código.

Depois de desabilitar as interrupções, o *byte* RUN_TX precisa ter seu valor maior que zero para indicar que a função TX_SNAP está sendo executada. Além disso, a função testa se o *byte* RUN_RX é maior que zero. Caso afirmativo, qualquer valor positivo é gravado no *byte* RUN_RTX para indicar que a função RX_SNAP foi executada antes da função TX_SNAP. Ambas as tarefas podem ser vistas na figura B.2 a seguir.

```
        AND         $t0, $zero, $t1     #RUN_TX>0 para indicar que TX esta
        ADDI        $t0, RUN_TX         #sendo executado
        SW          $t1, $t0, $zero

        AND         $t0, $zero, $t1     #SE RUN_RX>0, então RUN_RTX>0
        ADDI        $t0, RUN_RX
        LW          $t1, $t0, $zero
        AND         $t2, $zero, $t0
        ADDI        $t2, RUN_RTX
        BEQ         $t1, $zero, $1      #$t1 tem o valor de RUN_RX
        SW          $t0, $t2, $zero
```

Figura B.2: trecho da TX_SNAP que configura os controles RUN_TX e RUN_RTX

Repare que as figuras B.1 e B.2 acima realizam as mesmas tarefas que foram vistas na figura 3.23 no PIC. Novamente, o que o PIC fez com 4 instruções, o RISC16 precisou de 24 instruções para implementar. Por possuir um número muito menor de instruções, o RISC16 realmente gasta muito mais memória para implementar a mesma função. Entretanto, no decorrer desta dissertação, poderá ser visto que esse aumento foi

significativo, mas não impeditivo de que o código completo coubesse na escassa memória do RISC16.

Depois, a função TX_SNAP carrega o cabeçalho padrão que foi definido no arquivo SNAP16.INC. Conforme pode ser visto na figura B.3 abaixo, o *byte* RUN_RX é testado para saber se ele contém um valor maior que zero. Caso RUN_RX seja positivo, o nó está respondendo um pacote que foi recebido anteriormente. O *bit* mais significativo do ACK (*bit* 9 do HDB) tem seu valor alterado para 1, indicando que o pacote a ser transmitido é de resposta. O *bit* menos significativo do ACK (*bit* 8 do HDB) tem seu estado lógico definido de acordo com o estado lógico do *bit* ERR.

```
LUI     $a0, $02            #LIGA O BIT DE RESPOSTA DO ACK
OR      $t2, $a0, $t2       #$t2 continua tendo o valor do header
SW      $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1

AND     $a0, $zero, $t1     #SE ERR=1 NAO ZERA ULTIMO BIT ACK
ADDI    $a0, ERR
LW      $a1, $a0, $zero
BLT     $zero, $a1, $4      #Se ERR>0, então pula 4 linhas
LUI     $a0, $FE            #ZERA O ULTIMO BIT DO ACK (ACK=10)
ADDI    $a0, $FF
AND     $t2, $a0, $t2       #$t2 continua tendo o valor do header
SW      $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1
```

Figura B.3: trecho da TX_SNAP que configura os *bits* de *acknowledge*

Caso o *byte* RUN_RX seja maior que zero, a função TX_SNAP verifica se o pacote que está sendo enviado de resposta é o primeiro. Este teste é importante porque apenas no primeiro pacote de resposta que o endereço de origem do pacote recebido anteriormente deve ser transferido para o endereço de destino do pacote a ser enviado. Esse mecanismo previne que o endereço de origem do pacote que iniciou a comunicação seja perdido por um comando que exigiu múltiplos pacotes de resposta. Dessa forma, o endereço é trocado apenas no primeiro pacote de resposta, pois no resto permanece o mesmo endereço de destino. O trecho da função que garante a troca apenas no primeiro pacote de resposta é mostrado na figura B.4 a seguir.

```
AND     $a0, $zero, $t1     #Verifica se ja' HOUVE SAB=>DAB (1º)
ADDI    $a0, SAB_PARA_DAB
LW      $t0, $a0, $zero
BEQ     $t0, $zero, $1      #Se SAB_PARA_DAB=0, então pula
J       charge_sab          #Se SAB_PARA_DAB>0 então carrega SAB
AND     $t0, $zero, $t1
ADDI    $t0, SAB1
LW      $t1, $t0, $zero     #Carrega SAB1 em $t1
AND     $t0, $zero, $t1
ADDI    $t0, DAB1
SW      $t1, $t0, $zero     #SAB1 => DAB1
ADDI    $t1, $0F            #para garantir que $t1>0
SW      $t1, $a0, $zero     #Grava SAB1+0Fh p/ SAB_PARA_DAB>0
J       charge_sab
```

Figura B.4: trecho da TX_SNAP que verifica se é o primeiro pacote de resposta

Caso o *byte* RUN_RX seja igual a zero, o pacote a ser transmitido não é de resposta. A função TX_SNAP vai carregar os *bits* de *acknowledge* padrão e verificar se a aplicação principal ligou o *byte* ACK_PKT_TX. Este *byte* indica se a aplicação principal deseja ou não que o *bit* de *acknowledge* seja ligado. A função TX_SNAP vai ser coerente com a aplicação principal e alterar os *bits* de *acknowledge* do pacote a ser transmitido.

Neste ponto, é suposto que a aplicação principal tenha configurado o *byte* do endereço de destino DAB1. Pois, como o pacote a ser transmitido não é de resposta, a aplicação principal deve saber para quem transmitir o pacote.

Depois, pode ser visto na figura B.5 que a função TX_SNAP se encarrega de carregar o endereço do nó corrente no endereço de origem do pacote SAB1. Tendo quase todo o pacote definido, o próximo passo é calcular o método de detecção de erro (EDM) e incluir no final do pacote a ser transmitido. Assim, com o pacote completamente definido, a função TX_SNAP envia o pacote efetivamente através da função SEND_SNAP.

```
charge_sab
        LUI     $t0, MY_ADDR1up    #SE VOU TX, COLOCAR END. NO SAB1
        ADDI    $t0, MY_ADDR1lo
        AND     $t1, $zero, $t0
        ADDI    $t1, SAB1
        SW      $t0, $t1, $zero    #MYADDR1(up+lo)=>SAB1

        JAL     apply_edm          #CALC. EDM P/ O PKT A SER ENVIADO

        JAL     send_snap          #TRANSMITE O PACOTE
```

Figura B.5: trecho da TX_SNAP que completa o pacote e o transmite

Se o *byte* RUN_RX for positivo, então termina a execução da função TX_SNAP. Caso o *byte* RUN_RX seja igual a zero, a função TX_SNAP vai verificar se houve requisição de resposta do pacote transmitido. Caso tenha havido, a função TX_SNAP vai aguardar pelo pacote de resposta do nó de destino. Caso não tenha havido pedido de confirmação de recebimento (*acknowledge*), a função TX_SNAP retorna para a aplicação principal.

Para aguardar o pacote de resposta, a função TX_SNAP coloca algum valor no *byte* REJ para que este indique inicialmente que a resposta não foi recebida adequadamente. Caso algum pacote tenha sido recebido pelos critérios de recebimento, o valor do *byte* REJ é igualado a zero para indicar que houve coerência do pacote recebido com os filtros básicos para recebimento de pacotes pela função RX_SNAP. O segundo passo é habilitar as interrupções serial e RF (operação contrária a que foi realizada na figura B.1).

A função TX_SNAP inicia dois contadores conjugados para não ficar aguardando indefinidamente por um pacote de resposta, como pode ser visto na figura B.6.

```
wait_loop
        LW      $s3, $s1, $zero
        BLT     $zero, $s3, $1     #Se REJ>0, entao pula 1 linha
        J       exit_wait          #SE REJ=0, um pacote foi recebido, SAI
```

```
        ADDI        $s0, $01            #Incrementa o contador1
        BEQ         $s0, $zero, $1      #Se $s0=zero então pula 1 linha
        J           wait_loop

        ADDI        $s2, $01            #Incrementa o contador2
        BEQ         $s2, $zero, $1
        J           wait_loop
exit_wait
```

Figura B.6: trecho da TX_SNAP que aguarda pelo pacote de resposta

Decorrido o tempo dos contadores, a função TX_SNAP desabilita novamente as interrupções serial e RF tendo ou não recebido um pacote de resposta. Caso o pacote de resposta tenha sido recebido, o fluxo de execução da função não espera até o final das contagens e termina o *loop* de recepção.

Caso o pacote não tenha sido recebido, a função TX_SNAP desvia seu fluxo para o rótulo NAO_RECEBIDO que vai colocar o valor zero no *byte* RECEBIDO. Caso o pacote tenha sido recebido, a função TX_SNAP verifica se houve erro no pacote recebido através dos cálculo de detecção de erro do pacote de resposta. Se houve erro, a função desvia para o rótulo NAO_RECEBIDO. Caso não tenha havido erro, a função TX_SNAP testa se os *bits* do *acknowledge* do pacote recebido confirmam que o pacote é de resposta e se indica que o destinatário recebeu o pacote que iniciou a comunicação sem erros. Caso o pacote recebido seja de resposta e que indique que nenhum erro foi detectado no outro nó quanto a recepção do pacote que iniciou a comunicação, a função TX_SNAP colocar algum valor positivo no *byte* RECEBIDO para indicar para a aplicação principal que a mensagem chegou ao nó de destino sem erros. Todas as instruções que implementam o comportamento descrito podem ser vistas na figura B.7 a seguir.

```
        AND         $t1, $zero, $t0
        ADDI        $t1, REJ            #End. de REJ=>$t1
        LW          $t0, $t1, $zero
        BEQ         $t0, $zero, $1      #SE HOUVE RESPOSTA (REJ=0), PULA
        J           nao_recebido        #Se nao houve resposta --> time out
```

```
        AND     $t1, $zero, $t0
        ADDI    $t1, ERR              #End. de ERR=>$t1
        LW      $t0, $t1, $zero
        BLT     $zero, $t0, $7        #Se houve erro, VAI PARA nao_recebido
        AND     $t1, $zero, $t0
        ADDI    $t1, HDB1
        LW      $t0, $t1, $zero
        LUI     %t1, $01
        ADDI    $t1, $00
        AND     $t0, $t1, $t0
        BEQ     $t0, $zero, $1        #SE REJ=0 e ERR=0, TESTAR SE ACK=10
        J       nao_recebido
        AND     $t0, $zero, $t1
        ADDI    $t0, RECEBIDO
        SW      $t1, $t0, $zero       #Se REJ=0, ERR=0, ACK=10, então
                                      # RECEBIDO>0
        J       end_txsnap

nao_recebido
        AND     $t0, $zero, $t1
        ADDI    $t0, RECEBIDO
        SW      $zero, $t0, $zero     #Pacote não foi recebido adequadamente
```

Figura B.7: trecho da TX_SNAP que verifica o pacote de resposta recebido

O trecho apresentado da função TX_SNAP para o RISC16 (figura B.7) ficou três vezes maior que o mesmo trecho para o PIC (figura 3.28). Isso aconteceu porque foram necessárias muitas operações sobre *bits* e o RISC16 precisa de mais instruções para gerar o mesmo resultado, pois precisa criar máscaras que possam ser aplicadas sobre os *bytes*.

Na figura B.8 a seguir, a função TX_SNAP coloca o valor zero nos *bytes* RUN_RTX e RUN_TX. Depois, testa o *byte* RUN_RX para verificar se a função RX_SNAP está sendo executada. Caso negativo, a função TX_SNAP reabilita as interrupções para permitir que novos pacotes possam ser recebidos. Caso positivo, nada é realizado, pois a reabilitação das interrupções causaria um novo atendimento por parte do tratador de interrupções e

forçaria que a função RX_SNAP fosse novamente chamada sem ter sido terminada. Fica claro que essa reabilitação indevida tiraria toda a aplicação do fluxo normal, podendo esse comportamento em um efeito recursivo provocar o travamento do nó.

```
end_txsnap
        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RTX
        SW      $zero, $t0, $zero     #RUN_RTX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_TX
        SW      $zero, $t0, $zero     #RUN_TX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RX
        LW      $t1, $t0, $zero
        BEQ     $t1, $zero, $1        #Habilita interrup. se RX não estiver rodando
        J       $ra, $zero
```

Figura B.8: trecho final da função TX_SNAP

Assim, fica claro nas últimas cinco linhas da figura B.8 que caso o *byte* RUN_RX contiver algum valor positivo, a instrução BEQ vai deixar que a instrução seguinte seja executada. Caso o valor contido no *byte* RUN_RX seja igual a zero, então a instrução BEQ vai pular a linha que retorna para a aplicação principal e executar a reabilitação das interrupções. Após as interrupções terem sido reabilitadas, uma nova instrução “J $ra, $zero” fará com que a função TX_SNAP termine e obrigatoriamente retorne para a aplicação principal.

B.2 - Função SEND_SNAP

Para organizar o envio de cada *byte* do pacote de dados ou comandos, a função TX_SNAP chama a função SEND_SNAP. Esta função por sua vez chama a função SEND_BYTE para enviar um *byte* e SEND_WORD para enviar palavras de mais de um *byte*. Lembrando que *byte* neste contexto se refere a dois octetos, e não a apenas um octeto como usualmente.

Na figura B.9, pode ser visto o início da transmissão do pacote através do octeto de sincronismo SYNC. Este sincronismo é transmitido na sua forma padrão, sem usar os 16 *bits* possíveis dos registradores para manter a compatibilidade com outros dispositivos que utilizem o protocolo SNAP.

```
AND     $a0, $zero, $t2
LUI     $a0, SYNC
JAL     send_byte           #Envia octeto SYNC
```

Figura B.9: trecho da função SEND_SNAP que envia o octeto de sincronismo

Repare que o octeto de sincronismo é colocado no registrador $a0 e depois a função SEND_BYTE é chamada para transmiti-lo. O mesmo processo acontece para o cabeçalho HDB1 do pacote, os endereços de origem e destino (SAB1 e DAB1) e o *byte* do método de detecção de erros CRC1. Para o *byte* de endereço de destino DAB1, existe o cuidado de guardá-lo no registrador TEMP7 para que possa ser comparado com o pacote de resposta, caso a confirmação de recebimento tenha sido configurada pela aplicação principal. Isso pode ser verificado na figura B.10 a seguir.

```
AND     $t1, $zero, $t2
ADDI    $t1, DAB1
AND     $t2, $zero, $t1
ADDI    $t2, TEMP7
LW      $a0, $t1, $zero
SW      $a0, $t2, $zero     #Salva DAB1 => TEMP7
JAL     send_byte           #Envia byte DAB1
```

Figura B.10: trecho da SEND_SNAP que guarda o *byte* de endereço DAB1 do pacote

Conforme podem ser vistos na figura B.11, os *bytes* de dados do pacote são enviados através da função SEND_WORD. Para tal, a função SEND_SNAP aponta o ponteiro para o primeiro *byte* de dados do pacote e coloca o número de *bytes* a serem transmitidos no registrador $a0 antes de chamar a função SEND_WORD.

```
AND     $gp, $zero, $t2
ADDI    $gp, DB8          #Ponteiro aponta para DB8
AND     $a0, $zero, $t2
ADDI    $a0, $08          #$a0 contém o nº de bytes a serem enviados
JAL     send_word
```

Figura B.11: trecho da SEND_SNAP que mostra o envio de vários *bytes*

Após todos os *bytes* do pacote terem sido enviados, a função SEND_SNAP retorna para a função TX_SNAP.

B.3 - Função SEND_WORD

A função SEND_WORD inicia suas tarefas pela verificação do valor colocado no registrador $a0. Como este registrador deve conter o número de *bytes* a serem transmitidos, caso o valor de $a0 seja zero, a função SEND_WORD termina e retorna para a função que a chamou. Caso o valor seja positivo, a função SEND_WORD copia o valor do registrador $a0 para o registrador $s0, pois este é um registrador preservado quando uma função chama outras funções e tem seu valor restaurado quando o fluxo do programa retorna para a função que a chamou. Depois a função SEND_WORD salva o endereço do ponteiro no registrador TEMP, como pode ser visto na figura B.12.

```
BLT     $zero, $a0, $1    #$a0=0, então não tem bytes p/ enviar
J       end_send_word
ADD     $s0, $zero, $a0   #Copia $a0 em $s0 (registrador preservado)
AND     $t0, $zero, $t1
ADDI    $t0, TEMP
SW      $gp, $t0, $zero   #Salva ponteiro em TEMP
```

Figura B.12: trecho inicial da função SEND_WORD

A preservação dos valores de alguns registradores é um aspecto importante, pois a cada *byte* transmitido o valor do registrador $s0 é decrementado. O *loop* finito que determina quando a função SEND_WORD deve ser terminada é mostrado na figura B.13 a seguir.

```
sendw_loop
        LW      $a0, $gp, $zero     #$a0 tem o conteúdo do ponteiro
        JAL     send_byte           #Transmite o byte do registrador $a0

        AND     $t0, $zero, $t1     #INCREMENTA PONTEIRO
        ADDI    $t0, TEMP
        LW      $gp, $t0, $zero     #TEMP e' o endereço do ponteiro
        ADDI    $gp, $01
        SW      $gp, $t0, $zero     #Incrementa o endereço do ponteiro

        AND     $t1, $zero, $t0     #Decrementa contador de palavras
        ADDI    $t1, $01
        SUB     $s0, $t1, $s0       #Decrementa $s0 (registrador preservado)

        BEQ     $s0, $zero, $1      #CONTADOR $s0=0, ENTAO TERMINA
        J       sendw_loop          #Se $s0>0, então volta para o loop
```

Figura B.13: trecho da função SEND_WORD que contém o *loop* de envio de *bytes*

É importante notar que o endereço do ponteiro não é preservado no registrador $gp, por isso a cada iteração, a função SEND_WORD precisa ler o registrador TEMP com o endereço do ponteiro, incrementar este valor e gravar o novo valor novamente no registrador TEMP. Pode ser notado pelas ultimas duas linhas da figura B.13 que o *loop* termina quando o registrador $s0 é igual a zero.

B.4 - Função SEND_BYTE

A primeira tarefa da função SEND_BYTE é copiar o conteúdo do registrador $a0 para o registrador preservado $s0. A tarefa seguinte é verificar se a transmissão será serial ou por RF. Essa configuração foi definida pela aplicação principal quando configurou o *byte* TX_RF. Se TX_RF for igual a zero, a transmissão será pela interface serial. Caso o *byte* TX_RF seja positivo, a transmissão será pela interface RF. A seleção do tipo de transmissão pode ser vista na figura B.14 a seguir.

```
        AND      $t1, $zero, $t0#Verifica se a transmissão será serial ou RF
        ADDI     $t1, TX_RF
        LW       $t0, $t1, $zero#Se conteúdo de TX_RF>0, então vai p/ RF
        BEQ      $t0, $zero, $1  #Se TX_RF=0, então transmite pela serial
        J        start_rf
```

Figura B.14: trecho da SEND_BYTE que verifica se a transmissão é serial ou RF

Caso a transmissão esteja configurada para ser pela interface serial, a função SEND_BYTE carrega o endereço da transmissão serial em $s2 e inicializa o registrador $s3 com um valor não nulo. O registrador $s3 servirá de controle para transmitir cada octeto do *byte* separadamente, pois a interface serial comunica 8 *bits* de dados de cada vez. Depois o valor FFF8h é colocado no registrador $s1 para servir de contador de 8 unidades. É interessante notar que ao invés de colocar o valor 0008h no registrador para ser decrementado, a estratégia foi colocar o valor FFF8h para ser incrementado até zero. A razão de ter sido escolhido dessa forma vem da necessidade de otimização do código e da ausência de uma instrução de decremento no microcontrolador. Depois de carregado o contador, a função SEND_BYTE transmite o *start bit* conforme consta na figura B.15.

```
        LUI      $s2, TXDup          #INICIO TRANSMISSAO SERIAL
        ADDI     $s2, TXDlo          #Carrega endereço TX serial
        ADD      $s3, $zero, $t1     #Controla o octeto sendo transmitido
start_bit
        LUI      $s1, $FF
        ADDI     $s1, $F8            #n° de bits p/ enviar (FFF8h + 8 = zero)
        SW       $zero, $s2, $zero   #TRANSMITE O START BIT
        LUI      $a0, TEMPO_BITup #Carrega o tempo de um bit
        ADDI     $a0, TEMPO_BITlo
        JAL      tempo               #Espera o tempo de um bit
```

Figura B.15: trecho da SEND_BYTE que inicia a transmissão pela interface serial

Após a transmissão do *start bit*, a função SEND_BYTE transmite o primeiro *bit* de dados do registrador $s0 e rotaciona o seu conteúdo. Na figura B.16, pode ser visto como o *loop*

de transmissão transmite o *bit*, rotaciona o conteúdo do registrador e incrementa o contador até 8 vezes antes de continuar a execução da função SEND_BYTE.

```
sendb_loop
        SW       $s0, $s2, $zero     #Transmite o BIT na interface serial
        SHIFT    $s0, $01            #Deslocamento um bit para a direita

        LUI      $a0, TEMPO_BITup    #Carrega o tempo de um bit
        ADDI     $a0, TEMPO_BITlo
        JAL      tempo               #Espera o tempo de um bit

        ADDI     $s1, $01            #Incrementa o n° de bits para ser enviado
        BEQ      $s1, $zero, $1      #qdo $s1=zero (8 bits enviados), então pula
        J        sendb_loop
```

Figura B.16: trecho da SEND_BYTE que mostra o *loop* de envio pela interface serial

Terminada a transmissão do primeiro octeto, a função SEND_BYTE transmite um *stop bit* colocando o nível lógico alto na interface de saída da serial pelo tempo de um *bit*. Antes de transmitir o segundo octeto do *byte*, a função SEND_BYTE verifica se o octeto que foi transmitido foi o octeto de sincronismo, pois este é o único caso em que o segundo octeto não é transmitido. O reconhecimento do octeto de sincronismo é simples porque ele é a primeira parte do pacote a ser transmitido. Dessa forma, foi utilizado um *byte* de controle chamado de OCTSYNC. No início da função TX_SNAP, o *byte* OCTSYNC tem seu conteúdo igualado a zero para indicar que o *byte* de sincronismo ainda não foi transmitido. Na figura B.17 a seguir pode ser visto a verificação desse *byte*.

```
        AND      $t0, $zero, $t1     #Se for o SYNC, transmitir apenas 1 octeto
        ADDI     $t0, OCTSYNC
        LW       $a0, $t0, $zero
        BLT      $zero, $a0, $2      #Se OCTSYNC>0, o SYNC foi TX e continua
        SW       $t0, $a0, $zero     #Se OCTSYNC=0, este é o SYNC,
        J        fim_send_byte       #grava OCTSYNC>0 e termina a função
```

Figura B.17: trecho da SEND_BYTE que decide sobre a transmissão do *byte* de SYNC

Caso o *byte* OCTSYNC tenha seu valor igual a zero, o octeto transmitido foi o de sincronismo, então um valor positivo é escrito no *byte* OCTSYNC e a função SEND_BYTE retorna para a função SEND_SNAP. Como o valor do *byte* OCTSYNC é igualado a zero somente no início da função TX_SNAP, todos os outros *bytes* terão seus dois octetos transmitidos.

Caso o *byte* OCTSYNC tenha seu valor positivo, o octeto transmitido foi o primeiro octeto do *byte*. Na transmissão do segundo octeto (figura B.18), a função SEND_BYTE coloca o valor zero no registrador $s3 para indicar que o primeiro octeto foi transmitido, reinicializa os registradores de contagem da figura X-3 e executa novamente o *loop* de transmissão da figura X-2.

```
        BEQ     $s3, $zero, $2       #TRANSMITE O SEGUNDO OCTETO
        AND     $s3, $zero, $s1      #Zera o registrador $s3 de controle 2o octeto
        J       start_bit            #Retorna para transmitir o segundo octeto
        J       fim_send_byte        #Terminou a tx serial, vai p/ o fim da função
```

Figura B.18: parte da SEND_BYTE que decide sobre a transmissão do segundo octeto

Assim o *loop* de transmissão do primeiro octeto é reaproveitado para transmitir o segundo com apenas mais algumas instruções de controle. A transmissão pela interface serial termina, e a função SEND_BYTE retorna para a função que a chamou.

