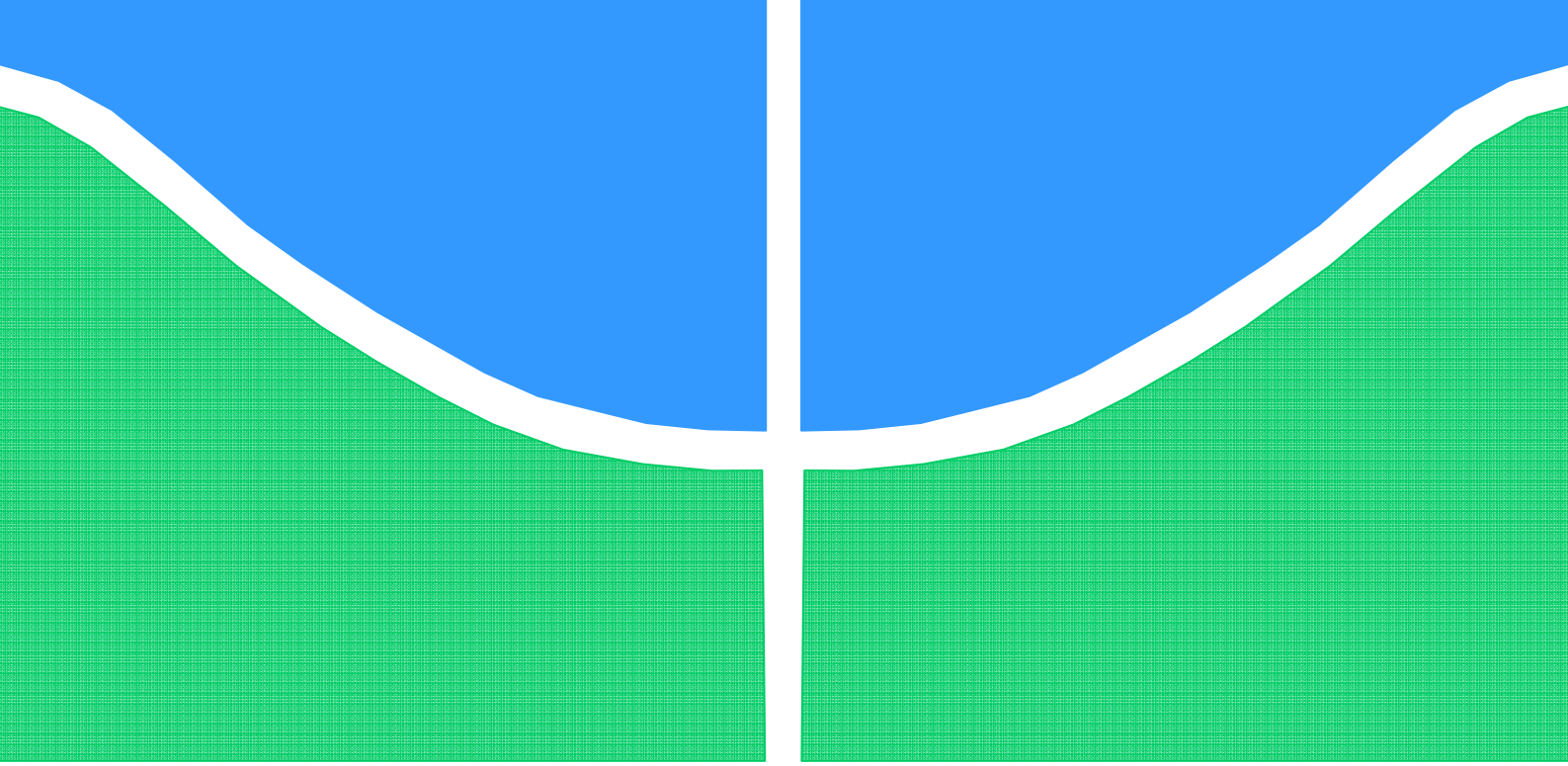

TRABALHO DE GRADUAÇÃO

INTEGRAÇÃO DE REDES DE SENSORES ZIGBEE PARA AUTOMAÇÃO PREDIAL UTILIZANDO MÓDULOS MESHBEAN

Davi Stoll Evangelista

Brasília, 10 de setembro de 2010

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA – ENGENHARIA ELÉTRICA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

INTEGRAÇÃO DE REDES DE SENSORES ZIGBEE PARA AUTOMAÇÃO PREDIAL UTILIZANDO MÓDULOS MESHBEAN

Davi Stoll Evangelista

Banca Examinadora

Prof. Adolfo Bauchspiess, UnB/ ENE (Orientador) _____

Prof. Geovany Araújo Borges, UnB/ ENE _____

Prof. Marco A. do Egito Coelho, UnB/ ENE _____

Brasília, Setembro de 2010

FICHA CATALOGRÁFICA

EVANGELISTA, DAVI STOLL

Integração de redes de sensores zigbee para automação predial utilizando módulos meshbean / Davi Stoll Evangelista. – – Distrito Federal, UnB / FT, 2010.

xii, 75 f. : il. ; 297 mm

Orientador: Adolfo Bauchspiess

Monografia (Graduação) – Universidade de Brasília, Faculdade de Tecnologia, FT, 2010.

Referências bibliográficas: f. 36

1. Controle de Temperatura 2. Integração de Redes 3. Redes Zigbee – Tese.

I. Universidade de Brasília, Faculdade de Tecnologia, ENE II. Título

REFERÊNCIA BIBLIOGRÁFICA

EVANGELISTA, D.S., (2010). : Integração de redes de sensores Zigbee para automação predial utilizando módulos Meshbean. Trabalho de Graduação em Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 75p.

CESSÃO DE DIREITOS

AUTOR: Davi Stoll Evangelista.

TÍTULO DO TRABALHO DE GRADUAÇÃO: Integração de redes de sensores zigbee para automação predial utilizando módulos meshbean.

GRAU: Engenheiro Eletricista ANO: 2010

_____

Davi Stoll Evangelista

Dedicatória

Dedico este trabalho a minha família, que sempre acreditou em meu potencial e à qual devo muito mais do que jamais poderia retribuir à altura.

Agradecimentos

Ao meu pai Francisco, por ter me proporcionado todas as oportunidades necessárias para que eu estivesse onde estou hoje.
A minha mãe Luciana, pela força e carinho que sempre demonstrou.
A minha tia Sueli, por tudo que fez por mim desde que cheguei à Brasília, em 2003.
Aos meus primos João, Débora e Tiago, que sempre estiveram presentes nos bons momentos e naqueles momentos nem tão bons assim, cada um com sua maneira de ser.
As minhas tias Liliana e Laura, por sempre terem um bom conselho a dar.
Aos meus avós, por sempre se preocuparem com meu bem estar.
A todos os outros membros da minha família, porque sem eles eu não seria quem eu sou.
Aos colegas de curso, que se tornaram grandes amigos no decorrer desses cinco anos. Em especial ao Aldo Farias, Edir Paulino, Rodrigo da Cunha Santos e Ricardo Dias.
Ao professor Adolfo, por sua paciência e confiança de que tudo daria certo, além da oportunidade para desenvolver esse projeto de conclusão de curso.
Aos colegas do LAVSI, LARA e GRAV, que ajudaram sempre que um novo problema se apresentava. Em especial ao Hamilson, Antônio Cássio, Urbano, Mariana, Rodrigo, Lucilia, Felipe Cavalcanti e Pedro H. Santana.
Por fim, a todos aqueles que ajudaram direta ou indiretamente na minha graduação. Muito obrigado!

RESUMO

A automação predial é um ramo da engenharia que cresce muito nos dias de hoje. Visando tornar os diversos sistemas presentes em uma edificação em sistemas inteligentes que irão consumir menos energia e desempenhar melhor suas funções (como são os casos dos sistemas de ar condicionado, de iluminação, de segurança, de controle de acesso, entre outros), vários projetos utilizando redes de sensores estão sendo desenvolvidos ao redor do mundo. O protocolo Zigbee foi especialmente criado para se adequar às necessidades de redes como essa: é capaz de comportar milhares de nós, oferecendo uma comunicação segura e de baixo consumo de energia. Dessa forma, utilizando-se dos conceitos de redes de sensores, do protocolo Zigbee e dos sistemas de ar condicionado convencional e híbrido, o presente trabalho tem como objetivo integrar duas redes Zigbee com controle embarcado liga-desliga para gerenciar as temperaturas de duas salas do LAVSI, localizadas no prédio SG-11, campus Darci Ribeiro - UnB.

ABSTRACT

The building automation is an engineering branch that grows a lot these days. Aiming to turn the various systems presents inside a building into intelligent systems that will consume less power and perform better (as in the cases of systems for air-conditioning, lighting, security, access control, among others), several projects using sensors networks are being developed all around the world. The Zigbee protocol was specially designed to suit the needs of networks such as these: it is capable of holding thousands of nodes, providing a secure communication with low power consumption. Thus, using the concepts of sensor networks, the Zigbee protocol and both conventional and hybrid air conditioning systems, the present project aims to integrate two Zigbee networks with embedded on-off control to maintain the temperatures of two rooms of LAVSI around a determinated setpoint. LAVSI is a laboratory located in the building SG-11, on campus Darci Ribeiro – UnB.

SUMÁRIO

1 INTRODUÇÃO 1
1.1 CONTEXTUALIZAÇÃO 1
1.2 MOTIVAÇÃO DO PROBLEMA 1
1.3 OBJETIVOS DO PROJETO 2
1.4 TRABALHOS ANTERIORES 2
1.5 APRESENTAÇÃO DO MANUSCRITO 2
2 FUNDAMENTAÇÃO TEÓRICA 3
2.1 INTRODUÇÃO 3
2.2 CONFORTO TÉRMICO 3
2.3 SISTEMAS DE CONTROLE 4
5.2.1 CONTROLE LIGA-DESLIGA 4
5.2.2 CONTROLE PID 5
5.2.3 CONTROLE FUZZY 5
2.4 REDES ZIGBEE 5
2.4.1 VISÃO GERAL 6
2.4.2 PADRÃO IEEE 802.15.4 7
2.4.3 PROTOCOLO DE COMUNICAÇÃO WIRELESS ZIGBEE 9
3 MÓDULOS E SOFTWARE UTILIZADOS 14
3.1 INTRODUÇÃO 14
3.2 MÓDULOS MESHBEAN 14
3.3 MÓDULO EXERNO PARA LM35 15
3.4 AR CONDICIONADO SPLIT 16
3.5 SISTEMA DE REFRIGERAÇÃO HÍBRIDO 17
3.6 CONTROLE POR SOFTWARE EMBARCADO 19
3.7 SOFTWARE SUPERVISÓRIO 20
4 DESCRIÇÃO DA REDE 24
4.1 INTRODUÇÃO 24
4.2 TOPOLOGIA DA REDE 24
4.3 APLICAÇÃO EMBARCADA 25
4.4 DISPOSITIVO COORDENADOR 26
4.5 DISPOSITIVO END DEVICE 27
4.6 DISPOSITIVO JUMPER 28
5 DESENVOLVIMENTO E ANÁLISE DOS RESULTADOS 30
5.1 ROTINA DE TESTES 30
5.2 ANÁLISE DOS RESULTADOS 30
5.2.1 PRIMEIRO TESTE 30
5.2.2 SEGUNDO TESTE 32
6 CONCLUSÕES E TRABALHOS FUTUROS 34
7 REFERÊNCIAS BIBLIOGRÁFICAS 36
ANEXO I 38

AI.1 SOFTWARE – PROJETOFINAL.C 38
AI.2 SOFTWARE – PROJETOFINAL.H 48
AI.3 SOFTWARE – COORDINATOR.C 52
AI.4 SOFTWARE – ENDDEVICE.C 59
AI.5 SOFTWARE – JUMPER.C 66

LISTA DE FIGURAS

Figura 1.1: Exemplo de automação predial [18] 1

Figura 2.1: Escala do índice PMV [9] 3
Figura 2.2: Relação entre os índices PMV x PPD [9] 4
Figura 2.3: Comparação entre as tecnologias wireless [1] 6
Figura 2.4: Modelo OSI [7] 7
Figura 2.5: Canais do IEEE 802.15.4 [1] 8
Figura 2.6: Modelo de camadas do IEEE 802.15.4 e Zigbee [1] 9
Figura 2.7: Redes Mesh roteiam pacotes automaticamente [1] 12
Figura 2.8: Mesh routing x tree routing [1] 12
Figura 2.9: Broadcast em uma rede [1] 12
Figura 2.10: O processo de descoberta da melhor rota [1] 13

Figura 3.1: Placa Meshbean2 [11] 14
Figura 3.2: Esquemático da placa externa 16
Figura 3.3: Circuito montado 16
Figura 3.4: Ar condicionado Split 17
Figura 3.5: Diagrama de força do sistema híbrido 17
Figura 3.6: Damper fechado 18
Figura 3.7: Ar condicionado Híbrido 19
Figura 3.8: Software supervisório 20
Figura 3.9: Bloco Obtenção dos dados 21
Figura 3.10: Bloco LAVSI 22
Figura 3.11: Bloco Sala de Reunião 22

Figura 4.1: Planta baixa do ambiente controlado 24
Figura 4.2: Diagrama de estados da aplicação topo 25
Figura 4.3: Diagrama de estados do coordenador 26
Figura 4.4: Diagrama de estados do *end device* 27
Figura 4.5: Diagrama de estados do *jumper* 28

Figura 5.1: Gráfico do desempenho do sistema - LAVSI 30
Figura 5.2: Gráfico do desempenho do sistema - Sala de Reunião 30
Figura 5.3: Gráfico do par sensor/atuador 1 - LAVSI 31
Figura 5.4: Gráfico do par sensor/atuador 2 - LAVSI 31
Figura 5.5: Gráfico do desempenho no 2° teste - LAVSI 32
Figura 5.6: Gráfico do desempenho no 2° teste - Sala de Reunião 32

LISTA DE TABELAS

Tabela 1: Métodos de roteamento do Zigbee [1] 11
Tabela 2: Valores dos DIP Switches e os nós da rede correspondente 26
Tabela 3: Temperaturas médias 31
Tabela 4: Consumo de energia dos sistemas 32
Tabela 5: Comparação entre os sistemas de controle 33

LISTA DE SÍMBOLOS

Subscritos

0x	Número em hexadecimal

Símbolos Latinos

G	Giga
M	Mega
k	Kilo

Unidades

bps	Bits por segundo
BTU/h	British Thermal Unit por hora
°C	Graus Celsius
Hz	Hertz
Ω	Omh

Siglas

ADC	Analog-to-Digital Converter
APS	Application Support Sublayer
AODV	Advanced Ad-hoc On-demand Distance Vectoring
ASHRAE	American Society of Heating, Refrigerating and Air-Conditioning Engineers
BACnet	Building Automation and Control Networks
BitCloud	Software embarcado desenvolvido pela empresa MeshNetics
Bluetooth	Protocolo de comunicação sem fio baseado no protocolo IEEE 802.15.1
BTT	Broadcast Transaction Table
CAV	Comissão Americana de Ventilação
COM	Communication
CSMA-CA	Carrier Sense Multiple Access With Collision Avoidance
DSSS	Direct-Sequence Spread Spectrum
FFD	Full-Function Device
Fuzzy	Estratégia de controle
GTS	Guaranteed Time Slots
I2C	Inter-Integrated Circuit
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
ISM	Industrial, Scientific and Medical
ISO	International organization for Standardization
LAVSI	Laboratório de Automação, Visão e Sistemas Inteligentes
LR-WPAN	Low Rate Wireless Personal Area Networks
MAC	Media Access Control ou camada de enlace do modelo OSI
MATLAB	Programa para manipulação de matrizes e computação numérica
MLME	MAC Layer Management Entity
MPDU	MAC Protocol Data Unit
NWK	Camada de rede do modelo OSI
O-QPSK	Offset Quadrature phase-shift keying
OSI	Open Systems Interconnection
PAN	Personnal Area Network
PHY	Camada física do modelo OSI
PID	Proporcional Integral Derivativo
PMV	Predicted Mean Vote
PPD	Predicted Percentage of Dissatisied

RFD	Reduced-Function Device
SIMULINK	Pacote gráfico para simulações multi-domínio presentente no MATLAB
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus
xBee	Dispostivo de comunicação Zigbee
Wi-Fi	Protocolo de comunicação sem fio baseado no IEEE 802.11
ZDO	*ZigBee Device Objects*
Zigbee	Protocolo de comunicação sem fio baseado no IEEE 802.15.4

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Inúmeros estudos de conforto térmico vêm sendo feitos desde 1916 - quando foram primeiramente propostos pela Comissão Americana de Ventilação (CAV) - para determinar as condições mais favoráveis e confortáveis no ambiente de trabalho com o intuito de intensificar as capacidades do trabalhador e torná-lo mais eficaz.

O uso consciente dos recursos naturais, de qualquer espécie, está em pauta desde meados dos anos 90 e deu espaço aos estudos de sistemas inteligentes nas pesquisas e projetos científicos ao redor do mundo desde então, visando amenizar os danos advindos do progresso do homem.

Com esse enfoque, redes de sensores sem fio começaram a ganhar espaço nos projetos de sistemas automatizados no início da década de 90. Surgiu na época uma aliança de empresas interessadas no tema – a Zigbee Alliance – cujo protocolo foi concluído em 2003 e é globalmente aceito desde então. É um protocolo de comunicação wireless desenvolvido especialmente para redes de sensores. Atualmente, as redes sem fio constituem um mercado em expansão que capta cada vez mais a atenção das empresas.

1.2 MOTIVAÇÃO DO PROBLEMA

Um sistema inteligente é aquele que gerencia determinada condição de maneira autônoma (sem interferência humana) para aperfeiçoar as condições do ambiente ao qual está inserido, proporcionando um uso eficiente dos recursos disponíveis. Um exemplo de sistema inteligente é mostrado na Figura 1.1. Os sistemas inteligentes de maior destaque comercial atualmente são aqueles que visam fazer algum tipo de gestão energética ou da segurança de uma residência/empresa. Entre os mais comuns, podem-se destacar os sistemas de controle. Muito se fala em um futuro onde as pessoas vão viver em ambientes completamente automatizados, e para que um dia cheguemos a esse estágio, se faz necessária a pesquisa em cada um dos ramos de interesse, no presente caso, o controle de temperatura.

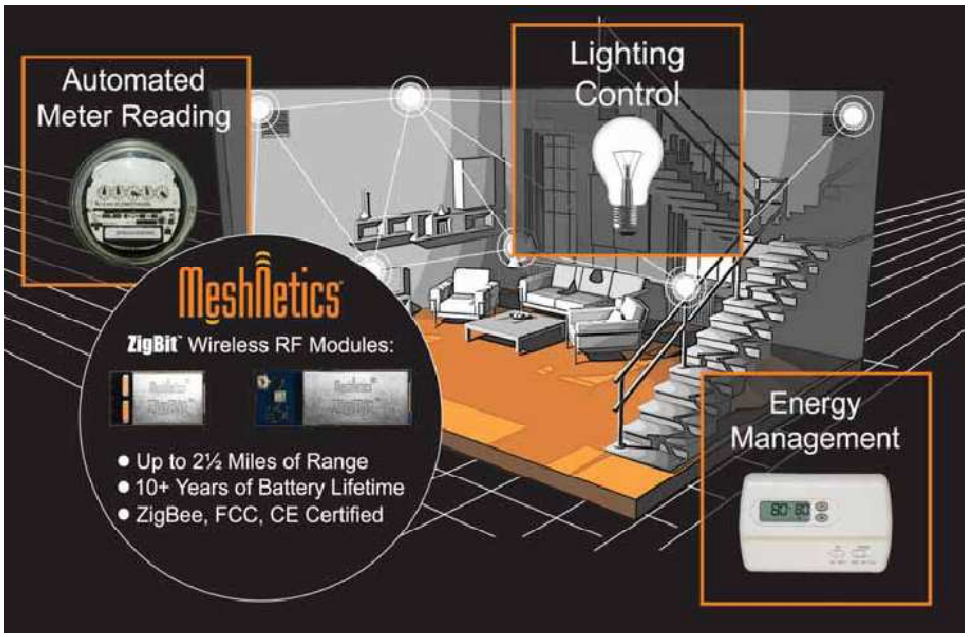

Figura 1.1: Exemplo de automação predial [18]

Este trabalho segue a linha dos projetos de sistemas inteligentes com uso de redes de sensores, visando atingir o conforto térmico dos usuários dos ambientes controlados.

1.3 OBJETIVOS DO PROJETO

O intuito do trabalho é integrar duas redes de sensores no laboratório LAVSI; fazer com que seja possível o controle de dois ambientes ao mesmo tempo, de uma maneira simples e eficaz, para que, eventualmente, seja possível expandir o controle de temperatura para todo o primeiro andar do prédio SG 11, localizado no campus Darci Ribeiro – UnB. Portanto, com a integração das redes de controle de temperatura, o presente projeto visa o conforto térmico das pessoas que trabalham no laboratório LAVSI, ao mesmo tempo em que permite uma redução no consumo de energia elétrica, culminando na redução de gastos.

1.4 TRABALHOS ANTERIORES

Há algum tempo o controle da temperatura de uma sala visando o conforto térmico e economia de energia é estudado no LAVSI. Dessa maneira, existe, pois, uma variedade de projetos realizados. No início, o objetivo principal era fazer o controle da temperatura da sala apenas usando o protocolo Zigbee e uma estratégia de controle liga-desliga. Depois, os temas ficaram mais diversos, com alguns projetos visando melhorar a estratégia de controle utilizada (fazendo um controle PID ou por lógica *Fuzzy)*, mudando o protocolo de comunicação para o BACNet (que atua sobre o Zigbee e é um protocolo para automação predial mais robusto), melhorando os dispositivos utilizados (passando das placas xBee para módulos com microprocessadores com mais memória para rodar as aplicações, como o ZigBit), fazendo o uso de um sistema de refrigeração mais completo (um sistema de refrigeração híbrido) e mais recentemente visando atender de uma maneira mais completa os parâmetros que determinam o conforto térmico ou então verificando a influência das janelas e portas no o sistema de controle para se determinar um modelo de função de transferência.

Com tantos projetos realizados, falta quem os integre. Todos eles visaram o controle em apenas um ambiente, e é nesse ponto que esse projeto tenta fazer a diferença. O primeiro passo é fazer a integração de duas redes distintas, para que depois seja possível expandir o conceito e fazer um controle em grande escala.

1.5 APRESENTAÇÃO DO MANUSCRITO

No capítulo 2 são revisados alguns conceitos importantes para o entendimento do projeto. Em seguida, no capítulo 3, apresenta-se uma descrição dos módulos e softwares utilizados para abordar o problema do controle da temperatura. O capítulo 4 visa descrever de maneira mais aprofundada os dispositivos da rede proposta, que foi a parte central do projeto. Os testes são abordados no capítulo 5, que também faz a análise dos resultados. Por fim, as conclusões e projeções são tema do capítulo 6, seguida pela bibliografia utilizada presente no capítulo 7. Para aqueles que visam entender mais a fundo os códigos e processos desenvolvidos, o Anexo I fornece os códigos-fonte em C de todos os dispositivos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 INTRODUÇÃO

Esse capítulo visa revisar alguns conceitos considerados importantes para o bom entendimento do problema e da solução apresentada nos próximos capítulos. Revisa-se primeiramente o conceito de conforto térmico, extremamente importante para entender a idéia central que serviu de inspiração para o desenvolvimento de um sistema de controle de temperatura. Em seguida, outra pequena revisão é feita sobre alguns métodos de controle para que o leitor possa entender as várias abordagens que podem ser utilizadas para melhorar a resposta do sistema controlado. Por fim, fala-se um pouco do protocolo Zigbee, afim de que os conceitos básicos de redes sejam revisados, incluindo alguns conceitos específicos para as redes de sensores sem fio.

2.2 CONFORTO TÉRMICO

O conceito de conforto térmico é usado para definir o estado que uma pessoa expressa quando está satisfeita com o ambiente que a cerca. Há muito tempo existem pesquisas para determinar os fatores que levam uma pessoa a se sentir termicamente confortável e, assim, ser capaz de desempenhar seu potencial nas atividades que realiza sem se distrair com coisas menores, como o calor que poderia estar sentindo.

O objetivo definido pela norma técnica ASHRAE 55 com relação ao conforto térmico dos usuários de um ambiente é *"determinar uma combinação de fatores pessoais e condições térmicas de ambientes internos de uma maneira que se torne aceitável a 80% dos ocupantes"* [10]. É nessa norma que o presente texto se baseia.

As variáveis que determinam o conforto térmico incluem fatores pessoais e fatores ambientais. Os fatores pessoais são divididos entre a atividade exercida e o vestuário utilizado, enquanto os fatores ambientais se dividem em temperatura do ar, temperatura média radiante, umidade relativa do ar, na velocidade do vento e na temperatura da superfície do chão. Além disso, ainda são consideradas as trocas de calor entre o ambiente e as pessoas.

Na ISO 7730, propõe-se a uma metodologia para o cálculo do PMV (*Predicted Mean Vote*) e do PPD (*Predicted Percentage of Dissatisied*) do sistema [9]. O PMV é uma escala quantitativa da sensação de calor/frio do ambiente analisado, enquanto o PPD é uma escala que apresenta a porcentagem das pessoas insatisfeitas com as situações do ambiente, para um determinado valor de PMV. O PPD, portanto, é uma função do PMV do sistema. A Figura 2.1 mostra a escala do PMV e a Figura 2.2 mostra graficamente o PPD. Os cálculos do PMV e do PPD podem ser vistos em [9].

Figura 2.1: Escala do índice PMV [9]

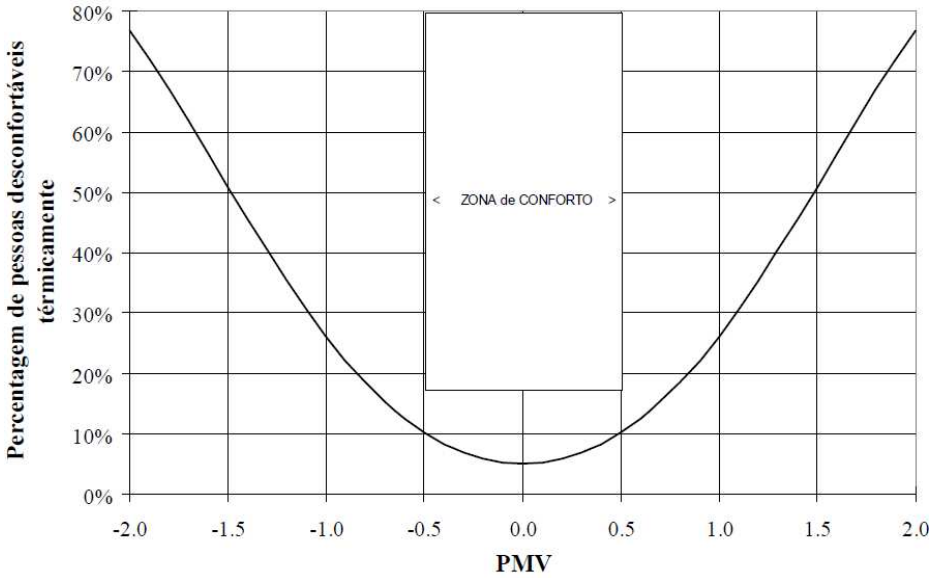

Figura 2.2: Relação entre os índices PMV x PPD [9]

A proposta do projeto é alcançar o conforto térmico por intermédio do controle da temperatura dos ambientes. A maior parte dos variáveis relacionadas ao PMV são muito difíceis de serem controladas e fogem à proposta inicial do presente trabalho. Mesmo dispondo de um sistema de refrigeração híbrido em um dos dois ambientes, o que possibilitaria o controle também da umidade relativa do ar, preferiu-se trabalhar com um hardware mais simples, que não fazia todas as medidas do índice e, assim, não tornar o projeto mais complexo do que o necessário. Mais sobre esse assunto será visto no capítulo 4, onde são descritos os equipamentos utilizados.

2.3 SISTEMAS DE CONTROLE

A abordagem do problema do controle da temperatura de um ambiente pode ser resolvida de várias formas, cada uma com suas vantagens e desvantagens tanto em relação ao desenvolvimento do projeto quanto em relação ao resultado desejado. O método mais simples de ser colocado em prática é o liga-desliga, aquele no qual o sistema liga quando uma determinada condição se torna verdadeira e desliga no momento em que outra condição é atingida. Em seguida, para uma estratégia de controle mais sofisticada, usa-se o controle proporcional, derivativo, integral ou uma combinação deles, que melhora consideravelmente a resposta do sistema, mas é mais complicado de ser colocado em prática e requer um estudo do sistema a ser controlado. Outro tipo de controle muito utilizado é o chamado *Fuzzy*, que facilita o projeto de controladores em sistemas que necessitem de um projeto matemático. A seguir fazemos uma revisão de cada uma dessas estratégias de controle.

5.2.1 CONTROLE LIGA-DESLIGA

Como mencionado, fazer um sistema de controle liga-desliga é bastante fácil. O controlador tem como função ligar assim que uma condição atingir um valor máximo e desligar quando essa condição atinge um valor mínimo. No caso do projeto de controle liga-desliga visando manter a temperatura de um ambiente relativamente constante, a variável de interesse é a própria temperatura do ambiente e os limites são definidos pela aplicação, para atender aos requisitos propostos.