Caso a função SEND_BYTE tenha sido chamada para transmitir pacotes através da interface RF, o fluxo do programa é desviado para o código mostrado na figura B.19. A função SEND_BYTE inicia carregando o valor FFF0h no registrador de contagem $s1 e o endereço da interface RF no registrador $s2.

```
start_rf
        LUI     $s1, $FF             #INICIO TRANSMISSAO PELA RF
        ADDI    $s1, $F0             #n° de bits p/ enviar (FFF0h + 16 = zero)
        LUI     $s2, TXFup           #Carrega endereço TX RF
        ADDI    $s2, TXFlo
```

Figura B.19: trecho da SEND_BYTE que inicia a comunicação pela interface RF

A comunicação pela interface RF não precisa transmitir apenas 8 *bits* de cada vez, nem precisa iniciar transmitindo o *start bit* e terminar com o *stop bit*. Ao contrário da interface serial, a interface paralela transmite os 16 *bits* do *byte* de uma só vez. Dessa forma, o *loop* de transmissão é mais simples e não precisa de *byte* de controle. No entanto, existe um cuidado especial a ser tomado com o octeto de sincronismo, pois nesse caso, são transmitidos apenas 8 *bits*. Esse cuidado pode ser visto no código da figura B.20 a seguir.

```
        AND     $t0, $zero, $t1     #Se for o octeto SYNC, transmitir 8 bits
        ADDI    $t0, OCTSYNC
        LW      $a0, $t0, $zero
        BLT     $zero, $a0, $3      #Se OCTSYNC>0, o octeto foi TX e continua
        SW      $t0, $a0, $zero     #Se OCTSYNC=0, este é o octeto de sinc.
        LUI     $s1, $FF            #torna o n° de bits a ser enviado igual a 8,
        ADDI    $s1, $F8            #pois FFF8h + 8 = zero
```

Figura B.20: trecho da SEND_BYTE que cuida da condição do octeto de sincronismo

Novamente, o *byte* de controle OCTSYNC tem o mesmo comportamento da transmissão serial. Caso ele seja igual a zero, significa que o octeto de sincronismo ainda não foi transmitido. Como esse octeto é a primeira parte do pacote deste protocolo a ser transmitido, então a função SEND_BYTE irá transmitir apenas 8 *bits* e gravar um valor positivo no *byte* OCTSYNC para indicar que deste ponto em diante, todos as partes do pacote a serem transmitidas serão compostas por 16 *bits*. Note que transmitir 8 ou 16 *bits* é simplesmente baseado no estouro do contador que determina a saída da função do *loop* de transmissão. Este *loop* pode ser visto na próxima figura B.21.

```
sendb_loop_rf

        SW      $s0, $s2, $zero     #TRANSMITE O BIT NA INTERFACE RF
        SHIFT   $s0, $01            #Deslocamento um bit para a direita

        LUI     $a0, TEMPO_BITup    #Carrega o tempo de um bit
        ADDI    $a0, TEMPO_BITlo
        JAL     tempo               #Espera o tempo de um bit
```

```
ADDI    $s1, $01            #Incrementa o n° de bits para ser enviado
BEQ     $s1, $zero, $1      #qdo $s1=zero (16 bits enviados), então pula
J       sendb_loop_rf
```

Figura B.21: trecho da SEND_BYTE que mostra o *loop* de transmissão de RF

O *loop* de transmissão transmite um *bit* pela interface RF, desloca o registrador $s0 que contém a informação que está sendo transmitida, espera o tempo de um *bit* na chamada da função TEMPO, incrementa o registrador de contagem e testa se houve estouro desse contador (retorno para zero). Caso não tenha havido estouro, o fluxo de execução da função é desviado para o rótulo "sendb_loop_rf". Caso tenha havido estouro do registrador de contagem, a função SEND_BYTE termina e retorna para a função que a chamou.

B.5 - Função RX_SNAP

A função RX_SNAP é executada pelo tratador de interrupções. Assim que um pacote de dados ou comando estiver chegando por alguma das interfaces serial ou RF, o fluxo da aplicação principal será desviado para o tratador de interrupções. Este por sua vez, verificando que a origem da interrupção é a recepção de um pacote do protocolo, desvia o fluxo para a função RX_SNAP para que ela possa tratar e receber os dados adequadamente.

A primeira tarefa que a função RX_SNAP realiza é indicar que a recepção do pacote está sendo realizada através da colocação de algum valor positivo no *byte* RUN_RX. Depois a função Rx_SNAP limpa o *byte* OCTSYNC com o valor zero para controlar a recepção do octeto de sincronismo, guarda o valor do registrador TX_RF no registrador TX_RF2 e chama a função RECEIVE_SNAP para efetivamente receber o pacote de dados ou comando. Todo esse procedimento inicial pode ser visto na figura B.22.

```
rx_snap
        LUI     $t0, $FF
        AND     $t1, $zero, $t0
        ADDI    $t1, RUN_RX
        SW      $t0, $t1, $zero     #Indica que rx_snap esta sendo executada
```

```
AND     $t1, $zero, $t0     #BACKUP DO VALOR DE TX_RF
ADDI    $t1, TX_RF
LW      $t0, $t1, $zero     #Carrego valor TX_RF => $t0
AND     $t1, $zero, $t0
ADDI    $t1, TX_RF2
SW      $t0, $t1, $zero     #Salva valor TX_RF => TX_RF2

AND     $t0, $zero, $t1     #OCTSYNC=0 p/ receber SYNC c/ 1 octeto
ADDI    $t0, OCTSYNC
AND     $t1, $zero, $t0
SW      $t1, $t0, $zero

JAL     receive_snap        #Efetivamente recebe o pacote do protocolo
```

Figura B.22: trecho inicial da função RX_SNAP

Repare que a função RX_SNAP tem o cuidado de guardar o valor do registrador TX_RF no registrador de *backup* TX_RF2. Isso se deve porque o valor de TX_RF é configurado pela aplicação principal e determina se o pacote a ser transmitido deve utilizar a interface serial ou RF. No entanto, quando a função RX_SNAP recebe um pacote, ele pode vir de qualquer das duas interfaces. Por isso, para que o nó responda adequadamente, os pacotes que vieram pela serial são respondidos na interface serial, e pacotes que vieram pela RF na interface RF. A função de recepção muda o valor do registrador TX_RF para que o pacote de resposta seja coerente com a interface de entrada. Essa mudança destrói o valor configurado pela aplicação principal, e por isso, deve ser guardado em um registrador de backup TX_RF2 e restaurado no fim da função RX_SNAP. Assim, a aplicação principal não tem que se preocupar em ficar configurando a interface de saída toda vez que for transmitir um pacote do protocolo.

Assim como foi feito para o PIC, após o pacote ter sido efetivamente recebido pela função RECEIVE_SNAP, a função RX_SNAP testa o *byte* REJ para saber se o pacote foi rejeitado por não ter passado pelos filtros de recepção. Como pode ser visto na figura B.23, caso o pacote tenha sido rejeitado a função termina, caso o pacote não tenha sido rejeitado

a função RX_SNAP chama a função APPLY_EDM para aplicar o método de detecção de erros e colocar o resultado no *byte* ERR.

```
AND     $t1, $zero, $t0
ADDI    $t1, REJ
LW      $t0, $t1, $zero     #Carrega REJ no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se o pacote foi rejeitado, sai da rx_snap

JAL     apply_edm           #Calcula o método de detecção de erros
```

Figura B.23: trecho da função RX_SNAP que verifica a validade do pacote

A seguir, a função RX_SNAP testa o *byte* ERR para verificar se houve erros na recepção do pacote. Caso o *byte* ERR seja positivo, houve erro no pacote e a função RX_SNAP termina. Caso o *byte* ERR seja igual a zero, não houve erro no pacote e a função RX_SNAP segue seu processamento. O próximo passo é verificar se a função TX_SNAP está sendo executada. Caso esteja, a função RX_SNAP recebeu um pacote de resposta de *acknowledge* e não precisa tomar nenhuma ação. Caso a função TX_SNAP não esteja sendo executada, o pacote recebido precisa ser processado e a função RX_SNAP continua sendo executada. A verificação do *byte* ERR e do *byte* RUN_TX é mostrada na figura B.24 a seguir.

```
AND     $t1, $zero, $t0
ADDI    $t1, ERR
LW      $t0, $t1, $zero     #Carrega ERR no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se o pacote teve erro, sai da rx_snap

AND     $t1, $zero, $t0
ADDI    $t1, RUN_TX
LW      $t0, $t1, $zero     #Carrega RUN_TX no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se RUN_TX esta rodando, sai da rx_snap
```

Figura B.24: trecho da função RX_SNAP que verifica os *bytes* ERR e RUN_TX

Se o fluxo da função RX_SNAP passou das instruções mostradas na figura B.24 anterior, implica que a função TX_SNAP não está sendo executada. Como o pacote recebido ainda deve ser processado, a primeira preocupação é colocar o valor zero no *byte* SAB_PARA_DAB para indicar que o endereço de origem do pacote anterior ainda não foi transferido para o endereço de destino do próximo pacote a ser enviado, se este for o caso. Depois, a função RX_SNAP chama a função COMANDO_EXEC incondicionalmente, como pode ser visto na figura B.25.

```
runtx_0                                  #TX NAO ESTA RODANDO
        AND     $t1, $zero, $t0
        ADDI    $t1, SAB_PARA_DAB
        SW      $zero, $t1, $zero        #Limpa registrador SAB_PARA_DAB

        JAL     comando_exec             #SE FOR UM COMANDO, EXECUTA
```

Figura B.25: trecho da RX_SNAP que limpa SAB_PARA_DAB e executa comando

A chamada incondicional da função COMANDO_EXEC é realizada porque a aplicação deste protocolo estabelece que o nó escravo sempre irá obedecer comandos da estação de campo. Entretanto, para que esse aspecto desta aplicação particular não frustre a utilização deste protocolo em outras aplicações mais genéricas, uma rotina de verificação foi colocada dentro da função COMANDO_EXEC para se certificar de que o pacote recebido contém um dado ou comando. Caso seja um dado, a função COMANDO_EXEC retorna para a função RX_SNAP sem ter executado nenhuma tarefa. Maiores detalhes serão explicados posteriormente.

Se além do pacote recebido ter sido um dado ou comando, ele estiver com uma requisição de *acknowledge*, a função RX_SNAP deve responder ao nó de origem que o pacote foi recebido e em que condições esse pacote foi recebido. A verificação da requisição de *acknowledge* é mostrada na figura B.26.

```
        AND     $t1, $zero, $t0
        ADDI    $t1, HDB1
        LW      $t0, $t1, $zero          #Carrega HDB1 no registrador $t0
```

```
        LUI     $t1, $01
        ADDI    $t1, $00
        AND     $t0, $t1, $t0         #Verifica se o pacote exige resposta
        BLT     $zero, $t0, $1        #Ou recebi um novo pacote que exige resposta
        J       end_rxsnap            #Ou recebi novo pacote e termino a recepção
```

Figura B.26: trecho da função RX_SNAP que verifica a requisição do *acknowledge*

Outro ponto a ser verificado é se o pacote recebido foi transmitido em *broadcast*, pois caso tenha sido, o nó atual não pode responder imediatamente sob pena de causar uma inundação de pacotes na estação de campo, que além de causar indesejadas colisões de pacotes, ainda desperdiçam a escassa energia que cada nó consegue armazenar. O código que verifica se o pacote recebido é ou não de *broadcast* é mostrado na figura B.27.

```
espera_broadcast
        AND     $t1, $zero, $t0
        ADDI    $t1, DAB1
        LW      $t0, $t1, $zero       #Carrega DAB1 no registrador $t0
        BEQ     $t0, $zero, $1        #SE FOR BROADCAST, PULA 1 LINHA
        J       continua_resposta     #Não sendo broadcast, continua a função
```

Figura B.27: trecho da função RX_SNAP que verifica se o pacote é de *broadcast*

Repare na figura B.28 que caso o pacote seja identificado como de *broadcast*, o número do endereço do nó e grava no registrador $s1. Este valor serve de referência para um contador de espera, assim como o endereço de cada nó é único, cada nó encontrará seu intervalo de tempo para responder, sem que haja colisões entre as respostas dos nós. O *loop* de espera do nó também é mostrado na figura B.28 a seguir.

```
        LUI     $s1, MY_ADDR1up   #Como é de broadcast, guarda end. do nó
        ADDI    $s1, MY_ADDR1lo   #como parâmetro da espera da resposta
        AND     $t1, $zero, $t0
        ADDI    $t1, $01          #Serve de referencia p/ decrementar o $s1
```

```
espera_broad
        AND     $s0, $zero, $t0     #Zero contador1: $s0 de 65536 unidades.
        LUI     $s2, $FE
        ADDI    $s2, $CF            #(FECFh+131h)=0000h. Contador2: 305 u

wait_loop_rx
        ADDI    $s0, $01            #Incrementa o contador1
        BEQ     $s0, $zero, $1      #Se $s0=zero então pula 1 linha
        J       wait_loop_rx

        ADDI    $s2, $01            #Incrementa o contador2
        BEQ     $s2, $zero, $1
        J       wait_loop_rx

        SUB     $s1, $t1, $s1       #Decrementa 1 do valor do endereço do $s1
        BEQ     $s1, $zero, $1      #Se $s1=0, então sai da espera
        J       espera_broad
```

Figura B.28: trecho da função RX_SNAP que mostra o *loop* de espera de *broadcast*

Caso o pacote não tenha sido identificado como sendo de *broadcast*, o fluxo da função RX_SNAP é desviado da figura B.27 para a figura B.29 a seguir. Como o pacote que mesmo não sendo de *broadcast* ainda exige resposta (*acknowledge*), a função RX_SNAP chama a função TX_SNAP para enviar a resposta. Tendo realizado todas as tarefas a que foi destinado a função RX_SNAP termina colocando o valor zero no *byte* RUN_RX para indicar que não mais está sendo executada, restaura o valor do registrador TX_RF2 para o registrador TX_RF e retorna para o tratador de interrupções.

```
continua_resposta
        JAL     tx_snap             #Responde caso TX não esteja rodando

end_rxsnap
        AND     $t1, $zero, $t0     #RESTAURA O VALOR DE TX_RF
        ADDI    $t1, TX_RF2
```

```
LW      $t0, $t1, $zero     #Carrego valor TX_RF2 => $t0
AND     $t1, $zero, $t0
ADDI    $t1, TX_RF
SW      $t0, $t1, $zero     #Salva valor TX_RF2 => TX_RF

AND     $t0, $zero, $t1
ADDI    $t0, RUN_RX
SW      $zero, $t0, $zero   #Zera o registrador RUN_RX

J       $ra, $zero          #FIM DA RX_SNAP
```

Figura B.29: trecho final da função RX_SNAP

O tratador de interrupções restaura todo o ambiente do microcontrolador no momento da interrupção da aplicação principal, deixando que esta assuma novamente o controle do microcontrolador como se nada tivesse ocorrido.

B.6 - Função RECEIVE_SNAP

No instante em que é chamada pela função RX_SNAP, a primeira tarefa da função RECEIVE_SNAP é receber o octeto de sincronismo do pacote através da função RECEIVE_BYTE. Tendo recebido o octeto de sincronismo, a função RECEIVE_SNAP vai impondo diversos filtros simples ao pacote que está sendo recebido. O objetivo dos filtros é proporcionar que o nó possa parar de receber o pacote o quanto antes ele perceber que seu conteúdo não terá utilidade. A seguir serão explicados os usos dos filtros juntamente com os códigos que os implementam.

A função RECEIVE_SNAP chama novamente a função RECEIVE_BYTE para receber o *byte* de cabeçalho do pacote HDB1 na figura B.30 a seguir.

```
JAL     receive_byte        #RECEBE O BYTE DE HEADER (HDB1)
AND     $t0, $zero, $t1
ADDI    $t0, HDB1
SW      $a0, $t0, $zero     #Salva byte lido ($a0) em HDB1
```

Figura B.30: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1

Tendo recebido o *byte* HDB1, a função RECEIVE_SNAP toma uma decisão sobre o tratamento do pacote baseado no fato da função TX_SNAP estar ou não sendo executada. Essa decisão é mostrada na figura B.31.

```
AND     $t0, $zero, $t1
ADDI    $t0, RUN_TX        #Verifica se tx_snap está sendo executada
LW      $t1, $t0, $zero
LUI     $t0, HEADER2
ADDI    $t0, HEADER1
BLT     $zero, $t1, $1     #Se tx_snap está executando, então pula
J       run_tx0
```

Figura B.31: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1

Essa decisão é importante para que a função RECEIVE_SNAP saiba como tratar o pacote. Caso a função TX_SNAP esteja sendo executada, a função RECEIVE_SNAP deve estar recebendo um pacote de resposta. Se este é o caso, então no cabeçalho recebido, os *bits* de ACK devem ser obrigatoriamente iguais a 10b ou 11b. Caso eles sejam diferentes, o pacote recebido não é o que está sendo esperado e a função RECEIVE_SNAP desvia para o código que assinala que o pacote foi rejeitado, retornando para a função RX_SNAP. A função RX_SNAP por sua vez retornará para a função TX_SNAP, onde o *loop* de espera pela resposta do pacote continuará sua execução até que seu tempo se expire ou que um pacote certo tenha sido recebido. Os filtros dos *bits* do ACK do pacote podem ser vistos na figura B.32 a seguir.

```
LUI     $t2, $02           #Estou recebendo um pacote de resposta
ADDI    $t2, $00
OR      $t1, $t2, $t0
LUI     $t2, $FE
ADDI    $t2, $FF
AND     $t1, $t2, $t1      #ACK[$t1]=10
XOR     $s0, $t1, $a0      #HDB1=HEADER[ACK=10]?
LUI     $t2, $01
```

```
        ADDI     $t2, $00
        OR       $t1, $t2, $t1      #ACK[$t1]=11
        XOR      $s1, $t1, $a0      #HDB1=HEADER[ACK=11]?
        AND      $s2, $s1, $s0      #Junta as duas comparações feitas
        BEQ      $s2, $zero, $1     #Rejeita se ACK é diferente de 10b ou 11b
        J        reject
        J        receive_dab        #Se ACK é igual a 10b ou 11b, continua
```

Figura B.32: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1

Caso a função TX_SNAP não esteja sendo executada, a função RECEIVE_SNAP verifica se o pacote recebido tem o *byte* HDB1 igual ao cabeçalho transmitido pelo nó nos pacotes, com exceção se o cabeçalho indica que há um comando no pacote ou se o pacote exige confirmação de recebimento. Essas verificações podem ser vistas na figura B.33.

```
run_tx0          #Se tx_snap não está executando, então recebo novo pacote
        LUI      $t2, $01
        ADDI     $t2, $00
        OR       $t1, $t2, $a0      #Seta o bit de ACK em $t1
        LUI      $t2, $FF
        ADDI     $t2, $7F
        AND      $t1, $t2, $t1      #Zera o bit de comando em $t1
        XOR      $s0, $t1, $t0      #$t0 ainda contém o cabeçalho
        BEQ      $s0, $zero, $1
        J        reject             #Não sendo rejeitado, a função continua
```

Figura B.33: trecho da função RECEIVE_SNAP que recebe o *byte* HDB1

Uma vez que o pacote em questão passou pelos filtros do cabeçalho, a função RECEIVE_SNAP recebe o *byte* de endereço de destino do pacote e verifica se é de *broadcast* ou se foi endereçado ao nó corrente. As verificações podem ser vistas na figura B.34, caso o pacote não atenda a qualquer uma desses critérios, a função RECEIVE_SNAP rejeita o pacote.

```
receive_dab
        JAL     receive_byte        #RECEBE O BYTE DE DESTINO (DAB1)
        AND     $t0, $zero, $t1
        ADDI    $t0, DAB1
        SW      $a0, $t0, $zero     #Salva byte lido ($a0) em DAB1

        XOR     $t1, $zero, $a0     #VERIFICA SE E' BROADCAST
        BLT     $zero, $t1, $1      #Caso negativo testa se é o end. deste nó
        J       receive_sab         #Caso positivo continua a função

        LUI     $t2, MY_ADDR1up     #Se MY_ADDR1 !=  DABi, então rejeita
        ADDI    $t2, MY_ADDR1lo
        XOR     $t1, $t2, $a0
        BEQ     $t1, $zero, $1      #o pacote e termina a função
        J       reject              #Caso não seja rejeitado, continua
```

Figura B.34: trecho da função RECEIVE_SNAP que verifica o endereço de destino

Caso a função não tenha rejeitado o pacote que está sendo recebido até este ponto, o próximo teste é receber o *byte* do endereço de origem e verificar se ele é diferente do endereço do nó. Caso seja igual, existe um outro nó com o mesmo endereço na rede e este pacote deve ser rejeitado conforme mostra a figura B.35.

```
receive_sab
        JAL     receive_byte        #RECEBE O BYTE DE ORIGEM (SAB1)
        AND     $t0, $zero, $t1
        ADDI    $t0, SAB1
        SW      $a0, $t0, $zero     #Salva byte lido ($a0) em SAB1

        LUI     $t2, MY_ADDR1up     #Se SAB1 for igual ao MY_ADDR1 então
        ADDI    $t2, MY_ADDR1lo
        XOR     $t1, $t2, $a0       #existem 2 nós com o mesmo endereço e
        BLT     $zero, $t1, $1
        J       reject              #o presente pacote e' rejeitado
```

Figura B.35: parte da função RECEIVE_SNAP que verifica a duplicidade de endereços

Como o pacote tem endereço de origem diferente do endereço do nó corrente, o próximo passo é verificar se a função TX_SNAP está sendo executada. Caso a função TX_SNAP esteja sendo executada, a função RECEIVE_SNAP está recebendo um pacote de resposta, então a função verifica se o pacote recebido provém do nó do qual o pacote anterior endereçou como destino. Se for outro nó que respondeu, a função RECEIVE_SNAP rejeita o pacote. Caso a função TX_SNAP não esteja sendo executada, a função RECEIVE_SNAP continua seu fluxo, recebendo os *bytes* de dados do pacote. Todo esse trâmite é visto na figura B.36.

```
also_sab
        AND         $t2, $zero, $t1
        ADDI        $t2, RUN_TX         #Checa se tx_snap está sendo executada
        LW          $t1, $t2, $zero
        BLT         $zero, $t1, $1
        J           receive_db          #Caso não esteja, pula para receive_db

        AND         $t2, $zero, $t1     #Caso esteja, verifica se a resposta veio
        ADDI        $t2, TEMP7
        LW          $t0, $t2, $zero     #do nó para o qual o pacote foi enviado
        XOR         $t1, $t0, $a0       #Se tiver vindo do nó o qual o pacote
        BEQ         $t1, $zero, $1      #foi transmitido anteriormente, pula 1 linha
        J           reject              #Caso seja de outro nó, rejeita o pacote
```

Figura B.36: parte da RECEIVE_SNAP que verifica se o nó certo enviou a resposta

Se a função TX_SNAP está sendo executada e o nó correto enviou a resposta, ou se a função TX_SNAP não está sendo executada, a função RECEIVE_SNAP recebe os *bytes* de dados do pacote conforme pode ser visto na figura B.37.

```
receive_db
        LUI         $gp, $00
        ADDI        $gp, DB8            #Aponta ponteiro para o DB8
        LUI         $a0, $00
        ADDI        $a0, $08            #Indica numero de palavras a receber
```

```
        JAL         receive_word        #RECEBE OS BYTES DE DADOS (DBi)
```

Figura B.37: trecho da RECEIVE_SNAP que recebe os *bytes* de dados do pacote

Repare que a recepção dos *bytes* de dados é feita através da função RECEIVE_WORD que será explicada posteriormente. Depois de recebidos os *bytes* de dados, a função RECEIVE_SNAP recebe o *byte* do método de detecção de erros CRC1 com o código mostrado na figura B.38.

```
receive_eb
        JAL         receive_byte        #RECEBE O BYTE DE CRC-8 (CRC1)
        AND         $t0, $zero, $t1
        ADDI        $t0, CRC1
        SW          $a0, $t0, $zero     #Salva byte lido ($a0) em CRC1

        AND         $t0, $zero, $t1
        ADDI        $t0, REJ
        SW          $zero, $t0, $zero   #Recebido pacote a ser verificado = REJ=0
        J           end_receivesnap
```

Figura B.38: trecho da RECEIVE_SNAP que recebe o *byte* da detecção de erro

Note que, caso a função RECEIVE_SNAP não tenha rejeitado o pacote em nenhum dos filtros mostrados anteriormente, o valor zero é gravado no *byte* REJ para indicar que o pacote foi recebido adequadamente. Caso algum filtro tenha feito a função RECEIVE_SNAP rejeitar o pacote, o fluxo de sua execução é desviado para o código descrito a seguir na figura B.39.

```
reject
        LUI         $t1, $FF
        AND         $t0, $zero, $t1
        ADDI        $t0, REJ
        SW          $t1, $t0, $zero     #PACOTE REJEITADO = REJ>0
```

Figura B.39: trecho da RECEIVE_SNAP que trata da rejeição do pacote

Tendo determinado o valor do *byte* REJ, a função RECEIVE_SNAP termina retornando para a função RX_SNAP.