Um sistema que queira manter a temperatura de um escritório comercial sempre em torno de 24°C com histerese de 1°C poderia ligar o sistema de ar-condicionado da sala quando a temperatura estivesse em 24,5°C e desligá-lo quando a temperatura chegasse em 23,5°C. É um sistema simples, bastante utilizado nos mais variados sistemas de refrigeração e consegue ser eficiente em fazer com que a temperatura acompanhe o nível de referência para uma grande variedade de condições ambientais.

O ponto negativo fica por parte do consumo de energia. Essa é uma estratégia de controle que não possui como co-objetivo economizar energia, apenas manter a temperatura próxima da referência. Não

que o sistema como um todo não economize energia, o que ocorre é que o sistema não é muito eficaz nesse aspecto.

O controle em liga-desliga foi o escolhido para a abordagem do controle da temperatura nos ambientes propostos no presente projeto. Isso se deve ao fato de que o real objetivo não é fazer o melhor controle possível, mas sim a integração de diversas redes Zigbee.

5.2.2 CONTROLE PID

Controladores PID são compostos por uma parte proporcional, uma parte integral e uma parte derivativa. Um controlador proporcional apenas introduz um ganho no sistema e diminui o erro em regime, desde que o ganho não seja grande muito alto, o que pode tornar o sistema instável. Um controlador integral é usado para anular o erro em regime permanente e um controlador derivativo é usado para reduzir o erro da resposta transitória. Quando bem projetado, o controle PID é aquele que apresenta a melhor resposta do sistema para as especificações, pois trabalha com o erro entre a variável controlada e a resposta atual do sistema e visa corrigi-lo.

O grande problema em uma abordagem PID é o modelo matemático do sistema. Se o modelo não for preciso, a resposta do sistema ficará comprometida. O modelo do sistema controlado nem sempre é fácil de ser encontrado e, no caso do controle de temperatura, existem muitas variáveis difíceis de serem medidas por causa das trocas de calor que acontecem entre os elementos do ambiente. O projeto muitas vezes acaba exigindo cálculos muito dispendiosos e de alto grau de complexidade, com modelos de ordem elevada.

É uma estratégia de controle muito utilizada em sistemas de automação que precisam atender a especificações da forma mais completa possível. Para sistemas térmicos, nos quais alguma variação na resposta não gera grandes problemas, não é muito utilizado.

5.2.3 CONTROLE FUZZY

O controlador *fuzzy* foi pensado para melhorar a resposta do sistema, mas sem exigir um modelo matemático muito detalhado. Ao contrário das lógicas convencionais, que trabalham com condições que são verdadeiras ou falsas, a lógica *fuzzy* é mais voltada para variáveis que podem ser parcialmente verdadeiras. Além disso, a solução do processo pode ser entendida de maneira mais simples por um operador humano.

Para dar um exemplo, quando se pensa em fazer o controle da temperatura de um ambiente a partir da lógica *fuzzy*, modela-se a temperatura não como um número, mas sim como uma sensação. Quando a temperatura está em 22°C, muitas pessoas podem se sentir confortáveis, enquanto muitas outras vão sentir um pouco de frio. A partir de análises como essa, um sistema de *fuzzyficação* é montado, transformando variáveis físicas em variáveis *fuzzyficadas* e a solução de controle é feita. No final do processo, ocorre o que se chama de *desfuzzyficação* para que o controlador possa gerar a ação de controle no sistema controlado.

É uma modelagem que gera um resultado não-ótimo, ou seja, não será o melhor resultado possível. Quando o projetista decide por um controlador *fuzzy* deve levar em conta a precisão do controle e a necessidade de uma modelagem mais precisa e decidir pela melhor maneira.

2.4 REDES ZIGBEE

As redes de sensores sem fio geralmente utilizam o protocolo Zigbee por causa das vantagens que este proporciona. É um protocolo que foi desenvolvido especialmente para redes de sensores e se adéqua às suas necessidades. A seguir mostra-se um breve retrospecto para que depois sejam discutidas as características de interesse dos protocolos usados na comunicação, o protocolo IEEE 802.15.4 e o protocolo Zigbee.

2.4.1 VISÃO GERAL

O padrão Zigbee define um conjunto de protocolos de comunicação para redes sem-fio de curto alcance e baixa taxa de tráfego de dados. As redes do tipo Zigbee começaram a ser projetadas em meados de 1998 quando muitas aplicações que outrora eram desenvolvidas sob a ótica *Wi-Fi* e Bluetooth começaram a se tornar inviáveis por questões como gerenciamento de energia, ineficiência de banda alocada, complexidade de protocolos, etc.

Na época, o principal foco das redes *wireless Wi-Fi* era de aumentar a taxa de transmissão de dados e desenvolver um protocolo seguro que permitisse acesso à internet para dispositivos fixos e móveis. Em redes Bluetooth, o objetivo era a criação de uma rede pequena, com limitação dos dispositivos e das taxas de dados. Dessa maneira, as aplicações para redes de sensores estavam fadadas a serem muito mais complexas e caras se utilizassem o padrão *Wi-Fi* ou limitadas demais caso fosse escolhido o padrão Bluetooth. Uma comparação entre essas tecnologias é mostrada na Figura 2.3, que destaca as diferenças entre as taxas de transmissão de dados e o alcance de cada uma.

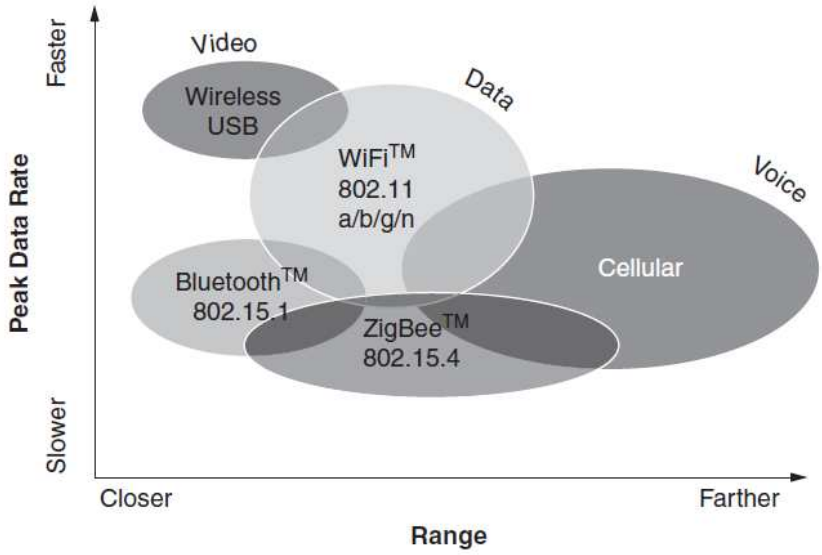

Figura 2.3: Comparação entre as tecnologias wireless [1]

A Zigbee Alliance surgiu em meados de 1997 do esforço conjunto de oito empresas para criar um padrão para redes sem-fio que possuam as características inerentes ao paradigma Zigbee, de ser um sistema *wireless* altamente confiável e seguro, capaz de atingir baixíssimos consumos de energia por parte dos dispositivos presentes na rede, que tenha uma alta relação custo-benefício e que seja um padrão de comunicação global aberto para o uso de qualquer pessoa ou instituição interessada. É um padrão que está voltado principalmente para sistemas que utilizem bateria, com baixa taxa de tráfego de dados, baixo custo, baixa complexidade e baixas potências de transmissão.

Pode-se explicar a função do protocolo Zigbee a partir do modelo OSI (que define, dada uma rede genérica, camadas para separar as funções de cada parte da rede). O padrão Zigbee define as camadas de aplicação e de rede, responsáveis por estabelecer circuitos lógicos e de roteamento entre dois pontos e da gerência da interface das aplicações específicas, respectivamente. Além disso, o Zigbee ainda define os métodos de segurança do protocolo como um todo. As camadas de mais baixo nível, leia-se as camada física e de enlace (responsáveis por transmitir os dados por meio de um sinal eletromagnético e de controlar a ordem de acessos ao meio, respectivamente), ficam a cargo do padrão 802.15.4, homologado pelo IEEE em 2003, que define redes sem-fio de baixa velocidade e baixa região de cobertura conhecidas como LR-WPANs.

Os dispositivos Zigbee foram concebidos para atuar na faixa de freqüências ISM, que não requer licença para uso, atuando na freqüência de 2.4 GHz, dividindo o espectro com outras tecnologias, como a própria *Wi-Fi*. A máxima taxa de transmissão de dados é de 250 kbps, que apesar de baixa

quando comparada a outras tecnologias sem fio presentes na mesma faixa de freqüência, se mostra bastante razoável às aplicações pretendidas.

O Zigbee foi concebido para prover interoperabilidade entre sistemas desenvolvidos por diferentes fabricantes, desde que esses sigam corretamente o padrão. Esse fato traz uma grande vantagem para esse tipo de rede. Por exemplo, supondo que um ambiente possua um nó Zigbee em uma lâmpada e outro nó Zigbee na porta, é possível que eles se comuniquem, mesmo que os eles tenham sido fabricados por diferentes empresas e possuam diferentes funções, desde que utilizem o padrão Zigbee para tal.

As redes que são baseadas no protocolo 802.15.4 são constituídas por dois tipos de dispositivos: os FFDs – full-function devices – e os RFDs – reduced-function devices. O Zigbee se utiliza desse conceito para que um dispositivo não acumule funções às quais não vai utilizar, melhorando ainda mais as características de potência e complexidade do projeto. Dispositivos FFDs são capazes de executar todas as funções especificadas no padrão 802.15.4, podendo assumir qualquer um dos papéis possíveis na rede. Em geral esse tipo de dispositivo é relacionado à coordenação da rede, podendo se comunicar com qualquer nó, mesmo quando se encontra inativo. Dispositivos RFDs são dispositivos relativamente simples que só podem se comunicar com FFDs e nunca poderão assumir o papel de coordenador da rede.

É muito importante saber que o padrão 802.15.4 e o padrão Zigbee não são a mesma coisa, apesar de que em muitos momentos sejam tratados tal qual fossem. Nos próximos tópicos apresenta-se o conceito de cada um em separado para que não reste nenhuma dúvida.

2.4.2 PADRÃO IEEE 802.15.4

Antes de aprofundar a discussão sobre o IEEE 802.15.4, é importante entender o que cada número significa. Todos os padrões IEEE que começam com 802.x.x definem as normas de uma rede. O leitor provavelmente já ouviu falar na norma para redes Ethernet (802.3), *Wi-Fi* (802.11) e/ou Bluetooth (802.15.1). Em seguida, o número 15 (802.15.4) representa redes de uso pessoal, as chamadas PANs. Note que o padrão Bluetooth também é definido para uso pessoal. Por fim, o número 4 (802.15.4) representa sistemas de baixas taxas de transmissão com grande vida útil das baterias, e nesse quesito já podemos diferenciá-lo do Bluetooth, que não tem essa preocupação.

O padrão IEEE 802.15.4 foi criado com o objetivo de especificar os protocolos das camadas de mais baixa ordem de uma rede. Esses protocolos são relativos à camada física e a camada de enlace. A Figura 2.4 mostra o modelo OSI de camadas de uma rede para melhor entendimento do leitor.

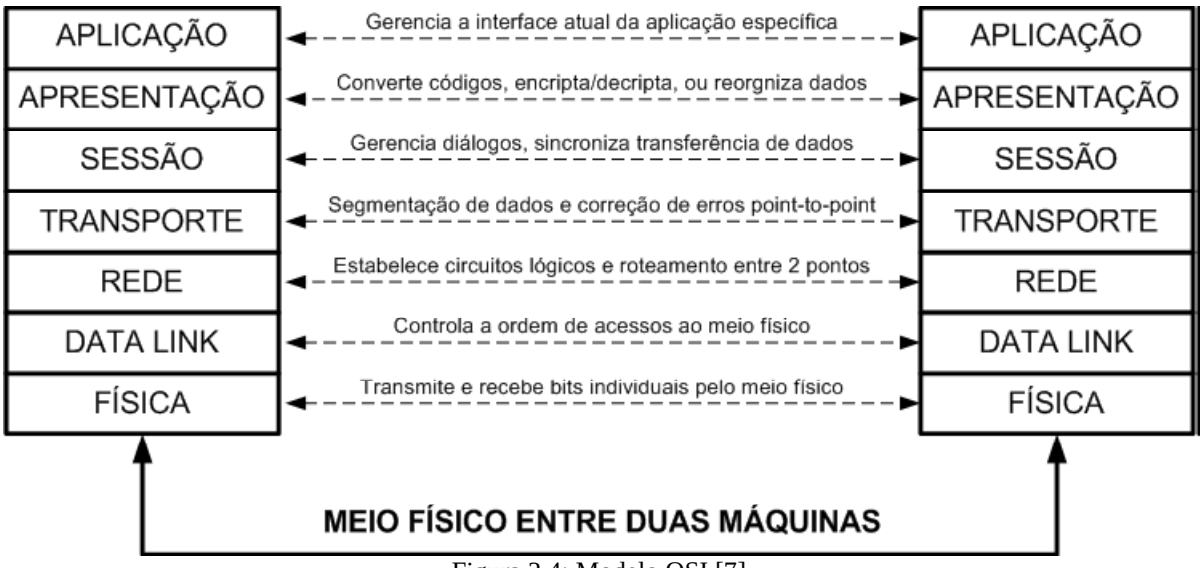

Figura 2.4: Modelo OSI [7]

O padrão não só especifica as funções do protocolo PHY e as interfaces com a camada MAC (camadas FÍSICA e DATA LINK da Figura 2.4, respectivamente), como também define os mínimos requisitos de hardware, como sensibilidade do receptor e potência de transmissão.

A camada PHY é a mais próxima ao hardware. Ela controla e se comunica com os transmissores dos dispositivos. Ela também é associada ao processo de seleção de canais para transmissão. Em geral, a PHY é responsável pela ativação e desativação dos transceptores, transmissão e recepção de dados, seleção do canal em que o transceptor irá operar, detecção de ocupação do canal, etc.

Os canais presentes no protocolo 802.15.4 não são nada mais do que porções do espectro de rádio-freqüências. São definidos 16 canais para atuarem na freqüência de 2.4GHz, numerados de 11 até 26, e cada um deles separados por 5MHz dos outros adjacentes. A Figura 2.5 mostra uma representação desses canais. Existem dois pontos interessantes a serem citados. Primeiro, os canais são *half-duplex*, ou seja, ou o dispositivo escuta o meio, ou ele transmite informações, nunca as duas coisas ao mesmo tempo. Segundo, o dispositivo só pode acessar um canal por vez, ou seja, quando o dispositivo está conectado em um determinado canal, ele não será capaz de perceber nada do que ocorre nos outros 15 canais.

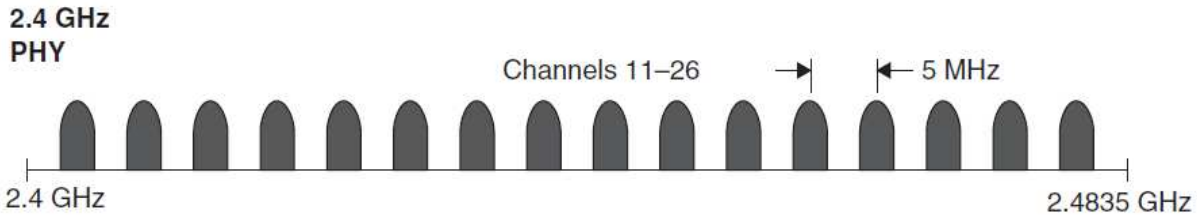

Figura 2.5: Canais do IEEE 802.15.4 [1]

As aplicações desenvolvidas não fazem uso da técnica de transmissão em vários canais, como ocorre no Bluetooth, por exemplo. Redes Zigbee costumam atuam em canais fixos. Quando uma aplicação fica saltando de canal em canal para transmitir seus dados, tem-se como objetivo aumentar a confiabilidade do sistema na presença de ruído, o que não é necessário porque as modulações em O-QPSK e o DSSS já são robustas o suficiente para garantir que os dados transmitidos na rede não sejam afetados pelo ruído, em outras palavras, a perda de pacotes é extremamente baixa, mesmo quando o meio está ocupado por outras transmissões em freqüências próximas.

A camada MAC fornece dois tipos de serviços: o *MAC data service* e o *MAC management service interfacing*. O primeiro é relacionado à transmissão e recepção dos MPDUs através da camada de serviço da PHY. Já a segunda, fica a cargo da MLME, que interage com a interface relacionada na camada superior a MAC. As funções características principais da camada MAC são o gerenciamento de *beacons*, acesso à canal, gerenciamento de GTS, validação de quadro, entrega de quadro de reconhecimento, associação e dissociação. Além disso, a camada MAC fornece interfaces para a implementação de mecanismos adequados de segurança. É responsável por criar uma rede, alocar os canais e transmitir os dados entre dois nós da rede (desde que sejam adjacentes) de forma razoavelmente confiável.

Nem todas as características do protocolo da camada MAC estão presentes no modelo Zigbee. Isso se deve à proposta de desenvolver dispositivos mais simples e ocupem pouca memória RAM e Flash. Por exemplo, das funcionalidades que tratam do *beaconings* da rede, somente o CSMA-CA é utilizado, para prevenir dois nós da rede de se comuniquem ao mesmo tempo, possibilitando que a rede seja assíncrona (os dispositivos podem transmitir a qualquer momento). As outras especificações são desconsideradas na implementação do modelo e, dessa forma, serão desconsideradas também nesse texto por não fazerem parte do escopo proposto. A maior parte das funcionalidades propostas pelo 802.15.4 são levemente alteradas nas próprias camadas do modelo da rede Zigbee, como é o que ocorre com o modelo de segurança utilizado, que desconsidera o que é proposto pelo padrão 802.15.4.

2.4.3 PROTOCOLO DE COMUNICAÇÃO WIRELESS ZIGBEE

2.4.3.1 MODELO EM CAMADAS

O protocolo Zigbee gerencia o funcionamento das camadas mais elevadas do modelo OSI e ainda adiciona algumas outras camadas que não estão presentes nele. A Figura 2.6 ilustra essa questão. Resumidamente, a camada de rede, NWR, é responsável por fazer a comunicação entre dois nós da rede, mesmo que estes nós não possam se comunicar diretamente. A camada APS é responsável por gerencial alguns processos entre a aplicação e a camada de rede, enquanto a camada ZDO é responsável por definições gerais do padrão Zigbee. Como foi dito, o Zigbee também implementa um processo de segurança próprio, representado pela camada de *Security Service Provider*. A camada mais alta é a *Application Framework*, que é definida pela aplicação utilizada.

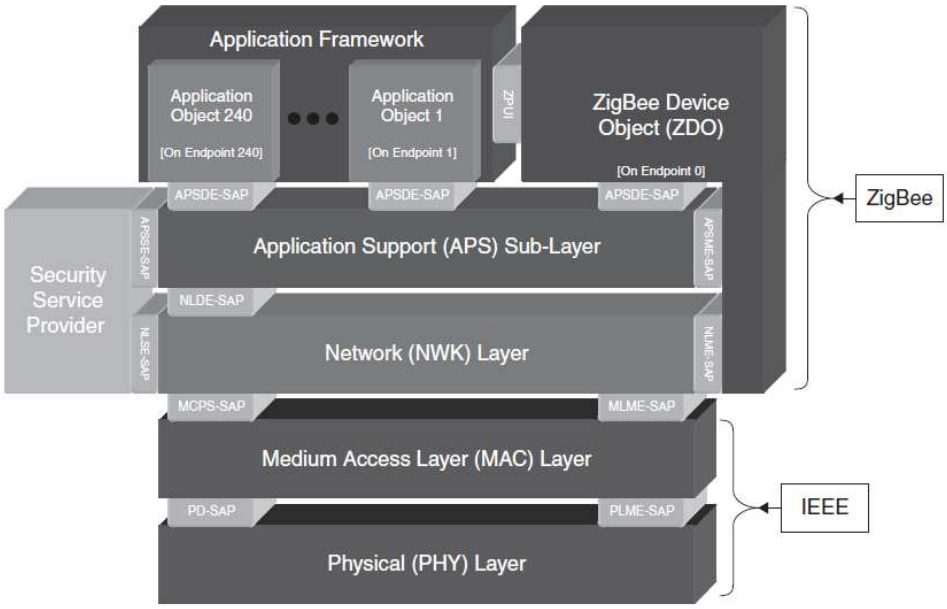

Figura 2.6: Modelo de camadas do IEEE 802.15.4 e Zigbee [1]

2.4.3.2 DISPOSITIVOS ZIGBEE E SUAS FUNÇÕES

Se faz necessário o conhecimento dos três tipos de dispositivos que um nó na rede podem ser, antes de se falar de maneira mais aprofundada sobre a rede Zigbee. O mais importante do ponto de vista da rede é o coordenador. Dispositivos finais (*end devices*) são aqueles que desempenham as funções de sensores ou atuadores da rede. Dispositivos roteadores fazem com que a rede possa ser expandida do ponto de vista físico, possibilitando o roteamento de pacotes e mensagens entre nós da rede que não necessariamente possam se comunicar diretamente.

Os dispositivos coordenadores são capazes de formar uma nova rede. Com a rede formada, o coordenador assume a função de agente central, concentrando as informações de interesse, que obviamente dependem da aplicação desejada.

Dispositivos finais são responsáveis por encontrar e se juntar a uma rede, perguntar se existem pacotes que foram enviados enquanto eles estavam no modo de economia de bateria, chamado *sleep*, procurar um novo enlace se o anterior não estiver mais disponível e ficar dormindo (em modo *sleep*) na maior parte do tempo para economizar bateria.

Dispositivos roteadores também são responsáveis por encontrar e se a uma determinada rede, mas possuem mais funções, como perpetuar *broadcasts*, descobrir e manter rotas, permitir que novos dispositivos se juntem a rede e salvar pacotes destinados para os dispositivos finais que estão dormindo.

Dispositivos coordenadores e roteadores são projetados para serem alimentados em tempo integral e ficam sempre escutando o canal. Dispositivos finais são projetados para passar a maior parte do tempo no modo *sleep* e transmitir rapidamente antes de voltarem para o modo de economia novamente, maximizando assim o tempo de vida da bateria.

Para que fique bem claro, dispositivos coordenadores são os únicos capazes de formar uma nova rede. Dispositivos finais e roteadores podem apenas se conectar a redes existentes e, por esse motivo, podem existir nós que desempenham mais de uma função.

Um coordenador forma a rede basicamente determinando um endereço único para identificá-la, chamado de PAN ID, e escolhendo um dos 16 canais possíveis. Na maioria das vezes, o coordenador atua como um roteador a partir desse ponto. Para determinar qual canal utilizar, o coordenador verifica qual canal está mais favorável (utilizando um *energy scan*, funcionalidade do IEEE 802.15.4) e para determinar qual PAN ID o dispositivo verifica quais redes estão ativas. Se a rede pretendida já está presente no meio, o coordenador pode tomar uma das seguintes decisões: ele se junta a ela, desistindo de criar uma nova rede, ele cria uma rede com outro endereço ou ele força a criação de uma rede com o mesmo endereço, o que é altamente desencorajado para evitar problemas.

Conectar-se a uma rede é um processo que pode ser resumido a descobrir quais são as redes e os nós presentes na vizinhança e escolher a qual delas se juntar. Ao iniciar a comunicação, um nó informa seu endereço MAC (64-bits) e a ele é atribuído um endereço único (naquela rede) de 16-bits, chamado de endereço de rede (NwkAddr), para o nó se comunique com outros nós pela rede.

Tipicamente, ao se juntar a uma rede, um dispositivo final procura determinada aplicação e, se não a encontra, sai da rede e tenta em outra. Isso acontece porque o processo de *beacon* contém apenas informações da rede (PAD ID, PAN ID estendido, *join enable*, quantidade de roteadores e dispositivos finais suportados) e não das aplicações que estão acontecendo dentro dela.

O que ocorre de fato é que roteadores e dispositivos finais juntam-se a um nó específico, e não a rede em geral. O nó que se junta é chamado de filho e o que recebe a solicitação é chamado de pai. Assim, vemos que os dispositivos finais serão sempre filhos de algum outro dispositivo na rede, enquanto os roteadores serão pais e filhos de outros nós.

A relação pai/filho não se aplica ao processo de roteamento de mensagens, chamado de *mesh routing*. Qualquer roteador pode redirecionar a mensagem ao seu destino, desde que ela possua o mesmo PAN ID e esteja no mesmo canal. Essa característica garante que a comunicação será satisfatória mesmo se alguns dos roteadores se desconecte, e se não fosse assim a comunicação entre alguns elementos da rede seria perdida com a saída de um rotador. Essa relação é especifica para designar a qual nó o dispositivo atualmente é pai/filho um do outro. Em compensação, sempre que um dispositivo final deseja mandar uma mensagem para outro dispositivo na rede, essa mensagem primeiramente passa pelo roteador/coordenador pai. Quando um dispositivo filho não encontra mais seu pai, seja porque o canal está com muita interferência, seja porque o dispositivo pai não está mais presente, automaticamente esse dispositivo se encontra fora da rede e deve fazer o processo de se re-conectar, encontrando um novo pai.

O processo de se re-conectar assume que um determinado nó já tenha conhecimento das características da rede, já citadas. O próprio dispositivo decide quando está órfão – devido ao número de tentativas de se comunicar com o dispositivo pai – num processo depende da aplicação.

Quando um nó deseja se re-conectar, ele manda um *beacon request* e decide entre os dispositivos que retornaram o pedido qual será seu novo pai. A única ressalva a ser feita é se o nó pretendido tem capacidade para mais um dispositivo final, por causa de uma possível limitação de memória. O novo pai deve estar presente na mesma rede anterior por razões óbvias (mesmo PAN ID). Ao se re-conectar, o dispositivo recebe um novo endereço de rede e informa a todos os nós da rede que o endereço mudou. Essa última parte é especialmente importante para preservar as amarras que foram feitas pela rede e evitar que novos processos de descobrimento de rota sejam feitos.

Outro tipo de re-conexão acontece quando ocorre uma falha na alimentação e todos os módulos se desligam de uma vez. Ao se religarem, os dispositivos simplesmente continuam a mandar e receber mensagens na mesma PAN ID que estavam trabalhando antes da falha, com os mesmo endereços. Esse processo é chamado de re-conexão silenciosa, ou *silent rejoin*, e ocorre quando existe algum meio dos dispositivos salvarem informações em componentes de memória não-voláteis, como memórias flash. Também pode ser usado quando a rede decide trocar de canal. Na verdade, a rede Zigbee, ao contrário dos outros modelos de rede, não necessita que um dispositivo esteja periodicamente em contato para que seu estado seja verificado. A atualização dos estados dos dispositivos é feita apenas quando alguma comunicação os envolve diretamente, caso contrário assume-se que eles ainda se encontram ativos.

2.4.3.3 ENDEREÇAMENTO E ROTEAMENTO DE PACOTES

Assim que um nó entra na rede, um endereço de rede deve ser destinado a ele. Esse processo é feito de duas formas: estocástica ou Cskip. Quando um nó decide qual endereço quer ter, de forma determinística ou aleatória, ele se enquadra no processo estocástico e deve informar à rede que seu endereço será aquele que foi escolhido, apenas para saber se algum outro nó já o utiliza. Quando a rede determina o endereço do nó através de uma série de cálculos, que surgem do modelo da rede em uma árvore simétrica (que será explicado mais adiante), o processo se enquadra no endereçamento em Cskip, cuja abordagem não é aprofundada além desse ponto por não ser um tema relevante ao que foi proposto no projeto.

De posse de um endereço de rede, o nó se encontra conectado à rede e pode começar a transmitir seus pacotes. Para tal, ele se utiliza de uma das diferentes formas de enviar uma mensagem pela rede Zigbee. Ao todo, são quatro as formas diferentes para que um nó se comunique, listadas abaixo:

- *Broadcast* (de um nó para vários nós);
- *Mesh routing: unicast* de um nó para outro;
- *Tree routing: unicast* de um nó para outro;
- *Source routing*: *unicast* de um nó para outro.

O três primeiros são os mais utilizados. A Tabela **1**: Métodos de roteamento do Zigbee [1]Tabela 1 mostra as vantagens e desvantagens de cada um:

Tabela 1: Métodos de roteamento do Zigbee [1]

	Broadcast	Mesh Routing	Tree Routing
Quantidade de saltos	Até 30 saltos	Até 30 saltos	Até 10 saltos
Múltiplos destinos	SIM	NÃO	
Um-a-um	NÃO	SIM	SIM
Espectralmente eficiente	NÃO	SIM	SIM
Confirmado	NÃO	SIM	SIM

Quando se utiliza um *broadcast*, possibilita-se que um nó alcance vários outros nós com apenas um *data request*. É um método que não recebe confirmação, logo não se tem certeza se a mensagem foi recebida ou não por todos os nós da rede, e é um processo que utiliza muito os recursos do canal.

Mesh routing é um método bem mais eficiente espectralmente, em termos de memória e de tempo. Isso acontece porque a rota já está definida e a comunicação visa entregar a mensagem de um ponto a outro. A recepção dos pacotes que são mandados pela rede são confirmados. A comunicação é feita de nó em nó até que a mensagem chegue ao destino. É o componente mais importante em uma rede Zigbee, pois é ele quem leva o sinal de um nó a outro quando eles não podem se comunicar diretamente. Uma rota do nó de origem ao nó de destino é determinada e a mensagem é encaminhada com sucesso. Caso ocorra um problema e um nó saia da rede, ou um bloqueio é colocado entre dois nós, a rede é capaz de descobrir uma nova rota sozinha, como mostrado na Figura 2.7.