B.7 - Função RECEIVE_WORD

A função RECEIVE_WORD é chamada para receber vários *bytes* de uma só vez porque é baseada em um contador de número de *bytes* a receber e um ponteiro que aponta para o próximo registrador a ser gravado. Então, a função RECEIVE_WORD verifica se o número de *bytes* é maior que zero. Se for igual a zero, a função termina e retorna para a função SEND_SNAP. O número de *bytes* a receber é colocado inicialmente no registrador $a0, mas a função RECEIVE_WORD trata de transferir esse valor para um registrador preservado $s0 como pode ser visto na figura B.40.

```
receive_word
        BLT         $zero, $a0, $1          #Verifica se $a0 é igual a zero
        J           end_receiveword         #Se #a0=0, então vai para o fim
        AND         $t0, $zero, $t1
        ADDI        $t0, $01
        ADD         $s0, $zero, $a0         #Transfere $a0 para $s0
```

Figura B.40: trecho inicial da função RECEIVE_WORD

Depois de preparada, a função RECEIVE_WORD entra no *loop* de recepção que consiste em ler o *byte* na interface de entrada, gravar o *byte* lido no endereço que o ponteiro está apontando, incrementar o endereço do ponteiro, decrementar o número de *bytes* a serem recebidos, testar para ver se o número de *bytes* a receber chegou a zero e condicionar o teste anterior ao retorno para o *loop* de recepção. Todo esse procedimento pode ser visto na figura B.41 a seguir.

```
receivew_loop
        JAL         receive_byte            #Lê o byte na interface de entrada => $a0

        SW          $a0, $gp, $zero         #Coloca o $a0 no conteúdo do ponteiro
        ADDI        $gp, $01                #Incrementa o endereço do ponteiro
        SUB         $s0, $t0, $s0           #Decrementa o contador de bytes a receber
```

```
        BEQ         $s0, $zero, $1
        J           receivew_loop
```

Figura B.41: trecho da função RECEIVE_WORD que evidencia o *loop* de recepção

Ao sair do *loop* de recepção, a função RECEIVE_WORD termina e retorna para a função RECEIVE_SNAP.

B.8 - Função RECEIVE_BYTE

A função RECEIVE_BYTE é a função que recebe efetivamente o *byte* do pacote através da interface física. Uma das funções mais importantes é o reconhecimento de qual a interface pela qual o pacote está sendo recebido. Como o tratador de interrupção vai sempre chamar a RX_SNAP para tratar das interrupções das interfaces serial e RF, se faz necessário analisar o conteúdo do registrador $int do microcontrolador para se verificar a interface que está recebendo o pacote. A vantagem deste procedimento é simplificar o protocolo, pois a única mudança para se receber *bits* pela interface serial ou pela interface RF é na função RECEIVE_BYTE. A figura B.42 a seguir mostra que os *bits* serão recebidos no registrador preservado $s0 e como é feita a verificação que leva o recebimento desses *bits* pela interface correta.

```
receive_byte
        AND         $s0, $zero, $t0         #Prepara para receber byte em $s0

        LW          $t0, $int, $zero        #Verifico registrador $int para saber a
        AND         $t1, $zero, $t0         #origem da interrupção
        ADDI        $t1, $01                #Estas funções somente são executadas
        AND         $t1, $t0, $t1           #se ocorrer int de RF ou int de Serial, então
        BEQ         $t1, $zero, $1          #se tiver 0 no final e' Serial
        J           start_rx_rf             #Se tiver 1 no final e' RF
```

Figura B.42: trecho inicial da função RECEIVE_BYTE

Dessa forma, caso a interrupção tenha sido originada pela interface serial, a instrução BEQ vai pular a instrução de salto incondicional e inicia a recepção serial como pode ser visto na figura B.43 a seguir.

```
        LUI         $s2, RXDup           #INICIO RECEPCAO SERIAL
        ADDI        $s2, RXDlo
        ADD         $s3, $zero, $s2      #Controla octeto que esta sendo recebido
recebe_start_bit
        LUI         $s1, $FF             #No de bits a receber (FFF8h + 8 = zero)
        ADDI        $s1, $F8
        AND         $t0, $zero, $t1      #Zero contador1: $t0 de 65536 unidades.
        AND         $t1, $zero, $t0
        ADDI        $t1, TX_RF
        SW          $t0, $t1, $zero      #Zera registrador TX_RF para indicar serial
        LUI         $t2, $FF
        ADDI        $t2, $6A        #(FF6Ah + 96h)=0000h. Contador2: 150 unidades.
```

Figura B.43: trecho da RECEIVE_BYTE que inicializa a recepção pela interface serial

Como condições iniciais para a recepção serial, o registrador $s2 guarda o endereço do registrador que fisicamente acessa o pino de entrada do RISC16. Enquanto que o registrador $s3 serve para controlar se o octeto que está sendo recebido é o primeiro ou o segundo do *byte*. Também são definidos dois contadores, um de 16 *bits* e o outro que conta 150 unidades antes de retornar para o valor zero. Combinados, os dois contadores implementam um tempo de espera pelo *start bit* de cada octeto do pacote, pois a comunicação serial foi estabelecida como assíncrona. Então, a função RECEIVE_BYTE grava o valor zero no registrador TX_RF para indicar que o pacote está sendo recebido pela interface serial. Caso uma resposta seja requerida pelo pacote recebido, ela deve ser enviada pela interface serial, para manter a coerência da comunicação. O *loop* de espera pelo *start bit* é mostrado na figura B.44 a seguir.

```
espera_byte
        LW          $t1, $s2, $zero      #Não usa mascara porque 15 bits são terra
        BLT         $zero, $t1, $1       #Se $t1>0, então pula 1 linha
        J           continua_byte        #Se $t1=0, então um bit foi recebido, sai

        ADDI        $t0, $01             #Incrementa o contador1
```

```
        BEQ     $t0, $zero, $1          #Se $s0=zero então pula 1 linha
        J       espera_byte

        ADDI    $t2, $01                #Incrementa o contador2
        BEQ     $t2, $zero, $1
        J       espera_byte
```

Figura B.44: trecho da RECEIVE_BYTE que mostra o *loop* de espera do *start bit*

A função RECEIVE_BYTE fica executando o *loop* de espera do *start bit* até que os contadores se esgotem ou que o *start bit* seja recebido, então o código mostrado na figura B.45 é executado.

```
continua_byte
        LUI     $t0, TEMPO_BITup              #Tempo de um bit
        ADDI    $t0, TEMPO_BITlo
        LUI     $t1, TEMPO_MEIO_BITup  #Tempo de meio bit
        ADDI    $t1, TEMPO_MEIO_BITlo
        ADD     $a0, $t0, $t1
        JAL     tempo                   #ESPERA O TEMPO DE UM BIT E MEIO
```

Figura B.45: trecho da RECEIVE_BYTE que espera o tempo de um *bit* e meio

A função RECEIVE_BYTE espera o tempo de um *bit* para deixar passar o *start bit* e o tempo de meio *bit* para ler o primeiro *bit* do octeto. As duas constantes de tempo foram somadas no registrador $a0 e a função TEMPO é chamada para gerar o tempo de um *bit* e meio. Depois, a função RECEIVE_BYTE entra no *loop* de recepção do octeto conforme pode ser visto na figura B.46.

```
receive_loop
        LW      $t1, $s2, $zero         #LE O BIT DA INTERFACE SERIAL
        BEQ     $t1, $zero, $1          #Se for zero, pula
        ADDI    $s0, $01                #Se for um, liga o bit menos significativo
        SHIFT   $s0, $01                #Deslocamento de um bit para a direita
```

```
LUI     $a0, TEMPO_BITup   #Tempo de um bit
ADDI    $a0, TEMPO_BITlo
JAL     tempo              #ESPERA O TEMPO DE UM BIT

ADDI    $s1, $01           #Incrementa o num de bits recebidos
BEQ     $s1, $zero, $1     #Se todos os 8 bits foram recebidos, pula
J       receive_loop       #Se não foram, volta p/ receive_loop
```

Figura B.46: trecho da RECEIVE_BYTE que recebe os *bits* pela interface serial

Depois de recebido o primeiro octeto do *byte*, a função RECEIVE_BYTE verifica se o octeto recebido foi o octeto de sincronismo através do *byte* OCTSYNC. Assim como foi feito na parte da transmissão, o *byte* OCTSYNC tem seu valor igual a zero para indicar que o octeto de sincronismo não foi transmitido, e o valor maior que zero para indicar que o octeto de sincronismo já foi transmitido. Na figura B.47 a seguir pode ser visto como a função RECEIVE_BYTE verifica se o octeto de sincronismo foi recebido ou não.

```
AND     $t0, $zero, $t1    #Se for o sync, receber apenas 1 octeto
ADDI    $t0, OCTSYNC
LW      $a0, $t0, $zero
BLT     $zero, $a0, $2     #Se OCTSYNC>0, o octeto foi TX e continua
SW      $t0, $a0, $zero    #Se OCTSYNC=0, é o octeto de sincronismo
J       fim_receive_byte   #então vai para o fim da função
```

Figura B.47: trecho da RECEIVE_BYTE que verifica se é o octeto de sincronismo

Caso o octeto de sincronismo não tenha sido recebido anteriormente, o octeto recebido é considerado como o octeto de sincronismo, um valor positivo é gravado no *byte* OCTSYNC e a função RECEIVE_BYTE termina. Caso o octeto de sincronismo já tenha sido recebido, a função RECEIVE_BYTE verifica se o octeto recebido foi o primeiro octeto, ou o segundo, de acordo com o valor gravado no registrador $s3 e com o código mostrado na figura B.48.

```
BEQ     $s3, $zero, $2     #RECEBE O SEGUNDO OCTETO
AND     $s3, $zero, $s1    #Zera registrador $s3 de controle 2o octeto
```

```
        J           recebe_start_bit
        J           fim_receive_byte      #Terminou a recepção serial, sai da função
```

Figura B.48: trecho da RECEIVE_BYTE que verifica qual foi o octeto recebido do *byte*

Caso o octeto recebido tenha sido o primeiro do *byte*, a função RECEIVE_BYTE grava o valor zero no registrador $s3 e retorna para o rótulo "recebe_start_*bit*". Caso o octeto recebido tenha sido o segundo do *byte*, a função RECEIVE_BYTE desvia para o rótulo "fim_receive_*byte*" e retorna para a função que a chamou. A recepção pela interface serial termina neste trecho do código.

Caso a interrupção tenha sido originada pela interface RF, a instrução BEQ da figura B.42 vai deixar que a instrução de salto incondicional para o rótulo "start_rx_rf" seja realizada, fazendo com que a função RECEIVE_BYTE pule toda a parte da recepção serial para executar a parte da recepção pela interface de RF. O início da recepção pela interface RF é mostrada a seguir na figura B.49.

```
start_rx_rf
        LUI         $s1, $FF              #INICIO DA RECEPCAO PELA RF
        ADDI        $s1, $F0              #Num de bits p/ receber (FFF0h+16=zero)
        LUI         $s2, RXFup
        ADDI        $s2, RXFlo            #Carrega end. da RX RF
```

Figura B.49: trecho da RECEIVE_BYTE que inicia a recepção pela interface RF

O endereço de memória que pode ler fisicamente o sinal presente no pino de entrada do microcontrolador RISC16 é gravado no registrador $s2. O número de *bits* a serem recebidos pela interface RF é 16, mas para facilitar a contagem, o valor FFF0h é gravado no registrador $s1 para que com 16 contagens o registrador tenha o valor zero. O próximo passo é verificar se o octeto de sincronismo já foi recebido, pois este é o única parte do pacote que tem apenas 8 *bits*. A verificação do octeto de sincronismo pode ser vista na figura B.50 a seguir.

```
        AND         $t0, $zero, $t1       #Se for o SYNC, receber apenas 1 octeto
        ADDI        $t0, OCTSYNC
```

```
LW      $a0, $t0, $zero
BLT     $zero, $a0, $3      #Se OCTSYNC>0, o octeto foi TX e continua
SW      $t0, $t0, $zero     #Se OCTSYNC=0, é o octeto de sincronismo
LUI     $s1, $FF            #torna o n° de bits a ser recebido igual a 8,
ADDI    $s1, $F8            #pois FFF8h + 8 = zero
```

Figura B.50: trecho da RECEIVE_BYTE que verifica se é o octeto de sincronismo

Como pode ser notado, este último trecho de código é quase igual ao da recepção por RF. A diferença fica por conta da transmissão serial se efetivar de 8 em 8 *bits*, então caso o octeto recebido seja o de sincronismo, basta terminar a função RECEIVE_BYTE que o protocolo mantém a estrutura do pacote intacta; e na transmissão RF, basta que o contador de *bits* a receber seja alterado de 16 para 8 *bits*, caso o octeto de sincronismo ainda não tenha sido recebido, para que a estrutura do pacote seja preservada. Assim, a dificuldade inicial de ter uma parte do pacote com 8 *bits* e as outras com 16 fica contornada, tanto na interface serial quanto na RF, preservando a compatibilidade com outros dispositivos SNAP.

Depois de verificar se a parte recebida é o octeto de sincronismo, a função RECEIVE_BYTE grava um valor positivo no registrador TX_RF para indicar que caso o pacote recebido exija *acknowledge*, a resposta deve ser enviada pela interface RF. O código mostrado na figura B.51 a seguir aproveita que o registrador $s1 contém um valor positivo para gravar no registrador TX_RF.

```
AND     $t1, $zero, $t0
ADDI    $t1, TX_RF
SW      $s1, $t1, $zero     #Registrador TX_RF>0 para indicar RF
```

Figura B.51: trecho da RECEIVE_BYTE que grava valor positivo no TX_RF

A transmissão por RF não utiliza *start* ou *stop bit*, então pode ser visto na figura B.52 que o tempo de espera para se ler o primeiro *bit* é limitado ao tempo de meio *bit*.

```
LUI     $a0, TEMPO_MEIO_BITup
ADDI    $a0, TEMPO_MEIO_BITlo
```

```
        JAL         tempo               #Espera meio bit para ler os bits no meio
```

Figura B.52: trecho da RECEIVE_BYTE que espera o tempo de meio *bit*

A função TEMPO será explicada posteriormente, mas como pode ser visto, ela recebe o valor da espera através do registrador $a0. Depois que o tempo de meio *bit* foi gerado, a função RECEIVE_BYTE entra no *loop* de recepção pela interface de RF como pode ser visto na figura B.53.

```
receive_loop_rf
        LW          $t1, $s2, $zero     #LE O BIT DA INTERFACE RF
        BEQ         $t1, $zero, $1      #Se for zero, pula
        ADDI        $s0, $01            #Se for um, liga o bit menos significativo
        SHIFT       $s0, $01            #Deslocamento de um bit para a direita

        LUI         $a0, TEMPO_BITup    #Tempo de um bit
        ADDI        $a0, TEMPO_BITlo
        JAL         tempo               #ESPERA O TEMPO DE UM BIT

        ADDI        $s1, $01            #Incrementa o num de bits recebidos
        BEQ         $s1, $zero, $1      #Se todos os 16 bits foram recebidos, pula
        J           receive_loop_rf     #Se não foram, retorna p/ receive_loop_rf
```

Figura B.53: trecho da RECEIVE_BYTE que mostra o *loop* de recepção pela RF

Após sair do *loop* de recepção da interface RF, a função RECEIVE_BYTE termina sua tarefa e retorna para a função que a chamou.

B.9 - Função COMANDO_EXEC

Como a chamada da função COMANDO_EXEC ocorre incondicionalmente pela função RX_SNAP, fica a responsabilidade de verificar se o pacote recebido contém um dado ou comando. Lembrando que esta função somente é executada quando a função TX_SNAP não está sendo executada. A verificação do comando é realizada pelo código mostrado na figura B.54 a seguir.

```
comando_exec                                    #TX NAO ESTA RODANDO
        AND         $t1, $zero, $t0
        ADDI        $t1, HDB1
        LW          $t0, $t1, $zero             #Carrega HDB1 => $t0
        AND         $t1, $zero, $t0
        ADDI        $t1, $80
        AND         $t1, $t0, $t1               #$t1=(HDB1 & 0080h) para ver se e' comando
        BLT         $zero, $t1, $1
        J           fim_comando
```

Figura B.54: trecho da COMANDO_EXEC que verifica se o pacote contém um comando

Essa verificação consiste na avaliação do *bit* 7 do cabeçalho HDB1, através da operação lógica AND com o HDB1 e a máscara 0080h. Caso o resultado seja maior que zero, a instrução BLT pula a instrução de salto incondicional e começa a verificar qual é o código do comando recebido. Caso o resultado seja igual a zero, o pacote recebido não contém um comando, a instrução de salto incondicional é executada, conduzindo para o final da função COMANDO_EXEC que retorna para a função RX_SNAP.

Neste trecho do código, os comandos devem ser programados de acordo com as ações desejadas de cada nó. Abertura e fechamento de válvulas, leituras de dados dos sensores e outros tipos de controle de equipamentos adicionados os nós podem ser implementados neste espaço do código. Como estas definições estão em constante mudança, assim como os equipamentos controlados, a função COMANDO_EXEC foi deixada como exemplo para que possa ser aperfeiçoada e readequada para futuras necessidades. Na figura B.55 a seguir é mostrada uma função de exemplo.

```
comando2            #Exemplo de comando
        AND         $t1, $zero, $t0
        ADDI        $t1, DB1
        LW          $t0, $t1, $zero             #Carrega comando que fica em DB1
        AND         $t1, $zero, $t0
        ADDI        $t1, $02                    #Verifica o código 2 do comando
        XOR         $t2, $t1, $t0
```

```
        BEQ         $t2, $zero, $1        #Se for igual, pula 1 linha
        J           comando4              #Próximo comando
        #.
        #ação do comando: abre válvula, desliga válvula, limpa memória, etc
        #.
        J           fim_comando
```

Figura B.55: parte da COMANDO_EXEC que verifica qual comando o pacote contém

Note que a instrução "ADDI $t1,$02" é executada para que a instrução XOR seguinte faça o teste do código do comando com o valor 2. Caso o código do comando recebido no pacote seja igual a 2, a ação do comando é executada e uma instrução de salto desvia para o fim da função. Caso o código do comando seja diferente de 2, a instrução de salto desvia o fluxo para o próximo comando. Este procedimento é realizado comando por comando até que o comando em questão seja o último da lista, conforme é mostrado na figura B.56.

```
comando4            #Exemplo de comando
        AND         $t1, $zero, $t0
        ADDI        $t1, $04              #Verifica o código 4 do comando
        XOR         $t2, $t1, $t0
        BEQ         $t2, $zero, $1        #Se for igual, pula 1 linha
        J           nenhum_comando        #Ultimo comando vai p/ nenhum_comando
        #.
        #ação do comando: transmite todos dados para a estação de campo, etc
        #.
        J           fim_comando
```

Figura B.56: trecho da COMANDO_EXEC que mostra como é o último comando da lista

No caso do "comando4" que é o último comando da lista, a diferença é que como não existe nenhum outro comando depois dele, caso o código do comando não seja encontrado, a função COMANDO_EXEC desvia seu fluxo para o rótulo nenhum_comando. Este rótulo serve para que a função COMANDO_EXEC possa tomar alguma atitude quando um pacote de comando é recebido, mas nenhum comando é executado. Isso pode ser um código de erro ou qualquer outro tipo de informação de controle que pode ser enviado de

volta para a estação central, caso o pedido de confirmação de recebimento (*acknowledge*) esteja ligado. Assim como os comandos da lista também podem enviar um código de confirmação de comando executado com sucesso para a estação central, caso o *acknowledge* esteja ligado.

Não havendo mais ações para serem desempenhadas, a função COMANDO_EXEC retorna para a função RX_SNAP.

B.10 - Função APPLY_EDM

A função APPLY_EDM é responsável pela preparação das partes dos pacotes do protocolo para serem passadas pelo cálculo do método de deteção de erro (*Error Detection Method* - EDM). Além disso, também é responsável pela ação tomada com o resultado do cálculo, pois a função APPLY_EDM deve distinguir se foi chamada pelas funções TX_SNAP ou RX_SNAP, ou por uma combinação delas no caso de haver sido requisitada uma confirmação do recebimento dos pacotes (*acknowledge*).

A primeira tarefa da função APPLY_EDM é colocar o valor zero no endereço de memória TEMP3 que vai acumular o resultado do método de detecção de erro, conforme pode ser visto na figura B.57.

```
apply_edm
        AND         $t1, $zero, $t0         #Prepara para calcular EDM em TEMP3
        ADDI        $t1, TEMP3
        SW          $s0, $t1, $zero         #Zera registrador temporário TEMP3
```

Figura B.57: trecho da APPLY_EDM que inicia o método de detecção de erro

A partir de iniciada, a função APPLY_EDM transfere parte por parte do pacote para o registrador $a0 e chama a função EDM_BYTE para calcular o valor parcial do método de detecção de erros. A figura B.58 mostra o cálculo para o cabeçalho do pacote.

```
start_edm
        AND         $t1, $zero, $t0         #Carrega HDB1 e chama o EDM para o byte
        ADDI        $t1, HDB1
```

```
        LW          $a0, $t1, $zero
        JAL         edm_byte
```

Figura B.58: trecho da APPLY_EDM que aplica o EDM no HDB1 do pacote recebido

Uma vez calculado o valor do método de detecção de erros para o cabeçalho, a função APPLY_EDM coloca tanto o endereço de destino DAB1 quanto o endereço de origem SAB1 no registrador $a0 e chama a função EDM_BYTE. Para realizar o cálculo dos *bytes* do conteúdo do pacote, a função APPLY_EDM entra no *loop* mostrado na figura B.59.

```
        AND         $s1, $zero, $t0        #Prepara para calcular os bytes de dados
        ADDI        $s1, DB8
        LUI         $s2, $FF               #FFF8h + 8 = 0000h -> Contador1: 8 unidades
        ADDI        $s2, $F8
edm_loop
        LW          $a0, $s1, $zero        #COMPUTA DBi (bytes de dados)
        JAL         edm_byte
        ADDI        $s1, $01
        ADDI        $s2, $01
        BEQ         $s2, $zero, $1
        J           edm_loop
```

Figura B.59: trecho da APPLY_EDM que mostra o *loop* de aplicação do EDM nos DBi

Dessa forma, os oito *bytes* de dados ou comando passam pelo método de detecção de erros, onde os valores são acumulados no registrador TEMP3. Então, a função APPLY_EDM transfere o resultado final do cálculo do registrador TEMP3 para o registrador $s0 e decide sobre o que deve ser feito de acordo com os estados dos *bytes* de controle RUN_TX, RUN_RX e RUN_RTX. O trecho da decisão é o mais importante da função e pode ser visto na figura B.60.

```
apply_final         #Decisão sobre o que fazer com o resultado do EDM calculado
        AND         $t1, $zero, $t0
        ADDI        $t1, RUN_RX
        LW          $t0, $t1, $zero
```

```
BLT     $zero, $t0, $1      #Se RUN_RX=0, novo pacote será enviado
J       apply_ass           #então aplica TEMP3 em CRC1

AND     $t1, $zero, $t0
ADDI    $t1, RUN_TX         #Se RUN_RX=1 e RUN_TX=0, então um
LW      $t0, $t1, $zero     # novo pacote esta sendo
BLT     $zero, $t0, $1      # recebido. Comparação dos CRCs
J       apply_confr         #recebido e calculado

AND     $t1, $zero, $t0     #Se RUN_RX=RUN_TX=1 e RUN_RTX=0,
ADDI    $t1, RUN_RTX        #então foi recebido o pacote de resposta.
LW      $t0, $t1, $zero     #Comparação entre os CRCs. Se não,
BLT     $zero, $t0, $1      #será transmitido um pacote de resposta e o
J       apply_confr         #TEMP3 deve ser aplicado em CRC1.
```

Figura B.60: trecho da APPLY_EDM que mostra o processo de decisão da função

Caso o *byte* de controle RUN_RX tenha seu valor igual a zero, significa que um pacote está sendo transmitido e o resultado do método de detecção de erro deve ser aplicado no campo CRC1 do pacote. Caso o *byte* RUN_RX tenha seu valor maior que zero, a função APPLY_EDM verifica o valor do *byte* RUN_TX. Se RUN_RX>0 e RUN_TX=0, significa que um pacote foi recebido, a função APPLY_EDM deve comparar o valor calculado com o campo CRC1 do pacote recebido para checar se são idênticos. Caso o *byte* RUN_RX>0 e RUN_TX>0, a função APPLY_EDM verifica o *byte* RUN_RTX. Se o RUN_RTX tiver seu valor igual a zero, então foi recebido um pacote de resposta e o valor calculado deve ser comparado com o campo CRC1 do pacote recebido. Se o RUN_RTX>0, então será transmitido um pacote de resposta e o valor calculado deve ser aplicado no campo CRC1 do pacote a ser transmitido.