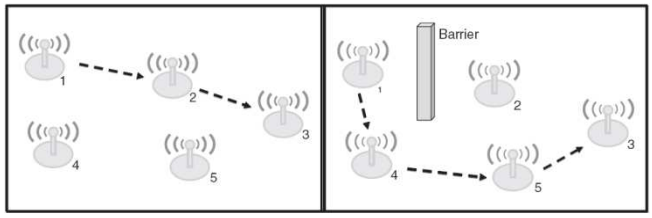

Figura 2.7: Redes Mesh roteiam pacotes automaticamente [1]

Por fim, *tree routing* é um método semelhante ao anterior, mas se utiliza do esquema de endereçamento *Cskip* que foi discutido brevemente. Com o endereçamento dos nós supondo-se uma rede simétrica, o roteamento fica mais fácil e mais eficiente em termos de memória e tempo quando comparado aos processos anteriores. A única desvantagem é que nesse método não pode haver quebra nos enlaces de nenhum pai/filho na rede, sob pena de não haver recuperação. Por isso, *mesh routing* é definido como roteamento padrão das redes Zigbee para entrega de mensagens.

A Figura 2.8 mostra uma mesma rede endereçada em Cskip e estocasticamente, para que fique clara a diferença entre *mesh routing* e *tree routing*. As linhas cheias representam as rotas no esquema em *tree routing*, enquanto as linhas tracejadas somadas às linhas cheias são a representação do esquema em *mesh routing*. A diferença é que todos os caminhos da rede em árvore são otimizados, enquanto na rede comum os caminhos em uso talvez não sejam os mais eficazes, mas sempre haverá uma maneira de se comunicar caso algum dos nós saia da configuração inicial.

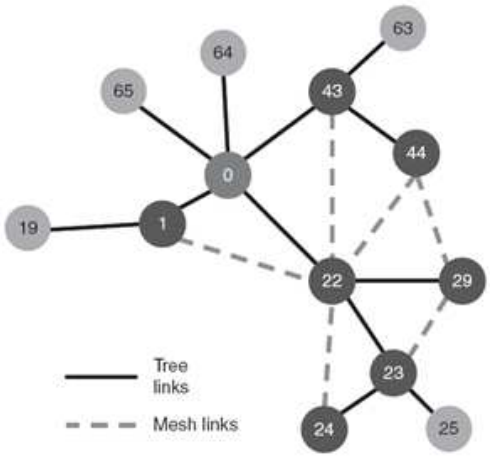

Figura 2.8: Mesh routing x tree routing [1]

O *broadcast* também é bastante importante. Antes de realizá-lo, um contador é colocado na mensagem enviada pelo nó emissor para determinar o raio da transmissão. Cada nó vizinho vai decrementar o contador em uma unidade e retransmite a mensagem caso o contador não tenha chegado ao valor zero. Se um nó recebe o mesmo *broadcast*, afim de evitar um loop, ele não faz nada. O contador com o valor 0x00 é um contador que não é decrementado e chega a todos os nós da rede. A Figura 2.9 mostra três estágios de um *broadcast* em uma rede qualquer.

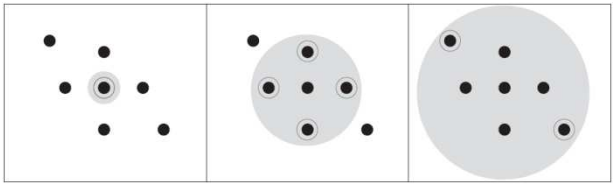
Figura 2.9: Broadcast em uma rede [1]

Em cada nó, são reservadas nove posições para *broadcasts*, na chamada Tabela de Transição de *Broadcast*, ou BTT. Essa tabela que é responsável por evitar que um broadcast passe duas vezes para a camada de aplicação. O tempo de vida de uma posição da tabela é de nove segundos e, portanto, a rede pode suportar um total de um *broadcast* por segundo para que não haja perda de informação (os *broadcasts* que chegam quando a BTT está lotada são descartados).

É importante citar que para que um dispositivo encontre uma rota, é necessário que este gere um *broadcast*. Outros comandos da camada de aplicação também o utilizam a cada novo nó que se junta à rede. Isso quer dizer que sempre que um novo nó entra, dois *broadcasts* são gerados, limitando então a entrada de novos nós para um a cada dois segundos. Ao tentar conectar vários nós de uma vez, várias tentativas deverão ser feitas até que todos tenham conseguido se conectar com sucesso, o que leva algum tempo dependendo do tamanho da rede.

Falou-se que uma rede em *mesh* é capaz de determinar as rotas automaticamente. Esse processo é fito com base em um *broadcast*. O algoritmo usado é baseado no AODV. As rotas são distribuídas e cada nó guarda qual o próximo salto de uma determinada rota numa tabela (*routing table*). Modelos mais novos do Zigbee, o *Zigbee Pro*, já interpretam o caminho como sendo bidirecional, mas até algum tempo as rotas eram unidirecionais, ou seja, a rota precisava ser descoberta para a ida e para a volta do sinal. Quando uma rota é determinada, ela continua a ser usada até que ela falhe em algum ponto, e quando isso ocorre uma mensagem é enviada ao nó inicial para que esse determine uma nova rota. O algoritmo AODV é utilizado porque em muitos sistemas Zigbee existe uma limitação de memória por causa dos microprocessadores de 8-bits. Existem outros métodos mais eficientes, porém mais complexos.

Para descobrir a melhor rota, cada roteador vira uma rota em potencial e uma mensagem é transmitida em *broadcast* a todos os nós da rede, uma vez que a priori o nó de origem não sabe aonde se encontra o nó de destino. Esse processo é chamado de *route discovery*. O Zigbee mantém uma tabela de *route discovery* (diferente da *routing table*) durante esse processo para achar a rota mais eficiente.

Em cada nó que a mensagem chega, uma variável correspondente ao quão eficiente é o enlace fica salva e a cada novo salto ela é adicionada de um novo valor, correspondente ao novo enlace. A variável têm valor 1 para a melhor situação possível e 7 para a pior situação possível, variando entre 1 e 7 para um enlace qualquer.

Para exemplificar, considere a Figura 2.10. Nela, o nó 1 deseja saber qual a melhor rota para o nó 10. Um *broadcast* é enviado e cada nó adiciona ao custo do caminho atual o valor correspondente ao último enlace. O *broadcast* então espalha o pedido por todos os nós. O menor custo de caminho que chegar ao nó 10 é dito o melhor caminho, e essa informação retorna ao nó 1 por meio de *unicast* para o nó de origem.

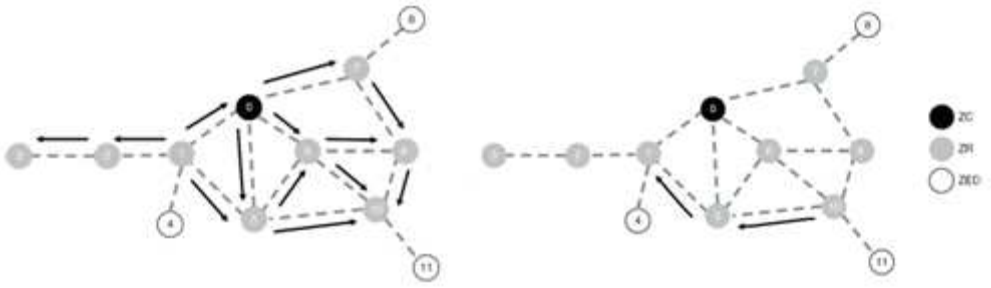

Figura 2.10: O processo de descoberta da melhor rota [1]

Os outros nós que guardam em suas tabelas de *route discovery* os valores dos custos de caminho eventualmente descartam o *route request*, limpando a tabela. Os nós 1 e 5 ainda guardam a rota, mas agora na *routing table*.

Por fim, quando uma mensagem é passada por *mesh* em uma determinada rota, em ambas as camadas MAC e NWK aparecem os endereços de rede correspondentes aos nós que fazem a transmissão da mensagem. Os endereços de origem e destino presentes na camada MAC são os endereços dos nós do enlace que está sendo feito. Os endereços da camada NWK são os endereços do nó de origem e do nó de destino.

3 MÓDULOS E SOFTWARE UTILIZADOS

3.1 INTRODUÇÃO

Esse capítulo visa introduzir o leitor aos equipamentos e métodos utilizados para fazer o controle dos dois ambientes propostos. Mostra-se primeiramente os módulos Zigbee escolhidos para fazer a rede de sensores e atuadores, os módulos Meshbeans, seguido por como foi pensado o circuito destinado a leitura da temperatura. Fala-se sobre os dois sistemas de refrigeração disponíveis, o sistema convencional e o sistema híbrido, terminando por fazer uma análise rápida do software embarcado que foi desenvolvido para fazer o monitoramento das redes.

3.2 MÓDULOS MESHBEAN

O Kit de Desenvolvimento da empresa Meshnetics, ou módulo Meshbean2, foi utilizado como o dispositivo de comunicação Zigbee do projeto. Mostra-se na Figura 3.1 uma foto com indicações a alguns de seus principais periféricos.

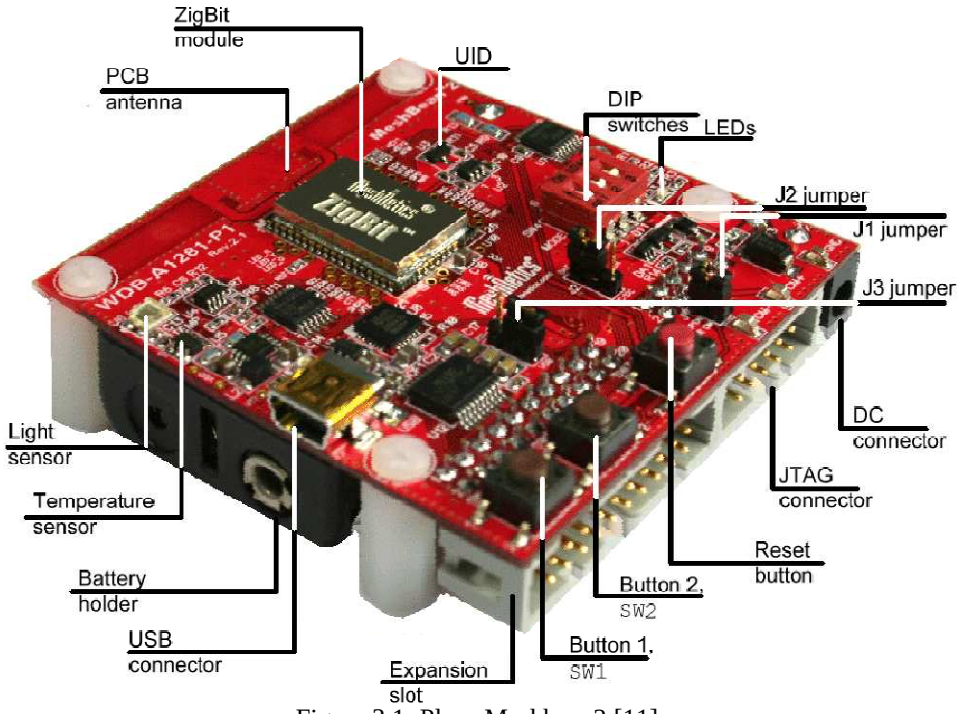

Figura 3.1: Placa Meshbean2 [11]

O Meshbean é um dispositivo muito prático. A quantidade de periféricos o torna bastante versátil e com isso pode ser usado para uma variedade de aplicações. Possui um sensor de temperatura LM73CIMK da National Semiconductors (com conversor A/D de 11-14 bits, configurável), um sensor de luminosidade TSL2550T da TAOS, um medidor de bateria com divisor resistivo, DIP switches, dois botões que podem ser utilizados pela aplicação (há um terceiro botão para reset), LEDs, além do microprocessador Zigbit, da Atmel, e de uma antena integrada.

A comunicação entre o Zigbit e os módulos sensores, na verdade com grande parte dos periféricos, é feita por um barramento cujo protocolo de comunicação é o I2C. Uma exceção é em relação à leitura da bateria, que é ligada diretamente ao conversor A/D do chip.

A comunicação com o computador é feita por USB (CP2102 da Silicon Labs) e é necessário que o usuário instale um driver da fabricante para se comunicar com a placa. Ocorre que o protocolo de comunicação usado é o RS-232, um protocolo de comunicação serial, mas o conector utilizado é USB. A função do driver é fazer com que a porta USB seja identificada como uma porta COM e fazer a interface entre a porta e o protocolo.

Uma peculiaridade encontrada diz respeito ao sensor de temperatura LM73. Seu datasheet [17] afirma que a resolução mínima feita por seu conversor é de 0,25°C (e a máxima de 0,03125°C), mas até então o valor fornecido pela aplicação embarcada disponível possuía uma resolução de 1°C, quatro vezes maior que a mínima e completamente inaceitável para um sistema de controle. Esse problema sempre foi atribuído ao hardware, mas na verdade era um problema de software. Quando a leitura de temperatura é feita, o sinal enviado ao Zigbit pelo barramento é formado por 8 bits (os bits 7:14) que correspondem à parte inteira do valor lido e outros 5 bits (os bits 2:6) que correspondem à parte decimal, sendo que os dois bits menos significativos nunca são utilizados (os bits 0 e 1, sempre iguais a zero) e o bit mais significativo (bit 15) é usado para indicar o sinal da temperatura lida (seguindo a estrutura de complemento de dois).

O software disponível pela Atmel para download, o BitCloud, ao ler a temperatura, retornava apenas o valor inteiro lido, provavelmente porque para evitar variações na temperatura medida. Ao retirar a parte do código que removia a parte decimal, foi possível obter a leitura com a resolução de 0,25°C esperada. Embora não se tenha conseguido acessar o barramento I2C para alterar o registrador responsável por determinar a quantidade de bits a ser usada na conversão, já é possível fazer um processo de controle aceitável. O padrão é o conversor de 11bits, por ser mais rápido que os outros, pois seu tempo de conversão é no máximo 14ms, enquanto para 14bits o tempo máximo é de 112ms [17].

Mesmo assim, verificou-se que a temperatura lida não condiz com a temperatura real do ambiente. De nada adiantou mudar o código fonte do BitCloud, porque entendeu-se que a causa era a localização do sensor LM73, bem em cima de uma das baterias, que poderiam esquentar um pouco a placa e causar um erro na leitura da temperatura. Foi por causa desse imprevisto que uma placa externa foi confeccionada para fazer a leitura da temperatura, aproveitando-se do conversor analógico-digital presente no chip Zigbit.

Existem alguns cuidados que devem ser observados. A placa possui três jumpers e é preciso ter certeza se eles estão nas posições certas em relação tanto ao tipo de comunicação utilizada quanto à alimentação para não danificá-la permanentemente. Outro cuidado é com eletricidade estática, que deve ser evitada na medida do possível porque a o módulo é sensível a ela, afim de não atrapalhar a aplicação ou danificar os componentes. Obviamente, não são recomendados choques, que podem causar a perda de algumas funcionalidades da placa por danificar periféricos, devido à característica um pouco frágil do aparelho.

3.3 MÓDULO EXERNO PARA LM35

Devido ao problema da leitura correta da temperatura, foi necessária a utilização de um circuito externo com outro sensor. Optou-se por utilizar o sensor de temperatura LM35DZ, da fabricante National Semiconductor, por causa da sua disponibilidade no laboratório e por causa da maneira simples da apresentação da tensão de saída com relação à temperatura, que é de 0,01V por °C, em uma escala praticamente linear para temperaturas de -55°C até 150°C [16]. O esquemático do circuito é mostrado na Figura 3.2. Nota-se que foi usado também um amplificador operacional para amplificar a leitura e facilitar a conversão analógico-digital. O amplificador escolhido foi o MCP604, da fabricante Microchip, porque este poderia ser alimentado com a fonte de tensão de $5V_{DC}$ disponível.

O conversor AD do Zigbit foi programado para trabalhar com uma tensão de referência de 1,25V e 10bits para a conversão. Devido à baixa tensão de referência, escolheu-se um ganho pequeno para o amplificador do circuito. Dado que a equação do ganho é

$$G = 1 + \frac{R_2}{R_1} \tag{3.1}$$

escolheu-se R_1 e R_2 com os valores de 1kΩ e 2,2kΩ, respectivamente, para que o ganho fosse igual à 3,2. Dessa forma, a temperatura ambiente pode chegar a 39°C antes da leitura do conversor AD ficar saturada.

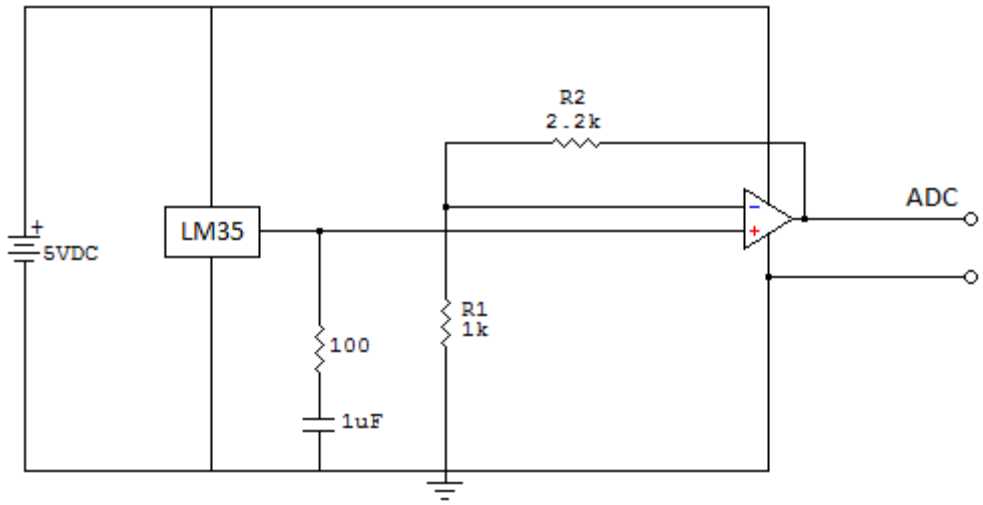

Figura 3.2: Esquemático da placa externa

Por fim, mostra-se uma foto do circuito montado, na Figura 3.3. A fonte de 5V alimenta o circuito e o sensor, a entrada do amplificador é a temperatura medida e a saída do circuito é a tensão amplificada, que é ligada diretamente no pino ADC_INPUT_1 da placa Meshbean. A outra saída do circuito é um pino que deve ser ligado ao terra da placa Meshbean.

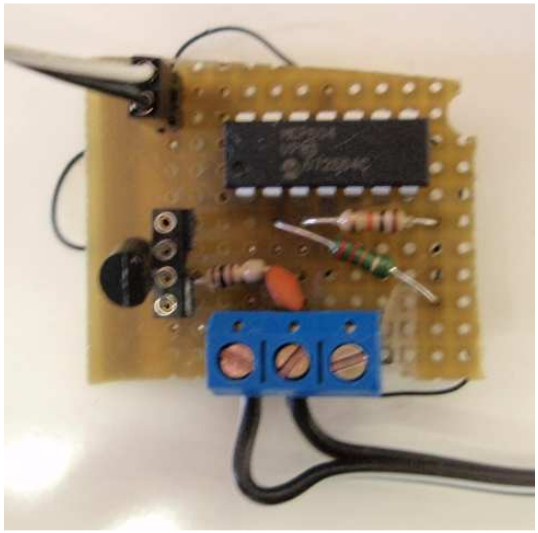

Figura 3.3: Circuito montado

3.4 AR CONDICIONADO SPLIT

O projeto se propõe a fazer o controle térmico de dois ambientes, e para isso precisa fazer o sensoriamento das condições térmicas para acionar os aparelhos de ar-condicionado que estão disponíveis. Nessa seção fala-se sobre os aparelhos Split presentes na sala do LAVSI.

Os dois aparelhos de ar condicionado instalados são do tipo Split, ou simplesmente Split. São ambos do mesmo modelo, MAXIFLEX, da marca Springer, e possuem capacidade de 22000 BTU/h cada.

A atuação neles é feita com um circuito composto por um relé de estado sólido, que fecha seus contatos na presença de um sinal enviado pelo módulo Meshbean com função de atuador, ligado ao circuito de compressão do ar. Quando os contatos estão abertos, o ar condicionado fica apenas ventilando o ambiente. A Figura 3.4 mostra uma foto do aparelho.

Figura 3.4: Ar condicionado Split

3.5 SISTEMA DE REFRIGERAÇÃO HÍBRIDO

A sala de reunião, ao contrário da parte externa a ela, não possui nenhum ar condicionador do tipo Split. O equipamento instalado é o de um sistema ar condicionado híbrido para a climatização do ambiente, que possibilita, além de controlar a temperatura, controlar também a umidade do ar. O sistema pode ser operado basicamente em quatro modos: ventilador, modo convencional, modo evaporativo e híbrido.

Para melhor compreendê-los, mostra-se primeiro o diagrama de forças do equipamento, em sua configuração de fábrica, na Figura 3.5.

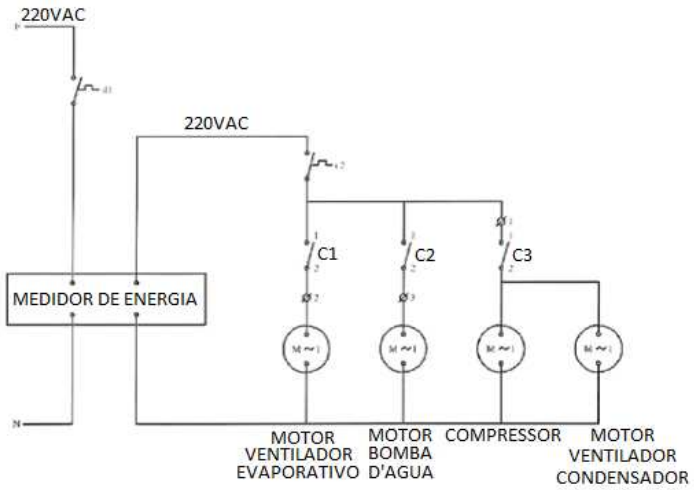

Figura 3.5: Diagrama de força do sistema híbrido

No diagrama, C1, C2 e C3 representam os contatos de uma contatora normalmente aberta. No sistema presente no laboratório, o contato C1, que aciona o ventilador do sistema evaporativo, também aciona o contato que habilita todos os modos de operação. Dessa forma, quando a chave do ventilador está fechada, pode-se atuar no sistema mudando a posição dos outros contatos.

Existe também um *damper*, que nada mais é do que um duto usado para mudar a entrada de ar do sistema. Quando o *damper* fechado, o ar vem exclusivamente do ambiente externo, passando

necessariamente por uma manta de celulose, que faz parte do sistema do modo evaporativo. Com o *damper* aberto, o ar vem do ambiente interno, com pouca entrada de ar externo por causa da maior resistência ao fluxo. Um detalhe importante é que a abertura e o fechamento do *damper* estão eletricamente ligados ao circuito que liga e a desliga, o que quer dizer que quando o compressor liga, necessáriamente o damper abre, e vice-versa. Um detalhe específico do *damper* presente na sala de reunião é que ele não possui nenhum estado intermediário entre totalmente aberto ou totalmente fechado. A Figura 3.6 mostra uma foto do damper, no momento em que este se encontra fechado.

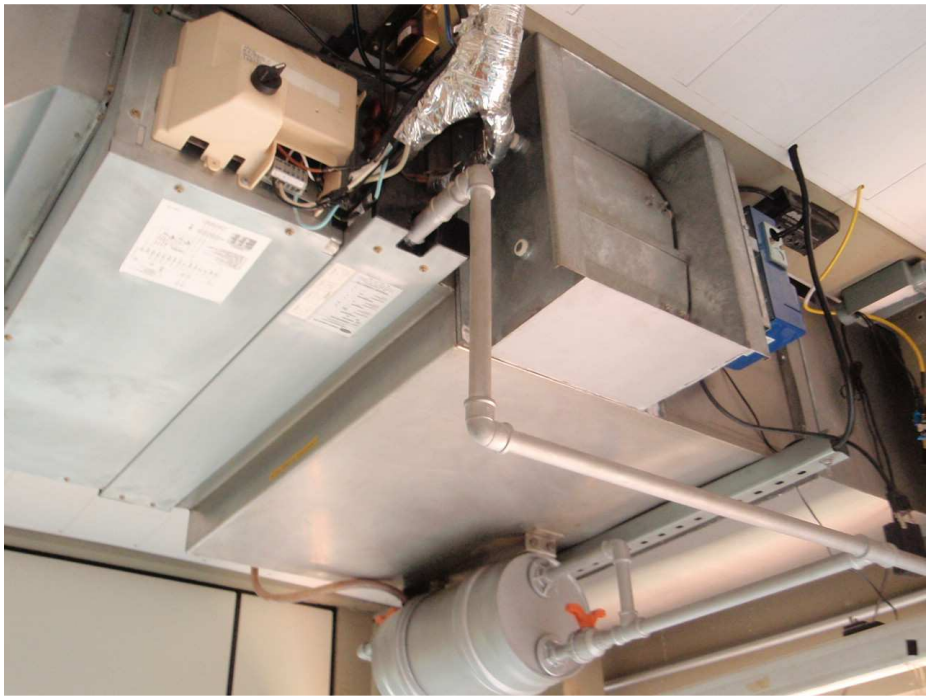

Figura 3.6: Damper fechado

Quando se usa o aparelho com o ventilador evaporativo habilitado, o sistema faz a circulação do ar. Esse modo é importante porque, sem ele, nenhum outro funciona (por causa da configuração da contatora). Se não fosse dessa forma, algumas considerações deveriam ser observadas para o bom funcionamento do sistema de refrigeração híbrido. É importante não deixar o ventilador desligado juntamente com o compressor ligado para evitar danos no sistema do próprio compressor, que pode entupir suas serpentinas por causa das baixas temperaturas e do fenômeno químico da condensação. Também não é recomendado que o motor da bomba d'água fique ligado sem ventilação, para não haver danos na manta de celulose presente no sistema evaporativo, como o surgimento de mofo/fungos por causa do acúmulo de água na celulose.

O modo de operação convencional caracteriza-se quando o sistema opera com os contatos do ventilador e do compressor ligados. O funcionamento é o mesmo de qualquer outro ar condicionado comum, no qual o compressor é responsável por resfriar um gás refrigerante dentro das serpentinas para o que o ar perca energia ao passar e resfria o ambiente. Nessa configuração, o *damper* está necessariamente aberto pelos motivos já explicados.

Ao ligar os contatos da bomba d'água e do ventilador ao mesmo tempo, o sistema opera no modo evaporativo. O *Damper* nesse caso está fechado porque nessa configuração a chave do compressor se encontra aberta. Dessa forma, todo o ar que entra no sistema provém do ambiente externo e é obrigado a passar pela manta de celulose. A manta é molhada com água e, por causa disso, o ar se umidifica ao mesmo tempo em que perde energia. O ambiente interno então recebe esse ar mais úmido e frio e acaba se resfriando. Nesse modo, a economia de energia é muito grande, e projetos que visam utilizá-lo devem maximizar seu uso. Apesar disso, não é tão eficiente e pode não conseguir resfriar o ambiente na presença de uma grande carga térmica. Além diso, existe naturalmente um problema quando a umidade relativa do ar externo estiver muito alta.

No último modo de operação, todas as chaves estão fechadas, acionando tanto o sistema responsável por molhar a manta de celulose quanto por resfriar a serpentina do compressor. O *damper*, nesse caso, está aberto. O ar que vem de fora passa pela manta e chega a uma temperatura mais baixa no compressor, facilitando então o processo de resfriamento do ar. Um inconveniente nessa configuração é a falta de estados inermediários do *damper*, que evitariam a dificuldade que o ar externo encontra para entrar no duto.

A Figura 3.7 mostra a parte do equipamento que se encontra na sala de reunião. Ainda existe uma parte externa, não mostrada.

Figura 3.7: Ar condicionado Híbrido

No desenvolvimento do projeto, escolheu-se trabalhar com o ar condicionado híbrido atuando-se apenas na chave do compressor. Manteve-se o ventilador sempre ligado e o dispositivo atuador é responsável por enviar um sinal que fecha os contatos de um relé de estado sólido para acionar o compressor. A contatora do sistema fica então inutilizada e os outros modos não são usados. Essa escolha se baseou na proposta inicial de integrar as redes, e não fazer o melhor processo de controle possível.

3.6 CONTROLE POR SOFTWARE EMBARCADO

No projeto, optou-se por fazer o controle liga-desliga diretamente nos módulos coordenadores da rede, ao contrário dos projetos anteriores. Até então, o software supervisório fazia todas as contas a partir das leituras recebidas pelo nó coordenador e enviar pela USART as informações necessárias para o dispositivo enviar uma mensagem aos atuadores, para ligar ou desligar os respectivos sistemas.

Com o controle embarcado, é possível que um sistema funcione mesmo quando o coordenador da rede não se encontra ligado a um computador. Esse é um detalhe importante para que fosse possível a integração de diferentes redes de sensores, pelo menos da forma como a solução foi proposta. A idéia é criar um dispositivo Zigbee que fique trocando periodicamente de rede, já que cada rede define um ambiente a ser controlado, e fazer o monitoramento remoto, obtendo os dados de temperatura para serem mostrados no software supervisório. Além do monitoramento, esse novo dispositivo também é responsável por atualizar a referência dos sistemas de controle, quando necessário.

Por suas características, o novo dispositivo ganhou o nome de *jumper* (saltador, em inglês). A cada 30 segundos, o *jumper* envia um pedido ao coordenador da rede para que este transmita as últimas leituras da temperatura, bem como do estado dos atuadores. Recebida a resposta do pedido, o *jumper*

envia os dados ao SIMULINK (mais sobre o software supervisório no próximo tópico, 3.7) e faz a troca de rede, para então fazer uma nova requisição de dados 30 segundos depois, agora no outro sistema controlado.

Na solução apresentada, o único problema seria quando o sistema controlado não estivesse no raio de alcance da antena do dispositivo que captura os dados. Esse representa um problema para quando a topologia da rede é definida, e não interferiu por causa das dimensões físicas do LAVSI, que possui 66m^2 e possibilita a comunicação de qualquer dispositivo que esteja dentro do laboratório. Para redes maiores, seria necessário que um dispositivo roteador estivesse presente, possibilitando a comunicação do *jumper* com o coordenador da rede.

3.7 SOFTWARE SUPERVISÓRIO

Uma vez que o *jumper* recebe os dados da rede, ele automaticamente os envia para o SIMULINK, que é responsável pela apresentação ao usuário dos parâmetros de interesse.

O SIMULINK é um pacote do software MATLAB que possibilita a criação de simulações por meio de diagramas de blocos. Nesse caso, o processo não é uma simulação porque depende de um componente externo, o próprio dispositivo Zigbee, para que os estados no programa desenvolvido sejam atualizados.

A função do modelo implementado para ser o supervisório é bastante simples: mostrar as variáveis do sistema graficamente e enviar a temperatura de referência para as redes. A Figura 3.8 mostra a parte mais alta do programa. A partir desse ponto, trata-se rede A como sendo a rede do LAVSI e rede B como sendo a rede da Sala de Reunião.

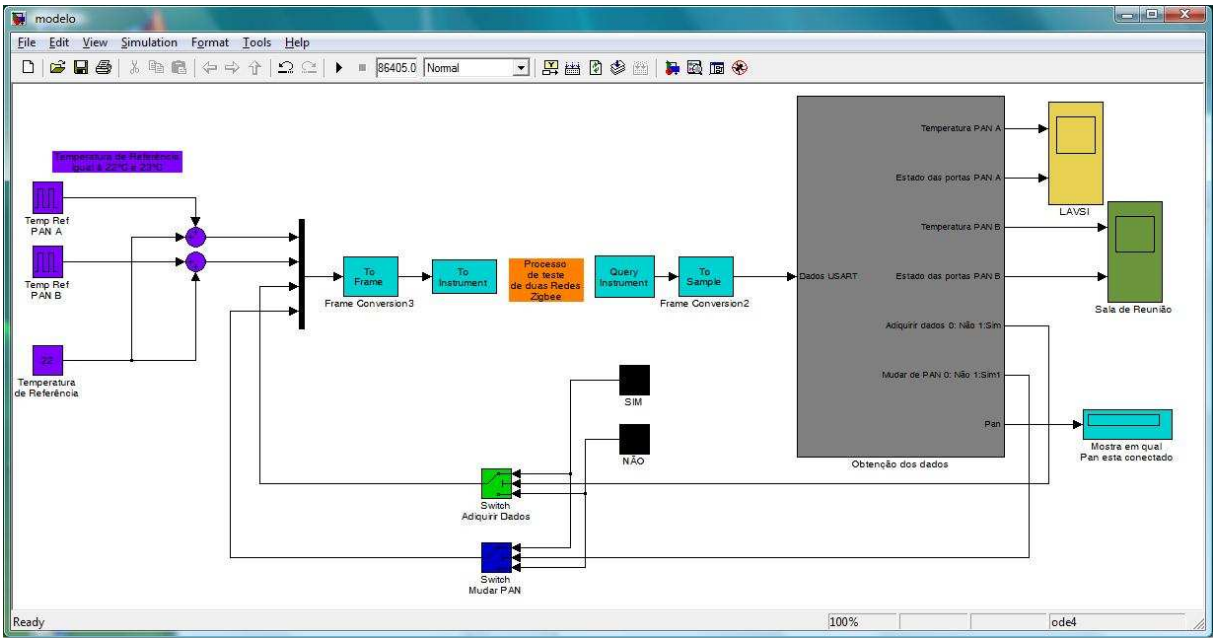

Figura 3.8: Software supervisório

Para entender o que acontece em cada um dos blocos do modelo, é necessário começar por aquele que recebe os dados do *jumper*. A comunicação é feita pelo protocolo de comunicação USART, que é um protocolo de comunicação serial. O bloco *Query Intrument* lê a porta serial em busca dos dados que são enviados pelos dispositivos e forma com eles um quadro, que é transformado em um vetor de amostras no bloco *To Sample*. Esses dados são então enviados ao bloco que faz o tratamento da informação, o bloco *Obtenção dos dados*, que será explicado logo mais. As saídas desse bloco são os valores das variáveis de temperatura lidas pelos sensores e as variáveis do estado da porta ADC_INPUT_1 dos atuadores. Essas leituras são mantidas por 1 minuto. Lembre-se que o sistema foi projetado para obter os dados de cada rede a cada 30 segundos, ou seja, as variáveis de uma rede (da

rede A ou da rede B) são atualizadas de minuto a minuto. Mostra-se também, em tempo real, em qual rede o *jumper* está conectado (visor em azul), apenas para conferência do usuário.

Antes de explicar as duas outras saídas do bloco *Obtenção dos dados*, se faz necessário um conhecimento mais profundo do que ele faz. A Figura 3.9 mostra o que se passa dentro dele.

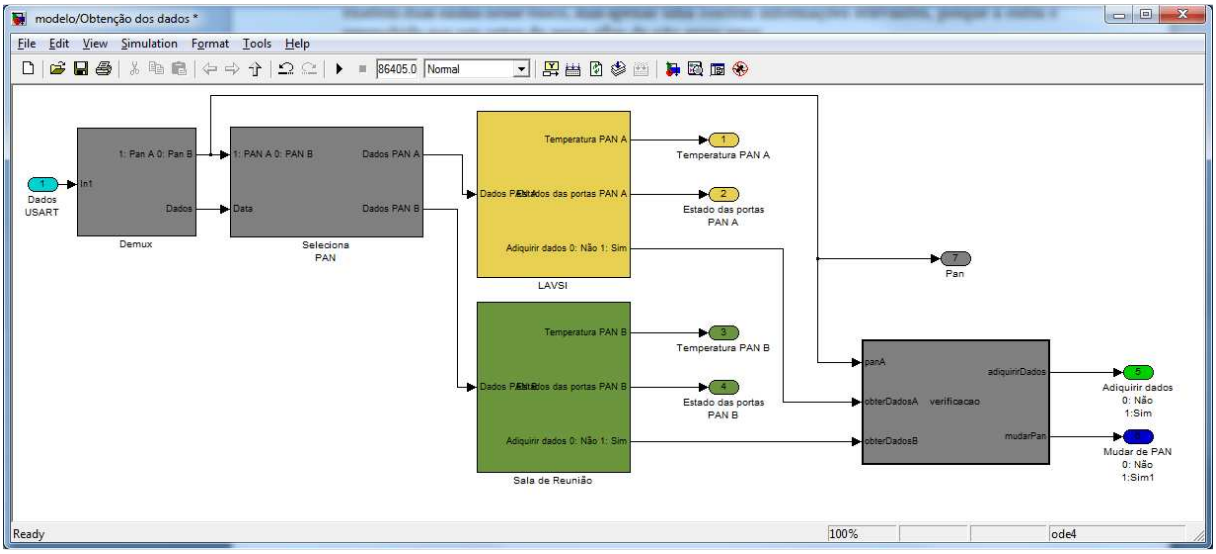

Figura 3.9: Bloco Obtenção dos dados

O vetor de amostras recebido contém seis informações, enviadas na seguinte ordem pelo dispositivo para a USART:

- *flag* para dizer se é uma mensagem para manutenção da comunicação ou se é uma mensagem de dados;
- *flag* indicando de qual das redes os dados pertencem;
- *flag* que identifica à qual dispositivo da rede se refere a mensagem;
- temperatura medida;
- bateria medida;
- estado da porta.

Por questões que serão explicadas no Capítulo 4, a leitura da bateria não é mais realizada, mas ainda é enviada para caso alguma aplicação futura for utilizá-la.

O bloco *Demux* separa as informações e possui duas saídas: a *flag* para diferenciar as redes e um vetor com o restante das informações. O tratamento da mensagem começa no bloco seguinte, denominado *Seleciona PAN*. Sua função é separar a mensagem que possui dados da rede A para a parte do programa que trata da rede A e fazer o mesmo quando a mensagem possui dados da rede B. Note que existem duas saídas nesse bloco, mas apenas uma contém informações relevantes, porque a outra é preenchida por um vetor de zeros afim de não gerar erros.

A partir desse ponto o programa começa a ficar escalonável. Os blocos *LAVSI* e *Sala de Reunião* são essencialmente a mesma coisa, o que muda é a quantidade de dispositivos da rede que cada um trabalha. A Figura 3.10 e a Figura 3.11 mostram em detalhes cada um dos blocos em questão, para que seja feita uma análise simultânea de seus componentes.

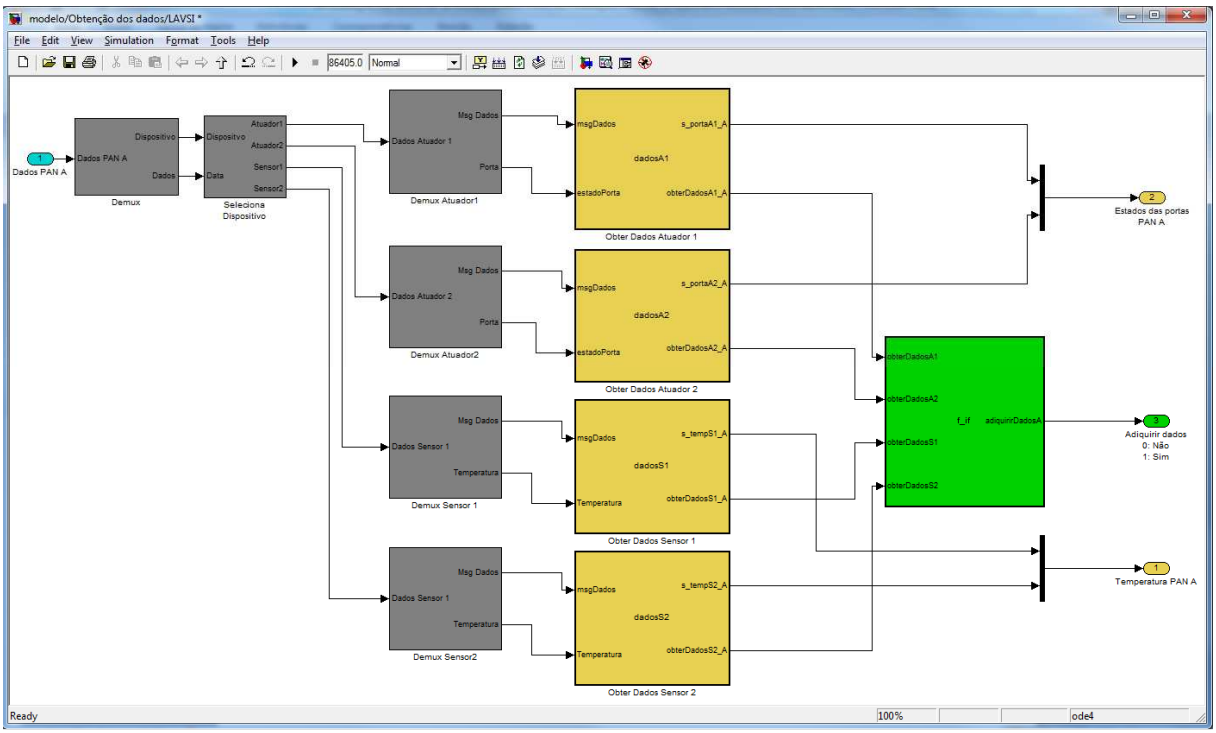

Figura 3.10: Bloco LAVSI

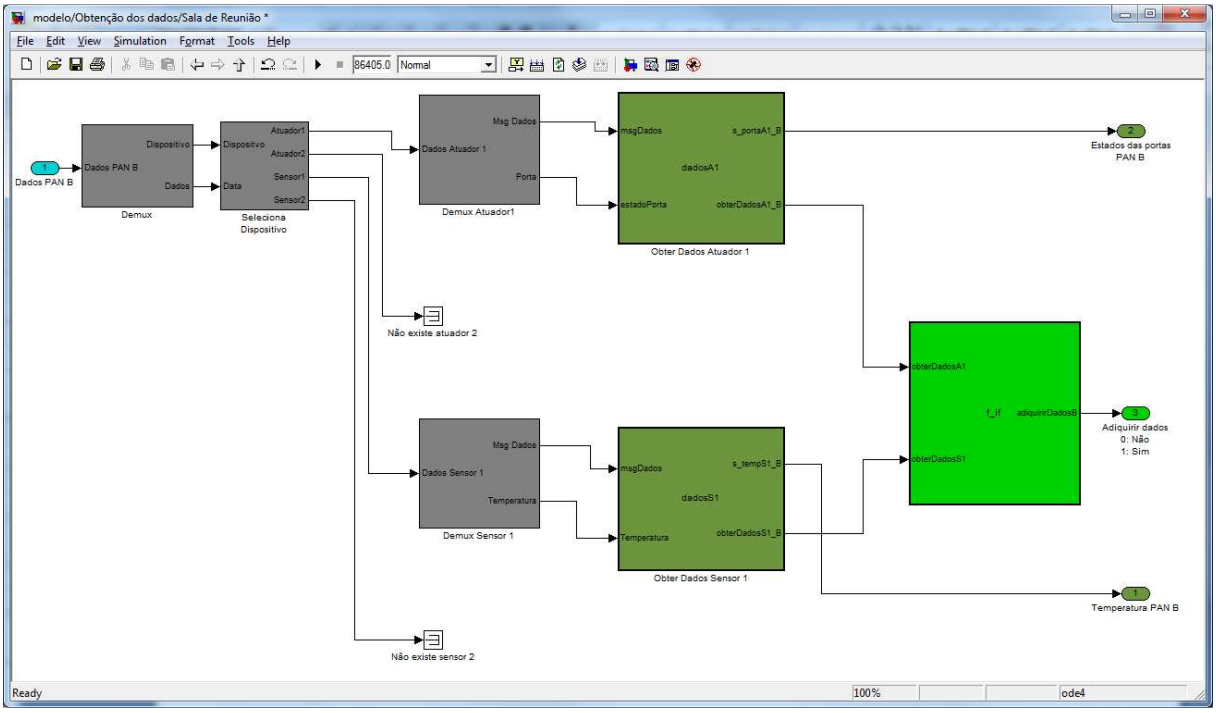

Figura 3.11: Bloco Sala de Reunião

Novamente, separa-se do vetor de dados uma componente importante: a *flag* que informa de qual dispositivo são os dados recebidos. No bloco *Seleciona Dispositivo* existe uma lógica que envia os dados referentes a cada dispositivo quando necessário, ou um vetor nulo quando o dispositivo não corresponde ao qual a mensagem se refere.

Novamente, o diagrama fica escalonável. Perceba que a diferença entre o bloco da *LAVSI* (Figura 3.10) e do bloco da *Sala de Reunião* (Figura 3.11) é a falta dos dispositivos *sensor 2* e *atuador 2* no segundo. Isso provém da tipologia da rede, que será vista em maiores detalhes no Capítulo 4.

Os próximos blocos, todos chamados *Demux {dispositivo}*, fazem a separação do vetor em variáveis simples, eliminando as que não são usadas. Por exemplo, se a mensagem é de um dispositivo atuador,

a posição que corresponde à temperatura é descartada, da mesma forma que quando a mensagem é de um sensor, a posição do vetor correspondente ao estado da porta também é descartada. Também é aqui que se descarta a informação do nível da bateria.

Os blocos seguintes seguem a mesma rotina, mudando apenas a variável envolvida. São responsáveis por manter a saída igual ao último valor enviado por um minuto e determinar se é necessário que uma nova medida seja feita, caso o tempo esteja acabando e ainda não existe um novo valor para ser mostrado.

As saídas de ambos os blocos são iguais. Na *saída 1*, segue o valor da temperatura para ser mostrada/guardada. Na *saída 2*, segue o estado da porta ADC_INPUT_1 do atuador correspondente. Na *saída 3*, segue uma *flag* que indica se é necessária a obtenção de novos dados para a rede em questão.

Voltando para o bloco *Obtenção de dados* (Figura 3.9), vê-se que este redireciona os valores de temperatura e estado das portas a serem mostrados para a parte mais alta do programa, gerando as saídas que já foram mencionadas. As duas outras saídas que não foram explicadas correspondem à necessidade de se obter novos dados de uma das redes. Caso o *jumper* esteja conectado à PAN A e faltam dados da PAN A, o bloco *verificação* propaga o pedido por dados para a parte mais alta, que aciona um switch para transmitir essa informação para o dispositivo. Caso os dados a serem obtidos pertencem à rede à qual o *jumper* não está conectado no momento, além de se propagar o pedido por novos dados, aciona-se uma *flag* para que a rede seja trocada, que também acionará um switch no programa da Figura 3.8.

Por fim, falta descrever a parte do modelo que envia o sinal de referência da temperatura. Novamente em relação à Figura 3.8, os blocos responsáveis por esse processo estão marcados em roxo. A temperatura de referência é composta por uma parte fixa e outra que varia entre 0 e 1 por meio de uma onda quadrada. Foi feito dessa forma para que houvesse um degrau em alguns momentos do teste para que fosse verificado se o sistema consegue mudar sua temperatura de referência com sucesso e fazer o processo de controle. Essas informações são enviadas ao *jumper* pelo bloco *To Instrument*, de maneira semelhante ao que é feito no *Query Intrument*, só que enviando e não recebendo dados.

Em [19] fala-se com mais detalhes sobre o uso de variáveis globais no SIMULINK, que são ferramentas novas, introduzidas no MATLAB 2010a, que possibilitaram a lógica em paralelo que foi implementada.

4 DESCRIÇÃO DA REDE

4.1 INTRODUÇÃO

Como foi muitas vezes mencionado, o projeto se propõe fazer o controle de temperatura de dois ambientes visando o conforto térmico de seus usuários. Esse capítulo tem como objetivo falar sobre cada dispositivo da rede e a lógica de programação que foi gravada no microprocessador Zigbit. Juntamente com o software supervisório, o desenvolvimento da lógica da rede foi a parte chave do projeto.

Primeiramente, fala-se um pouco dos ambientes controlados e da topologia da rede. Grande parte do assunto já foi coberto no Capítulo 3, principalmente com respeito à atuação nos aparelhos, portanto o texto não aborta muitos assuntos que supõe-se entendidos.

Em seguida, uma explicação detalhada de cada módulo da rede é feita, mostrando-se sempre um diagrama de estados para facilitar o entendimento do leitor. Para detalhes no código em C, vide Anexo I.

4.2 TOPOLOGIA DA REDE

A Figura 4.1 mostra a planta do laboratório LAVSI, localizado no prédio SG-11, campus Darci Ribeiro – UnB. Repetindo o que já foi dito, cada rede controla um ambiente: a rede A (em amarelo) controla o LAVSI e a rede B (em verde) controla a Sala de Reunião.

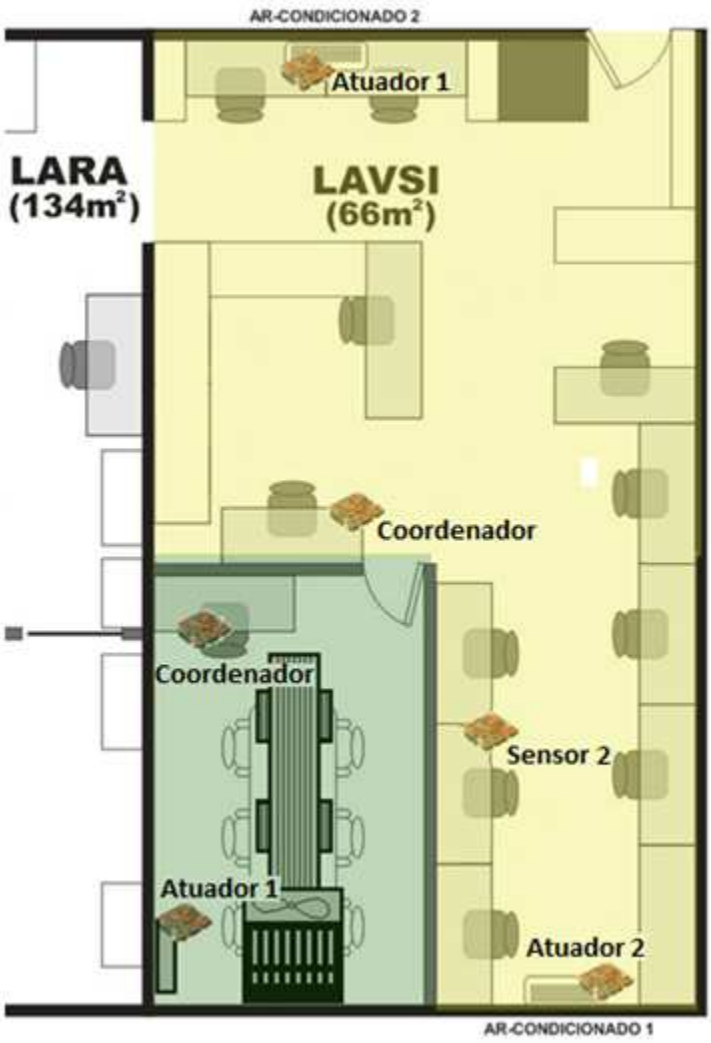

Figura 4.1: Planta baixa do ambiente controlado

Por ser um ambiente menor, a Sala de Reunião não precisa de muitos dispositivos sensores espalhados pra que a leitura da temperatura fosse feita e o controle realizado. Por outro lado, o LAVSI já é um ambiente maior e mais difícil de ser analisado, inclusive por causa das perdas de calor para o laboratório LARA, que se encontra ao lado, sem nenhum tipo de isolamento. Colocou-se então dois pares de dispositivos sensores/atuadores na rede A e um par para fazer parte da rede B. Suas localizações estão as indicadas na Figura 4.1.

Um detalhe importante é que, por falta de dispositivos Meshbean, o coordenador também atua como nó sensor. No caso, o coordenador de cada uma das redes é também o sensor 1. Apesar de não ter sido a idéia inicial, agregar funções não foi uma solução ruim, porque assim economiza-se tanto em módulos como no uso do espectro, evitando inclusive a geração de possíveis ruídos nas outras redes que atuam na mesma freqüência de 2,4GHz. Para habilitar/desabilitar essa função, existe uma chave de compilação chamada _FALTAM_MESHBEANS_ no código *projetoFinal.h*, que quando definida habilita o acúmulo de funções do coordenador.

4.3 APLICAÇÃO EMBARCADA

A aplicação desenvolvida baseia-se no código do *lowpower*, um programa disponível no pacote do BitCloud fornecido pela fabricante Atmel. O BitCloud é o software multitarefa baseado em estados criado para facilitar o desenvolvimento de aplicações para o chip Zigbit. Mais detalhes sobre o assunto podem ser vistos em [12].

Em todos os módulos, existem dois processos que funcionam em cascata: o da aplicação topo e o do dispositivo. O processo da aplicação topo é responsável por iniciar o módulo e criar (ou conectar-se à) uma rede. Assim que a rede está formada, a própria aplicação topo se encarrega de redirecionar o programa para a máquina de estados do dispositivo. Os estados são atualizados sempre por meio da função "SYS_PostTask(APL_TASK_ID);", que redireciona o código para a aplicação topo, que faz outro redirecionamento para a aplicação específica do dispositivo, caso não exista nenhuma mudança na rede.

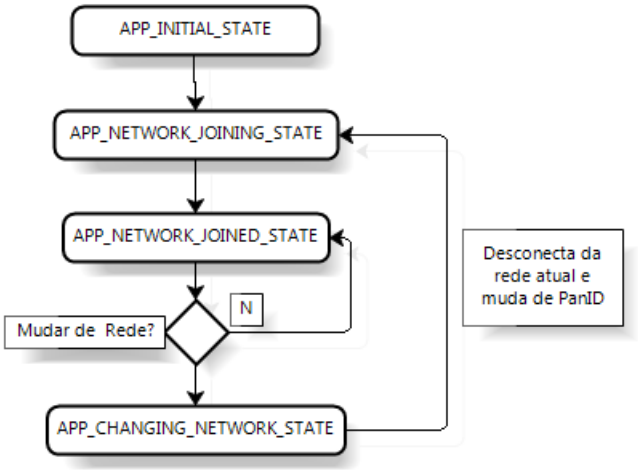

Figura 4.2: Diagrama de estados da aplicação topo

A lógica utilizada é bastante simples e pode ser vista no diagrama da Figura 4.2. Quando o dispositivo é ligado, o estado APP_INITIAL_STATE verifica, através de uma leitura dos DIP Switches, qual a função do dispositivo (coordenador, enddevice ou jumper) e chama a respectiva função para iniciar seu processo. Além disso, essa função é responsável por fazer as configurações necessárias para o bom funcionamento da rede, como por exemple determinar os endereços da rede, o *network address*, que foram configurados para serem aleatoreamente escolhidos. As duas únicas configurações que não são feitas aqui correspondem aos endereços da rede pretendida. Por causa do *jumper*, que fica trocando seu endereço de PAN, essa configuração foi colocada no próximo estado.

A correspondência entre o valor dos DIP Switches e da função do dispositivo é mostrada na Tabela 2.

Tabela 2: Valores dos DIP Switches e os nós da rede correspondente

Valor DIP Switch	Função do Dispositivo
0	Coordenador
1	Sensor 1
2	Sensor 2
3	Atuador 1
4	Atuador 2
5	Não utilizado
6	Não utilizado
7	Jumper

Entre o APP_INITIAL_STATE e o estado seguinte, APP_NETWORK_JOINING_STATE, o usuário precisa pressionar o botão 1 da placa Meshbean para que se confirme o desejo de iniciar a comunicação. Feito isso, o dispositivo determina os endereços de PAN que vai criar (caso seja um dispositivo coordenador) ou que vai procurar (caso seja um dispositivo *enddevice* ou *jumper*). São configurados dois endereços, o endereço extendido, de 64bits, e o endereço curto, de 16bits. Existe uma *flag* para indicar a rede pretendida, chamada *conectedPanA*, que também servirá para a troca das redes. Com todos os parâmetros definidos, envia-se um pedido para a camada ZDO do BitCloud para iniciar-se a rede.

Nesse momento, caso a resposta do pedido seja positiva, o dispositivo atualiza seu estado global para APP_NETWORK_JOINED_STATE e chama o processo correspondente ao módulo, iniciando a respectiva máquina de estados. Caso não ocorra um sucesso, o estado não é atualizado até que se a comunicação seja estabelecida. É um problema que acontece quando a rede não está presente ou por algum problema com o ruído no canal.

A transição de APP_NETWORK_JOINED_STATE para APP_CHANGING_NETWORK_STATE é feita pelo *jumper*. Esse estado tem como função deixar a rede atual e se conectar novamente, mas agora na outra rede, porque os endereços PAN (extendido e curto) serão diferentes.

Por fim, precisa-se lidar com as interrupções que são geradas quando mensagens da rede são recebidas. A função da aplicação topo que lida com esse tipo de interrupção simplesmente redireciona o programa para a função do dispositivo que faz o tratamento do que foi recebido.

4.4 DISPOSITIVO COORDENADOR

O coordenador faz o controle embarcado e se comunica com o *jumper*. Seu diagrama de estados é mostrado na Figura 4.3.

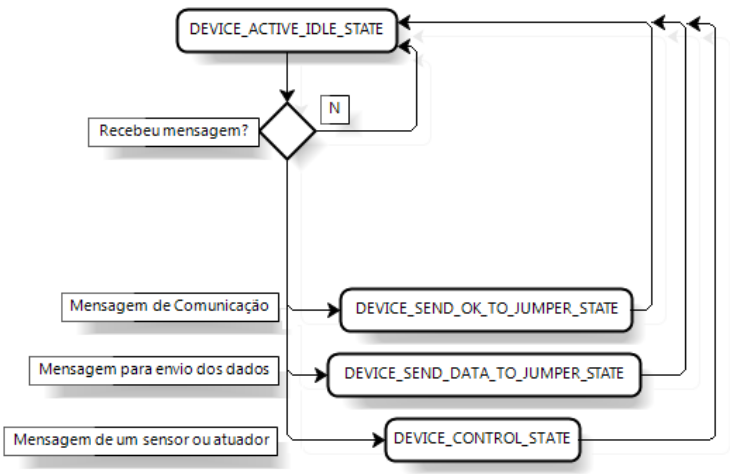

Figura 4.3: Diagrama de estados do coordenador

Basicamente, o coordenador trabalha apenas com as interrupções geradas quando uma mensagem é recebida. O estado inicial, DEVICE_ACTIVE_IDLE_STATE, não faz absolutamente nada.