Tendo a função APPLY_EDM decidido o que deve ser feito, o fluxo de execução é desviado para o respectivo rótulo que realiza o devido tratamento. O tratamento da tarefa de aplicar o resultado calculado no campo CRC1 do pacote é visto na figura B.61.

```
apply_ass                               #Aplica CRC calculado => CRC1 do pacote
        AND         $t1, $zero, $t0
        ADDI        $t1, CRC1
        SW          $s0, $t1, $zero     #Aplica $s0 => CRC1
        J           end_apply_edm
```

Figura B.61: trecho da APPLY_EDM que mostra o CRC calculado sendo transferido

E o tratamento da tarefa de comparar o valor calculado com o valor recebido no campo CRC1 do pacote, configurando o valor do *byte* ERR apropriadamente para indicar a ocorrência de erros na recepção é mostrado na figura B.62 a seguir.

```
apply_confr         #Confere $s0 com o CRC1 recebido e seta o bit ERR
        AND         $t1, $zero, $t0
        ADDI        $t1, ERR
        SW          $zero, $t1, $zero   #Limpa byte ERR=0

        AND         $t2, $zero, $t0
        ADDI        $t2, CRC1
        LW          $t0, $t2, $zero     #$t0=CRC do pacote
        XOR         $t2, $s0, $t0       #$s0=CRC calculado
        BEQ         $t2, $zero, $2      #Confere e seta erro de acordo
        LUI         $s3, $FF
        SW          $s3, $t1, $zero     #Houve erro: ERR>0
```

Figura B.62: trecho da APPLY_EDM que compara os CRCs e configura o *byte* ERR

Realizada uma das duas tarefas, a função APPLY_EDM termina e retorna para a função que a chamou.

B.11 - Função EDM_BYTE

Esta função recebe *byte* por *byte* do pacote de dados ou comando para realizar um cálculo de detecção de erros. Os *bytes* são passados como argumentos para a função EDM_BYTE através do registrador $a0. Os resultados calculados são acumulados no registrador TEMP3, que tem seu conteúdo igualado a zero apenas no início da função APPLY_EDM.

Esse procedimento se faz necessário porque o método de detecção de erros escolhido foi o CRC-16, que leva em consideração o valor calculado da iteração anterior para um novo cálculo.

O algoritmo do CRC-16 utilizado não foi desenvolvido por causa da necessidade da otimização que uma rede de sensores impõe no seu código, dados os escassos recursos de cada nó. Assim como foi feito no PIC, o algoritmo utilizado é de autoria de Dattalo [45] que é altamente otimizado. O algoritmo foi fielmente traduzido de sua implementação do PIC para o RISC16 e se encontra na figura B.63 a seguir.

```
edm_byte
crc16               #entrada acontece através do registrador $a0

        AND         $t2, $zero, $t0         #Separa os dois octetos do byte corrente
        ADDI        $t2, $FF
        AND         $a1, $t2, $a0           #$a1 (octeto menos significativo)
        NOT         $t2
        AND         $a2, $t2, $a0           #$a2 (octeto mais significativo)
        SHIFT       $a2, $08

        AND         $t1, $zero, $t0         #Separa os dois octetos do CRC anterior
        ADDI        $t1, TEMP3
        LW          $a0, $t1, $zero         #$a0 agora contem o resultado anterior
        AND         $s1, $t2, $a0           #$s1 octeto mais significativo do CRC
        SHIFT       $s1, $08
        NOT         $t2
        AND         $s0, $t2, $a0           #$s0 octeto menos significativo do CRC

        ADD         $gp, $zero, $t2         #Ponteiro $gp>0 para controlar octetos

        ADD         $t0, $zero, $a2         #Copia octeto mais significativo em $t0

inicia_crc16
        XOR         $s2, $s0, $t0
```

```
ADD     $s0, $zero, $s1

AND     $s3, $zero, $t0     #Limpa $s3(temp) para guardar o padrão do CRC
AND     $a0, $zero, $t0
ADDI    $a0, $01            #$a0 agora guarda a posição do bit -signif

ADD     $t0, $zero, $s2
AND     $t1, $a0, $t0       #Carry virtual
SHIFT   $t0, $01
LUI     $t2, $00
ADDI    $t2, $7F
AND     $t0, $t2, $t0

XOR     $s2, $t0, $s2

ADD     $a0, $zero, $s3     #$a0 guarda temp temporariamente
SHIFT   $s3, $01            #Rotaciona temp
BEQ     $t1, $zero, $3
AND     $t2, $zero, $t1
ADDI    $t2, $80
OR      $s3, $t2, $s3       #Seta bit 7 do temp
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $s3, $t2, $s3       #Desliga bit 7 do temp
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $a0       #Guarda em $t1 o carry do temp anterior

ADD     $t0, $zero, $s2
SHIFT   $t0, $01
BEQ     $t1, $zero, $3      #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $80
```

```
OR      $t0, $t2, $t0     #Seta bit 7
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $t0, $t2, $t0     #Desliga bit 7
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $s2     #Guarda em $t1 o carry do index anterior

ADD     $s1, $zero, $t0

ADD     $a0, $zero, $s3   #$a0 guarda temp temporariamente
SHIFT   $s3, $01          #Rotaciona temp
BEQ     $t1, $zero, $3    #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $80
OR      $s3, $t2, $s3     #Seta bit 7 do temp
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $s3, $t2, $s3     #Desliga bit 7 do temp
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $a0     #Guarda em $t1 o carry do temp anterior

ADD     $a0, $zero, $s2   #$a0 guarda index temporariamente
SHIFT   $s2, $81          #Rotaciona index para esquerda
BEQ     $t1, $zero, $3    #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $01
OR      $s2, $t2, $s2     #Seta bit 0
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $FE
```

```
AND     $s2, $t2, $s2     #Desliga bit 0
LUI     $t2, $00
ADDI    $t2, $80
AND     $t1, $t2, $a0     #Guarda em $t1 o carry do index anterior

XOR     $s2, $t0, $s2

ADD     $t0, $zero, $s2   #Trocar nibbles do octeto menos significativo
LUI     $t2, $00
ADDI    $t2, $FF
AND     $t0, $t2, $t0     #Limpa byte mais significativo
SHIFT   $t0, $04          #Nibble mais significativo trocado
LUI     $t2, $F0
ADDI    $t2, $00
AND     $a0, $t2, $t0
SHIFT   $a0, $08          #Nibble menos significativo trocado
OR      $t0, $a0, $t0     #Nibbles trocados no octeto menos signif.

XOR     $s2, $t0, $s2

ADD     $t0, $zero, $s3

LUI     $t2, $00
ADDI    $t2, $02
AND     $a0, $t2, $s2
BEQ     $a0, $zero, $3    #Verifica bit 1 do index
LUI     $t2, $00
ADDI    $t2, $01
XOR     $t0, $t2, $t0     #Se o bit e' igual a 1, faca esta linha
XOR     $s0, $t0, $s0     #Se o bit e' igual a 0, faca esta linha

LUI     $t0, $00
ADDI    $t0, $C0
LUI     $t2, $00
```

```
ADDI    $t2, $02
AND     $a0, $t2, $s2
BEQ     $a0, $zero, $1      #Verifica bit 1 do index
XOR     $s1, $t0, $s1       #Se o bit e' igual a 1, faca esta linha

BEQ     $gp, $zero, $3      #Processa no CRC o octeto menos SIGNIFIC.
AND     $gp, $zero, $t0     #Zera $gp para indicar 1o octeto calculado
ADD     $t0, $zero, $a1     #Copia octeto menos significativo em $t0
J       inicia_crc16

LUI     $t2, $00            #AGRUPA OS OCTETOS PARA SALVAR
ADDI    $t2, $FF
AND     $s0, $t2, $s0       #Limpa octeto mais significativo de $s0
AND     $s1, $t2, $s1       #Limpa octeto mais significativo de $s1
SHIFT   $s1, $88            #Rotaciona $s1 8 bits para esquerda
ADD     $a0, $s1, $s0       #Concatena ambos os octetos (16 bits)

AND     $t2, $zero, $t0     #Salva resultado do byte processado
ADDI    $t2, TEMP3          #Zera registrador temporario TEMP3
SW      $a0, $t2, $zero

J       $ra, $zero          #FIM DA EDM_BYTE
```

Figura B.63: função EDM_BYTE que calcula o método de detecção de erro para cada *byte*

As únicas modificações que foram feitas na função EDM_BYTE foram as separações e concatenações necessárias para que as palavras de 16 *bits* pudessem ser processadas em octetos, pois o algoritmo de Dattalo [45] foi desenvolvido para o PIC.

B.12 - Função TEMPO

A função TEMPO desempenha um dos mais importantes papéis do protocolo de comunicação. Sem ela, os *bits* do pacote não poderiam ser lidos corretamente, e se fossem transmitidos, os *bits* não poderiam ser reconhecidos. A função TEMPO tem como tarefa a

geração dos atrasos entre as leituras do *bits* nas interfaces de entrada, ou entre a variação dos *bits* nas escritas dos *bits* nas interfaces de saída.

Como pode ser vista na figura B.64, a função TEMPO é bem simples e com poucas instruções.

```
tempo           #REGISTRADOR $a0 TEM O VALOR DA ESPERA
        AND     $t1, $zero, $t0
        ADDI    $t1, $01

loop_bit
        SUB     $a0, $t1, $a0       #Decrementa 1 unid do valor lido (4ciclos)
        BEQ     $a0, $zero, $1      #(3ciclos)
        J       loop_bit            #(3ciclos)

        J       $ra, $zero          #RETURN - fim da função TEMPO
```

Figura B.64: função TEMPO completa que mostra o *loop* de espera com 10 ciclos

Repare que as primeiras duas instruções servem apenas para gravar o valor 1 para que a instrução SUB possa realizar um decremento unitário do valor recebido como parâmetro de espera. Então, o loop de espera tem exatamente 10 ciclos de processamento, de acordo com os dados das especificações da dissertação do Benício (2002), pois a função SUB consome 4 ciclos do microcontrolador e as instruções BEQ e J consomem 3 ciclos cada. Em outras palavras, a cada 10 ciclos do microcontrolador é feito um decremento unitário do valor do contador. Como o relógio do microcontrolador funciona a 250 MHz, ele consegue fazer 25 milhões de decrementos unitários por segundo. Se for considerado que a taxa de *bits* fixada para a comunicação é de 9600 bps, o tempo de cada *bit* é 104,166µs. A regra de três a seguir foi que definiu os valores para o tempo de um *bit* e o tempo de meio *bit* que constam no arquivo SNAP16.INC.

Para 250 MHz:

$$\begin{array}{ccc} 25 * 10^{6} \text{ dec} & \text{-----------------} & 1 \text{ s} \\ X & \text{-----------------} & 104{,}166 * 10^{-6} \text{ s} \end{array} \quad \text{(B.1)}$$

$$X = 2604 \text{ decrementos}$$

Como o microcontrolador também poderá funcionar a 200 MHz, os mesmos cálculos são feitos a seguir.

Para 200 MHz:

$$\begin{array}{ccc} 20 * 10^{6} \text{ dec} & \text{-----------------} & 1 \text{ s} \\ X & \text{-----------------} & 104{,}166 * 10^{-6} \text{ s} \end{array} \quad (B.2)$$

$$X = 2083 \text{ decrementos}$$

Dessa forma, foram calculados os valores das constantes TEMPO_BIT e TEMPO_MEIO_BIT do arquivo SNAP16.INC. Caso a velocidade do relógio do microcontrolador seja alterada, novos valores tem que ser configurados. Lembrando que os valores calculados são da base decimal, e os valores que devem ser configurados devem ser hexadecimais. No caso, para 250 MHz o tempo de um *bit* fica 0A2Ch e para 200 MHz o tempo de um *bit* fica 0823h, que são os equivalentes em hexadecimal dos valores 2604 e 2083, respectivamente.

A figura B.65 a seguir mostra como os valores são configurados no arquivo SNAP16.INC.

```
#///////////////////////////////////////////////////////////////////////////
#CONFIGURA A VELOCIDADE DA TRANSMISSAO DE ACORDO COM O CLOCK:
#       250 MHz:   9600bps --> Tbit=104us --> TEMPO_BIT= 2604D ou 0A2Ch
#       200 MHz:   9600bps --> Tbit=104us --> TEMPO_BIT= 2083D ou 0823h
#///////////////////////////////////////////////////////////////////////////
TEMPO_BITup         EQU   $08           #Configurado para 200MHz
TEMPO_BITlo         EQU   $23
TEMPO_MEIO_BITup          EQU   $04     #Metade do valor de um bit
TEMPO_MEIO_BITlo          EQU   $11
```

Figura B.65: trecho do arquivo SNAP16.INC que é configurado em função do *clock*

Note que os valores são separados em octetos porque não existem instruções no RISC16 que carreguem 16 *bits* de uma única vez.

Depois de terminado o *loop* de espera quando o contador tem seu valor decrementado para zero, a função TEMPO retorna para a função que a chamou.

C – CÓDIGO COMPLETO PARA O PIC

C.1 - LINGUAGEM ASSEMBLY DO PROTOCOLO

```
;-----------------------------------------------------------------------------------------------------
;           **********   S.N.A.P. Modificado para o PIC   **************
;     ******************** GUSTAVO LUCHINE ***********************
;        *Realizado a partir do trabalho de CASTRICINI & MARINANGELI*
;-----------------------------------------------------------------------------------------------------

        PROCESSOR       16F84A          ;PROCESSADOR UTILIZADO
        RADIX           DEC
        INCLUDE         "P16F84A.inc"           ;ENDERECOS DO PROCESSADOR
        INCLUDE         "SNAP.INC"  ;MAPA DE MEMORIA DO PROTOCOLO
        INCLUDE         "APPL.INC"  ;DEFINICOES SOBRE END. DOS NODOS
        ERRORLEVEL      -302

        __CONFIG    _CP_OFF & _PWRTE_OFF & _WDT_ON & _XT_OSC

        ORG   0000H
        GOTO start

;////////////////////////////////////////////////////////////////////////
;  TRATADOR DE INTERRUPCAO
;////////////////////////////////////////////////////////////////////////
        ORG   0004H             ;End. definido para o tratador de interrupção

Push
        BTFSS       STATUS,RP0
        GOTO        RP0CLEAR
        BCF         STATUS,RP0
        MOVWF       W_TEMP
        SWAPF       STATUS,W
```

```
        MOVWF   STATUS_TEMP
        BSF     STATUS_TEMP,1
        GOTO    ISR_CODE
RP0CLEAR
        MOVWF   W_TEMP
        SWAPF   STATUS,W
        MOVWF   STATUS_TEMP

ISR_CODE                        ;subprograma de interrupção
        CALL    rx_snap

POP
        SWAPF   STATUS_TEMP,W
        MOVWF   STATUS
        BTFSS   STATUS,RP0
        GOTO    RETURN_WREG
        BCF     STATUS,RP0
        SWAPF   W_TEMP,F
        SWAPF   W_TEMP,W
        BSF     STATUS,RP0
        BCF     INTCON,INTF
        RETFIE
RETURN_WREG
        SWAPF   W_TEMP,F
        SWAPF   W_TEMP,W
        BCF     INTCON,INTF
        RETFIE                  ;Fim da interrupção
```

```
;//////////////////////////////////////////////////////////////////////
start
;//////////////////////////////////////////////////////////////////////
;  Configurações do Protocolo
;//////////////////////////////////////////////////////////////////////

        BSF         STATUS,RP0 ;seleciona banco de memória 1

        CLRF        TRISB       ;Configura todos os pinos da porta B como saídas
        BSF         RXD         ;Pino de entrada do protocolo
        BSF         GNA         ;entrada do gerador de números

        ; Inicio parte configura disparo interrupção
        ;Bordo descendente dispara Int externa
        MOVLW       00101111B   ;Prescaler para o Watchdog = 1/16
        MOVWF       OPTION_REG          ;TMR parado OPTION_REG,5=1
        ; Fim parte configura disparo interrupção

        BCF         STATUS,RP0          ;seleciona banco de memória 0

        BSF         INTCON,INTE         ;habilita interrupção externa

        BSF         NCD                 ;Estado default=1
        BSF         TXD                 ;Estado default=1

        CLRF        CONTROL             ;Limpa registrador de controle

        MOVLW       DAB1                ;DAB1 E' O MAIOR ENDERECO DE DAB
        MOVWF       DAB_MAX
        MOVLW       SAB1                ;SAB1 E' O MAIOR ENDERECO DE SAB
        MOVWF       SAB_MAX
        MOVLW       1                   ;1 BYTE DE SAB E DAB
        MOVWF       NSAB
        MOVWF       NDAB
```

```
        MOVLW       DB8               ;DB1 E' O MAIOR ENDERECO DE DB
        MOVWF       DB_MAX
        MOVLW       8                 ;NUMERO DE BYTES DE DB
        MOVWF       NDB

        MOVLW       CRC1              ;CRC1 E' O MAIOR ENDERECO DE CRC
        MOVWF       EB_MAX
        MOVLW       1                 ;NUMERO DE BYTES DE CRC
        MOVWF       NEB

        BSF         INTCON,GIE        ;habilitação global de interrupções

;////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
;       ************************************************************
;       ******************* PROGRAMA PRINCIPAL ********************
;       ************************************************************
;////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
begin
        CLRWDT

        NOP         ;Aqui entra o aplicativo desenvolvido que usa o
        NOP         ; protocolo

        GOTO        begin       ;Gera loop infinito

;//////////////////////////////////////////////////////////////////////
;  Função TEMPO
;//////////////////////////////////////////////////////////////////////
tempo           ;Função que espera o "tempo" colocado no registrador W
                ;(pode ser o tempo de um bit ou de um bit e meio)

        MOVWF       TEMP+4
```

```
loop_bit
        CLRWDT
        DECFSZ      TEMP+4,F
        GOTO        loop_bit

        RETURN

;//////////////////////////////////////////////////////////////////////
;  Função TX_SNAP
;//////////////////////////////////////////////////////////////////////
tx_snap
        BCF         INTCON,GIE          ;DEsabilitacao global de interrupções
        CLRWDT

        BCF         STATUS,RP0          ;Selecionar banco 0

        BSF     RUN_TX      ;RUN_TX=1 para indicar que tx_snap esta executando

        BTFSC       RUN_RX      ;Se RUN_RX=1 entao RUN_RTX=1 (RX->TX)
        BSF         RUN_RTX

        MOVLW       HEADER2
        MOVWF       HDB2        ;Carrega HDB2 para transmitir um pacote

        MOVLW       HEADER1
        MOVWF       HDB1        ;Carrega HDB1 para transmitir um pacote

        BTFSS       RUN_RX      ;Se RUN_RX=1 então rx_snap estava executando
        GOTO        carrega_ack ;Se RUN_RX=0 entao verifica bit de acknowledge
        BSF         HDB2,1      ;Liga o bit de resposta do pkt pq RX esta rodando

        BTFSS       ERR         ;ERRO GERADO NO EDM (PARTE RECEPCAO)
        BCF         HDB2,0      ;zerar o bit se estiver certo, pois já vem com 1
                                ; do HEADER2 por default
```

```
        BTFSC       SAB_PARA_DAB   ;Verifica se ja' houve SAB=>DAB (1o pkt resp)
        GOTO        charge_sab
        MOVFW       SAB1           ;Se RX esta rodando então deve-se responder
        MOVWF       DAB1           ;ao no' que iniciou a troca de pacotes
        BSF         SAB_PARA_DAB
        GOTO        charge_sab
```