Dessa forma, existem alguns tipos esperados de mensagem que são aguardados. Quando o *jumper* envia uma mensagem para verificar se a comunicação está funcionando, o estado é atualizado para DEVICE_SEND_OK_TO_JUMPER_STATE, encarregado de enviar uma resposta para manter a comunicação. Quando o *jumper* envia um pedido fazendo a requisição os dados salvos da rede, o estado é atualizado para DEVICE_SEND_DATA_TO_JUMPER_STATE, responsável por transmitir as informações que chegarão ao software supervisório. Por fim, quando algum nó sensor ou atuador se comunica com o coordenador, o estado é atualizado para DEVICE_CONTROL_STATE, que gerencia o controle liga-desliga do sistema.

O controle embarcado é feito quando uma mensagem chega do sensor, que envia um dado de temperatura que é salvo. Dependendo da temperatura lida, o coordenador envia uma mensagem para o atuador ligar ou desligar o ar condicionado. A mensagem do atuador apenas informa o estado da porta que controla do sinal para o ar condicionado para que seu valor seja guardado.

Devido à falta de dispositivos, a lógica muda um pouco para o par sensor/atuador 1: o controle é feito quando uma mensagem do atuador é recebida. Após ter seu valor da porta atualizado no código do coordenador, chama-se uma função para que seja feita a leitura da temperatura e assim simular o nó sensor, inclusive salvando o que foi lido na variável correspondente a ele.

A temperatura de referência é atualizada quando uma mensagem vinda do *jumper* chega.

4.5 DISPOSITIVO END DEVICE

Os dispositivos finais da rede são exatamente os nós sensores e atuadores. O diagrama de estados é mostrado na Figura 4.4.

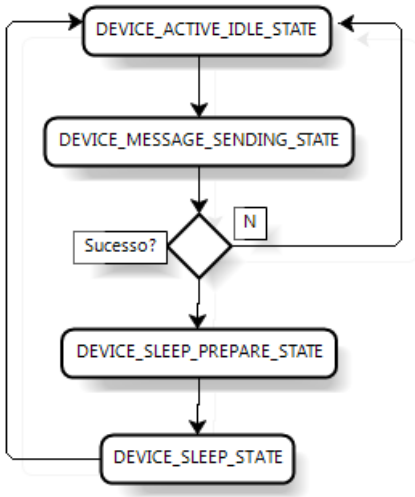

Figura 4.4: Diagrama de estados do *end device*

O estado inicial DEVICE_ACTIVE_IDLE_STATE faz a leitura da temperatura, no caso do sensor, ou do estado da porta, no caso do atuador, e se prepara para enviar os dados para o coordenador da rede, atualizando o estado para DEVICE_MESSAGE_SENDING_STATE.

Se a mensagem foi enviada com sucesso, o dispositivo se prepara para entrar no modo de economia de bateria e muda seu estado para DEVICE_SLEEP_PREPARE_STATE. Caso contrário, o processo recomeça do estado inicial.

Antes que o dispositivo fique inativo por certo período de tempo, é preciso saber se ele pode ser desligado. Módulos sensores não possuem restrições a essa condição, porém os atuadores não podem simplesmente desligar, haja vista que desligariam também o ar condicionado nesse processo. O estado

DEVICE_SLEEP_PREPARE_STATE funciona então para fazer essa diferenciação e colocar o dispositivo para ficar inativo durante 10 segundos. Para o sensor, antes de hibernar, os módulos abertos são fechados. Para o atuador, o dispositivo apenas não faz nada durante esse período de tempo.

O acionamento dos aparelhos é feito quando uma mensagem do coordenador chega com destino a um dos atuadores, que verifica se é necessário ligar ou desligar o sistema de refrigeração correspondente. Esse sinal é enviado para a porta ADC_INPUT_1, que controla um transistor para ligar ou desligar os contatos de um relé de estado sólido que aciona o sistema de compressão dos equipamentos de ar condicionado.

4.6 DISPOSITIVO JUMPER

O *jumper* é o dispositivo que permite o controle de duas redes simultaneamente. Seu diagrama de estados é mostrado na Figura 4.5.

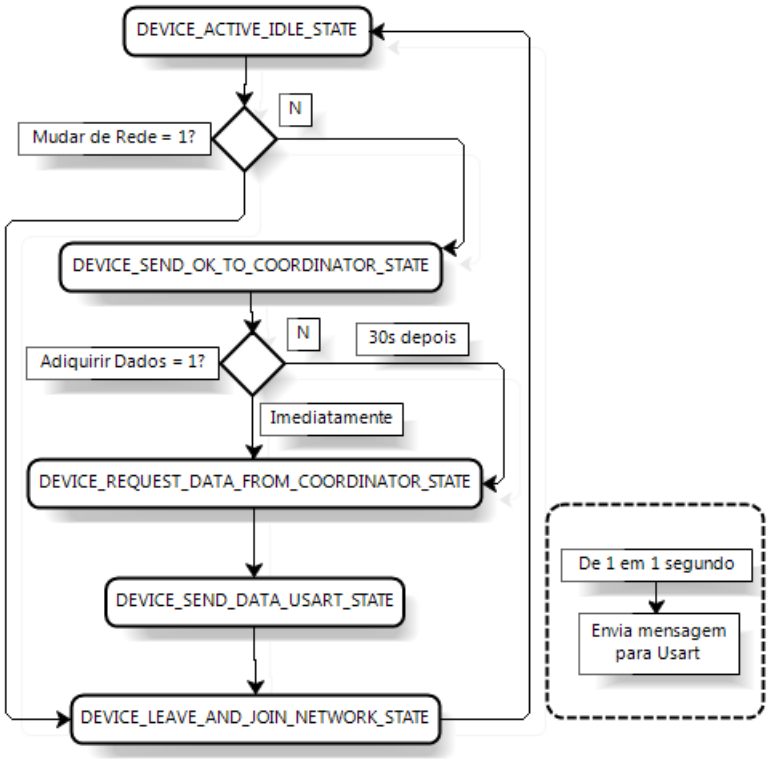

Figura 4.5: Diagrama de estados do *jumper*

Note que existe um processo rodando em segundo plano na aplicação do *jumper*. Esse processo, que não representa um estado porque na verdade trabalha com interrupções, envia uma mensagem padrão para o software supervisório para que a comunicação não seja perdida. Nessa mensagem, existe uma *flag* para dizer se algum dado da rede está sendo transmitido no momento. Para essas mensagens que são usadas afim de manter a comunicação serial ativa essa *flag* é desabilitada.

Ao invés de começar a explicação pelo estado inicial, como foi feito até agora, inicia-se a explicação do diagrama pelo estado DEVICE_SEND_OK_TO_COORDINATOR_STATE para que depois fiquem mais claras as funções do estado inicial.

A idéia de mandar uma mensagem de comunicação que não carrega nenhum dado surgiu para que o *jumper* ficasse monitorando constantemente a rede e verificando se existia algum problema com o coordenador da mesma, apesar do fato da rede Zigbee trabalhar sem mensagens periódicas de verificação. Sem essa mensagem, o *jumper* não se comunica com dispositivo nenhum durante 30 segundos, um período de tempo considerado grande para uma parte tão importante da solução apresentada. Inicialmente, a proposta era fazer essa troca de mensagens a cada 5 segundos, mas dois problemas foram encontrados e a lógica foi alterada. O primeiro problema era que, ao chegar ao

segundo 30, o dispositivo eventualmente não enviava o pedido de dados porque a verificação da rede era feita primeiro. O segundo problema era bem mais grave, dizia respeito ao processo de confirmação da entrega de pacotes na rede, que vez ou outra fazia o dispositivo esperar uma resposta que demorava muito para chegar e por causa disso comprometia a seqüência do programa, por vezes causando a perda de comunicação com o software supervisório e comprometendo todo o teste.

Uma vez que a mensagem de comunicação funcionava bem quando não era usada repetidamente, escolheu-se mudar sua função para que a lógica desenvolvida não fosse perdida. No estado DEVICE_SEND_OK_TO_COORDINATOR_STATE, uma mensagem de comunicação é enviada e, ao receber a resposta, o dispositivo verifica se é necessário enviar mais dados ao SIMULINK e faz a atualização imediata para o próximo estado que faz o pedido dos dados para o coordenador, DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE. Caso não exista essa necessidade, espera-se passar 30 segundos para que a atualização do estado seja feita.

Ao se encontrar no estado DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE, o dispositivo envia uma mensagem para o coordenador com uma *flag* para avisar que está na hora de enviar as últimas leituras para o supervisório. O coordenador responde colocando os dados da temperatura dos sensores e o estado da porta dos atuadores em uma única mensagem e o estado é atualizado para DEVICE_SEND_DATA_USART_STATE.

Nesse momento, o dispositivo envia seqüencialmente mensagens para o MATLAB contendo os dados recebidos, cada mensagem correspondendo a um componente da rede. Agora, a *flag* de comunicação de dados é habilitada. O SIMULINK nesse momento trata as mensagens recebidas (Capítulo 3.7) e o *jumper* muda seu estado para DEVICE_LEAVE_AND_JOIN_NETWORK_STATE.

Para mudar de rede, inverte-se a *flag* que indica a qual rede deseja-se conectar (*conectedPanA*) e muda-se o estado da aplicação topo para APP_CHANGING_NETWORK_STATE. Ao atualizar o estado, o programa principal não vai mais redirecionar o código diretamente para o dispositivo *jumper*, mas sim se desconectar da rede atual. Quando esse processo termina, a própria aplicação topo muda seu estado para APP_NETWORK_JOINING_STATE, que então muda os valores da PAN com base na *flag* que foi alterada anteriormente. Ao fazer um novo processo de iniciar a comunicação, agora o módulo buscará a outra rede e se conectará a ela. Quando toda essa parte é finalizada, a aplicação topo retorna ao estado APP_NETWORK_JOINED_STATE e novamente redireciona as tarefas a maquina de estados do *jumper*.

Nesse momento, o estado inicial do dispositivo, DEVICE_ACTIVE_IDLE_STATE, é chamado. Quando ocorre uma mudança da rede A para a rede B, ou vice-versa, o estado inicial atualiza o estado para DEVICE_SEND_OK_TO_JUMPER_STATE por causa da função que verifica se é necessário o envio de dados ao supervisório e volta-se ao que foi explicado no começo. Acontece que o dispositivo permanece no estado inicial enquanto não ocorre a interrupção que muda o estado para DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE. No estado inicial, o dispositivo verifica se a última mensagem para a rede foi enviada, fazendo uma nova transmissão caso seja necessário, e monitora as *flags* enviadas pelo MATLAB para atualizar os dados ou mudar de rede.

Caso seja preciso atualizar os dados da outra rede, o dispositivo muda o estado para DEVICE_LEAVE_AND_JOIN_NETWORK_STATE e envia o pedido de dados assim que chegar a confirmação da mensagem de comunicação gerada em DEVICE_SEND_OK_TO_JUMPER_STATE. Caso seja preciso atualizar os dados da rede que se encontra, o dispositivo muda o estado para DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE e segue o fluxo da máquina de estados.

Um detalhe que não ficou claro diz respeito às *flags adiquirirDados* e *mudarPan*, enviadas pelo software supervisório. O único momento no qual elas são habilitadas é quando o supervisório envia o pedido de dados, mas nada foi dito sobre quando elas são desabilitadas. A *flag adiquirirDados* é desabilitada assim que um pedido de requisição de dados é feito, da mesma forma que *mudarPan* é desabilitada quando o *jumper* troca de rede.

5 DESENVOLVIMENTO E ANÁLISE DOS RESULTADOS

5.1 ROTINA DE TESTES

O sistema que foi descrito nos capítulos anteriores foi colocado para funcionar e uma rotina de testes foi desenvolvida. Primeiramente tentou-se fazer um teste simples de duração de 5 horas no qual dois sinais de referência eram enviados para as redes, o mesmo sinal para ambas. Nas primeiras 2h30, a temperatura de referência é igual a 23°C e depois muda-se a referência para 22°C. O objetivo é verificar se o controle da temperatura foi bem feito.

Um segundo teste foi desenvolvido com o objetivo de medir o consumo de energia do sistema em comparação a um controle feito por outros sistemas. No caso, para o ambiente do LAVSI, o controle é feito pelo próprio ar condicionado Split e, para o caso da Sala de Reunião, o controle é feito pelo sistema *FullGauge*, que controla o ar condicionado híbrido.

5.2 ANÁLISE DOS RESULTADOS

5.2.1 PRIMEIRO TESTE

No dia 06 de setembro foi feito o primeiro teste. O controle da temperatura começou a ser feito às 15 horas e terminou às 20 horas.

Mostra-se como foi o resultado do teste no LAVSI na Figura 5.1. O resultado da rede da Sala de Reunião é apresentado na Figura 5.2. Nos gráficos, apresentam-se as leituras da temperatura, a temperatura de referência e, na parte de baixo, quando o atuador estava ligado ou desligado. As cores azul e vermelha do gráfico são para identificar os pares sensor/atuador.

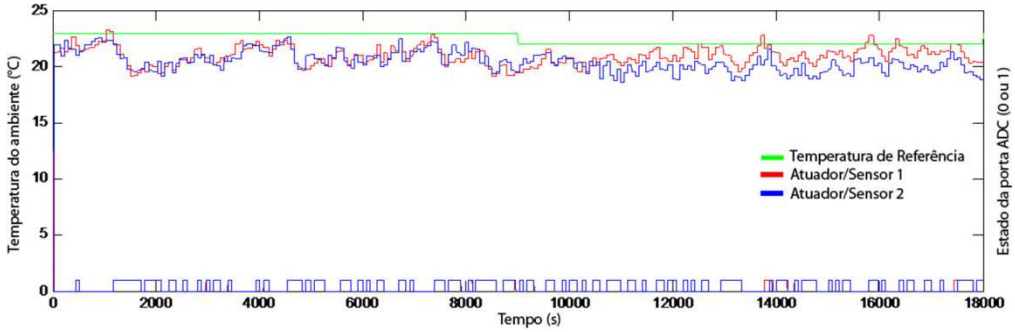

Figura 5.1: Gráfico do desempenho do sistema - LAVSI

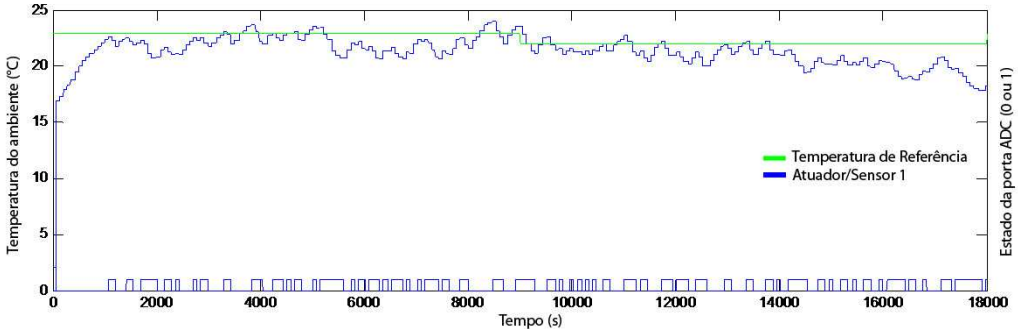

Figura 5.2: Gráfico do desempenho do sistema - Sala de Reunião

Logo a primeira vista, vê-se que a temperatura do ambiente não acompanhou a temperatura de referência com uma histerese de 1°C. Esse foi um problema constatado apenas depois que o sistema começou a funcionar de fato, ou seja, quando foi possível a obtenção dos dados e sua correta apresentação no SIMULINK. O mesmo ocorre para a rede da Sala de Reunião, o que não é nenhuma surpresa já que o código de todos os módulos Meshbean é o mesmo. Mesmo com esse problema no controle, resolveu-se continuar com a análise porque não havia mais tempo hábil para que se encontrasse o erro na lógica e fazer novas mudanças nas redes.

Uma vez que não foi atingido o controle esperado, parte-se para uma análise do controle que foi realmente feito, visto que o coordenador envia comandos para ligar e desligar os aparelhos de refrigeração. Para verificar se o sistema responde de alguma forma ao degrau de temperatura que ocorre com 2h30 de teste, fez-se a média das leituras de temperatura de cada par sensor/atuador das redes.

A Figura 5.3 mostra como foi o controle em uma parte do LAVSI enquanto a Figura 5.4 mostra o controle da outra parte. Não é necessário mais um gráfico para a rede da Sala de Reunião, que só possui dois dispositivos e sua resposta foi mostrada na Figura 5.2.

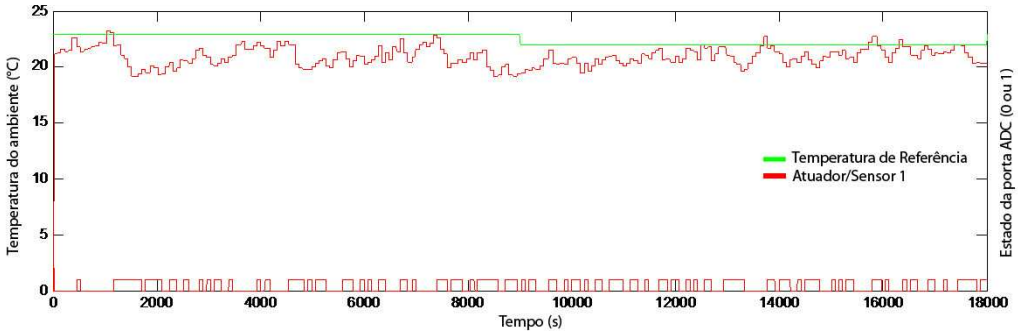

Figura 5.3: Gráfico do par sensor/atuador 1 - LAVSI

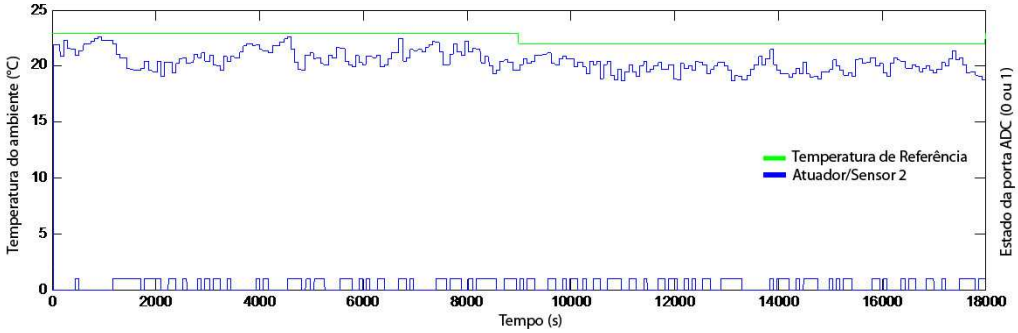

Figura 5.4: Gráfico do par sensor/atuador 2 - LAVSI

Mostram-se as médias calculadas por intermédio da função *mean* do MATLAB na Tabela 3.

Tabela 3: Temperaturas médias

Sensor	Média na primeira metade do teste	Média na segunda metade do teste
Sensor 1 - LAVSI	20,95°C	20,04 °C
Sensor 2 - LAVSI	20,94 °C	19,92 °C
Sensor 1 – Sala de Reunião	22,14 °C	20,78 °C

Claramente, a temperatura média diminuiu da primeira metade do teste para a segunda, indicando que, apesar de errada, algum tipo de referência é atualizada nos coordenadores das redes. A priori, não é um problema difícil de ser solucionado. Pode ser um erro no valor da temperatura de referência, no

processo que determina quando enviar ao atuador o sinal para ligar ou desligar o ar condicionado ou algum outro problema de lógica.

Mesmo que não tenha sido o controle ideal, pode-se verificar que foi o monitoramento de duas redes diferentes foi um sucesso, com indícios de que conseguiu-se mudar a temperatura de referência.

5.2.2 SEGUNDO TESTE

O segundo teste foi realizado no dia 07 de setembro. O controle foi feito das 10 horas até as 18 horas, totalizando 8 horas de teste. Previamente, no dia 05 de setembro, havia sido realizado um teste semelhante, com a mesma duração, mas com os sistemas que já estavam presentes funcionando. A idéia é verificar se o sistema desenvolvido gasta menos energia elétrica do que os outros.

A resposta do sistema para ambas as redes é mostrada na Figura 5.5 e Figura 5.6. Novamente, o controle não foi bem realizado. A Referência aqui não é a mesma para os dois modelos, visava-se mostrar o sistema permite o ajuste em separado do *setpoint* dos ambientes.

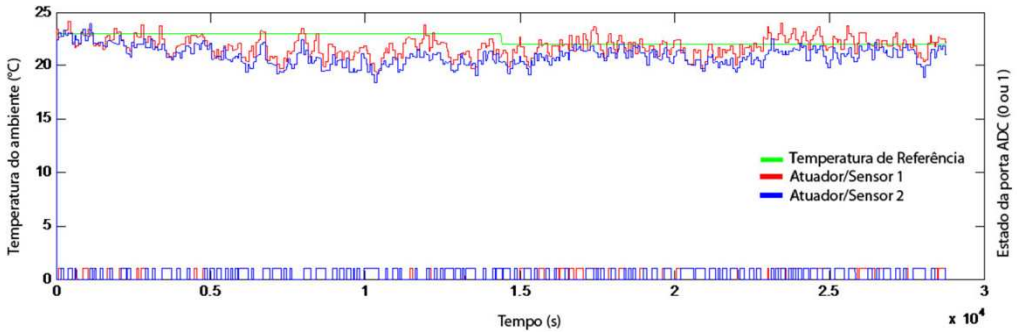

Figura 5.5: Gráfico do desempenho no 2° teste - LAVSI

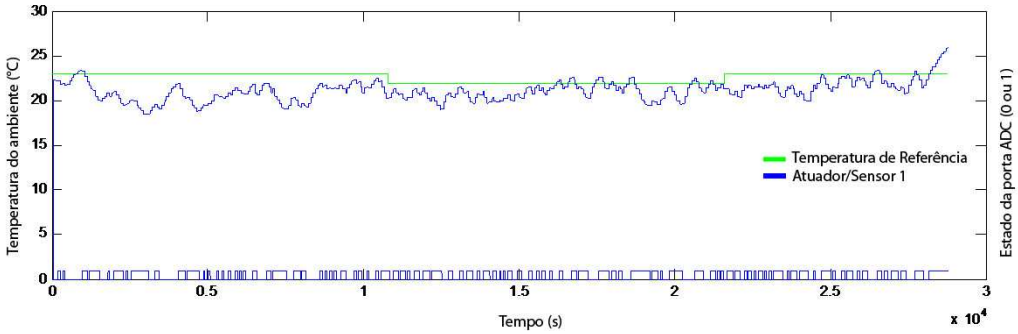

Figura 5.6: Gráfico do desempenho no 2° teste - Sala de Reunião

A análise é feita da seguinte forma: verifica-se quanto é o consumo da energia no começo e no final do teste e comparam-se os resultados obtidos. Os dados foram colocados na Tabela 4. É importante dizer que a medição de energia no LAVSI foi feita apenas com relação a um ar condicionado, não em relação aos dois usados nos testes, pela simples razão de haver apenas um dispositivo hábil para fazer a leitura da energia consumida.

Tabela 4: Consumo de energia dos sistemas

Ambiente	Sistema de controle	Medição inicial (kWh)	Medição final (kWh)	Consumo total (kWh)
LAVSI	Nenhum	1469,16	1503,59	34,433
LAVSI	Sistema proposto	1517,33	1545,54	28,21
Sala de Reunião	FullGauge	2895,14	2906,06	10,92
Sala de Reunião	Sistema proposto	2919,14	2944,64	25,50

A partir dos dados, pode-se fazer uma comparação entre os dois sistemas. A Tabela 5 mostra como foi o desempenho do sistema de controle proposto no projeto em relação ao outros dois apresentados para comparação.

Tabela 5: Comparação entre os sistemas de controle

Ambiente	Consumo do sistema pré-existente (kWh)	Consumo do sistema proposto (kWh)	Diferença
LAVSI	34,433	28,21	-17,8 %
Sala de Reunião	10,92	25,50	+133,5 %

Observando a diferença encontrada, vemos que uma solução de controle, mesmo que não ideal como a obtida após os testes, ainda é melhor que a opção de não haver solução de controle alguma. O consumo de energia de um ar condicionado Split foi 17,8% menor.

Na sala de reunião, o consumo na verdade foi maior, aumentou 133,5%. Isso aconteceu porque o sistema se propôs a fazer a atuação do ar condicionado híbrido apenas no modo convencional de refrigeração, atuando apenas no sistema de compressão, que gasta muita energia. O controle feito pelo sistema *FullGauge*, que é o sistema de controle do fabricante, atua também no modo evaporativo do equipamento, que gasta bem menos energia por não utilizar o condensador do sistema. Na verdade, essa diferença de consumo já era esperado.

6 CONCLUSÕES E TRABALHOS FUTUROS

Após os testes, verificam-se dois pontos que devem ser discutidos mais a fundo. Se por um lado a integração das redes de sensores de diferentes ambientes foi bem sucedida, a solução de controle proposta não funcionou como se esperava.

Vê-se que o sistema faz o monitoramento de duas redes distintas com sucesso, inclusive com indícios de que foi possível mudar o valor da temperatura de referência. Essa era a principal preocupação quando se desenvolveu o projeto e pode-se dizer que o resultado obtido foi bastante satisfatório.

Já o controle não funcionou porque não houve tempo para que se descobrisse qual parte da lógica embarcada não estava de acordo com o esperado. A verdade é que fazer o software supervisório funcionar demorou mais do que o previsto por causa de pequenos detalhes no SIMULINK e, quando de fato foi possível ver a resposta do sistema, já não havia muito mais tempo para grandes mudanças no código dos módulos.

Devido ainda ao problema no controle da temperatura, não foi possível fazer uma análise do grau de satisfação das pessoas no ambiente controlado para verificar se o conforto térmico seria atendido. Com a temperatura de referência média mais baixa do que a esperada, os ambientes acabaram ficando muito frios e desconfortáveis na maior parte do tempo. Logo, nenhum teste relacionado ao conforto térmico foi feito porque sem um controle de temperatura eficaz, não haveria a possibilidade de o teste ser bem sucedido.

Voltando à estratégia de integração das redes, é preciso dizer que a proposta oferecida não foi a única solução encontrada. Para manter o controle no próprio supervisório, o monitoramento deveria ser feito por meio de outras redes, como a TCP/IP ou MS-TP, que seriam integradas ao software supervisório. Já existem empresas que fazem automação predial com monitoramento remoto em uma central, distante do ambiente controlado usando configurações desse tipo. Ainda, essa configuração poderia ser feita no caso de o controle ser embarcado em um determinado dispositivo. Essa última proposta visa tornar o software supervisório apenas um elemento supervisório, mas exige um gasto maior com desenvolvimento de um hardware que possibilite estratégias de controle mais complexas, além de um maior custo em desenvolvimento de software.

Devem-se observar também as possíveis limitações encontradas pela solução que foi proposta. Usar um nó para monitorar muitas redes de sensores exige que todas elas estejam ao alcance desse nó, o que em si gera uma limitação física. Para sistemas maiores ou distantes entre si, haveria a necessidade do desenvolvimento de um dispositivo roteador que conseguisse se comunicar com o *jumper*. Pode-se então dizer que a solução proposta é simples e atende os requisitos do ambiente controlado, porém se encontra incompleta.

Os trabalhos futuros podem, portanto, explorar diferentes vertentes para fazer uma melhor integração das redes. Obviamente, o primeiro ponto a ser melhorado diz respeito ao controle embarcado, que precisa funcionar para que o projeto como um todo faça sentido. Pode-se desenvolver um controle embarcado liga-desliga que efetivamente funcione ou se propor uma solução mais avançada, como o controle por lógica *fuzzy*, que se adéqua melhor às necessidades encontradas do que um controle PID, mais complexo, mas que pode ser utilizado com o auxílio de um controle adaptativo.

Outra vertente é no monitoramento de redes maiores ou mais distantes, que necessitariam de um módulo roteador para tornar possível a comunicação de todos os dispositivos da rede. Existe uma meta de fazer a automação de todo o primeiro andar do prédio SG 11, portanto não falta local para a implementação da rede.

Uma terceira possibilidade de projeto é na parte do protocolo Zigbee. Até agora, nenhuma rede Zigbee que foi projetada seguiu de fato o que é dito na norma. Como conseqüência, cada novo projeto de rede desenvolve uma nova aplicação em BitCloud que é incapaz de se comunicar com as aplicações anteriores ou futuras, tornando ainda mais difícil a integração de diferentes redes.

Por fim, seria interessante o uso de um software supervisório mais apropriado para a validação do modelo proposto. O SIMULINK é uma ferramenta extraordinária para simular processos, mas não foi desenvolvida para aplicações em tempo real. Até existe uma biblioteca para aplicações em *real time*, que foi desenvolvida recentemente, mas existe uma preocupação se essa seria a melhor solução. Softwares como o *ActionView* poderiam ser usados para solucionar esse problema.

7 REFERÊNCIAS BIBLIOGRÁFICAS

[1] GISLASON, D. **Zigbee Wireless Networking**. Newnes, 2008.

[2] FARAHANI, S. **Zigbee Wireless Networks and Transceivers**. Newnes, 2008.

[3] WILSON, J. **Sensor Technology Handbook**. Newnes, 2005.

[4] Flores, J. L. O. **Sistema Híbrido de Climatização Visando Conforto Térmico e Eficiência Energética**. Dissertação de mestrado, Faculdade de Tecnologia, Universidade de Brasília, 2009.

[5] QUEIROZ, R.B. e AZEVEDO, R.C.A. **Rede de Sensores Sem Fio Para Automação Predial Com Módulos Meshbean**. Trabalho de graduação, Faculdade de Tecnologia, Universidade de Brasília, 2009

[6] SUCH, R.N. **Implementação de um Controlador Fuzzy em um Sistema de Refrigeração Híbrido Através de uma Rede de Sensores Zigbee**. Trabalho de graduação, Faculdade de Tecnologia, Universidade de Brasília, 2009

[7] BRITO, M.V.T e SILVA, R.P.F. **Implementação de BACnet Sobre Zigbee para Rede de Automação Predial Wireless**. Trabalho de graduação, Faculdade de Tecnologia, Universidade de Brasília, 2008

[8] ÁVILA, A.G. e SALOIO, B.H. **Instrumentação e Controle de um Sistema de Ar Condicionado Híbrido**. Trabalho de graduação, Faculdade de Tecnologia, Universidade de Brasília, 2009

[9] ÁGUAS, M.P.N. **Conforto térmico**. Módulo da disciplina de mestrado "Métodos instrumentais em energia e ambiente". Instituto Superior Técnico. Lisboa, 2000/2001

[10] MARKOV, D. **Standards in Thermal Comfort**. In: ANNUAL INTERNATIONAL COURSE: VENTILATION AND INDOOR CLIMATE, Sofia, 2002. P. Stankov, 2002 (Ed) pp. 147-157

[11] MeshNetics. **ZigBit™ Development Kit 1.3 User's Guide**, Doc. S-ZDK-451~01 v.1.10, MeshNetics, Dezembro de 2007

[12] Atmel. **BitCloud User Guide**, Atmel, Maio de 2009

[13] MeshNetics. **BitCloud SDK for ATZB-DK-24 / ATZB-DK-A24 / ATZB-DK-900**, v1.6.0, BitCloud Stack Documentation, MeshNetics, Junho de 2008

[14] Deitel, P.J. **C++ How to Program**. Prentice Hall, 2005

[15] Atmel. **Datasheet ZigBit™ 2.4 GHz Wireless Modules ATZB-24-A2/B0**, Atmel, Junho de 2009

[16] National Semiconductor. **Datasheet LM35**, National Semiconductor, Dezembro de 2000

[17] National Semiconductor. **Datasheet LM73**, National Semiconductor, LM73, Maio de 2009

[18] **ZigBee Resource Guide**, versão Primavera de 2008, disponível em:
http://www.zigbeeresourceguide.com/images/ZigBee_RG_2008.pdf

[19] **What's New in Simulink in R2010a**, disponível em:
http://www.mathworks.com/products/new_products/Simulink_R2010a.pdf

ANEXO I

AI.1 SOFTWARE – PROJETOFINAL.C

```
/*****************************************************************************//***
  \file lowpower.c

  \brief Lowpower application: Coordinator part of application implementation.

  \author
    Atmel Corporation: http://www.atmel.com \n
    Support email: avr@atmel.com

  Copyright (c) 2008 , Atmel Corporation. All rights reserved.
  Licensed under Atmel's Limited License Agreement (BitCloudTM).

  \internal
    History:
    - (junho/2009) Adaptado do original por
      Raphael Carvalho de Almeida Azevedo
              Rodrigo Bertuol de Queiroz
            - (julho/2010) Modificado ainda por
              Davi Stoll Evangelista
*****************************************************************************/
#include <projetoFinal.h>
/*****************************************************************************
  Variáveis Globais
*****************************************************************************/
//Variável para determinar o estado do módulo
AppState_t appState = APP_INITIAL_STATE;
//Variável para determinar o estado do dispositivo
AppDeviceState_t appDeviceState = DEVICE_ACTIVE_IDLE_STATE;
// Variável para guardar o valor lido no DIP Switch
uint8_t sliders;
/* Variável para definir em qual PAN se conectar
true  - PAN A (0x000A)
false - PAN B (0x000B) */
bool conectedPanA = true;
/*****************************************************************************
  Variáveis Locais
*****************************************************************************/
// Endpoint simple descriptor (ZDO endpoint descriptor)
SimpleDescriptor_t simpleDescriptor = {APP_ENDPOINT, APP_PROFILE_ID, 1, 1, 0, 0 , NULL, 0,
NULL};
//Variável para configuração do timer da rede
static HAL_AppTimer_t networkTimer;
//Parâmetro para propriedades da rede (endpoint e indicação de mensagem)
static APS_RegisterEndpointReq_t apsRegisterEndpointReq;
//Parâmetro para a requisição de criação da rede
static ZDO_StartNetworkReq_t  zdoStartNetworkReq;
// Parâmetro para a troca de redes
static ZDO_ZdpReq_t leaveReq;
//Parâmetro da rede para o tipo de dispositivo
static DeviceType_t appDeviceType;
static DeviceTypes_t whichDevice;
/*****************************************************************************
  Static functions
*****************************************************************************/
static void initApp(void);
#ifdef _BUTTONS_
static void buttonReleased(uint8_t);
```

```
#endif
static void startNetwork(void);
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *);
static void startingNetworkTimerFired(void);
static void APS_DataInd(APS_DataInd_t *);
void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *);
static void zdpLeaveResp(ZDO_ZdpResp_t *);
void ZDO_BindIndication(ZDO_BindInd_t *);
/*****************************************************************************
  Implementação
*****************************************************************************/
/*****************************************************************************
  Description: Task handler da aplicação (principal)
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
void APL_TaskHandler(void) {
  appTaskHandler(APP_EVENT_NULL, NULL);
}

void appTaskHandler(AppEvent_t event, void *param) {
        //Estados comuns a todos os módulos
        switch (appState)
        {
                //Estado inicial após reset
                case APP_INITIAL_STATE:
                {
                        initApp();                      // Função para iniciar os módulos
                        break;
                }
                //Estado para iniciar a rede
                case APP_NETWORK_JOINING_STATE:
                {
                        startNetwork();                 // Função para iniciar a rede
                        break;
                }
                //Estado em que a rede foi criada
                case APP_NETWORK_JOINED_STATE:
                {
                        //Se estiver configurado para coordenador
                        #ifdef _COORDINATOR_
                          if (COORDINATOR == whichDevice)
                                appCoordinatorTaskHandler(); //Task handler do coordenador
                        #endif

                        // Se estiver configurado para jumper
                        #ifdef _ENDDEVICE_
                          if (JUMPER == whichDevice)
                                appJumperTaskHandler(); //Task handler do end-device
                        #endif

                        //Se estiver configurado para end-device
                        #ifdef _ENDDEVICE_
                          if (END_DEVICE == whichDevice)
                                appEndDeviceTaskHandler(); //Task handler do end-device
                        #endif
                        break;
                }

                case APP_CHANGING_NETWORK_STATE:
                {
                  ZDO_MgmtLeaveReq_t                    *zdpLeaveReq                    =
&leaveReq.req.reqPayload.mgmtLeaveReq;
```

```
switch(event)
{
  case APP_PROCESS:
    appOffLed(APP_NETWORK_STATUS_LED);
    leaveReq.ZDO_ZdpResp =  zdpLeaveResp;
    leaveReq.reqCluster = MGMT_LEAVE_CLID;
    leaveReq.dstAddrMode = EXT_ADDR_MODE;
    leaveReq.dstExtAddr = 0;
    zdpLeaveReq->deviceAddr = 0;
    zdpLeaveReq->rejoin = 0;
    zdpLeaveReq->removeChildren = 1;
    zdpLeaveReq->reserved = 0;
    ZDO_ZdpReq(&leaveReq);
    break;

  case APP_LEAVE_DONE:
    if (ZDO_SUCCESS_STATUS == ((ZDO_ZdpResp_t *)param)->respPayload.status) {
      appState = APP_STOP_STATE;
    }
    SYS_PostTask(APL_TASK_ID);
    break;

  case APP_NETWORK_STATE_UPDATED:
  {
    ZDO_MgmtNwkUpdateNotf_t *updateParam = param;

    switch (updateParam->status)
    {
      case ZDO_NETWORK_LOST_STATUS:
        appState = APP_STOP_STATE;
        break;
      case ZDO_NETWORK_LEFT_STATUS:
        appState = APP_NETWORK_JOINING_STATE;
        SYS_PostTask(APL_TASK_ID);
        break;
      default:
        break;
    }
    break;
  }

  case APP_STOP_STATE:
  {
    switch(event)
    {
      case APP_NETWORK_STATE_UPDATED:
      {
        ZDO_MgmtNwkUpdateNotf_t *updateParam = param;

        switch (updateParam->status)
        {
          case ZDO_NETWORK_STARTED_STATUS:
            appState = APP_NETWORK_JOINED_STATE;
            SYS_PostTask(APL_TASK_ID);
            break;

          case ZDO_NETWORK_LEFT_STATUS:
            appState = APP_NETWORK_JOINING_STATE;
            SYS_PostTask(APL_TASK_ID);
            break;

          default:
```

```
            break;
          }
         }
         default:
          break;
       }
       break;
     }

     default:
       break;
   }
   break;
                }
                default:
                break;
        }
}

/*****************************************************************************
  Description:  Função para iniciar os módulos e configurar parâmetros   na rede
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
static void initApp(void) {
        // Variáveis para formação da rede
//      ShortAddr_t nwkAddr; // Não é usado porque os endereços são definidos de maneira aleatória
//      uint64_t nwkExtPanId;
        uint32_t channelMaskId=1;
        bool nwkUniqueAddrId, rxOnWhenIdle;
// bool nwkPreDefinedPanId;

//      nwkPreDefinedPanId = true;
//  nwkExtPanId = extPanId;

        // Determines the channel on which the device may operate
        channelMaskId<<=CHANNEL_MASK;
        // Define stochastic addressing mode to be employed
        nwkUniqueAddrId = false;
        //Leitura dos DIP switches para determinar a função do dispositivo
        sliders = appReadSliders();

        if (sliders == 0) {
          //Se DIP switch igual a zero
         #ifdef _COORDINATOR_
                appDeviceType = DEVICE_TYPE_COORDINATOR;
                whichDevice = COORDINATOR;
                rxOnWhenIdle = true; // Habilitar recebimento em estado IDLE
                //Função para inicializar o coordenador (coordinator.c)
                appCoordinatorInit();
         #else
                return; // This device can not be coordinator
         #endif // _COORDINATOR_
        }
        else { if (sliders == 7) {
          //Se DIP switch maior ou igual a 7
         #ifdef _ENDDEVICE_
                appDeviceType = DEVICE_TYPE_END_DEVICE;
                whichDevice = JUMPER;
                rxOnWhenIdle = true; // Habilitar recebimento em estado IDLE
                //Função para inicializar o router (router.c)
                appJumperInit();
         #else
```

```
            return; // This device can not be router
           #endif // _ENDDEVICE_
          }
  else {
    //Se DIP switch diferente de zero ou 7 (1, 2, 3, 4, 5 ou 6)
   #ifdef _ENDDEVICE_
    appDeviceType = DEVICE_TYPE_END_DEVICE;
                    whichDevice = END_DEVICE;
    rxOnWhenIdle = false; // Não habilitar recebimento em estado IDLE
    //Função para inicializar o end-device (enddevice.c)
    appEndDeviceInit();
   #else
    return; // This device can not be end device
   #endif // _ENDDEVICE_
          } }

          {
            uint8_t numBlocos;
            numBlocos = 2;
   #ifdef _APS_FRAGMENTATION_
            CS_WriteParameter(CS_APS_MAX_BLOCKS_AMOUNT_ID, &numBlocos);        // Maxima
quantidade de fragmentos de uma mensagem de rede
           #endif //_APS_FRAGMENTATION_
          }
          // Parâmetros para configurar a rede (definidos em configServer.c)
 #ifndef _MAC2_ // Na verdade não é definido em lugar algum, mas assim fica igual ao configServer.c
  CS_WriteParameter(CS_CHANNEL_MASK_ID, &channelMaskId);       // Determines the channel on
which the device may operate
//   CS_WriteParameter(CS_NWK_PREDEFINED_PANID_ID, &nwkPreDefinedPanId); // Usado para
que o cooredenador use uma PANID fixa
//   CS_WriteParameter(CS_NWK_EXT_PANID_ID, &nwkExtPanId);                    // Determines the
extended PAN ID of a network where a device may start without having to join
  CS_WriteParameter(CS_NWK_UNIQUE_ADDR_ID, &nwkUniqueAddrId); /* If set to true, the
CS_NWK_ADDR will be used as the device's short address\n
                                        Otherwise, the stochastic addressing mode will be employed
*/
  CS_WriteParameter(CS_DEVICE_TYPE_ID, &appDeviceType);                // Type of device
(DEVICE_TYPE_COORDINATOR, DEVICE_TYPE_ROUTER, DEVICE_TYPE_END_DEVICE)
  CS_WriteParameter(CS_RX_ON_WHEN_IDLE_ID, &rxOnWhenIdle);      // Whether the stack is to
enable its receiver during idle periods
 #endif // _MAC2_
//  A parte abaixo foi comentada para que o valor seja encontrado no chip UID
//  {
//    uint64_t uidId;
//    uidId = (uint64_t) sliders;
//    CS_WriteParameter(CS_UID_ID, &uidId); // Should be unique for each device in a network.
//  }

          //Função para liberar o funcionamento dos LEDs
          appOpenLeds();
          //Função para aguardar a liberção do botão para atualizar o estado
          appOpenButtons(NULL, buttonReleased);

          appOnLed(APP_RECEIVING_STATUS_LED);  // Indica que a placa esta ligada

          //Atualiza o estado e espera o botão 1 do meshbean ser apertado
          appState = INITIAL_APP_STATE;
          SYS_PostTask(APL_TASK_ID);
}

/*************************************************************************
  Description: Função chamada quando um botão do MeshBean é liberado
  Parameters:  Número do botão pressionado
```

```
            (KEY1 as BSP_KEY0 - Inicializar a rede,
             KEY2 as BSK_KEY1 - Acordar e enviar dados - end device)
  Returns:    none
******************************************************************************/
#ifdef _BUTTONS_
static void buttonReleased(uint8_t button) {
          switch (button)
          {
                    //Se o botão 1 da MeshBean for pressionado e liberado
                    case BSP_KEY0:
                    {
                              //Se o módulo estiver no estado inicial
                              if (INITIAL_APP_STATE == appState) {
                                appOffLed(APP_RECEIVING_STATUS_LED);
                                        //Próximo estado do módulo
                                        appState = APP_NETWORK_JOINING_STATE;
                                        //Atualiza o estado retornando para o Task Handler da aplicação
(principal)
                                        SYS_PostTask(APL_TASK_ID);
                              }
                              if (JUMPER == whichDevice) {  // Jumper
                                appDeviceState                                                    =
DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE;
                                        SYS_PostTask(APL_TASK_ID);
                              }
                              break;
                    }
                    //Se o dispositivo estiver configurado como end-device
                    //E o botão 2 da MeshBean for pressionado e liberado

                    case BSP_KEY1:
                    {
                      if (JUMPER == whichDevice) {  // Jumper
                                        appDeviceState = DEVICE_LEAVE_AND_JOIN_NETWORK_STATE;
                                        SYS_PostTask(APL_TASK_ID);
                              }
                     #ifdef _ENDDEVICE_
                              //Se o módulo estiver em modo sleep
                              if    ((DEVICE_SLEEP_STATE    ==    appDeviceState)&&(whichDevice    ==
END_DEVICE)) { // EndDevice Sensor
                                        //Próximo estado do dispositivo
                                        appDeviceState = DEVICE_AWAKENING_STATE;
                                        //Atualiza o estado retornando para o Task Handler da
                                                  //aplicação (principal)
                                        SYS_PostTask(APL_TASK_ID);
                              #endif
                              }
                      break;
                    }

                    default:
                      break;
          }
}
#endif

/******************************************************************************
  Description: Requisição de inicialização da rede
  Parameters:  nenhum.
  Returns:     nenhum.
******************************************************************************/
static void startNetwork(void) {
  uint64_t extPanId;
```

```
        PanId_t nwkPanId;

  if (conectedPanA == true) { // Conectar-se a PAN A
    extPanId = ENDERECO_EXT_PAN_ID_A;
    nwkPanId = ENDERECO_PAN_ID_A;
  }
  else {                    // Conectar-se a PAN B
    extPanId = ENDERECO_EXT_PAN_ID_B;
    nwkPanId = ENDERECO_PAN_ID_B;
  }

  CS_WriteParameter(CS_EXT_PANID_ID, &extPanId);          /* Extended PAN ID of the network to
which the device should join.
                                                          For a coordinator, this is an extended PAN ID of a
network to be started. */
  CS_WriteParameter(CS_NWK_PANID_ID, &nwkPanId);          // Endereço PAN_ID simples.
Análogo ao anterior.


        //Configuração de um timer para piscar o LED da rede (verde)
        //Parâmetros do timer
        networkTimer.interval = APP_JOINING_INDICATION_PERIOD;
        networkTimer.mode     = TIMER_REPEAT_MODE;
        networkTimer.callback = startingNetworkTimerFired;
        //Solicitação para a camada HAL iniciar o timer
        HAL_StartAppTimer(&networkTimer);
        //Parâmetro para solicitação de criação da rede
        zdoStartNetworkReq.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
        //Solicitação para a camada ZDO para inicializar a rede
        ZDO_StartNetworkReq(&zdoStartNetworkReq);
}

/***************************************************************************
  Description: Função de confirmação de criação da rede
  Parameters:  confirmInfo - informação de confirmação da camada ZDO
  Returns:     nenhum.
***************************************************************************/
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confInfo) {
  //Desabilita o timer iniciado na função startNetwork
  HAL_StopAppTimer(&networkTimer);
  //Se a camada retornar status de sucesso (rede criada)
  if (ZDO_SUCCESS_STATUS == confInfo->status) {
    //Próximo estado do módulo
    appState = APP_NETWORK_JOINED_STATE;
    //Próximo estado do dispositivo
    appDeviceState = DEVICE_ACTIVE_IDLE_STATE;
    //Liga o LED correspondente da rede (verde)
    appOnLed(APP_NETWORK_STATUS_LED);
    //Registra propriedades da rede (endpoint e indicação de mensagem)
    apsRegisterEndpointReq.simpleDescriptor = &simpleDescriptor;
    apsRegisterEndpointReq.APS_DataInd = APS_DataInd;
    //Envia os parâmetros para a camada APS
    APS_RegisterEndpointReq(&apsRegisterEndpointReq);
  }
  //Atualiza o estado retornando para o Task Handler da aplicação (principal)
  SYS_PostTask(APL_TASK_ID);
}

/***************************************************************************
  Description: Função de callback do timer configurado para a rede
  Parameters:  nenhum.
  Returns:     nenhum.
***************************************************************************/
```

```
static void startingNetworkTimerFired(void) {
        //Função de inversão do estado do LED
        appToggleLed(APP_NETWORK_STATUS_LED);
}

/**************************************************************************
  Description:  Função que indica que o módulo recebeu mensagem na rede
  Parameters:   ind - Indicação primitiva da camada APS
  Returns:      nenhum.
**************************************************************************/
static void APS_DataInd(APS_DataInd_t* ind) {
        if (sliders == 0) {
          //Se DIP switch igual a zero
         #ifdef _COORDINATOR_
                appCoordinatorDataInd(ind);
         #else
                return; // This device can not be coordinator
         #endif // _COORDINATOR_
        }
        else { if (sliders == 7) {
          //Se DIP switch maior ou igual a 7
         #ifdef _ENDDEVICE_
                appJumperDataInd(ind);
         #else
          return; // This device can not be router
         #endif // _ENDDEVICE_
        }
  else {
    //Se DIP switch diferente de zero ou 7 (1, 2, 3, 4, 5 ou 6)
   #ifdef _ENDDEVICE_
    appEndDeviceDataInd(ind);
   #else
    return; // This device can not be end device
   #endif // _ENDDEVICE_
        } }
}

/**************************************************************************
  Description: Função que notifica alteração na rede
  Parameters:  ZDO_MgmtNwkUpdateNotf_t *nwkParams - Notificação
  Returns:     nenhum.
**************************************************************************/
void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *nwkParams) {
        ZDO_StartNetworkConf_t conf;

        if (APP_CHANGING_NETWORK_STATE == appState) {
          appTaskHandler(APP_NETWORK_STATE_UPDATED, nwkParams);
        }

        if (ZDO_NETWORK_STARTED_STATUS == nwkParams->status) {
                conf.status = ZDO_SUCCESS_STATUS;
                ZDO_StartNetworkConf(&conf);
        }
        else { if (ZDO_NETWORK_LEFT_STATUS == nwkParams->status) {
                appState = APP_NETWORK_JOINING_STATE;
                SYS_PostTask(APL_TASK_ID);
        } }
}

/**************************************************************************
  Description: Função que notifica alteração na rede
  Parameters:  ZDO_ZdpResp_t *zdpResp - The response means that the command has
                        been received successfully but not
```

```
                                precessed yet
  Returns:     nenhum.
*****************************************************************************/
static void zdpLeaveResp(ZDO_ZdpResp_t *zdpResp) {
  appTaskHandler(APP_LEAVE_DONE, zdpResp);
}

#ifdef _BINDING_
/*****************************************************************************
  Description: Stub for ZDO Binding Indication
  Parameters:  bindInd - indication
  Return:      none
 *****************************************************************************/
void ZDO_BindIndication(ZDO_BindInd_t *bindInd) {
  (void)bindInd;
}

/*****************************************************************************
  Description: Stub for ZDO Unbinding Indication
  Parameters:  unbindInd - indication
  Return:      none
 *****************************************************************************/
void ZDO_UnbindIndication(ZDO_UnbindInd_t *unbindInd) {
  (void)unbindInd;
}
#endif //_BINDING_

/*****************************************************************************
  Description: Função para debug do status recebido apos uma mensagem ser enviada
  Parameters:  APS_Status_t statusEnvio - Status retornado pela função APS_DataReq
  Returns:     statusReceivedInt - valor inteiro do enum.
*****************************************************************************/
int16_t debugStatusReceived(APS_Status_t statusReceivedTemplate) {
  int16_t statusReceivedInt;

  statusReceivedInt = (int16_t)statusReceivedTemplate;

  switch (statusReceivedTemplate)
  {
    case APS_SUCCESS_STATUS:
    {
      statusReceivedInt = 0x00;   //!<SUCCESS
      break;
    }
    case APS_ASDU_TOO_LONG_STATUS:
    {
      statusReceivedInt = 0xa0;   //!<ASDU_TOO_LONG
      break;
    }
    case APS_DEFRAG_DEFERRED_STATUS:
    {
      statusReceivedInt = 0xa1;   //!<DEFRAG_DEFERRED
      break;
    }
    case APS_DEFRAG_UNSUPPORTED_STATUS:
    {
      statusReceivedInt = 0xa2;   //!<DEFRAG_UNSUPPORTED
      break;
    }
    case APS_ILLEGAL_REQUEST_STATUS:
    {
      statusReceivedInt = 0xa3;   //!<ILLEGAL_REQUEST
      break;
```

```
}
case APS_INVALID_BINDING_STATUS:
{
  statusReceivedInt = 0xa4;   //!<INVALID_BINDING
  break;
}
case APS_INVALID_GROUP_STATUS:
{
  statusReceivedInt = 0xa5;   //!<INVALID_GROUP
  break;
}
case APS_INVALID_PARAMETER_STATUS:
{
  statusReceivedInt = 0xa6;   //!<INVALID_PARAMETER
  break;
}
case APS_NO_ACK_STATUS:
{
  statusReceivedInt = 0xa7;   //!<NO_ACK
  break;
}
case APS_NO_BOUND_DEVICE_STATUS:
{
  statusReceivedInt = 0xa8;   //!<NO_BOUND_DEVICE
  break;
}
case APS_NO_SHORT_ADDRESS_STATUS:
{
  statusReceivedInt = 0xa9;   //!<NO_SHORT_ADDRESS
  break;
}
case APS_NOT_SUPPORTED_STATUS:
{
  statusReceivedInt = 0xaa;   //!<NOT_SUPPORTED
  break;
}
case APS_SECURED_LINK_KEY_STATUS:
{
  statusReceivedInt = 0xab;   //!<SECURED_LINK_KEY
  break;
}
case APS_SECURED_NWK_KEY_STATUS:
{
  statusReceivedInt = 0xac;   //!<SECURED_NWK_KEY
  break;
}
case APS_SECURITY_FAIL_STATUS:
{
  statusReceivedInt = 0xad;   //!<SECURITY_FAIL
  break;
}
case APS_TABLE_FULL_STATUS:
{
  statusReceivedInt = 0xae;   //!<TABLE_FULL
  break;
}
case APS_UNSECURED_STATUS:
{
  statusReceivedInt = 0xaf;   //!<UNSECURED
  break;
}
case APS_UNSUPPORTED_ATTRIBUTE_STATUS:
{
```

```
      statusReceivedInt = 0xb0;   //!<UNSUPPORTED_ATTRIBUTE
      break;
    }
  }
  return statusReceivedInt;
}

// eof lowpower.c
```