;DAB1 DEVE SER CARREGADO NO APLICATIVO PRINCIPAL

```
carrega_ack                        ;APENAS PARA O TX=1 E RX=0
        BTFSS       ACK_PKT_TX
        BCF         HDB2,0         ;COLOCA ZERO SE ACK=0

charge_sab
        MOVLW       MY_ADDR1
        MOVWF       SAB1           ;Se vou transmitir, colocar o meu end. no SAB

        CALL        apply_edm      ;calcula o EDM para o pacote a ser enviado

        CALL        send_snap      ;TRANSMITE O PACOTE

        BTFSS       RUN_RX         ;Se rx_snap não esta sendo executado então
        BTFSS       HDB2,LSB       ;verifico se ACK=01
        GOTO        end_txsnap     ;Se RX estiver rodando vai para o fim da função

        BSF   STATUS,RP0           ;Banco 1
        BCF   OPTION_REG,PSA       ;Seleciona o pre-scaler para o TMR0 1/256
        BCF   OPTION_REG,5         ;HABILITA TMR
        BCF   STATUS,RP0           ;Banco 0

wait_response
        MOVLW       125            ;ESPERO 8 SEGUNDOS (125 vezes 250 vezes 256)
        MOVWF       TEMP+5
```

```
        BSF         REJ             ;Inicia o "REJ" com resposta nao recebida
        BSF    INTCON,GIE           ;Habilitação global de interrupções (RX não está
                                    ; rodando)

wait_loop1
        MOVLW       6               ;faz a contagem do TMR0 comecar de 6
        MOVWF       TMR0            ;assim teremos 250 contagens no byte

wait_loop2
        CLRWDT
        BTFSS       REJ
        GOTO        exit_wait       ;SE REJ=0, ENTAO UM PACOTE FOI RECEBIDO

        MOVF        TMR0,W
        BTFSS       STATUS,Z        ;QUANDO TMR=0, SAI DO LOOP2
        GOTO        wait_loop2

        DECFSZ      TEMP+5,F        ;SAI DO LOOP1 DEPOIS DE 125 VEZES
        GOTO        wait_loop1

exit_wait
        BCF    INTCON,GIE           ;DEsabilitacao global de interrupções
        BSF    STATUS,RP0           ;Banco 1
        BSF    OPTION_REG,PSA       ;Seleciona pre-scaler para o watchdog 1/128
        BSF    OPTION_REG,5         ;DESABILITA TMR
        BCF    STATUS,RP0           ;Banco 0

        BTFSC       REJ             ;SE HOUVE RESPOSTA PULA
        GOTO        nao_recebido    ;SE NAO HOUVE RESPOSTA --> time out
        BTFSS       ERR             ;SE HOUVE ERRO, VAI PARA nao_recebido
        BTFSC       HDB2,LSB        ;SE REJ=0, ENTAO ACK=10 OU 11 -> TESTAR SE
        GOTO        nao_recebido    ; ACK RESPONSE
        BSF    RECEBIDO             ;REJ=0, ERR=0, ACK=10, ENTAO PACOTE FOI
        GOTO        end_txsnap      ; RECEBIDO PELO NO' DE DESTINO
```

```
nao_recebido
        BCF    RECEBIDO            ;pacote não foi recebido adequadamente

end_txsnap
        BCF         RUN_RTX
        BCF         RUN_TX
        BTFSS       RUN_RX      ;Habilita se RX nao estiver rodando
        BSF         INTCON,GIE ;Habilitacao global de interrupcoes

        RETURN              ;FIM DA TRANSMISSAO

;///////////////////////////////////////////////////////////////////////
;       Função SEND_SNAP
;///////////////////////////////////////////////////////////////////////
send_snap
        MOVLW       SYNC
        CALL        send_byte   ;Envia byte SYNC

        MOVLW       HDB2        ;Envia os bytes HDB2 e HDB1
        MOVWF       FSR
        BCF         STATUS,Z
        MOVLW       2           ;NUMERO DE BYTES PARA ENVIAR
        CALL        send_word

        MOVF        DAB_MAX,W
        MOVWF       FSR         ;Apontar ponteiro FSR para DAB_MAX
        MOVF        NDAB,W      ;e colocar NDAB no registrador W
        CALL        send_word

        MOVF        SAB_MAX,W       ;Apontar ponteiro FSR para SAB_MAX
        MOVWF       FSR             ;e colocar NSAB no registrador W
        MOVF        NSAB,W
        CALL        send_word
```

```
        MOVF        DB_MAX,W         ;Apontar ponteiro FSR para DB_MAX
        MOVWF       FSR              ;e colocar NDB no registrador W
        MOVF        NDB,W
        CALL        send_word

send_eb
        MOVF        EB_MAX,W         ;Apontar ponteiro FSR para EB_MAX
        MOVWF       FSR              ;e colocar NDB no registrador W
        MOVF        NEB,W
        CALL        send_word

        MOVF        DAB1,W      ;guarda DAB1 do pacote que foi transmitido
        MOVWF       TEMP+7      ;para comparacao posterior, caso ACK=01

        RETURN          ;Fim da send_snap


;///////////////////////////////////////////////////////////////////////
;  Funcao SEND_WORD
;///////////////////////////////////////////////////////////////////////
send_word
        BTFSC       STATUS,Z
        GOTO        end_send_word
        MOVWF       TEMP        ;Le no W o numero de bytes a serem enviados

sendw_loop
        MOVF        INDF,W      ;INDF e' o conteudo do ponteiro FSR
        CALL        send_byte   ;Transmite o byte no registrador W

        INCF        FSR,F       ;Incrementa o ponteiro e decrementa o contador
        DECFSZ      TEMP,F
        GOTO        sendw_loop
```

```
end_send_word
        RETURN                  ;Fim da send_word

;/////////////////////////////////////////////////////////////////////
;  Funcao SEND_BYTE
;/////////////////////////////////////////////////////////////////////
send_byte
        CLRWDT

        MOVWF       TEMP+1      ;Carrega TEMP+1 com o byte a ser enviado
        MOVLW       8
        MOVWF       TEMP+2      ;Carrega o n° de bits para ser enviado

        BCF         TXD         ;Transmite o start bit
        MOVLW       TEMPO_BIT   ;Espera o tempo de um bit
        CALL        tempo

sendb_loop
        BCF         STATUS,C    ;Zera o carry
        RRF         TEMP+1,F    ;Rotaciona os bits do byte a ser enviado

        BTFSC       STATUS,C
        BSF         TXD         ;Liga o pino de saída
        BTFSS       STATUS,C
        BCF         TXD         ;Desliga o pino de saída

        MOVLW       TEMPO_BIT   ;Espera o tempo de um bit
        CALL        tempo

        DECFSZ      TEMP+2,F    ;Decrementa o contador de bits
        GOTO        sendb_loop

        BSF         TXD         ;Transmite o stop bit
```

```
        MOVLW       TEMPO_BIT  ;Espera o tempo de um bit
        CALL        tempo

        RETURN                 ;Return at routine send_word

;//////////////////////////////////////////////////////////////////////
;  Funcao RX_SNAP
;//////////////////////////////////////////////////////////////////////
rx_snap
        CLRWDT

        BCF         STATUS,RP0

        BSF         RUN_RX       ;Indica que a função rx_snap esta sendo executada

        CALL        receive_snap ;Efetivamente recebe o pacote do protocolo

        BTFSC       REJ          ;Se o pacote tiver sido rejeitado, então termina
        GOTO        end_rxsnap

        CLRWDT

        CALL        apply_edm    ;Calcula o método de detecção de erros

        BTFSC       ERR          ;SE DEU ERRO (ERR=1), SAIR DO RX
        GOTO        end_rxsnap

        BTFSC       RUN_TX       ;[ SE TX ESTA RODANDO E ESTAMOS NO RX,
        GOTO        end_rxsnap   ;] entao estou recebendo o pacote de resposta

runtx_0                          ;TX NAO ESTA RODANDO
        BCF         SAB_PARA_DAB

        CALL        comando_exec ;SE FOR UM COMANDO, EXECUTA
```

```
        BTFSS       HDB2,LSB    ;Ou estou recebendo novo pacote que exige resposta
        GOTO        end_rxsnap  ;Ou estou recebendo novo pkt e termino a recepcao

espera_broadcast
        CLRW                ;VERIFICA SE ENDERECO DAB E' DE BROADCAST
        XORWF       DAB1,W
        BTFSS       STATUS,Z
        GOTO        continua_resposta   ;não sendo broadcast, continua normalmente

        MOVLW       MY_ADDR1    ;como e' de broadcast, entao pega endereco do no'
        MOVWF       TEMP+1      ; E COLOCA NO

        BSF     STATUS,RP0          ; BANCO 1
        BCF     OPTION_REG,PSA      ;Seleciona o pre-scaler para o TMR0 1/256
        BCF     OPTION_REG,5        ;HABILITA TMR
        BCF     STATUS,RP0          ; BANCO 0

espera_broad1
        MOVLW       6           ;faz a contagem do TMR0 começar de 6
        MOVWF       TMR0        ;assim teremos 250 contagens no byte

espera_broad2
        CLRWDT
        MOVF        TMR0,W
        BTFSS       STATUS,Z
        GOTO        espera_broad2

        DECFSZ      TEMP+1,F            ;numero_endereco vezes 250 * 32 contagens
        GOTO        espera_broad1

        BSF     STATUS,RP0          ;BANCO 1
        BSF     OPTION_REG,PSA      ;Seleciona pre-scaler para o watchdog 1/128
        BSF     OPTION_REG,5        ;DESABILITA TMR
```

```
	BCF		STATUS,RP0		;BANCO 0

continua_resposta
	call		tx_snap		; responde caso TX não esteja rodando

end_rxsnap
	BCF		RUN_RX

	RETURN		;RETFIE e INTF ESTAO NO TRATADOR DE INTERRUPCAO

;//////////////////////////////////////////////////////////////////////
;	Função RECEIVE_SNAP
;//////////////////////////////////////////////////////////////////////
receive_snap
	CALL		receive_byte	;Recebe o byte de sincronismo (SYNC)

	MOVLW		HDB2			;Recebe os bytes HDB2 e HDB1 do cabeçalho
	MOVWF		FSR
	BCF		STATUS,Z		;Limpa do lixo que pode estar armazenado
	MOVLW		2			;Numero de bytes a serem recebidos
	CALL		receive_word

	BTFSS		RUN_TX		;[ SE TX ESTA RODANDO,
	GOTO		run_tx0
	MOVLW		HEADER2		;] Estou recebendo um pacote de resposta
	MOVWF		TEMP
	BSF		TEMP,1
	BCF		TEMP,LSB		; ACK[TEMP]=10
	MOVF		TEMP,W
	XORWF		HDB2,W		; HDB2=HEADER2[ACK=10]?
	MOVF		STATUS,W		; O XOR FICA DENTRO DO STATUS
	MOVWF		TEMP+1
	BSF		TEMP,LSB		; ACK[TEMP]=11
	MOVF		TEMP,W
```

```
        XORWF   HDB2,W      ; HDB2=HEADER2[ACK=11]?
        MOVF    STATUS,W
        IORWF   TEMP+1,F    ;JUNTA AS DUAS COMPARACOES FEITAS
        BTFSS   TEMP+1,2    ;Testa por STATUS,Z das duas comparações
        GOTO    reject      ;Rejeita se ACK recebido for diferente de 10 ou 11
        GOTO    receive_dab

run_tx0         ;Se TX não está rodando, entao estou recebendo um novo pacote

; Compara se o pacote recebido tem HEADERS validos (IGUAIS ENVIADOS)
; com a exceção de receber comandos (HDB1,7) e pacotes sem ACK (HDB2,0)
        MOVFW   HDB2
        MOVWF   TEMP        ;HDB2=>TEMP
        BSF     TEMP,LSB    ;SETA O BIT DE ACK
        MOVLW   HEADER2     ;Joga HEADER2 no W para comparação com TEMP
        XORWF   TEMP,W      ;para receber também um pacote que não exija
        BTFSS   STATUS,Z    ;ACKNOWLEDGE (ACK=00)
        GOTO    reject

        MOVFW   HDB1
        MOVWF   TEMP        ;HDB1=>TEMP
        BCF     TEMP,MSB    ;ZERA O BIT DO COMANDO NO TEMP
        MOVLW   HEADER1     ;Joga HEADER1 no W para comparação com TEMP
        XORWF   TEMP,W      ; para receber também um pacote que seja
        BTFSS   STATUS,Z    ;um comando
        GOTO    reject

receive_dab
        MOVF    DAB_MAX,W      ;Recebe o(s) byte(s) de endereço de destino DABi
        MOVWF   FSR
        MOVF    NDAB,W
        CALL    receive_word

        CLRW
```

```
        XORWF    DAB1,W      ;VERIFICA SE E' BROADCAST
        BTFSC    STATUS,Z    ;Caso positivo continua a partir do receive_sab
        GOTO     receive_sab ;Caso negativo testa se e' o endereço deste no'

        MOVLW    MY_ADDR1    ;Se MY_ADDR1 e' diferente de DABi entao rejeita
        XORWF    DAB1,W      ;o pacote e termina a funcao
        BTFSS    STATUS,Z
        GOTO     reject

receive_sab
        MOVF     SAB_MAX,W   ;Recebe o(s) byte(s) de endereco de origem SABi
        MOVWF    FSR
        MOVF     NSAB,W
        CALL     receive_word

        MOVLW    MY_ADDR1    ;Se SABi for igual ao MY_ADDRi entao
        XORWF    SAB1,W      ;existem dois nodos com o mesmo endereco e o
        BTFSC    STATUS,Z    ;presente pacote e' rejeitado
        GOTO     reject

also_sab
        BTFSS    RUN_TX      ;Se o bit RUN_TX for igual a 1, entao
        GOTO     receive_db  ;o protocolo verifica se a resposta veio do nodo
        MOVF     NDAB,W      ;para o qual o pacote foi enviado
        BTFSC    STATUS,Z
        GOTO     receive_eb

        MOVF     TEMP+7,W    ;Se TEMP+7 (DAB1 do pacote anterior) for diferente
        XORWF    SAB1,W      ;de SAB1, entao rejeita o pacote recebido
        BTFSS    STATUS,Z
        GOTO     reject

receive_db
        MOVF     DB_MAX,W    ;Recebe todos os bytes de dados do pacote
```

```
        MOVWF       FSR
        MOVFNDB,W
        CALL receive_word

receive_eb
        MOVF        EB_MAX,W   ;Recebe o byte de CRC-8 do pacote
        MOVWF       FSR
        MOVF        NEB,W
        CALL        receive_word
        BCF         REJ         ;recebido pacote a ser verificado = REJ=0
        GOTO        end_receivesnap

reject
        BSF         REJ         ;REJ=1 para indicar que o pacote foi REJEITADO

end_receivesnap
        RETURN                          ;Fim da receive_snap



;////////////////////////////////////////////////////////////////////////
;       Funcao RECEIVE_WORD
;////////////////////////////////////////////////////////////////////////
receive_word
        BTFSC       STATUS,Z     ;Verifica se a palavra a ser recebida tem ZERO byte
        GOTO        end_receiveword
        MOVWF       TEMP

receivew_loop
        CALL        receive_byte  ;Lê o byte na interface de entrada => W

        MOVWF       INDF        ;COLOCA O W NO CONTEUDO DO PONTEIRO
        INCF        FSR,F       ;INCREMENTA ENDERECO DO PONTEIRO
        DECFSZ      TEMP,F      ;Decrementa o contador de bytes a receber
        GOTO        receivew_loop
```

```
end_receiveword
        RETURN                  ;Fim da receive_word

;//////////////////////////////////////////////////////////////////////
;       Função RECEIVE_BYTE
;//////////////////////////////////////////////////////////////////////
receive_byte
        CLRWDT

        CLRF        TEMP+2      ;Recebe o byte da interface no TEMP+2
        MOVLW       8
        MOVWF       TEMP+1      ;Quantidade de bits a serem recebidos

;INICIO DA ESPERA PELO START BIT
        BSF     STATUS,RP0          ; BANCO 1
        BCF     OPTION_REG,PSA      ;Seleciona o pre-scaler para o TMR0
        BCF     OPTION_REG,5        ;HABILITA TMR 1/256
        BCF     STATUS,RP0          ;BANCO 0

        MOVLW       6           ;faz a contagem do TMR0 comecar de 6
        MOVWF       TMR0        ;assim teremos 250 contagens no byte

espera_byte
        CLRWDT
        BTFSS       RXD         ;Espera o start bit / bom para não perder o sincronismo
        GOTO        continua_byte       ;E se o start bit não vier, continua após tempo

        MOVF        TMR0,W      ;Tempo do timer = 256*250 = 64000 => 0,064s
        BTFSS       STATUS,Z
        GOTO        espera_byte

continua_byte
        BSF     STATUS,RP0              ;BANCO 1
```

```
        BSF     OPTION_REG,PSA          ;Seleciona pre-scaler para o watchdog 1/128
        BSF     OPTION_REG,5      ;DESABILITA TMR
        BCF     STATUS,RP0        ;BANCO 0

;FIM DA ESPERA DO START BIT (OU START BIT VEIO, OU HOUVE TIMEOUT)
        MOVLW       TEMPO_BIT               ;Espera de tempo de um bit
        ADDLW       TEMPO_MEIO_BIT          ;Espera de tempo de meio bit
        CALL        tempo

receive_loop

        BCF     STATUS,C                ;Coloca zero no proximo bit
        RRF     TEMP+2,F

        BTFSC       RXD             ;Se a entrada for zero, nao faz nada
        BSF         TEMP+2,MSB      ;Se a entrada for um, seta o bit

        MOVLW       TEMPO_BIT       ;Espera de tempo de um bit
        CALL        tempo

        DECFSZ      TEMP+1,F      ;Decrementa o número de bits a serem recebidos
        GOTO        receive_loop
        MOVF        TEMP+2,W        ;COLOCA O BYTE RECEBIDO NO W

        RETURN          ;Fim da receive_byte

;//////////////////////////////////////////////////////////////////////
;       Função COMANDO_EXEC
;       Esta função verifica se o pacote recebido tem um comando a ser
;       realizado (HDB1,7) e o executa.
;//////////////////////////////////////////////////////////////////////
comando_exec                    ;TX NAO ESTA RODANDO
;VERIFICAR SE PACOTE RECEBIDO É COMANDO, SE FOR COLOCAR O # NO W
        BTFSS       HDB1,MSB
```

```
        GOTO        fim_comando  ;Nao é comando, ir para fim_comando para continuar

;-----------------------------------------------
;INTERPRETAR O COMANDO E TOMAR ACAO
comando2
        MOVFW       DB1                 ;COLOCA NO W O CONTEUDO DO DB1
        XORLW       2
        BTFSS       STATUS,Z
        GOTO        comando4
        BSF         NCD                 ;COMANDO 2: LIGA SAIDA NCD
        CALL        limpa_dados_pacote
        MOVLW       3
        MOVWF       DB1                 ;Retorna CODIGO 3 (EXECUTADO CMD2)
        GOTO        fim_comando

comando4
        MOVFW       DB1                 ;COLOCA NO W O CONTEUDO DO DB1
        XORLW       4
        BTFSS       STATUS,Z
        GOTO        nenhum_comando      ;Vai p/ fim, se ultimo comando não realizado
        BCF         NCD                 ;COMANDO 4: DESLIGA SAIDA NCD
        CALL        limpa_dados_pacote
        MOVLW       5
        MOVWF       DB1                 ;Retorna CODIGO 5 (EXECUTADO CMD4)
        GOTO        fim_comando

;INCLUIR MAIS COMANDOS AQUI, conforme a necessidade da aplicação
;LEMBRANDO QUE CADA COMANDO PULA PARA O SEGUINTE E O ULTIMO
;CONTEM "GOTO fim_comando" NA ULTIMA LINHA
;-----------------------------------------------
nenhum_comando
        CALL        limpa_dados_pacote  ;COMANDO NAO REALIZADO
        MOVLW       55
        MOVWF       DB1                 ;RETORNA CODIGO 55
```

```
fim_comando
        BTFSS       HDB1,MSB
        CALL        limpa_dados_pacote   ;NAO SENDO COMANDO, LIMPA DADOS

        RETURN

;**********************************************************************
;  Rotina limpa_dados_pacote:
;       Limpa todos os dados do pacote: DBi=0 (1<i<DB_MAX)
;**********************************************************************
limpa_dados_pacote
        MOVF        DB_MAX,W  ;LIMPA OS DADOS DO PACOTE
        MOVWF       FSR
        MOVFW       NDB
        MOVWF       TEMP+5
limpa_byte
        CLRF        INDF
        INCF        FSR
        DECFSZ      TEMP+5
        GOTO        limpa_byte     ;fim da limpeza dos dados

        RETURN

;///////////////////////////////////////////////////////////////////////
;       Funcao APPLY_EDM
;///////////////////////////////////////////////////////////////////////
apply_edm
        CLRWDT
        CLRF        TEMP+3        ;Limpa o TEMP+3 que vai acumular o EDM

start_edm
        MOVF        HDB2,W        ;Carrega HDB2 e chama o EDM para o byte
        CALL        edm_byte
```

```
        MOVF    HDB1,W        ;Carrega HDB1 e chama o EDM para o byte
        CALL    edm_byte

        MOVF    DAB1,W        ;Carrega DAB1 e chama o EDM para o byte
        CALL    edm_byte

        MOVF    SAB1,W        ;Carrega SAB1 e chama o EDM para o byte
        CALL    edm_byte

        MOVF    NDB,W         ;Carrega os dados e chama o EDM byte a byte
        BTFSC   STATUS,Z
        GOTO    apply_final   ;SE NAO HOUVER BYTES DE DADOS, PULA
        MOVWF   TEMP
        MOVF    DB_MAX,W
        MOVWF   FSR

edm_loop3
        MOVF    INDF,W        ;COMPUTA DBi
        CALL    edm_byte
        INCF    FSR,F
        DECFSZ  TEMP,F
        GOTO    edm_loop3

apply_final     ;Decisão sobre o que fazer com o resultado do EDM calculado
        BTFSS   RUN_RX        ;Se RUN_RX=0 um novo pacote esta sendo enviado
        GOTO    apply_ass     ;então aplica TEMPi em CRCi
        BTFSS   RUN_TX        ;Se RUN_RX=1 e RUN_TX=0, então um novo pacote
        GOTO    apply_confr   ;está sendo recebido. Comparação dos CRCs recebido
                              ;e calculado
        BTFSS   RUN_RTX       ;Se RUN_RX=RUN_TX=1 e RUN_RTX=0, então foi
        GOTO    apply_confr   ;recebido o pacote de resposta. Deve haver a
                              ;comparação entre os CRCs. Se RUN_RTX=1 então
                              ;será transmitido um pacote de resposta e o
```

```
                                    ;TEMPi deve ser aplicado em CRCi.

apply_ass                                   ;Aplica TEMP+3 => CRC1
        MOVF        TEMP+3,W
        MOVWF       CRC1                    ;COLOCA RESULTADO EDM NO CRC1
        GOTO        end_apply_edm

apply_confr         ;Confere TEMP+3 com o CRC1 recebido e seta o bit ERR
        BCF         ERR             ;Inicia o bit ERR para sem erro

        MOVF        TEMP+3,W        ;Compara TEMP+3 com o CRC1
        XORWF       CRC1,W
        BTFSS       STATUS,Z
        BSF         ERR             ;Houve erro detectado, TEMP+3 diferente de CRC1

end_apply_edm
        RETURN                              ;Fim da apply_edm

;////////////////////////////////////////////////////////////////////////
;       Funcao EDM_BYTE
;////////////////////////////////////////////////////////////////////////
edm_byte
        ;GOTO       crc8    ;Configura EDM: GOTO crc8 / GOTO checksum
                            ;Lembrar de alterar HDB1 se alterar a detecção de erros

;-----------------------------------------------------------------------
;  ALGORITMO CRC-8: http://www.dattalo.com/technical/software/pic/crc_8.asm
;  Desenvolvido por T. Scott Dattalo
;-----------------------------------------------------------------------
crc8
        xorwf       TEMP+3,f
        clrw

        btfsc       TEMP+3,0
```

```
        xorlw       0x5e

        btfsc       TEMP+3,1
        xorlw       0xbc

        btfsc       TEMP+3,2
        xorlw       0x61

        btfsc       TEMP+3,3
        xorlw       0xc2

        btfsc       TEMP+3,4
        xorlw       0x9d

        btfsc       TEMP+3,5
        xorlw       0x23

        btfsc       TEMP+3,6
        xorlw       0x46

        btfsc       TEMP+3,7
        xorlw       0x8c

        movwf       TEMP+3

        RETURN

;----------------------------------------------------------------------
;  ALGORITMO CHECKSUM (para configurar caso seja desejado)
;----------------------------------------------------------------------
checksum
        ;LEMBRAR DE ZERAR TEMP+3 ANTES DE CHAMAR ESTE METODO
        ADDWF       TEMP+3,F    ;CHECKSUM
        RETURN
```

```
;//////////////////////////////////////////////////////////////////////
;  FIM DO PROTOCOLO
;//////////////////////////////////////////////////////////////////////
        END
```

C.2 - PROGRAMA PRINCIPAL DO SIMULADOR DE TRÁFEGO

```
;////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
;       **********************************************************
;       ****************** PROGRAMA PRINCIPAL ********************
;       **********************************************************
;////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

        MOVLW   200                 ;Seqüência de X pacotes para simular trafego
        MOVWF   TEMP+8
envia_pacotes
        MOVLW   1                   ;transmite um pacote para o no' 1
        MOVWF   DAB1

        CLRF    TEMP+2              ;GNA Recebe byte no TEMP+2 => DB1
        MOVLW   8
        MOVWF   TEMP+1
le_numero_aleatorio
        CLRWDT
        BCF     STATUS,C            ;Coloca zero no próximo bit
        RRF     TEMP+2,F
        BTFSC   GNA                 ;Se a entrada for zero, não faz nada
        BSF     TEMP+2,MSB          ;Se a entrada for um, seta o bit
        MOVLW   TEMPO_BIT           ;Tempo de um bit
        CALL    tempo
        DECFSZ  TEMP+1,F            ;Enquanto houver bits para ler, volta
        GOTO    le_numero_aleatorio
```

```
        MOVFW   TEMP+2          ;TEMP+2 = Numero Aleatório
        MOVWF   DB1             ;FIM GNA / joga dados no DB1
;---------------
        BSF     STATUS,RP0      ;Banco 1
        MOVF    OPTION_REG,W
        MOVWF   TEMP            ;OPTION_REG => TEMP
        MOVLW   11000000B
        ANDWF   OPTION_REG,F    ;Limpa seis primeiros bits do byte
        BSF     OPTION_REG,PS2  ;Seta pre-scaler 1/32 no timer
        BCF     STATUS,RP0      ;Banco 0

;Espero X segundos (X = ALEATORIO vezes 250 vezes 32 / 1MHz)
espera_aleatorio1
        MOVLW   6               ;faz a contagem do TMR0 começar de 6
        MOVWF   TMR0            ;assim teremos 250 contagens no byte

espera_aleatorio2
        CLRWDT
        MOVF    TMR0,W
        BTFSS   STATUS,Z        ;Pula quando timer for zero
        GOTO    espera_aleatorio2

        DECFSZ  TEMP+2,F        ;Decrementa numero aleatorio
        GOTO    espera_aleatorio1

        BSF     STATUS,RP0      ;Banco 1
        MOVF    TEMP,W
        MOVWF   OPTION_REG      ;Restaura OPTION_REG
        BCF     STATUS,RP0      ;Banco 0
;---------------
        BTFSS   TEMP+8,0        ;ALTERNA ESTADO DO ACK
        BSF     ACK_PKT_TX
        BTFSC   TEMP+8,0
        BCF     ACK_PKT_TX
```

```
        CALL        tx_snap;
        DECFSZ      TEMP+8
        GOTO        envia_pacotes
;------------------------------------

begin
        CLRWDT
        NOP
        GOTO        begin   ;Gera loop infinito
```