AI.2 SOFTWARE – PROJETOFINAL.H

```
/**************************************************************************//***
  \file lowpower.h

  \brief Lowpower application header file.

  \author
    Atmel Corporation: http://www.atmel.com \n
    Support email: avr@atmel.com

  Copyright (c) 2008 , Atmel Corporation. All rights reserved.
  Licensed under Atmel's Limited License Agreement (BitCloudTM).

  \internal
    History:
    - (junho/2009) Adaptado do original por
      Raphael Carvalho de Almeida Azevedo
              Rodrigo Bertuol de Queiroz
            - (julho/2010) Modificado ainda por
              Davi Stoll Evangelista
******************************************************************************/

#ifndef _PROJETOFINAL_H
#define _PROJETOFINAL_H

/******************************************************************************
                  Includes section
******************************************************************************/
#include <configServer.h>                       // Config Server header
#include <appFramework.h>                         // Main stack types
#include <zdo.h>                                // Main ZDO header
#include <aps.h>                                // Main APS header
#include <appTimer.h>                             // Application timer header
#include <sensors.h>                            // BSP sensors header
#include <usart.h>                              // HAL USART header
#include <stdio.h>
#include <stdlib.h>
#include <adc.h>
#include <gpio.h>
#include <avr/io.h>
#include <taskManager.h>
#include <util/delay.h>
#include <halInterrupt.h>

#ifdef TEST_NETWORK
  #undef _BUTTONS_
  #undef _SLIDERS_
#endif // TEST_NETWORK

#ifdef _SLIDERS_
  #include "sliders.h"
```

```
#endif // _SLIDERS_

#ifdef _LEDS_
  #include "leds.h"
#endif // _LEDS_

#ifdef _BUTTONS_
  #include "buttons.h"
#endif // _BUTTONS_

#ifndef _APS_FRAGMENTATION_
#define _APS_FRAGMENTATION_
#endif // _APS_FRAGMENTATION_

// O define abaixo é porque a rede completa exigia 9 dispositivos, mas tinha apenas 7.
// O coordenador então também faz a função de um sensor
#define _FALTAM_MESHBEANS_

/*****************************************************************************
                    Defines section
*****************************************************************************/
// Endereços das PANs que serão usados. Mudar variável conectPanA para mudar o PANID
#define ENDERECO_EXT_PAN_ID_A           0x000000000000000ALL
#define ENDERECO_PAN_ID_A               0x000A
#define ENDERECO_EXT_PAN_ID_B           0x000000000000000BLL
#define ENDERECO_PAN_ID_B               0x000B

// Determines the channel on which the device may operate.
#define CHANNEL_MASK                    25

#define APP_MAX_DATA_SIZE               APS_MAX_ASDU_SIZE

// USART Tx buffer size
#ifndef APP_USART_TX_BUFFER_SIZE
#define APP_USART_TX_BUFFER_SIZE        500
#endif

// Temporary data received via network buffer size
#ifndef APP_TMP_DATA_BUFFER_SIZE
#define APP_TMP_DATA_BUFFER_SIZE        500
#endif

#ifndef APP_JOINING_INDICATION_PERIOD
#define APP_JOINING_INDICATION_PERIOD       500L        // Period of blinking during starting
network
#endif

#define APP_ENDPOINT                    2       // Endpoint will be used
#define APP_PROFILE_ID                  1       // Profile Id will be used
#define APP_CLUSTER_ID                  1       // Cluster Id will be used

// Leds aliases definition
#define APP_NETWORK_STATUS_LED          LED_GREEN   // Network status LED
#define APP_RECEIVING_STATUS_LED        LED_YELLOW  // Data receiving status LED
#define APP_SENDING_STATUS_LED          LED_RED     // Data transmission status LED

#define USART_RX_BUFFER_LENGTH          64
#define USART_TX_BUFFER_LENGTH          64

// Define usado porque faltavam 2 dispositivos para ter-se uma rede completa.
// Com o define, o coordenador atua também como sensor1
#define _FALTA_DISPOSITIVO_MESHBEAN_
```

```
typedef enum
{
  APP_INITIAL_STATE,                        // Application initial state (after Power On or Reset)
  APP_START_WAIT_STATE,                     // Waiting while the Button0 was not pressed
  APP_NETWORK_JOINING_STATE,                // Joining network state
  APP_NETWORK_JOINED_STATE,                 // Network available
  APP_ERROR_STATE,                          // Error state
  APP_CHANGING_NETWORK_STATE,
  APP_STOP_STATE
} AppState_t;

typedef enum
{
  APP_EVENT_NULL,
  APP_PROCESS,
  APP_NETWORK_STATE_UPDATED,
  APP_LEAVE_DONE
} AppEvent_t;

typedef enum
{
  DEVICE_ACTIVE_IDLE_STATE,                 // Device is not in sleep state and the temperature
must be measured
  DEVICE_MEASURING_STATE,                   // Temperature measuring
  DEVICE_MESSAGE_SENDING_STATE,             // Current temperature sending to the
coordinator
  DEVICE_SLEEP_PREPARE_STATE,               // Message was sent successfully. Node
ready to sleep.
  DEVICE_SLEEP_STATE,                       // Actually sleep state
  DEVICE_AWAKENING_STATE,                   // Node was interrupted. Awakening.
  DEVICE_SEND_DATA_TO_JUMPER_STATE,         // Transmite dados salvo para o
dispositivo supervisório
  DEVICE_SEND_OK_TO_JUMPER_STATE,
  DEVICE_CONTROL_STATE,                     // Coordenador faz o controle sem supervisório
  DEVICE_SEND_OK_TO_COORDINATOR_STATE,
  DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE,   // Jumper faz a
requisição dos dados salvos no Coordenador
  DEVICE_SEND_DATA_USART_STATE,             // Jumper transmite dados para o
Matlab
  DEVICE_LEAVE_AND_JOIN_NETWORK_STATE       // Jumper se conecta a uma nova
PAN (Novo PAN ID)
} AppDeviceState_t;

typedef enum                                // Data transmission feature state
{
  APP_DATA_TRANSMISSION_IDLE_STATE,         // Data transmission was finished or(and)
not started yet
  APP_DATA_TRANSMISSION_BUSY_STATE          // Data transmission in progress
} AppDataTransmissionState_t;

typedef enum                                // Data transmission feature state
{
  ACTUATOR_1=0,                             // Igual a zero para facilitar código em MATLAB
  ACTUATOR_2,
  SENSOR_1,
  SENSOR_2,
  COORDINATOR,
  ROUTER,
  JUMPER,
  END_DEVICE
} DeviceTypes_t;

typedef struct
```

```
{
  int8_t              porta;
} PACK AppDeviceActuator_t;

typedef struct
{
  int16_t             temperature;
}  PACK AppDeviceSensor_t;

typedef struct
{
  AppDeviceSensor_t     sensor1;
  AppDeviceSensor_t     sensor2;
  AppDeviceActuator_t   actuator1;
  AppDeviceActuator_t   actuator2;
} PACK AppDataSaved_t;

typedef struct                                  // Application message
{
  int16_t             temperature;
  int8_t              porta;
  // Flag para o coordenador enviar dados para ojumper
  bool                requestData;
  bool                dataMessage;
  bool                appHandshake;
  bool                novaTemperaturaReferencia;
  /* Indica qual é o dispositivo que enviou a mensagem. Usado para não depender
  dos valores do shortAddress (que são aleatórios). Seu valor depende dos dispositivos na rede */
  DeviceTypes_t         dispositivo;
  AppDataSaved_t        dataSaved;          // Dados salvos que serão enviados
} PACK AppMessage_t;

typedef struct                                  // Aplication message buffer
{
  uint8_t             header[APS_ASDU_OFFSET];            // Auxilliary header (stack required)
  AppMessage_t          message;                          // Actually application message
  uint8_t                 footer[APS_AFFIX_LENGTH - APS_ASDU_OFFSET]; // Auxilliary footer (stack
required)
} PACK AppMessageBuffer_t;
/******************************************************************************
                Global variables section
******************************************************************************/
extern AppState_t appState;
extern AppDeviceState_t appDeviceState;
extern uint8_t sliders;
extern bool conectedPanA;
/******************************************************************************
                Functions' prototypes section
******************************************************************************/
extern void appTaskHandler(AppEvent_t, void *);
int16_t debugStatusReceived(APS_Status_t);

extern void appCoordinatorInit(void);
extern void appCoordinatorTaskHandler(void);
extern void appCoordinatorDataInd(APS_DataInd_t*);

extern void appJumperInit(void);
extern void appJumperTaskHandler(void);
extern void appJumperDataInd(APS_DataInd_t*);

extern  void appEndDeviceInit(void);
extern  void appEndDeviceTaskHandler(void);
extern  void appEndDeviceDataInd(APS_DataInd_t*);
```

```
extern void sendControlMessageActuator(DeviceTypes_t, bool);

#ifdef _LEDS_
  #define appOpenLeds() BSP_OpenLeds()
  #define appCloseLeds() BSP_CloseLeds()
  #define appOnLed(id) BSP_OnLed(id)
  #define appOffLed(id) BSP_OffLed(id)
  #define appToggleLed(id) BSP_ToggleLed(id)
#else
  #define appOpenLeds()
  #define appCloseLeds()
  #define appOnLed(id)
  #define appOffLed(id)
  #define appToggleLed(id)
#endif // _LEDS_

#ifdef _BUTTONS_
  #define appOpenButtons(pressed, released) BSP_OpenButtons(pressed, released)
  #define appCloseButtons() BSP_CloseButtons()
  #define appReadButtonsState() BSP_ReadButtonsState()
  #define INITIAL_APP_STATE  APP_START_WAIT_STATE
#else
  #define appOpenButtons(pressed, released)
  #define appCloseButtons()
  #define appReadButtonsState() 0
  #define INITIAL_APP_STATE  APP_NETWORK_JOINING_STATE
  #define BSP_KEY0 0
  #define BSP_KEY1 1
#endif // _BUTTONS_

#ifdef _SLIDERS_
  #define appReadSliders() BSP_ReadSliders()
#else
  #define appReadSliders() NWK_NODE_ADDRESS
#endif // _SLIDERS_

#endif // _PROJETOFINAL_H
// eof projetoFinal.h
```

AI.3 SOFTWARE – COORDINATOR.C

```
/**************************************************************************//***
  \file coordinator.c

  \brief Lowpower application: Coordinator part of application implementation.

  \author
    Atmel Corporation: http://www.atmel.com \n
    Support email: avr@atmel.com

  Copyright (c) 2008 , Atmel Corporation. All rights reserved.
  Licensed under Atmel's Limited License Agreement (BitCloudTM).

  \internal
    History:
    - (junho/2009) Adaptado do original por
      Raphael Carvalho de Almeida Azevedo
              Rodrigo Bertuol de Queiroz
            - (julho/2010) Modificado ainda por
              Davi Stoll Evangelista
```

```
*****************************************************************************/
#ifdef _COORDINATOR_
#include <projetoFinal.h>
/*****************************************************************************
  Variáveis Globais
 *****************************************************************************/
// Definidas no arquivo lowpower.h
/*
extern AppState_t appState;
extern AppDeviceState_t appDeviceState;
extern uint8_t sliders;
extern bool conectedPanA;
*/
/*****************************************************************************
  Variáveis Locais
 *****************************************************************************/
//Parâmetros e variáveis para mensagem na rede
static APS_DataReq_t apsDataReq;
static AppMessageBuffer_t appMessageBuffer; // Application message buffer
static AppDataTransmissionState_t appDataTransmissionState =
APP_DATA_TRANSMISSION_IDLE_STATE;
//Variáveis de controle
static int16_t tempRef;
static int16_t tempRefSuperior;
static int16_t tempRefInferior;
//Variáveis para os dados recebidos pela rede
static int16_t temp_sensor_1;
static int16_t temp_sensor_2;
//Variáveis de endereço na rede (destinatário)
static ShortAddr_t destAdress;
//Variáveis de endereço na rede (remetente)
static ShortAddr_t remetAdress;
//Variáveis de endereço na rede (end devices)
static ShortAddr_t addrsAtuador1=0;
static ShortAddr_t addrsAtuador2=0;
//Variável para status das portas dos atuadores
static int8_t status_atuador_1;
static int8_t status_atuador_2;
// Dispositivo de destino
static DeviceTypes_t deviceDest;
// Variavel para ser usada no envio de mensagens pela rede
static DeviceTypes_t device;
static AppDeviceState_t previousDeviceState;
#ifdef _FALTA_DISPOSITIVO_MESHBEAN_
static HAL_AdcParams_t adcParam;
static int16_t temperatureAdc=0;
static int open_succes=1;
static int read_succes=1;
#endif //_FALTA_DISPOSITIVO_MESHBEAN_
/*****************************************************************************
  Local functions
*****************************************************************************/
static void doControl(void);
static void messageInit(void);
static void APS_DataConf(APS_DataConf_t *);
static void sendMessageToJumper(void);
#ifdef _FALTAM_MESHBEANS_
static void coordinatorReadData(void);
static void adcInit(void);
static void readTemperature(void);
static void temperatureCalback(void);;
#endif
/*****************************************************************************
```

```
  Description:  Rotina de inicialização do coordenador
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
void appCoordinatorInit(void) {
  appMessageBuffer.message.dispositivo = COORDINATOR;
  tempRef = 2300;   // Temperatura de referência inicial de 23°C
          tempRefSuperior = tempRef + 50;// Histerese
          tempRefInferior = tempRef - 50;// Histerese
#ifdef _FALTAM_MESHBEANS_
  adcInit();
#endif
          //Função para configurar os parâmetros de mensagem na rede
          messageInit();
}

/*****************************************************************************
  Description:  Task handler do coordenador
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
void appCoordinatorTaskHandler(void) {
          switch (appDeviceState)
          {
                    //Estado único do coordenador
                    case DEVICE_ACTIVE_IDLE_STATE:
                              break;   // Não existem funções a serem feitas antes de chegar alguma
mensagem da rede

             case DEVICE_CONTROL_STATE:
                    {
                      doControl();
                      break;
                    }

                    case DEVICE_SEND_OK_TO_JUMPER_STATE:
                    {
                      sendMessageToJumper();
                      break;
                    }

             case DEVICE_SEND_DATA_TO_JUMPER_STATE:
                    {
                      sendMessageToJumper();
                      break;
                    }

                    default: // ERRO, não deveria chegar nesse ponto
                      break;
          }
}

/*****************************************************************************
  Description:  Função que indica que o módulo recebeu mensagem na rede
  Parameters:   ind - Indicação primitiva da camada APS
  Returns:      nenhum.
*****************************************************************************/
void appCoordinatorDataInd(APS_DataInd_t* ind) {
          //Parâmetros da mensagem
          AppMessage_t *appMessage = (AppMessage_t *) ind->asdu;

          //Atribui a variável o endereço do remetente
          remetAdress = ind->srcAddress.shortAddress;
```

```
        if (JUMPER == appMessage->dispositivo) {
          if (appMessage->novaTemperaturaReferencia == true) {
            tempRef = appMessage->temperature;   // Temperatura de referência
            tempRefSuperior = tempRef + 50;      // Histerese
            tempRefInferior = tempRef - 50;      // Histerese
           }
          if (appMessage->appHandshake == true) {
            destAdress = remetAdress;                    // Endereço do Jumper
            appMessageBuffer.message.appHandshake = true;
            appDeviceState = DEVICE_SEND_OK_TO_JUMPER_STATE;
  SYS_PostTask(APL_TASK_ID);
          }
          // Mensagem para transmitir dados guardados para o jumper
          if (appMessage->requestData == true) {
            destAdress = remetAdress;                    // Endereço do Jumper
            appMessageBuffer.message.dataMessage = true;
            appDeviceState = DEVICE_SEND_DATA_TO_JUMPER_STATE;
  SYS_PostTask(APL_TASK_ID);
          }
        }

        //Verifica as fontes e atribui as variáveis correspondentes os dados recebidos

        if (conectedPanA == true) { // A rede A possui 4 dispositivos diferentes
 if (SENSOR_1 == appMessage->dispositivo) { //Sensor 1
                temp_sensor_1 = appMessage->temperature;
                device = SENSOR_1;
            appDeviceState = DEVICE_CONTROL_STATE;
            SYS_PostTask(APL_TASK_ID);
          }
          if (SENSOR_2 == appMessage->dispositivo) { //Sensor 2
                temp_sensor_2 = appMessage->temperature;
            device = SENSOR_2;
            appDeviceState = DEVICE_CONTROL_STATE;
            SYS_PostTask(APL_TASK_ID);
          }

          if (ACTUATOR_1 == appMessage->dispositivo) { //Atuador 1

            if (addrsAtuador1 != remetAdress)
              addrsAtuador1 = remetAdress;
                status_atuador_1 = appMessage->porta;
            device = ACTUATOR_1;
            appDeviceState = DEVICE_CONTROL_STATE;
            SYS_PostTask(APL_TASK_ID);
          }

          if (ACTUATOR_2 == appMessage->dispositivo) { //Atuador 2
            if (addrsAtuador2 != remetAdress)
              addrsAtuador2 = remetAdress;
                status_atuador_2 = appMessage->porta;
            device = ACTUATOR_2;
            appDeviceState = DEVICE_CONTROL_STATE;
            SYS_PostTask(APL_TASK_ID);
          }
}
else { // A rede B possui apenas 2 dispositivos diferentes
          if (SENSOR_1 == appMessage->dispositivo) { //Sensor 1
                temp_sensor_1 = appMessage->temperature;
            device = SENSOR_1;
            appDeviceState = DEVICE_CONTROL_STATE;
            SYS_PostTask(APL_TASK_ID);
```

```
            }
            if (ACTUATOR_1 == appMessage->dispositivo) { //Atuador 1
              if (addrsAtuador1 != remetAdress)
                addrsAtuador1 = remetAdress;
                      status_atuador_1 = appMessage->porta;
              device = ACTUATOR_1;
              appDeviceState = DEVICE_CONTROL_STATE;
              SYS_PostTask(APL_TASK_ID);
            }
  }
}

/*******************************************************************************
  Description: Função para fazer o controle e não usar o supervisório para tal
  Parameters:  nenhum.
  Returns:     nenhum.
*******************************************************************************/
static void doControl(void) {
  switch (device)
  {
    case ACTUATOR_1:
    {
      appMessageBuffer.message.dataSaved.actuator1.porta = status_atuador_1;
     #ifndef _FALTAM_MESHBEANS_
      appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
      SYS_PostTask(APL_TASK_ID);
     #else
      coordinatorReadData();
     #endif
      break;
    }

    case ACTUATOR_2:
    {
      appMessageBuffer.message.dataSaved.actuator2.porta = status_atuador_2;
      appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
      SYS_PostTask(APL_TASK_ID);
      break;
    }

    case SENSOR_1:
    {
      if (addrsAtuador1 == 0) {
                appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
                SYS_PostTask(APL_TASK_ID); // ERRO, Ainda não recebeu mensagens do Atuador
               }
      appMessageBuffer.message.dataSaved.sensor1.temperature = temp_sensor_1;
      if (temp_sensor_1 > tempRefSuperior) {
        deviceDest = ACTUATOR_1;
        sendControlMessageActuator(deviceDest,true);
      }
      if (temp_sensor_1 < tempRefInferior) {
        deviceDest = ACTUATOR_1;
        sendControlMessageActuator(deviceDest,false);
      }
      appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
      SYS_PostTask(APL_TASK_ID);
      break;
    }

    case SENSOR_2:
    {
      if (addrsAtuador2 == 0) {
```

```
            appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
            SYS_PostTask(APL_TASK_ID); // ERRO, Ainda não recebeu mensagens do Atuador
           }
     appMessageBuffer.message.dataSaved.sensor2.temperature = temp_sensor_2;
     if (temp_sensor_2 > tempRefSuperior) {
       deviceDest = ACTUATOR_2;
       sendControlMessageActuator(deviceDest,true);
     }
     if (temp_sensor_2 < tempRefInferior) {
       deviceDest = ACTUATOR_2;
       sendControlMessageActuator(deviceDest,false);
     }
     appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
     SYS_PostTask(APL_TASK_ID);
     break;
    }

    default:
            break;
  }
}

/*****************************************************************************
  Description: Função para enviar dados para o jumper
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
static void sendMessageToJumper(void) {
  messageInit();
  previousDeviceState = appDeviceState;
  appDeviceState = DEVICE_ACTIVE_IDLE_STATE;
  if (APP_DATA_TRANSMISSION_IDLE_STATE == appDataTransmissionState) { // If previous data
transmission was finished
    appDataTransmissionState = APP_DATA_TRANSMISSION_BUSY_STATE;     // Data transmission
entity is busy while sending not finished
    appOnLed(APP_SENDING_STATUS_LED);
    //Requisição para a camada APS enviar a mensagem
    APS_DataReq(&apsDataReq);
  }
}

/*****************************************************************************
  Description: Parâmetros de envio de mensagem para a rede
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
static void messageInit(void) {
          //Parâmetros dos dados na rede
          apsDataReq.dstAddrMode                    = APS_SHORT_ADDRESS;
          apsDataReq.dstAddress.shortAddress          = destAdress;
          apsDataReq.dstEndpoint                   = APP_ENDPOINT;
          apsDataReq.profileId                     = APP_PROFILE_ID;
          apsDataReq.clusterId                      = APP_CLUSTER_ID;
          apsDataReq.srcEndpoint                   = APP_ENDPOINT;
          apsDataReq.asduLength                     = sizeof (AppMessage_t);
          apsDataReq.asdu                          = (uint8_t *) &appMessageBuffer.message;
          apsDataReq.txOptions.acknowledgedTransmission = 1;
          apsDataReq.radius                        = 0;
  apsDataReq.txOptions.fragmentationPermitted   = 1;
          apsDataReq.APS_DataConf                   = APS_DataConf;
}

/*****************************************************************************
```

```
  Description: Função de Confirmação do envio para a rede pela camada APS
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
static void APS_DataConf(APS_DataConf_t *conf) {
  conf = conf; // Evita warning
  appDataTransmissionState = APP_DATA_TRANSMISSION_IDLE_STATE; // Data transmission
entity is idle

  if (appMessageBuffer.message.appHandshake == true)
    appMessageBuffer.message.appHandshake = false;

  if (appMessageBuffer.message.dataMessage == true)
    appMessageBuffer.message.dataMessage = false;

          //Desliga o LED de envio para a rede (vermelho)
          appOffLed(APP_SENDING_STATUS_LED);
          SYS_PostTask(APL_TASK_ID);
}

/*****************************************************************************
  Description: Função para enviar o sinal de controle para atuador 1
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
void sendControlMessageActuator(DeviceTypes_t dest, bool status) {
          //Definindo o endereço de destino
          if (ACTUATOR_1 == dest)
            destAdress = addrsAtuador1;
          else if (ACTUATOR_2 == dest)
            destAdress = addrsAtuador2;
          else {
    appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
    SYS_PostTask(APL_TASK_ID); // ERRO, Ainda não recebeu mensagens do Atuador
  }

          if (status == true) // Liga atuador selecionado
            appMessageBuffer.message.porta = 1;
          else              // Desliga atuador selecionado
            appMessageBuffer.message.porta = 0;

          //Chama a função para iniciar os parâmetros da mensagem
          messageInit();
          appDeviceState = DEVICE_ACTIVE_IDLE_STATE;  // Retorna ao estado de espera
          //Se a rede estiver desocupada
          if(appDataTransmissionState == APP_DATA_TRANSMISSION_IDLE_STATE)  {
                    //Atualiza o estado da rede para ocupado
                    appDataTransmissionState = APP_DATA_TRANSMISSION_BUSY_STATE;
                    appOnLed(APP_SENDING_STATUS_LED);
                    //Requisição para a camada APS enviar a mensagem
                    APS_DataReq(&apsDataReq);
          }
          else
            SYS_PostTask(APL_TASK_ID); // ERRO, Tenta novamente
}

#ifdef _FALTAM_MESHBEANS_
/*****************************************************************************
  Description:  Função que atua como sensor
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void coordinatorReadData(void) {
```

```
  readTemperature();  // Leitura da temperatura
  device = SENSOR_1;
  SYS_PostTask(APL_TASK_ID); // Application task posting
}

/*****************************************************************************
  Description:  Init ADC
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void adcInit(void) {
  adcParam.bufferPointer = &temperatureAdc;
  adcParam.callback = temperatureCalback;
  adcParam.resolution = RESOLUTION_10_BIT;
  adcParam.sampleRate = ADC_9600SPS;
  adcParam.selectionsAmount = 1;
  adcParam.voltageReference = AREF;

  open_succes = HAL_OpenAdc(&adcParam);
}

/*****************************************************************************
  Description:  Leitura da temperatura do LM35
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void readTemperature(void) {
  read_succes = HAL_ReadAdc(HAL_ADC_CHANNEL1);
}

/*****************************************************************************
  Description:  Callback da função que lê a porta ADC, na qual foi colocado um
                LM35 para ler a temperatura.
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void temperatureCalback(void) {
  int temperaturaLidaInt;
  float temperaturaLidaFloat;

  if (read_succes == 0) {
    // Vref*10000/(Ganho*2^Resolução) = 1.25*10000/(3.2*1023)
    temperaturaLidaFloat = (float)temperatureAdc*3.818426;
    temperaturaLidaInt = (int)temperaturaLidaFloat;
    temp_sensor_1 = (int16_t)temperaturaLidaInt;
  }
  else // Leitura não foi feita porque o modulo ADC não foi aberto
    appOnLed(APP_RECEIVING_STATUS_LED); // ERRO!
}

#endif // _FALTAM_MESHBEANS_

#endif // _COORDINATOR_
// eof coordinator.c
```

AI.4 SOFTWARE – ENDDEVICE.C

```
/*****************************************************************************//***
  \file enddevice.c

  \brief Lowpower application: Coordinator part of application implementation.
```

```
  \author
    Atmel Corporation: http://www.atmel.com \n
    Support email: avr@atmel.com

  Copyright (c) 2008 , Atmel Corporation. All rights reserved.
  Licensed under Atmel's Limited License Agreement (BitCloudTM).

  \internal
    History:
    - (junho/2009) Adaptado do original por
      Raphael Carvalho de Almeida Azevedo
              Rodrigo Bertuol de Queiroz
            - (julho/2010) Modificado ainda por
              Davi Stoll Evangelista
******************************************************************************/
#ifdef _ENDDEVICE_
#include <projetoFinal.h>
/******************************************************************************
  Global variables
 ******************************************************************************/
// Definidas no arquivo lowpower.h
/*
extern AppState_t appState;
extern AppDeviceState_t appDeviceState;
extern uint8_t sliders;
extern bool conectedPanA;
*/
/******************************************************************************
  Local variables
 ******************************************************************************/
static                AppDataTransmissionState_t                appDataTransmissionState                =
APP_DATA_TRANSMISSION_IDLE_STATE;
static APS_DataReq_t apsDataReq;                          // APS Data Request primitive (for application
message sending)
static ZDO_SleepReq_t zdoSleepReq;                      // Request parameters for stack sleep
static ZDO_WakeUpReq_t zdoWakeUpReq;                        // Request parameters for stack awakening
static AppMessageBuffer_t appMessageBuffer;              // Application message buffer
static bool atuador;
static HAL_AppTimer_t sendDataTimer;
static ShortAddr_t destAdress;
HAL_AdcParams_t adcParam;
static int16_t temperatureAdc=0;
static int open_succes=1;
static int read_succes=1;
/******************************************************************************
  Static functions
******************************************************************************/
static void ZDO_SleepConf(ZDO_SleepConf_t *);          // Sleep confirmation handler
static void ZDO_WakeUpConf(ZDO_WakeUpConf_t *);        // Wake up confirmation handler
static void APS_DataConf(APS_DataConf_t *);            // Data transmission confirmation handler
static void messageInit(void);
static void openPeriphery(void);                       // Open LEDs and Temperature Sensor
static void closePeriphery(void);                      // Close LEDs and Temperature Sensor
static void sendMessage(void);                          // Send the application message
static void prepareToSleep(void);
static void adcInit(void);
static void readTemperature(void);
static void temperatureCalback(void);
static void timerInit(void);
static void sendData(void);
/******************************************************************************
  Description:  End device initialization routine
```

```
  Parameters:   none
  Returns:      none
******************************************************************************/
void appEndDeviceInit(void) {
  appMessageBuffer.message.requestData = false;   // requestData é flag do jumper da rede
  if (conectedPanA == true) { // A rede A possui 4 dispositivos diferentes
    //Analisa para ver se é atuador ou sensor
    sliders = appReadSliders();
            if ((sliders == 1)||(sliders == 2)) {
                    atuador = false;
      adcInit();
                    if (sliders == 1) appMessageBuffer.message.dispositivo = SENSOR_1;
                    else              appMessageBuffer.message.dispositivo = SENSOR_2;
            }
            if((sliders == 3)||(sliders == 4)) {
                    atuador = true;
      GPIO_ADC_INPUT_1_make_out();  // Habilita a saída ADC_INPUT_1 como porta
                    if (sliders == 3) appMessageBuffer.message.dispositivo = ACTUATOR_1;
                    else              appMessageBuffer.message.dispositivo = ACTUATOR_2;
            }
  }
  else { // A rede B possui apenas 2 dispositivos diferentes
    //Analisa para ver se é atuador ou sensor
    sliders = appReadSliders();
            if ((sliders == 1)||(sliders == 2)) {
                    atuador = false;
      adcInit();
                    appMessageBuffer.message.dispositivo = SENSOR_1;
            }
            else { // ((sliders == 3)||(sliders == 4))
                    atuador = true;
      GPIO_ADC_INPUT_1_make_out();  // Habilita a saída ADC_INPUT_1 como porta
                    appMessageBuffer.message.dispositivo = ACTUATOR_1;
            }
  }

          destAdress = 0; // Endereço do coordenador (único dispositivo que ele se comunica)
          messageInit();
}

/******************************************************************************
  Description:  Device common task handler
  Parameters:   none
  Returns:      none
******************************************************************************/
void appEndDeviceTaskHandler(void) {
  switch (appDeviceState)                       // Actual device state when one joined network
  {
    case DEVICE_ACTIVE_IDLE_STATE:                    // Device ready to temperature measuring
            {
      if (atuador == true)                      //Leitura da porta
          appMessageBuffer.message.porta = GPIO_ADC_INPUT_1_read();
      else
        readTemperature();  // Leitura da temperatura

      appDeviceState = DEVICE_MESSAGE_SENDING_STATE; //Switch device state to application
message sending
      SYS_PostTask(APL_TASK_ID); // Application task posting
              break;
            }

    case DEVICE_MESSAGE_SENDING_STATE:        // Message sending state
    {
```

```
    sendMessage();                    // Application message sending
    break;
   }

   case DEVICE_SLEEP_PREPARE_STATE:        // Prepare to sleep state
   {
          if (appReadButtonsState() & BSP_KEY1)
     SYS_PostTask(APL_TASK_ID);          // Still in current state.
    else
            prepareToSleep();                // Prepare to sleep
          break;
         }

   case DEVICE_AWAKENING_STATE:                              // Awakening state
         {
    zdoWakeUpReq.ZDO_WakeUpConf = ZDO_WakeUpConf; // ZDO WakeUp confirm handler
defining
    ZDO_WakeUpReq(&zdoWakeUpReq);             // ZDO WakeUp Request sending
    break;
        }

   case DEVICE_SLEEP_STATE:
   {
    // Não faz nada!
    break;
   }

   default:
    break;
  }
}

/**************************************************************************************
  Description:  Data intication handler
  Parameters:   ind - APS Data Indication primitive
  Returns:      none
**************************************************************************************/
void appEndDeviceDataInd(APS_DataInd_t* ind) {
  uint8_t control = 0;
  AppMessage_t *appMessage = (AppMessage_t *) ind->asdu;
  ind = ind; // Warning prevention

  if (COORDINATOR == appMessage->dispositivo) {
    control = appMessage->porta;

    if(control == 1) {
      appOnLed(APP_RECEIVING_STATUS_LED);
          GPIO_ADC_INPUT_1_set();
         }
         else {
      appOffLed(APP_RECEIVING_STATUS_LED);
          GPIO_ADC_INPUT_1_clr();
    }
  }
}

/**************************************************************************************
  Description: Parâmetros de envio de mensagem para a rede
  Parameters:  nenhum.
  Returns:     nenhum.
**************************************************************************************/
static void messageInit(void) {
 // Prepare APS Data Request
```

```
  apsDataReq.dstAddrMode                    = APS_SHORT_ADDRESS;                    // Short
addressing mode
  apsDataReq.dstAddress.shortAddress        = destAdress;                           // Destination
node short address
  apsDataReq.dstEndpoint                    = APP_ENDPOINT;                 // Destination endpoint
  apsDataReq.profileId                      = APP_PROFILE_ID;               // Profile ID
  apsDataReq.clusterId                      = APP_CLUSTER_ID;               // Destination cluster ID
  apsDataReq.srcEndpoint                    = APP_ENDPOINT;                 // Source endpoint
  apsDataReq.asduLength                     = sizeof (AppMessage_t);        // ASDU size
  apsDataReq.asdu                           = (uint8_t *) &appMessageBuffer.message; // ASDU pointer as
an application message
  apsDataReq.txOptions.acknowledgedTransmission = 1;                        // Acknowledged
transmission enabled
  apsDataReq.radius                         = 0;                            // Default radius
  apsDataReq.txOptions.fragmentationPermitted = 1;
  apsDataReq.APS_DataConf                   = APS_DataConf;                 // Confirm handler
}

/*****************************************************************************
  Description:  Open LEDs and Sensor
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void openPeriphery(void) {
  appOpenLeds();
  adcInit();
}

/*****************************************************************************
  Description:  Close LEDs and Sensor
  Parameters:   none
  Returns:      none
*****************************************************************************/
static void closePeriphery(void) {
  appOffLed(APP_NETWORK_STATUS_LED);
  appOffLed(APP_SENDING_STATUS_LED);
  appOffLed(APP_RECEIVING_STATUS_LED);
  appCloseLeds();

  HAL_CloseAdc();
}

/*****************************************************************************
  Description:  Data sent handler
  Parameters:   conf - APS Data Confirm primitive
  Returns:      none
*****************************************************************************/
static void APS_DataConf(APS_DataConf_t *conf) {
  appDataTransmissionState = APP_DATA_TRANSMISSION_IDLE_STATE; // Data transmission
entity is idle
  if (APS_SUCCESS_STATUS == conf->status)                  //Se a mensagem tiver sido enviada com
sucesso
    appDeviceState = DEVICE_SLEEP_PREPARE_STATE;           // Switch device state to prepare
for sleep
  else                                                     // Caso contrário
    appDeviceState = DEVICE_ACTIVE_IDLE_STATE;             // Data transmission wasn't
successfully finished. Retry.
          //Desliga o LED de envio para a rede (vermelho)
          appOffLed(APP_SENDING_STATUS_LED);
          SYS_PostTask(APL_TASK_ID);
}

/*****************************************************************************
```

```
  Description:  Init ADC
  Parameters:   none
  Returns:      none
******************************************************************************/
static void adcInit(void) {
  adcParam.bufferPointer = &temperatureAdc;
  adcParam.callback = temperatureCalback;
  adcParam.resolution = RESOLUTION_10_BIT;
  adcParam.sampleRate = ADC_9600SPS;
  adcParam.selectionsAmount = 1;
  adcParam.voltageReference = AREF;

  open_succes = HAL_OpenAdc(&adcParam);
}

/******************************************************************************
  Description:  Leitura da temperatura do LM35
  Parameters:   none
  Returns:      none
******************************************************************************/
static void readTemperature(void) {
  read_succes = HAL_ReadAdc(HAL_ADC_CHANNEL1);
}

/******************************************************************************
  Description:  Callback da função que lê a porta ADC, na qual foi colocado um
                LM35 para ler a temperatura.
  Parameters:   none
  Returns:      none
******************************************************************************/
static void temperatureCalback(void) {
  int temperaturaLidaInt;
  float temperaturaLidaFloat;

  if (read_succes == 0) {
    // Vref*10000/(Ganho*2^Resolução) = 1.25*10000/(3.2*1023)
    temperaturaLidaFloat = (float)temperatureAdc*3.818426;
    temperaturaLidaInt = (int)temperaturaLidaFloat;
    appMessageBuffer.message.temperature = (int16_t)temperaturaLidaInt;
  }
  else // Leitura não foi feita porque o modulo ADC não foi aberto
    appOnLed(APP_RECEIVING_STATUS_LED); // ERRO!
}

/******************************************************************************
  Description:  Send the application message
  Parameters:   none
  Returns:      none
******************************************************************************/
static void sendMessage(void) {
  if (APP_DATA_TRANSMISSION_IDLE_STATE == appDataTransmissionState) { // If previous data
transmission was finished
    appDataTransmissionState = APP_DATA_TRANSMISSION_BUSY_STATE;     // Data transmission
entity is busy while sending not finished
    appOnLed(APP_SENDING_STATUS_LED);
    //Requisição para a camada APS enviar a mensagem
    APS_DataReq(&apsDataReq);
  }
}

/******************************************************************************
  Description:  ZDO Sleep Confirm handler
  Parameters:   conf - ZDO Sleep Confirm primitive
```

```
  Returns:      none
******************************************************************************/
static void ZDO_SleepConf(ZDO_SleepConf_t *conf) {
  if (ZDO_SUCCESS_STATUS == conf->status) {  // Stack was slept successfully
          if(atuador == false)
             closePeriphery();                    // LEDs and Temperature Sensor closing
    appDeviceState = DEVICE_SLEEP_STATE;    // Device actually slept
  }
  else
    SYS_PostTask(APL_TASK_ID);              // Still in current state.
                                  // Application task posting for attempt repeat.
}

/******************************************************************************
  Description:  Prepare to sleep
  Parameters:   none
  Returns:      none
******************************************************************************/
static void prepareToSleep(void) {
  if(atuador == false) {
    zdoSleepReq.ZDO_SleepConf = ZDO_SleepConf;  // Sleep Confirm handler defining
    ZDO_SleepReq(&zdoSleepReq);                 // Sleep Request sending
  }
  else {
    appDeviceState = DEVICE_SLEEP_STATE;
    timerInit();
  }
}

/******************************************************************************
  Description:  Prepare to sleep
  Parameters:   none
  Returns:      none
******************************************************************************/
static void timerInit(void) {
  // Configure timer for send message to usart
  sendDataTimer.interval = 10000;
  sendDataTimer.mode     = TIMER_REPEAT_MODE;
  sendDataTimer.callback = sendData;
  HAL_StartAppTimer(&sendDataTimer);
}

/******************************************************************************
  Description:  Prepare to sleep
  Parameters:   none
  Returns:      none
******************************************************************************/
static void sendData(void) {
  HAL_StopAppTimer(&sendDataTimer);
  appDeviceState = DEVICE_ACTIVE_IDLE_STATE;
  SYS_PostTask(APL_TASK_ID);
}

/******************************************************************************
  Description: Device wakeup handler.
  Parameters: none
  Returns:    none
******************************************************************************/
static void wakeUpHandler(void) {
  appDeviceState = DEVICE_ACTIVE_IDLE_STATE;

  openPeriphery();
```

```
  // Turn network indication on
  appOnLed(APP_NETWORK_STATUS_LED);

  SYS_PostTask(APL_TASK_ID);
}

/**************************************************************************
  Description: End device wake up indication
  Parameters:  none.
  Returns:     nothing.
**************************************************************************/
void ZDO_WakeUpInd(void) {
  if (DEVICE_SLEEP_STATE == appDeviceState)
    wakeUpHandler();
}

/**************************************************************************
  Description: Wake up confirmation handler

  Parameters: conf - confirmation parameters

  Returns: nothing.
**************************************************************************/
void ZDO_WakeUpConf(ZDO_WakeUpConf_t *conf) {
  if (ZDO_SUCCESS_STATUS == conf->status)
    wakeUpHandler();
  else
    SYS_PostTask(APL_TASK_ID);
}

#endif // _ENDDEVICE_
// eof enddevice.c
```