D – MÁQUINAS DE ESTADOS FINITOS

Figura D.1: máquina de estados do protocolo SNAP Modificado

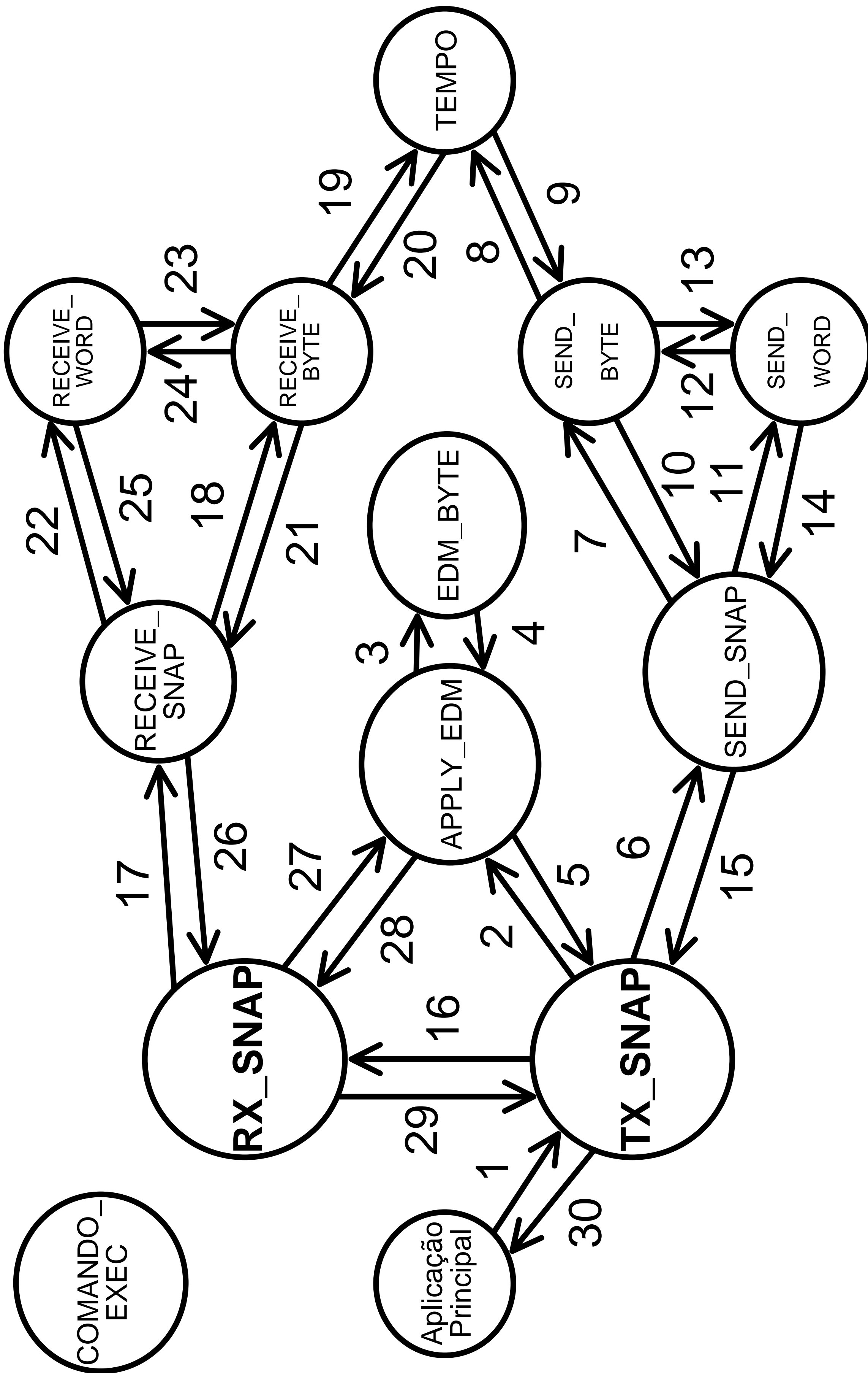

Figura D.2: máquina de estados do protocolo na transmissão

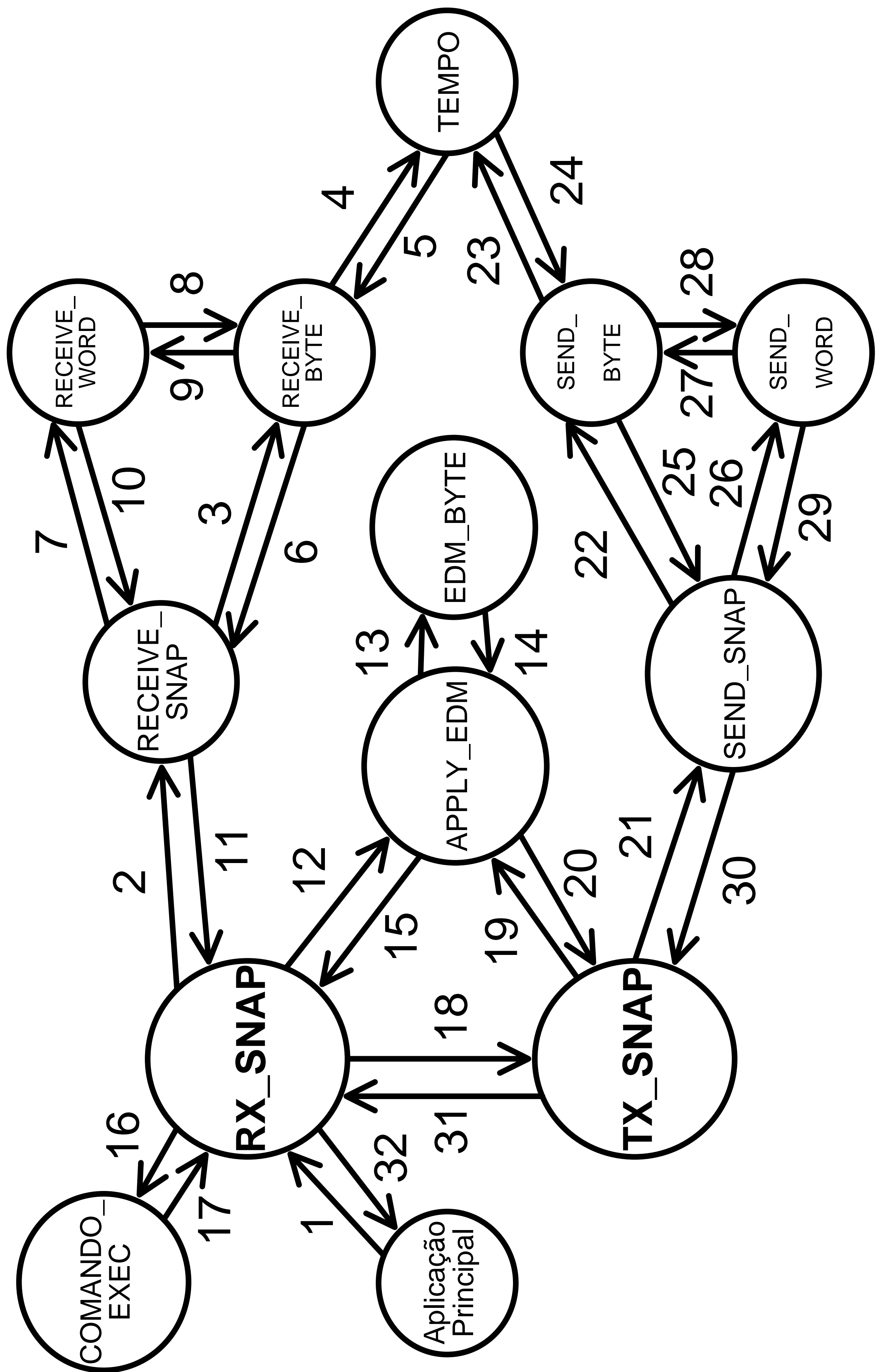

Figura D.3: máquina de estados do protocolo na recepção

E – MAPAS DE REGISTRADORES

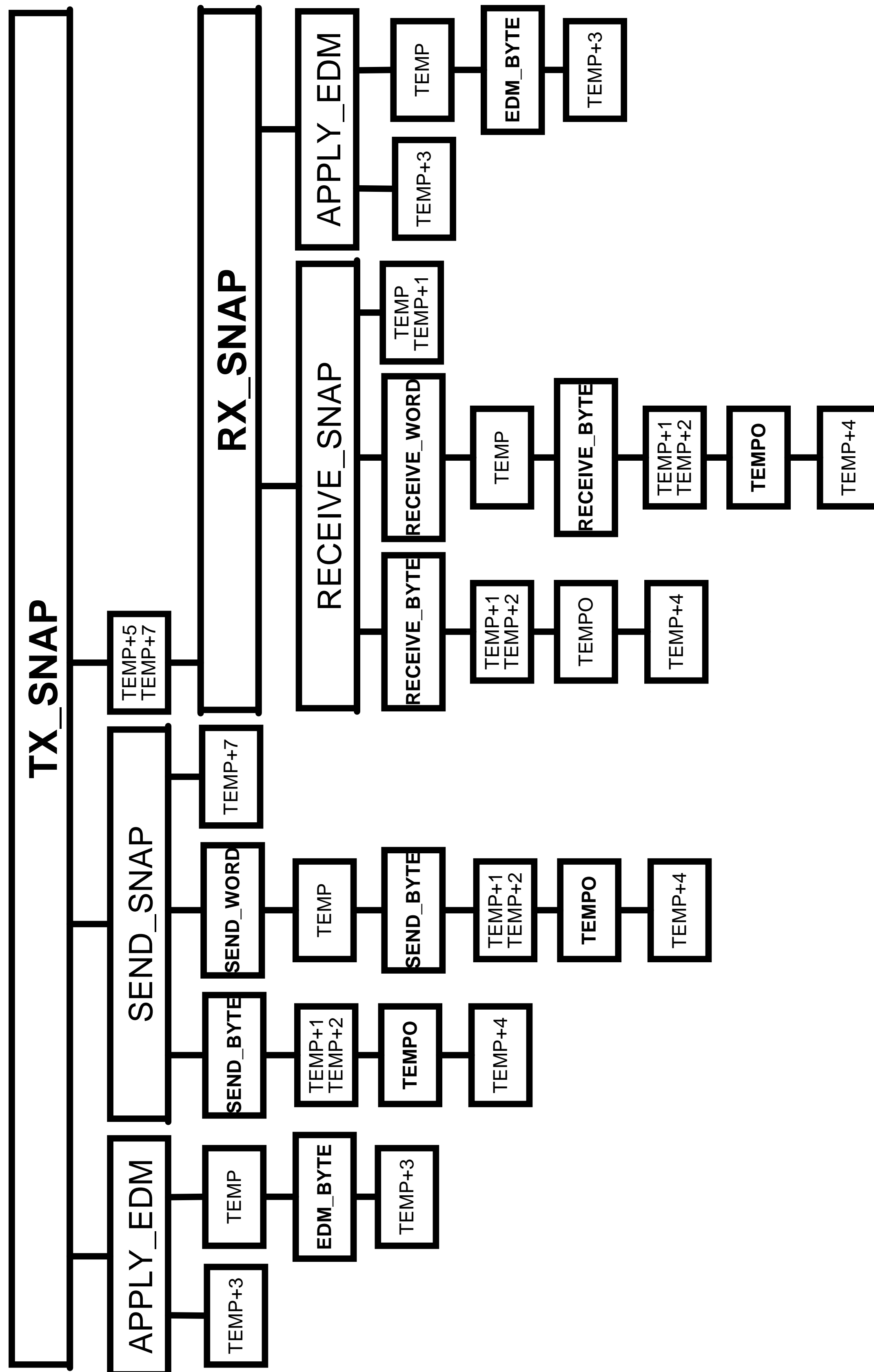

Figura E.1: mapa de registradores da transmissão

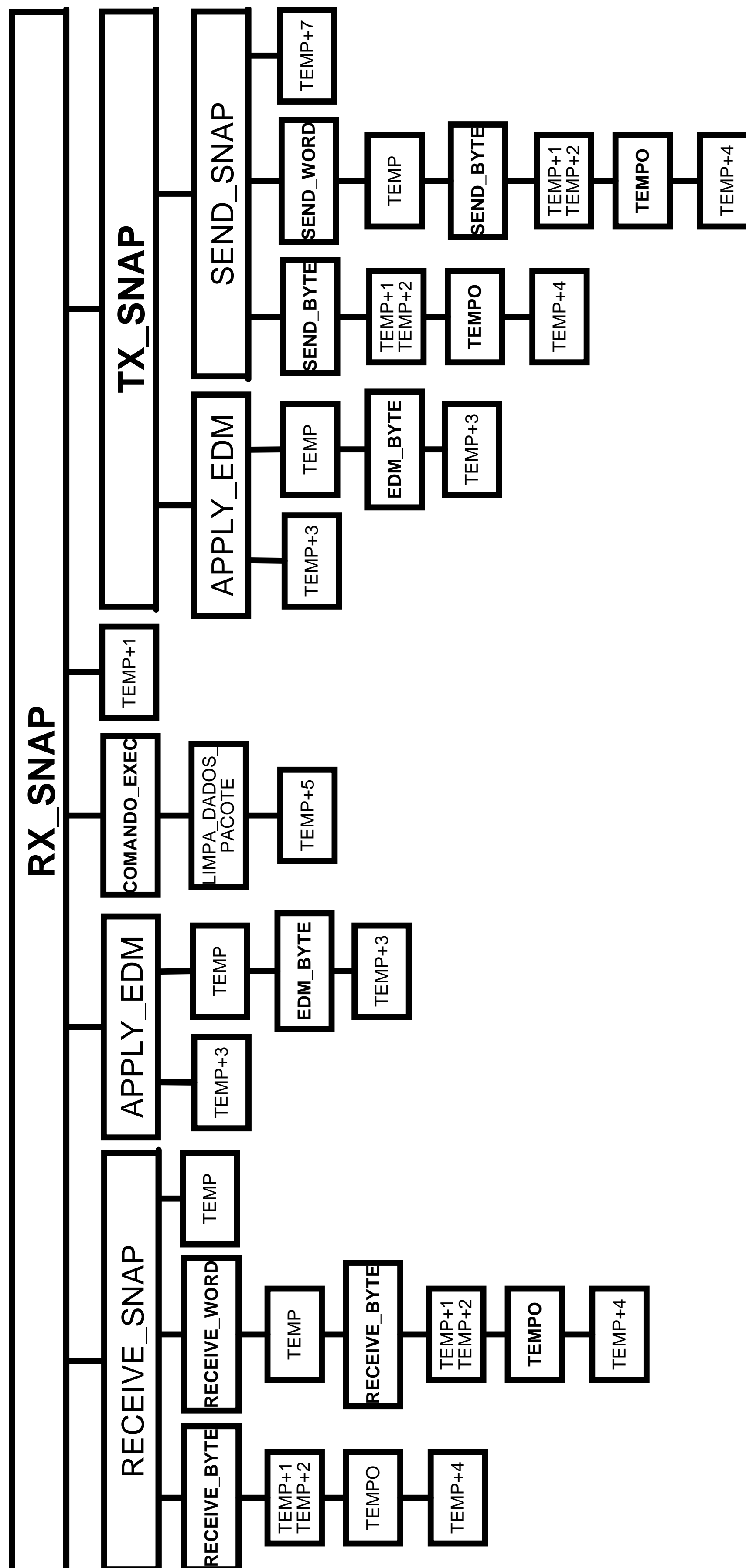

Figura E.2: mapa de registradores da recepção

F – CÓDIGO COMPLETO PARA O RISC16

F.1 - ARQUIVO SNAP16.ASM

```
#--------------------------------------------------------------------------------------------------------
#            **********  S.N.A.P. Modificado para o Risc16  *************
#      ******************* GUSTAVO LUCHINE **********************
#--------------------------------------------------------------------------------------------------------

        INCLUDE     "SNAP16.INC"       #MAPA DE MEMORIA DO PROTOCOLO
        INCLUDE     "APPL16.INC"       #DEFINICOES SOBRE END. DOS NODOS

        ORG   $0000           #Endereço chamado pelo processador quando interrompido
        J     tratador        #Pula para o tratador das interrupções

        ORG   $0001
        J     start           #Pula para as configurações iniciais


#/////////////////////////////////////////////////////////////////////////
#  TRATADOR DE INTERRUPCAO
#/////////////////////////////////////////////////////////////////////////
        ORG   $001C           #Mapa de memória termina no end. 1BH (snap16.inc)
tratador

        #ver trabalho da Juliana Zago Franca Diniz

#Diniz, J. Z. F. "Desenvolvimento de aplicação e rotina de tratamento
#de exceções para o sistema de comunicações em chip (SOC) sem fio"
#Orientador: Professor Doutor Jose Camargo da Costa. Setembro/2002.
#Universidade de Brasília (UnB).
```

```
#//////////////////////////////////////////////////////////////////////////
#  Configurações do Protocolo
#//////////////////////////////////////////////////////////////////////////
start
        LUI         $t1, $XX            #Configura SERIAL com palavra XXXXH
        ADDI        $t1, $XX
        LUI         $t2, SERup
        ADDI        $t2, SERlo
        SW          $t1, $t2, $zero

        LUI         $t1, $XX        #Configura RF com palavras XXXXH e XXXXH
        ADDI        $t1, $XX
        LUI         $t2, RF1up
        ADDI        $t2, RF1lo
        SW          $t1, $t2, $zero
        LUI         $t1, $XX
        ADDI        $t1, $XX
        LUI         $t2, RF2up
        ADDI        $t2, RF2lo
        SW          $t1, $t2, $zero

        AND         $t1, $t2, $zero     #INICIA PINO SAIDA SERIAL COM BIT 1
        ADDI        $t1, $01
        LUI         $t2, TXDup
        ADDI        $t2, TXDlo
        SW          $t1, $t2, $zero

        ADDI        $t1, $09            #ZERA OS REGISTROS DO CONTROLE
        AND         $t2, $t1, $zero
        AND         $t0, $t2, $zero
        ADDI        $t0, $01
        ADDI        $t2, RUN_RX
loop_zera_controle
        SW          $zero, $t2, $zero
```

```
        ADDI    $t2, $01                #Incrementa endereço de registro $t2
        SUB     $t1, $t0, $t1           #Decrementa índice $t1
        BEQ     $zero ,$t1, $1          #$t1=0, então sai do loop
        J       loop_zera_controle      #$t1>0, então volta para loop_zera_controle

        LUI     $t1, $01                #HABILITA INTERRUPCAO SERIAL
        LUI     $t2, SERup
        ADDI    $t2, SERlo
        LW      $t0, $t2, $zero         #Carrega conteúdo config. serial em $t0
        OR      $t0, $t0, $t1           #Liga o bit de int. da serial
        SW      $t0, $t2, $zero         #Salva conteúdo na config. serial

        AND     $s1, $zero, $s2         #HABILITA INTERRUPCAO RF
        ADDI    $s1, $40
        LUI     $s2, RF2up
        ADDI    $s2, RF2lo
        LW      $s0, $s2, $zero         #Carrega conteúdo config. RF em $t0
        OR      $s0, $s0, $s1           #Liga o bit de int. da RF
        SW      $s0, $s2, $zero         #Salva conteúdo na config. RF

#////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#       **********************************************************
#       ******************* PROGRAMA PRINCIPAL ********************
#       **********************************************************
#////////////////////////////////////////////////////////////////////////////////////////////////////////////////
begin
        AND     $t1, $zero, $t2         #Esta instrução pode ser retirada

        #Aqui entra a Aplicação Principal que usa o protocolo

        J       begin                   #Gera loop infinito
```

```
#///////////////////////////////////////////////////////////////////////
#  Função TEMPO
#///////////////////////////////////////////////////////////////////////
tempo  #REGISTRADOR $a0 TEM O VALOR DA ESPERA

        AND         $t1, $zero, $t0
        ADDI        $t1, $01

loop_bit

        SUB         $a0, $t1, $a0       #Decrementa 1 unid do valor lido (4ciclos)
        BEQ         $a0, $zero, $1      #(3ciclos)
        J           loop_bit            #(3ciclos)

        J           $ra, $zero          #RETURN - fim da função TEMPO
#///////////////////////////////////////////////////////////////////////
#  Função TX_SNAP
#///////////////////////////////////////////////////////////////////////
tx_snap
        LUI         $s1, $FE            #DESABILITA INTERRUPCAO SERIAL
        ADDI        $s1, $FF
        LUI         $s2, SERup
        ADDI        $s2, SERlo
        LW          $s0, $s2, $zero     #Carrega conteúdo config. serial em $t0
        AND         $s0, $s0, $s1       #Desliga o bit de int. da serial
        SW          $s0, $s2, $zero     #Salva conteúdo na config. serial

        LUI         $s1, $FF            #DESABILITA INTERRUPCAO RF
        ADDI        $s1, $BF
        LUI         $s2, RF2up
        ADDI        $s2, RF2lo
        LW          $s0, $s2, $zero     #Carrega conteúdo config. RF em $t0
        AND         $s0, $s0, $s1       #Desliga o bit de int. da RF
        SW          $s0, $s2, $zero     #Salva conteúdo na config. RF
```

```
AND     $t0, $zero, $t1     #RUN_TX>0 para indicar que TX esta rodando
ADDI    $t0, RUN_TX
SW      $t1, $t0, $zero

AND     $t0, $zero, $t1     #Zera OCTSYNC para o SYNC ser TX c/ 1 octeto
ADDI    $t0, OCTSYNC
AND     $t1, $zero, $t0
SW      $t1, $t0, $zero

AND     $t0, $zero, $t1     #Se RUN_RX>0 entao RUN_RTX>0 (RX->TX)
ADDI    $t0, RUN_RX
LW      $t1, $t0, $zero
AND     $t2, $zero, $t0
ADDI    $t2, RUN_RTX
BEQ     $t1, $zero, $1      #$t1 tem o valor de RUN_RX
SW      $t0, $t2, $zero

LUI     $t2, HEADER2        #Carrega HDB1 para transmitir um pacote
ADDI    $t2, HEADER1
AND     $t0, $zero, $t1
ADDI    $t0, HDB1           #end HDB1 => $t0
SW      $t2, $t0, $zero
BLT     $zero, $t1, $1      #$t1 continua tendo o valor de RUN_RX
J       carrega_ack         #Se RUN_RX=0 então verifica bit de ack

LUI     $a0, $02            #Liga o bit de resposta do ACK (RUN_RX=1)
OR      $t2, $a0, $t2       #$t2 continua tendo o valor do header
SW      $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1

AND     $a0, $zero, $t1     #SE ERR=1 NAO ZERA ULTIMO BIT ACK
ADDI    $a0, ERR
LW      $a1, $a0, $zero
BLT     $zero, $a1, $4      #Se ERR>0, então pula as 4 linhas seguintes
```

```
        LUI         $a0, $FE            #ZERA O ULTIMO BIT DO ACK (ACK=10)
        ADDI        $a0, $FF
        AND         $t2, $a0, $t2       #$t2 continua tendo o valor do header
        SW          $t2, $t0, $zero     #%t0 continua tendo o endereço de HDB1

        AND         $a0, $zero, $t1     #Verif. se já houve SAB=>DAB (1o pkt resp)
        ADDI        $a0, SAB_PARA_DAB
        LW          $t0, $a0, $zero
        BEQ         $t0, $zero, $1      #Se SAB_PARA_DAB=0, então pula
        J           charge_sab          #Se SAB_PARA_DAB>0 então carrega SAB
        AND         $t0, $zero, $t1
        ADDI        $t0, SAB1
        LW          $t1, $t0, $zero     #Carrega SAB1 em $t1
        AND         $t0, $zero, $t1
        ADDI        $t0, DAB1
        SW          $t1, $t0, $zero     #SAB1 => DAB1
        ADDI        $t1, $0F            #para garantir que $t1>0
        SW          $t1, $a0, $zero     #Grava (SAB1+0FH) para SAB_PARA_DAB>0
        J           charge_sab

#DAB1 DEVE SER CARREGADO NO APLICATIVO PRINCIPAL

carrega_ack                             #APENAS PARA O TX>0 E RX=0
        AND         $t0, $zero, $t1     #Verifica se bit do ACK deve ser desligado
        ADDI        $t0, ACK_PKT_TX
        LW          $t1, $t0, $zero
        LUI         $t2, $FE
        ADDI        $t2, $FF
        BLT         $zero, $t1, $5      #Se ACK_PKT_TX>0, então pula 5 linhas
        AND         $t1, $zero, $t2     #Se ACK_PKT_TX=0, então bit deve ser desligado
        ADDI        $t1, HDB1
        LW          $t0, $t1, $zero
        AND         $t0, $t2, $t0
        SW          $t0, $t1, $zero     #(FEFFH & HDB1)=>HDB1: desliga bit ACK
```

```
charge_sab
        LUI         $t0, MY_ADDR1up   #Se vou TX, colocar meu endereço no SAB1
        ADDI        $t0, MY_ADDR1lo
        AND         $t1, $zero, $t0
        ADDI        $t1, SAB1
        SW          $t0, $t1, $zero       #MYADDR1(up+lo)=>SAB1

        JAL         apply_edm             #Calcula o EDM para o pacote a ser enviado

        JAL         send_snap             #TRANSMITE O PACOTE

        AND         $t1, $zero, $t0       #SE RUN_RX>0, entao termina TX_SNAP
        ADDI        $t1, RUN_RX
        LW          $t0, $t1, $zero
        AND         $t1, $zero, $t2
        ADDI        $t1, HDB1
        BLT         $zero, $t0, $5        #Se rx_snap não esta sendo executado então
        LW          $t2, $t1, $zero
        LUI         $t1, $01
        ADDI        $t1, $00
        AND         $t2, $t1, $t2
        BLT         $zero, $t2, $1        #verifico se ACK=01. Se for pula 1 linha
        J           end_txsnap        #Se RX>0 ou ACK=00, vai para o fim da tx_snap

wait_response               #ESPERO PELA RESPOSTA (ACK=01)
        AND         $t0, $zero, $t1       #$t1 continua tendo 0100H
        ADDI        $t0, REJ
        SW          $t1, $t0, $zero       #Salva REJ>0: inicia c/resposta não recebida

        LUI         $t1, $01              #HABILITA INTERRUPCAO SERIAL
        LUI         $t2, SERup
        ADDI        $t2, SERlo
        LW          $t0, $t2, $zero       #Carrega conteúdo config. serial em $t0
        OR          $t0, $t0, $t1         #Liga o bit de int. da serial
```

```
        SW      $t0, $t2, $zero         #Salva conteúdo na config. serial

        AND     $s1, $zero, $s2         #HABILITA INTERRUPCAO RF
        ADDI    $s1, $40
        LUI     $s2, RF2up
        ADDI    $s2, RF2lo
        LW      $s0, $s2, $zero         #Carrega conteúdo config. RF em $t0
        OR      $s0, $s0, $s1           #Liga o bit de int. da RF
        SW      $s0, $s2, $zero         #Salva conteúdo na config. RF

        AND     $s0, $zero, $t0         #Zero contador1: $s0 de 65536 unidades.
        AND     $s1, $zero, $t0
        ADDI    $s1, REJ                #End. de REJ=>$s1
        LUI     $s2, $F6
        ADDI    $s2, $76        #(F676H + 98AH)=0000H. Contador2: 2442 unidades

wait_loop
        LW      $s3, $s1, $zero
        BLT     $zero, $s3, $1          #Se REJ>0, então pula 1 linha
        J       exit_wait               #SE REJ=0, entao um pacote foi recebido, SAI

        ADDI    $s0, $01                #Incrementa o contador1
        BEQ     $s0, $zero, $1          #Se $s0=zero então pula 1 linha
        J       wait_loop

        ADDI    $s2, $01                #Incrementa o contador2
        BEQ     $s2, $zero, $1
        J       wait_loop

exit_wait
        LUI     $s1, $FE                #DESABILITA INTERRUPCAO SERIAL
        ADDI    $s1, $FF
        LUI     $s2, SERup
        ADDI    $s2, SERlo
```

```
LW      $s0, $s2, $zero     #Carrega conteúdo config. serial em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da serial
SW      $s0, $s2, $zero     #Salva conteúdo na config. serial

LUI     $s1, $FF            #DESABILITA INTERRUPCAO RF
ADDI    $s1, $BF
LUI     $s2, RF2up
ADDI    $s2, RF2lo
LW      $s0, $s2, $zero     #Carrega conteúdo config. RF em $t0
AND     $s0, $s0, $s1       #Desliga o bit de int. da RF
SW      $s0, $s2, $zero     #Salva conteúdo na config. RF

AND     $t1, $zero, $t0
ADDI    $t1, REJ            #End. de REJ=>$t1
LW      $t0, $t1, $zero
BEQ     $t0, $zero, $1      #SE HOUVE RESPOSTA (REJ=0), PULA
J       nao_recebido        #Se nao houve resposta --> time out
AND     $t1, $zero, $t0
ADDI    $t1, ERR            #End. de ERR=>$t1
LW      $t0, $t1, $zero
BLT     $zero, $t0, $7      #SE HOUVE ERRO, VAI PARA nao_recebido
AND     $t1, $zero, $t0
ADDI    $t1, HDB1
LW      $t0, $t1, $zero
LUI     %t1, $01
ADDI    $t1, $00
AND     $t0, $t1, $t0
BEQ     $t0, $zero, $1      #Se REJ=0 e ERR=0, entao testar se ACK=10
J       nao_recebido
AND     $t0, $zero, $t1
ADDI    $t0, RECEBIDO
SW      $t1, $t0, $zero     #REJ=0, ERR=0, ACK=10, entao RECEBIDO>0
J       end_txsnap
```

```
nao_recebido
        AND     $t0, $zero, $t1
        ADDI    $t0, RECEBIDO
        SW      $zero, $t0, $zero   #Pacote não foi recebido adequadamente

end_txsnap
        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RTX
        SW      $zero, $t0, $zero   #RUN_RTX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_TX
        SW      $zero, $t0, $zero   #RUN_TX=0

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_RX
        LW      $t1, $t0, $zero
        BEQ     $t1, $zero, $1      #Habilita interrup. se RX não estiver rodando
        J       $ra, $zero

        LUI     $t1, $01            #HABILITA INTERRUPCAO SERIAL
        LUI     $t2, SERup
        ADDI    $t2, SERlo
        LW      $t0, $t2, $zero     #Carrega conteúdo config. serial em $t0
        OR      $t0, $t0, $t1       #Liga o bit de int. da serial
        SW      $t0, $t2, $zero     #Salva conteúdo na config. serial

        AND     $s1, $zero, $s2     #HABILITA INTERRUPCAO RF
        ADDI    $s1, $40
        LUI     $s2, RF2up
        ADDI    $s2, RF2lo
        LW      $s0, $s2, $zero     #Carrega conteúdo config. RF em $t0
        OR      $s0, $s0, $s1       #Liga o bit de int. da RF
        SW      $s0, $s2, $zero     #Salva conteúdo na config. RF
```

```
        J           $ra, $zero          #FIM DA TRANSMISSAO

#//////////////////////////////////////////////////////////////////////
#       Função SEND_SNAP
#//////////////////////////////////////////////////////////////////////
send_snap
        AND         $a0, $zero, $t2
        LUI         $a0, SYNC
        JAL         send_byte           #Envia octeto SYNC

        AND         $t1, $zero, $t2
        ADDI        $t1, HDB1
        LW          $a0, $t1, $zero
        JAL         send_byte           #Envia byte HDB1

        AND         $t1, $zero, $t2
        ADDI        $t1, DAB1
        AND         $t2, $zero, $t1
        ADDI        $t2, TEMP7
        LW          $a0, $t1, $zero
        SW          $a0, $t2, $zero     #Salva DAB1 => TEMP7
        JAL         send_byte           #Envia byte DAB1

        AND         $t1, $zero, $t2
        ADDI        $t1, SAB1
        LW          $a0, $t1, $zero
        JAL         send_byte           #Envia byte SAB1

        AND         $gp, $zero, $t2
        ADDI        $gp, DB8            #Ponteiro aponta para DB8
        AND         $a0, $zero, $t2
        ADDI        $a0, $08          #$a0 contém o número de bytes a serem enviados
        JAL         send_word
```

```
        AND     $t1, $zero, $t2
        ADDI    $t1, CRC1
        LW      $a0, $t1, $zero
        JAL     send_byte           #Envia byte CRC1

        J       $ra, $zero          #Fim da send_snap

#//////////////////////////////////////////////////////////////////////
#  Função SEND_WORD
#//////////////////////////////////////////////////////////////////////
send_word
        BLT     $zero, $a0, $1      #$a0=0, então não tem bytes a serem enviados
        J       end_send_word
        ADD     $s0, $zero, $a0     #Copia $a0 em $s0 (registrador preservado)
        AND     $t0, $zero, $t1
        ADDI    $t0, TEMP
        SW      $gp, $t0, $zero     #Salva ponteiro em TEMP

sendw_loop
        LW      $a0, $gp, $zero     #$a0 tem o conteúdo do ponteiro
        JAL     send_byte           #Transmite o byte do registrador $a0

        AND     $t0, $zero, $t1     #INCREMENTA PONTEIRO
        ADDI    $t0, TEMP
        LW      $gp, $t0, $zero     #TEMP e' o endereço do ponteiro
        ADDI    $gp, $01
        SW      $gp, $t0, $zero     #Incrementa o endereço do ponteiro

        AND     $t1, $zero, $t0     #Decrementa o contador de palavras
        ADDI    $t1, $01
        SUB     $s0, $t1, $s0       #Decrementa $s0 (registrador preservado)

        BEQ     $s0, $zero, $1      #CONTADOR $s0=0, ENTAO TERMINA
```

```
        J           sendw_loop          #Se $s0>0, então volta para o loop

end_send_word
        J           $ra, $zero          #FIM DA SEND_WORD

#//////////////////////////////////////////////////////////////////////
#  Funcao SEND_BYTE
#//////////////////////////////////////////////////////////////////////
send_byte
        ADD         $s0, $zero, $a0     #Copia $a0 em $s0 (registrador preservado)

        AND         $t1, $zero, $t0    #Verifica se a transmissão será SERIAL OU RF
        ADDI        $t1, TX_RF
        LW          $tO, $t1, $zero     #Carrega conteúdo RF, se>0, então vai p/ RF
        BEQ         $t0, $zero, $1      #Se TX_RF=0, então transmite pela serial
        J           start_rf



        LUI         $s2, TXDup          #INICIO TRANSMISSAO SERIAL
        ADDI        $s2, TXDlo          #Carrega endereço TX serial
        ADD         $s3, $zero, $t1     #Controla o octeto que esta sendo TX
start_bit
        LUI         $s1, $FF
        ADDI        $s1, $F8           #n° de bits para ser enviado (FFF8H + 8 = zero)
        SW          $zero, $s2, $zero   #TRANSMITE O START BIT
        LUI         $a0, TEMPO_BITup #Carrega o tempo de um bit
        ADDI        $a0, TEMPO_BITlo
        JAL         tempo               #Espera o tempo de um bit

sendb_loop
        SW          $s0, $s2, $zero     #Transmite o bit na interface SERIAL
        SHIFT       $s0, $01            #Deslocamento um bit para a direita

        LUI         $a0, TEMPO_BITup #Carrega o tempo de um bit
```

```
        ADDI    $a0, TEMPO_BITlo
        JAL     tempo                 #Espera o tempo de um bit

        ADDI    $s1, $01              #Incrementa o n° de bits para ser enviado
        BEQ     $s1, $zero, $1        #quando $s1=zero (8 bits enviados), então pula
        J       sendb_loop

        LUI     $t0, $FF
        ADDI    $t0, $FF
        SW      $t0, $s2, $zero       #TRANSMITE O STOP BIT
        LUI     $a0, TEMPO_BITup      #Carrega o tempo de um bit
        ADDI    $a0, TEMPO_BITlo
        JAL     tempo                 #Espera o tempo de um bit

        AND     $t0, $zero, $t1       #Se for o SYNC, transmitir apenas 1 octeto
        ADDI    $t0, OCTSYNC
        LW      $a0, $t0, $zero
        BLT     $zero, $a0, $2        #Se OCTSYNC>0, o octeto foi TX e continua
        SW      $t0, $a0, $zero       #Se OCTSYNC=0, é o octeto de sincronismo,
        J       fim_send_byte         #grava OCTSYNC>0 e termina função

SEND_BYTE
        BEQ     $s3, $zero, $2        #TRANSMITE O SEGUNDO OCTETO
        AND     $s3, $zero, $s1       #Zera o regist. $s3 de controle 2o octeto
        J       start_bit             #Retorna para transmitir o segundo octeto
        J       fim_send_byte         #Terminou TX serial, vai para fim da função


start_rf
        LUI     $s1, $FF              #INICIO TRANSMISSAO PELA RF
        ADDI    $s1, $F0              #n° de bits para ser enviado (FFF0H + 16 = zero)
        LUI     $s2, TXFup            #Carrega endereço TX RF
        ADDI    $s2, TXFlo
```

```
        AND     $t0, $zero, $t1 #Se for o SYNC, transmitir apenas 1 octeto
        ADDI    $t0, OCTSYNC
        LW      $a0, $t0, $zero
        BLT     $zero, $a0, $3        #Se OCTSYNC>0, o octeto foi TX e continua
        SW      $t0, $a0, $zero    #Se OCTSYNC=0, este é o octeto de sincronismo,
        LUI     $s1, $FF              #torna o n° de bits a ser enviado igual a 8,
        ADDI    $s1, $F8              #pois FFF8H + 8 = zero

sendb_loop_rf
        SW      $s0, $s2, $zero       #TRANSMITE O BIT NA INTERFACE RF
        SHIFT   $s0, $01              #Deslocamento um bit para a direita

        LUI     $a0, TEMPO_BITup  #Carrega o tempo de um bit
        ADDI    $a0, TEMPO_BITlo
        JAL     tempo                 #Espera o tempo de um bit

        ADDI    $s1, $01              #Incrementa o n° de bits para ser enviado
        BEQ     $s1, $zero, $1       #quando $s1=zero(16 bits enviados), então pula
        J       sendb_loop_rf

fim_send_byte
        J       $ra, $zero            #FIM DA SEND_BYTE

#////////////////////////////////////////////////////////////////////////
#  Função RX_SNAP
#////////////////////////////////////////////////////////////////////////
rx_snap
        LUI     $t0, $FF
        AND     $t1, $zero, $t0
        ADDI    $t1, RUN_RX
        SW      $t0, $t1, $zero       #Indica que rx_snap esta sendo executada

        AND     $t1, $zero, $t0       #BACKUP DO VALOR DE TX_RF
        ADDI    $t1, TX_RF
```

```
LW      $t0, $t1, $zero     #Carrego valor TX_RF => $t0
AND     $t1, $zero, $t0
ADDI    $t1, TX_RF2
SW      $t0, $t1, $zero     #Salva valor TX_RF => TX_RF2

AND     $t0, $zero, $t1     #Zera OCTSYNC para o SYNC ser RX C/ 1 octeto
ADDI    $t0, OCTSYNC
AND     $t1, $zero, $t0
SW      $t1, $t0, $zero

JAL     receive_snap        #Efetivamente recebe o pacote do protocolo

AND     $t1, $zero, $t0
ADDI    $t1, REJ
LW      $t0, $t1, $zero     #Carrega REJ no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se o pacote foi rejeitado, sai da rx_snap

JAL     apply_edm           #Calcula o método de detecção de erros

AND     $t1, $zero, $t0
ADDI    $t1, ERR
LW      $t0, $t1, $zero     #Carrega ERR no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se o pacote teve erro, sai da rx_snap

AND     $t1, $zero, $t0
ADDI    $t1, RUN_TX
LW      $t0, $t1, $zero     #Carrega RUN_TX no registrador $t0
BEQ     $t0, $zero, $1
J       end_rxsnap          #Se RUN_TX esta rodando, sai da rx_snap
```

```
runtx_0         #TX NAO ESTA RODANDO
        AND     $t1, $zero, $t0
        ADDI    $t1, SAB_PARA_DAB
        SW      $zero, $t1, $zero       #Limpa registrador SAB_PARA_DAB

        JAL     comando_exec            #SE FOR UM COMANDO, EXECUTA

        AND     $t1, $zero, $t0
        ADDI    $t1, HDB1
        LW      $t0, $t1, $zero         #Carrega HDB1 no registrador $t0
        LUI     $t1, $01
        ADDI    $t1, $00
        AND     $t0, $t1, $t0           #Verifica se o pacote exige resposta
        BLT     $zero, $t0, $1          #Ou estou recebendo novo PKT que exige ACK
        J       end_rxsnap      #Ou estou recebendo novo PKT e termino a recepção

espera_broadcast
        AND     $t1, $zero, $t0
        ADDI    $t1, DAB1
        LW      $t0, $t1, $zero         #Carrega DAB1 no registrador $t0
        BEQ     $t0, $zero, $1          #SE FOR BROADCAST, PULA 1 LINHA
        J       continua_resposta       #Não sendo broadcast, continua normalmente

        LUI     $s1, MY_ADDR1up #Como é de broadcast, pega endereço do no'
        ADDI    $s1, MY_ADDR1lo # como parametro para a espera antes da
        AND     $t1, $zero, $t0         # resposta.
        ADDI    $t1, $01                #Serve de referencia para decrementar o $s1

espera_broad
        AND     $s0, $zero, $t0         #Zero contador1: $s0 de 65536 unidades.
        LUI     $s2, $FE
        ADDI    $s2, $CF        #(FECFH + 131H)=0000H. Contador2: 305 unidades.
```

```
wait_loop_rx
        ADDI        $s0, $01              #Incrementa o contador1
        BEQ         $s0, $zero, $1        #Se $s0=zero entao pula 1 linha
        J           wait_loop_rx

        ADDI        $s2, $01              #Incrementa o contador2
        BEQ         $s2, $zero, $1
        J           wait_loop_rx



        SUB         $s1, $t1, $s1         #Decrementa 1 do valor do endereco do nodo
        BEQ         $s1, $zero, $1        #Se valor=0, entao sai da espera
        J           espera_broad

continua_resposta
        JAL         tx_snap               #Responde caso TX nao esteja rodando

end_rxsnap
        AND         $t1, $zero, $t0       #RESTAURA O VALOR DE TX_RF
        ADDI        $t1, TX_RF2
        LW          $t0, $t1, $zero       #Carrego valor TX_RF2 => $t0
        AND         $t1, $zero, $t0
        ADDI        $t1, TX_RF
        SW          $t0, $t1, $zero       #Salva valor TX_RF2 => TX_RF

        AND         $t0, $zero, $t1
        ADDI        $t0, RUN_RX
        SW          $zero, $t0, $zero     #Zera o registrador RUN_RX

        J           $ra, $zero            #FIM DA RX_SNAP
```

```
#////////////////////////////////////////////////////////////////////////
#        Funcao RECEIVE_SNAP
#////////////////////////////////////////////////////////////////////////
receive_snap
        JAL     receive_byte        #Recebe o octeto de sincronismo (SYNC)

        JAL     receive_byte        #RECEBE O BYTE DE HEADER (HDB2 e HDB1)
        AND     $t0, $zero, $t1
        ADDI    $t0, HDB1
        SW      $a0, $t0, $zero     #Salva byte lido ($a0) em HDB1

        AND     $t0, $zero, $t1
        ADDI    $t0, RUN_TX         #Verifica se tx esta sendo executado
        LW      $t1, $t0, $zero
        LUI     $t0, HEADER2
        ADDI    $t0, HEADER1
        BLT     $zero, $t1, $1      #SE TX ESTA RODANDO, ENTAO
        J       run_tx0
        LUI     $t2, $02            #Estou recebendo um pacote de resposta
        ADDI    $t2, $00
        OR      $t1, $t2, $t0
        LUI     $t2, $FE
        ADDI    $t2, $FF
        AND     $t1, $t2, $t1       #ACK[$t1]=10
        XOR     $s0, $t1, $a0       #HDB1=HEADER[ACK=10]?
        LUI     $t2, $01
        ADDI    $t2, $00
        OR      $t1, $t2, $t1       #ACK[$t1]=11
        XOR     $s1, $t1, $a0       #HDB1=HEADER[ACK=11]?
        AND     $s2, $s1, $s0       #JUNTA AS DUAS COMPARACOES FEITAS
        BEQ     $s2, $zero, $1      #Rejeita se ACK recebido é diferente de 10 ou 11
        J       reject
        J       receive_dab
```

```
run_tx0         #Se TX não está rodando, então estou recebendo um novo pacote

# Compara se o pacote recebido tem HEADERS VALIDOS (IGUAIS ENVIADOS)
# COM A EXCECAO DE RECEBER COMANDOS (HDB1,7=1) E
# PACOTES SEM ACK (HDB1,8=0)
        LUI     $t2, $01
        ADDI    $t2, $00
        OR      $t1, $t2, $a0           #Seta o bit de ACK em $t1
        LUI     $t2, $FF
        ADDI    $t2, $7F
        AND     $t1, $t2, $t1           #Zera o bit de comando em $t1
        XOR     $s0, $t1, $t0           #$t0 ainda tem o HEADER (HDB2 + HDB1)
        BEQ     $s0, $zero, $1
        J       reject

receive_dab
        JAL     receive_byte            #RECEBE O BYTE DE DESTINO (DAB1)
        AND     $t0, $zero, $t1
        ADDI    $t0, DAB1
        SW      $a0, $t0, $zero         #Salva byte lido ($a0) em DAB1

        XOR     $t1, $zero, $a0         #VERIFICA SE E' BROADCAST
        BLT     $zero, $t1, $1          #Caso negativo teste se é o end. deste nó
        J       receive_sab             #Caso positivo continua a partir do receive_sab

        LUI     $t2, MY_ADDR1up         #Se MY_ADDR1 != DABi então rejeita
        ADDI    $t2, MY_ADDR1lo
        XOR     $t1, $t2, $a0
        BEQ     $t1, $zero, $1          #o pacote e termina a função
        J       reject

receive_sab
        JAL     receive_byte            #RECEBE O BYTE DE ORIGEM (SAB1)
        AND     $t0, $zero, $t1
```

```
        ADDI    $t0, SAB1
        SW      $a0, $t0, $zero         #Salva byte lido ($a0) em SAB1

        LUI     $t2, MY_ADDR1up         #Se SABi for igual ao MY_ADDR1 então
        ADDI    $t2, MY_ADDR1lo
        XOR     $t1, $t2, $a0           #existem dois nodos com o mesmo endereço e o
        BLT     $zero, $t1, $1
        J       reject                  #presente pacote e' rejeitado

also_sab
        AND     $t2, $zero, $t1
        ADDI    $t2, RUN_TX             #Verifica se tx esta sendo executado
        LW      $t1, $t2, $zero
        BLT     $zero, $t1, $1
        J       receive_db              #Caso não esteja, pula para receive_db

        AND     $t2, $zero, $t1         #Caso esteja, verifica se a resposta veio
        ADDI    $t2, TEMP7
        LW      $t0, $t2, $zero         #do nodo para o qual o pacote foi enviado
        XOR     $t1, $t0, $a0           #Se tiver vindo do nodo o qual o pacote
        BEQ     $t1, $zero, $1          #foi transmitido anteriormente, pula 1 linha
        J       reject                  #Caso seja de outro nodo, rejeita o pacote

receive_db
        LUI     $gp, $00
        ADDI    $gp, DB8                #Aponta ponteiro para o DB8
        LUI     $a0, $00
        ADDI    $a0, $08                #Indica numero de palavras a receber
        JAL     receive_word            #RECEBE OS BYTES DE DADOS (DBi)

receive_eb
        JAL     receive_byte            #RECEBE O BYTE DE CRC-8 (CRC1)
        AND     $t0, $zero, $t1
        ADDI    $t0, CRC1
```

```
        SW      $a0, $t0, $zero         #Salva byte lido ($a0) em CRC1

        AND     $t0, $zero, $t1
        ADDI    $t0, REJ
        SW      $zero, $t0, $zero       #Recebido pacote a ser verificado = REJ=0
        J       end_receivesnap

reject
        LUI     $t1, $FF
        AND     $t0, $zero, $t1
        ADDI    $t0, REJ
        SW      $t1, $t0, $zero         #PACOTE REJEITADO = REJ>0

end_receivesnap
        J       $ra, $zero              #FIM DA RECEIVE_SNAP

#//////////////////////////////////////////////////////////////////////
#       Funcao RECEIVE_WORD
#//////////////////////////////////////////////////////////////////////
receive_word
        BLT     $zero, $a0, $1          #Checa se a palavra recebida tem
        J       end_receiveword         # ZERO BYTE
        AND     $t0, $zero, $t1
        ADDI    $t0, $01
        ADD     $s0, $zero, $a0         #Passa $a0 para $s0 (registrador preservado)

receivew_loop
        JAL     receive_byte            #Lê o byte na interface de entrada => $a0

        SW      $a0, $gp, $zero         #Coloca o $a0 no conteúdo do PONTEIRO
        ADDI    $gp, $01                #Incrementa o ENDERECO DO PONTEIRO
        SUB     $s0, $t0, $s0           #Decrementa o contador de bytes a receber
        BEQ     $s0, $zero, $1
        J       receivew_loop
```

```
end_receiveword
        J               $ra, $zero

#/////////////////////////////////////////////////////////////////////////////
#       Funcao RECEIVE_BYTE
#/////////////////////////////////////////////////////////////////////////////
receive_byte
        AND             $s0, $zero, $t0         #Prepara para receber byte em $s0
                                                # que é um registrador preservado

        LW              $t0, $int, $zero        #Verifico registrador $int para saber a
        AND             $t1, $zero, $t0         # origem da interrupção
        ADDI            $t1, $01                #Como estas funções somente são executadas
        AND             $t1, $t0, $t1           # se ocorrer int de RF ou de Serial, então:
        BEQ             $t1, $zero, $1          #   Se tiver 0 no final e' Serial
        J               start_rx_rf             #   Se tiver 1 no final e' RF


        LUI             $s2, RXDup              #INICIO RECEPCAO SERIAL
        ADDI            $s2, RXDlo
        ADD             $s3, $zero, $s2         #Controla octeto que esta sendo recebido
recebe_start_bit
        LUI             $s1, $FF              #No de bits para ser recebido(FFF8H + 8 = zero)
        ADDI            $s1, $F8
        AND             $t0, $zero, $t1         #Zero contador1: $t0 de 65536 unidades.
        AND             $t1, $zero, $t0
        ADDI            $t1, TX_RF
        SW              $t0, $t1, $zero         #Zera regist. TX_RF para indicar resp.
        LUI             $t2, $FF                # p/serial
        ADDI            $t2, $6A        #(FF6AH + 96H)=0000H. Contador2: 150 unidades.

espera_byte
        LW              $t1, $s2, $zero         #Não precisa de mascara pq 15 bits são terra
        BLT             $zero, $t1, $1          #Se $t1>0, então pula 1 linha
```

```
        J           continua_byte       #Se $t1=0, entao um bit foi recebido, SAI

        ADDI        $t0, $01            #Incrementa o contador1
        BEQ         $t0, $zero, $1      #Se $s0=zero então pula 1 linha
        J           espera_byte

        ADDI        $t2, $01            #Incrementa o contador2
        BEQ         $t2, $zero, $1
        J           espera_byte

continua_byte
#FIM DA ESPERA DO START BIT (OU START BIT VEIO, OU HOUVE TIMEOUT)
        LUI         $t0, TEMPO_BITup              #Tempo de um bit
        ADDI        $t0, TEMPO_BITlo
        LUI         $t1, TEMPO_MEIO_BITup         #Tempo de meio bit
        ADDI        $t1, TEMPO_MEIO_BITlo
        ADD         $a0, $t0, $t1
        JAL         tempo               #ESPERA O TEMPO DE UM BIT E MEIO

receive_loop
        LW          $t1, $s2, $zero     #LE O BIT DA INTERFACE SERIAL
        BEQ         $t1, $zero, $1      #Se for zero, pula
        ADDI        $s0, $01            #Se for um, liga o bit menos significativo
        SHIFT       $s0, $01            #Deslocamento de um bit para a direita

        LUI         $a0, TEMPO_BITup    #Tempo de um bit
        ADDI        $a0, TEMPO_BITlo
        JAL         tempo               #ESPERA O TEMPO DE UM BIT

        ADDI        $s1, $01            #Incrementa o num de bits recebidos
        BEQ         $s1, $zero, $1      #Se todos os 8 bits foram recebidos, pula
        J           receive_loop        #Enquanto não forem, retorna p/ receive_loop

        AND         $t0, $zero, $t1     #Se for o SYNC, receber apenas 1 octeto
```

```
        ADDI    $t0, OCTSYNC
        LW      $a0, $t0, $zero
        BLT     $zero, $a0, $2          #Se OCTSYNC>0, o octeto foi TX e continua
        SW      $t0, $a0, $zero         #Se OCTSYNC=0, é o octeto de sincronismo,
        J       fim_receive_byte        #então vai para o fim da função

        BEQ     $s3, $zero, $2          #RECEBE O SEGUNDO OCTETO
        AND     $s3, $zero, $s1         #Zera registrador $s3 de controle 2o octeto
        J       recebe_start_bit
        J       fim_receive_byte        #Terminou a recepção serial, sai da função


start_rx_rf
        LUI     $s1, $FF                #INICIO DA RECEPCAO PELA RF
        ADDI    $s1, $F0                #Num de bits para ser recebido (FFF0H+16=zero)
        LUI     $s2, RXFup
        ADDI    $s2, RXFlo              #Carrega end. da RX RF

        AND     $t0, $zero, $t1         #Se for o SYNC, receber apenas 1 octeto
        ADDI    $t0, OCTSYNC
        LW      $a0, $t0, $zero
        BLT     $zero, $a0, $3          #Se OCTSYNC>0, o octeto foi TX e continua
        SW      $t0, $t0, $zero         #Se OCTSYNC=0, é o octeto de sincronismo,
        LUI     $s1, $FF                # torna o n° de bits a ser recebido igual
        ADDI    $s1, $F8                # a 8,pois FFF8H + 8 = zero

        AND     $t1, $zero, $t0
        ADDI    $t1, TX_RF
        SW      $s1, $t1, $zero         #TX_RF>0 p/ indicar resposta p/RF

        LUI     $a0, TEMPO_MEIO_BITup
        ADDI    $a0, TEMPO_MEIO_BITlo
        JAL     tempo                   #Espera meio bit para ler os bits no meio
```

```
receive_loop_rf
        LW      $t1, $s2, $zero         #LE O BIT DA INTERFACE RF
        BEQ     $t1, $zero, $1          #Se for zero, pula
        ADDI    $s0, $01                #Se for um, liga o bit menos significativo
        SHIFT   $s0, $01                #Deslocamento de um bit para a direita

        LUI     $a0, TEMPO_BITup        #Tempo de um bit
        ADDI    $a0, TEMPO_BITlo
        JAL     tempo                   #ESPERA O TEMPO DE UM BIT

        ADDI    $s1, $01                #Incrementa o num de bits recebidos
        BEQ     $s1, $zero, $1          #Se todos os 16 bits foram recebidos, pula
        J       receive_loop_rf         #Enquanto não forem, retorna p/ receive_loop_rf

fim_receive_byte
        J       $ra, $zero              #FIM DA RECEIVE_BYTE

#////////////////////////////////////////////////////////////////////////
#       Função COMANDO_EXEC
#       Esta função verifica se o pacote recebido tem um comando a ser
#       realizado e o executa.
#////////////////////////////////////////////////////////////////////////
comando_exec                            #TX NAO ESTA RODANDO
#VERIFICAR SE PACOTE RECEBIDO E' COMANDO, se for colocar o # no $s0
        AND     $t1, $zero, $t0
        ADDI    $t1, HDB1
        LW      $t0, $t1, $zero         #Carrega HDB1 => $t0
        AND     $t1, $zero, $t0
        ADDI    $t1, $80
        AND     $t1, $t0, $t1           #$t1=(HDB1 & 0080H) para ver se e' comando
        BLT     $zero, $t1, $1
        J       fim_comando
```

```
#---------------------------------------------
#INTERPRETAR O COMANDO E TOMAR ACAO

comando2            #Exemplo de comando
        AND         $t1, $zero, $t0
        ADDI        $t1, DB1
        LW          $t0, $t1, $zero         #Carrega comando que fica em DB1
        AND         $t1, $zero, $t0
        ADDI        $t1, $02                #Verifica o código 2 do comando
        XOR         $t2, $t1, $t0
        BEQ         $t2, $zero, $1          #Se for igual, pula 1 linha
        J           comando4                #Próximo comando
        #.
        #ação do comando: abre válvula, desliga válvula, limpa memória, etc
        #.
        J     fim_comando

comando4            #Exemplo de comando
        AND         $t1, $zero, $t0
        ADDI        $t1, $04                #Verifica o código 4 do comando
        XOR         $t2, $t1, $t0
        BEQ         $t2, $zero, $1  #Se for igual, pula 1 linha
        J           nenhum_comando     #Ultimo comando vai para "nenhum_comando"
        #.
        #ação do comando: transmite os dados para a Estação de Campo, etc
        #.
        J     fim_comando

#INCLUIR MAIS COMANDOS AQUI, conforme a necessidade da aplicação
#LEMBRANDO QUE CADA COMANDO PULA PARA O SEGUINTE E O ULTIMO
#CONTEM "J nenhum_comando" DEPOIS DA INSTRUCAO "BEQ"
```

```
#----------------------------------------------
nenhum_comando
        #Retorna algum código de erro nos campos de dados do pacote para
        #indicar que nenhum comando foi realizado

fim_comando
        J       $ra, $zero              #FIM DO COMANDO_EXEC

#/////////////////////////////////////////////////////////////////////
#       Funcao APPLY_EDM
#/////////////////////////////////////////////////////////////////////
apply_edm
        AND     $s0, $zero, $t0         #Prepara para calcular EDM em $s0
        AND     $t1, $zero, $t0         # que é um registrador preservado
        ADDI    $t1, TEMP3              #Zera registrador temporário TEMP3
        SW      $s0, $t1, $zero



start_edm
        AND     $t1, $zero, $t0         #Carrega HDB1 e chama o EDM para o byte
        ADDI    $t1, HDB1
        LW      $a0, $t1, $zero
        JAL     edm_byte

        AND     $t1, $zero, $t0         #Carrega DAB1 e chama o EDM para o byte
        ADDI    $t1, DAB1
        LW      $a0, $t1, $zero
        JAL     edm_byte

        AND     $t1, $zero, $t0         #Carrega SAB1 e chama o EDM para o byte
        ADDI    $t1, SAB1
        LW      $a0, $t1, $zero
        JAL     edm_byte
```

```
        AND     $s1, $zero, $t0         #Prepara para calcular os bytes de dados
        ADDI    $s1, DB8
        LUI     $s2, $FF                #FFF8H + 8 = 0000H -> Contador1: 8 unidades
        ADDI    $s2, $F8

edm_loop
        LW      $a0, $s1, $zero         #COMPUTA DBi
        JAL     edm_byte
        ADDI    $s1, $01
        ADDI    $s2, $01
        BEQ     $s2, $zero, $1
        J       edm_loop

        AND     $t1, $zero, $t0         #Carrega resultado final de CRC16 em $s0
        ADDI    $t1, TEMP3
        LW      $s0, $t1, $zero

apply_final     #Decisão sobre o que fazer com o resultado do EDM calculado
        AND     $t1, $zero, $t0
        ADDI    $t1, RUN_RX
        LW      $t0, $t1, $zero
        BLT     $zero, $t0, $1          #Se RUN_RX=0 um novo pkt está sendo enviado
        J       apply_ass               #então aplica TEMP3 em CRCi

        AND     $t1, $zero, $t0
        ADDI    $t1, RUN_TX             #Se RUN_RX=1 e RUN_TX=0, então um novo pkt
        LW      $t0, $t1, $zero
        BLT     $zero, $t0, $1          #esta sendo recebido. Comparação dos CRCs
        J       apply_confr             #recebido e calculado

        AND     $t1, $zero, $t0         #Se RUN_RX=RUN_TX=1 e RUN_RTX=0, então
        ADDI    $t1, RUN_RTX            #foi recebido o pacote de resposta. Deve haver
        LW      $t0, $t1, $zero         #a comparação entre CRCs. Se RUN_RTX=1
        BLT     $zero, $t0, $1          # então será transmitido um pacote de resposta
```

```
        J           apply_confr          #e o TEMP3 deve ser aplicado em CRCi.

apply_ass           #Aplica CRC calculado => CRC1 do pacote
        AND         $t1, $zero, $t0
        ADDI        $t1, CRC1
        SW          $s0, $t1, $zero      #Aplica $s0 => CRC1
        J           end_apply_edm

apply_confr         #Confere $s0 com o CRC1 recebido e seta o bit ERR

        AND         $t1, $zero, $t0
        ADDI        $t1, ERR
        SW          $zero, $t1, $zero    #Limpa byte ERR=0

        AND         $t2, $zero, $t0
        ADDI        $t2, CRC1
        LW          $t0, $t2, $zero      #$t0=CRC do pacote
        XOR         $t2, $s0, $t0        #$s0=CRC calculado
        BEQ         $t2, $zero, $2       #Confere e seta erro de acordo
        LUI         $s3, $FF
        SW          $s3, $t1, $zero      #Houve erro: ERR>0

end_apply_edm
        J           $ra, $zero           #FIM DA APPLY_EDM
```

```
#////////////////////////////////////////////////////////////////////
#        Funcao EDM_BYTE
#////////////////////////////////////////////////////////////////////
edm_byte
#--------------------------------------------------------------------
# ALGORITMO CRC-16
# Traduzido e adaptado por Gustavo Luchine da função do PIC
# de T. Scott Dattalo acessível no sitio
# http://www.dattalo.com/technical/software/pic/crc16.asm
#--------------------------------------------------------------------
crc16              #entrada acontece através do registrador $a0

        AND     $t2, $zero, $t0     #Separa os dois octetos do byte corrente
        ADDI    $t2, $FF
        AND     $a1, $t2, $a0       #$a1 (octeto menos significativo)
        NOT     $t2
        AND     $a2, $t2, $a0       #$a2 (octeto mais significativo)
        SHIFT   $a2, $08

        AND     $t1, $zero, $t0     #Separa os dois octetos do CRC anterior
        ADDI    $t1, TEMP3
        LW      $a0, $t1, $zero     #$a0 agora contem o resultado anterior
        AND     $s1, $t2, $a0       #$s1 octeto mais significativo do CRC
        SHIFT   $s1, $08
        NOT     $t2
        AND     $s0, $t2, $a0       #$s0 octeto menos significativo do CRC

        ADD     $gp, $zero, $t2     #Ponteiro $gp>0 para controlar octetos

        ADD     $t0, $zero, $a2     #Copia octeto mais significativo em $t0

inicia_crc16
        XOR     $s2, $s0, $t0
        ADD     $s0, $zero, $s1
```

```
AND     $s3, $zero, $t0     #Limpa $s3(temp) p/ guardar o padrão do CRC
AND     $a0, $zero, $t0
ADDI    $a0, $01            #$a0 agora guarda a posição do bit -signif

ADD     $t0, $zero, $s2
AND     $t1, $a0, $t0       #Carry virtual
SHIFT   $t0, $01
LUI     $t2, $00
ADDI    $t2, $7F
AND     $t0, $t2, $t0

XOR     $s2, $t0, $s2

ADD     $a0, $zero, $s3     #$a0 guarda temp temporariamente
SHIFT   $s3, $01            #Rotaciona temp
BEQ     $t1, $zero, $3
AND     $t2, $zero, $t1
ADDI    $t2, $80
OR      $s3, $t2, $s3       #Seta bit 7 do temp
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $s3, $t2, $s3       #Desliga bit 7 do temp
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $a0       #Guarda em $t1 o carry do temp anterior

ADD     $t0, $zero, $s2
SHIFT   $t0, $01
BEQ     $t1, $zero, $3      #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $80
OR      $t0, $t2, $t0       #Seta bit 7
```

```
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $t0, $t2, $t0       #Desliga bit 7
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $s2       #Guarda em $t1 o carry do index anterior

ADD     $s1, $zero, $t0

ADD     $a0, $zero, $s3     #$a0 guarda temp temporariamente
SHIFT   $s3, $01            #Rotaciona temp
BEQ     $t1, $zero, $3      #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $80
OR      $s3, $t2, $s3       #Seta bit 7 do temp
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $7F
AND     $s3, $t2, $s3       #Desliga bit 7 do temp
LUI     $t2, $00
ADDI    $t2, $01
AND     $t1, $t2, $a0       #Guarda em $t1 o carry do temp anterior

ADD     $a0, $zero, $s2     #$a0 guarda index temporariamente
SHIFT   $s2, $81            #Rotaciona index para esquerda
BEQ     $t1, $zero, $3      #Verifica carry virtual
AND     $t2, $zero, $t1
ADDI    $t2, $01
OR      $s2, $t2, $s2       #Seta bit 0
BLT     $zero, $t1, $3
LUI     $t2, $00
ADDI    $t2, $FE
AND     $s2, $t2, $s2       #Desliga bit 0
```

```
LUI     $t2, $00
ADDI    $t2, $80
AND     $t1, $t2, $a0        #Guarda em $t1 o carry do index anterior

XOR     $s2, $t0, $s2

ADD     $t0, $zero, $s2      #Trocar nibbles do octeto menos significativo
LUI     $t2, $00
ADDI    $t2, $FF
AND     $t0, $t2, $t0        #Limpa byte mais significativo
SHIFT   $t0, $04             #Nibble mais significativo trocado
LUI     $t2, $F0
ADDI    $t2, $00
AND     $a0, $t2, $t0
SHIFT   $a0, $08             #Nibble menos significativo trocado
OR      $t0, $a0, $t0        #Nibbles trocados no octeto menos signif.

XOR     $s2, $t0, $s2

ADD     $t0, $zero, $s3

LUI     $t2, $00
ADDI    $t2, $02
AND     $a0, $t2, $s2
BEQ     $a0, $zero, $3       #Verifica bit 1 do index
LUI     $t2, $00
ADDI    $t2, $01
XOR     $t0, $t2, $t0        #Se o bit e' igual a 1, faca esta linha
XOR     $s0, $t0, $s0        #Se o bit e' igual a 0, faca esta linha

LUI     $t0, $00
ADDI    $t0, $C0
LUI     $t2, $00
ADDI    $t2, $02
```

```
        AND     $a0, $t2, $s2
        BEQ     $a0, $zero, $1      #Verifica bit 1 do index
        XOR     $s1, $t0, $s1       #Se o bit e' igual a 1, faca esta linha

        BEQ     $gp, $zero, $3      #Processa no CRC o octeto menos SIGNIFIC.
        AND     $gp, $zero, $t0     #Zera $gp para indicar 1o octeto calculado
        ADD     $t0, $zero, $a1     #Copia octeto menos significativo em $t0
        J       inicia_crc16

        LUI     $t2, $00            #AGRUPA OS OCTETOS PARA SALVAR
        ADDI    $t2, $FF
        AND     $s0, $t2, $s0       #Limpa octeto mais significativo de $s0
        AND     $s1, $t2, $s1       #Limpa octeto mais significativo de $s1
        SHIFT   $s1, $88            #Rotaciona $s1 8 bits para esquerda
        ADD     $a0, $s1, $s0       #Concatena ambos os octetos (16 bits)

        AND     $t2, $zero, $t0     #Salva resultado do byte processado
        ADDI    $t2, TEMP3          #Zera registrador temporário TEMP3
        SW      $a0, $t2, $zero

        J       $ra, $zero          #FIM DA EDM_BYTE

#//////////////////////////////////////////////////////////////////////////
#  FIM DO PROTOCOLO
#//////////////////////////////////////////////////////////////////////////
        END
```