AI.5 SOFTWARE – JUMPER.C

```
/**************************************************************************//***
 \file router.c

  \brief Lowpower application: Router part of application implementation.

  \author
    Davi Stoll Evangelista \n
    Support email: davi87@gmail.com

  \internal
    History:
    - Arquivo original criado em julho/2010
**************************************************************************/
#ifdef _ENDDEVICE_
#include <projetoFinal.h>
/**************************************************************************
  Variáveis Globais
 **************************************************************************/
// Definidas no arquivo lowpower.h
/*
extern AppState_t appState;
extern AppDeviceState_t appDeviceState;
extern uint8_t sliders;
extern bool conectedPanA;
*/
/**************************************************************************
```

```
  Variáveis Locais
 ******************************************************************************/
// Temporary data received via network buffer
static uint8_t tmpDataBuffer[APP_TMP_DATA_BUFFER_SIZE];
static uint8_t tmpDataBufferActualLength = 0;
static HAL_UsartDescriptor_t appUsartDescriptor;        // USART descriptor (required by stack)
static bool usartTxBusyFlag = false;                    // USART transmission transaction status
static uint8_t usartTxBuffer[APP_USART_TX_BUFFER_SIZE]; // USART Tx buffer
static APS_DataReq_t apsDataReq;                        // APS Data Request primitive (for application
message sending)
static AppMessageBuffer_t appMessageBuffer;             // Application message buffer
static                AppDataTransmissionState_t                appDataTransmissionState                =
APP_DATA_TRANSMISSION_IDLE_STATE;
static HAL_AppTimer_t sendMessageTimer;
static uint8_t rxBuffer[USART_RX_BUFFER_LENGTH];// read buffer
static uint8_t read_msg[10] = "";
static uint16_t readBytesCount=0;
static ShortAddr_t destAdress;
static ShortAddr_t remetAdress;
//Flags usadas durante a aplicação
static bool flagEnviouDados=true;
static bool flagEnviouOk=true;
static bool mudandoDeRede=false;
static bool justStarted=true;
// Variavel do contador
static uint8_t contador1s;
// Temperatura de referência recebida pela Usart
static int16_t tempRefPanA;
static int16_t tempRefPanB;
static int16_t tempRefUsartPanA;
static int16_t tempRefUsartPanB;
static int8_t msgDados=0;
static int8_t panA;
// Flags vindas do MATLAB para determinar se os dados enviados estão desatualizados
static uint8_t adiquirirDados;
static uint8_t mudarPan;
static AppDeviceState_t previousDeviceState;
static int16_t temperaturaSensor1;
static int16_t temperaturaSensor2;
static int8_t estadoPortaAtuador1;
static int8_t estadoPortaAtuador2;
/******************************************************************************
  Local functions
******************************************************************************/
static void messageInit(void);
static void sendMessageToCoordinator(void);
static void APS_DataConf(APS_DataConf_t *);
static void timerInit(void);
static void usartInit(void);
static void sendUsartMessage(void);
static void sendValuesToUsart(void);
inline void sendDataSavedToUsart(int8_t *, int8_t *, DeviceTypes_t *, int16_t *, int16_t *, int8_t *);
void sendDataToUsart(uint8_t *, uint8_t);
void appReadByteEvent(uint16_t);
static void usartWriteConf(void);
static void appLeaveNetwork(void);
/******************************************************************************
  Description:  Rotina de inicialização do roteador/jumper
  Parameters:   nenhum.
  Returns:      nenhum.
******************************************************************************/
void appJumperInit(void) {
  appMessageBuffer.message.appHandshake = false;
```

```
  appMessageBuffer.message.requestData = false;
  mudarPan = 0;
  contador1s = 0;
  if (justStarted == true) {
    justStarted = false;
    appMessageBuffer.message.dispositivo = JUMPER;
   // Temperatura de referência inicial de 23℃ enq uanto não for atualizada pela Usart
   tempRefPanA = 2300;
   tempRefPanB = 2300;
   adiquirirDados=0;
   destAdress = 0; // Endereço do coordenador (único dispositivo que ele se comunica)
   //Função para inicializar a USART
   usartInit();
   timerInit(); //Função para iniciar o timer para envio de dados para a USART
  }
}

/*****************************************************************************
  Description:  Task handler do coordenador
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
void appJumperTaskHandler(void) {
  switch (appDeviceState)
  {
          case DEVICE_ACTIVE_IDLE_STATE:
          {
            if (mudandoDeRede == false) {
       /* Verifica se a última mensagem requerindo os dados da rede foi bem
          sucedida ou se o jumper precisa tentar novamente */
       if (flagEnviouDados == false) {
         appDeviceState = DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE;
         SYS_PostTask(APL_TASK_ID);
       }
       /* Verifica se a última mensagem de verificação da rede foi bem
          sucedida ou se o jumper precisa tentar novamente */
       if (flagEnviouOk == false) {
         appDeviceState = DEVICE_SEND_OK_TO_COORDINATOR_STATE;
         SYS_PostTask(APL_TASK_ID);
       }
       // Verifica se é necessário mudar de Pan para obter novos dados da rede
       if ((adiquirirDados == 1)&&(mudarPan==1)) {
         appDeviceState = DEVICE_LEAVE_AND_JOIN_NETWORK_STATE;
         SYS_PostTask(APL_TASK_ID);
       }
       // Verifica se é necessário obter novos dados da rede
       if ((adiquirirDados == 1)&&(mudarPan==0)) {
         adiquirirDados = 0;
         appDeviceState = DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE;
         SYS_PostTask(APL_TASK_ID);
       }
      }
      else {
           appDeviceState = DEVICE_SEND_OK_TO_COORDINATOR_STATE;
           SYS_PostTask(APL_TASK_ID);
      }
                    break;
          }

    case DEVICE_SEND_OK_TO_COORDINATOR_STATE:
    {
      appMessageBuffer.message.appHandshake = true;
      sendMessageToCoordinator();
```

```
      break;
    }

    case DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE:
          {
            if (appMessageBuffer.message.appHandshake == false) {  // Garante que não há conflito
              appMessageBuffer.message.requestData = true;
                    sendMessageToCoordinator();
            }
                  break;
          }

    case DEVICE_SEND_DATA_USART_STATE:
          {
                  sendValuesToUsart();
                  appDeviceState = DEVICE_LEAVE_AND_JOIN_NETWORK_STATE;
      SYS_PostTask(APL_TASK_ID);
                  break;
          }

    case DEVICE_LEAVE_AND_JOIN_NETWORK_STATE:
          {
            if (APP_NETWORK_JOINED_STATE == appState) {
        if (conectedPanA == true)  // Esta conectado na PAN A
          conectedPanA = false;
        else                       // Esta conectado na PAN B
          conectedPanA = true;
       appLeaveNetwork();
                  }
                  break;
          }

          default:
            break;
  }
}

/**************************************************************************************
  Description:  Função que indica que o módulo recebeu mensagem na rede
  Parameters:   ind - Indicação primitiva da camada APS
  Returns:      nenhum.
**************************************************************************************/
void appJumperDataInd(APS_DataInd_t* ind) {
  AppMessage_t *appMessage = (AppMessage_t *) ind->asdu;

          //Atribui a variável o endereço do remetente
          remetAdress = ind->srcAddress.shortAddress;

  if ((appMessage->appHandshake==true)&&(COORDINATOR == appMessage->dispositivo)) {
    appMessageBuffer.message.appHandshake = false;
    appOffLed(APP_RECEIVING_STATUS_LED);
    if (mudandoDeRede == true) {
      mudandoDeRede = false;
      if (adiquirirDados == 1) {
        adiquirirDados = 0;
        appDeviceState = DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE;
        SYS_PostTask(APL_TASK_ID);
      }
    }
    appDeviceState = DEVICE_ACTIVE_IDLE_STATE;
    SYS_PostTask(APL_TASK_ID);
  }
```

```
  if ((appMessage->dataMessage==true)&&(COORDINATOR == appMessage->dispositivo)) {
    appMessageBuffer.message.requestData = false;
    appOffLed(APP_RECEIVING_STATUS_LED);
    if (conectedPanA == true) { // A rede A possui 4 dispositivos diferentes
      temperaturaSensor1 = appMessage->dataSaved.sensor1.temperature;
      temperaturaSensor2 = appMessage->dataSaved.sensor2.temperature;
      estadoPortaAtuador1 = appMessage->dataSaved.actuator1.porta;
      estadoPortaAtuador2 = appMessage->dataSaved.actuator2.porta;
    }
    else { // A rede B possui apenas 2 dispositivos diferentes
      temperaturaSensor1 = appMessage->dataSaved.sensor1.temperature;
      estadoPortaAtuador1 = appMessage->dataSaved.actuator1.porta;
    }
    appDeviceState = DEVICE_SEND_DATA_USART_STATE;
    SYS_PostTask(APL_TASK_ID);
  }
}

/*****************************************************************************
  Description: Parâmetros de envio de mensagem para a rede
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
static void messageInit(void) {
          //Parâmetros dos dados na rede
          apsDataReq.dstAddrMode                  = APS_SHORT_ADDRESS;
          apsDataReq.dstAddress.shortAddress          = destAdress;
          apsDataReq.dstEndpoint                  = APP_ENDPOINT;
          apsDataReq.profileId                    = APP_PROFILE_ID;
          apsDataReq.clusterId                    = APP_CLUSTER_ID;
          apsDataReq.srcEndpoint                  = APP_ENDPOINT;
          apsDataReq.asduLength                   = sizeof (AppMessage_t);
          apsDataReq.asdu                         = (uint8_t *) &appMessageBuffer.message;
          apsDataReq.txOptions.acknowledgedTransmission = 1;
          apsDataReq.radius                       = 0;
  apsDataReq.txOptions.fragmentationPermitted   = 1;
          apsDataReq.APS_DataConf                 = APS_DataConf;
}

/*****************************************************************************
  Description:  Mandar pedido para obtenção dos dados da rede salvos
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
static void sendMessageToCoordinator(void) {
  messageInit();
  if (conectedPanA == true) {
    if (tempRefPanA != tempRefUsartPanA) {
      appMessageBuffer.message.novaTemperaturaReferencia = true;
      appMessageBuffer.message.temperature = tempRefPanA;
    }
    else {
      appMessageBuffer.message.novaTemperaturaReferencia = false;
    }
  }
  else {
      if (tempRefPanB != tempRefUsartPanB) {
      appMessageBuffer.message.novaTemperaturaReferencia = true;
      appMessageBuffer.message.temperature = tempRefPanB;
    }
    else {
      appMessageBuffer.message.novaTemperaturaReferencia = false;
    }
```

```
  }
  appOnLed(APP_RECEIVING_STATUS_LED);             // Indica se o dispositivo recebeu ou não
resposta
  previousDeviceState = appDeviceState;
  appDeviceState = DEVICE_ACTIVE_IDLE_STATE;

  //Se a rede estiver desocupada
  if(appDataTransmissionState == APP_DATA_TRANSMISSION_IDLE_STATE)          {
          //Atualiza o estado da rede para ocupado
          appDataTransmissionState = APP_DATA_TRANSMISSION_BUSY_STATE;
          appOnLed(APP_SENDING_STATUS_LED);
          //Requisição para a camada APS enviar a mensagem
          APS_DataReq(&apsDataReq);
  }
}

/**************************************************************************
  Description: Função de Confirmação do envio para a rede pela camada APS
  Parameters:  nenhum.
  Returns:     nenhum.
**************************************************************************/
static void APS_DataConf(APS_DataConf_t *conf) {
  appDataTransmissionState = APP_DATA_TRANSMISSION_IDLE_STATE; // Data transmission
entity is idle
  if (APS_SUCCESS_STATUS == conf->status) {                  //Se a mensagem tiver sido enviada com
sucesso
    if (DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE == previousDeviceState)
      flagEnviouDados = true;
    if (DEVICE_SEND_OK_TO_COORDINATOR_STATE == previousDeviceState)
      flagEnviouOk = true;
  }
  else {                                          // Caso contrário
    if (DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE == previousDeviceState)
      flagEnviouDados = false;
    if (DEVICE_SEND_OK_TO_COORDINATOR_STATE == previousDeviceState)
      flagEnviouOk = false;
  }

  //Desliga o LED de envio para a rede (vermelho)
          appOffLed(APP_SENDING_STATUS_LED);
          SYS_PostTask(APL_TASK_ID);
}

/**************************************************************************
  Description:  Timer para envio de dados para a USART
  Parameters:   nenhum.
  Returns:      nenhum.
**************************************************************************/
void timerInit(void) {
          //Parâmetros para configuração do timer
          sendMessageTimer.interval = 1000;
          sendMessageTimer.mode     = TIMER_REPEAT_MODE;
          sendMessageTimer.callback = sendUsartMessage;
          //Requisição para a camada HAL iniciar o timer
          HAL_StartAppTimer(&sendMessageTimer);
}

/**************************************************************************
  Description: Função para determinar os parâmetros da USART e iniciar
  Parameters:  nenhum.
  Returns:     nenhum.
**************************************************************************/
void usartInit(void) {
```

```
  usartTxBusyFlag = false;

  appUsartDescriptor.tty            = APP_USART_CHANNEL;
  appUsartDescriptor.mode           = USART_MODE_ASYNC;
  appUsartDescriptor.baudrate       = USART_BAUDRATE_38400;
  appUsartDescriptor.dataLength     = USART_DATA8;
  appUsartDescriptor.parity         = USART_PARITY_NONE;
  appUsartDescriptor.stopbits       = USART_STOPBIT_1;
  appUsartDescriptor.rxBuffer       = rxBuffer;
  appUsartDescriptor.rxBufferLength  = USART_RX_BUFFER_LENGTH;
  appUsartDescriptor.txBuffer       = NULL; // If txBuffer is NULL then callback method is used
  appUsartDescriptor.txBufferLength  = 0;
  appUsartDescriptor.rxCallback     = appReadByteEvent;
  appUsartDescriptor.txCallback     = usartWriteConf;
  appUsartDescriptor.flowControl    = USART_FLOW_CONTROL_NONE;

  HAL_OpenUsart(&appUsartDescriptor);
}

/*****************************************************************************
  Description:  Função para enviar os dados recebidos para a USART
  Parameters:   nenhum.
  Returns:      nenhum.
*****************************************************************************/
static void sendUsartMessage(void) {
  //Variável para a mensagem da USART
          uint8_t str[100];
          uint8_t length;

          appOnLed(APP_SENDING_STATUS_LED);

          if (conectedPanA == true)
    panA = 1;          // Esta conectado na PAN A
  else
    panA = 0;          // Esta conectado na PAN B

  DeviceTypes_t jumper=JUMPER;
  int zero=0;

  //Disabilita todas as interrupções
  HAL_DisableInterrupts();

  length = sprintf((char *) str, "%d\n%d\n%d\n%d\n%d\n%d\r\n", (int)msgDados,
                (int)panA, (int)jumper, zero, zero, zero);

  //Função para enviar os dados para a USART
          sendDataToUsart(str, length);

          //Habilita todas as interrupções
          HAL_EnableInterrupts();

          appOffLed(APP_SENDING_STATUS_LED);

          if (APP_NETWORK_JOINED_STATE == appState) {
    contador1s++; // Incrementa o contador de tempo
    if (contador1s >= 30) { // Caso tenham se passado 60s
      appDeviceState = DEVICE_REQUEST_DATA_FROM_COORDINATOR_STATE;
      SYS_PostTask(APL_TASK_ID);
    }
  }
}

/*****************************************************************************
```

```
  Description: Monta os vetores para serem enviados pela usart
  Parameters:  nenhum.
  Returns:     nenhum.
*****************************************************************************/
static void sendValuesToUsart(void) {
  DeviceTypes_t sensor1=SENSOR_1;     // Necessário para passar por referência
  DeviceTypes_t sensor2=SENSOR_2;     // Necessário para passar por referência
  DeviceTypes_t atuador1=ACTUATOR_1;  // Necessário para passar por referência
  DeviceTypes_t atuador2=ACTUATOR_2;  // Necessário para passar por referência
  int8_t zero8=0;                  // Necessário para passar por referência
  int16_t zero16=0;                  // Necessário para passar por referência

        // Habilita flag para o MATLAB indicando que se trata de uma mensagem de dados
  msgDados = 1;

  if (conectedPanA == true)
    panA = 1;          // Esta conectado na PAN A
  else
    panA = 0;          // Esta conectado na PAN B

  //Disabilita todas as interrupções
        HAL_DisableInterrupts();
  if (conectedPanA == true) { // A rede A possui 4 dispositivos diferentes
    sendDataSavedToUsart(&msgDados,&panA,&sensor1,&temperaturaSensor1,&zero16,&zero8);

    sendDataSavedToUsart(&msgDados,&panA,&sensor2,&temperaturaSensor2,&zero16,&zero8);

    sendDataSavedToUsart(&msgDados,&panA,&atuador1,&zero16,&zero16,&estadoPortaAtuador1);

    sendDataSavedToUsart(&msgDados,&panA,&atuador2,&zero16,&zero16,&estadoPortaAtuador2);
  }
  else { // A rede B possui apenas 2 dispositivos diferentes
    sendDataSavedToUsart(&msgDados,&panA,&sensor1,&temperaturaSensor1,&zero16,&zero8);

    sendDataSavedToUsart(&msgDados,&panA,&atuador1,&zero16,&zero16,&estadoPortaAtuador1);
  }

  //Habilita todas as interrupções
        HAL_EnableInterrupts();
        //Garante que o led termina apagado para não gerar nenhum tipo de dúvida
        appOffLed(APP_SENDING_STATUS_LED);
  // Desabilita flag para o MATLAB indicando que as mensagens são apenas para manutenção da
comunicação
  msgDados = 0;
}

/*****************************************************************************
  Description: Parâmetros de envio dos dados salvos para o MATLAB
  Parameters:  dados da rede.
  Returns:     nenhum.
*****************************************************************************/
inline void sendDataSavedToUsart(int8_t *msgDados, int8_t *panA, DeviceTypes_t *dispositivo,
                        int16_t *temperaturaMedida, int16_t *nivelBateria,
                        int8_t *estadoPorta) {
        //Variável para a mensagem da USART
        uint8_t str[100];
        uint8_t length;

        length = sprintf((char *) str, "%d\n%d\n%d\n%d\n%d\n%d\r\n", (int)*msgDados, (int)*panA,
                    (int)*dispositivo, (int)*temperaturaMedida, (int)*nivelBateria, (int)*estadoPorta);
        //Função para enviar os dados para a USART
        sendDataToUsart(str, length);
```

```
        //Função para inverter o status do LED da USART (amarelo)
        appToggleLed(APP_SENDING_STATUS_LED);
}

/**************************************************************************
  Description: Função para enviar os dados para a USART
  Parameters:  data - ponteiro para o endereço da mensagem
               length - comprimento da mensagem
  Returns:     nenhum.
**************************************************************************/
void sendDataToUsart(uint8_t* data, uint8_t length) {
  if (APP_TMP_DATA_BUFFER_SIZE > tmpDataBufferActualLength + length) {
    memcpy(&tmpDataBuffer[tmpDataBufferActualLength], data, length);
    tmpDataBufferActualLength += length;
  }
  if (false == usartTxBusyFlag)
    usartWriteConf();
}

/**************************************************************************
  Description: Função para leitura de dados recebidos pela USART
  Parameters: are not used.
  Returns: nothing.
**************************************************************************/
void appReadByteEvent(uint16_t readBytesLen) {
        uint16_t leDadosusart = 1;
        uint8_t temp;

        appOnLed(APP_SENDING_STATUS_LED);
        readBytesLen = readBytesLen;  // Warning prevention
        //Função do BitCloud para leitura do Byte na USART
        HAL_ReadUsart(&appUsartDescriptor,&read_msg,leDadosusart);
        //Flag criada para atribuição das variáveis de controle
        readBytesCount++;

        // Os dados são enviados na forma "%2d%2d%1d%1d" pelo MATLAB
        switch (readBytesCount)
  {
        case 1: // 1°parte do valor recebido
        {
                temp = (uint8_t)atoi(read_msg);
                tempRefUsartPanA = 10*temp;
                  read_msg[0] = "";
                break;
        }
        case 2: // 2°parte do valor recebido
        {
                temp = (uint8_t)atoi(read_msg);
                tempRefUsartPanA = tempRefUsartPanA+temp;
                tempRefUsartPanA = tempRefUsartPanA*100;
                  read_msg[0] = "";
                break;
        }
        case 3: // 1°parte do valor recebido
        {
                temp = (uint8_t)atoi(read_msg);
                tempRefUsartPanB = 10*temp;
                  read_msg[0] = "";
                break;
        }
        case 4: // 2°parte do valor recebido
        {
                temp = (uint8_t)atoi(read_msg);
```

```
                    tempRefUsartPanB = tempRefUsartPanB+temp;
    tempRefUsartPanB = tempRefUsartPanB*100;
                     read_msg[0] = "";
                    break;
          }
          case 5: // 1°parte do valor recebido
          {
            temp = (uint8_t)atoi(read_msg);
            if (mudandoDeRede == false)
                     adiquirirDados = temp;
                     read_msg[0] = "";
                    break;
          }
          case 6: // 1°parte do valor recebido
          {
            temp = (uint8_t)atoi(read_msg);
            if (mudandoDeRede == false)
                     mudarPan = temp;
                     read_msg[0] = "";
                     readBytesCount=0;
                     appOffLed(APP_SENDING_STATUS_LED);
                    break;
          }
  }
}

/**************************************************************************************
  Description: Writing confirmation has been received. New message can be sent.
  Parameters: none.
  Returns: nothing.
**************************************************************************************/
static void usartWriteConf(void) {
  int8_t bytesWritten;

  if (0 < tmpDataBufferActualLength) { // data waiting to be written to USART
    memcpy(usartTxBuffer, tmpDataBuffer, tmpDataBufferActualLength);
    bytesWritten = HAL_WriteUsart(&appUsartDescriptor, usartTxBuffer, tmpDataBufferActualLength);
    if (0 < bytesWritten) {
      tmpDataBufferActualLength -= bytesWritten;
      usartTxBusyFlag = true;
    }
  }
  else
    usartTxBusyFlag = false;
}

/**************************************************************************************
  Description: Função para fazer a troca de rede (A para B ou B para A)
  Parameters: none.
  Returns: nothing.
**************************************************************************************/
static void appLeaveNetwork(void) {
  mudandoDeRede = true;      // Seta flag para não reiniciar o dispositivo
  appOffLed(APP_RECEIVING_STATUS_LED);
  appOffLed(APP_SENDING_STATUS_LED);
  appState = APP_CHANGING_NETWORK_STATE;
  appTaskHandler(APP_PROCESS,NULL);
}

#endif // _ENDDEVICE_
// eof router.c
```