F.2 - ARQUIVO SNAP16.INC

```
#///////////////////////////////////////////////////////////////////////////
#  MAPA DE MEMORIA
#///////////////////////////////////////////////////////////////////////////
        ORG   $0002

TEMP            EQU   $02          #Endereços para registros temporários
TEMP3           EQU   $03
TEMP7           EQU   $04

RUN_RX          EQU   $05   #RX_SNAP esta sendo executado
RUN_TX          EQU   $06   #TX_SNAP esta sendo executado
RUN_RTX         EQU   $07   #primeiro RX_SNAP, depois TX_SNAP
REJ             EQU   $08   #pacote rejeitado
RECEBIDO        EQU   $09   #pacote recebido pelo nodo de destino
SAB_PARA_DAB    EQU   $0A   #houve troca de endereços
ERR             EQU   $0B   #pacote recebido com erro
ACK_PKT_TX      EQU   $0C   #pede confirmação no pacote a ser TX
OCTSYNC         EQU   $0D   #Octeto de sincronismo transmitido ou recebido
TX_RF           EQU   $0E   #Transmissão por RF se TX_RF>0
TX_RF2          EQU   $0F   #Backup temporário do registrador TX_RF

HDB1            EQU   $10   #Cabeçalhos do pacote (HDB2 e HDB1)

DAB1            EQU   $11   #Endereço do nodo de destino
SAB1            EQU   $12   #Endereço do nodo de origem

DB8             EQU   $13   #Palavras de dados
DB7             EQU   $14
DB6             EQU   $15
DB5             EQU   $16
DB4             EQU   $17
```

```
DB3             EQU   $18
DB2             EQU   $19
DB1             EQU   $1A

CRC1            EQU   $1B     #Resultado do Método de detecção de erro

#///////////////////////////////////////////////////////////////////////////
#  DEFINICOES
#///////////////////////////////////////////////////////////////////////////
SYNC        EQU   $54     #01010100B   Octeto de sincronismo

HEADER2     EQU $A1       #10100001B   #configuração SNAP Modificado
HEADER1     EQU $49       #01001001B

SERup       EQU   $FF     #Endereço FFFAH de configuração da serial
SERlo       EQU   $FA
RXDup       EQU   $FF     #Endereço FFF7H de recepção da serial
RXDlo       EQU   $F7
TXDup       EQU   $FF     #Endereço FFF8H de transmissão da serial
TXDlo       EQU   $F8

RF1up       EQU   $FF     #Endereços FFFFH e FFFEH de config. da RF
RF1lo       EQU   $FF
RF2up       EQU   $FF
RF2lo       EQU   $FE
RXFup       EQU   $FF     #Endereço FFFBH de recepção da RF
RXFlo       EQU   $FB
TXFup       EQU   $FF     #Endereço FFFCH de transmissão da RF
TXFlo       EQU   $FC
#////////////////////////////////////////////////////////////////////////
#       Configura a VELOCIDADE DA TRANSMISSAO de acordo com o CLOCK:
#       250 MHz:   9600bps --> Tbit=104us --> TEMPO_BIT= 2604D ou 0A2CH
#       200 MHz:   9600bps --> Tbit=104us --> TEMPO_BIT= 2083D ou 0823H
#////////////////////////////////////////////////////////////////////////
```

```
TEMPO_BITup          EQU  $08         #Configurado para 200MHz
TEMPO_BITlo          EQU  $23
TEMPO_MEIO_BITup     EQU  $04    #Metade do valor de um bit
TEMPO_MEIO_BITlo     EQU  $11
      END            #FIM DO ARQUIVO DE MAPA DE MEMÓRIA
```

F.3 - ARQUIVO APPL16.INC

```
;//////////////////////////////////////////////////////////////////////
;      Definições de endereço dos nodos da rede
;//////////////////////////////////////////////////////////////////////
NODE1up        EQU  $00          #Estação de Campo
NODE1lo        EQU  $01

NODE2up        EQU  $00          #Nodo 1
NODE2lo        EQU  $02

NODE3up        EQU  $00          #Nodo 2
NODE3lo        EQU  $03

NODE4up        EQU  $00          #Nodo 3
NODE4lo        EQU  $04

MY_ADDR1up     EQU  NODE2up      #Endereço do Nodo 1 (0002H)
MY_ADDR1lo     EQU  NODE2lo
```